HAL
open science

Improving Deep Learning Networks for Profiled Side-channel Analysis Using Performance Improvement Techniques

Damien Robissout, Lilian Bossuet, Amaury Habrard, Vincent Grosso

To cite this version:

Damien Robissout, Lilian Bossuet, Amaury Habrard, Vincent Grosso. Improving Deep Learning Networks for Profiled Side-channel Analysis Using Performance Improvement Techniques. ACM Journal on Emerging Technologies in Computing Systems, 2021, 17 (3), pp.1-30. 10.1145/3453162 . hal-03438972

HAL Id: hal-03438972

https://hal.science/hal-03438972

Submitted on 22 Nov 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Improving Deep Learning Networks for Profiled Side-Channel Analysis Using Performance Improvement Techniques

Damien Robissout Lilian Bossuet Amaury Habrard
Vincent Grosso

firstname.lastname@univ-st-etienne.fr

Univ Lyon, UJM-Saint-Etienne, CNRS Laboratoire Hubert Curien
UMR 5516 F-42023, Saint-Etienne, France

Abstract

The use of deep learning techniques to perform side-channel analysis attracted the attention of many researchers as they obtained good performances with them. Unfortunately, the understanding of the neural networks used to perform side-channel attacks is not very advanced yet. In this paper, we propose to contribute to this direction by studying the impact of some particular deep learning techniques for tackling side-channel attack problems. More precisely, we propose to focus on three existing techniques: batch normalization, dropout and weight decay, not yet used in side-channel context. By combining adequately these techniques for our problem, we show that it is possible to improve the attack performance, *i.e.* the number of traces needed to recover the secret, by more than 55%. Additionally, they allow us to have a gain of more than 34% in terms of training time. We also show that an architecture trained with such techniques is able to perform attacks efficiently even in the context of desynchronized traces.

1 Introduction

Side-channel attacks are a class of attacks targeting electronic systems by exploiting vulnerabilities in their physical properties, *e.g.* power consumption [10] or electromagnetic emanation [1], to recover secret information. This kind of attacks uses the fact that, during the execution of cryptographic algorithms, sensitive variables depending on the secret are manipulated. Profiling side-channel attacks are among the most powerful attacks. To perform profiling attacks, an adversary uses a device similar to the targeted one in order to create a template of, for example, the power consumption

for a given secret [5]. The adversary has access to all the variables manipulated by the device therefore he can estimate the conditional distribution between the power consumption of the device during the computations and the sensitive variable for every possible value. This distribution is then used to predict the value of the sensitive variable of the device under attack and retrieve the secret. However, this technique has some limitation. Indeed, it is based on the assumption that the distribution of the leakage follows a gaussian distribution. The estimation of the probability density function of such a distribution becomes expensive in high dimensional scenarios.

Recently, thanks to advances in technology, deep neural networks gained in popularity due to their approximation power. Researchers began to use machine learning and deep learning techniques to estimate the conditional distribution of the sensitive variable and perform profiling attacks. The similarities between the two problems led to good performances of this new type of profiling attacks. In particular, convolutional neural networks (CNN) showed a great robustness against classical countermeasures, which was highlighted in some articles, such as *masking* in [14, 15] and *desynchronization* in [4, 28]. Their main advantage is that they do not require a heavy pre-processing of the data contrary to other profiling attacks, *e.g. template attacks* [5]. On the other hand, they need to go through a learning process to be able to perform specific tasks. It is composed of two phases, the forward propagation and the backward propagation. The first one consists in feeding the network training examples, for which the network outputs a prediction after processing the examples using a sequence of linear and non-linear transformations. Once this is done, the error between the prediction and the correct output is measured with respect to a loss function, and the backward propagation aims at propagating back this error to the start of the network in order to update the parameters of the network accordingly.

The evaluation of the training process and its performance is done using machine learning metrics, among which the accuracy is one of the most popular. Unfortunately, as Picek *et al.* have shown [18], this metric is poorly suited in the context of side-channel analysis. By definition, the accuracy tends to favor the class with the highest output score for each example. However, in side-channel analysis, as the leaked information contained in the data is very small, the classifiers are only loosely correlated to the true prediction. Therefore, to perform a successful attack, the adversary combines the predictions of multiple traces to extract the estimate of the true class. In addition, the lack of information in the side-channel data means that the network has a tendency to overfit, *i.e.* learn the training data perfectly even if it reduces the performance on new data and this is confirmed by the evaluation metrics. This phenomenon is common when training deep neural networks and there exists different techniques to reduce its effect. The next step is then to apply regularization techniques to limit the overfitting of the network. Those techniques rely on the possibility of evaluating

the network during its learning phase. However, the metrics, such as the accuracy, are not efficient when applied to the problem of side channel analysis. More precisely, they are able to tell when a network is overfitting but are not enough to distinguish which network would perform best. This brings the question of how to compare networks and their performances. Answering this problem would also lead to the possibility of applying early stopping during the training phase to further improve the performance of the networks.

Contributions The first aim of this article is to do an overview of the current state of deep learning based side-channel analysis. This overview underlines some of the problems linked to this kind of attacks which depends on the performance of deep learning algorithms. The goal is to explain in details the principles of deep learning based side channel analysis and study the effects of the addition of some deep learning techniques, specifically dropout and L_2 regularization, also known as weight decay, as well as batch normalization, on the performances. They help to prevent and mitigate the overfitting of the network which brings significant improvements to the performance by reducing the number of traces needed to perform the attacks. This is validated experimentally using a public architecture CNN_{best} and dataset ASCAD [2].

Article Organization This article is organized as follows. Section 2 will serve as an introduction to the notations and important notions of profiled side-channel analysis as well as deep learning and neural networks. Section 3 will describe batch normalization and regularization, techniques that can be used to improve the performance of neural networks. In Section 4, we present the metric used for the evaluations and introduced in [19]. The previously mentioned techniques are then used in Section 5 together with $\Delta^d_{train,val}$ to improve a neural network architecture existing in the literature. Finally, Section 6 contains the conclusion on the results obtained and a discussion for future works that can be explored.

2 Preliminaries

2.1 Profiled Side-Channel analysis

The goal of profiled side-channel analysis is to recover a secret value or parts of a secret value used in the computation of an encryption algorithm. The recovery of this value leads to the possibility of deciphering a message, be it a direct recovery, a partial recovery or a reduction of the difficulty of the problem the encryption is based on. This kind of attacks is based on several assumptions:

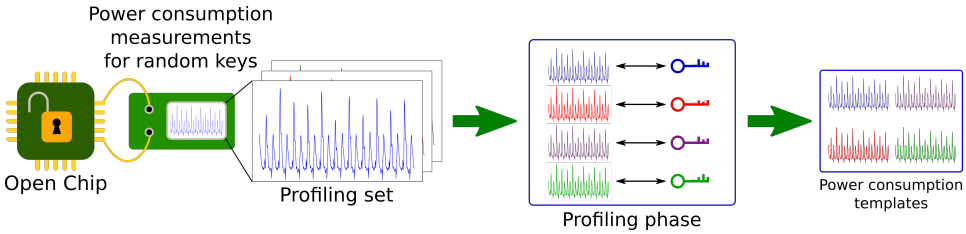


Figure 1: Diagram describing the steps of a profiling phase.

- the adversary has access to a physical measurement from the device used to perform the encryption;
- information on intermediate variables is leaking in the measurements, *i.e.* it is correlated to values manipulated by the algorithms. This is usually referred as the leakage model;
- the values leaking are related to the secret or part of the secret;
- the adversary has a copy of the device over which he has total control of the values manipulated.

For the rest of the article, the physical measurements are assumed to be power consumption. They are stored in the form of temporal vectors called traces, denoted $\mathbf{t}$ to represent one of those traces, and regrouped in a set $\mathbf{T} \in \mathbb{R}^{N \times D}$ where N is the number of traces in $\mathbf{T}$ and D is the dimension of the traces. The intermediate value leaking in the traces is $Z = f(P, K)$, where f denotes a cryptographic primitive, P ($\in \mathcal{P}$) denotes a public variable (*e.g.* a plaintext or ciphertext) and K ($\in \mathcal{K}$) denotes a part of the key (*e.g.* a byte). The goal of the attack is to retrieve the value of $\mathbf{k}^*$, the secret key used by the cryptographic algorithm. In order to do that, a common method used is the *divide and conquer* approach which consists in finding fractions of the key (*e.g.* a bit or a byte) separately and combining them to obtain the full key. The type of attacks considered in this article to recover the parts of the key are profiling attacks. They are done in two phases: the first one is the profiling phase and the second the matching phase.

During the profiling phase, illustrated in Figure 1, the adversary has access to a test device, which is a copy of the device he wants to attack. He has control of every intermediate variables used in the computations done during the encryption and therefore can determine when the sensitive variable Z ($\in \mathcal{Z}$) is leaking. This allows the adversary to build a model $F : \mathbb{R}^D \rightarrow \mathbb{R}^{|\mathcal{Z}|}$ to estimate the probability $Pr[Z = z|\mathbf{T}]$, *i.e.* what is the probability, given the set of traces $\mathbf{T}$, that the value of the sensitive variable is z.

The model is then used during the matching phase to estimate the values of the intermediate variable on the device under attack. The adversary first

performs encryptions of random public variables with this device and then uses the power consumptions obtained from those encryptions to get the estimations. The probabilities obtained for each trace are combined and the value with the highest probability is considered as the recovered key.

In order to evaluate the performance of the model, all the key candidates are classified in a vector of size $|\mathcal{K}|$, denoted $\mathbf{g} = (g_1, g_2, ..., g_{|\mathcal{K}|})$, following their respective probability. g_1 is considered to be the most likely candidate and $g_{|\mathcal{K}|}$ the least likely. The actual position of the bth byte of the secret key in $\mathbf{g}$ is denoted $\mathbf{g}(\mathbf{k}^*[b])$ and is called *rank*. The *guessing entropy* [22] is defined as the average rank of a byte b of $\mathbf{k}^*$, denoted $\mathbf{k}^*[b]$, among all key hypotheses. It is a common metric in side-channel analysis to evaluate the performance of attacks. A successful attack, using N_a traces, is equivalent to a guessing entropy equal to 1.

A related metric to the rank is the *success rate* defined as the probability for an attack to succeed in recovering the correct key byte $\mathbf{k}^*[b]$ among all the hypotheses. A success rate of p is equivalent to having p recoveries of $\mathbf{k}^*[b]$ in 100 attacks. In [22], Standaert *et al.* propose to extend the notion of success rate to an arbitrary order d. Let $A_{E_k,L}$ be an adversary trying to attack the cryptographic computation E_k using to a leakage model L. The adversary conducts the experiment $Exp^d_{A_{E_k,L}}$ multiple times in order to exploit the relevant leaking information. The result of the attack is a vector $\mathbf{g}$ of length d that is composed of the d most likely key candidates sorted according to the experiments result. If $\mathbf{k}^*[b] \in \mathbf{g}$, the attack is considered a success and $Exp^d_{A_{E_k,L}} = 1$. Thus, the dth order success rate can be defined as:

$$SR^d_{A_{E_k,L}} = Pr[Exp^d_{A_{E_k,L}} = 1].$$

In other words, the dth order success rate is defined as the probability that the target secret $\mathbf{k}^*[b]$ is ranked among the d first key guesses in the score vector. In the rest of the article, the dth order success rate is denoted SR^d.

