Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database

– Long Paper –

Ryad Benadjila[1], Emmanuel Prouff[1], Rémi Strullu[1], Eleonora Cagli[2] and Cécile Dumas[2]

[1] ANSSI, France
ryad.benadjila,emmanuel.prouff,remi.strullu@ssi.gouv.fr
[2] CEA, LETI, MINATEC Campus, F-38054 Grenoble, France
eleonora.cagli,cecile.dumasg@cea.fr

Abstract. To provide insurance on the resistance of a system against side-channel analysis, several national or private schemes are today promoting an evaluation strategy, common in classical cryptography, which is focussing on the most powerful adversary who may train to learn about the dependency between the device behaviour and the sensitive data values. Several works have shown that this kind of analysis, known as *Template Attacks* in the side-channel domain, can be rephrased as a classical Machine Learning *classification problem* with learning phase. Following the current trend in the latter area, recent works have demonstrated that deep learning algorithms were very efficient to conduct security evaluations of embedded systems and had many advantages compared to the other methods. Unfortunately, their *hyper-parametrization* has often been kept secret by the authors who only discussed on the main design principles and on the attack efficiencies. This is clearly an important limitation of previous works since (1) the latter parametrization is known to be a challenging question in Machine Learning and (2) it does not allow for the reproducibility of the presented results. This paper aims to address these limitations in several ways. First, completing recent works, we propose a comprehensive study of deep learning algorithms when applied in the context of side-channel analysis and we discuss the links with the classical template attacks. Secondly, we address the question of the choice of the hyper-parameters for the class of multi-layer perceptron networks and convolutional neural networks. Several benchmarks and rationales are given in the context of the analysis of a masked implementation of the AES algorithm. To enable perfect reproducibility of our tests, this work also introduces an open platform including all the sources of the target implementation together with the campaign of electromagnetic measurements exploited in our benchmarks. This open database, named ASCAD, has been specified to serve as a common basis for further works on this subject. Our work confirms the conclusions made by Cagli *et al.* at CHES 2017 about the high potential of convolutional neural networks. Interestingly, it shows that the approach followed to design the algorithm VGG-16 used for image recognition seems also to be sound when it comes to fix an architecture for side-channel analysis.

Keywords: Side-Channel Analysis · Machine Learning · Deep Learning

1 Introduction

Side-channel analysis is a class of cryptanalytic attacks that exploit the physical environment of a cryptosystem to recover some *leakage* about its secrets. It is often more efficient

than a cryptanalysis mounted in the so-called *black-box model* where no leakage occurs. In particular, *continuous side-channel attacks* in which the adversary gets information at each invocation of the cryptosystem are especially threatening. Common attacks as those exploiting the running-time, the power consumption or the electromagnetic radiations of a cryptographic computation fall into this class. Many implementations of block ciphers have been practically broken by continuous side-channel analysis — see for instance [KJJ99, BCO04, Mes00, MPO05] — and securing them has been a long-standing issue for the embedded systems industry.

Side channel attacks exploit information leaked from the physical implementations of cryptographic algorithms. Since this leakage (e.g. the power consumption or the electromagnetic emanations) depends on some small part of the internally used secret key, the adversary may perform an efficient key-recovery attack to reveal this sensitive data. Amongst the Side-Channel Attacks (SCA), two classes may be distinguished:

- The set of so-called *profiling* SCA attacks which are actually the most powerful ones since they assume that the adversary may priorly use an open copy of the final target to precisely tune all the parameters of the attack. A profiling SCA consists of two steps. First, the adversary procures a copy of the target device and uses it to characterize the physical leakage. Second, he performs a key-recovery attack on the target device. This category of attacks includes Templates Attacks [CRR02] and Stochastic models (a.k.a. Linear Regression Analysis) [DPRS11, Sch08, SLP05].
- The set of so-called *non-profiling SCA* which corresponds to a much weaker adversary who has only access to the physical leakage captured on the target device. To recover the secret key used, he performs some statistical analyses to detect the dependency between the leakage measurements and this sensitive variable. This set of non-profiling attacks includes, among others, Differential Power Analysis (DPA) [KJJ99], Correlation Power Analysis (CPA) [BCO04] and Mutual Information Analysis (MIA) [GBPV09, BGP+11].

A line of works has studied new profiling attacks based on Machine Learning (ML).

1.1 Related Works.

Several works have investigated the application of Machine Learning (ML) techniques to defeat both unprotected [BLR13, HZ12, HGM+11, LBM14, LPB+15] and protected cryptographic implementations [GHO15, LMBM13]. These contributions focus mainly on two techniques: the *Support Vector Machine* (SVM) [CV95, WW98] and *Random Forest* (RF) [RM08]. Practical results on several datasets have demonstrated the ability of these attacks to perform successful key recoveries. Besides, authors in [HZ12] have shown that the SVM-based attack outperforms the Template Attack when applied on highly noisy traces while [LPB+15] have experimentally argued that ML (and RF in particular) become(s) interesting if the amount of observations available for profiling is small while the dimension of the latter observations is high. Following the current trend in the Machine Learning area, recent works have paid more attention to Deep Learning (DL) algorithms like multi-layer perceptron networks (MLP) [MDM16, MHM13, MMT15] or convolutional neural networks (CNN) [CDP17, MPP16]. In the series of papers [MDM16, MHM13, MMT15], Martinasek and co-authors have compared methods based on MLP with other (more classical) approaches such as Templates Attacks or Stochastic Attacks. The target is an unprotected implementation of the AES-128 algorithm running on a PIC 8-bit micro-controller. The hyper-parameters of the MLP are given together with some partial information about the training. These studies show that techniques coming from Machine Learning theory are valuable alternatives to the original profiling attacks published in [CRR02] and even often outperform them. They moreover go further by showing that

they can be applied to measurements with very high dimension (while e.g. Templates Attacks are difficult to apply when the measurements dimension is greater than 100) and are robust to signal deformation like jittering (for CNN). The counterpart of the great efficiency of these attacks is that they are difficult to parametrized which has slowed down their deployment in the security evaluation industry. Moreover, while the papers present promising attack results, they do not give precise information about the parametrization of the algorithms, nor about their training. This is an important limitation which does not allow for the reproducibility of the analyses and hence hampers the development of the Deep Learning approach in the embedded security community. More generally, a common framework to study and compare the effectiveness of Machine Learning methods against embedded implementations of cryptographic algorithms is today missing.

1.2 Contributions.

In this paper, our main objective is to conduct a comprehensive and in-depth study of the application of Deep Learning theory in the context of side-channel attacks. In particular, we discuss several parametrization options and we present a large variety of benchmarks which have been used to either experimentally validate our choices or to help us to take the adequate decision.[1] The methodologies followed for the hyper-parameters' selection may be viewed as a proposal to help researchers to make their own choice for the design of new deep learning models. For most of the final choices made for the configuration of our models, we were not able to get a formal explanation and hence we of course do not claim that they are optimal. Since the current state of Machine Learning theory does not yet provide clear foundations to conduct such analyses, we think that having methodologies (even *ad hoc*) is a first necessary step which opens the way for further research in this domain. Our study also shows that convolutional neural networks are almost as efficient as multi-layer perceptron networks in the context of perfectly synchronized observations, and outperform them in presence of desynchronization/jittering. This suggests that CNN models should be preferred in the context of SCA (even if they are more difficult to train). When it comes to choose a base architecture for the latter models, our study shows that, surprisingly, the 16-layer network VGG-16 used by the VGG team in the ILSVRC-2014 competition [SZ14] is a sound starting point (other public models like ResNet-50 [HZRS16] or Inception-v3 [SVI+16] are shown to be inefficient against masked implementations). It allows us to design architectures which, after training, are better than classical Templates Attacks even when combined with dimension reduction techniques like Principal Component Analysis (PCA) [Pea01]. By training (with 75 epochs) our CNN_{best} architecture on a subset of $50,000$ 700-dimensional traces of ASCAD database, we outperformed the other tested models on highly desynchronized traces while we achieved one of the best performances on small desynchronized traces. Figure 1 gives a brief summary of our conclusions. MLP_{best} corresponds to the most efficient multi-linear perceptrons networks we succeed to train on ASCAD database, and the template attack has been preceded by a PCA to reduce the dimension of the input traces from 700 to 50. A desynchronization amount of N_{max} signifies that each trace (for the training and the attack) has been randomly shifted by $\delta \in [0..N_{max}]$ points to the left. More detailed descriptions of this benchmark and of the testing framework are given in Sect(s). 3, 4 and 5.

All the benchmarkings have been done with the same target (and database) which corresponds to an AES implementation secured against first order side-channel attacks and

[1] Some libraries (such as `hyperopt` or `hyperas`, [BYC13]) could have been tested to automatize the search of accurate hyper-parameters in pre-defined sets. However, since they often perform a random search of the best parameters ([BB12]), they do not allow studying the impact of each hyper-parameter independently of the others on the side-channel attack success rate. Moreover, they have been defined to maximize classical machine learning evaluation metrics and not SCA ranking functions which require a batch of test traces.

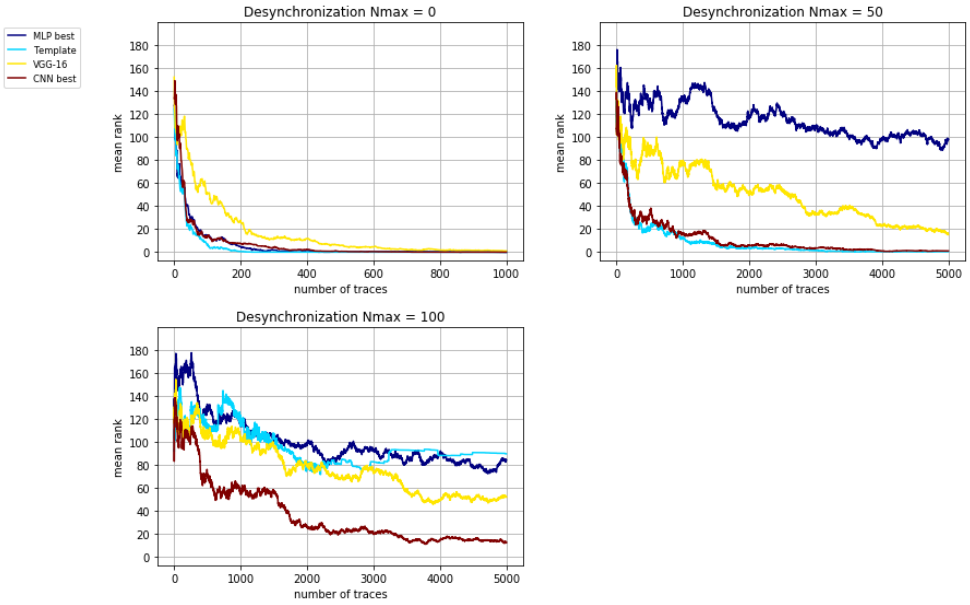

Figure 1: Mean ranks of the best models for a desynchronization amount in $\{0, 50, 100\}$.

developed in assembly for an ATMega8515 component. A signal-to-noise characterization has been done to validate that there is no first-order leakage. This project has been published on [ANS18b]. To enable perfect reproducing of our experiments and benchmarks, we also chose to publish the electromagnetic measurements acquired during the processing of our target AES implementation (available in [ANS18a]) together with example Python scripts to launch some initial training and attacks based on these traces. We think that this database may serve as a common basis for researchers willing to compare their new architectures or their improvements of existing models.

2 Preliminaries

2.1 Notations

Throughout this paper we use calligraphic letters as $\mathcal{X}$ to denote sets, the corresponding upper-case letter X to denote random variables (random vectors $\vec{X}$ if with an arrow) over $\mathcal{X}$, and the corresponding lower-case letter x (resp. $\vec{x}$ for vectors) to denote realizations of X (resp. $\vec{X}$). Matrices will be denoted with bold capital letters. The i-th entry of a vector $\vec{x}$ is denoted by $\vec{x}[i]$, while the i-th observation of a random variable X is denoted by x_i. The *probability mass function* (aka the *probability distribution function*, pdf for short) of a *discrete* random variable $\vec{X}$ will be denoted by $f_{\vec{X}}$. It is defined for any possible realization $\vec{x}$ of $\vec{X}$ by $f_{\vec{X}}(\vec{x}) = \mathsf{P}[\vec{X} = \vec{x}]$. The symbol $\mathsf{E}[\,]$ denotes the expected value, and might be subscripted by a random variable $\mathsf{E}_X[\,]$, or by a probability distribution $\mathsf{E}_{f_X}[\,]$, to specify under which probability distribution it is computed. Side-channel traces will be viewed as discrete realizations of a random column vector $\vec{L}$ with values in $[0, \omega]^D$ where ω depends on the vertical resolution of the oscilloscope used for the acquisitions (usually, we have $\omega \in \{8, 10, 12\}$). During their acquisition, a target sensitive variable $Z = \varphi(P, K)$ is handled, where P denotes some public variable, *e.g.* a plaintext chunk, and K the part of a secret key the attacker aims to retrieve. The value assumed by such a variable is viewed as a realization $z \in \mathcal{Z} = \{z^1, z^2, \ldots, z^{|\mathcal{Z}|}\}$ of a discrete finite random variable Z. We will

sometimes represent the values $z^j \in \mathcal{Z}$ *via* the so-called *one-hot encoding* representation, assigning to z^j a $|\mathcal{Z}|$-dimensional vector, with all entries equal to 0 and the j-th entry equal to 1: $z^j \to \vec{z^j} = (0, \ldots, 0, \underbrace{1}_{j}, 0, \ldots, 0)$. Under this notation, the random variable Z turns into a random vector $\vec{Z}$.

2.2 Side-Channel Analysis

Side-Channel Analysis (SCA) aims at exploiting noisy observations $\vec{L}$ of the processing of an algorithm to recover its secret parameter. When the SCA adversary has the ability to use an open device (*i.e.* a device on which he can control, at least partially, all the inputs of the algorithm, including the secret parameters), a particular class of attacks named *profiling* may be executed.

2.2.1 Profiling SCA.

A *profiling SCA* is composed of two phases: a profiling (or *characterization*, or *training*) phase, and an attack (or *matching*) phase.

During the profiling step, the attacker may construct what is called in Machine Learning language a *generative model*[2] [Bis06], computing for every possible $k \in \mathcal{K}$ an estimation $\hat{g}_k$ of the following conditional probability distribution function:

$$g_k : (\vec{\ell}, p) \mapsto \mathsf{P}[\vec{L} = \vec{\ell} | (P, K) = (p, k)] \ . \tag{1}$$

The estimation $\hat{g}_k$ is processed on a *profiling set* $\mathcal{D}_{\text{profiling}} \doteq \{\vec{\ell_i}, p_i, k_i\}_{i=1,\ldots,N_p}$ of size N_p, which is a set of traces $\vec{\ell_i}$ acquired under known guessable chunks p_i and k_i of the cryptographic algorithm inputs. In the rest of this paper, the set of traces is denoted by $\mathcal{L}$, while the set of corresponding inputs is denoted by $\mathcal{Y}$.

Remark 1. In cases where the SCA targets a particular processing in the form $\varphi(P, K)$, the probability in the right-hand side of (1) may w.l.g. be rewritten $\mathsf{P}[\vec{L} = \vec{\ell} | \varphi(P, K) = z]$ and the profiling set may be rewritten $\mathcal{D}_{\text{profiling}} \doteq \{\vec{\ell_i}, z_i\}_{i=1,\ldots,N_p}$ with $\mathcal{L} = \{\vec{\ell_i}; i \leqslant N_p\}$ and $\mathcal{Y} = \{z_i; i \leqslant N_p\}$.

During the attack step, the attacker gets a new *attack set* $\mathcal{D}_{\text{attack}} \doteq \{\vec{\ell_i}, p_i\}_{i=1,\ldots,N_a}$ for which the secret parameter, say $k^\star$, is fixed but unknown. His goal is to recover the latter key. For such a purpose, the attacker who built a generative model must decide which of the pdf estimations $\hat{g}_k$, $k \in \mathcal{K}$, is the most likely knowing the attack set. It is well known that, under realistic assumptions on the distributions' nature, the most efficient way to make such a decision is to follow a *Maximum Likelihood* strategy which amounts to estimate the following *likelihood* $\vec{\mathsf{d}}_{N_a}[k]$ for every key candidate $k \in \mathcal{K}$, and then to select the key candidate that maximizes it:

$$\vec{\mathsf{d}}_{N_a}[k] = \prod_{i=1}^{N_a} \mathsf{P}[(P, K) = (p_i, k) \mid \vec{L} = \vec{\ell_i}] = \prod_{i=1}^{N_a} \frac{\mathsf{P}[\vec{L} = \vec{\ell_i} \mid (P, K) = (p_i, k)]}{f_{\vec{L}}(\vec{\ell_i})} \times f_{P,K}(p_i, k), \tag{2}$$

where (2) is obtained *via* Bayes' Theorem under the hypothesis that acquisitions are independent.[3] The estimation of (2) is simply done by replacing the probabilities $\mathsf{P}[\vec{L} = \vec{\ell_i} | (P, K) = (p_i, k)]$ in the right-hand term by their estimations $\hat{g}_k(\vec{\ell_i}, p_i)$. The vector $\vec{\mathsf{d}}_{N_a}$ is called *scores vector* and its kth coordinate is the score corresponding to key candidate k.

[2] The name *generative* is due to the fact that it is possible to generate synthetic traces by sampling from such probability distributions.

[3] In Templates Attacks the profiling set and the attack set are assumed to be different, namely the traces $\vec{\ell_i}$ involved in (2) have not been used for the profiling.

2.2.2 SCA Efficiency.

As it is classical in the context of side-channel analysis (see *e.g.* [SMY06]), the *efficiency of an attack at order o* will be defined in this paper as the minimum size N_a of the attack set which is needed to ensure that the attack succeeds in ranking the correct key among the o first likely ones. We will implicitly assume that the size N_p of the profiling set is fixed but sufficiently high. Clearly, for our definition of the efficiency, the more accurate the estimations of (1), the more efficient the attack step. The latter accuracy therefore plays a central role and we shall come back to this point in Section 2.4.

2.2.3 Leakage Dimensionality Issue.

The potentially huge dimensionality of $\vec{L}$ may make the estimation of (1) a very complex problem. To circumvent it, the adversary usually priorly exploits some statistical tests (*e.g.* SNR or T-Test) and/or dimensionality reduction techniques (*e.g.* Principal Component Analysis [Pea01], Linear Discriminant Analysis [Fis36], Kernel Discriminant Analysis [CDP16]) to select points of interest or an opportune combination of them. Then, denoting $\varepsilon(\vec{L})$ the result of such a dimensionality reduction, the attack is performed as described previously with the simple difference that $\vec{L}$ and $\vec{\ell_i}$ are respectively replaced by $\varepsilon(\vec{L})$ and $\varepsilon(\vec{\ell_i})$ in (1) and (2).

