Methodology for Efficient CNN Architectures in Profiling Attacks

Gabriel Zaid[1,2], Lilian Bossuet[1], Amaury Habrard[1] and Alexandre Venelli[2]

[1] Univ Lyon, UJM-Saint-Etienne, CNRS Laboratoire Hubert Curien UMR 5516 F-42023, Saint-Etienne, France, firstname.lastname@univ-st-etienne.fr

[2] Thales ITSEF, Toulouse, France, firstname.lastname@thalesgroup.com

Abstract. The side-channel community has recently investigated a new approach, based on deep learning, to significantly improve profiled attacks against embedded systems. Previous works have shown the benefit of using Convolutional Neural Networks (CNN) to limit the effect of some countermeasures such as desynchronization. In comparison to Template Attacks, deep learning techniques can deal with traces misalignment and the high dimensionality of the data. The pre-processing phases are no longer mandatory. However, the performance of attacks highly depend on the choice of each hyperparameters that compose a CNN architecture. Hence, we cannot perfectly harness the potential of deep neural networks without a clear comprehension of the networks inner-workings. In order to reduce this gap, we propose to clearly explain the role of each hyperparameters during the feature selection phase by using some specific visualization techniques such as *Weight Visualization*, *Gradient Visualization* and *Heatmap*. By highlighting which features are retained by filters, Heatmaps come in handy when a security evaluator tries to interpret and understand the efficiency of CNN. We propose a methodology for building efficient CNN architectures in terms of attack efficiency and network complexity, even in the presence of desynchronization. We evaluate our methodology on public datasets with and without desynchronization. In each case, we outperform the previous state-of-the-art CNN models while significantly reducing the network complexity. Our networks are up to 25 times more efficient than previous state-of-the-art while their complexity is up to 31810 times smaller. Our results show that CNN networks do not need to be too complex for getting good performance in the side-channel context.

Keywords: Side-Channel Attacks · Deep Learning · Architecture · Weight Visualization · Heatmap · Feature selection · Desynchronization

1 Introduction

Side-Channel Analysis (SCA) is a class of cryptographic attack in which an adversary tries to exploit vulnerabilities from a system by analyzing its physical properties, such as power consumption [KJJ99] or electromagnetic emanations [AARR03], to reveal secret information. During its execution, a cryptographic implementation manipulates sensitive variables which directly depend on the secret. Through its attack, an adversary tries to recover these information by finding some leakage related to the secret. One of the most powerful type of SCA attacks are *profiled attacks*. In this scenario, an adversary has access to a test device from which he knows the intermediate values of the target variable. Then, he is able to estimate the conditional distribution associated to each sensitive variable. On a target device containing a secret to retrieve, he can predict the proper sensitive value using multiple traces and reveal the secret. In 2002, the first profiled attack was introduced by [CRR03], but their proposal was limited by the computational complexity. Very similar

to profiled attacks, the application of machine learning algorithms was inevitably explored in the side-channel context [HGM+11, BL12, HZ12, LBM14, LPMS18].

Recently, some papers have shown the robustness of Convolutional Neural Networks (CNNs) to the most common countermeasures, namely *masking* [MPP16, MDP19a] and *desynchronization* [CDP17]. They are at least as efficient as classical profiled attacks. One of the main advantage is that they do not require pre-processing. A CNN consists of two parts namely a *convolutional part*, which goals is to locate which features impact the most the classification, and a *fully-connected part*, that aggregates the most relevant features, in order to correctly classify each trace. Nonetheless, finding a suitable architecture is one of the most challenging problems in deep learning because we have to properly set parameters that compose the network to obtain a good efficiency. Two types of parameters exist: *trainable parameters* and *hyperparameters*. Trainable parameters are non-parametric variables that are internal to the model and are automatically estimated during the training process. Hyperparameters are parametric variables that we need to be set before applying a learning algorithm. The hyperparameters can be decomposed into two categories:

- Optimizer hyperparameters that are related to the optimization and training process (learning rate, batch size, number of epochs, optimizer, activation functions, weight initialization, ...);
- Model hyperparameters, that are involved in the structure of the model (number of hidden units, number of layers, length of filters, number of convolutional blocks, ...).

Depending on the choice of these values, the performance of the network differs a lot. Both types of hyperparameters affect each others, they both need to be correctly selected. Choosing correct model hyperparameters is the first step towards obtaining an optimal neural network. Our work focuses on building efficient and suitable architectures. Our results would of course benefit from also selecting suitable optimizer hyperparameters that have been investigated in previous literature [CDP17, KPH+19].

Contributions. In this paper, we focus on the explainability and interpretability of model hyperparameters that compose the convolutional part of a CNN, which is the part that aims at revealing the most relevant features. In side-channel attacks, the information that helps the decision-making is defined by the PoIs which are revealed by leakage detection tools such as *Signal-to-Noise Ratio* (SNR) [MOP07]. Then, by accurately recovering these points, we can significantly reduce the complexity of the classification part because the aggregation phase will be much easier. To evaluate the impact of each model hyperparameter, we apply visualization techniques, such as *Gradient Visualization* [MDP19b] in order to get a global evaluation of how the network selects its features to predict the sensitive variable. However, we observe that this technique is not sufficient when we want to interpret only the convolutional part of the network. Hence, we propose to apply two new visualization techniques widely used in the deep learning field which are *Weight Visualization* and *Heatmap*. These tools help us to evaluate the impact of each model hyperparameters such as the number of filters, the length of each filter and the number of convolutional blocks in order to find a suitable architecture, by minimizing its complexity while increasing its attack efficiency. Then, we propose a methodology for building efficient CNN architectures. We confirm the relevance of our methodology by obtaining state-of-the-art results on the main public datasets. Through this new methodology, we are also able to find a better tradeoff between the complexity of the model and its performance allowing us to improve the learning time.

Paper Organization. The paper is organized as follows. Section 2 is dedicated to the presentation of the convolutional neural network and particularly the convolutional part. After the introduction of classical feature selection techniques in Section 3, we present two

new visualization tools applied to the SCA context in Subsection 3.2 and Subsection 3.3. In Section 4, we highlight the impact of each model hyperparameters on the convolutional part. Then, we propose a methodology for building a efficient architecture depending on the countermeasures implemented. The new methodology is applied on public datasets to obtain state-of-the-art results in Section 5.

2 Preliminaries

2.1 Notation and terminology

Let calligraphic letters $\mathcal{X}$ denote sets, the corresponding capital letters X (resp. bold capital letters) denote random variables (resp. random vectors $\mathbf{T}$) and the lowercase x (resp. $\mathbf{t}$) denote their realizations. The i-th entry of a vector $\mathbf{t}$ is defined as $\mathbf{t}[i]$. Side-channel traces will be constructed as a random vector $\mathbf{T} \in R^{1\times D}$ where D defines the dimension of each trace. The targeted sensitive variable is $Z = f(P, K)$ where f denotes a cryptographic primitive, P ($\in \mathcal{P}$) denotes a public variable (e.g. plaintext or ciphertext) and K ($\in \mathcal{K}$) denotes a part of the key (e.g. byte) that an adversary tries to retrieve. Z takes values in $\mathcal{Z} = \{s_1, ..., s_{|\mathcal{Z}|}\}$. Let us denotes k^* the secret key used by the cryptographic algorithm.

2.2 Profiled Side-Channel Attacks

When attacking a device using a profiled attack, two stages have to be considered: a building phase and a matching phase. During the first phase, an adversary has access to a test device on which he can control the input and the secret key of the cryptographic algorithm. He uses this knowledge to find the relevant leakages depending on Z. To characterize the points of interest (PoIs), the adversary generates a model $F : \mathbb{R}^D \to \mathbb{R}^{|\mathcal{Z}|}$ that estimates the probability $Pr[\mathbf{T}|Z = z]$ from a profiled set $\mathcal{T} = \{(\mathbf{t}_0, z_0), \ldots, (\mathbf{t}_{N_p-1}, z_{N_p-1})\}$ of size N_p. The high dimensionality of $\mathbf{T}$ can be a problem for building a leakage model. Popular techniques use dimensionality reduction, such as Principal Components Analysis [APSQ06] or Kernel Discriminant Analysis [CDP16], to select PoIs where most of the secret information is contained. Once the leakage model is generated, the adversary estimates which intermediate value is processed thanks to a predicting function $F(\cdot)$ designed by the adversary. By predicting this sensitive variable and knowing the input used during the encryption, the adversary can compute a score vector, based on $F(\mathbf{t}_i), i \in [0, |N_a| - 1]$, for each trace included in a dataset of N_a attack traces. Then, log-likelihood scores are computed to make predictions for each key hypothesis. The candidate with the highest value will be defined as the recovered key.

Metric in Side-Channel Attacks. During the building phase, the adversary wants to find a model F that optimizes the secret key recovery by minimizing the size of N_a. In SCA, a common metric used is called guessing entropy (GE) [SMY09] that defines the average rank of k^* through all key hypotheses. Let us denote $g(k^*)$ the guessing value, associated with the secret key. We considered an attack successful when the guessing entropy is permanently equal to 1. The rank of the correct key gives us an insight into how well our model performs.

2.3 Neural Networks

Profiled SCA can be formulated as a *classification problem*. Given an input a neural network aims at constructing a function $F : R^n \to R^{|\mathcal{Z}|}$ that computes an output called *a prediction*. To solve a classification problem, the function F has to find the right prediction $y \in \mathcal{Z}$ associated with the input $\mathbf{t}$ with high confidence. To find the optimized solution, a

neural network has to be trained given a profiled set of N pairs $(\mathbf{t}_p^i, y_p^i)$ where $\mathbf{t}_p^i$ is the i-th profiled input and y_p^i is the label associated with the i-th input. In SCA, the input of a neural network is a side-channel measurement and the related label is defined by the corresponding sensitive value z. To construct a function F, a neural network is composed of several simple functions called *layers*. Three kinds of layers exist: input layer (composed of input $\mathbf{t}$), output layer (gives an estimation of the prediction vector y) and one or multiple hidden layers. First, the input goes through the network to estimate the corresponding score $\hat{y}_p^i = F(\mathbf{t}_p^i)$. A *loss function* is computed that quantifies the classification error of F over the profiled set. This stage is called *forward propagation*. Each trainable parameters is updated in order to minimize the loss function. This is the *backward propagation* [GBC16]. To accurately and efficiently find the loss extremum, an optimizer is used (e.g. Stochastic Gradient Descent [RM51, KW52, BCN18], RMSprop, Momentum [Qia99], Adam [KB15], ...). These steps are processed until the network reachs a sufficient local minimum. The choice of hyperparameters is one of the most challenging problems in the machine learning field. To find them, some techniques exist such as Grid-Search Optimization, Random-Search Optimization [BB12, PGZ+18, BBBK11] but none of them are deterministic. This paper focuses on understanding model hyperparameters in order to build efficient neural network architectures for side-channel attacks.

2.4 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is specific type of neural network. In this paper, we choose to simplify its presentation by considering that it can be decomposed into two parts: a feature extraction part and a classification part (see Figure 1-a). Features selection aims at extracting information from the input to help the decision-making. To select features, a CNN is composed of stacked convolutional blocks that correspond to n_2 convolutional layers (denoted γ), an activation function (σ) and one pooling (denoted δ) layer [ON15]. This feature recognition part is plugged into the classification part of n_1 Fully-Connected (FC) layers (denoted λ). Finally, we denote s the softmax layer (or prediction layer) composed of $|Z|$ classes. To sum up, a common convolutional network can be characterized by the following formula:

$$s \circ [\lambda]^{n_1} \circ [\delta \circ [\sigma \circ \gamma]^{n_2}]^{n_3}. \tag{1}$$

2.4.1 Convolutional layer

The convolutional layer performs a series of convolutional operations on its inputs to help the patterns recognition (see Figure 1-b). During the forward propagation, each input is convoluted with a filter (or *kernel*). The output of the convolution shows temporal instants that influence the classification. These samples are called **features**. To build a convolutional layer, some model hyperparameters have to be configured: length of kernels, number of kernels, pooling stride and padding.