2.2 Deep Neural networks

In this article, the focus is made on convolutional neural networks (CNN) and therefore the use of multilayer perceptrons in profiled side-channel analysis is not discussed. There exists articles which present multilayer perceptrons in more details such as [24, 2] .

2.2.1 Description of a neural network

A neural network is the representation of a sequence of linear and non-linear transformations applied to input values that results in an output which can vary depending on the problem at hand, *e.g.* classification, regression, *etc...* In addition, the transformations composing the network are organized using *layers* of *neurons*. Each neuron represents the combination of a linear and

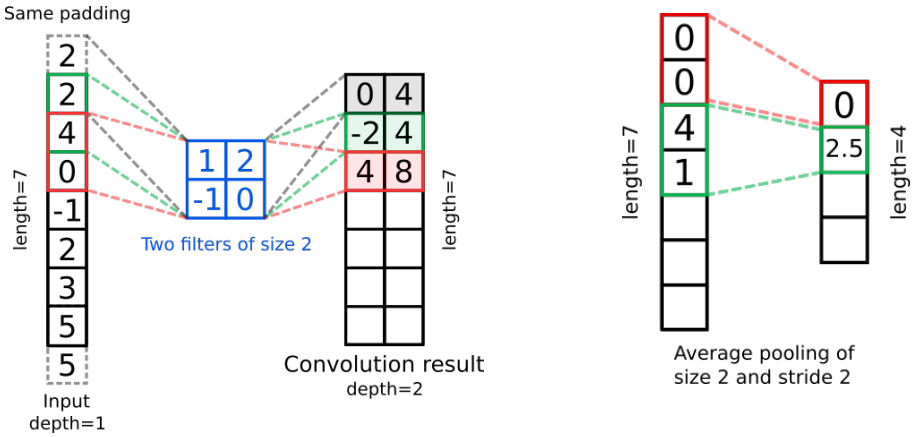
non-linear transformation. The output of each layer is formed by the output of all the neurons and is fed to the next layer as an input repeating until the last layer, which outputs the solution to the problem. Neural networks are composed with two kinds of parameters: the *hyperparameters* and the *trainable parameters*. The hyperparameters are the set of parameters that defines the architecture of the network, *e.g.* the number of neurons or the number of layers, and they will be mentioned in the next sections. The trainable parameters are the parameters updated during the learning process in order to perform a task. The two main trainable parameters are the *weights* and the *biases* associated to each neuron and layer. The weights are specific to each neurons and are thus learned independently from neuron to neuron while the bias is a layer-wide parameter that is used in the computation of every neuron of a layer.

2.2.2 Convolutional neural network

The specificity of CNNs is the use a *convolution operation* as illustrated in Figure 2a. The convolution is mathematical operation that processes the layer input with respect to a filter. Each convolutional layer has a given number of filters of fixed size. The number of filters and their sizes as well as the *stride* are all hyperparameters of the network. The size of the filters is equivalent to the number of weights they contain and the stride corresponds to the number of feature shifts when processing sequentially the filter over the input data. Each filter slides through the input, *i.e.* the same weights are applied to all the input, to generate the output which will correspond to the input of the next layer. When the size of the filters does not correspond exactly to the size of the input, a padding is applied to the data. In this application, a same padding is used. The padding consists in adding an additional features around the input in order to ensure that each application of a filter is centered around each original feature of the input. The size of the padding indicates the number of additional features added.

During the convolution operations, the computations are made locally with respect to the input. When transposed to the side-channel problem and the power consumption traces, this means that only points close together on the temporal scale are used in the computation for the first layer. This is what constitutes the linear transformation. To obtain the approximation power of the neural network, non-linearity is needed. This is handled by applying a non-linear function to the result of the convolution operation that is called the *activation function*. In the case of side channel analysis, this function is the *rectified linear unit* (ReLU) and is used in the rest of this article. It was shown in [2] to be more efficient than other activation functions in the side-channel context and is defined as follows:

$$\mathrm{ReLU}(x) = \begin{cases} 0, \text{ if } x < 0 \\ x, \text{ if } x \geq 0 \end{cases}$$



(a) Example of a convolution operation

(b) Example of an average pooling operation

Figure 2: Examples of a convolution operation using two filters of size 2, using a padding of one with repetition of the first value and with a stride of 1, and an average pooling operation of size 2 and stride 2.

In summary, to compute the output of a given neuron n of a layer l, the network uses the following formula:

$$X^{(l)}[n] = \mathrm{ReLU}\big(W_n^{(l)} * X^{(l-1)} + B^{(l)}\big),$$

where $W_n^{(l)}$ is the weight matrix of the filters of the layer l, $B^{(l)}$ the bias vector of the layer l, $*$ is the convolution operation and $X^{(l-1)}$ and $X^{(l)}$ are the outputs of the layers $l-1$ and l respectively.

In the example of Figure 2a, the size and number of filters is set to 2 and the stride to 1. Therefore, the dimension of the output is twice the one of the input. If not controlled, the dimension of the data would grow out of control. This is why a *pooling layer* is often used after a convolution layer. As illustrated in Figure 2b, the application of a pooling of size 2 results in reduction of the dimension by a factor 2. There are different kinds of pooling, *e.g.* average, min, max, used in different context. An average pooling is used in the rest of the article following the recommendations in [2]. It computes the average of the value within its window. Finally, in order to exploit the information extracted in the convolutional layers, *fully connected layers* can be used as the last layers of the network. They are layers of fully connected neurons which means that every neuron of the previous layer is connected to every neuron of the next.

The final objective of the network is to define a probability function for estimating the classification probability of the input. Therefore the activation function of the last layer is a softmax function as its output is the output of the network. It is projecting each input of the previous layer val-

ues between 0 and 1 while making sure they add up to 1. Given a vector $\mathbf{x}$ of n elements, the result of the softmax of $\mathbf{x}$ is $\text{Softmax}(\mathbf{x}) = (s_i)_{1 \leq i \leq n}$ where:

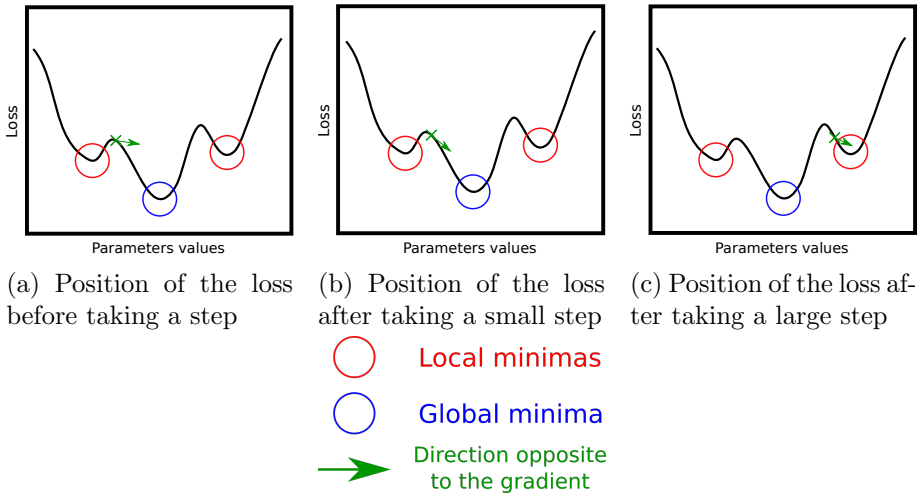
$$s_i = \frac{e^{\mathbf{x}[i]}}{\sum_{j=1}^{n} e^{\mathbf{x}[j]}}$$

In this case, the networks are used to perform classification of our data. Given a sample of our data, *e.g.* a power consumption trace $\mathbf{t}$, the network has to make a prediction on what class (or value), z, is associated with the trace. This is equivalent to say: the network takes as an input a trace $\mathbf{t}$ and returns the value z of the intermediate variable Z as the class with the highest value.

To summarize, the input of our network is a vector containing values of power consumption over time. Those values then go through linear transformations done using convolutional filters and non-linear transformation done using the ReLU function. Fully connected layers are then used and finally the prediction is made using the output of the softmax function. In our case, the number of classes is 256. This is due to the fact the sensitive variable targeted is the output of a substitution box of a symmetric encryption algorithm, *e.g.* the advanced encryption standard (AES). The information leaking is the value of one byte of the output of the substitution box and therefore the key is recovered byte per byte which can take 256 possible values. It is thus possible to use the network to perform a profiling side-channel attack and retrieve the key on the condition that the prediction is correct. Therefore the difficulty comes from training a network that is precise enough to retrieve the key.

2.2.3 Description of the training phase

As mentioned before a neural network is as a set of linear and non-linear transformations but the choice of the transformations is not random. The weights are indeed learned during a process called the *training phase*. The learning process can be seen as trials and errors made by the network when predicting the training examples in order to, step by step, reduce the errors it is making. During the training, the training data is split into two sets: the *training set* and the *validation set*. The training set contains the samples used to train the network while the validation set contains samples of training data that the network will never see during training and therefore can be used to evaluate the generalization performance of the network. The errors on the training and validation sets made by the network are computed using a function called the *cost function*, also known as *loss function*. This loss function is differentiable thus it is possible to compute its derivative with respect to each parameter of the network. Therefore, by changing the weights of the network using the backpropagation and the gradients of the



(a) Position of the loss before taking a step

(b) Position of the loss after taking a small step

(c) Position of the loss after taking a large step

Figure 3: Example of the shape of a loss function to minimize.

loss function, it can be minimized. Indeed, the direction of the gradients indicate how to maximize the loss so, by going in the opposite direction, a minimum can be found that is either local or global as illustrated in Figure 3a. This is equivalent to reducing the errors made by the network on the training samples. In this application, the cost function used is the categorical cross-entropy (CCE) defined as:

$$CCE(p, q) = -\sum_{x} p(x) log(q(x)),$$

where p is the true output, q the prediction of the network, *i.e.* the output of the softmax activation function, and x an input.

To learn the parameters of the network, the optimization of an objective function, defined as the average categorical cross entropy over the training sample, is required. This function is often completed with a regularization function as seen later on in the article. The optimisation of the objective function is done by a (stochastic) gradient descent approach. Thanks to the backward procedure, this optimization allows to change the parameters of the network accordingly. This change is controlled by an hyperparameter called the *learning rate* that has to be set when creating the network. The learning rate, if set correctly, prevents the changes applied to the weights from being too large and thus missing potential minimas of the loss function as can be seen in Figure 3c. The larger the learning rate, the larger the step in the opposite direction of the gradient is taken. Learning rates too high means higher chances of missing a minima but learning rates too low means slow optimization and risks of being stuck in bad local minimas as shown in Figure 3b.