2.2.4 (Gaussian) Template Attacks.

The most popular way adopted until now to estimate the conditional probability (1) is the one that led to the well-established Gaussian Template Attack [CRR02]. It assumes that $\vec{L} \mid (P, K)$ (or equivalently $\varepsilon(\vec{L}) \mid (P, K)$ if a dimensionality reduction has been priorly applied) has a multivariate Gaussian distribution, and estimates the mean vector $\vec{\mu}_{p,k}$ and the covariance matrix $\mathbf{\Sigma}_{p,k}$ for each possible $(p, k) \in \mathcal{P} \times \mathcal{K}$ (*i.e.* the so-called templates). In this way the pdf (1) is approximated by the Gaussian pdf $f_{\vec{\mu}_{p,k}, \mathbf{\Sigma}_{p,k}}$. So, the Gaussian Template Attack is a strategy that makes use of a generative model. The same multivariate Gaussian assumption is the one that is made in *Quadratic Discriminant Analysis*, which is a well-known generative strategy in the Machine Learning literature [Fis36] to perform classification.

2.3 Machine Learning and Deep Learning

In Machine Learning theory, the problem of estimating $\mathsf{P}[\vec{L} \mid (P, K) = (p, k)]$, for some $(p, k) \in \mathcal{Y}$ with $\mathcal{Y} \doteq \mathcal{P} \times \mathcal{K}$, is known as a *prediction problem* (a.k.a. *generation* problem), while the estimation of $\mathsf{P}[(P, K) = (p, k) \mid \vec{L}]$ is referred to as a *classification problem*. In [LT16], the authors recall that the former pdf $\mathsf{P}[\vec{L} \mid (P, K) = (p, k)]$ "often has many simplifying features enabling accurate approximation, because it follows from some simple physical law or some generative model with relatively few free parameters (for example, its dependence on (p, k) may exhibit symmetry, locality and/or be of a simple form such as the exponential of a low-order polynomial). In contrast, the pdf $\mathsf{P}[(P, K) = (p, k) \mid \vec{L}]$ tends to be more complicated; [roughly speaking because] it makes no sense to speak of symmetries or polynomials involving a pair of discrete variables." Fortunately, the complex pdf is determined by the hopefully simpler one via Bayes' Theorem:

$$\mathsf{P}[\vec{L} \mid (P, K) = (p, k)] = \frac{\mathsf{P}[(P, K) = (p, k) \mid \vec{L}] \times f_{\vec{L}}}{f_{(P,K)}(p, k)} \quad .$$

Consequently, the both problems are linked. In particular to solve a classification problem one may or not priorly consider the prediction problem, constructing a generative model

as it is done in Gaussian Template Attacks. Otherwise a *discriminative model* can be sufficient: it consists in directly address the classification problem, *i.e.* directly estimate $\mathsf{P}[(P,K)=(p,k) \mid \vec{L}]$ without making use of the Bayes' inversion [Bis06]. If discriminative models have to be preferred to generative ones this is a widely debated issue in Machine Learning communities.

Deep Learning, and in particular deep neural networks, are nowadays the privileged tool to address the classification problem, and they can be exploited in a discriminative way. In such a case, which corresponds to our choice, they aim at directly constructing an approximation $\hat{\mathbf{g}}_{\vec{L},P}$ of the function $\vec{\ell}, p \mapsto (\mathsf{P}[(P,K)=(p,k) \mid \vec{L}=\vec{\ell}])_{k\in\mathcal{K}}$. The classification of a new leakage $\vec{\ell}$ observed for an input p is done afterwards by processing $\vec{y} = \hat{\mathbf{g}}_{\vec{L},P}(\vec{\ell}, p)$ and by choosing the key candidate $\hat{k}$ (or equivalently the label in the formalism of Machine Learning) such that $\hat{k} = \mathrm{argmax}_{k\in\mathcal{K}}\ \vec{y}[k]$. If the key-discrimination is done from several, say N_a, pairs $(\vec{\ell}_i, p_i)$ then the maximum likelihood approach is followed as in (2):

$$\vec{\mathsf{d}}_{N_a}[k] = \prod_{i=1}^{N_a} \vec{y}_i[k] \ , \tag{3}$$

where $\vec{y}_i$ denotes the output of (the model function) $\hat{\mathbf{g}}_{\vec{L},P}$ input with the pair $(\vec{\ell}_i, p_i)$. It may be checked that (3) is a simple rewriting of (2) obtained by replacing the conditional probability $\mathsf{P}[(P,K)=(p_i,k) \mid \vec{L}=\vec{\ell}_i]$ by its approximation $\vec{y}_i[k] = \hat{\mathbf{g}}_{\vec{L},P}(\vec{\ell}_i, p_i)[k]$.

Remark 2. In the context of side-channel analysis against block cipher implementations, it is common to label the observations/traces by an appropriate function $\varphi(p,k)$ instead of (p,k) (and both labelling are equivalent when the observations exactly corresponds to the processing of $\varphi(\cdot)$ since p is assumed to be known). It will be the case for the database used in the rest of the paper where $\varphi(\cdot)$ corresponds to the AES sbox. This leads to define $\hat{\mathbf{g}}_{\vec{L},P}$ as the approximation of the function $\vec{\ell}, p \mapsto (\mathsf{P}[\varphi(p,K)=z \mid \vec{L}=\vec{\ell}])_{z\in\mathrm{Im}(\varphi)}$ with $\mathrm{Im}(\varphi)$ denoting the image set of φ. The output of the model function $\hat{\mathbf{g}}_{\vec{L},P}$ input with $(\vec{\ell}, p)$ is hence a vector $\vec{y}'$ indexed by the values $z=\varphi(p,k)$ for k ranges over $\mathcal{K}$. To build a second vector $\vec{y}$ indexed by the key candidates k it suffices to process $\vec{y}[k] = \vec{y}'[\varphi(p,k)]$ for every $k \in \mathcal{K}$.

Remark 3. Another formulation of the classification problem could consist in directly looking for an estimation of the pdf $\mathsf{P}[K \mid \vec{L}, P]$: the deep neural networks will here be trained to classify the key candidates knowing the public input of the cryptographic primitive and the information leakage. However, even if this formulation may seem more natural (it perfectly matches the problem we want to resolve), it implies that the deep neural networks must not only recover the statistical dependency between the values of the manipulated data and the leakage, but also the function that links it to the key (*e.g.* the function $\varphi^{-1}(\cdot, P)$). Since the latter function may be complex (*e.g.* can be affinely equivalent to the inverse of an sbox), this can made the task of the deep neural networks harder, whereas the function φ is often already known by the adversary.

Deep Learning (DL) is a branch of Machine Learning whose characteristic is to avoid any manual feature extraction step from the model construction work-flow. For example, in Deep Learning the dimensionality issue discussed in Sect. 2.2.3 is not necessarily tackled out by preprocessing a dimensionality reduction function ε. As described below, the cascade of multiple layers that characterize DL models is indeed in charge of directly and implicitly extracting interesting features and of estimating the classifying model $\hat{\mathbf{g}}_{\vec{L},P}$. This approximation is searched in a family of functions (aka *models* in the Machine Learning terminology) specified *a priori* by the data analyst according to the specificities of the problem which is tackled out.

We conclude this sub-section by recalling some basic definitions and notions about neural networks and their training.

Neural Networks. A neural network has an *input layer* (the identity over the input datum $\vec{\ell}$), an *output layer* (the last function, whose output $\vec{y}$ is an estimation of the vector of conditional probabilities) and all other layers are called *hidden* layers. The so-called *neurons*, that give the name to the architecture, are the computational units (also named *nodes*) of the network and essentially process a scalar product between the coordinates of its input and a vector of *trainable weights* (or simply *weights*) that have to be *trained*. Each layer processes some neurons and the outputs of the neuron evaluations will form new input vectors for the subsequent layer. The numbers of layers in the neural networks, the dimension of the elementary units or the algebraic nature of the non-linear layers form the *architecture* of the network and define the family of functions/models. The identification of the best approximating function in this family is made by solving a minimization problem with respect to a metric which is specific to the application.

Training of Neural Networks. In a privileged setting, the *training phase* (*i.e.* the automatic tuning of the weights of the neurons) is done *via* an iterative approach which locally applies the (Stochastic) Gradient Descent algorithm [GBC16a] to minimize a *loss function* quantifying the *classification error* of the function $\hat{\mathbf{g}}_{\vec{L},P}$ over a training set which is a part of the profiling set. The cross-entropy [LH05, GBC16b] metric is a classical (and often by default) choice. It is smooth and decomposable, and therefore amenable to optimization with standard gradient-based methods. However, other metrics may be investigated and can potentially lead to better results [MHK10, SSZU15]. A training is said to be *full batch learning* if the full training database is processed before one update. At the opposite, if a single training input is processed at a time then the approach is named *stochastic*. In practice, one often prefers to follow an approach in between, called *mini-batch learning*, and to use small *batch* (aka group) of training inputs at a time during the learning. The approach is moreover said to be *online* when the training examples are drawn from a stream of continually created inputs rather than from a fixed-size training set over which several passes are made. The size of the mini-batch is generally driven by several efficiency/accuracy factors which are e.g. discussed in [GBC16b] (*e.g.* optimal use of the multi-core architectures, parallelization with GPUs, trade-off between regularization effect and stability, etc.).

An iteration over all the training datasets during the Stochastic Gradient Descent is called an *epoch*. The number of epochs is an important parameter to tune because small values may lead to under-fitting (the number of steps of the Gradient Descent is not sufficient and the model is too poor to capture a trend in the training dataset) and high values may lead to over-fitting (the model is too complex, it perfectly fit the training dataset but is not able to generalized its predictions to other datasets).

Several extensions and variants of the Stochastic Gradient Descent have been proposed in the context of deep learning. These variants, called *optimizers*, aim to adapt the *learning rate* (the step size) of the Gradient Descent during the training process. More details about the specification of neural networks will be given in the dedicated sections 3 and 4, but we will not go further on the optimization approaches and the interested reader may refer to [GBC16a].

Hyper-Parameters. All the parameters that define an architecture (called *architecture hyper-parameters* or simply *architecture parameters*), together with some other parameters that govern the training phase (called *training hyper-parameters* or simply *training parameters*), have to be carefully set by the attacker. [4] This point will discussed in

[4] When no ambiguity is present we will call simply *hyper-parameters* the architecture ones.

the following sections. Essentially, the strategy we have followed to finalize a choice of hyper-parameters is composed of three consecutive steps. First, we perform some ad-hoc preliminary tests to select a base model with fixed architecture parameters and then, we study the impact of each training parameter on the accuracy and efficiency of the base model after training. These two steps lead us to determine the hyper-parameters of a training procedure $\texttt{Training}(n_{\text{epochs}}, \mathit{batch\_size}, \mathit{optimizer}, \mathit{learning\_rate})$. Eventually, a third step consists in coming back to the initial choices made for the model architecture and to study the impact of each of them on the efficiency of the resulting attack after training with the procedure frozen during previous step.

2.4 Model Assessment and Selection

2.4.1 Evaluation Methodology

In the Machine Learning community, several evaluation frameworks are commonly applied to assess the performances of a model or to select the best parameters that suit to a parametrized family of models. These methods aim to provide an estimator of the performance of a metric (*e.g.* the accuracy) which does not depend on the choice of the training set $\mathcal{D}_{\text{train}}$ (on which the model is trained) and of the test set $\mathcal{D}_{\text{test}}$ (on which the model is tested) but only on their size.

The so-called *t-fold cross-validation* [FHT01] is currently the preferred evaluation method. Let c be a metric, $\hat{\mathbf{g}}$ a model to evaluate, and $\mathcal{D}_{\text{profiling}} = (\mathcal{L}, \mathcal{Y})$ a dataset with labels, the outline of the method is the following:

1. [optional] randomize the order of the labelled traces in $\mathcal{D}_{\text{profiling}}$,
2. split the samples and their corresponding labels into t disjoint parts of equal size $(\mathcal{L}_1, \mathcal{Y}_1), \ldots, (\mathcal{L}_t, \mathcal{Y}_t)$. For each $i \in [1..t]$, do:
   (a) set $\mathcal{D}_{\text{test}} \doteq (\mathcal{L}_i, \mathcal{Y}_i)$ and $\mathcal{D}_{\text{train}} \doteq (\bigcup_{j \neq i} \mathcal{L}_j, \bigcup_{j \neq i} \mathcal{Y}_j)$,
   (b) (re-)train[5] the model $\hat{\mathbf{g}}$ on $\mathcal{D}_{\text{train}}$,
   (c) compute the performance metric by evaluating the model on $\mathcal{D}_{\text{test}}$:
   $$\mathsf{c}_i = \mathsf{c}(\hat{\mathbf{g}}, \mathcal{D}_{\text{test}}) \ , \tag{4}$$
3. return the mean $\frac{1}{t} \sum_{i=1}^{t} \mathsf{c}_i$.

It is known that the t-fold cross-validation estimator is an unbiased estimator of the generalization performance. Its main drawback is its variance which may be large and difficult to estimate [B+96, BG05]. In this paper (Sect(s). 3 and 4), we perform a 10-fold cross-validation for each selection of the model parameters. The choice of $t = 10$ results in a trade-off between evaluation complexity and accuracy, since for each choice of parameters the model is trained 10 times with a substantial computing time, and the generalization performance estimator is computed among 10 values on different training sets, reducing the uncertainty on the evaluation metrics. The dataset $\mathcal{D}_{\text{profiling}}$ on which is performed the cross-validation is a fixed subset comprised of $50,000$ labelled traces, split at each iteration into $N_{\text{train}} = 45,000$ labelled traces for $\mathcal{D}_{\text{train}}$ and $N_{\text{test}} = 5,000$ labelled traces for $\mathcal{D}_{\text{test}}$.

2.4.2 Evaluation Metrics

We evaluate the performance of our models with three different metrics, which are: the *rank function*, the *accuracy* and the *computational time*.

[5] We insist here on the fact that the model is trained from scratch at each iteration of the loop over t.

The rank function is a commonly used metric in SCA for assessing the performance of an attack. Let us denote by $k^\star \in \mathcal{K}$ the key that has been used during the acquisition of the dataset $\mathcal{D}_{\text{profiling}}$. The *rank function* corresponding to a model $\hat{\mathbf{g}}$ trained with the dataset $\mathcal{D}_{\text{train}}$ and tested with the dataset $\mathcal{D}_{\text{test}}$ is defined by:

$$\mathsf{rank}(\hat{\mathbf{g}}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}}, n) = |\{k \in \mathcal{K} \mid \vec{\mathsf{d}}_n[k] > \vec{\mathsf{d}}_n[k^\star]\}| \ , \tag{5}$$

where $\vec{\mathsf{d}}_n[k]$ is the score for the candidate k as defined in (3) (replacing $\mathcal{D}_{\text{attack}}$ by $\mathcal{D}_{\text{test}}$ and N_a by n) and estimated from a modelling of the conditional probability done with $\mathcal{D}_{\text{train}}$ and a test done with the n first traces in $\mathcal{D}_{\text{test}}$. For example, if $k^\star$ has the highest score (resp. the lowest score), then its rank is 0 (resp. $|\mathcal{K}|-1$). Note that in this definition, the rank function depends on the choices of the training and test datasets. To get a better measure of the rank for given cardinalities N_{train} and N_{test} of $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ respectively, it is therefore more suitable to estimate its mean over several pairs of datasets:[6]

$$\mathsf{RANK}_{N_{\text{train}},n}(\hat{\mathbf{g}}) = \mathsf{E}[\mathsf{rank}(\hat{\mathbf{g}}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}}, n)] \ , \tag{6}$$

where the mean is defined over all the datasets $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ respectively of cardinality N_{train} and N_{test}, and where n is assumed to be bounded above by N_{test}. For any pair of cardinalities, an approximation of the mean can be obtained by cross-validation as detailed previously just by replacing c in (4) by the rank function $\mathsf{rank}(\cdot)$. This is exactly what has been done for the benchmarks discussed in Sect(s). 3 and 4. More precisely, the following attack has been repeated $t = 10$ times and the average rank of the correct key is plotted: (1) select a training set of fixed size N_{train} and (2) compute the evolution of the rank of the correct key when the model is tested with an increasing number n of traces in $\mathcal{D}_{\text{test}}$ (of size N_{test}).

A second metric which is commonly used in Machine Learning is the accuracy. With the same previous notations, we can define it as:

$$\mathsf{acc}(\hat{\mathbf{g}}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}}) = \frac{|\{(\vec{\ell}_i, p_i, k^\star) \in \mathcal{D}_{\text{test}} \mid k^\star = \text{argmax}_{k \in \mathcal{K}} \ \vec{y}_i[k]\}|}{|\mathcal{D}_{\text{test}}|} \ , \tag{7}$$

where we recall that $\vec{y}_i$ denotes the $|\mathcal{K}|$-dimensional output $\hat{\mathbf{g}}(\vec{\ell}_i, p_i)$. Then, similarly as for the rank function but for possibly unbounded size of $\mathcal{D}_{\text{test}}$, we can define from (7) an *Expected Accuracy of the model* (ACC) by:

$$\mathsf{ACC}_{N_{\text{train}}}(\hat{\mathbf{g}}) = \mathsf{E}[\mathsf{acc}(\hat{\mathbf{g}}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}})] \ ,$$

where the mean is defined over all the datasets $\mathcal{D}_{\text{train}}$ of size N_{train} and all the datasets $\mathcal{D}_{\text{test}}$ (with unbounded size).[7]

Finally our selection of parameters is also guided by the *computational time* of the model training. The mean of the training time is computed in the same manner as the other evaluation metrics during the 10-fold cross-validation.

2.4.3 About the Profiling Set-Up

Our implementations of Machine Learning algorithms have been developed with Keras library [C+15] (version 2.1.1) or directly with Tensorflow library [AAB+15] (version 1.4.0). We run the trainings over ordinary computers equipped with 16 GB of RAM and gamer market GPUs Nvidia GTX 1080 Ti. The computation of all the benchmarks took approximately 12 days by using 3 GPU cards.

[6] and also different values of $k^\star$ if this is relevant for the attacked algorithm.

[7] Another metric, the *prediction error* (PE), is sometimes used in combination with the accuracy: it is defined as the expected error of the model over the training sets; $\mathsf{PE}_{N_{\text{train}}}(\hat{\mathbf{g}}) = 1 - \mathsf{ACC}_{N_{\text{train}}}(\hat{\mathbf{g}})$.