- Length of filters – Kernels are generated to identify features that could increase the efficiency of the classification. However, depending on their size, filters reveal local or global features. Smaller filters tend to identify local features while larger filters focus on global features. In Figure 1-b, an example is given where the length of filters is set to 3.
- Stride – It indicates the step between two consecutive convolutional operations. Using small stride corresponds to the generation of an overlapping between different filters while a greater stride reduces the output dimension. By default, the stride is set to 1 (see Figure 1-b).
- Padding – Let $\mathbf{a}$ and $\mathbf{b}$ be two vectors, the dimension of the convolution between these two vectors will be $dim(\mathbf{a} \circledast \mathbf{b}) = \left(\frac{dim(\mathbf{a}) - dim(\mathbf{b})}{stride} + 1\right)$ [GBC16] where $\circledast$ refers

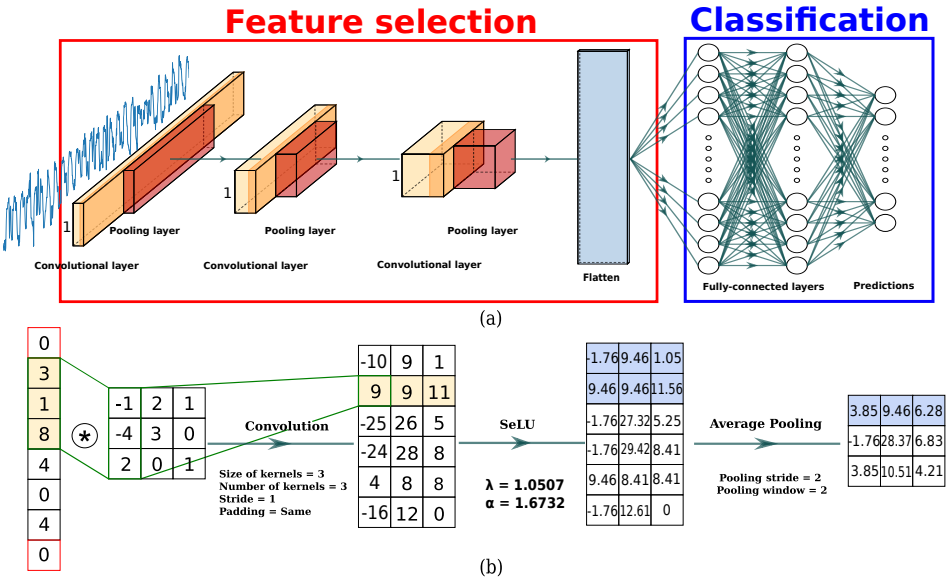

Figure 1: (a) - CNN architecture for side-channel attacks (red area: feature selection part ; blue area: classification part). (b) - Operations in convolutional blocks (convolutions, activation, average pooling)

to the convolution operation. In some cases, a subsampling can be generated. To avoid this phenomenon and loose information, we can use padding that adds a "border" to our input to keep the same dimension after the convolutional operation. By default, two kinds of padding are used: *valid padding* and *same padding*. *Valid padding* means "no-padding" while *same padding* refers to a zero-padding (the output has the same dimension as the input). In Figure 1-b, an example is given where we select a *same padding*. Indeed, two 0 values are added at the endpoints of the vector in order to get an output vector of dimension 6.

After each convolutional operation, an *activation function* (denoted σ) is applied to identify which features are relevant for the classification. As explained in [KUMH17], the *scaled exponential linear unit function* (SeLU) is recommended for its self-normalizing properties. The SeLU is defined as follows:

$$selu(x) = \lambda \begin{cases} x & \text{if} \quad x > 0, \\ \alpha(exp(x) - 1) & \text{if} \quad x \leqslant 0. \end{cases} \tag{2}$$

It pushes neuron activation towards zero mean and unit variance in order to prevent the vanishing and exploding gradient problems. This activation function is used by default in our architectures.

2.4.2 Pooling layer

Pooling layer is a non-linear layer that divides the dimension of the input such that the most relevant information is preserved. To apply its down sampling function, a pooling window and a pooling stride have to be configured. Usually, these variables have the same value to avoid overlapping. The window slides through the input values to select a segment to be applied to the pooling function. In deep learning, two pooling functions are commonly used:

- MaxPooling – The output is defined as the maximum value contained in the pooling window.

- AveragePooling – The output is defined as the average of the values contained in the pooling window. In Figure 1-b, we show an example of this function applied to a 1-D input with pooling_window = pooling_stride = 2.

2.4.3 Flatten layer

The flatten layer concatenates each intermediate trace of the final convolutional block in order to reduce the 2-D space, corresponding to the dimension at the end of the convolutional part, into a 1-D space to input into the classification part. Let us denote M the input of the flatten layer such that $M \in \mathbb{M}_{n,d}(\mathbb{R})$ where n denotes the number of output after the last convolutional block and d denotes the number of samples for each these outputs such that:

$$M = \begin{bmatrix} \mathbf{t}^0[x_0] & \mathbf{t}^0[x_1] & \mathbf{t}^0[x_2] & \cdots & \mathbf{t}^0[x_{d-1}] \\ \mathbf{t}^1[x_0] & \ddots & \cdots & \cdots & \vdots \\ \mathbf{t}^2[x_0] & \cdots & \ddots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{t}^n[x_0] & \cdots & \cdots & \cdots & \mathbf{t}^{n-1}[x_{d-1}] \end{bmatrix} \tag{3}$$

where $(\mathbf{t}^i)_{0 \leqslant i \leqslant n}$ is the i-th intermediate trace and $(x_j)_{0 \leqslant j \leqslant d}$ the j-th sample of the trace.

The output of the flatten layer is a concatenated vector C that can be constructed:

- Column-wise – $C = \left[\mathbf{t}^0[x_0], \mathbf{t}^1[x_0], \mathbf{t}^2[x_0], ..., \mathbf{t}^{n-1}[x_{d-1}]\right]$,
- Row-wise – $C = \left[\mathbf{t}^0[x_0], \mathbf{t}^0[x_1], \mathbf{t}^0[x_2], ..., \mathbf{t}^{n-1}[x_{d-1}]\right]$.

2.4.4 Fully-Connected Layers

Once the convolutional part has selected features, the Fully-Connected (FC) (denoted λ) layers recombine each neuron to efficiently and accurately classify each input. FC layers can be compared to a MultiLayer Perceptron (MLP) where each neuron has full connections to all activations of the previous layer (see Figure 1-a).

In this paper, we propose a way to select appropriate model hyperparameters in order to find a suitable network architecture.

3 Evaluation of features selection

Due to the black-box nature of neural networks, it can be challenging to explain and interpret its decision-making. In order to overcome this problem, some visualization techniques were developed [vdMH08, ZF14, SVZ14, ZKL+16] but their application in the SCA context is not exploited enough. As shown in the following papers [MDP19b, HGG19, PEC19], these techniques could be useful in SCA to evaluate the ability of a network to extract the points of interest.

3.1 State of the art

Signal-to-Noise Ratio. Signal-to-Noise Ratio (SNR) [Man04, MOP07] is one of the most useful tools in SCA to characterize temporal instants where sensitive information leaks.

Unfortunately, in presence of countermeasures such as desynchronization, estimation of PoIs becomes much more complex. Indeed, desynchronization disturbs the detection of PoIs by randomly spreading the sensitive variable over time samples.

Visualization techniques. Gradient Visualization computes the derivatives of a CNN model with respect to an input trace such that the magnitude of the derivatives indicates which features need to be modified the least to affect the class score the most. This technique is helpful for identifying the temporal instants that influence the classification. Comparing these points to PoIs, we are able to interpret the learning phase of a network [MDP19b].

Layer-wise Relevance Propagation is introduced in [BBM+15] and applied in [HGG19] in the side-channel context. LRP propagates the prediction score through the network until the input layer by indicating the relevant layers. In [HGG19], the authors use this technique for the purpose of underlining which temporal instants influence the most the learning process.

Limitations. These proposals aim at interpreting a CNN decision by finding each temporal instant of the trace which contributes the most towards a particular classification. However these techniques are not useful in order to understand how the convolutional or the classification part selects its feature. Indeed, they only give a global interpretation of how a network perform. However, in the case where the network is not able to find the right PoIs, we cannot evaluate the faulty part (convolutional or classification). A more precise technique should be used to interpret each part independently. To reduce this issue, we propose to apply *Weight Visualization* and *Heatmaps* to provide information related to the convolutional part.

3.2 Weight visualization

One way to interpret the performance of a network is to visualize how it selects its features. As explained in introduction, one common strategy is to visualize the trainable weights to interpret the recognition patterns. This method was introduced in [BPK92] as a framework useful when dealing with spatial information. During the training process, the network evaluates the influencing neurons which generate an efficient classification. If the network is confident in its predictions, it will associate large weights to these neurons. In deep learning, this technique is usually used for evaluating features extracted by the first layers of a deep architecture [HOWT06, LBLL09, OH08, HOT06]. The role of the convolutional part is to select the relevant time samples (i.e. PoIs) that compose a trace for allowing an efficient classification. Therefore, by applying the weight visualization, we propose a new tool that estimate the performance related to the convolutional part.

As we defined in Subsection 2.4, the flatten operation sub-samples each intermediate trace following an axis. By concatenating the output of the flatten layer following the columns (see Equation 3), we keep timing information to be able to reveal leakages and interpret it (see Figure 2-a). If the feature selection is effective, the neurons where information leaks will be evaluated with high weights by the training process. Then, by visualizing the weight of the flatten layer, we can understand which neurons have a positive impact on the classification and thus, thanks to the *feedforward propagation*, we can interpret which time samples influence the most our model.

Let us denote $n_u^{[\text{flatten}+1]}$ the number of neurons in the layer following the flatten, $n_f^{[\text{flatten}-1]}$ the number of filters in the last convolutional blocks and $dim_{\text{traces}}^{[\text{flatten}-1]}$ the dimension associated with each intermediate trace after the last pooling layer. Let us denote $W^{[\text{flatten}]}$ the weights corresponding to the flatten layer such that $dim(W^{[\text{flatten}]}) = (dim_{\text{traces}}^{[\text{flatten}-1]} \times n_f^{[\text{flatten}-1]})$. Let us denote $W_m^{\text{vis}} \in \mathbb{R}^{dim_{\text{traces}}^{[\text{flatten}-1]}}$ a vector that allows visualizing the weights related to the m-th neurons of the layer following the flatten:

$$W_m^{\text{vis}}[i] = \frac{1}{n_f^{[\text{flatten}-1]}} \sum_{j=i\times n_f^{[\text{flatten}-1]}}^{(i+1)\times n_f^{[\text{flatten}-1]}} |W_m^{[\text{flatten}]}[j]| \tag{4}$$

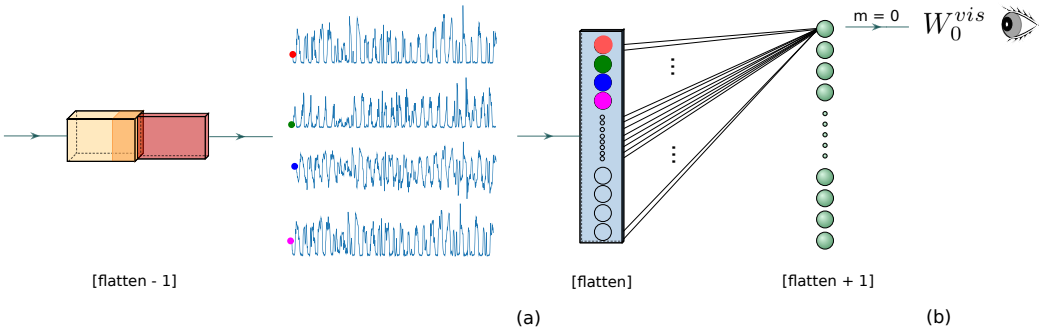

Figure 2: Methodology for weight visualization ((a) - concatenation of each intermediate trace following their temporal axis ; (b) - computes the average of W_0^{vis} to get a temporal correspondence)

where $i \in [0, dim_{\text{traces}}^{[\text{flatten}-1]}]$.