During the training phase, the network makes a prediction on every sample of the training set and the weights of the network are updated depending on the errors made in the predictions. This process is called an *epoch*. Once the network is updated, the process is iteratively repeated using the samples from the training set. When the training set is too large though, the actual computations of going through the whole dataset and updating the network for every examples can be computationally too expensive. Therefore, the dataset is decomposed in smaller *batches* for which the loss of each examples is averaged in order to update the network. This is called *batch* or *mini-batch* training. The size of the batch of examples fed to the network is part of the hyperparameters chosen before the training.

Once the training phase starts, the stopping criterion used to stop the training phase has to be chosen with care. One way is to wait until the loss function reaches zero, *i.e.* the network makes no errors on the training set. However, this method has the drawback that, for most problems, it is not possible to know if such a value is reachable or how long it would take. It also leads to another common problem when training neural networks, the *overfitting* of the network. Overfitting comes from the fact that the network is able to predict the training data perfectly but is unable to generalize its knowledge to the validation data. It can originate from different sources. For example, if the network architecture is too complex to solve the problem, it will have the tendency to learn the training data perfectly to reach a loss of 0 but it does not guarantee good performance on the validation set. Therefore, it is possible that what the network is going to learn might help to reduce the error it is making on the training data but it will also perturb its prediction on the test data. It generally happens when the network trains for too many epochs. The training error can be reduced drastically but the training set is almost learned by heart, *i.e.* the network learns features from the input that are not relevant for generalization but help to predict this specific sample. Overfitting is often associated to the opposite problem, *underfitting*. Underfitting can happen for several reasons: the architecture of the network is not complex enough to approximate the underlying target function, the network has not seen the data enough to reduce its errors, or the training set is too small or not representative enough of the real data. Both of these problems can seriously impair the performances of the neural networks if not taken into account during the training therefore it is important to be able to detect them. To detect those phenomena, a comparison can be done between the performance of the network on samples from the training set and from the validation set. It takes place during the training phase at the end of an epoch and, if the average loss of the validation set starts increasing, the training can be stopped.

The metrics often used for the comparison are the *empirical risk* and the *accuracy* of the network. The empirical risk represents the average of the loss function over the training examples and the accuracy the proportion

of examples correctly classified by the network. In terms of side-channel analysis, the accuracy corresponds to a *simple power analysis* where only one trace is used to find the intermediate value, and by extension the key. This is the evaluation metric commonly used in the articles studying the application of deep learning to perform side-channel analysis. It has been shown to misrepresent the actual side-channel analysis performance of the networks [18].

2.2.4 Application to Side-Channel Analysis

In order to perform an attack, the attacker first needs to measure a physical quantity coming from the training device in order to create the training set. As he has control over the value manipulated by the device, the attacker knows the label associated with each trace and can train the network. The training process is described in Figure 4 and the resulting network is later used during the attack. Once the training is done, he measures the power consumption of the target device during the encryption of random plaintexts for which he knows the value with a fixed, unknown key and this constitutes the *attack set* or test set. The key recovery process is illustrated in Figure 5. The network can then generate prediction for every trace obtained and the byte with the highest value is selected to be the right candidate. Once the byte is predicted, using the public value of the plaintext, the adversary is able to recover the key byte used in the encryption. This is the ideal case where the network predicts the correct value every time but in practice it rarely happens. It mainly depends on the validation accuracy the network reaches at the end of its training. To be able to recover the key in one trace consistently, the attacker would need a network with a validation accuracy of 100% and hope that the distribution of the examples inside the validation set is identical to the one in the attack set. This scenario is very unlikely but the key byte can still be recover even though the accuracy of the network is close to that of a random prediction, *i.e.* around 0.4%, the equivalent of 1/256.

During the measurement of the traces, the adversary is able to control the value of the plaintext and thus he chooses it following a uniform distribution among all possible byte values. Therefore, the intermediate value follows a uniform distribution, whether the key is fixed (case of the target device) or the key follows a uniform distribution (case of the training device). This means that the classes are balanced, *i.e.* the number of examples per class is roughly the same for every class. Thus, training a specialized network to predict a specific class is not possible otherwise it will never be better than an average prediction. This paradigm is relatively different from the one usually found in other classification problems solved using neural networks. For example, in image classification or error detection, the network user is interested in being accurate for as many examples as possible since there is

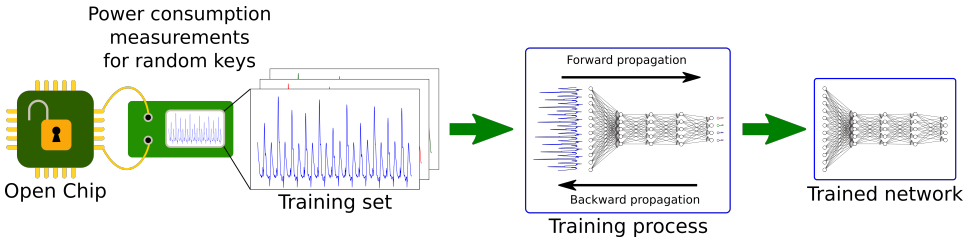


Figure 4: Training process.

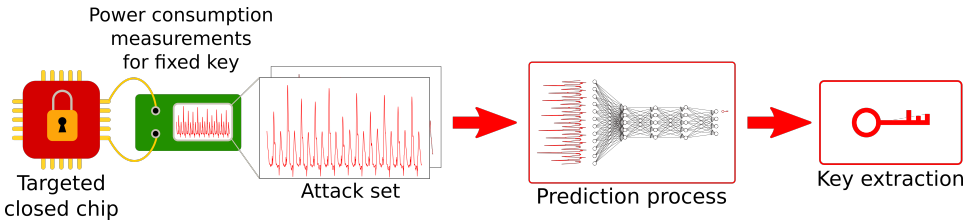


Figure 5: Key recovery.

no connection between the different images. On the contrary, in the case of side-channel analysis, all the traces in the attack set share the same key and therefore share a common value. This fact allows the attacker to accumulate information with several predictions and combine it to make a "meta prediction" on the value of the shared secret, *i.e.* the key. Therefore the network does not need to predict the correct intermediate value every time as long as the correct label is among the values with the highest predictions. The accuracy of a network is then a good indication that the network is working if and only if it has a high value. When the accuracy is low it is not possible to know if the network will perform well or not. According to Picek et al. [18], it is more relevant to use the success rate when a side-channel attacker wants to evaluate the performance related to his network. Indeed, contrary to the accuracy, the success rate is based on the accumulation of information over several traces. To facilitate the comparison of the performance of the network on the training set and on the validation set, it is possible to reorder the random keys in order to simulate attacks. This is described in the next section.

2.3 Performing attacks against random keys

Side-channel attacks are usually done against a fixed key as the goal of the attacks is to recover this specific value. The fact that the key varies for each traces in the training set creates a problem as typical attack scenarios are not possible anymore. However, this problem can be solved by reordering the key values. To illustrate this method, let us consider an AES computation. For an AES, the plaintext byte, $\mathbf{p}_i[b]$, is *xor*ed with the key byte, $\mathbf{k}_i^*[b]$,

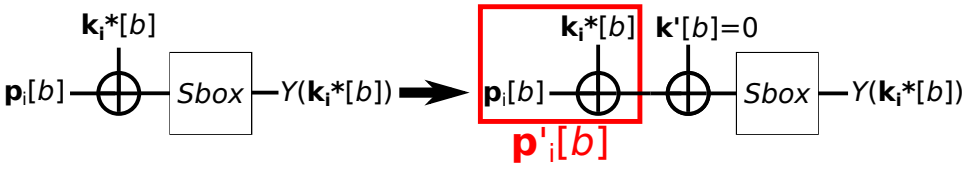


Figure 6: Illustration of the reordering the key. On the left is the normal method of considering the attack and on the right the alternate method using a reordered key.

as part of the *AddRoundKey* operation and the result is processed by the *SubBytes* operation, using the *Sbox*, which outputs the intermediate value targeted by the network, $Y(\mathbf{k}_i^*[b])$. This is illustrated on the left part of Figure 6. During the training phase, the value of the key is known for each traces i, therefore it is possible to consider the result of the *xor* between the key and the plaintext as the new plaintext, $\mathbf{p}'_i[b]$. This new plaintext is now *xor*ed with a new key value, $\mathbf{k}'[b] = 0$, and fed to the *Sbox*, as shown on the right part of Figure 6. The output of the *Sbox* stays the same as a *xor* with the value 0 does not change the *xor*ed value. By doing so, a regular attack can now be performed against this new key and the rank of $\mathbf{k}'[b]$ becomes the rank of the good key.

The results of the attacks on the validation and attack sets should be similar as they target the leak of the output of the *Sbox* which is not impacted by the reordering. To avoid this problem in the future, it can be interesting to keep a validation set using a fixed key when acquiring the training set. It would allow for an easier comparison between the results on the validation and attack sets.

2.4 Related work

Recent research articles focusing on the application of deep neural networks to side-channel analysis have most notably studied the problem of hyper-parameter tuning. In [28], Zaid *et al.* propose a way to determine a good network architecture by carefully setting up the number of layers, filters and the size of the filter. They focus on the overall architecture of the network and manage to reduce the size of a public network while improving its performances. This study is analyzed and further developed by Wouters *et al.* in [26] and Zaid *et al.* in [29] in which they mention that the use of batch normalization could allow to skip the normalization of the input traces. However, they do not go further in explaining the effect of batch normalization. In [30], Zhang *et al.* follow a similar approach and use a metric they introduce, the *cross-entropy ratio*, to validate their experiments. Contrary to these articles, we focus on applying machine learning techniques in order to improve the performances of a given architecture. In other words,

Table 1: Summary of the different methods used and explored for the improvement of the performances of neural networks in side-channel analysis.

Reference \ Improvement method	Architecture tuning	Data augmentation	Noise addition	Batch Norm.	Dropout	L2 reg.	Early Stopping
Zaid *et al.* [28] [29]	Explored	-	-	-	-	-	-
Wouters *et al.* [26]	Explored	-	-	-	-	-	-
Cagli *et al.* [4]	-	Explored	-	-	-	-	-
Picek *et al.* [18]	-	-	Explored	-	-	-	-
van der Valk *et al.* [23]	-	-	-	Used	Used	-	-
Masure *et al.* [15]	-	-	-	Used	-	-	-
Perin *et al.* [17]	-	-	-	-	Used	-	Explored
Li *et al.* [13]	-	-	-	-	Used	-	-
Weissbart *et al.* [24]	-	-	-	-	-	Used	-
Zhang *et al.* [30]	Explored	-	-	-	-	-	-
[this article]	-	-	-	Explored	Explored	Explored	Explored

our objective is not to design new models or to define the best composition of neurons. We rather consider deep learning techniques that keep the same model size. Indeed, adding a batch normalization layer represents generally a small increase of the original architecture (*e.g.* for the CNN_{bn} model, the increase corresponds to almost 0.00005% of the original architecture). Dropout and L2 regularization act directly on the value of the weights and not on the architecture, therefore our approach is complementary to the methodologies proposed in these articles.