2.5 Target of the Attacks Experiments and Leakage Characterization

For our attack experiments, we targeted a Software protected AES implementation running over an 8-bit `AVR` architecture. More precisely, the device is an `ATMega8515`.

2.5.1 About the Implementation

To maximize our control on the code executed by the device, we choose to implement the AES in assembly language. We developed two versions which merely aim at defeating first-order SCA attacks (*i.e.* attacks exploiting a single temporal leakage without (re-)combining of temporal points). The first one makes use of the classical *table re-computation* method (see *e.g.* [AG01, KJJ99, PR08] for a detailed description). The AES state is secured with 16 different masks for the linear parts and, for the SubBytes processing, the same pair of input and output masks is used for each state element. The outlines of the implementation are summed up in Algorithm 1. A Signal-to-Noise Ratio (SNR)[8] has been done to validate that there is no first-order leakage ($snr1$ in Figure 2). To help evaluators to perform elementary attacks against the first two bytes of the AES state during the first round, the corresponding masks of the linear parts ($r[1]$ and $r[2]$ in Algorithm 1) have been fixed to 0.

Attack experiments reported in the rest of the paper only target the output of the third sbox processing during the first round (namely $\text{sbox}^\star[\text{state0}[3]] \doteq \text{sbox}(p[3] \oplus k[3]) \oplus r_{\text{out}}$ with $i = 3$ in Algorithm 1).[9]

2.5.2 About the Acquisition Phase

The side-channel observations were obtained by measuring the electromagnetic (EM) radiations emitted by the device. To this aim, a sensor made of several coils of copper was plugged into a low-noise amplifier. To sample measurements, a digital oscilloscope was used with a sampling rate of 2G samples per second. We insist on the fact that the temporal acquisition window was set to record the first round of the AES only. As the MCU clock was quite stable, the resynchronization of the measurements was not difficult and resulted in a campaign of $100,000$ traces composed of $100,000$ time samples. Among them, we chose to finally extract only $60,000$ traces after validating that it was sufficient to accurately realize all our benchmarks (see *e.g.* Sect. 3.2). To identify the leakage samples related to the secure processing of $\text{sbox}(p[3] \oplus k[3])$, several SNRs have been processed:

Table 1: details of the SNR processings in Fig. 2

Name	Type	Definition of the target variable Z
$snr1$	unmasked sbox output	$\text{sbox}(p[3] \oplus k[3])$
$snr2$	masked sbox output	$\text{sbox}(p[3] \oplus k[3]) \oplus r_{\text{out}}$
$snr3$	common sbox output mask	r_{out}
$snr4$	masked sbox output in linear parts	$\text{sbox}(p[3] \oplus k[3]) \oplus r[3]$
$snr5$	sbox output mask in linear parts	$r[3]$

It may be observed in Figure 2 that $snr1$ (in gray) is very low, which essentially shows that there is no first-order leakage on the unmasked sbox output $\text{sbox}(p[3] \oplus k[3])$. The leakages on the sbox output masked with $r[3]$ and on the mask $r[3]$ itself are relatively high ($snr4$ and $snr5$ respectively). The SNR $snr4$ shows three peaks because the sbox output with mask $r[3]$ is not only manipulated during the SubBytes step but also during the ShiftRows and the MixColumns. Eventually, one can also observe a leakage on the

[8] The SNR is sometimes named F-Test to refer to its original introduction by Fischer [Fis22]. For a noisy observation L_t at time sample t of an event Z, it is defined as $\mathsf{Var}[\mathsf{E}[L_t \mid Z]]/\mathsf{E}[\mathsf{Var}[L_t \mid Z]]$.

[9] Another possibility would have been to target $\text{state0}[3] = \text{sbox}(p[3] \oplus k[3]) \oplus r[3]$ which is manipulated at the end of Step `8]` in Algorithm 1.

Algorithm 1: Secure AES Implementation with Table Recomputation

Input : a 16-byte plaintext $(p[1], \cdots, p[16])$,
an 18-byte mask vector $(r[1], \cdots, r[16], r_{\text{in}}, r_{\text{out}})$,
and a 16-byte master key $(mk_1, \cdots, mk_{16})$

Output : a 16-byte ciphertext $(c_1, \cdots, c_{16})$

steps] function **MaskedAES:**

// SBox recomputation

for $i = 0$ ***to*** 255 **do**

1] $\text{sbox}^\star[i] \leftarrow \text{sbox}[i \oplus r_{\text{in}}] \oplus r_{\text{out}}$

// Initialization

for $i = 1$ ***to*** 16 **do**

2] $\text{state0}[i] \leftarrow p[i] \oplus r[i]$

3] $\text{state1}[i] \leftarrow r[i]$

4] $\text{key}[i] \leftarrow mk_i$

// AES processing

for $round = 1$ ***to*** 10 **do**

/* Key scheduling */

5] $(\text{key}[1], \cdots, \text{key}[16]) \leftarrow \text{KeyScheduling}(\text{key}[1], \cdots, \text{key}[16])$

for $i = 1$ ***to*** 16 **do**

/* AddRoundKey and SubBytes */

6] $\text{state0}[i] \leftarrow (\text{state0}[i] \oplus \text{key}[i] \oplus r_{\text{in}}) \oplus \text{state1}[i]$

7] $\text{state0}[i] \leftarrow \text{sbox}^\star[\text{state0}[i]]$

8] $\text{state0}[i] \leftarrow (\text{state0}[i] \oplus \text{state1}[i]) \oplus r_{\text{out}}$

/* ShiftRows */

9] $(\text{state0}[1], \cdots, \text{state0}[16]) \leftarrow \text{ShiftRows}(\text{state0}[1], \cdots, \text{state0}[16])$

10] $(\text{state1}[1], \cdots, \text{state1}[16]) \leftarrow \text{ShiftRows}(\text{state1}[1], \cdots, \text{state1}[16])$

/* MixColumns except for the last round */

if $round \neq 10$ **then**

11] $(\text{state0}[1], \cdots, \text{state0}[16]) \leftarrow \text{MixColumns}(\text{state0}[1], \cdots, \text{state0}[16])$

12] $(\text{state1}[1], \cdots, \text{state1}[16]) \leftarrow \text{MixColumns}(\text{state1}[1], \cdots, \text{state1}[16])$

// Last AddRoundKey

13] $(\text{key}[1], \cdots, \text{key}[16]) \leftarrow \text{KeyScheduling}(\text{key}[1], \cdots, \text{key}[16])$

for $i = 1$ ***to*** 16 **do**

14] $c_i \leftarrow (\text{state0}[i] \oplus \text{key}[i]) \oplus \text{state1}[i]$

// Return the ciphertext

return $(c_1, \ldots, c_{16})$

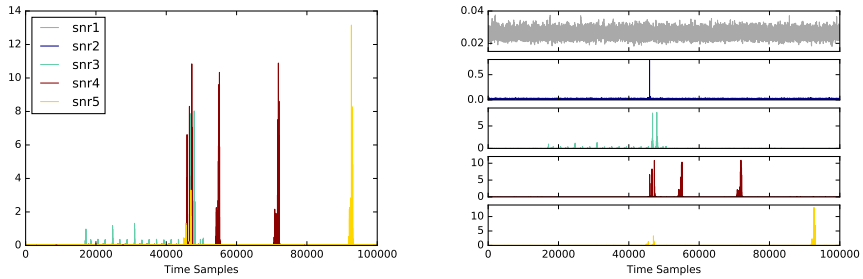

Figure 2: SNRs for various intermediate values related to the processing of $\text{sbox}(p[3] \oplus k[3])$.

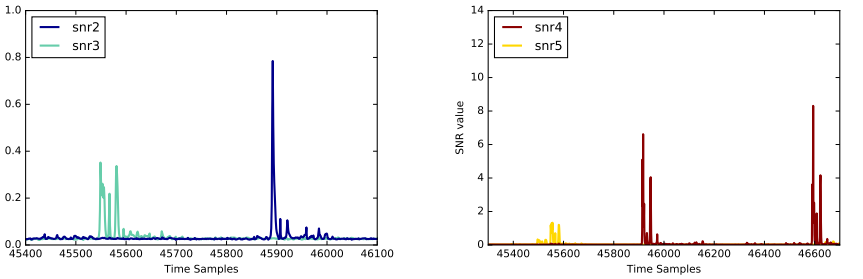

Figure 3: SNRs for various intermediate values related to the processing of $\text{sbox}(p[3] \oplus k[3])$ in the interval [45400..46100].

sbox output masked with r_{out} and on the mask r_{out} itself ($snr2$ and $snr3$). Since this leakage is smaller than for the sbox with mask $r[3]$ we found it more challenging and preferred to focus on it in our attack experiments.

For the reasons explained in previous paragraph, we chose to enshorten the initial traces (composed of $100,000$ samples) and to only keep, for each trace, the 700 samples in the interval [45400..46100] which contains information on the two pairs of $\text{sbox}(p[3] \oplus k[3]) \oplus r[3], r[3])$ and $(\text{sbox}(p[3] \oplus k[3]) \oplus r_{\text{out}}, r_{\text{out}}))$ (see[10] Fig. 3).

2.6 Design and Parameters of the new ASCAD Database

2.6.1 Trace Format

For the storage of the observations and the metadata (plaintext/ciphertext/key/mask values), we chose to use the current version 5 of the *Hierarchical Data Format* (HDF5). The latter one is a multi-purpose hierarchical container format capable of storing large numerical datasets with their meta-data. The specification is open and the tools are open source. The development of HDF5 is done by the HDF Group, a non-profit corporation [Groa]. A HDF5 file contains a POSIX-like hierarchy of numerical arrays (aka datasets) organized within groups and subgroups. Effectively, HDF5 may be seen as a file system within a file, where files are datasets and folders are groups. Moreover, HDF5 also supports lossless compression of datasets. To manipulate our HDF5 files we used the `h5py` python package [Grob].

Our HDF5 file `ATMega8515_raw_traces.h5` is composed of two datasets within two groups: `metadata` and `traces`. The type of the latter one is HDF5 `Scalar Dataset` (*i.e.* may be viewed as a 2-dimensional array of 8-bit integers, the first dimension being the

[10]Note that some peaks appearing in Fig. 2 have not been selected.

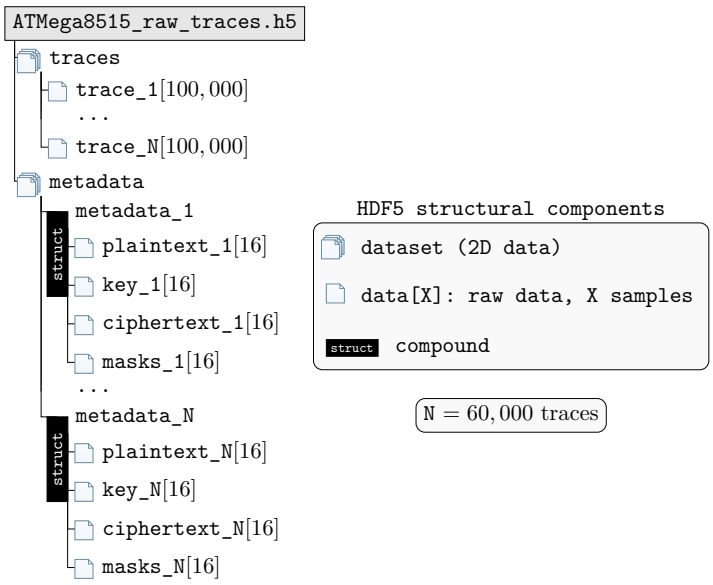

Figure 4: Structure of the `ATMega8515_raw_traces.h5` HDF5 data file

observation index, the second dimension being a time index and 8-bit integer being the type of the measure). The type of `metadata` is HDF5 `Compound Dataset` which is similar to a `struct` in `C` language. The `members` of the compound dataset `metadata` are *plaintext*, *ciphertext*, *key* and *mask* which all are arrays of 16 unsigned 8-bit integers. The 14 first elements of the *mask* array correspond to the masks $r[3], \cdots, r[16]$ in Algorithm 1 and the two last elements respectively correspond to r_{in} and r_{out} (as explained before the masks $r[1]$ and $r[2]$ have been forced to 0 for test/validation purpose). We give an overview of this structure on Fig. 4.

2.6.2 MNIST database and adaptations to SCA

Our raw traces format, as described in the previous subsection, is a classical representation of data for SCA analysis. This however suffers from some issues when considering it in the light of ML analysis:

- When considering a classification problem, one wants to get explicit and distinct classes where each trace is sorted (i.e. labelled) to help with the profiling phase.
- From the traces in `ATMega8515_raw_traces.h5`, it is not clear which dataset is to be used for training, and which is to be used for the tests and the accuracy computation.
- Finally, the raw `ATMega8515_raw_traces.h5` file does not contain explicit *labels* for the SCA classification problem, though these can be computed given the *plaintext* and *key* metadata.

The **MNIST database** [LCB] is a reference in the ML image classification community, allowing any new Machine Learning algorithm to be fairly compared to the state-of-the-art results. The efficiency of a new algorithm is tested against the classification of 28×28 pixels grayscale, normalized and centered images of handwritten digits. The database is split in groups, each one containing data and labels:

1. The *training dataset* group ($50,000$ samples) contains the samples used during the training phase. This group is composed of the raw images in a file, and their labels with the same index in another file.

2. Similarly, the *test dataset* group (10,000 samples) is composed of the raw images in a file, and their labels with the same index in another file.

Following the path of the MNIST database, we propose a novel approach that fits the needs of testing ML algorithms against the SCA classification problems described in previous sections. We provide a database ASCAD with labelled datasets that will allow the *SCA community to objectively compare the efficiency of ML and DL methods*. To fit the SCA context, we have adapted the so-called MNIST training and test concepts to the more appropriate *profiling* and *attack* semantics as introduced and described in subsection 2.2.1.[11]

The database information is extracted from the raw `ATMega8515_raw_traces.h5` data file, and its structure is presented on Fig. 5. For the sake of efficiency and simplicity, the HDF5 file format has been kept for our ASCAD database. The new file `ASCAD.h5` is composed of:

- two main groups: one for profiling (`Profiling_traces`) which contains N_p information, and one for attacking (`Attack_traces`) which contains N_a information.[12] In our case, over the 60,000 labelled traces, we have chosen $N_p = 50,000$ and $N_a = 10,000$.
- In each main group, we find three HDF5 datasets;
  - the `traces` dataset contains the raw traces zoomed in on the 700 samples window of interest, namely the [45400..46100] interval containing the relevant information as previously described (only keeping the relevant samples in the traces allows to have a reasonably sized database),
  - the `labels` dataset contains the labels (following the ML classification meaning) for each trace. In our case, the value of the byte $\text{sbox}(p[3] \oplus k[3])$ is the label of interest, leading to 256 possible classes (the sequel of the article discusses this choice, and compares it to other possible classes such as using the Hamming weight of $\text{sbox}(p[3] \oplus k[3])$). In Remark 2, we explain how this labelling over the outputs of the sbox processing can be simply converted into a labelling over the different key candidates. It is to be noted that the masks *are not used* when computing the labels.
  - The `metadata` dataset contains the information related to each trace in a HDF5 compound (aka structure), taken from `ATMega8515_raw_traces.h5` almost without any modification (an additional field is added, see below). From a strict ML perspective, this metadata is useless since the labels are the only necessary information to check the efficiency of an algorithm. These data are however useful from a SCA perspective since the *plaintext* byte $p[3]$ is necessary to extract the estimated $\hat{k}[3]$ from the label values, and the real value of the key byte $k[3]$ is useful for the key ranking with regard to each class probability. Even though only $p[3]$ and $k[3]$ are useful for key ranking, we have decided to keep all the other metadata (the other *plaintext* and *key* bytes, the *ciphertext* and the *masks*) for the sake of completeness: the size of this metadata is very reasonable. Finally, a *desync* field is added to the compound structure: this `uint32` field represents the optional random desynchronization applied to the traces, which simulates a jitter as explained hereafter.

We feel that our ASCAD database is versatile enough to check the efficiency and accuracy of ML and DL algorithms applied to side-channel analysis, and we also aim at providing general purpose `python` scripts that will ease the process of:

[11] Additionally, beware that in this paper training and testing are used in the context of cross-validation and are subsets of the profiling dataset $\mathcal{D}_{\text{profiling}}$.

[12] We recommend to perform the cross-validation only with the profiling set.

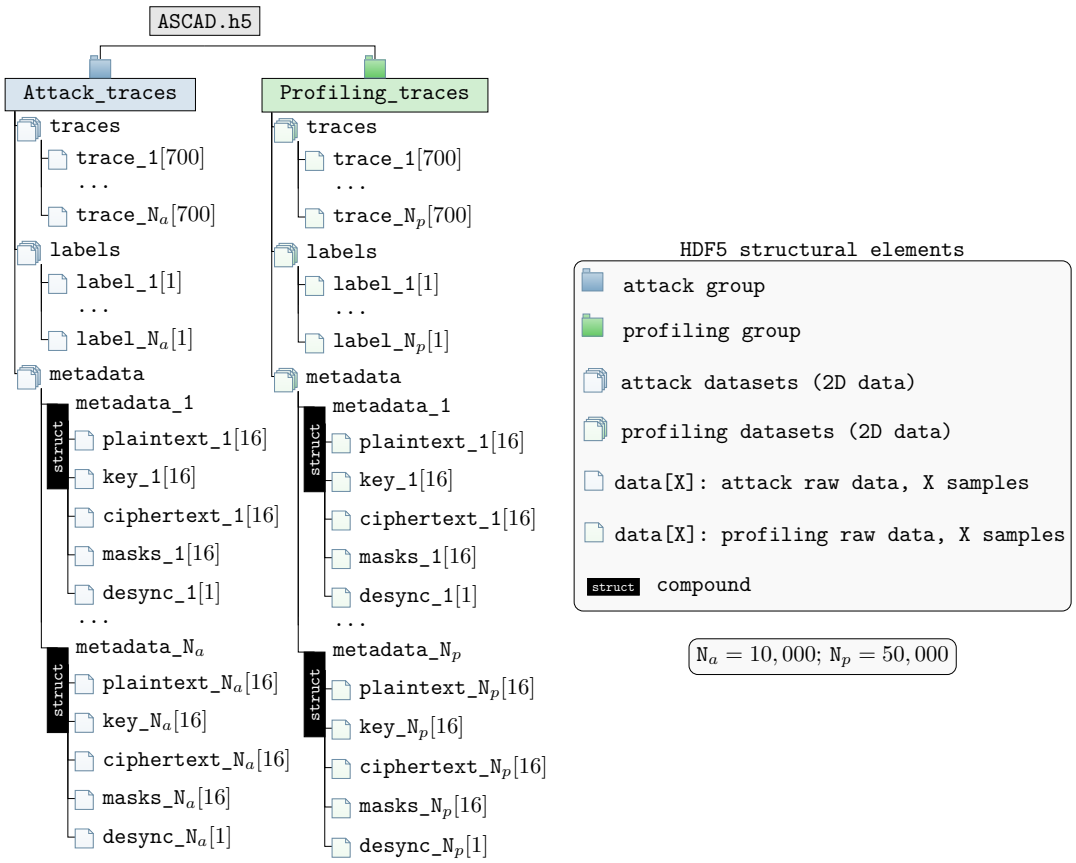

Figure 5: Structure of the `ASCAD.h5` HDF5 data file

- creating new databases following the same structure to attack other outputs in other AES rounds (with data extracted from `ATMega8515_raw_traces.h5` or any other similarly structured HDF5 file),
- modifying the profiling and attack datasets sizes and index to check their effect,
- adding a parametrized desynchronization to the traces to check the efficiency of the algorithm against jitter, and its impacts on the hyper-parameters. See the sequel of the article for a discussion on this.