Then, let us denote $W^{\text{vis}} \in \mathbb{R}^{dim_{\text{traces}}^{[\text{flatten}-1]}}$ a vector that allows visualizing the mean weight related to each neuron of the layer [flatten +1] such that:

$$W^{\text{vis}}[i] = \frac{1}{n_u^{[\text{flatten}+1]}} \sum_{m=0}^{n_u^{[\text{flatten}+1]}} W_m^{\text{vis}}[i] \tag{5}$$

where $i \in [0, dim_{\text{traces}}^{[\text{flatten}-1]}]$.

In other words, first, we reduce the dimension of $W^{[\text{flatten}]}$ so that its dimension corresponds with the temporal space defined by $dim_{\text{traces}}^{[\text{flatten}-1]}$. This reduction is made by averaging the weights, associated to the same point of intermediate traces (see Figure 2-b). Finally, in order to obtain a more precise evaluation, we have to compute the average of mean weight associated to each neuron in order to evaluate the confidence of the network for revealing PoIs. An example of weight visualization is given in Appendix B Figure 7.

3.3 Heatmap

Even if the weight visualization is a useful tool, we are not able to understand which feature is selected by each filter. One solution is to analyze the activation generated by the convolution operation between an input layer and each filter. Introduced in [ZF14], Heatmaps (or feature maps) help to understand and interpret the role of each filter for generating suitable CNN architectures. By applying this technique in the side-channel context, we propose a new tool for interpreting the impact of filters and thus we can adapt the network according to their features selection.

Let us denote $n_f^{[h]}$ the number of filters for the h-th layer of the convolutional part, $\text{input}^{[h]}$ the input(s) associated with the h-th convolutional block such that $\text{input}^{[0]}$ corresponds to the side-channel trace and $\text{output}^{[h-1]} = \text{input}^{[h]}$. Then, the convolution between $\text{input}^{[h]}$ and the $n_f^{[h]}$ filters returns $n_f^{[h]}$ intermediate traces. We denote $H^{[h]}$ the heatmap (or feature map) associated to the h-th layer such that:

$$H^{[h]} = \frac{1}{n_f^{[h]}} \sum_{i=0}^{n_f^{[h]}} (\text{input}^{[h]} \circledast n_f^{[h]})[i] \tag{6}$$

Then, we can evaluate which neurons are activated by the filters of each convolutional layer and understand how the features are selected (see Figure 3).

The weight visualization and the heatmaps are two tools that helps to explain and interpret more clearly the selection of features. Using these techniques to understand the

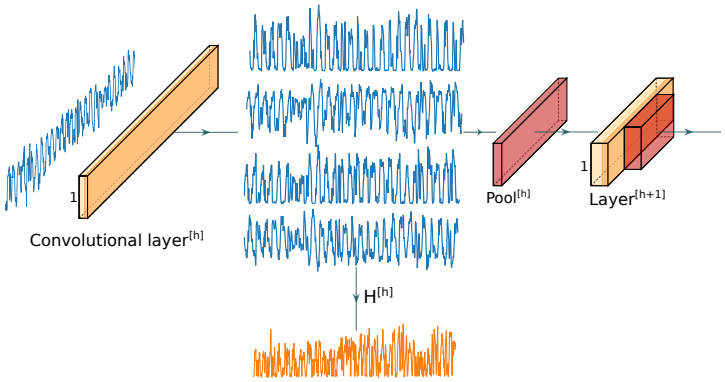

Figure 3: Heatmap

internal layers of a CNN, a methodology to build suitable neural network architectures can be defined in the presence of desynchronization.

4 Methodology for efficient CNN architectures

In this section, we analyse the effect of each model hyperparameters in the convolutional part of a CNN and propose a methodology for building efficient architectures. Using the presented visualization techniques, we can understand how each hyperparameter impacts the efficiency of the CNN and how it should be tweaked in the presence of desynchronization.

4.1 Dataset

To illustrate and explain as much as possible the effect of each hyperparameters, we consider an unmasked implementation of AES-128 on the Chipwhisperer[a] (8-bit XMEGA Target). Our experience is done with 45000 power traces of 3000 points. The targeted operation is $Z = Sbox[P_1 \oplus k^*]$ where P_1 denotes the first byte of the plaintext and k^* denotes the secret key. The measured SNR equals 2.28 (see Appendix A).
The CNN implementation is done in Python using the *keras* library [C+15] and is run on a workstation equipped with 8GB RAM and a NVIDIA GTX1060 with 8GB memory. We use the *Categorical Cross-Entropy* as loss function because minimizing the cross entropy is equivalent to maximizing the lower bound of the mutual information between the traces and the secret variable [MDP19a]. The optimization is done by the *Adam* [GBC16] approach on *batch size* 50 and the *learning rate* is set to 10^{-3}. As explained in Subsection 2.4, we use the *SeLU* activation function to avoid the vanishing and exploding gradient problems. For a better weight initialization, we use the *He Uniform* initialization [HZRS15]. This choice is motivated by the fact that it provides good results in terms of classification. We use 40000 traces for the training process, 5000 traces for the validation and 20 epochs are used. Finally, in order to accelerate the learning phase, we pre-process the data such as all trace samples are standardized and normalized between 0 and 1 [GBC16]. In this section, we only focus on the pattern selection made by the convolutional part and not on the attack exploitation.

4.2 Length of filters

In this section, we demonstrate that increasing the filter length causes entanglement and reduces the weight related to a single information and therefore, the network confidence.

[a] https://newae.com/tools/chipwhisperer/

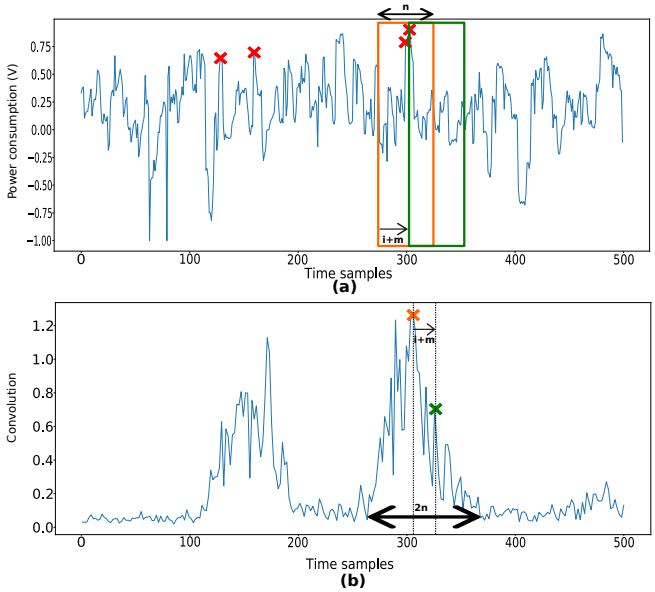

Figure 4: Convolution between a trace $\mathbf{t}$ and filter W of size n ((a) - trace; (b) - convolution operation between $\mathbf{t}$ and W)

To this end, we need an assumption such that there exist only a small dataset $\mathbb{E} = \{x_0, x_1, ..., x_l\}$ of time samples such that $Pr[Z|\mathbf{T}] = Pr\left[Z|\mathbf{T}[x_0], ..., \mathbf{T}[x_l]\right]$ where $l << dim(\mathbf{T})$. Let us denote $W \in \mathbb{R}^n$ a filter of length n. The convolution operation between a trace $\mathbf{t}$ and a filter W following the equation:

$$(\mathbf{t} \circledast W)[i] = \sum_{j=0}^{n} \left(\mathbf{t}\left[j + i - \frac{n}{2} \right] \times W[j] \right) \tag{7}$$

such that

$$\mathbf{t}\left[j + i - \frac{n}{2} \right] \times W[j] = \begin{cases} \alpha_j \times \mathbf{t}\left[j + i - \frac{n}{2} \right] & \text{if} \quad j \in \mathbb{E}, \\ \epsilon & \text{otherwise.} \end{cases} \tag{8}$$

where $\epsilon \approx 0$ and α_j denotes the weight related to the index j.
Let assume one filter W of size n such that $W = \{\alpha_0, \alpha_1, ..., \alpha_n\}$ is optimized for maximizing the detection of l PoIs. Denote $\mathbf{t}$ a trace such that $(\mathbf{t} \circledast W)[i]$ covers the l relevant points and $(\mathbf{t} \circledast W)[i+m]$ covers less than l relevant points. Because W maximize the detection of the l PoIs, then,

$$(\mathbf{t} \circledast W)[i] > (\mathbf{t} \circledast W)[i+m] \tag{9}$$

Figure 4 gives an example of this operation. Four relevant information can be extracted from the input trace (see red crosses). In this figure, we show the result of two convolution operations $(\mathbf{t} \circledast W)[i]$ (see blue cross) and $(\mathbf{t} \circledast W)[i+m]$ (see green cross) that share the same information related to the highest relevant leakage. Consequently, when entanglement occurs the convoluted samples share the same relevant information. Therefore, in our example, the highest PoI is spread over the two convoluted samples. By increasing the size n of the filters, we increase the entanglement, consequently, more PoIs are shared between these convoluted points. Thus, the relevant information is spread over $2n$ convoluted samples. This phenomenon can be observed in the Appendix B Figure 7. If we want to precisely define the temporal space where leakages occur, we recommend to minimize the length of the filters in order to reduce the risk of entanglement.
Let two filters $W \in \mathbb{R}^n$ and $W^* \in \mathbb{R}^m$ such that $n > m$. Assume that these filters share a same PoI denoted x_l. In other words, the PoI is respectively shared by n samples (when

W is applied) and m samples (when W^* is applied). Let α_l^{opt} be the weight related to the PoI when x_l is assigned to a single convoluted sample. Then,

$$(\mathbf{t} \circledast W)[i] = (n-1) \times \epsilon + \frac{\alpha_l^{opt}}{n} \times \mathbf{t}[x_l] \tag{10}$$

and,

$$(\mathbf{t} \circledast W^*)[i] = (m-1) \times \epsilon + \frac{\alpha_l^{opt}}{m} \times \mathbf{t}[x_l]. \tag{11}$$

Therefore, the weight related to $(\mathbf{t} \circledast W^*)[i]$ is higher than $\frac{\alpha_l^{opt}}{n}$. As conclusion, increasing the length of the filters also reduces the weight related to a single information because entanglement occurs.

We propose to illustrate these properties through an example. We apply different architectures to our Chipwhisperer dataset in order to efficiently evaluate the impact of the filter's length. We select a range of lengths (see Appendix B) and visualize the weights to experimentally verify this claim. The smaller the length, the less the relevant information is spread. Appendix B Figure 7 shows that the weight visualization succeeds in recovering the leakage localization. Indeed, the PoIs correspond to the relevant information detected by the SNR (see Appendix A). The number of samples is divided by 2 because of the pooling stride of 2 that we use in the first convolutional block. As explained, small filters allow a better extraction of information because the PoIs are not shared with a lot of convoluted samples. While using larger filters generate entanglement and provide the detection of "global features".

By visualizing the heatmap associated to the length 1 (see Appendix B Figure 8), we note that the filters try to maximize the extremum of each trace in order to exploit the most of relevant information that is contained in the extremum of each trace. After the convolution, the network can extract PoIs much easier.