Other works have used the aforementioned techniques in the context of side-channel analysis such as [23, 13, 15] for batch normalization, [18, 17] for dropout and [24] for L2 regularization. In another context, Cagli *et al.* [4] focus on the impact of *data augmentation*, a technique consisting in artificially increasing the amount of traces in the training set by creating new training traces by applying transformations to the original training traces. This method makes the network more robust to modifications in the input by focusing on the addition of a novel and diverse training examples.

We provide in Table 1 a summary of the different techniques used in the recent state of the art papers mentioned previously for improving deep learning-based side-channel analysis. The articles are separated depending on weather they offer detailed explanations on the techniques used (mentioned as Explored) or if they are only present to improve performances (mentioned as Used). Our article is the only one that provides a thorough study considering together batch normalization, L2 regularization and dropout. We provide in the following section a presentation of these techniques and their impacts on neural network training.

3 Batch Normalization and Regularization of Deep Learning Networks

We begin this section by a description of the principle of batch normalization in Section 3.1. Then, we introduce the L2-regularization considered in this paper in Section 3.2. Finally, Section 3.3 is devoted to the presentation of the dropout technique.

3.1 Batch Normalization

Normalization, or batch normalization, for deep neural networks was introduced by Ioffe *et al.* [8] in 2015. Their goal was to improve the learning phase of a network by reducing the *internal covariant shift* (ICS). The ICS represents the shift in the data distribution that happens after the update of the weights of a layer. Their hypothesis was that the layers were learning based on the distribution of the previous layer and were using that to treat the information. The goal of the batch normalization was to normalize the distributions of the output of the layers to prevent this phenomenon.

Normalization consists in taking a set of samples and modifying it to force the mean of the samples to 0 and its standard deviation to 1. It is a way to homogenize the distribution of the samples. This homogenization results in numerical values of the samples closer together but the proportional variations stay the same. The idea behind normalizing the data is to make the network focus more on the variations between samples rather than the numerical values of those examples. Without normalization, it is more difficult for the network to notice and use the variations since it focuses on the high numerical values first. Normalization also implies a more stable distribution of the data for each layer even after weights update. The data distribution of a layer is less affected by the weight update, therefore the next layer does not have to adapt to the new distribution and instead focuses on solving the problem. Indeed, when the gradient is computed to determine how to update the weights, it is under the assumption that the rest of the layers will stay the same.

The addition of the normalization process showed a gain in the performances of the network. This is why the batch normalization technique is included in most of the recent deep neural network architectures [3].

3.1.1 Normalization layer

Ioffe and Szegedy [8] introduced the normalization layer that performs the normalization of the data between the layers of the network. This layer is usually placed between the activation and the pooling layers in a convolutional block. It is applied on each batch of training examples and is also called *batch normalization*. Indeed, the scaling parameters will be unique

for each batch as the goal is to make the distribution the same for every batch. The normalization process for an input y_i of a batch is:

$$y_i^* = \gamma \frac{y_i - \mu}{\sigma + \epsilon} + \beta, \tag{1}$$

where μ is the mean of the batch, σ is the standard deviation and $\epsilon > 0$ is a small constant added for numerical stability. γ and β are trainable parameters used to scale and shift the distribution if needed. This reparametrization of the distribution of the input brought a side-effect, as studied in [20], the smoothing of the optimization landscape.

3.1.2 Smoothing of the loss landscape

Santurkar *et al.* [20] uncovered in their article another reason, and possibly the main reason, for the success of batch normalization. They show, by placing noise after the batch normalization layers, that the normalization has little effect over the ICS. They continue by studying the landscape of the loss function and conclude that, by making it smoother, the normalization allows for a better learning and the use of higher learning rate. Indeed, since there are less sharp changes in the loss, the optimization algorithm can take bigger steps without the fear of missing a minimum.

Batch normalization allows the network to learn faster and helps to regularize it a little but it can also lead to faster overfitting. To control this effect, regularization techniques can be applied to fully benefit from the normalization without the drawback of increased overfitting.

3.2 L2-Regularization

This section focuses on a technique used specifically for regularization called L_2 norm regularization or *weight decay* [11]. This technique consists in adding a regularization term to the value of the loss function. In the case of the L_2 norm, this term is the sum of the square of the weights of the layer controlled by a coefficient α. The objective function becomes:

$$O(\theta) = CCE + \alpha \sum ||W||^2,$$

where θ is the parameters of the network and W the weight matrix.

This term will grow larger as the value of the weights grow larger and smaller as the values get smaller. In practice, this means that the weights with small value have a low influence in the weight update contrary to the high weights. The penalty will thus focus on the large weights and control their values. To summarize, this regularization term encourages the weight to be close to zero without pushing them to become zero. The goal of this change is to prevent the network from relying only on the high weights to make its prediction as doing that would most likely lead to overfitting. This

is a way to reduce the overfitting of the network at the cost of a slower learning/convergence and is one reason why it is interesting to use it in combination with batch normalization. It is necessary to apply it when the performance of the network on the training set is good but the network has poor performance on the validation set at the end of the training.

Other forms of regularization exist and they use different values for the regularization term. The L_1 regularization, for example, replaces the sum of square by a sum of the absolute values of the weights. The effect is different as this will push the low weights towards zero while keeping only a few values relatively high. To compare the two forms of regularization, the L_2 will encourage a larger distribution of smaller weights while L_1 will push for a distribution of a few important weights with others at zero. Therefore L_1 regularization implies scarcity in the weights of the network.

3.3 Dropout

The previously mentioned techniques acted on the data flowing through the network and the values of the weights used in the computations. Dropout [21] has yet another kind of effect on the network. There are different types of dropout but the standard one is used to disable a given percentage of neurons randomly for each batch of samples. As shown in Figure 7, when applying a dropout with coefficient 0.5, half of the neurons of the layer are dropped. The activation of those neurons is set to zero and they are thus not used in the computation of the output. These neurons are chosen randomly for every batch. This results in a smaller network that performs the prediction and changes for every batch of samples. Moreover, as the computation of the gradient is an average over the training samples of a given batch, the backpropagation is not applied to the dropped neurons of this batch. One of the possible interpretation of dropout is that it is similar to averaging over the ensemble of smaller neural networks [12]. It was also shown that layers without dropout tend to have more dead neurons. A dead neuron is a neuron which, as the result of the training, has a fix activation. It means that it is stuck in this state and will not change no matter the training. On the other hand, layers using dropout have almost no dead neurons [16]. Another benefit of using dropout is the reduction of co-adaptation in the network. Co-adaptation happens when neurons depend too much on each other and as a result loose generalization power. For example, neurons could learn to correct mistakes from other neurons which slows down the learning process. Dropout prevents that as it makes the presence of neurons unreliable therefore it forces neurons to act more independently and reduce the overfitting coming from co-adaptation [7].

The effect of dropout in the convolutional layers more specifically has been studied in [16] by Park and Kwak. They exhibit that it helps filters to learn more informative features as well as improving the generalization

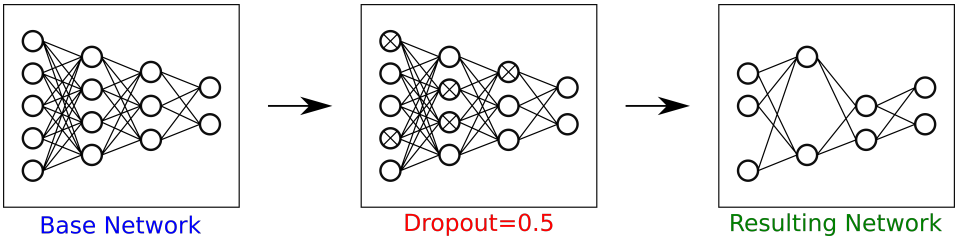


Figure 7: Effect of dropout on a fully-connected network.

performance of the network by adding noise to the output of the layers. Finally, they showed that applying more dropout the deeper the layer is leads to better performance. This is taken into account during our experimentation phase to tune the amount of dropout used in each layer. Those techniques are applied in the experimental section to observe the effect they have on a neural network used in the side-channel analysis context. For that, the starting architecture is the one used in the ASCAD database [2] and other articles of the literature, to allow for a fair comparison between the results.

Before presenting our experimental study on the impact of the three techniques described above, we present in the next section a metric tailored to side-channel analysis, introduced in [19], that we will use to evaluate the performance of neural networks.

4 $\Delta^d_{train,val}$: an evaluation metric for side-channel analysis

Following the remark in [18] about the lack of metrics to evaluate the training of networks for side-channel analysis, several articles were published trying solve this problem. In [23], van der Valk *et al.* use the bias-variance decomposition of the loss function to compare the performance of different architecture and perform some hyperparameters tuning. Unfortunately, they do not explore the early stopping aspect of the metric nor do they use a public architecture. Later, in [30], Zhang *et al.* use a more theoretical approach by computing what they call the *cross entropy ratio*. They apply it to compare the performances of different custom network and to try to find the best architecture. However, their metric does not seem to allow for a good comparison of networks trained on different datasets contrary to the one presented in [19], $\Delta^d_{train,val}$, that we selected for our experimental study.

This metric aims indeed at combining aspects from side-channel analysis and machine learning to create a relevant metric in order to evaluate both the training and the performance during the attack phase of a network. It is based on the success rate achieved using the network on the training and val-

idation set. The comparison of the results gives information on the internal state of the network, namely if the network is underfitting or overfitting.

4.1 $\Delta^d_{train,val}$: internal state detection

Let a *model* be the result function F of the training of an architecture for a given amount of epochs. $N^d_{train}(model)$ and $N^d_{val}(model)$ are defined as the minimal number of traces that a model needs in order to reach a dth-order success rate:

$$N^d_{train}(model) = min\{n_{train} \mid \forall n \geq n_{train},\ SR^d_{train}(model(n)) = 90\%\}$$

and,

$$N^d_{val}(model) = min\{n_{val} \mid \forall n \geq n_{val},\ SR^d_{val}(model(n)) = 90\%\}.$$

Information can be extracted from those values on how well the network generalizes beyond the training data, using the Euclidean distance. $\Delta^d_{train,val}$ is indeed obtained as follows:

$$\Delta^d_{train,val} = |N^d_{val} - N^d_{train}|.$$

The success rate is a well-known side-channel analysis metric, therefore $\Delta^d_{train,val}$ combines the machine learning and the side-channel approaches. It allows to evaluate any network used to perform side channel analysis. The computation of the metric can be done during the training, similarly to the accuracy and the loss, as it only needs the training and validation set. It also means that it is possible to visualize and detect the underfitting or overfitting of the network.

4.2 Detection of overfitting/underfitting

The evolution of $\Delta^d_{train,val}$ with respect to the internal state of a model are illustrated with three areas in Figure 8, showing an example of the evolution of $\Delta^1_{train,val}$ during the training of a network.

On this figure, the underfitting state of the network appears in the area on the left. It is characterized by a high value of $\Delta^d_{train,val}$ due to the fact that the network has not learned enough from the training data to generalize well on the validation traces, *i.e.* N^d_{val} is still large compared to N^d_{train}.