As a benchmarking baseline, we will actually provide three HDF5 files that form our reference database:

- `ASCAD.h5`, which contains profiling and attack datasets as previously described. The traces are synchronized and there is no jitter,
- `ASCAD_desync50.h5`, which contains traces with a 50 samples window maximum jitter.
- `ASCAD_desync100.h5`, which contains traces with a 100 samples window maximum jitter.

The method used to simulate the jitter is described in 4.2.3.

3 Multi-Layer Perceptrons (MLP)

3.1 Core Principles and Constructions

Multi-Layer Perceptrons (MLPs) are associated with a model/function $\hat{\mathbf{g}}$ that is composed of multiple linear functions and some non-linear *activation functions* which are efficiently-computable and whose derivatives are bounded and efficient to evaluate. In short, an MLP can be defined as follows:

$$\hat{\mathbf{g}} : \vec{\ell} \mapsto \hat{\mathbf{g}}(\vec{\ell}) = \mathsf{s} \circ \lambda_n \circ \alpha_{n-1} \circ \lambda_{n-1} \circ \cdots \circ \lambda_1(\vec{\ell}) = \vec{y} \,, \tag{8}$$

where:

- the λ_i functions are the so-called *Fully-Connected* (FC) layers and are expressible as affine functions: denoting $\vec{v}$ the D-dimensional input of an FC, its output is given by $\mathbf{A}\vec{\ell} + \vec{B}$, being $\mathbf{A} \in \mathbb{R}^{C \times D}$ a matrix of weights and $\vec{B} \in \mathbb{R}^C$ a vector of biases. These weights and biases are the trainable weights of the FC layer,[13]
- the α_i are the so-called *activation functions* (ACT): an activation function is a non-linear real function that is applied independently to each coordinate of its input (*e.g.* the ReLU activation function processes $\max(0, x)$ to each coordinate x),
- s is the so-called *softmax*[14] function (SOFT): $\mathsf{s}(\vec{\ell})[i] = \frac{e^{\vec{\ell}[i]}}{\sum_j e^{\vec{\ell}[j]}}$.

In the rest of the paper, $\texttt{MLP}(n_{\text{layer}}, n_{\text{units}}, act)$ will denote an MLP architecture with n_{layer} layers, n_{units} units (a.k.a. nodes or neurons) and act as activation function for each hidden layer. Such an MLP corresponds to (8) with $\alpha_i = act$ for every i, with $n = n_{\text{layer}}$, and with λ_i defined for $D = C = n_{\text{units}}$ if $i \in [2...n_{\text{layer}} - 1]$ and for $(C, D) = (n_{\text{units}}, 700)$ if $i = 1$ and $(C, D) = (256, n_{\text{units}})$ for $i = n_{\text{layer}}$ (indeed inputs of the model are 700-dimensional leakage traces while the outputs are in $[0..255]$).

3.2 Choice of the Hyper-parameters

As explained in Sect. 2.3, our strategy for tuning the hyper-parameters is divided into two steps. Starting from a base architecture that we denote MLP_{base}, we first tune the training parameters, leading to the parametrization of a training procedure $\texttt{Training}(n_{\text{epochs}}, batch\_size, optimizer, learning\_rate)$. Then, different variations of MLP_{base} are tested by studying the impact of each architecture parameter on the model efficiency after training with the procedure fixed during previous step. The full strategy aims at providing us with an architecture MLP_{best} and a training procedure that are good w.r.t. the evaluation metrics listed in Sect. 2.4.2.

3.2.1 Training Parameters

This subsection aims at studying how the mean rank of the side-channel attack involving the trained model is impacted by the length of the training dataset, the number of epochs, the batch size and the learning rates/optimizers. MLP_{base} denotes the 6-layers MLP with 200 units and the ReLU activation function for each layer, i.e. MLP_{base}=$\texttt{MLP}(6, 200, ReLU)$.

[13] They are called *Fully-Connected* because each i-th input coordinate is *connected* to each j-th output via the $\mathbf{A}[i, j]$ weight. FC layers can be seen as a special case of the linear layers where not all the connections are necessarily present. The absence of some (i, j) connections can be formalized as a constraint for the matrix $\mathbf{A}$ consisting in forcing to 0 its (i, j)-th coordinates.

[14] To prevent underflow, the log-softmax is usually preferred if several classification outputs must be combined.

First we evaluate the impact of the size N_{train} of the training set on the success of a neural network based SCA. We performed a 10-fold cross validation with different sizes of dataset, while keeping a constant computational time during the training step for fair comparison. This is done by adapting the number of epochs to the number of traces in the dataset. We expect that the performance of the model increases with the size of the training set until a certain threshold that determined the optimal number of traces. The neural network used for this experiment is MLP_{base} trained with RMSProp optimizer, learning rate 10^{-5} and batch size 100. The initialization of the weights is performed from an uniform distribution of mean 0 as defined in Glorot and Bengio's article [GB10]. Fig. 6 shows the mean rank function for different sizes of training set. Our empirical results on the full `ATMega8515_raw_traces.h5` show that approximately $50,000$ training traces are required for a full success of the attack/test in less than $1,000$ traces. That is why ASCAD is composed of a training set $\mathcal{D}_{\text{profiling}}$ of size $50,000$ and an attack set $\mathcal{D}_{\text{attack}}$ of size $10,000$. Based on these results, the benchmarks in the rest of the paper were performed on ASCAD profiling traces $\mathcal{D}_{\text{profiling}}$.

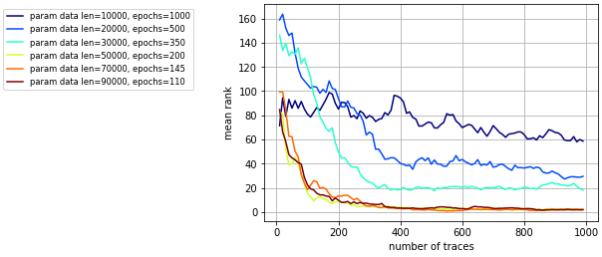

Figure 6: Mean rank function (6) after 10-fold cross-validation of MLP_{base} with $\texttt{Training}(\cdot, 100, \text{RMSProp}, 10^{-5})$ for different sizes of training set, for an increasing number n of test traces and for different epochs chosen to keep the overall computation time roughly constant.

Then we select the best values for the number of epochs and the batch size of the training step. Fig. 7 shows the empirical results for different values n_{epochs} with a 10-fold cross-validation on $\mathcal{D}_{\text{profiling}}$. We notice that the number of epochs has a significant impact on the rank functions. Taking into account the trade-off between computation time and SCA-efficiency, best results are obtained by choosing 400 epochs and a batch size equal to 500 or 200 epochs and a batch size equal to 100. However, it appears that we have a best accuracy and a best stability on the rank functions with the latter pair of parameters, which leads us to select these values for the rest of our benchmarks on MLP. We insist on the fact that theses values are obtained as a trade-off that allows us to perform multiple cross-validations in a reasonable amount of time. When the batch size parameter is fixed to 100 we can obtained better results by increasing the number of epochs and consequently the training time. Therefore in the case of a single SCA attack in a given amount of time, we recommend to fix the batch size to 100 and to increase progressively the number of epochs after 200 until the dedicated amount of time for the training step is reached (in our experimental results we did not notice any improvement in the SCA-efficiency after 800 epochs).

The last training parameters that we tune are the gradient descent optimization method (also called optimizer) and the learning rate. Empirical results in Fig(s). 8 and 9 show that these parameters also have a high impact on the success of the attack. We managed to obtain good results with $optimizer = RMSProp$ and a learning rate equal to 10^{-5} (which confirms the soundness of the choices made for experiments reported in Fig(s). 6 and 7).

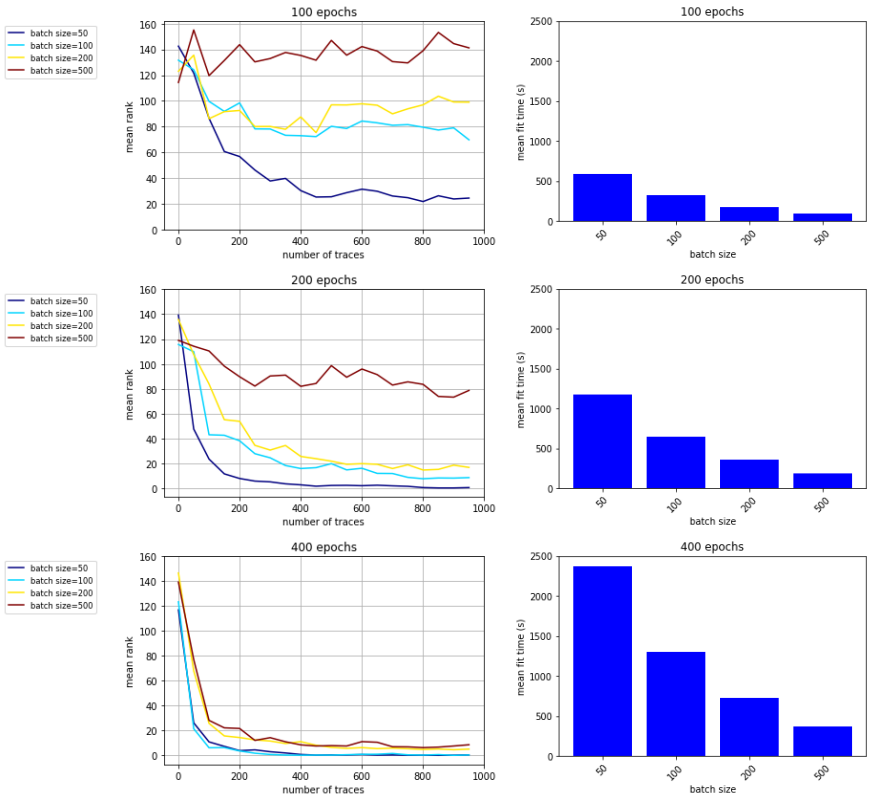

Figure 7: Mean ranks and training time of MLP_{base}=`MLP`$(6, 200, relu)$ trained with `Training`$(n_{\text{epochs}}, batch\_size, \text{RMSProp}, 10^{-5})$ for varying values of n_{epochs} and $batch\_size$.

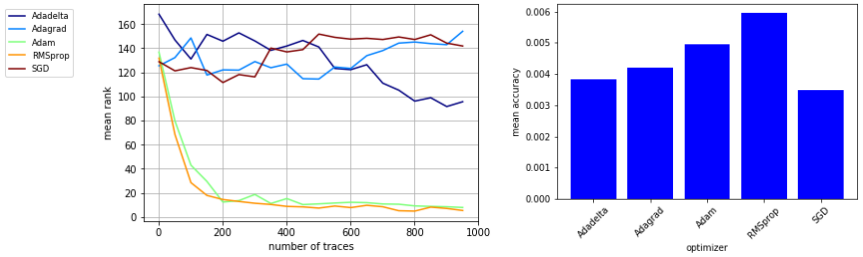

Figure 8: Mean ranks and accuracy of MLP_{base}=`MLP`$(6, 200, ReLU)$ trained with `Training`$(200, 100, optimizer, 10^{-5})$ for different optimizers.

3.2.2 Architecture Parameters

As described in previous subsection, an MLP architecture is characterized by three architecture hyper-parameters: the number of layers, the number of units of each layer and the activation functions. In this section, we use the training procedure `Training`$(200, 100, RMSProp, 10^{-5})$ determined in previous section and we come back on our initial choice of MLP_{base} to challenge its hyper-parameters.

First we evaluate the optimal number of layers with a fixed number of nodes, namely

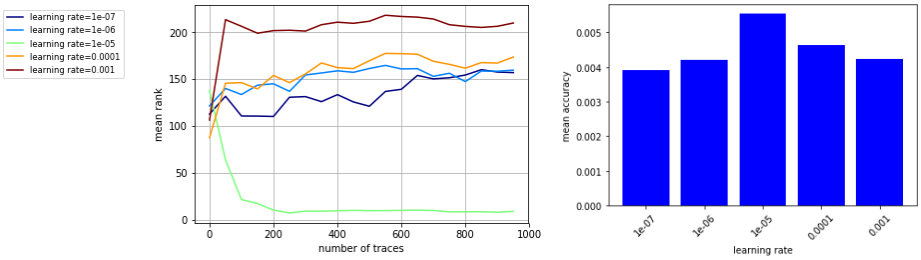

Figure 9: Mean ranks and accuracy of MLP_{base}=`MLP`$(6, 200, ReLU)$ with RMSProp optimizer and different values of learning rate.

we train models `MLP`$(n_{\text{layers}}, 200, ReLU)$ for different values $n_{\text{layers}} \in [3..11]$. Fig. 10 plots the mean rank function, the mean accuracy and the average training time. All the mean rank functions converge to 0 when the number of traces increases. However, the 6-layers MLP has a slight advantage on less than 600 traces and has the best mean accuracy.

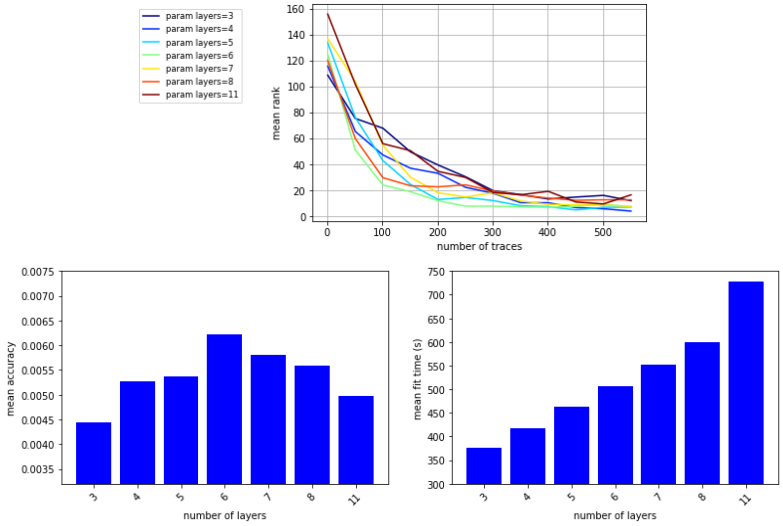

Figure 10: Mean ranks, mean accuracy and mean training time of `MLP`$(n_{\text{layers}}, 200, ReLU)$ with different numbers of n_{layers}.

Then we evaluate the optimal number of units per layer. Small values lead to simple models that are not powerful enough to represent the dataset trends and high values lead to complex models that are difficult to train and are more susceptible to over-fitting. We limit our empirical study to MLPs with the same number of units by layer. Fig. 11 shows the obtained results. With the previously fixed training parameters, the performance of the attack seems to increase once the number of units per layer equals or exceeds 200.

Finally we study the effect of the activation function on the performance of the neural network. Since its introduction in Deep Learning, Rectified Linear Units (ReLUs) have proved to be the best suitable choice for a number of problems, and most specifically in image recognition [JKL+09, NH10, GBB11]. The obtained networks have sparse representation, and the simple definition of the activation function $ReLU(x) = \max(0, x)$ allows quick computations. Fig. 12 plots the experimental results obtained with `MLP`$(6, 200, activation\_function)$ for different activation functions. The best results are obtained with $ReLU$, $tanh$ and $softsign$ which is a variation of $tanh$. We select $ReLU$

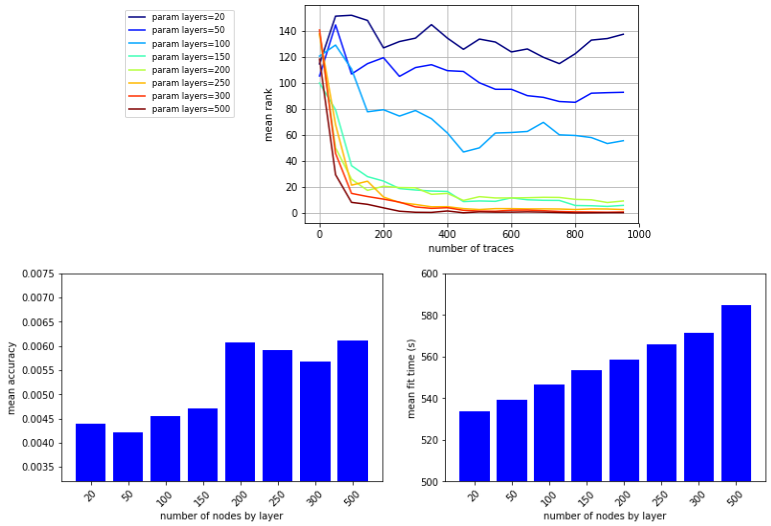

Figure 11: Mean ranks, mean accuracy and mean training time of $\texttt{MLP}(6, n_{\text{units}}, ReLU)$ with different n_{units}.

activation function since it provides state-of-the-art results and its computation time is below the two other functions.

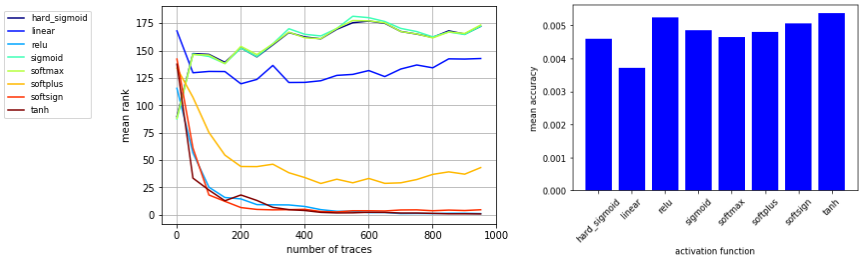

Figure 12: Mean ranks of $\texttt{MLP}(6, 200, activation\_function)$ with different activation functions.