4.3 Number of convolutional blocks

In this section, we provide an interpretation of the number of convolutional blocks. We compare the pooling functions that can be denoted as $f^{[h]} \colon \mathbb{R}^{n^{[h]}} \to \mathbb{R}^{\frac{n^{[h]}}{\text{pooling\_stride}^{[h]}}}$ where h corresponds to the h^{th} convolutional block. We theoretically show that the role of these functions is to reduce the trace dimension while the most relevant features are preserved. Let $\mathbf{t}^{[h]}$ be the input of the h^{th} convolutional block and $W^{[h]} \in \mathbb{R}^n$ the filter related to the h^{th} convolutional block. Let $ps^{[h]}$ be the pooling stride related to the h^{th} convolutional block. Assume that the j^{th} convoluted sample $(\mathbf{t}^{[h]} \circledast W^{[h]})[j]$ covers $l^{[h]}$ relevant information from the h^{th} convolutional block. The resulted $l_{conv}^{[h]}$ relevant convoluted samples are covered by the i^{th} pooling sample $AvgPool(\mathbf{t}^{[h]} \circledast W^{[h]})[i]$. Finally, the $(i+m)^{th}$ pooling sample $AvgPool(\mathbf{t}^{[h]} \circledast W^{[h]})[i+m]$ covers $(l+m)_{conv}^{[h]}$ PoIs such that the first $l_{conv}^{[h]}$ leakages are shared with $AvgPool(\mathbf{t}^{[h]} \circledast W^{[h]})[i]$. Assume that the most relevant leakage, denoted $x_{l_{conv,opt}^{[h]}}$, is included in both situation. Then,

$$\begin{aligned}
AvgPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]}))[i] &= \sum_{j=i}^{i+ps^{[h]}} \frac{\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})[j]}{ps^{[h]}} \\
&= \frac{1}{ps^{[h]}} \sum_{j=i}^{i+ps^{[h]}} \sigma\left(\sum_{k=0}^{n} \mathbf{t}^{[h]}\left[k+j-\frac{n}{2}\right] \times W^{[h]}[k]\right) \\
&\approx \frac{1}{ps^{[h]}} \sum_{j=0}^{l_{conv}^{[h]}} \sigma\left(\sum_{\substack{k=0 \\ k \in \mathbb{E}}}^{l_j^{[h]}} \left(\alpha_k^{[h]} \times \mathbf{t}^{[h]}\left[k+j-\frac{n}{2}\right]\right)\right).
\end{aligned}$$

Because $k \in \mathbb{E}$, $\alpha_k^{[h]} \times \mathbf{t}^{[h]}[k]$ represent the most relevant features in the trace $\mathbf{t}$ then, $\alpha_k^{[h]} \times \mathbf{t}^{[h]}[k] > 0$. Following the SeLU function (see Equation 2), we can simplify our solution such that,

$$\frac{1}{ps^{[h]}} \sum_{j=0}^{l_{conv}^{[h]}} \sigma \left(\sum_{\substack{k=0 \\ k \in \mathbb{E}}}^{l_j^{[h]}} \left(\alpha_k^{[h]} \times \mathbf{t}^{[h]} \left[k + j - \frac{n}{2} \right] \right) \right) = \frac{\lambda}{ps^{[h]}} \sum_{j=0}^{l_{conv}^{[h]}} \sum_{\substack{k=0 \\ k \in \mathbb{E}}}^{l_j^{[h]}} \left(\alpha_k^{[h]} \times \mathbf{t}^{[h]} \left[k + j - \frac{n}{2} \right] \right).$$

Moreover,

$$\begin{aligned} AvgPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]}))[i+m] &= \sum_{j=i+m}^{i+m+ps^{[h]}} \frac{\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})[j]}{ps^{[h]}} \\ &= \frac{1}{ps^{[h]}} \sum_{j=i+m}^{i+m+ps^{[h]}} \sigma \left(\sum_{k=0}^{n} \mathbf{t}^{[h]} \left[k + j - \frac{n}{2} \right] \times W^{[h]}[k] \right) \\ &\approx \frac{\lambda}{ps^{[h]}} \sum_{j=0}^{(l+m)_{conv}^{[h]}} \sum_{\substack{k=0 \\ k \in \mathbb{E}}}^{l_j^{[h]}} \left(\alpha_k^{[h]} \times \mathbf{t}^{[h]} \left[k + j - \frac{n}{2} \right] \right). \end{aligned}$$

then,

$$\begin{aligned} &AvgPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]}))[i+1] - AvgPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]}))[i] = \\ &\qquad \frac{\lambda}{ps^{[h]}} \sum_{j=(l+1)_{conv}^{[h]}}^{(l+m)_{conv}^{[h]}} \sum_{\substack{k=0 \\ k \in \mathbb{E}}}^{l_j^{[h]}} (\alpha_k^{[h]} \times \mathbf{t}^{[h]} \left[k + j - \frac{n}{2} \right]). \end{aligned}$$

Therefore, using an average pooling has the advantage that the unshared information are preserved. However, when the pooling stride $ps^{[h]}$ is large, the relevant information is reduced. As consequence, using a lot of convolutional blocks impact the leakage detection because the information is divided by pooling stride after each convolutional block.
In the case where the MaxPooling is used, then,

$$\begin{aligned} MaxPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]}))[i] &= max(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})[j] \mid j \in \{i, ..., i + ps^{[h]}\}) \\ &= x_{l_{conv,opt}^{[h]}} \end{aligned}$$

and,

$$\begin{aligned} MaxPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]}))[i+m] &= max(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]})[j] \mid j \in \{i+m, ..., i+m+ps^{[h]}\}) \\ &= x_{l_{conv,opt}^{[h]}} \end{aligned}$$

then,

$$MaxPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]}))[i+m] - MaxPool^{[h]}(\sigma(\mathbf{t}^{[h]} \circledast W^{[h]}))[i] = 0.$$

When MaxPooling is used, if two consecutive pooling computation share the same optimal leakage $x_{l_{conv,opt}^{[h]}}$, then this information is spread over the pooling samples. As explained in Equation 10 and Equation 11, if the same relevant information is spread over lots of samples, then the related weight decreases. Furthermore, because only $x_{l_{conv,opt}^{[h]}}$ is selected, all other leakages are discarded. Thus, we can lose information that seem less important while it is essential for the detecting relevant points. For this reason, we recommand using **Average Pooling** as much as possible because no relevant points are discarded.
Furthermore, incrementing the number of convolutional blocks reduce the trace dimension while preserving relevant information associated with the leakages. Indeed, in most cases, $dim_{\mathbf{t}}^{[h]} = \frac{dim_{\mathbf{t}}^{[h-1]}}{pooling\_stride^{[h-1]}}$. Then, we reduce by a factor $pooling\_stride^{[h-1]}$ the distance between the relevant points. Hence, the impact of desynchronization is reduced by the same

factor. It will be much easier, for the network, to characterize the desynchronization. By adding more convolutional blocks, we can drastically reduce the impact of desynchronization by choosing a suitable $pooling\_stride^{[h]}$.
These properties show that the pooling functions reduce the trace dimension while the most relevant samples are preserved. To illustrate these properties, we apply an average pooling or a max pooling (see Appendix C) to the Chipwhisperer dataset. Surprisingly, the detection of PoIs is more or less the same. For the first layers, the network seems to be more confident in it features detection when average pooling is applied. While, for much deeper network, the max pooling seems to be more suitable. However, in both cases, we can see that, the deeper the network, the less confident it is in its feature detection. In the presence of desynchronization, a trade-off should be found in order to detect the desynchronization and preserve maximum information related to the relevant points.

4.4 Methodology

Let us assume a Device Under Test (DUT) in which L leakages are detected using classical leakage detection such as SNR. Let us denote the following variables:

- $N^{[h]}$: the maximum amplitude of desynchronization after the h-th convolutional block (such that $N^{[0]}$ is the desynchronization associated with the input traces and $N^{[h]} = \frac{N^{[h-1]}}{\text{pooling\_stride}^{[h-1]}}$),
- $\text{pooling\_stride}^{[h]}$: the pooling stride associated to the h-th convolutional block,
- $D^{[h]}$: the trace dimension after the h-th convolutional block (such that $D^{[0]}$ is the input dimension) and in most cases, $D^{[h]} = \frac{D^{[h-1]}}{\text{pooling\_stride}^{[h-1]}}$.

4.4.1 Synchronized traces

Following the observation made in Subsection 4.3, adding convolutional blocks reduce the distance between the features in order to facilitate the detection of desynchronization. When we want to build an architecture such that traces in the training, validation and test sets are synchronized (*i.e.* $N^{[0]} = 0$), we don't need to configure more than one convolutional block. Indeed, using more than one convolutional block has two main drawbacks. First, in the case where PoIs are close to each other temporally, adding convolutional blocks increases the risk of entanglement. As we shown in Subsection 4.2, the entanglement can generate a spreading of relevant information through the convoluted time samples. Secondly, because of the pooling (see Subsection 4.3), some information can be lost: the same most relevant feature can be spread over the samples (MaxPooling) or each relevant information can be reduced following the pooling stride value (Average Pooling). When no desynchronization occurs, we recommend to set the number of convolutional blocks to 1.
In [MPP16] and [PSB+18], authors show that MLP can be a good alternative when an adversary is looking to build a suitable architecture when traces are synchronized. However, MLPs are only composed of fully-connected layers. A CNN is viewed as a MLP where only each neuron of the layer l is linked with a set of neurons of the layer $l-1$ [Kle17]. In the side-channel context, only few samples are needed for the decision-making. Using a CNN with small filter size helps to focus its interest in local perturbation and drastically reduces the complexity of the networks. CNNs are recommended with the smallest length of filters possible (i.e. 1). Moreover, thanks to the validation phase, an evaluator can estimate which length is the best in its context. Finally, the number of filters depends on the "imbalanced" sampling representation. For a set of synchronized traces, the distribution is uniform then using few filters is recommended (*i.e.* $2, 4, 8$).

4.4.2 Desynchronization: Random delay effect

Let us denote $N^{[0]}$ the maximal amplitude of the random delay effect. We define $Nt_{GE}(model_{N^{[0]}})$ the number of traces that a model, trained on desynchronized traces, needs in order to converge towards a constant guessing entropy of 1 on test traces. Let us denote T_{test} this set of traces such that:

$$Nt_{GE}(model_{N^{[0]}}) := min\{t_{test} \mid \forall t \geq t_{test},\ g(k^*)(model_{N^{[0]}}(t)) = 1\}. \tag{12}$$

Adding desynchronization does not reduce information contained in a trace set. However, the secret retrieval is more challenging. Through this section, we want to find a methodology such that we optimize a $model_{N^{[0]}}$ where the associated $Nt_{GE}(model_{N^{[0]}})$ is as close as possible to $Nt_{GE}(model_0)$. In other words, we want to find $model_{N^{[0]}}$ such that:

$$\min_{model_N} (|Nt_{GE}(model_0) - Nt_{GE}(model_{N^{[0]}})|). \tag{13}$$

To build a suitable architecture, we propose to use a new methodology that helps the desynchronization detection and the reduction of trace samples in order to focus the network on the leakages (see Figure 5). We divide our convolutional part into three blocks (an example is given in Appendix E):

- As shown in Subsection 4.2, the first layer aims at minimizing the length of the filters in order to minime the entanglement between each PoIs and extract the relevant information. Reducing the length of the filters helps the network to maximize the extremum of a trace in order to easily extract secret information. Then, we suggest setting the filters' length, of the first convolutional block, to 1 in order to optimize the entanglement minimization. The first pooling layer is set to 2 in order to reduce the dimension of the trace and help the desynchronization detection.
- The second block tries to detect the value of the desynchronization. By applying filter length of size $\frac{N^{[0]}}{2}$, we focus the interest of the network on the detection of the desynchronization of each trace. The network gets a global evaluation of the leakages by concentrating its detection on the leakage desynchronization and not on the leakages themselves. Then, we set the $pooling\_layer^{[1]}$ to $\frac{N^{[0]}}{2}$ to maximize the reduction of the trace dimension while preserving information related to the desynchronization.
- The third block aims at reducing the dimensionality of each trace in order to focus the network on the relevant points and remove the irrelevant ones. In the case where our traces have L PoIs, then, only L time samples help the network to make a decision. By dividing our trace in L different parts, we force the network to focus its interest on this information. Indeed, each part contains the information related to a single spread leakage. This process reduces the dimensionality of each trace by L such that each point of the output of the convolutional block defines a leakage point (see Appendix E). Furthermore, applying this technique limits the desynchronization effect because we force the network to concentrate the initial desynchronized PoIs into a single point.