Next, in the middle area, the network is in a good trade-off state. It is able to perform well on both the validation and training sets which leads to a low value of $\Delta^d_{train,val}$. The network where $\Delta^d_{train,val}$ reaches its minimal value is kept as it is a sign of good generalization although without guaranteeing that it is optimal. This network is more robust to changes in the traces as it has not yet learned non-informative features.

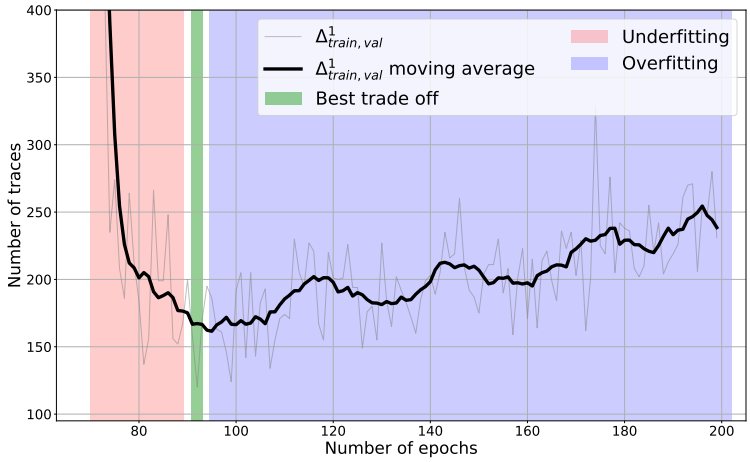


Figure 8: Evolution of $\Delta^1_{train,val}$ for different number of epochs. The plot of $\Delta^1_{train,val}$ is done using a moving average of size 10.

Past the state of good trade-off, the network enters a state of overfitting, represented in the right part of the figure. This state occurs when the network sees the training data too many times[1] and starts to learn features for the training traces by heart in order the reduce its errors. The resulting network is more sensitive to changes in the traces, *e.g.* a small difference in a noisy part of the traces can impact the prediction of the network. This leads to decreasing performance on validation and, on the other hand, better performance on the training traces. Thus, the value of $\Delta^d_{train,val}$ increases for the rest of the training.

4.3 $\Delta^d_{train,val}$: a suitable metric for early stopping

Early stopping consists in using a metric, *e.g.* the accuracy or the loss, to monitor the learning of the network in order to stop it when the metric is optimal meaning before the network starts to overfit. As mentioned in [6], early stopping has other effects on the network. It applies regularization without having to penalize weights and can be used along with other methods of regularization. In that way, the number of training epochs can be considered as an hyperparameter that is determined using the appropriate metric, which is, in this context, $\Delta^d_{train,val}$. All in all, it is recommended to perform early stopping when such a metric is available. The learning metrics are computed both on the training set and on a validation set to be able to properly tell whether or not the network performs well. The comparison between performance at training and on validation yields important information about the network. In this case, $\Delta^d_{train,val}$ gives us the information

[1] which can be of course reduced by the use of regularization of strong biases on the architecture.

needed to identify when to stop the training (see Figure 8). If the metric does not grow for a given number of epochs then it can be assumed that the optimal state is reached and the training can be stopped.

The next section contains the experimental results of the application of batch normalization and regularization to a public architecture CNN_{best}. $\Delta^1_{train,val}$ is used to evaluate the training of the resulting networks.

5 Experimental Results

5.1 Experimental Setup

For all the experiments presented in this section, the metrics are computed on neural networks during their training phase to evaluate their best capacity. The networks are trained using the ASCAD[2] variable key database, introduced in [2] to be a common database for researchers. The database contains a set of traces acquired using variable keys used for the training of the networks and a set using a fixed key to perform the attacks once the training is done. The device used to acquire the power consumption measurements is an 8-bit AVR ATMega8515 running an AES implementation secured against first order side-channel attacks. It means that the values manipulated during the encryption are masked, *i.e.* the bytes are *xor*ed with a mask randomly drawn. This is a common countermeasure used to prevent leakages directly related to the value of the sensitive variable. The dataset is composed of a training set of 200000 traces using a variable key and an attack set of 100000 traces using a fixed key both coming from the same device. The raw traces are composed of 100000 samples and from those points, 1400 are selected as they contain leakages of the mask and masked value of the third key byte in the first round. The leakage model associated with the traces is:

$$Y(\mathbf{k}^*) = Sbox(\mathbf{p}[3] \oplus \mathbf{k}^*[3]),$$

where $\mathbf{p}$ is the plaintext and $\mathbf{k}^*$ the correct key. The success rates are computed over 100 attacks to create $\Delta^1_{train,val}$. Each attack is limited to 5000 traces and attacks succeeding past 5000 traces are not considered successful. This is done to prevent a high redundancy in the traces used for the attacks on validation while keeping a large training set. The attacks performed on the variable key dataset are done using the method described in Section 2.3 while the ones against the fixed key dataset are done classically. The values of $\Delta^1_{train,val}$ as displayed in the figures are computed using an moving average with a window of size three. This is done to smooth out the curve so it is easier to see and interpret underfitting and overfitting and explains why the minimum of the curve is not always exactly at the minimal value of $\Delta^1_{train,val}$. The main advantage of using this database is to compare the

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

results obtained to the ones presented in the ASCAD reference article [2] and other articles using this database.

5.1.1 Presentation of the starting network

The starting neural network comes from the ASCAD database article. In this article, after doing some hyper-parameter optimization, they found a good configuration. They named this architecture CNN_{best}. It is composed of 5 convolutional layers and 2 fully connected layers using ReLU activation function as presented in 2.2.2, filters of size 11 and the quantity of filters doubling for each layer from 64 up to 512. There are also average pooling layers after each convolutional layer. A summary of this network can be found in Table 4.

5.1.2 Presentation of the datasets

The datasets used in the experiments are subsets of the ASCAD variable key dataset for training and validation and of the ASCAD fixed key dataset for the attack set. They are decomposed into three subsets: Desync0, Desync50 and Desync100. Desync0 contains samples that are synchronized, *i.e.* every point in the samples corresponds to the same instant in the execution of the algorithm. Desync50 and Desync100 are composed of desynchronized samples. This means that when the 1400 points are selected from the raw traces, a random shift between 0 and 50 or 0 and 100 is applied to the trace. Figure 9 illustrates the effect of desynchronization. On the figure, one trace is synchronized, *i.e.* the shift equals 0, and the other is shifted by 100 points. As a consequence, power spikes that occurs at the same time in the algorithm end up far appart. The shift is randomly drawn for every traces and applied to every points. Desynchronization is a common countermeasure to side-channel analysis and it was shown in [2] that the neural networks were still able to learn and perform attacks when using desynchronized traces. Therefore it is interesting to study how the batch normalization and regularization will react to this more difficult problem.

5.2 Study of the accuracy of CNN_{best}

A good way to start the study of the network is to look at its accuracy. It was previously mentioned that in this application, accuracy can be an indicator of the network performances on the attack set but only when the accuracy is high on both the training and validation sets. Figure 10 shows the training and validation accuracy for networks trained with training sets containing 50000, 100000 and 190000 traces on Desync0. It appears on Figure 10a that, the more traces in the training set, the faster the training accuracy will start increasing and the higher it will end up. Meanwhile, Figure 10b represents the validation accuracy of the same networks. Throughout the

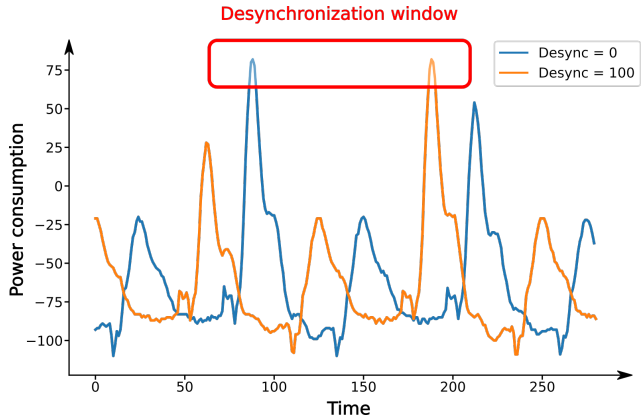


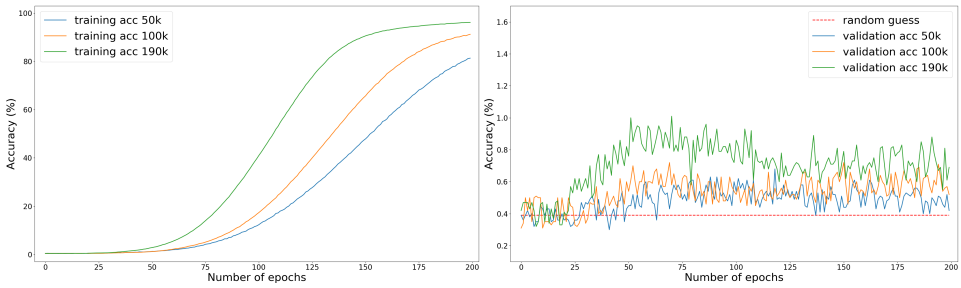
Figure 9: Power consumption traces from Desync0 and Desync100.

training, the validation accuracies barely increase above the accuracy of a random guess and reach their maximum values at epoch 118, 69 (tied with 150) and 70 for the training sets of size 50000, 100000 and 190000 samples respectively. It still appears that the more traces in the training set, the higher the validation accuracy will be but it is still close to the values of the other network. The gap between the accuracy on the training set and the accuracy on the validation set is quite substantial. It is safe to assume that the performance of the attacks done on the training set will be good but it is hard to say anything about the performances on the validation set and after that on the attack set. Indeed with a validation accuracy about twice the value of a random prediction, not much can be deduced on the performance of the attacks. Next, Figures 11 and 12 show the training and validation accuracies for the datasets Desync50 and Desync100 respectively. The training accuracy for those dataset start to increase faster than for Desync0 and end up close to 100% no matter the number of traces in the training set. It seems that the added difficulty of the datasets pushes the networks to learn the training examples perfectly. As expected, this also influences the validation accuracy which is much closer to the accuracy of a random prediction. With those results, it is safe to assume that the networks will have good attack performance on the training set but no conclusion can be reached on the performance they will have on the attack set. The next section uses the $\Delta^1_{train,val}$ metric to solve this uncertainty.