Benchmarks reported in this section confirms that the architecture $\texttt{MLP}(6, 200, ReLU)$ leads to good compromise efficiency *versus* computational time when trained with the procedure $\texttt{Training}(200, 100, RMSProp, 10^{-5})$. In the rest of this paper, this architecture is denoted MLP_{best}. We insist on the fact that MLP_{best} has a decent SCA-efficiency with 200 epochs but the latter efficiency continues to improve when the number of epochs increases until 800 epochs (in our experiments we did not notice any significant improvement after 800 epochs). Hence, depending on the amount of time allocated to the training of MLP_{best}, it may be interesting to increase the number of epochs in the range [200..800].

3.3 Open Discussions

3.3.1 Self-Normalizing Neural Networks

Recently, a new type of MLP called Self-Normalizing Neural Networks (SNN) has been introduced in [KUMH17]. It aims to improve the robustness of MLPs against perturbation during the training step and to reduce the variance of the training error. Its architecture is a slight variation of the standard MLP architecture: the activation function, called "scaled

exponential linear units" (SELU) is given by:

$$selu(x) \;=\; \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases} . \tag{9}$$

Furthermore, the initialization of the weights is performed from a standard normal distribution. These two modifications imply that the neural network is self-normalizing, i.e. the mean and variance of the activation functions across the different layers stay within small predefined intervals. This new architecture outperformed standard MLPs on a number of benchmarks, including MNIST.

We test on the ASCAD a SNN architecture with 6 layers and 200 units for each layer and we compare it with MLP_{best}. Experimental results in Fig. 13 show that rank functions are very similar between the two architectures. This highlights the fact that there does not seem to be any significant improvement with the SNN architecture in the context of SCA. The accuracy is slightly higher with SNN as expected in a Machine Learning perspective, however it does not have an influence on the overall rank function.

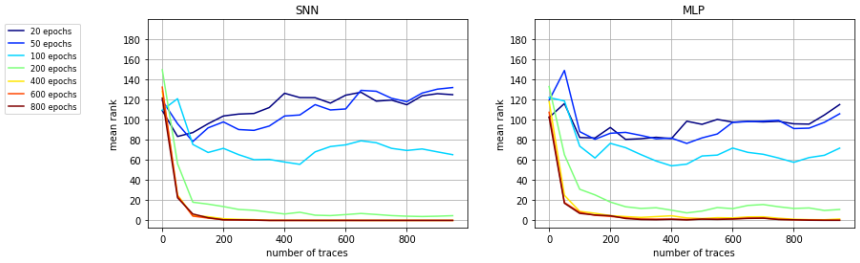

Figure 13: Mean ranks and accuracy of a SNN and MLP_{best} with different numbers of epochs.

3.3.2 Hamming weight vs identity labelling

We test our MLP_{best} architecture on the SCA dataset with a labelling of the traces modified to take the Hamming weight of the sensitive value instead of the real value itself. This strategy of data labelling reduces the number of classes to predict (9 values for the Hamming weight instead of 256 values for a byte). Consequently, the model trained on the new dataset is less complex than the model trained on the full values. We also modify the computation of the rank function in (2) by taking into account the distribution of the Hamming weight values. In Fig. 14, the corresponding rank functions are plotted. They show that the new labelling strategy is less interesting. Indeed, even if the Hamming weight model is less complex and requires a smaller number of epochs for the training step, the conditional probability approximated by the neural network is less discriminating (which is a consequence of the reduced number of classes). Moreover, the weighting coefficients in (2) (deduced from the Hamming weight distribution for uniform data) may increase the variance of the rank (viewed as a random variable) since *e.g.* an error on a value with Hamming weight 0 or 8 accounts for $\binom{8}{4} = 70$ times an error on a value of Hamming weight 4. Eventually, assuming that the deterministic part of the leakage corresponds to an Hamming weight may be an incorrect abstraction and induces error in the modelling.

3.3.3 Comparison with Template Attacks

We compare MLP_{best} with standard Template Attacks (aka Quadratic Discriminant Analysis, or QDA in the Machine Learning community). We first perform an unsupervised

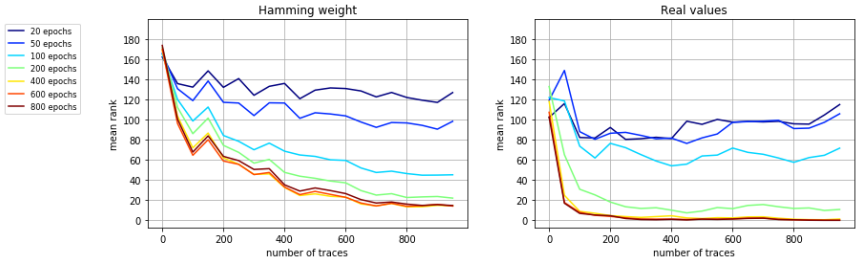

Figure 14: Mean ranks of MLP_{best} with Hamming weights as labels and MLP_{best} with real values as labels.

dimension reduction to extract meaningful features. For this task we use a classical PCA which is parametrized by the number of components to extract. Then the classification task is performed with a QDA (*i.e.* Template Attacks). Note that, contrary to QDA, neural networks do not require the preprocessing feature extraction step since this task is realized by the first layers of the networks. Figure 15 shows the results obtained with different numbers of components extracted from the PCA.

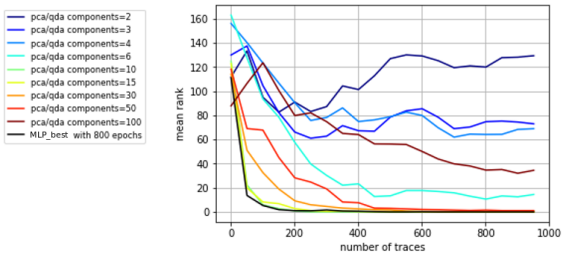

Figure 15: Mean ranks of a PCA on n components followed by a QDA.

3.3.4 First order attacks

By using the mask values contained in the ASCAD, it is possible to compute the masked output after the first round AES Sbox:

$$z = \text{sbox}(p[3] \oplus k[3]) \oplus r_{\text{out}}$$

where z is the sensitive value and $p[3], k[3], r_{\text{out}}$ are the plaintext byte, the key byte and the mask byte.

Therefore we can mount a first order SCA by labelling the traces with the masked output values and we can test the performance of MLP_{best} in this weaker context. The results in Fig. 16 show that, without any modification in the architecture and the training parameters, MLP_{best} easily succeeds in this attack. The rank functions converge to 0 with 20 epochs and only 4 traces are required to determine the correct key. We also managed to obtain an accuracy of 0.028, and we did not notice any overfitting with 200 epochs.

4 Convolutional Neural Networks (CNN)

4.1 Core Principles and Constructions

Convolutional Neural Networks (CNNs) complete the classical principle of MLP with a so-called *convolutional* layer based on a convolutional filtering, and a *pooling* layer. We

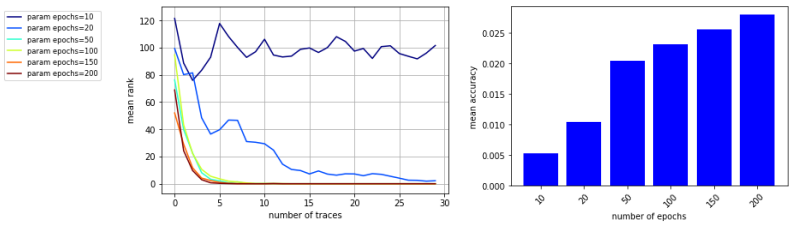

Figure 16: Mean ranks and accuracy of MLP_{best} on a first order SCA.

describe them hereafter, together with a third one called *batch-normalization* layer which has been recently introduced to improve the effectiveness of the model.

4.1.1 Convolutional (CONV) layers

CONV layers are linear layers that share weights across space. The representation is given in Fig. 17.[15] To apply a convolutional layer to an input trace, n_{filter} small column vectors, called *convolutional filters*, of size W (aka *kernel size*) are slid over the trace by some amount of units, called *stride*.[16] The column vectors form a window[17] which define a linear transformation of the W consecutive points of the data into a new vector $\vec{v}$. When the window slides over the last points, the input trace can be either padded with 0 resulting in a vector $\vec{v}$ which has the same number of points than the input data (*same* padding) or the data is not padded and the window only slides over the valid part of the data, resulting in a vector $\vec{v}$ smaller than the input trace (*valid* padding). The coordinates of the window (viewed as a matrix) are among the trainable weights which are constrained to be unchanged for every input. This constraint is the main difference between a CONV layer and an FC layer; it allows the former to learn shift-invariant features. The reason why several filters are applied is that we expect each filter to extract a different kind of characteristic from the input. As one goes along convolutional layers, higher-level abstraction features are expected to be extracted. These high-level features are arranged side-by-side over an additional data dimension, the so-called *depth*.[18] This is this geometric characteristic that makes CNNs robust to temporal deformations [LB+95].

To avoid complexity explosion due to this depth increasing, the insertion of pooling layers is recommended.

4.1.2 Pooling (POOL) layers

POOL layers are non-linear layers that reduce the spatial size in order to limit the amount of neurons, and by consequence the complexity of the minimization problem (see Fig. 17). As the CONV layers, they make some filters slide across the input. Filters are 1-dimensional, characterized by a length W and a stride, that is usually chosen equal to W; for example in Fig.17 both the length and the stride equal 3, so that the selected segments of the input do not overlap. In contrast with convolutional layers, the pooling filters do not contain trainable weights. They only slide across the input to select a segment, then a pooling function is applied: the most common pooling functions are the *max pooling* which

[15] CNNs have been introduced for images [LB+95]. So, usually, layer interfaces are arranged in a 3D-fashion (height, weight and depth). In Fig.17 we show a 2D-CNN (length and depth) adapted to 1D-data as side-channel traces are.

[16] Amount of units by which a filter shifts across the trace.

[17] *patches* in the Machine Learning language

[18] Ambiguity: Neural networks with many layers are sometimes called *Deep Neural Networks*, where the *depth* corresponds to the number of layers.

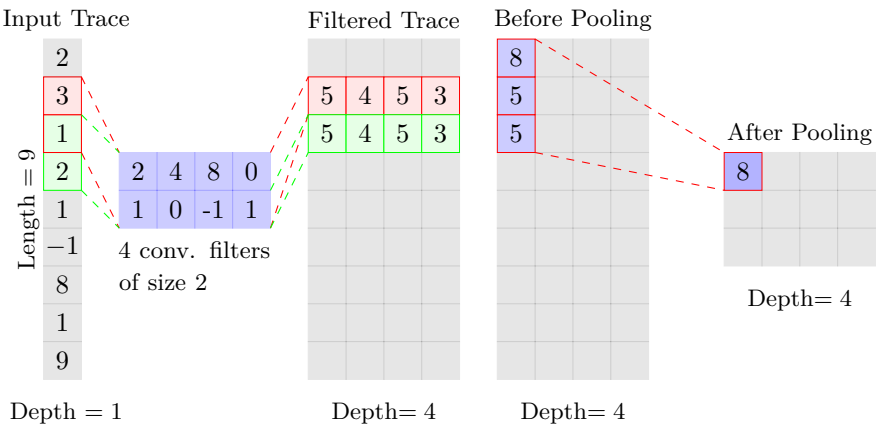

Figure 17: Convolutional filtering: $W = 2$, $n_{\text{filter}} = 4$, stride $= 1$, padding $=$ same. Max pooling layer: $W =$ stride $= 3$.

outputs the maximum values within the segment and the *average pooling* which outputs the average of the coordinates of the segment.

The two previous layer types are today often completed with a so-called *Batch Normalization* (BN) layer.

4.1.3 Batch Normalization layer

Batch Normalization has been introduced in [IS15] by Ioffe Szegedy to reduce the so-called *internal covariate shift* in neural networks and to eventually allow for the usage of higher learning rates. Covariate shift refers to the variation of the input distribution of a learning system. If this change happens on the input of internal units of (deep) neural networks, it is called an internal covariate shift. The reasoning behind the soundness of this layer is well argued in [GBC16b]. Let us denote by $\mathbf{H}$ a batch of inputs of an internal activation layer of the model (this means that $\mathbf{H}$ is composed of the outputs of several, let's say m, trainings which are processed in parallel). Assuming that $\mathbf{H}$ is arranged as a design matrix, meaning that the ith row $\vec{H}_i$ of $\mathbf{H}$ corresponds to the input for the ith training, the BN layer will replace $\vec{H}_i$ by a (normalized) new batch $\vec{H'}_i$ such that:

$$\vec{H'}_i = \frac{\vec{H}_i - \vec{\mu}}{\vec{\sigma}} \ , \tag{10}$$

where $\vec{\mu}$ and σ are respectively defined as:

$$\vec{\mu} = \frac{1}{m}\sum_i \vec{H}_i \tag{11}$$

and

$$\vec{\sigma} = \sqrt{\vec{\delta} + \frac{1}{m}\sum_i (\vec{H}_i - \vec{\mu})^2} \ , \tag{12}$$

where the processing of the division, the square and the square root is done element-wise and where $\vec{\delta}$ is a vector only composed of a same small positive value such as 10^{-8}, imposed to avoid encountering the undefined gradient of $\sqrt{z}$ at $z = 0$.

During the test/matching phase, statistics $\vec{\mu}$ and $\vec{\sigma}$ are not deduced from the batch elements but defined as the averages that were collected during the training phase (*e.g.* if the training has been done over let's say 100 batches, then $\vec{\mu}$ and $\vec{\sigma}$ correspond to the average over the 100 processings of (11) and (12)).

4.1.4 Common architecture

The main block of a CNN is a CONV layer γ directly followed by an ACT layer α. The former locally extracts information from the input thanks to filters and the latter increases the complexity of the learned classification function thanks to its non-linearity. After some $(\alpha \circ \gamma)$ blocks, a POOL layer δ is usually added to reduce the number of neurons: $\delta \circ [\alpha \circ \gamma]^{n_2}$. This new block is repeated in the neural network until obtaining an output of reasonable size. Then, some FC are introduced in order to obtain a global result which depends on the entire input. To sum-up, a common convolutional network can be characterized by the following formula:[19]

$$\mathrm{s} \circ [\lambda]^{n_1} \circ [\delta \circ [\alpha \circ \gamma]^{n_2}]^{n_3} \quad , \tag{13}$$

where we recall that s and λ respectively denote a softmax layer and a fully-connected layer.

4.1.5 A Brief Overview of Current CNN architectures

The first successful CNN network, best known as LeNet, was developed in the nineties and was mostly applied to handwritten digit recognition [LBD+89, LBBH98]. The last version of the network, LeNet-5 [LBBH98], is a small architecture which operates on images of 32×32 pixels split into 10 classes. The architecture is comprised of 2 convolutional layers with respectively 6 and 16 filters of size 5×5, and 3 final dense layers of respectively 120, 84 and 10 units. Each convolutional layer is followed by an average pooling layer and the activation function is the hyperbolic tangent. The network achieved an accuracy of nearly 99% on the test dataset.

CNN networks gained popularity with their breakthrough as a contender in the Imagenet Large Scale Visual Recognition Challenge (ILSVRC, [RDS+15]) and since 2012, deep CNN networks have constantly established new records in computer vision [KSH12, SZ14, SLJ+15, HZRS16]. ILSVRC is an image classification challenge which provides each year a labelled dataset of roughly $1,000,000$ images of 200×200 pixels split into $1,000$ classes. Candidates train and validate their algorithm on the provided dataset and submit it to the competition. Then algorithms are evaluated with an (unknown) test dataset and they are ranked according to two metrics, the top-1 accuracy and the top-5 accuracy, where the top-5 accuracy is the fraction of the test dataset for which the correct label is amongst the best 5 predictions returned by the algorithm.

The first CNN architecture presented at ILSVRC challenge in 2012 [KSH12] obtained a great success in the competition by outperforming all the challengers with a top-5 accuracy rate of 84.7% (against 73.8% for the second-best entry). This CNN network, well-known as AlexNet, has 8 layers, with 5 convolutional layers dispatched in 3 blocks and 3 dense layers of $4,096$ units each. The convolutional layers have $3, 96, 256$ and 384 filters of size 11×11, 5×5, and 3×3. Each block has a final max pooling layer and ReLU activation functions are used instead of hyperbolic tangents (as in LeNet). The subsequent winner of ILSVRC challenge, ZFNet [ZF14], improved the previous architecture by reducing the size of the first convolutional layer to 7×7 and by increasing the number of filters to $1,024$ for the last convolutional layers; it achieved a top-5 accuracy of 85.2%.

The trend of reducing the size of the filters by increasing the depth of the network was later confirmed to be a successful strategy. The runner-up architecture of the ILSVRC 2014 challenge, VGGNet [SZ14], obtained a 92.7% top-5 accuracy with an architecture comprised of (up to) 16 convolutional layers of 512 filters of size 3×3 distributed in 5 blocks (for the VGG-19 version).

The winner of ILSVRC 2014, GoogLeNet [SLJ+15], also used a deep network architecture with a total of 27 layers, but managed to decrease the number of parameters to

[19] where each layer of the same type appearing in the composition is not to be intended as exactly the same function (e.g. with same input/output dimensions), but as a function of the same form.

train by using a new element in the architecture, the Inception module, which is a stack of small size convolutional layers (1×1, 3×3, 5×5) with few parameters. Furthermore, the last dense layers are replaced by an average pooling layer, which also decreases the number of parameters to train. GoogLeNet obtained a top-5 accuracy rate of 93.3% with 12 times fewer parameters than AlexNet.

Finally, deep residual networks recently grow in popularity with the success of ResNet at ILSVRC 2015 [HZRS16]. These architectures manage to overcome the degradation problem that occurs when the depth of the network increases. They rely on residual units that learn for each layer the residual function $\mathcal{F}(x) = \mathcal{H}(x) - x$ where x is the input of the layer and $\mathcal{H}(x)$ is the desired function of the layer. The top of the networks also have an average pooling layer like GoogLeNet. ResNet has up to 152 layers and achieves a 96.6% top-5 accuracy at ILSVRC 2015, winning the challenge.

4.2 Choice of a Base Architecture

As in Sec. 3.2.2, we will in a first time fix a CNN architecture, namely fix a CNN_{base} model in the form (13), and then we will analyze the performances we can obtain with such a model while making the training parameters vary. In a second time, fixing the best solutions for the training parameters, we will make the network hyper-parameters vary in order to optimize the architecture.

4.2.1 Description of the CNN_{base}

To get a first idea about the kind of architectures relevant in our context, we chose to test some of state-of-the-art CNN architectures listed in previous section:[20] VGG-16 [SZ14], ResNet-50 [HZRS16] and Inception-v3 [SVI+16]. Our purpose was not to compare their efficiency after some specific tuning but was to check whether one of them seems straightforwardly more adapted to our context than the others. Results are summed-up in Fig. 18 where we have plotted the evolution of the mean rank of the correct key-hypothesis according to the number of epochs. Results are obtained with a 10-fold cross-validation.