5 Experimental results

In the following section, we apply our methodology on different publicly available datasets. The methodology is applied on both unprotected and protected implementations. We compare the performance of our architectures[b] with state-of-the-art results. For a good

[b] https://github.com/gabzai/Methodology-for-efficient-CNN-architectures-in-SCA

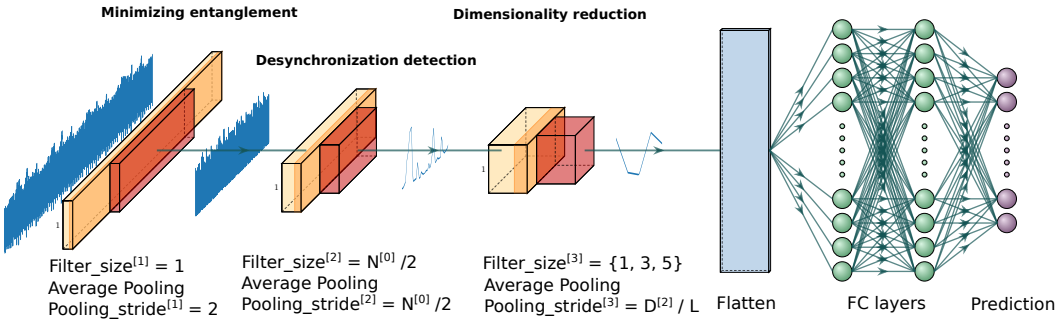

Figure 5: Architecture for desynchronized traces

estimation of Nt_{GE}, the attack traces are randomly shuffled and 100 Nt_{GE} are computed to give the average value for Nt_{GE}. We denote $\bar{N}t_{GE}$ this average over the 100 tests. Because the attack efficiency is not the only metric, we also compare the complexity of our architectures with the state-of-the-art.

5.1 Datasets

We use four different datasets for our experiments. All datasets correspond to implementations of *Advanced Encryption Standard* (AES) [DR02]. The datasets offer a large variety of use-cases: high-SNR unprotected implementation on a smart card, low-SNR unprotected implementation on a FPGA, low-SNR protected with a random delay desynchronization, low-SNR protected implementation with first-order masking [SPQ05] and desynchronization with a random delay.

- **DPA contest v-4** is an AES software implementation with a first-order masking [BBD+14][c]. Knowing the mask value, we can consider this implementation as unprotected and recover directly the secret. In this experiment, we attack the first round S-box operation. We identify each trace with the sensitive variable $Y^{(i)}(k^*) = Sbox[P_0^{(i)} \oplus k^*] \oplus M$ where M denotes the known mask and $P_0^{(i)}$ the first byte of the i-th plaintext. The measured SNR equals 4.33 (see Appendix D Figure 11).

- **AES_HD** is an unprotected AES-128 implemented on FPGA. Introduced in [PHJ+19][d], the authors decided to attack the register writing in the last round such that the labellisation of the i-th trace is $Y^{(i)}(k^*) = Sbox^{-1}[C_j^{(i)} \oplus k^*] \oplus C_{j'}^{(i)}$ where $C_j^{(i)}$ and $C_{j'}^{(i)}$ are two ciphertext bytes associated to the i-th trace, and the relation between j and j' is given through the ShiftRows operation of AES. The authors use $j = 12$ and $j' = 8$. The measured SNR equals 0.01554 (see Appendix D Figure 12).

- **AES_RD** is obtained from an 8-bit AVR microcontroller where a random delay desynchronization is implemented [CK09][e]. This countermeasure shifts each trace following a random variable of 0 to $N^{[0]}$. This renders the attack more difficult because of the misalignment. Similarly to DPA-contest v4, the sensitive variable is the first round S-box operation where each trace is labeled as such $Y^{(i)}(k^*) = Sbox[P_0^{(i)} \oplus k^*]$. The measured SNR equals 0.0073 (see Appendix D Figure 13).

[c] http://www.dpacontest.org/v4/42_traces.php
[d] https://github.com/AESHD/AES_HD_Dataset
[e] https://github.com/ikizhvatov/randomdelays-traces

- **ASCAD** is introduced in [PSB+18] and is the first open database [f] that has been specified to serve as a common basis for further works on the application of deep learning techniques in the side-channel context. The target platform is an 8-bit AVR microcontroller (ATmega8515) where a masked AES-128 is implemented. In addition, a random delay is applied to the set of traces in order to make it more robust against side-channel attacks. The leakage model is the first round S-box operation such that $Y^{(i)}(k^*) = Sbox[P_3^{(i)} \oplus k^*]$. As explained in the paper, the third byte is exploited. The measured SNR equals 0.007 (see Appendix D Figure 14).

Remark on the learning rate. During the training phase, for each architecture, we use a technique called *One Cycle Policy* [ST17, Smi17, Smi18] that helps choosing the right *learning rate*. The learning rate (LR) is one of the most challenging hyperparameter to tune because it defines the learning time and the robustness of the training. If it is too small, the network is going to take a long time to learn. If it is too high, each learning step will go over the loss minimum. The *One Cycle Policy* gives very fast results to train complex models. Surprisingly, using this policy we can pick much larger learning rates and significantly improve the learning time while preventing overfitting.

5.2 Results on synchronized traces

As explained in Subsubsection 4.4.1, only 1 convolutional block is set for each network. Following our methodology, we want to locate local perturbations that generate confusion during the classification of the network. Thus, the length of the filters is set to 1. Finally, the number of filters and the dense layers that compose the classification part should be managed depending on the system. Moreover, to accelerate the learning phase, we pre-process each dataset such as all trace samples are standardized and normalized between 0 and 1 [LBOM12].

5.2.1 DPA-contest v4

DPA-contest v4 is the easiest dataset because we consider it without countermeasures and the relevant information leak a lot. Thus the extraction of sensitive variables is straightforward for the network. To generate our CNN, we divide the dataset into three subsets such as 4000 traces are used for the training, 500 for the validation set and 500 for attacking the device. Only 2 filters are used and the classification part is composed of one dense layer of 2 nodes. Finally, we set our learning rate to 10^{-3}. The network is trained for 50 epochs with a batch size of 50.
Visualizations help to evaluate the training process (see Appendix F Figure 16). As explained in section Subsection 3.2, the weight visualization shows that the convolutional part is accurately set because the PoIs are extracted. Furthermore, by looking at the gradient visualization, we can consider that the classification part is also accurately working because the PoIs detected at the end of the network are the same as those recognized by the weight visualization. Thus, no information is lost between the convolutional part and the end of the network.
We compare our result with [PHJ+19] in which the authors published the best performance on DPA-contest v4 by using deep learning techniques. The results related to $\bar{N}t_{GE}$ seem to be similar. However, when we compare the complexity of the networks, we verify that the complexity associated with our network is 6 times lower than the previously best architecture. By reducing the complexity of the network, we remarkably reduce the learning time (see Table 1). As conclusion, our network is more appropriate.

[f] https://github.com/ANSSI-FR/ASCAD

Table 1: Performance comparison on DPA-contest v4

	State-of-the-art ([PHJ+19])	**Our methodology** (Subsection 4.4)
Complexity (trainable parameters)	52112	8782
Guessing Entropy $\bar{N}t_{GE}$	4	3
Learning time (seconds)	1000	23

5.2.2 AES_HD

We use 75000 measurements such that 50000 are randomly selected for the training process (45000 for the training and 5000 for the validation) and we use 25000 traces for the attack phase. We set the number of filters to 2. Because the PoIs should be accurately revealed, we minimize the classification part with 1 dense layer of 2 neurons. Finally, we set our learning rate to 10^{-3}. The training runs during 20 epochs with a batch size of 256.
From the Appendix F Figure 17, we can conclude that our model is not optimized. Indeed, the weight visualization shows us that the selection of features is effective because our network is able to find the relevant leakage points. When we look at the SNR value (see Appendix D Figure 12), the sensitive information leaks between samples 950 and 1200. That corresponds to the features that the convolutional layer detects. However, by visualizing the gradient, the global learning made by our network is not able to significantly recognize the relevant points. Through these visualization techniques, we are able to identify which part of the network needs to be optimized. Currently, there is no tool that allowing an interpretation of the classification part in order to improve its efficency. However, even if the classification part is not powerful, our network performs much better than the state-of-the-art.
We compare our results with the architecture proposed in [KPH+19] where they get the best performance on this dataset. In average, $\bar{N}t_{GE}$ reaches 25000 traces. By applying our methodology, we drastically improve this result. First, the new architecture is 31810 times smaller in terms of number of parameters. Now, only 31 seconds are necessary to train the network. Finally, our network outperforms the ASCAD network by getting $\bar{N}t_{GE}$ around 1300 (see Table 2).

Table 2: Performance comparison on AES_HD

	State-of-the-art ([KPH+19])	**Our methodology** (Subsection 4.4)
Complexity (trainable parameters)	104401280	3282
Guessing Entropy $\bar{N}t_{GE}$	25000	1300
Learning time (seconds)	6075	31

5.2.3 ASCAD with $N_0 = 0$

Following the previous studies, we can find a suitable architecture for the ASCAD database. For generating our network, we divide the dataset of ASCAD into three subsets: 45000 traces for the training set, 5000 for the validation set and 10000 for the attack phase. We initialize the number of filters to 4. Then, two dense layers composed of 10 neurons complete the network. Thanks the one cycle policy, we are able to configure our learning rate in the range $\left[5 \times 10^{-4}; 5 \times 10^{-3}\right]$. The network is trained for 50 epochs with a batch size of 50.
We compare our result with the original paper [PSB+18]. Surprisingly, we can notice that the network detects more PoIs than the mask and the masked value. Through the weight visualization, we can evaluate that the network recognizes relevant information related to the mask and the masked values, however a sensitive area (between samples 380 and

450) is detected by the network while nothing appears on the SNR. As the DPA-contest v4 experiment, the weight visualization and the gradient visualization are similar. The classification part appears optimized because no feature information is lost between the end of the convolutional part and the prediction layer.
When we compare the new performance, we notice that our new network is 3930 times less complex than the original paper. Finally, in terms of performance, we find a simpler network that achieves $\bar{N}t_{GE}$ in 191 traces while the original network reaches the same performance for 1146 traces (see Table 3).

Table 3: Performance comparison on ASCAD with $N_0 = 0$

	State-of-the-art ([PSB+18])	**Our methodology** (Subsection 4.4)
Complexity (trainable parameters)	66652444	16960
Guessing Entropy $\bar{N}t_{GE}$	1146	191
Learning time (seconds)	5475	253

5.3 Results on desynchronized traces

Once we have shown that our methodology seems efficient on synchronized traces for unprotected AES and 1^{st}-order masked AES, we decide to experiment our methodology on systems that implement a random delay desynchronization.