5.3 Study of $\Delta^1_{train,val}$ applied to the different datasets

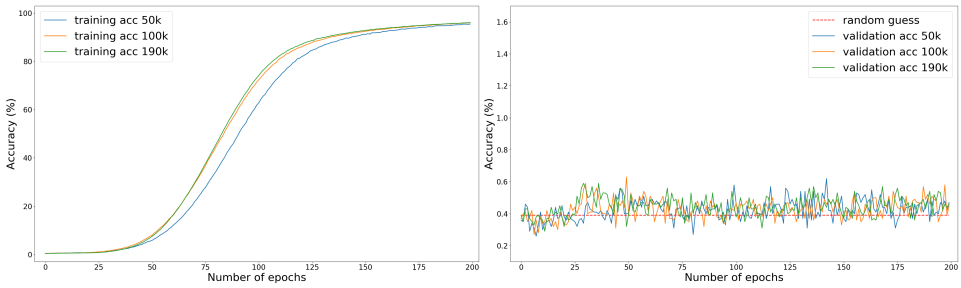
5.3.1 CNN_{best}

Figure 13 shows the evolution of $\Delta^1_{train,val}$ for CNN_{best} on Desync0 and different sizes of training set. This figure shows that the more traces contained in the training set, the faster the network will converge towards a good so-



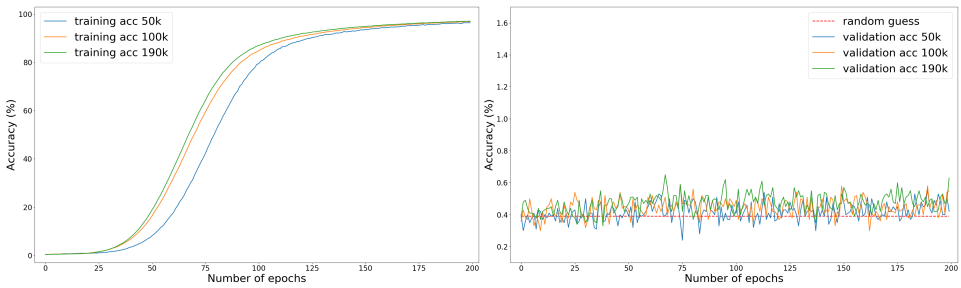
(a) Training accuracy (b) Validation accuracy

Figure 10: Evolution of training accuracy and validation accuracy of CNN_{best} during training for different sizes of training set on Desync0.



(a) Training accuracy (b) Validation accuracy

Figure 11: Evolution of training accuracy and validation accuracy of CNN_{best} during training for different sizes of training set on Desync50.



(a) Training accuracy (b) Validation accuracy

Figure 12: Evolution of training accuracy and validation accuracy of CNN_{best} during training for different sizes of training set on Desync100.

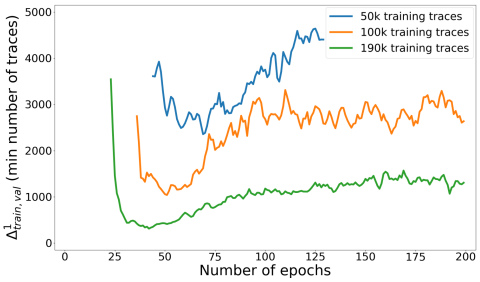
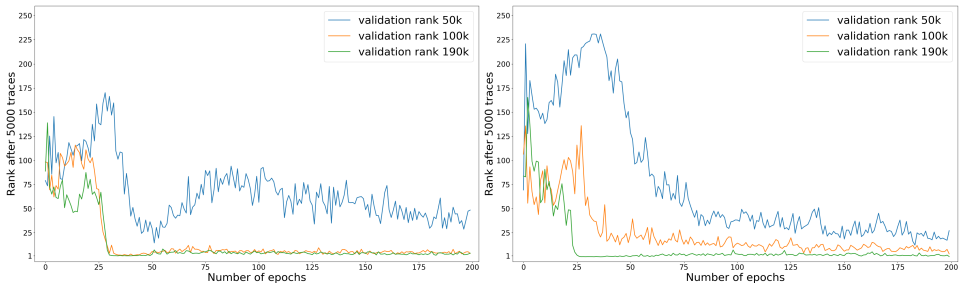


Figure 13: Evolution of $\Delta^1_{train,val}$ of CNN_{best} for different sizes of training set on Desync0.

lution and the better the performance will be. Indeed, the minimal value of $\Delta^1_{train,val}$ when using 50000 training traces is 2329, reached after 67 epochs, while the minimals for 100000 and 190000 training traces are 910 and 398, reached after 49 and 44 epochs. It shows that letting the network train for 75 epochs, as done in [2], is too much as the network starts to overfit. There is an improvement of both the performance on training and validation. In that sense, the addition of traces in the training set acts as a regularization for the network as seen on Figure 13 where $\Delta^1_{train,val}$ shows less overfitting for 100000 and 190000 training traces. Even though the network is able to reach good performance using 190000 training traces by needing around 500 traces to recover the key byte, as soon as the problem becomes more complex, the performance heavily drops. Figure 14 displays the evolution of the rank after using 5000 traces throughout the training on both Desync50 and Desync100. For those networks, $\Delta^1_{train,val}$ cannot be computed with less than 5000 traces as the networks are not able to reach a success rate of 90% on the validation traces as opposed to their performance on the training traces where they reach such a success rate after only a few epochs. This added difficulty leads to more overfitting as the performance on the training set is stable or even better in the case of 100000 and 190000 traces. This is most likely due to the fact that the complexity of the network allows it to learn the training set by heart and thus it is not affected by the desynchronization. Indeed, by focusing on features only characteristic to each trace, it is able to predict them as if there were no desynchronisation. Despite this, the networks are still able to reach good ranks on the validation data, as seen on Figure 14, especially when using 190000 training traces for which the networks almost obtain an average rank of 1 at 35 and 37 epochs for Desync50 and Desync100 respectively. Those results show us that the addition of traces to the training set can counteract the effect of overfitting but in a marginal way. The next section focuses on the effect of batch normalization on the performance of the network in similar scenarios.

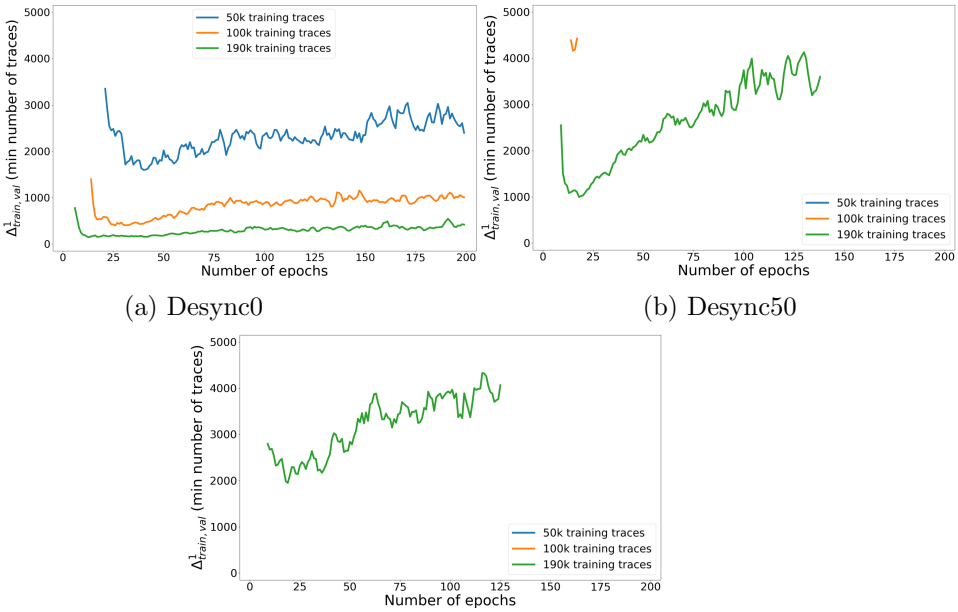


(a) Desync50 (b) Desync100

Figure 14: Evolution of the rank after 5000 validation traces during the training of the network CNN_{best} for different sizes of training set.

5.3.2 CNN_{bn}

Figure 15 represents the evolution of $\Delta^1_{train,val}$ for the network CNN_{bn} on Desync0, Desync50 and Desync100 for different number of traces in the training set. The architecture of this network can be found in Table 5. It has batch normalization layers between the activation function and the pooling layer. Compared to CNN_{best}, there is a significant improvement of the performance of the network on the validation set. On Figure 15a, $\Delta^1_{train,val}$ converges faster towards a lower value and is more stable throughout the rest of the training. Indeed, with only 14 epochs, the network manages to reach similar training and validation performance, as illustrated by the fact that the value of $\Delta^1_{train,val}$ reaches 116. Figure 15b shows the same metric but on the dataset Desync50. In this more difficult context, the effect of batch normalization allows the network to obtain a success rate greater than 90% for the training sets of size 100000 and 190000. The value of $\Delta^1_{train,val}$ of the latter network is minimal at 899 after 19 epochs. On the other hand, the shape of $\Delta^1_{train,val}$ indicates that overfitting occurs faster than for Desync0 and has a greater effect on the performance on validation. Finally, the results for the dataset Desync100 are displayed on Figure 15c. This time, the use of the entire training set is needed to be able to compute $\Delta^1_{train,val}$. The performance are lower than before, reaching 1793 after 19 epochs. The effect of overfitting is more pronounce as it becomes impossible to compute $\Delta^1_{train,val}$ after epoch 125. Despite the more difficult context of Desync50 and Desync100, Figure 16, which shows the evolution of the rank using 5000 traces for Desync50 and Desync100, indicates that the rank is close to one throughout the training for all sizes of training set which is a sign that the network is still able to generalize well even though it is not enough to perform successful attacks. In the end, the addition of batch normalization greatly improves the learning time of the network and reduces the number of epochs needed to reach its best performances. However, the overfitting happens faster and has more effect on the performance on



(a) Desync0

(b) Desync50

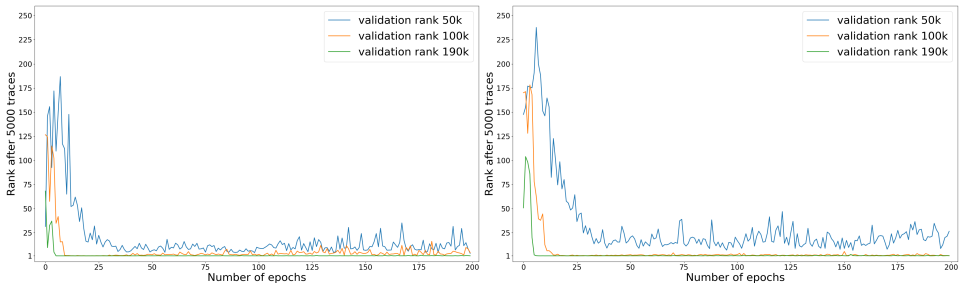
(c) Desync100

Figure 15: Evolution of $\Delta^1_{train,val}$ of CNN_{bn} for different sizes of training set and desynchronization.

validation as can be seen on the evolutions of $\Delta^1_{train,val}$. The addition of more traces to the training set has a regularization effect, similar to CNN_{best}, on the network as it slows down the convergence of N_{train}, and therefore accelerate the convergence of $\Delta^1_{train,val}$, as well as reducing the number of epochs needed to obtain good performance on validation. Furthermore, the batch normalization improves the overall performance of the network for all amounts of desynchronization even if there is still a large gap between the training and validation performances for Desync50 and Desync100. The goal of the regularization is now to reduce this gap and the next section will show its effect.