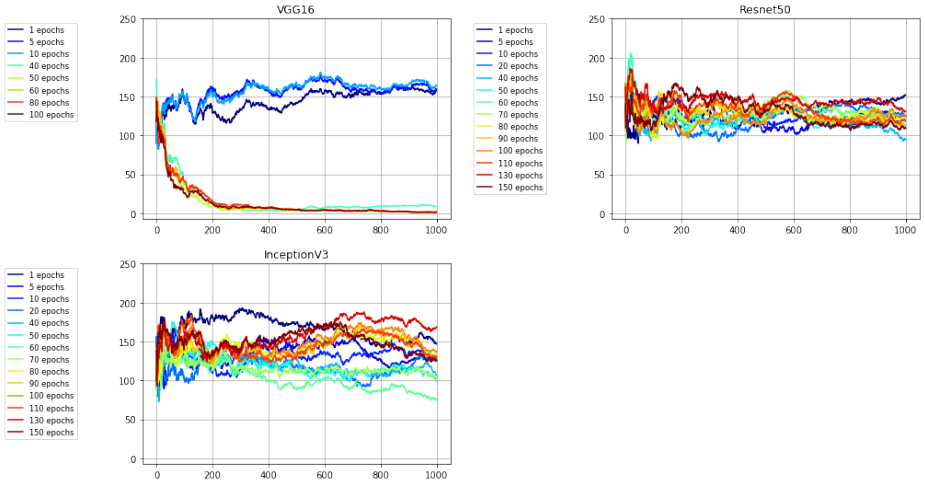

Figure 18: Mean ranks obtained with VGG-16, ResNet-50 and Inception-v3 w.r.t. different epochs.

[20] straightforwardly customized to apply on 1-dimensional inputs of 700 units and outputs of 256 units.

Clearly, ResNet-50 and Inception-v3 do not seem to succeed in extracting key-dependent information for the observations whereas VGG-16 does very well. Based on these preliminary results, we chose to apply the same design principles as in VGG-16 architecture and we investigated the impact on several parameters' configuration on the side-channel attack efficiency. We hereafter gives the general structure of the CNNs studied in the rest of this section: [21]

- n_{blocks} blocks
  - n_{conv} layers of the form CONV + (BN) + ACT
  - 1 POOL layer
- n_{dense} layers FC + ACT
- 1 softmax layer SOFT

Figure 19: Example of CNN architecture based on VGG-16

We moreover added the following rules, which are today classical in literature and enable us to limit the number of different configurations to test.

Rule 1. *CONV layers in the same block have exactly the same configuration (to keep the global volume constant)*

Rule 2. *Each pooling has dimension* 2 *(and hence divides the size of the input by* 2*).*

Rule 3. *The number of filters $n_{filters,i}$ in a CONV layer of the ith block (starting from $i = 1$) satisfies for $i \geq 2$:*

$$n_{filters,i} = \max(n_{filters,1} \times 2^{i-1}, 512) \ .$$

Remark 4. The core idea behind Rule 3 is to keep the global amount of information treated by the different layers as constant as possible. Since each pool layer divides the input dimension by 2, the number of filters is itself multiplied by 2 to compensate it. This idea is inspired by VGG-16.

Rule 4. *All the CONV layers have the same kernel size.*

An example of the tested CNN architecture is given in Fig. 20.

For the initial configuration of our CNN_{base} model used to test the impact of the different hyper-parameters, we select the following values: the number of blocks of CONV layers n_{blocks} is equal to 4, there is only one CONV layer by block ($n_{\text{conv}} = 1$), the number of filters for the first block $n_{filter,1}$ is equal to 64, each filter has kernel size 3 (*same* padding) with ReLU activation functions and a max pooling layer for each block ($W = 2$). CNN_{base} has $n_{\text{dense}} = 2$ final dense layers of $4,096$ units.

4.2.2 Choice of the Hyper-parameters in SCA context

The goal of the following benchmarks is twofold. First, it is to study, for the architecture CNN_{base} presented in previous section and for our dataset, the impact of each (hyper)-parameter on the accuracy and SCA efficiency. Secondly, it is to choose the final hyper-parameters for a new CNN model (hopefully) better than the original one and hence

[21] In our case, we observed that the BN layer was not necessary to get good performances.

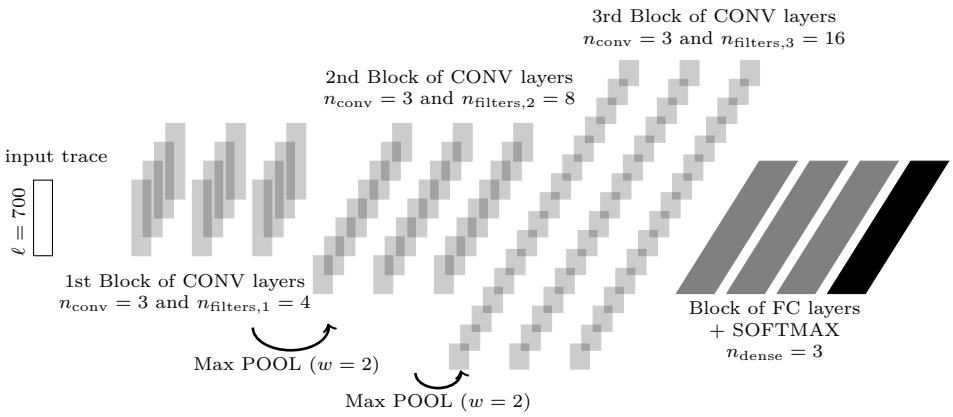

Figure 20: Example of our CNN_{base} architecture

called CNN_{best}. The number of the latter parameters being too large for an exhaustive test of all the possible configurations, we chose to arbitrarily follow a pre-determined sequence of tests. Note that the same CNN_{base} architecture is used for each benchmarking (which each focuses on a single hyper-parameter). For completeness, we also validated the soundness of our choices when the observations are desynchronized. Our goal was not to find the optimal configuration/training strategies but was to identify one of them making sense in our context and leading to accurate results. Other choices are certainly possible and we let the question of determining the most pertinent strategy as an open problem for further studies on this subject. The roadmap followed for our benchmarks is summarized in Table 2. Since our goal is to improve the SCA efficiency, *i.e.* to have the correct hypothesis ranked first with the minimum of traces during the matching/test phase, we always privileged rank-flavoured criteria for parameters selection.

Table 2: Benchmarks Summary

Parameter	Reference	Metric	Range	Choice
Training Parameters				
Epochs	-	rank *vs* time	$10, 25, 50, 60, \ldots, 100, 150$	up to 100
Batch Size	-	rank *vs* time	$50, 100, 200$	200
Architecture Parameters				
Blocks	n_{blocks}	rank, accuracy	[2..5]	5
CONV layers	n_{conv}	rank, accuracy	[0..3]	1
Filters	$n_{\text{filters},1}$	rank *vs* time	$\{2^i; i \in [4..7]\}$	64
Kernel Size	-	rank	$\{3, 6, 11\}$	11
FC Layers	n_{dense}	rank, accuracy *vs* time	[0..3]	2
ACT Function	α	rank	ReLU, Sigmoid, Tanh	ReLU
Pooling Layer	-	rank	Max, Average, Stride	Average
Padding	-	rank	Same, Valid	Same

4.2.3 Training Parameters

The tuning of some training parameters is inherited by the analogous study in MLP context, in order to make results obtained with CNN models and MLP ones comparable. In particular we fixed the training set size to $50,000$,[22] and chose to use the RMSProp optimizer with a learning rate of 10^{-5}.

[22]Leading to 10 training sets of size $45,000$ and 10 test sets of size $5,000$ to perform the 10-fold cross-validation.

Number of epochs and batch size For our campaigns, we did not observe any overfitting (relatively to our rank function) when the number of epochs is increasing. As a direct consequence, the quality of the trained model in terms of our rank function never decreases when the number of epochs increases. Based on this observation, the following benchmarks aim to get the best trade-off between the SCA-efficiency and the training duration/time as a function of the number of epochs and the training batch size.

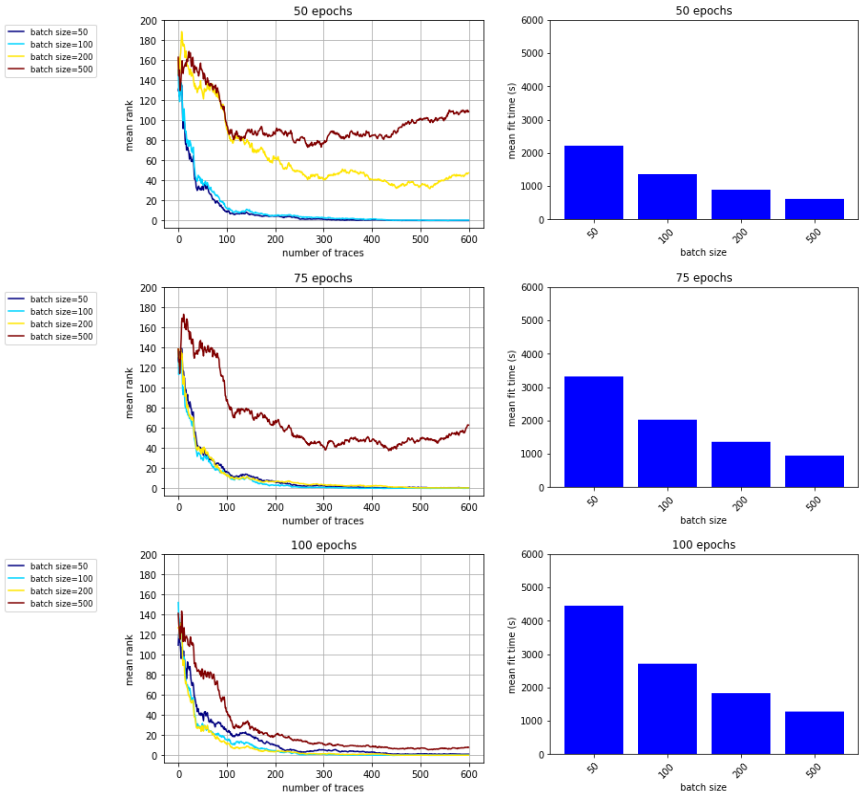

Figure 21: Mean ranks and training time of CNN_{base} with different values of epochs and batch sizes.

The first benchmark series are obtained by training CNN_{base} with different values of epoch and batch size. In a similar way than for MLP, the results show that the SCA-efficiency increases not only when the number of epochs increases but also when the batch decreases. We consider that as a natural behavior since the number of gradient descents (and thus the number of steps in the solving of the minimization problem) increases linearly with the number of epochs and linearly with $(|\mathcal{D}_{\text{train}}|/batch\_size)$. However, as a counterpart, the training duration linearly increases with the number of gradient descent steps, as well. We selected one of the best trade-off, namely 100 epochs and a batch size equal to 200[23]. This choice will allow us to test the impact of the other parameters in our CNN experiments while keeping the training time acceptable.

The following benchmark validates, for a batch size equal to 200, that the previous observation (namely the fact that the SCA-efficiency increases with the number of epochs) stays true when traces are desynchronized.

[23]Having 50 epochs and a batch size equal to 50 is also a good trade-off, but between two options that seem equivalent, we chose to prefer the solution with the highest number of epochs.

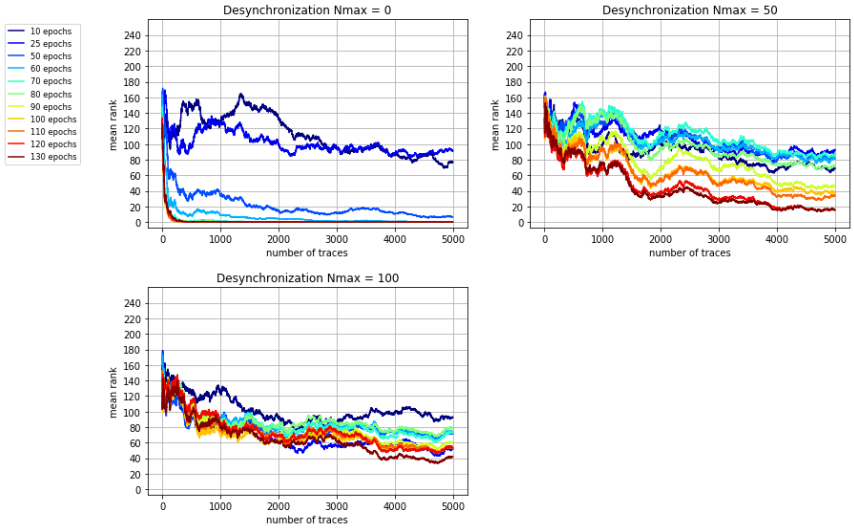

Figure 22: Mean ranks of CNN_{base} for a desynchronization amount in $\{0, 50, 100\}$ and different numbers of epochs.

The desynchronizations of the traces are simulated by generating for each trace a random number δ in $[0..N_{max}]$ and by shifting the original trace of δ points to the left. The samples added to a trace by this processing directly come from the corresponding full trace in `ATMega8515_raw_traces.h5`. For a chosen value N_{max} we then generate a new dataset $\mathcal{D}_{N_{max}}$ from the original one.

With CNN_{base} we manage to obtain with $5,000$ traces a mean rank close to 20 for a maximal desynchronization value $N_{max} = 50$, and a mean rank close to 40 for a maximal desynchronization value $N_{max} = 100$. These results highlight the success of CNN architectures in the context of desynchronized traces, as studied in [CDP17]. Results of Fig. 22 also show the impact of the epoch parameter on the SCA-efficiency with the desynchronization amount: an epoch value of 100 is enough without desynchronization, but it does not yield to the best performance when traces are desynchronized.

4.2.4 Architecture Parameters

In each case, we study the parameters in the context of 256 classes.

Number of Layers The architecture of our CNN_{base} suggests to divide the study of the number of layers in two phases: in a first time we make the number n_{blocks} of blocks vary where a block is composed of convolutional layers followed by a single pooling layer. In this phase we consider, for each block, only $n_{conv} = 1$ convolutional layer per block (which is the configuration of our CNN_{base} model). In a second time, we look for the optimal number n_{conv} of convolutional layers per block assuming that this number is fixed to for all the blocks and that the number blocks equals 4.

Results of the first phase are plotted in Fig. 23. As expected, we notice that the SCA-efficiency increases with the number of blocks. A large difference can be observed in the mean rank between 4 blocks and 5 blocks when desynchronization is in the middle range. This fact can be explained in term of dimension of the input layer before applying the dense layers. The input trace has dimension 700, *i.e.* contains 700 temporal features, corresponding to its time samples. When 5 blocks are used, 5 max pooling layers of stride 2 are applied, and the temporal input dimension is divided by $2^5 = 32$,

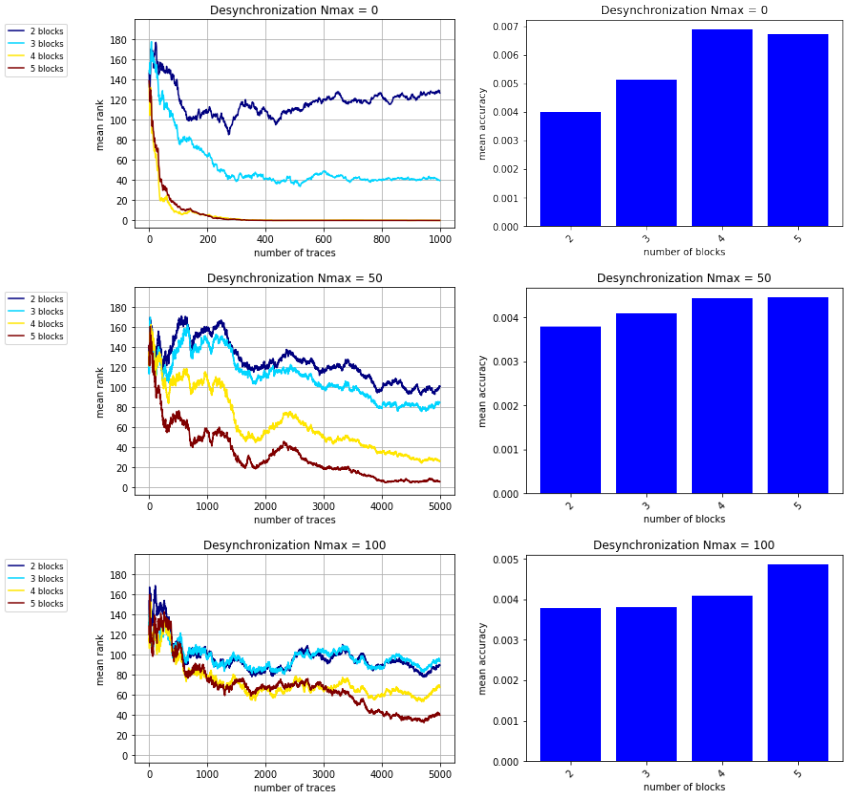

Figure 23: For a desynchronization amount in $\{0, 50, 100\}$ and several numbers of blocks, the mean ranks (left-hand side) and the mean accuracy (right-hand side) of CNN_{base}.

resulting in an input for the first dense layer composed of 21 temporal features times 512 abstract features (43 temporal features times 512 abstract ones if only 4 blocks). The temporal features are those which are directly impacted by the temporal noisy effect of desynchronization. So the less they are the more the model is robust to desynchronization. This explains why adding blocks increases the SCA-efficiency in presence of desynchronization. Thus, we choose $n_{blocks} = 5$ as best parameter. However in our further benchmarks we keep the value $n_{blocks} = 4$; choosing this mid range value allows us to have a better understanding of the impact of the other parameters on the SCA-efficiency.

Results of the second phase are plotted in Fig. 24. We observe that optimal performances are obtained with only one CONV layer per block, even if the SCA-efficiency seems to be dimly impacted by n_{conv}. This is probably due to the fact that increasing number of layers, the number of trainable weights increases as well. To observe a benefit we should let models with more weights train longer, but for our benchmark we fixed the number of epochs (100 in our experiment). So not only we do not observe any benefit in augmenting the n_{conv}, but actualy we observe performances slightly decreasing for an under-fitting phenomenon due to the lack of epochs. Observing results obtained with $n_{conv} = 0$, we verify the fact that the performance of the network is impacted by desynchronization when no CONV layer at all is exploited. The claim of CONV layers overcoming desynchronization issue by extracting patterns in the trace is then verified. Those conclusions are perfectly in line with [PSH+18] which observes that CONV layers play a minor role when the observations traces are perfectly synchronized.