5.3.1 AES_RD

The dataset consists of 50000 traces of 3500 features each. For this experiment, we use 20000 traces for the training set, 5000 for the validation set and 25000 for attacking the device. Because desynchronization is generated, we have to use more than one convolutional block. Following Subsubsection 4.4.2, we recommend using three convolutional blocks. Concentrating the attention of the network into the PoIs is helpful in order to reduce the desynchronization effect. Our new architecture can be set following the Figure 5. Each hyperparameter is set as follows: for the first convolutional block, we set $filter\_size^{[1]} = 1$, $pooling\_stride^{[1]} = 2$, $number\_filters^{[1]} = 8$. For the second convolutional block, we configure $filter\_size^{[2]} = 50$, $pooling\_stride^{[2]} = 50$, $number\_filters^{[2]} = 16$. Finally, the hyperparameters of the third convolutional block are $filter\_size^{[3]} = 3$, $pooling\_stride^{[3]} = 7$, $number\_filters^{[3]} = 32$. Then, two fully-connected layers composed of 10 neurons define the classification part. Thanks to the one cycle policy, we are able to set our learning rate in the range $\left[10^{-4}; 10^{-3}\right]$. The network is trained for 50 epochs with a batch size of 50.
To evaluate the feature selection, we visualize the heatmaps associated with each layer in the convolutional part. Through the Appendix F Figure 19, we can evaluate the filters' selection by analyzing the heatmaps. As defined in Subsubsection 4.4.2, our methodology reduces the dimension of each trace in order to focus the features selection on the relevant points. Other points are discarded. Then, the information aggregation is easier for the classification part.

When our methodology is applied, we drastically reduce the impact of the desynchronization and the performance related to the network will be less impacted by this countermeasure. By comparing $\bar{N}t_{GE}$ with the state-of-the-art [KPH+19],our performance is similar. However, the complexity related to our network proposal is greatly reduced (40 times smaller).

Table 4: Performance comparison on AES_RD

	State-of-the-art ([KPH+19])	**Our methodology** (Subsection 4.4)
Complexity (trainable parameters)	512711	12760
Guessing Entropy $\bar{Nt}_{GE}$	10	5
Learning time (seconds)	4500	380

5.3.2 ASCAD

Finally, we test our methodology on a system that implements random delay and 1^{st} order-masked AES. We evaluate our methodology when $N^{[0]} = 50$ and $N^{[0]} = 100$. As in Subsubsection 5.2.3, we split our dataset into three subsets: 45000 traces for the training process, 5000 for the validation set and 10000 traces for allowing the attack.

Random delay: $N^{[0]} = 50$. By applying our new methodology, we want to generate a suitable architecture following Subsubsection 4.4.2. Thanks to Appendix F Figure 18, we can set $L = 3$ because three global leakage areas appear. For the first convolutional block, we set $filter\_size^{[1]} = 1$, $pooling\_stride^{[1]} = 2$, $number\_filters^{[1]} = 32$. For the second convolutional block, we configure $filter\_size^{[2]} = 25$, $pooling\_stride^{[2]} = 25$, $number\_filters^{[2]} = 64$. Finally, the hyperparameters of the third convolutional block are $filter\_size^{[3]} = 3$, $pooling\_stride^{[3]} = 4$, $number\_filters^{[3]} = 128$. The best performance is obtained when we configure three fully-connected layers with 15 neurons. We apply a range of learning rate between $\left[5 \times 10^{-4}; 5 \times 10^{-3}\right]$ and we set the number of epochs to 50 with a batch size 256.
To illustrate that the convolutional part is well customized, we visualize the heatmaps for evaluating the features selection. Through these heatmaps (see Appendix F Figure 20), we can notice that the network recognizes some influent patterns for the classification. Most of these patterns seem to correspond to the different leakages revealed by Appendix F Figure 18.
By applying this our methodology on this dataset, we remarkably outperform the state of the art. In the original paper, Prouff and al. don't reach a constant guessing entropy of 1 with 5000 traces. However, by applying our methodology, we converge towards a constant guessing entropy of 1 for 244 traces while the complexity of our networks is divided by 763. As a reminder, the performance related to the network trained with synchronized traces converges towards a guessing entropy of 1 with around 200 traces. Thus, we succeed at drastically reducing the impact of the random delay effect. Obviously, the network can be optimized with random search optimization, data augmentation or noise addition.

Table 5: Performance comparison on ASCAD with $N^{[0]} = 50$

	State-of-the-art ([PSB+18])	**Our methodology** (Subsection 4.4)
Complexity (trainable parameters)	66652444	87279
Guessing Entropy $\bar{Nt}_{GE}$	> 5000	244
Learning time (seconds)	5475	380

Random delay: $N^{[0]} = 100$. Finally, we apply our methodology on an even more complex system implementing a random delay with $N^{[0]} = 100$. For the first convolutional block, we set $filter\_size^{[1]} = 1$, $pooling\_stride^{[1]} = 2$, $number\_filters^{[1]} = 32$. For the second convolutional block, we configure $filter\_size^{[2]} = 50$, $pooling\_stride^{[2]} = 50$, $number\_filters^{[2]} = 64$. Finally, the hyperparameters of the third convolutional block are $filter\_size^{[3]} = 3$, $pooling\_stride^{[3]} = 2$, $number\_filters^{[3]} = 128$. Thanks to the one cycle policy, we define our learning rate in the range $\left[10^{-3}; 10^{-2}\right]$ for allowing a robust

training and reaching a good local minimum. As previously, our network is trained during 50 epochs with a batch size 256.
Thanks to the visualization tools (see Appendix F Figure 21, we can interpret our network and conclude that it is suitable for our evaluation because the gradient visualization and heatmaps are similar and show the same relevant points as the SNR.
Our network is widely less complex than the architecture introduced in the original ASCAD paper and $\bar{N}t_{GE}$ is outperformed. Indeed, we converge towards a constant guessing entropy of 1 with 270 traces while the original paper couldn't converge with 5000 traces. Furthermore, by reducing the complexity, we are able to train our network much faster. By comparing this performance with Subsubsection 5.2.3, we note that our methodology awfully reduce the impact of the desynchronization. Indeed, the performance of the two models is similar. As consequence, our methodology removes the effect of the random delay countermeasure.

Table 6: Performance comparison on ASCAD with $N^{[0]} = 100$

	State-of-the-art ([PSB+18])	**Our methodology** (Subsection 4.4)
Complexity (trainable parameters)	66652444	142044
Guessing Entropy $\bar{N}t_{GE}$	> 5000	270
Learning time (seconds)	5475	512

6 Conclusion

In this paper, we study the interpretability of Convolutional Neural Networks. We interpret the impact of different hyperparameters that compose the feature selection part in order to generate more suitable CNN. To evaluate it, we introduce two visualization tools called Weight visualization and Heatmaps that help analyzing which patterns are influent during the training process. These patterns are similar to PoIs that are well-known in the side-channel context.
We show theoretically the effect of the length of filters and on the number of convolutional blocks and using these visualization techniques in order to verify our demonstrations. These visualization techniques help us in order to find a methodology for generating appropriate CNN architecture according to the countermeasures implemented. When an evaluator wants to generate a network for synchronized traces, we recommend using only one convolutional block and minimizing the length of the filters for accurately identifying the relevant information. However, in the case where desynchronization is applied, we introduce an architecture that enables the network to focus its features selection on the PoIs themselves while the dimensionality of the traces is reduced. This helps to drastically reduce the complexity of the CNN.
Our methodology is applied on 4 datasets that offer a large variety of use-cases. For all of them, our methodology outperforms the state-of-the-art. Indeed, our attacks are performed much faster and each new architecture is less complex than the state of-the-art. Thus, this paper shows that the networks needed for performing side channel attacks don't need to be complex. This methodology is not focused on optimizer hyperparameters and could benefit from techniques such as the data augmentation [CDP17] or adding noise [KPH+19].
In some cases, we show that the aggregation of information through the classification part could be difficult. Future works could study this issue to generate suitable fully-connected layers. Finally, more complex systems with dynamic desynchronization or in presence of higher order masking scheme could be analyzed.

Acknowledgement

The authors would like to thank François Dassance for the fruitful discussions about this work.

References

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The em side—channel(s). In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 29–45, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[APSQ06] Cédric Archambeau, Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. Template attacks in principal subspaces. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2006.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

[BBBK11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain.*, pages 2546–2554, 2011.

[BBD+14] Shivam Bhasin, Nicolas Bruneau, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. Analysis and improvements of the dpa contest v4 implementation. In Rajat Subhra Chakraborty, Vashek Matyas, and Patrick Schaumont, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 201–218, Cham, 2014. Springer International Publishing.

[BBM+15] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLOS ONE*, 10(7):1–46, 07 2015.

[BCN18] L. Bottou, F. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[BL12] Timo Bartkewitz and Kerstin Lemke-Rust. Efficient template attacks based on probabilistic multi-class support vector machines. In Stefan Mangard, editor, *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2012.

[BPK92] H. Bischof, A. Pinz, and W. G. Kropatsch. Visualization methods for neural networks. In *Proceedings., 11th IAPR International Conference on Pattern Recognition. Vol.II. Conference B: Pattern Recognition Methodology and Systems*, pages 581–585, Aug 1992.

[C+15] François Chollet et al. Keras. https://keras.io, 2015.

[CDP16] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Kernel discriminant analysis for information extraction in the presence of masking. In Kerstin Lemke-Rust and Michael Tunstall, editors, *Smart Card Research and Advanced Applications - 15th International Conference, CARDIS 2016, Cannes, France, November 7-9, 2016, Revised Selected Papers*, volume 10146 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2016.

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without pre-processing. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 45–68. Springer, 2017.

[CK09] Jean-Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 156–170, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, pages 13–28, London, UK, UK, 2003. Springer-Verlag.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[HGG19] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Deep neural network attribution methods for leakage analysis and symmetric key recovery. *IACR Cryptology ePrint Archive*, 2019:143, 2019.

[HGM+11] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *J. Cryptographic Engineering*, 1(4):293–302, 2011.

[HOT06] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[HOWT06] Geoffrey Hinton, Simon Osindero, Max Welling, and Yee-Whye Teh. Unsupervised discovery of nonlinear structure using contrastive backpropagation. *Cognitive science*, 30:725–31, 07 2006.

[HZ12] Annelie Heuser and Michael Zohner. Intelligent machine homicide - breaking cryptographic devices using support vector machines. In Werner Schindler and Sorin A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, volume 7275 of *Lecture Notes in Computer Science*, pages 249–264. Springer, 2012.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision*

(ICCV), ICCV '15, pages 1026–1034, Washington, DC, USA, 2015. IEEE Computer Society.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[Kle17] Matthew Kleinsmith. CNNs from different viewpoints - prerequisite : Basic neural networks, 2017.

[KPH+19] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):148–179, 2019.

[KUMH17] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 971–980. Curran Associates, Inc., 2017.

[KW52] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466, 09 1952.

[LBLL09] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *J. Mach. Learn. Res.*, 10:1–40, June 2009.

[LBM14] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. Power analysis attack: an approach based on machine learning. *IJACT*, 3(2):97–115, 2014.

[LBOM12] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. *Efficient BackProp*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[LPMS18] Liran Lerman, Romain Poussier, Olivier Markowitch, and François-Xavier Standaert. Template attacks versus machine learning revisited and the curse of dimensionality in side-channel analysis: extended version. *J. Cryptographic Engineering*, 8(4):301–313, 2018.

[Man04] Stefan Mangard. Hardware countermeasures against DPA ? A statistical analysis of their effectiveness. In Tatsuaki Okamoto, editor, *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, volume 2964 of *Lecture Notes in Computer Science*, pages 222–235. Springer, 2004.

[MDP19a] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. A comprehensive study of deep learning for side-channel analysis. *IACR Cryptology ePrint Archive*, 2019:439, 2019.

[MDP19b] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. Gradient visualization for general characterization in profiling attacks. In Ilia Polian and Marc Stöttinger, editors, *Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings*, volume 11421 of *Lecture Notes in Computer Science*, pages 145–167. Springer, 2019.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, volume 10076 of *Lecture Notes in Computer Science*, pages 3–26. Springer, 2016.

[OH08] Simon Osindero and Geoffrey E Hinton. Modeling image patches with a directed hierarchy of markov random fields. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1121–1128. Curran Associates, Inc., 2008.

[ON15] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[PEC19] Guilherme Perin, Baris Ege, and Lukasz Chmielewski. Neural network model assessment for side-channel analysis. Cryptology ePrint Archive, Report 2019/722, 2019. https://eprint.iacr.org/2019/722.