5.3.3 $\text{CNN}_{\text{bnreg}}$

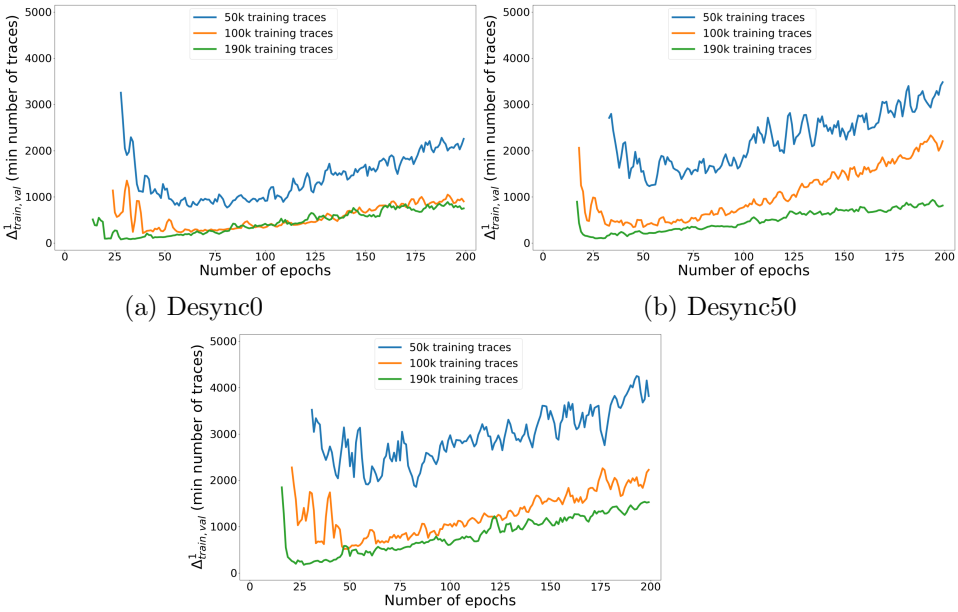
Figure 17 shows the evolution of $\Delta^1_{train,val}$ for the network $\text{CNN}_{\text{bnreg}}$ when trained and applied on different training sets with increasing amount of traces and desynchronisation. This network contains batch normalization layers as CNN_{bn} and both dropout and L_2 regularization. Its architecture is summarized in Table 6. The parameters of the regularization were determined experimentally and the best setup was kept. The values considered for the dropout and L_2 on the first two convolutional layers were between 0 and 0.3. As none of the values showed improvement, they were set to 0. The same thing happened with the fully-connected layers therefore no



(a) Desync50 (b) Desync100

Figure 16: Evolution of the rank after 5000 validation traces during the training of the network CNN_{bn} for different sizes of training set.

regularization is applied on them. For the three convolutional layers left, the values of dropout tested were between 0 and 0.8 and 0 and 0.3 for the L_2 regularization. The best performances were found when applying 0.5, 0.6 and 0.7 of dropout to the third, forth and fifth layers respectively. It means dropping 50%, 60% and 70% of the neurons in those layers. In addition, 0.2, 0.3 and 0.3 of L_2 was applied to the same layers. In Figure 17a, it appears that the performance of $\text{CNN}_{\text{bnreg}}$ are about the same as CNN_{bn} for all sizes of training set on Desync0. The only difference is that $\text{CNN}_{\text{bnreg}}$ needs to train for more epochs to reach its best performance after 29 epochs with a value of $\Delta^1_{train,val}$ of 40 when using 190000 training traces. The improvement brought by the regularization on top of the batch normalization starts to appear in Figure 17b in which the network is trained and performs the attacks on Desync50. In a desynchronized context, the network is still able to perform attacks in less than 5000 traces for all sizes of training set which was not the case for CNN_{bn}. Even though the performance of the network is reduced, it still obtains a value of $\Delta^1_{train,val}$ of 52 at epoch 26 when using the full training set. Finally, in Figure 17c, the addition of more desynchronization in Desync100 only amplifies this effect. On this dataset, the best network reaches a value of $\Delta^1_{train,val}$ of 111 at epoch 25 using 190000 training traces. Once again, the addition of more desynchronization increases the overfitting of the network but, as for Desync50, it is less pronounced than for CNN_{bn} and mainly present for the largest training set. However, it remains that as the network continues to train past its optimal performance, and with the additional effect of the desynchronization, $\Delta^1_{train,val}$ rapidly increases. It is therefore important to know when to stop otherwise the performance on validation might rapidly decrease. Another effect of the regularization, in desynchronized context, is to maintain similar performance on the validation even when the desynchronization increases. This means that the regularization allows the network to extract more information from the training samples and therefore to better generalize. It is



(a) Desync0

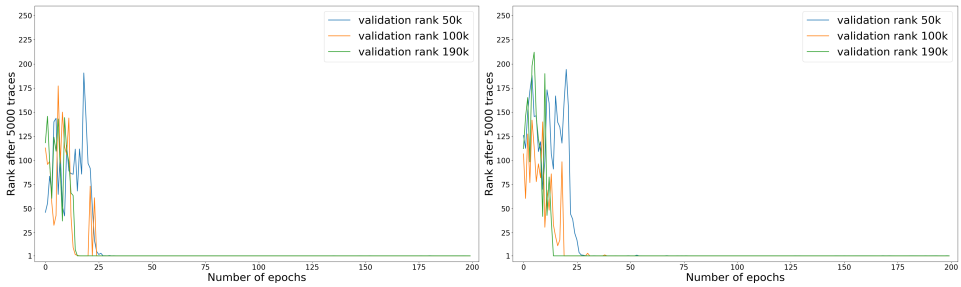
(b) Desync50

(c) Desync100

Figure 17: Evolution of $\Delta^1_{train,val}$ for $\text{CNN}_{\text{bnreg}}$ during training for different desynchronization levels.

harder to see this phenomenon on Figure 18, which shows the evolution of the rank after 5000 traces for $\text{CNN}_{\text{bnreg}}$ on Desync50 and Desync100, hence the need to focus on $\Delta^1_{train,val}$.

A summary of the results of the networks trained using 190000 traces can be found in Table 2. It also includes the performance, in number of traces needed to reach a success rate of 90%, of the networks on the attack set denoted N^*_a. As mentioned in 2.3, the value of N^*_a is close to the value of N^1_{val} for most networks. Therefore, N^1_{val} is a good indicator of the performance of the networks on the attack set. Using the metric $\Delta^1_{train,val}$ to perform early stopping, the performance of CNN_{best} were improved by 42.0% on Desync0 and the number of epochs was reduced by 41.3%. However, the early stopping did not help on Desync50 and Desync100 as the network was never able to reach a success rate of 90% in less than 5000 traces. On the other hand, the application of batch normalization greatly improved the performance of the network by reducing the number of traces needed to obtain 90% of successful attacks on Desync0 by another 46.5%. In addition, the learning was even faster as the best trade-off was obtained after only 14 epochs, which is a gain of 68.2% in number of epochs. This improvement also led to the possibility of successfully attacking the sets Desync50 and Desync100. Finally, the use of regularization led to a reduction of the number of traces needed to attack Desync50 and Desync100 by respectively 70.6% and 81.7% at the



(a) Desync50 (b) Desync100

Figure 18: Evolution of the rank after 5000 validation traces during the training of the network $\text{CNN}_{\text{bnreg}}$ for different sizes of training set.

cost of increasing the number of training epochs.

To conclude, the empirical results presented here confirm the importance of applying the batch normalization technique as well as to properly regularize the network via, for example, dropout and weight decay. The application of batch normalization allows the network to learn faster as well as to improve its performance on the validation set. This is not enough, though, to obtain good performance in the more complex context of desynchronized traces. In this context, the faster learning also leads to more overfitting. This is where the effect of regularization appears. Indeed, $\text{CNN}_{\text{bnreg}}$ shows more resilience against desynchronization even if the performance decreases a little. A summary of the best results for the different networks can be found in Table 2. Finally, the desynchronized context illustrates the need to know when to stop the training as the networks trained on desynchronized traces overfit faster than synchronized ones. If not controlled properly, this can lead to a significant decrease in performances during the attack phase.

5.4 Comparison with other methods used in state-of-the-art

Table 3 compares the attack success, in terms of minimum number of traces required to perform the attack, between state-of-the-art networks and the proposed work. Recall that, in our experimental analysis, we have decided to focus on the ASCAD variable key dataset which is known to be more difficult to attack than the fixed key one according to Wu *et al.* [27]. Before us, only two articles [27, 17] published results on this variable key dataset and our network is the best performing using only batch normalization and carefully tuned regularization techniques. We can also note that these results are also significantly better than the ones from CNN_{best} when applied to the variable key dataset, as we have seen in Section 5.3.1. This shows the importance and the benefit of the techniques studied in this paper. For the sake of completeness, we also discuss the scope of the results obtained on the fixed key version of the dataset. While the results of the two datasets

Table 2: Summary of the results for CNN_{best}, CNN_{bn} and $\text{CNN}_{\text{bnreg}}$ in terms of $\Delta^1_{train,val}$, N^1_{val}, N^*_a and number of epochs for a training set size of 190000 traces.

Networks	Reference	Number of epochs	Desync	$\Delta^1_{train,val}$	N^1_{val}	N^*_a	Improvement in number of traces and epochs compared to CNN_{best}	Improvement in number of traces and epochs compared to [2]	Improvement in number of traces and epochs compared to CNN_{bn}
CNN_{best}	[2]	75	0	972	935	1275	-	-	-
			50	-	-	-	-	-	-
			100	-	-	-	-	-	-
	[this article]	44	0	398	542	589	-	N^1_{val}: **-42**% N^*_a: **-53**% Epoch: **-41**%	-
			50	-	-	-	-	-	-
			100	-	-	-	-	-	-
CNN_{bn}	[this article]	14	0	116	290	228	N^1_{val}: **-46**% N^*_a: **-61**% Epoch: **-68**%	N^1_{val}: -69% N^*_a: **-82**% Epoch: -81%	-
		19	50	899	927	964	-	N^1_{val}: - Epoch: -74%	-
		19	100	1793	1805	3333	-	N^1_{val}: - Epoch: -74%	-
$\text{CNN}_{\text{bnreg}}$	[this article]	29	0	40	244	150	N^1_{val}: **-55%** N^*_a: **-74**% Epoch: **-34%**	N^1_{val}: -74% N^*_a: **-88**% Epoch: -61%	N^1_{val}: -16% N^*_a: **-34**% Epoch: +107%
		26	50	52	273	301	-	N^1_{val}: - Epoch: -65%	N^1_{val}: **-70**% N^*_a: **-68**% Epoch: +36%
		25	100	111	330	347	-	N^1_{val}: - Epoch: -66%	N^1_{val}: **-81**% N^*_a: **-89**% Epoch: +31%

can obviously not be compared directly, we can nevertheless extract some common lines. First of all, lots of the state-of-the-art methods have been evaluated on this fixed key dataset and, among the best results obtained for the different settings, we can cite the ones by Wu *et al.* [27], Zaid *et al.* [28], Won *et al.* [25] and Wouters *et al.* [26]. As a comparison, these results are significantly better than the ones obtained with the original architecture proposed by Benadjila *et al.* in [2]. This improvement behavior is in the same order of magnitude as the one we obtained with our techniques for the variable key dataset. Since some of these previous methods have been specifically tuned for the fixed key dataset, our behavior on the variable key one indicated that our approach tends to achieve comparable results with other state of the art approaches.