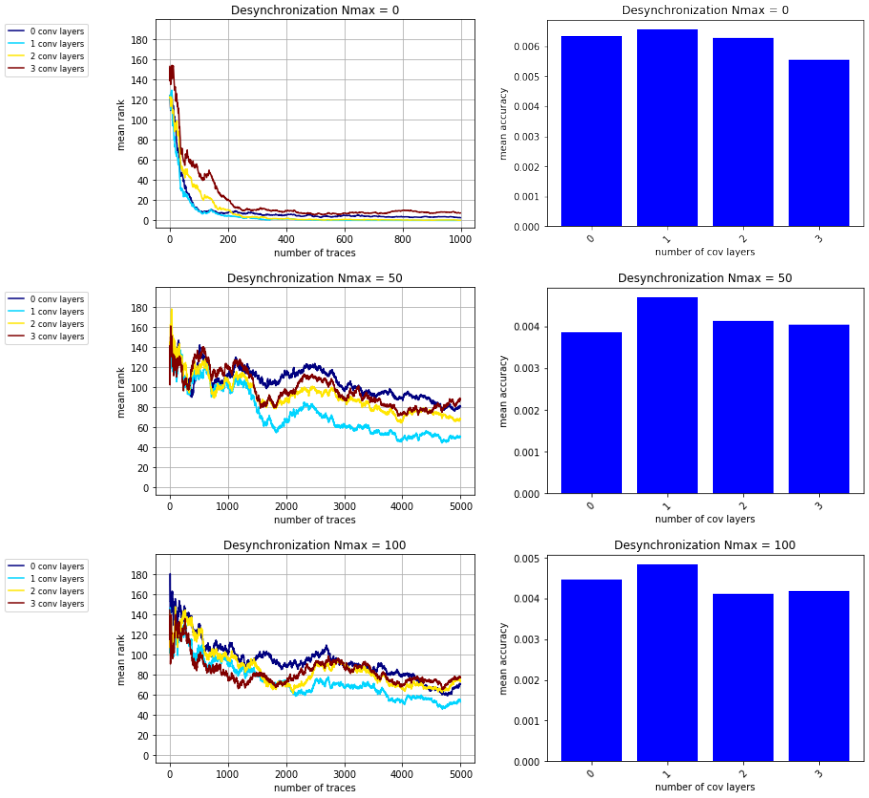

Figure 24: For a desynchronization amount in $\{0, 50, 100\}$ and different numbers of CONV layers per block, the mean ranks (left-hand side) and the mean accuracy (right-hand side) of CNN_{base}.

Number of filters. Following Rule 3, the aim of the next benchmark is to test several values for the number of filters in the CONV layers of the first block (denoted $n_{filters,1}$ in Rule 3). Fig. 25 shows that increasing the number of filters also increases the SCA-efficiency. However it also obviously increases the time of the training which leads us to look at a good trade-off between efficiency and computational time.

Kernel size. We here study the impact of the kernel size (aka the dimension of the convolutional filters) on the model efficiency. In parallel, we compare two different approaches which either consist in combining several convolutional layers with small dimension or in selecting one unique convolutional layer with high dimension.

Unexpectedly, the kernel size significantly impacts the SCA-efficiency. We obtain a mean rank below 10 with a desynchronization amount of 100 by increasing the size of the filters to 11, whereas we only obtain a mean rank of 40 by increasing the number of convolutional layers to 3 for each block with filters of size 3. For a complete comparison we also increased the number of epochs to 200 in our second experiment but it clearly does not improve the efficiency in the same scale than the kernel size. This behaviour is very different than the one expected if we refer to recent results in computer vision where the trend is to increase the number of layers with filters of small size.

Number of fully-connected final layers For this benchmark, we train four versions of CNN_{base} with different numbers of fully-connected final layers. Each of these dense layers

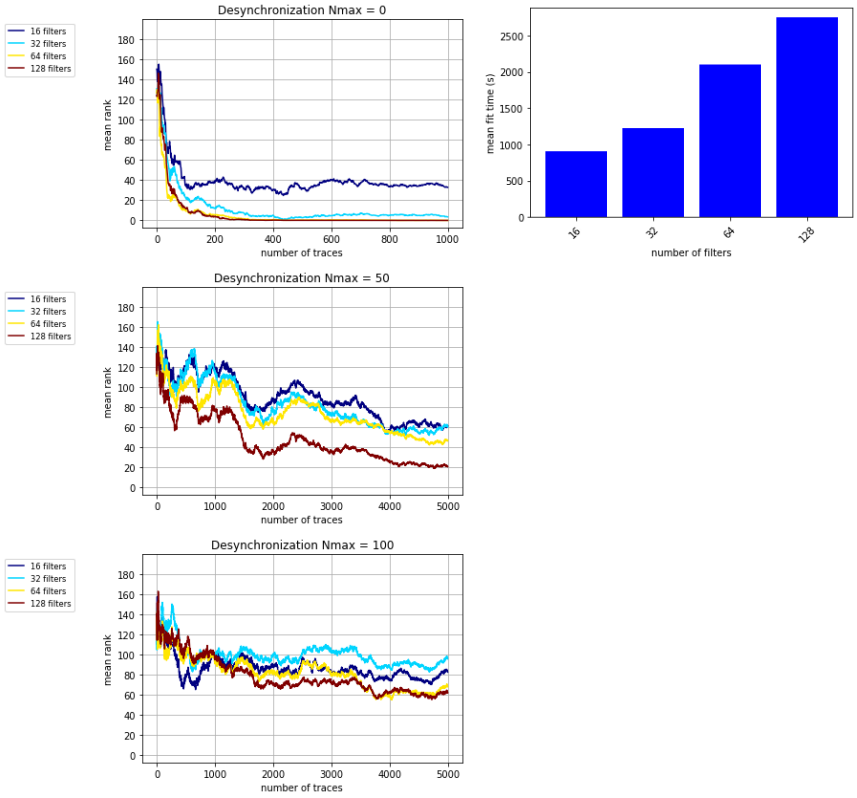

Figure 25: For a desynchronization amount in $\{0, 50, 100\}$ and different values of initial number of filters, the mean ranks (left-hand side) and the mean training time (right-hand side) of CNN_{base}.

has $4,096$ units. We observe from Fig. 27 that the network requires at least one dense layer when the traces are synchronized. Roughly speaking, this suggest that, for SCA attacks, the QDA part of CNN networks is simulated by dense layers, while convolutional layers essentially extract information (*e.g.* by combining leakage points and/or by dealing with the desynchronization). The results also confirm that the number of dense layers increases the SCA-efficiency. Hence fully connected layers are a critical part of the CNN network in the context of SCA, and shall be not removed. This differs from the recent trend in computer vision where dense layers are replaced by an average pooling layer and it explains the poor results of Inception-v3 and ResNet-50 in our experiments (Fig. 18).

Activation Functions For the three tested activation functions, Fig. 28 shows that only ReLU can afford a good efficiency for SCA when desynchronization is below 50. Therefore we recommend the usage of ReLU activation function for CNN architecture.

Pooling Layer We test three pooling methods which are applied in our experiment to all the pooling layers of CNN_{base}: max pooling, average pooling, and *stride pooling*[24]. Contrary to standard CNN architectures used in computer vision that rely on max pooling, we obtain the best results with average pooling layers (Fig. 29).

[24]Stride pooling consists in taking the first value on each input window defined by the stride.

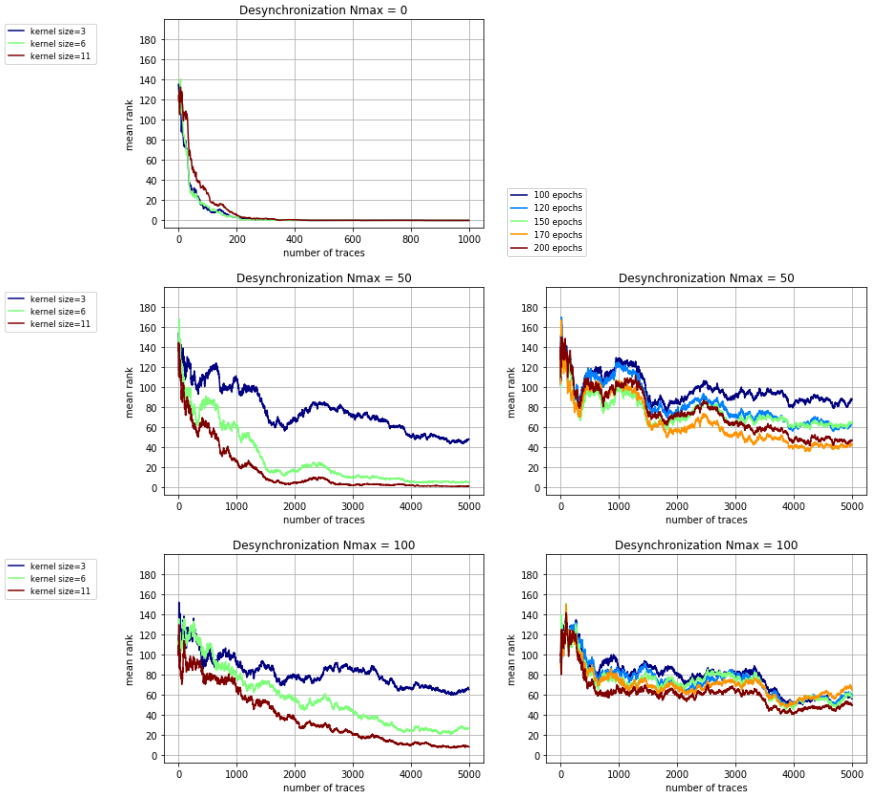

Figure 26: For a desynchronization amount in $\{0, 50, 100\}$, the mean ranks for CNN_{base} with different kernel sizes (left-hand side) and the mean ranks of CNN_{base} with 3 layers of dimension 3 in each block for different epochs (right-hand side).

Padding Finally we test two configurations of padding. Results in Fig. 30 show that this parameter does not have a significant impact on the SCA-efficiency.

CNN_{best} At the end of these benchmarks we are able to define the best CNN architecture based on our selection of the parameters. Therefore we define CNN_{best} as the CNN architecture with 5 blocks and 1 convolutional layer by block, a number of filters equal to $(64, 128, 256, 512, 512)$ with kernel size 11 (*same* padding), ReLU activation functions and an average pooling layer for each block. The CNN has 2 final dense layers of $4,096$ units. CNN_{best} is trained with a batch size equal to 200 by using RMSprop optimizer with an initial learning rate of 10^{-5}. According to previous benchmarks and results of Fig. 22, a number of epochs above 130 is necessary to obtain the best results with CNN_{base} on desynchronized traces. In our experiments on CNN_{best}, we noticed that a training with 75 epochs is sufficient. For robustness and because it had an acceptable computing time impact, we eventually chose to benchmark until a number of epochs equal to 100.

5 Attack Comparisons on desynchronized traces

In this section we perform on desynchronized traces a last comparison of the efficiency of the four models studied throughout this article, namely: VGG-16, PCA-QDA (aka Template Attacks), MLP_{best} and CNN_{best}. As in the previous sections, models are evaluated with a 10-fold cross-validation on $50,000$ traces.

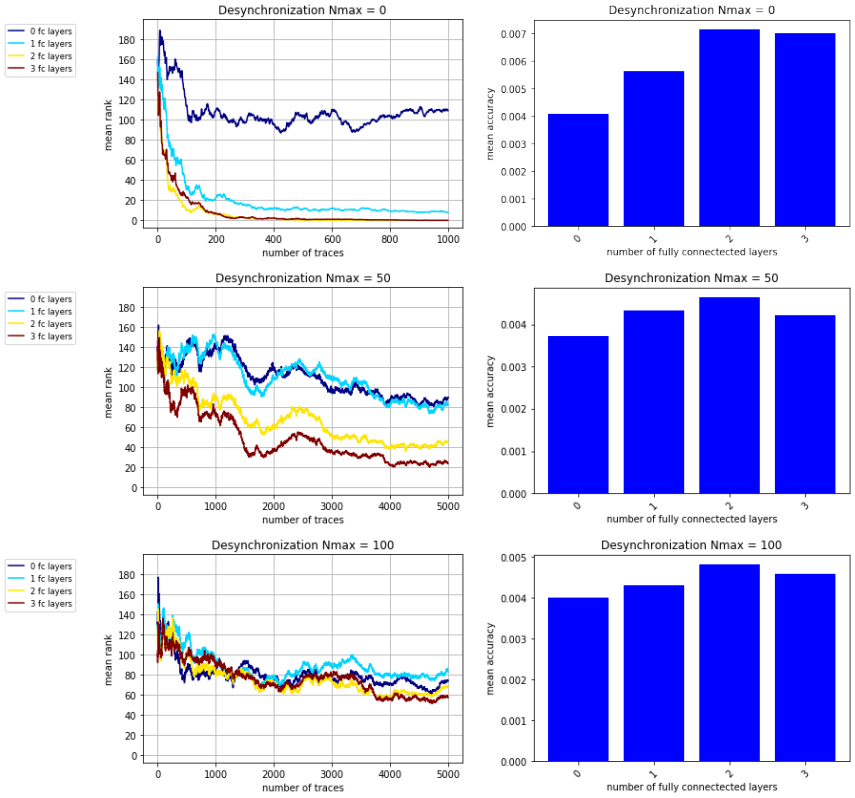

Figure 27: For a desynchronization amount in $\{0, 50, 100\}$ and different numbers of final fully-connected layers, the mean ranks (left-hand side) and the mean accuracy (right-hand side) of CNN_{base}.

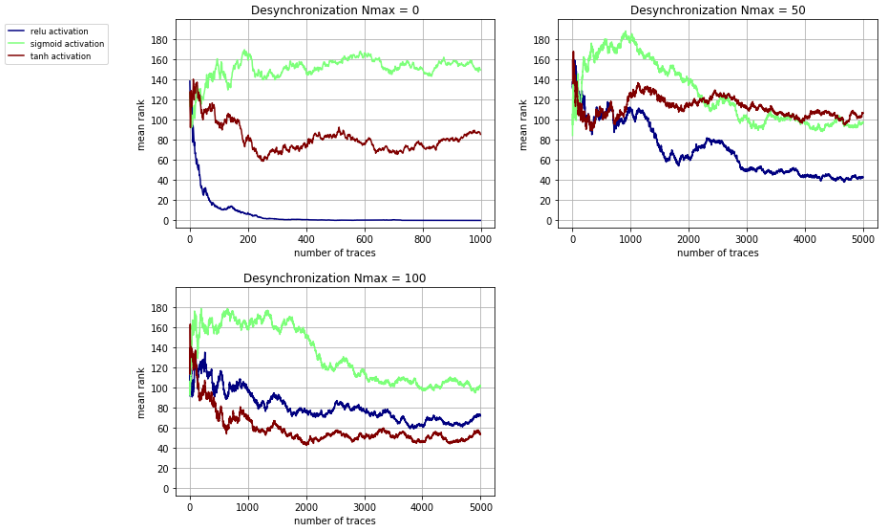

Figure 28: For a desynchronization amount in $\{0, 50, 100\}$ and activation function in $\{relu, tanh, sigmoid\}$, the mean ranks of CNN_{base}.

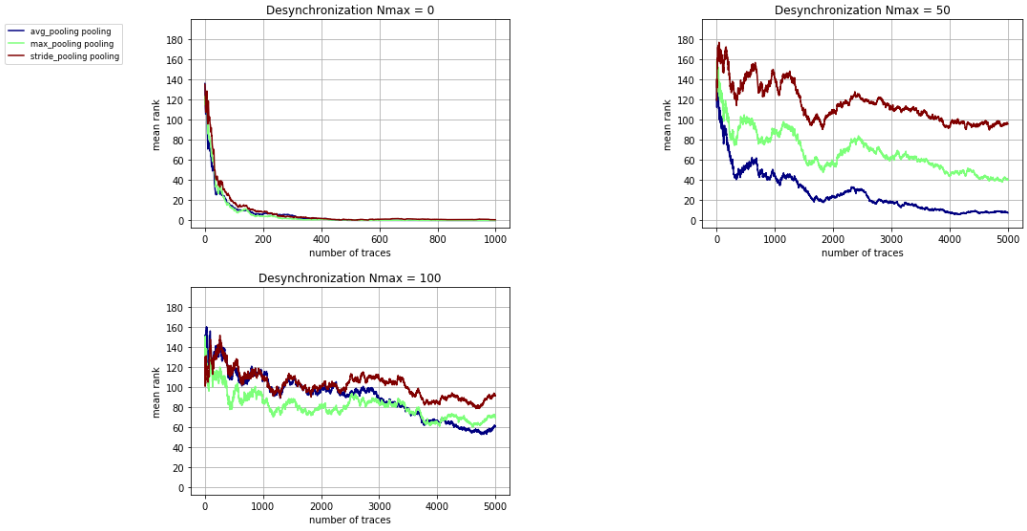

Figure 29: For a desynchronization amount in $\{0, 50, 100\}$ and different pooling layers (either max or average or stride pooling), the mean ranks of CNN_{base}.

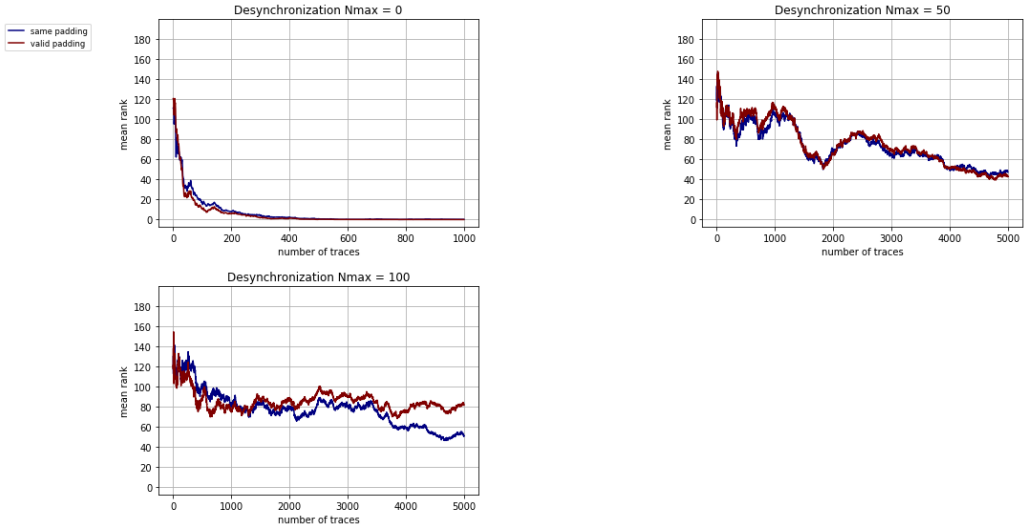

Figure 30: For a desynchronization amount in $\{0, 50, 100\}$ and a padding option either set to *same* or to *valid*, the mean ranks of CNN_{base}.