[PGZ+18] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *JMLR Workshop and Conference Proceedings*, pages 4092–4101. JMLR.org, 2018.

[PHJ+19] Stjepan Picek, Annelie Heuser, Alan Jovic, Shivam Bhasin, and Francesco Regazzoni. The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):209–237, 2019.

[PSB+18] Emmanuel Prouff, Remi Strullu, Ryad Benadjila, Eleonora Cagli, and Cécile Dumas. Study of deep learning techniques for side-channel analysis and introduction to ASCAD database. *IACR Cryptology ePrint Archive*, 2018:53, 2018.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, 1999.

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951.

[Smi17] Leslie N. Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017, Santa Rosa, CA, USA, March 24-31, 2017*, pages 464–472. IEEE Computer Society, 2017.

[Smi18] Leslie N. Smith. A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay. *CoRR*, abs/1803.09820, 2018.

[SMY09] François-Xavier Standaert, Tal Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 443–461. Springer, 2009.

[SPQ05] François-Xavier Standaert, Eric Peeters, and Jean-Jacques Quisquater. On the masking countermeasure and higher-order power analysis attacks. In *International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA*, pages 562–567. IEEE Computer Society, 2005.

[ST17] Leslie N. Smith and Nicholay Topin. Super-convergence: Very fast training of residual networks using large learning rates. *CoRR*, abs/1708.07120, 2017.

[SVZ14] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Workshop Track Proceedings*, 2014.

[vdMH08] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[ZF14] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 818–833, Cham, 2014. Springer International Publishing.

[ZKL+16] Bolei Zhou, Aditya Khosla, Àgata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2921–2929. IEEE Computer Society, 2016.

A Signal to Noise Ratio (Chipwhisperer dataset)

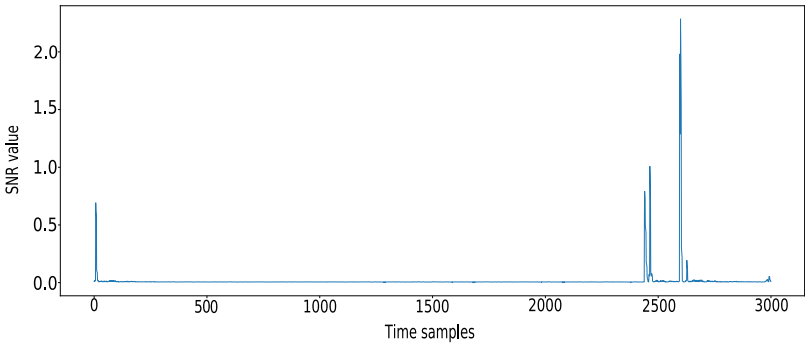

Figure 6: Signal to Noise Ratio related to the Chipwhisperer dataset

B Architectures for Hyperparameter

Table 7: Networks (length of filters)

Hyperparameters	value
nb_blocks	1
nb_filters	2
length_filter	**[1,3,5,7,9,11,13,15,17,19,21,23,25,50,75,100]**
nb_FC_layers	1
nb_neurons_FC	4

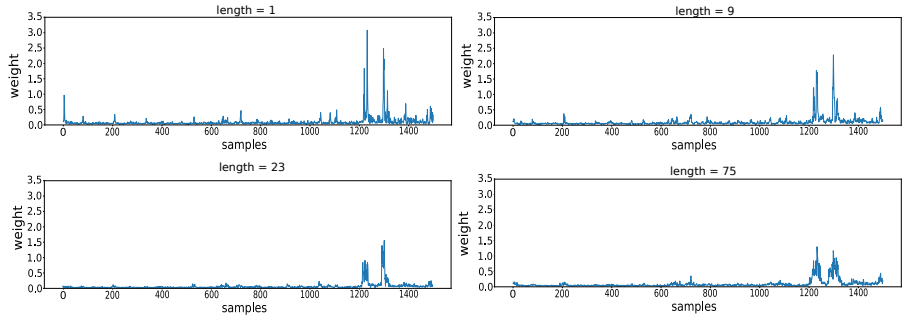

Figure 7: Impact of the filters length in the PoIs detection using weight visualization

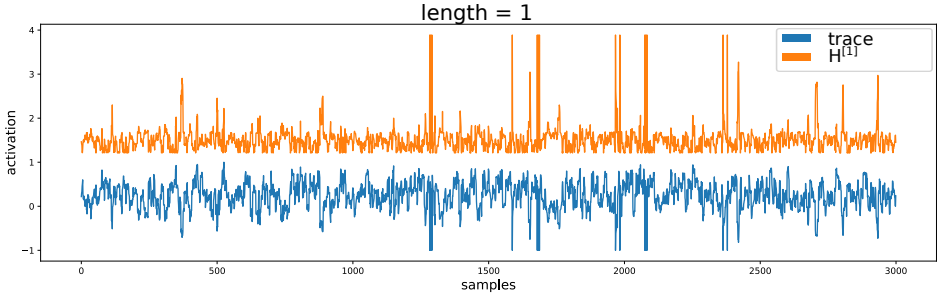

Figure 8: Heatmap of the first convolutional block $H^{[1]}$ with length 1

C Number of convolutional blocks : Pooling

Table 8: Networks (number of convolutional block)

Hyperparameters	value
nb_blocks	**[1,2,3,4,5]**
nb_filters	2
length_filter	1
nb_FC_layers	1
nb_neurons_FC	4

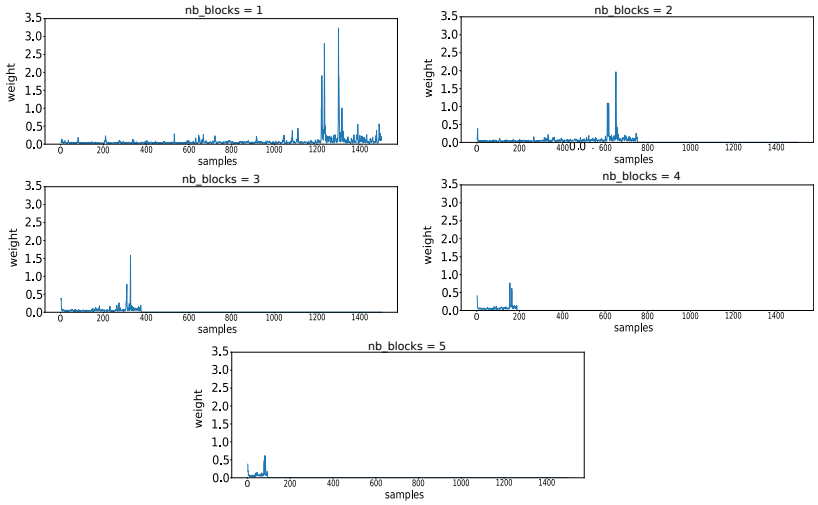

Figure 9: Impact of the number of convolutional blocks in the PoIs detection (Average Pooling) using weight visualization

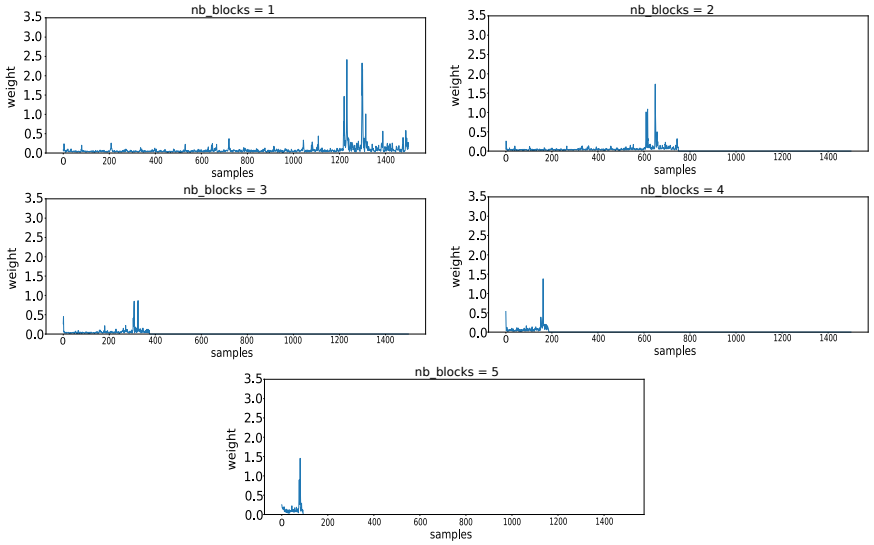

Figure 10: Impact of the number of convolutional blocks in the PoIs detection (Max Pooling) using weight visualization

D Signal to Noise Ratio for experimental datasets

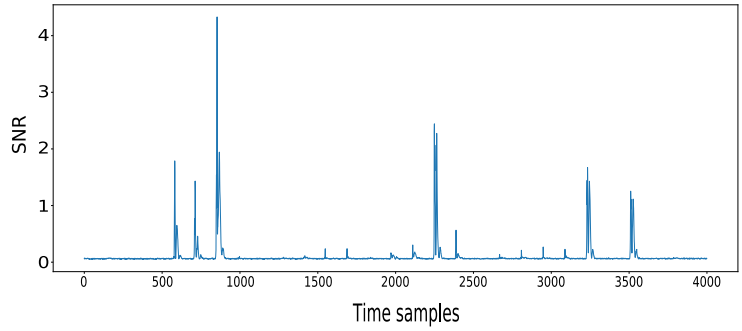

Figure 11: SNR for **DPA-contest v-4** dataset

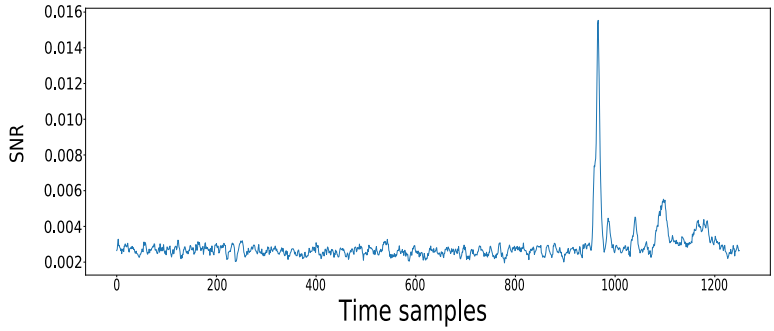

Figure 12: SNR for **AES_HD** dataset

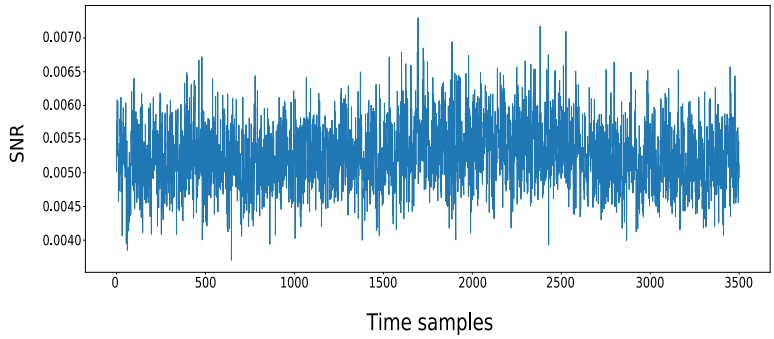

Figure 13: SNR for **AES_RD** dataset

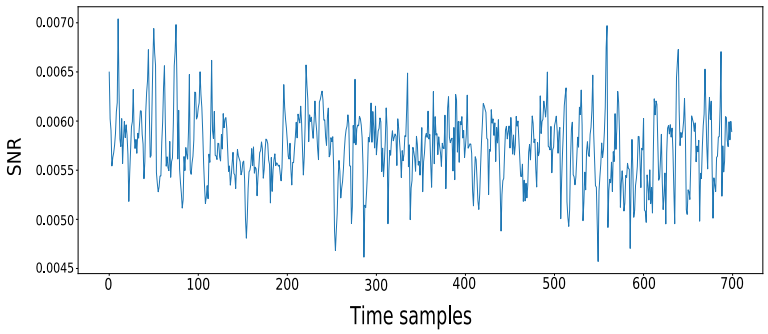

Figure 14: SNR for **ASCAD** dataset

E Methodology for generating architecture (desynchronized traces)