VGG-16 We train VGG-16 with different numbers of epochs. The size of the batch is equal to 200 and we use RMSprop optimizer with a initial learning rate of 10^{-5}. The results for different desynchronization values are plotted in Fig. 31. As expected, the SCA-efficient increases with the number of epochs and we select 150 epochs for this model.

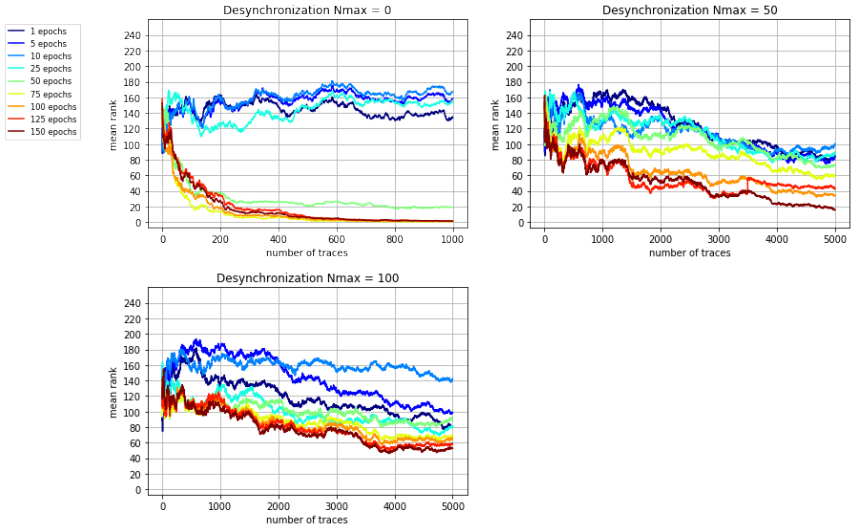

Figure 31: VGG-16 mean ranks for a desynchronization amount in $\{0, 50, 100\}$ and different numbers of epochs.

Template attacks As described in Subsection 3.3.3, we first perform an unsupervised PCA on the 700 samples of the traces in the dataset $\mathcal{D}_{\text{profiling}}$ and we apply a QDA on the resulting components. Fig. 32 shows the impact of the desynchronization on Template Attacks. We note that the attack still succeeds when desynchronization is less than 50, but fails for a desynchronization amount of 100. Best results are obtained with a PCA reduction on 10 and 50 components with respectively a desynchronization amount of 0 and 50.

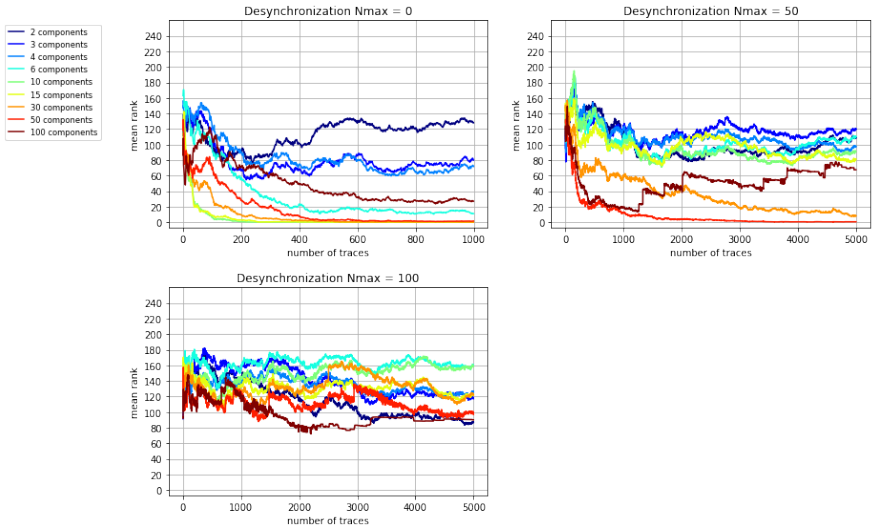

Figure 32: Template Attacks mean ranks for a desynchronization amount in $\{0, 50, 100\}$ and different values of PCA reduction.

MLP_{best} We train MLP_{best} on the desynchronized traces with different numbers of epochs. As shown in Fig. 33, MLP is very sensitive to desynchronization and increasing the number of epochs is not enough to get better results.

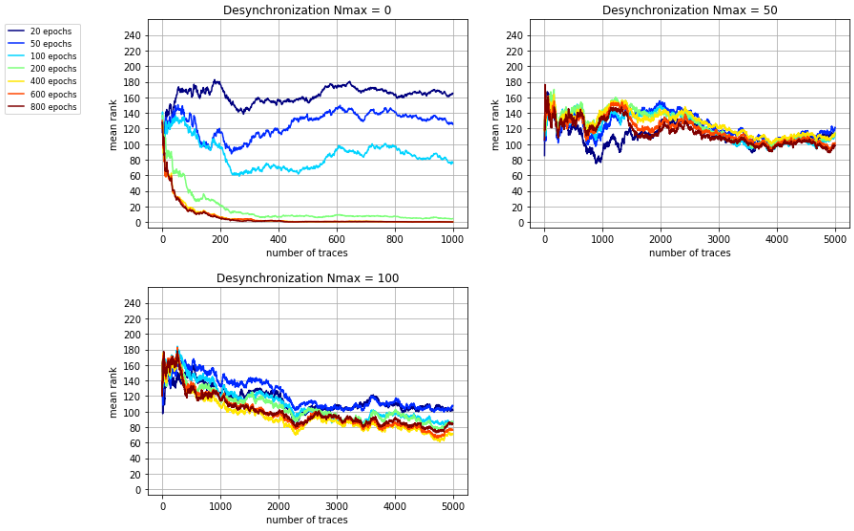

Figure 33: Mean ranks of MLP_{best} for a desynchronization amount in $\{0, 50, 100\}$ and different values of epochs.

CNN_{best} Finally we evaluate CNN_{best} with the parameters described in the previous subsection. Results are displayed in Fig. 34.

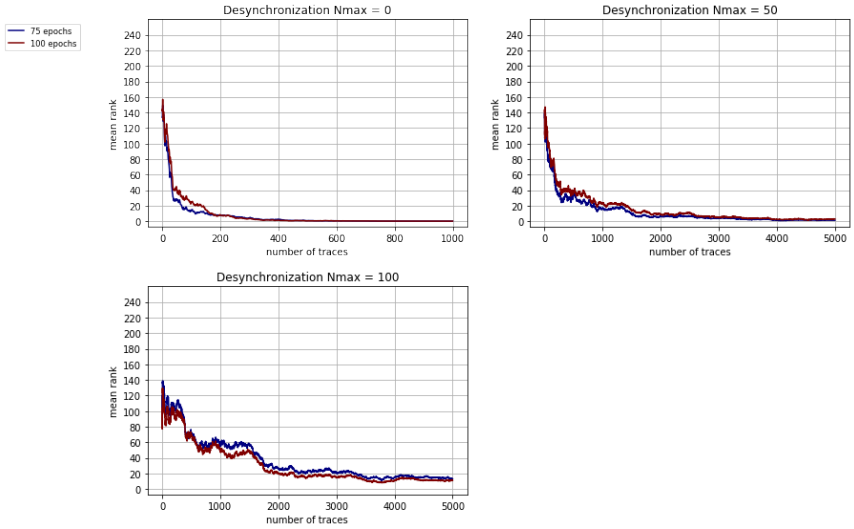

Figure 34: Mean ranks of CNN_{best} for a desynchronization amount in $\{0, 50, 100\}$.

Summarize of the results In Fig. 35 we compare the best results obtained from the models. CNN_{best} outperforms all the other models on desynchronized traces with only 75 epochs. VGG-16 has decent results too, but with an higher number of epochs. CNN_{best} and Template Attacks combined to a PCA have similar results with small desynchronization, however this second model performs poorly with a high desynchronization amount. MLP_{best} has good performances on synchronized traces, but is very sensitive to desynchronization[25].

[25]For the sake of completeness, we have also tested the SCANet model introduced in [PSH+18]. This did not yield to good performances on our dataset: we have obtained a mean rank of approximatively 128 for each of our desynchronizations 0, 50 and 100.

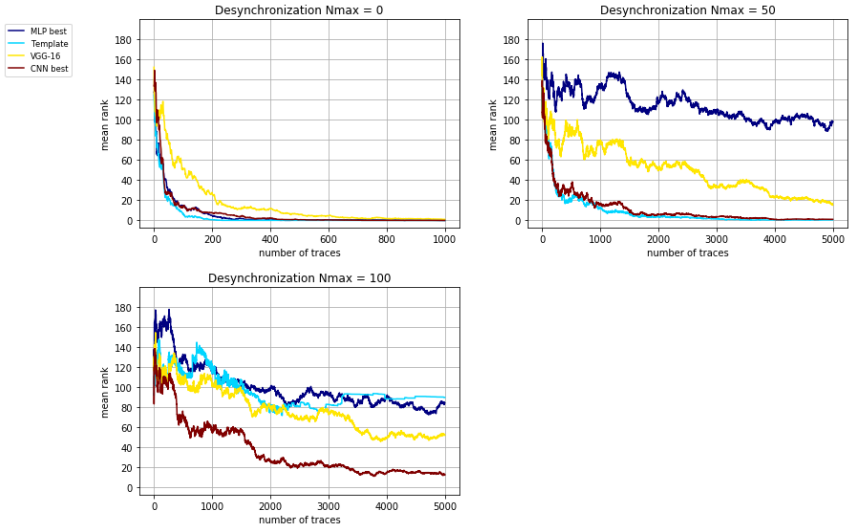

Figure 35: Mean ranks of the best models for a desynchronization amount in $\{0, 50, 100\}$.

6 Conclusions And Perspectives

In this paper, we have conducted a thorough study of the application of deep learning theory in the context of side-channel attacks. In particular, we have discussed several parametrization options and we have presented a large variety of benchmarks which have been used to either experimentally validate our choices or to help us to take the adequate decision. The methodologies followed for the hyper-parameters selection may be viewed as a proposal to help researchers to make their own choice for the design of new deep learning models. They also open the way for further research in this domain. Since convolutional neural networks are shown almost similarly efficient as multi-layer perceptron networks in the context of perfectly synchronized observations but outperform them in presence of desynchronization/jittering, our study suggests that CNN models should be preferred in the context of SCA (even if they are more difficult to train). When it comes to choose a base architecture for the latter models, our study shows that the 16-layer network VGG-16 used by the VGG team in the ILSVRC-2014 competition [SZ14] is a sound starting point (other public models like ResNet-50 [HZRS16] or Inception-v3 [SVI+16] are shown to be inefficient). Our results show that VGG-16 allows to design architectures which, after training, are better than classical Templates Attacks even when combined with dimension reduction techniques like Principal Component Analysis (PCA) [Pea01]. All the benchmarkings have been done with the same target (and database) which corresponds to an AES implementation secured against first order side-channel attacks and developed in assembly for an ATMega8515 component. This project has been published on [ANS18b]. To enable perfect reproducing of our experiments and benchmarks, we also chose to publish the electromagnetic measurements acquired during the processing of our target AES implementation (available in [ANS18a]) together with example Python scripts to launch some initial training and attacks based on these traces. We think that this ASCAD database may serve as a common basis for researchers willing to compare their new architectures or their improvements of existing models.

Acknowledgments

This work has been funded in parts by the European Commission through the H2020 project 731591 (acronym REASSURE).

References

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[AG01] Mehdi-Laurent Akkar and C. Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Ç.K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 2001.

[ANS18a] ANSSI. Ascad database. https://github.com/ANSSI-FR/ASCAD, 2018.

[ANS18b] ANSSI. secaes-atmega8515. https://github.com/ANSSI-FR/secAES-ATmega8515, 2018.

[B+96] Leo Breiman et al. Heuristics of instability and stabilization in model selection. *The annals of statistics*, 24(6):2350–2383, 1996.

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

[BCO04] É. Brier, C. Clavier, and F. Olivier. Correlation Power Analysis with a Leakage Model. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.

[BG05] Yoshua Bengio and Yves Grandvalet. Bias in estimating the variance of k-fold cross-validation. In *Statistical modeling and analysis for complex data problems*, pages 75–95. Springer, 2005.

[BGP+11] L. Batina, B. Gierlichs, E. Prouff, M. Rivain, F.-X. Standaert, and N. Veyrat-Charvillon. Mutual information analysis: a comprehensive study. *to appear in the Journal of Cryptology*, 24(2):269–291, April 2011.

[Bis06] Christopher M.. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[BLR13] 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 CARDIS*, volume 7771 of *Lecture Notes in Computer Science*, pages 263–276. Springer Berlin Heidelberg, 2013.

[BYC13] James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in Science Conference*, pages 13–20, 2013.

[C+15] François Chollet et al. Keras. https://github.com/fchollet/keras, 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.

[CRR02] S. Chari, J.R. Rao, and P. Rohatgi. Template Attacks. In B.S. Kaliski Jr., Ç.K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2002.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[DPRS11] Julien Doget, Emmanuel Prouff, Matthieu Rivain, and François-Xavier Standaert. Univariate Side Channel Attacks and Leakage Modeling. *Journal of Cryptographic Engineering*, 1(2):123–144, 2011.

[FHT01] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[Fis22] Ronald A. Fisher. On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society*, 1922.

[Fis36] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(7):179–188, 1936.

[FR14] Aurélien Francillon and Pankaj Rohatgi, editors. *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*. Springer, 2014.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[GBC16a] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GBC16b] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[GBPV09] Benedikt Gierlichs, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. Revisiting Higher-Order DPA Attacks: Multivariate Mutual Information Analysis. Cryptology ePrint Archive, Report 2009/228, 2009. http://eprint.iacr.org/.

[GHO15] Richard Gilmore, Neil Hanley, and Máire O'Neill. Neural network based attack on a masked implementation of AES. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*, pages 106–111. IEEE Computer Society, 2015.

[Groa] HDF Group. The hdf group.

[Grob] HDF Group. HDF5 For Python.

[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.

[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.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[JKL+09] Kevin Jarrett, Koray Kavukcuoglu, Yann LeCun, et al. What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2146–2153. IEEE, 2009.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M.J. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[KUMH17] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *arXiv preprint arXiv:1706.02515*, 2017.

[LB+95] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBD+89] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

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

[LCB] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits.

[LH05] Yann LeCun and Fu Jie Huang. Loss functions for discriminative training of energy-based models. In Robert G. Cowell and Zoubin Ghahramani, editors, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, AISTATS 2005, Bridgetown, Barbados, January 6-8, 2005*. Society for Artificial Intelligence and Statistics, 2005.

[LMBM13] Liran Lerman, Stephane Fernandes Medeiros, Gianluca Bontempi, and Olivier Markowitch. A machine learning approach against a masked AES. In Francillon and Rohatgi [FR14], pages 61–75.

[LPB+15] Liran Lerman, Romain Poussier, Gianluca Bontempi, Olivier Markowitch, and François-Xavier Standaert. Template attacks vs. machine learning revisited (and the curse of dimensionality in side-channel analysis). In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*, volume 9064 of *Lecture Notes in Computer Science*, pages 20–33. Springer, 2015.

[LT16] Henry W. Lin and Max Tegmark. Why does deep and cheap learning work so well? *CoRR*, abs/1608.08225, 2016.

[MDM16] Zdenek Martinasek, Petr Dzurenda, and Lukas Malina. Profiling power analysis attack based on MLP in DPA contest V4.2. In *39th International Conference on Telecommunications and Signal Processing, TSP 2016, Vienna, Austria, June 27-29, 2016*, pages 223–226. IEEE, 2016.

[Mes00] T.S. Messerges. Using Second-order Power Analysis to Attack DPA Resistant software. In Ç.K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 238–251. Springer, 2000.

[MHK10] David A. McAllester, Tamir Hazan, and Joseph Keshet. Direct loss minimization for structured prediction. In John D. Lafferty, Christopher K. I. Williams, John Shawe-Taylor, Richard S. Zemel, and Aron Culotta, editors, *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada.*, pages 1594–1602. Curran Associates, Inc., 2010.

[MHM13] Zdenek Martinasek, Jan Hajny, and Lukas Malina. Optimization of power analysis using neural network. In Francillon and Rohatgi [FR14], pages 94–107.

[MMT15] Zdenek Martinasek, Lukas Malina, and K. Trasy. Profiling Power Analysis Attack Based on Multi-layer Perceptron Network. *Computational Problems in Science and Engineering*, 343, 2015.

[MPO05] S. Mangard, N. Pramstaller, and E. Oswald. Successfully Attacking Masked AES Hardware Implementations. In Rao and Sunar [RS05], pages 157–171.

[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.

[NH10] Vinod Nair and Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 807–814. Omnipress, 2010.

[Pea01] Karl Pearson. On Lines and Planes of Closest Fit to Systems of Points in Space. *Philosophical Magazine*, 2(11):559–572, 1901.

[PR08] Emmanuel Prouff and Matthieu Rivain. A Generic Method for Secure SBox Implementation. In Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, *WISA*, volume 4867 of *Lecture Notes in Computer Science*, pages 227–244. Springer, 2008.

[PSH+18] Stjepan Picek, Ioannis Petros Samiotis, Annelie Heuser, Jaehun Kim, Shivam Bhasin, and Axel Legay. On the Performance of Deep Learning for Side-channel Analysis). *IACR Cryptology ePrint Archive*, 2018:004, 2018.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[RM08] Lior Rokach and Oded Maimon. *Data Mining with Decision Trees: Theroy and Applications*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2008.

[RS05] J.R. Rao and B. Sunar, editors. *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*. Springer, 2005.

[Sch08] Werner Schindler. Advanced Stochastic Methods in Side Channel Analysis on Block Ciphers in the Presence of Masking. *Journal of Mathematical Cryptology*, 2:291–310, 2008.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[SLP05] Werner Schindler, Kerstin Lemke, and Christof Paar. A Stochastic Model for Differential Side Channel Cryptanalysis. In Rao and Sunar [RS05].

[SMY06] Francois-Xavier Standaert, Tal G. Malkin, and Moti Yung. A Formal Practice-Oriented Model For The Analysis of Side-Channel Attacks. Cryptology ePrint Archive, Report 2006/139, 2006.

[SSZU15] Yang Song, Alexander G. Schwing, Richard S. Zemel, and Raquel Urtasun. Direct loss minimization for training deep neural nets. *CoRR*, abs/1511.06411, 2015.

[SVI+16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[WW98] J. Weston and C. Watkins. Multi-class support vector machines, 1998.

[ZF14] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.