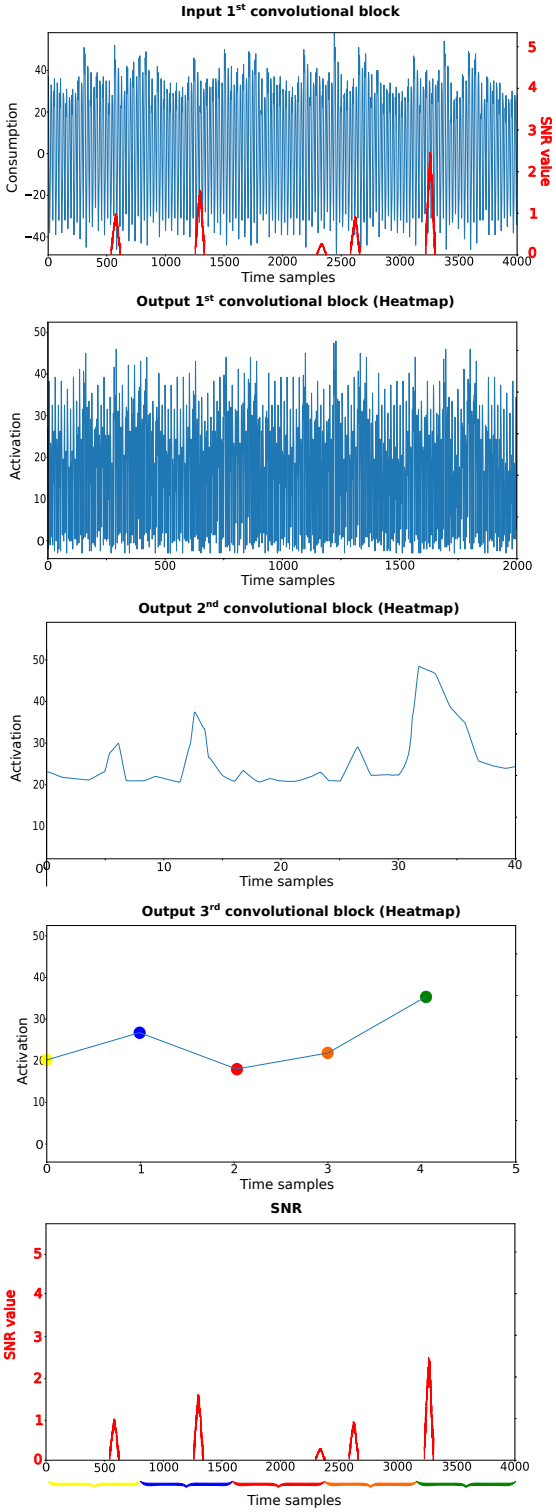

Figure 15: Output of each convolutional layer

F Performance visualization

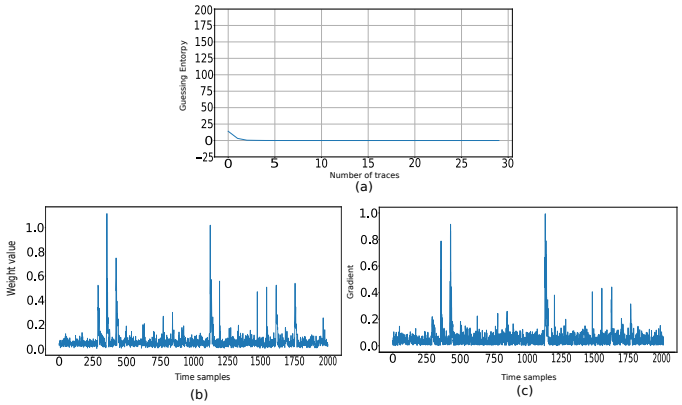

Figure 16: **DPA-contest v4** (a) Guessing entropy result ; (b) Weight visualization ; (c) Gradient visualization

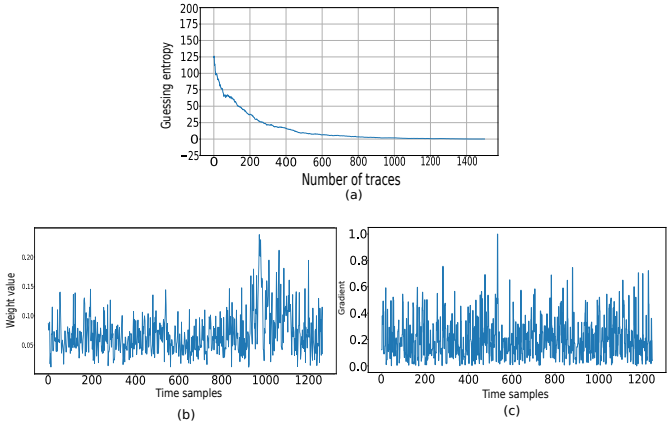

Figure 17: **AES_HD** (a) Guessing entropy result ; (b) Weight visualization ; (c) Gradient visualization

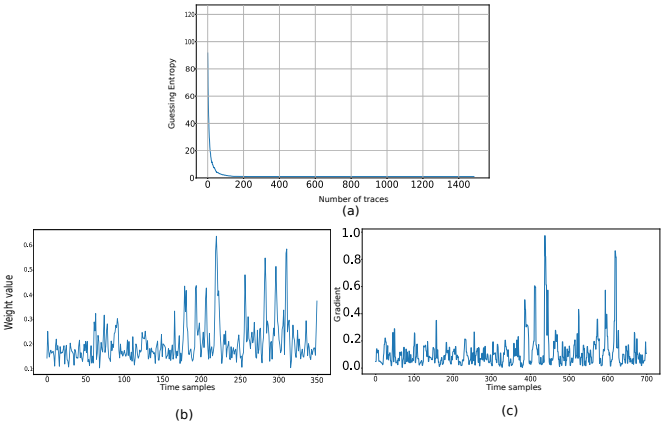

Figure 18: **ASCAD (traces synchronized)** (a) Guessing entropy result ; (b) Weight visualization ; (c) Gradient visualization

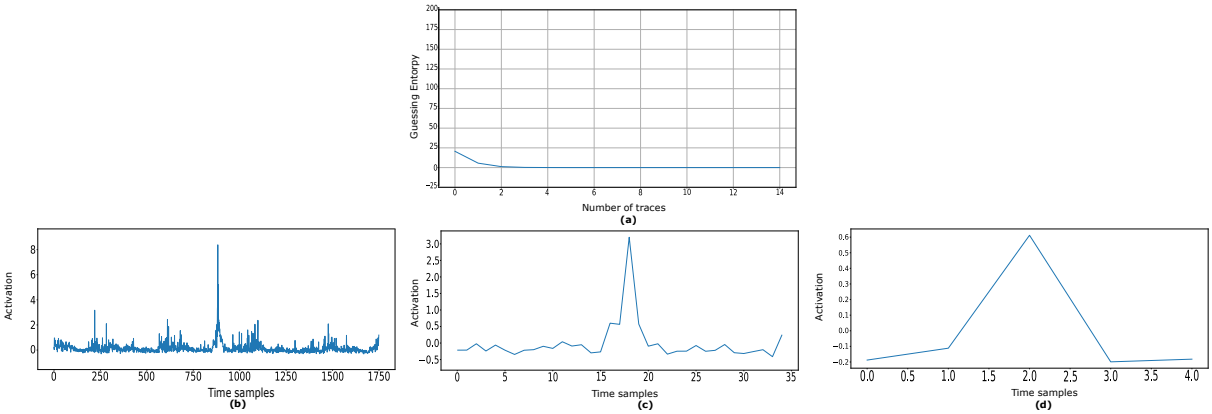

Figure 19: **AES_RD** (a) Guessing entropy ; (b) Output 2^{nd} convolutional layer ; (c) Output 3^{rd} convolutional layer ; (d) Output 3^{rd} convolutional block

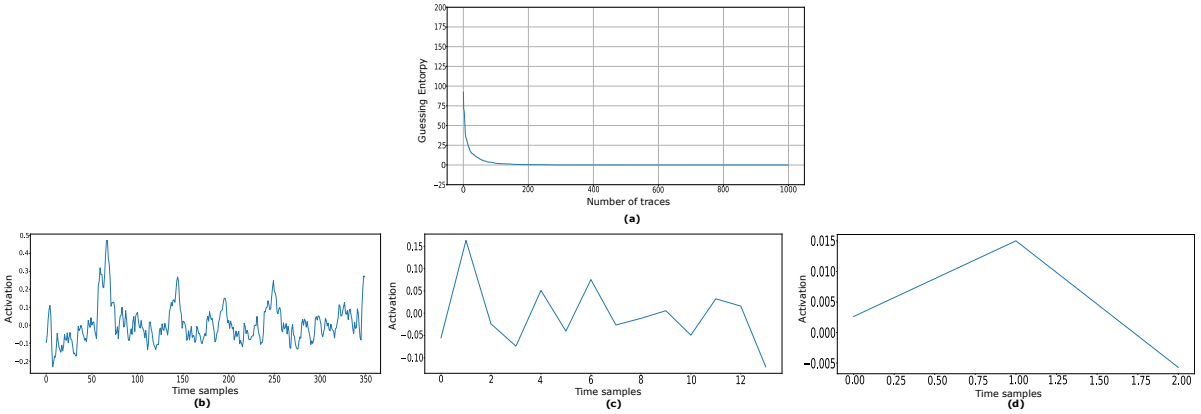

Figure 20: **ASCAD dataset with** $N^{[0]} = 50$ (a) Guessing entropy ; (b) Output 2^{nd} convolutional layer ; (c) Output 3^{rd} convolutional layer ; (d) Output 3^{rd} convolutional block

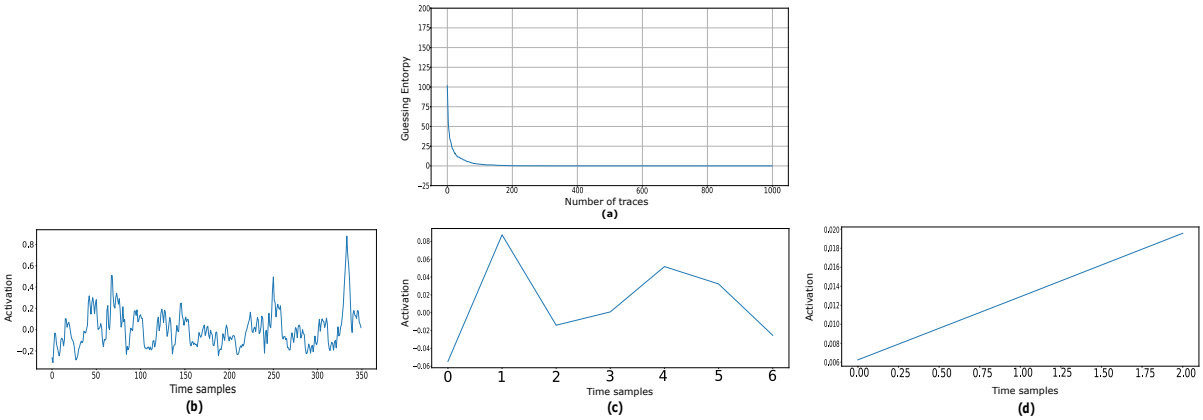

Figure 21: **ASCAD dataset with** $N^{[0]} = 100$ (a) Guessing entropy ; (b) Output 2^{nd} convolutional layer ; (c) Output 3^{rd} convolutional layer ; (d) Output 3^{rd} convolutional block