Other works [4, 9] conducted experiments using data augmentation. It consists in artificially increasing the training set by adding more difficult examples obtained through applying transformations to the original ones. In [4], Cagli et al. obtain a network that is more efficient and more robust to some shifting deformation and add-remove deformation. The first one corresponds to a desynchronization effect similar to the one applied in the Desync50 and Desync100 datasets and the second one to a clock jitter effect that modifies the signal by adding and removing random time samples. Our experiments show a reduction of the impact of desynchronization with the

Table 3: Summary of the results of state-of-the-art networks on ASCAD datasets expressed in number of attack traces needed to perform a successful attack.

Reference \ Dataset	ASCAD variable key			ASCAD fixed key		
	D0	D50	D100	D0	D50	D100
Wu *et al.* [27]	1000	-	-	80	-	-
Perin *et al.* [17]	~ 180	-	-	-	-	-
Won *et al.* [25]	-	-	-	-	-	190
Wouters *et al.* [26]	-	-	-	-	~ 200	~ 300
Zaid *et al.* [28]	-	-	-	191	244	270
Robissout *et al.* [19]	-	-	-	802	-	-
Kim *et al.* [9]	-	-	-	>500	-	-
Benadjila *et al.* [2]	1275	>5000	>5000	1151	>5000	>5000
[this article]	150	301	347	-	-	-

application of regularization. In addition, the two methods apply regularization at different levels. The data augmentation regularizes the network by modifying the training set while the dropout and weight decay impacts the network directly. Therefore these techniques are actually complementary and their combination could lead to even better results. Another kind of data augmentation is explored in [9] in which Kim et al. add white noise directly to the traces in the training set. The goal is as previously to improve the training of the network to reduce the number of traces needed in the attacks. The results showed in [9] on the ASCAD fixed key dataset indicate better performance compared to the base CNN_{best} network. However, they are outperformed by the networks of Zaid *et al.* and Wu *et al.* but similarly to the data augmentation in [4], it can be combined to other regularization techniques that act at the network level.

In conclusion, the method to use to improve the performances of neural network for side-channel analysis depends on different constrains presented to the attacker, such as time and computational power for example. Having a common database allows for a good comparison between the different architectures and improvement techniques. It also allows us to emphasize the importance of proper regularization of neural network through the use of early stopping and techniques like dropout and weight decay. Indeed, using them, we are able to reach state-of-the-art level of performances. It also opens the question of the combination of those different techniques as we see that they can be complementary.

6 Conclusion

The study of deep neural networks applied to side-channel analysis has seen great interest in the past years. A lot of work is done trying to understand the learning process of those networks in order to improve their performances. There are still some problems that need to be addressed such as finding a suitable metric to evaluate the networks and prevent the overfitting phenomenon. This article tries to answer the first question by using a side-channel dedicated metric to optimize the training and perform early stopping on $\mathrm{CNN}_{\mathrm{best}}$, an open architecture used by the community. It leads to a gain in performance of 42% and a reduction of the number of epochs of 41% but also exposes the overfitting of the network. The application of batch normalization as well as dropout and weight decay, commonly found in the machine learning community, aims at reducing this effect. It allows for a significant reduction of the overfitting and thus an improvement of the performances. In a context with no desynchronization, the number of traces needed to attack the key is reduced by 55% and the epochs by 34%. However, the effect of those techniques is more significant in desynchronized contexts. Indeed, where before the attacks were not successful, the proposed network $\mathrm{CNN}_{\mathrm{bnreg}}$ is able to attack desynchronized traces using less traces than $\mathrm{CNN}_{\mathrm{best}}$ on synchronized ones. This study shows the need to perform early stopping and to properly regularize the neural networks when performing side-channel analysis to prevent overfitting and, as a consequence, perform attacks using less traces.

References

[1] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM side—channel(s). In B. S. Kaliski, ç. K. Koç, and C. Paar, editors, Cryptographic Hardware and Embedded Systems - CHES 2002, pages 29–45, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[2] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas. Deep learning for side-channel analysis and introduction to ascad database. Journal of Cryptographic Engineering, 10:163–188, 11 2019.

[3] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. SIAM Review, 60:223–311, 2018.

[4] E. Cagli, C. Dumas, and E. Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures. In W. Fischer and N. Homma, editors, Cryptographic Hardware and Embedded Systems – CHES 2017, pages 45–68, Cham, 2017. Springer International Publishing.

[5] S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems, CHES '02, page 13–28, Berlin, Heidelberg, 2002. Springer-Verlag.

[6] I. Goodfellow, Y. Bengio, and A. Courville. Deep Learning. MIT Press, 2016. `http://www.deeplearningbook.org`.

[7] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. 2012.

[8] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In F. Bach and D. Blei, editors, Proceedings of the 32nd International Conference on Machine Learning, volume 37 of Proceedings of Machine Learning Research, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[9] J. Kim, S. Picek, A. Heuser, S. Bhasin, and A. Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019(3):148–179, May 2019.

[10] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, Advances in Cryptology — CRYPTO' 99, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[11] A. Krogh and J. A. Hertz. A simple weight decay can improve generalization. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, Advances in Neural Information Processing Systems 4, pages 950–957. Morgan-Kaufmann, 1992.

[12] A. Labach, H. Salehinejad, and S. Valaee. Survey of dropout methods for deep neural networks. ArXiv, abs/1904.13310, 2019.

[13] H. Li, M. Krček, and G. Perin. A comparison of weight initializers in deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2020/904, 2020. `https://eprint.iacr.org/2020/904`.

[14] H. Maghrebi, T. Portigliatti, and E. Prouff. Breaking cryptographic implementations using deep learning techniques. In C. Carlet, M. A. Hasan, and V. Saraswat, editors, Security, Privacy, and Applied Cryptography Engineering, pages 3–26, Cham, 2016. Springer International Publishing.

[15] L. Masure, C. Dumas, and E. Prouff. A comprehensive study of deep learning for side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(1):348–375, Nov. 2019.

[16] S. Park and N. Kwak. Analysis on the dropout effect in convolutional neural networks. In S.-H. Lai, V. Lepetit, K. Nishino, and Y. Sato, editors, Computer Vision – ACCV 2016, pages 189–204, Cham, 2017. Springer International Publishing.

[17] G. Perin, I. Buhan, and S. Picek. Learning when to stop: a mutual information approach to fight overfitting in profiled side-channel analysis. IACR Cryptol. ePrint Arch., 2020:58, 2020.

[18] S. Picek, A. Heuser, A. Jovic, S. Bhasin, and F. Regazzoni. The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019(1):209–237, Nov. 2018.

[19] D. Robissout, G. Zaid, B. Colombier, L. Bossuet, and A. Habrard. Online performance evaluation of deep learning networks for side-channel analysis. In Constructive Side-Channel Analysis and Secure Design - 11th International Workshop, COSADE 2020, October 5-7, 2020, Proceedings, 2020.

[20] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. How does batch normalization help optimization? In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, Advances in Neural Information Processing Systems 31, pages 2483–2493. Curran Associates, Inc., 2018.

[21] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1):1929–1958, 2014.

[22] F.-X. Standaert, T. G. Malkin, and M. Yung. A unified framework for the analysis of side-channel key recovery attacks. In A. Joux, editor, Advances in Cryptology - EUROCRYPT 2009, pages 443–461, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[23] D. van der Valk and S. Picek. Bias-variance decomposition in machine learning-based side-channel analysis. Cryptology ePrint Archive, Report 2019/570, 2019. `https://eprint.iacr.org/2019/570`.

[24] L. Weissbart, S. Picek, and L. Batina. On the performance of multilayer perceptron in profiling side-channel analysis. 2019. `https://eprint.iacr.org/2019/1476`.

[25] Y.-S. Won, D. Jap, and S. Bhasin. Push for more: On comparison of data augmentation and smote with optimised deep learning architecture for side-channel. Cryptology ePrint Archive, Report 2020/655, 2020. `https://eprint.iacr.org/2020/655`.

[26] L. Wouters, V. Arribas, B. Gierlichs, and B. Preneel. Revisiting a methodology for efficient cnn architectures in profiling attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(3):147–168, Jun. 2020.

[27] L. Wu, G. Perin, and S. Picek. I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis. Cryptology ePrint Archive, Report 2020/1293, 2020. https://eprint.iacr.org/2020/1293.

[28] G. Zaid, L. Bossuet, A. Habrard, and A. Venelli. Methodology for efficient cnn architectures in profiling attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(1):1–36, Nov. 2019.

[29] G. Zaid, L. Bossuet, A. Habrard, and A. Venelli. Understanding methodology for efficient cnn architectures in profiling attacks. Cryptology ePrint Archive, Report 2020/757, 2020. https://eprint.iacr.org/2020/757.

[30] J. Zhang, M. Zheng, J. Nan, H. Hu, and N. Yu. A novel evaluation metric for deep learning-based side channel analysis and its extended application to imbalanced data. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(3):73–96, Jun. 2020.

A Networks

Table 4: Network hyperparameters for CNN_{best} [2].

Layer type	Hyperparameters
Trace input	1400
Convolution 1D	Filter = 64, Filter length = 11, Padding = Same, Activation = ReLU
Average Pooling	Pool length = 2
Convolution 1D	Filter = 128, Filter length = 11, Padding = Same, Activation = ReLU
Average Pooling	Pool length = 2
Convolution 1D	Filter = 256, Filter length = 11, Padding = Same, Activation = ReLU
Average Pooling	Pool length = 2
Convolution 1D	Filter = 512, Filter length = 11, Padding = Same, Activation = ReLU
Average Pooling	Pool length = 2
Convolution 1D	Filter = 512, Filter length = 11, Padding = Same, Activation = ReLU
Average Pooling	Pool length = 2
Flatten	-
Fully-connected	Neurons = 4096
Fully-connected	Neurons = 4096
Output	Softmax: 256 classes

Table 5: Network hyperparameters for CNN_{bn}.

Layer type	Hyperparameters
Trace input	1400
Convolution 1D	Filter = 64, Filter length = 11, Padding = Same, Activation = ReLU
Batch Normalization	-
Average Pooling	Pool length = 2
Convolution 1D	Filter = 128, Filter length = 11, Padding = Same, Activation = ReLU
Batch Normalization	-
Average Pooling	Pool length = 2
Convolution 1D	Filter = 256, Filter length = 11, Padding = Same, Activation = ReLU
Batch Normalization	-
Average Pooling	Pool length = 2
Convolution 1D	Filter = 512, Filter length = 11, Padding = Same, Activation = ReLU
Batch Normalization	-
Average Pooling	Pool length = 2
Convolution 1D	Filter = 512, Filter length = 11, Padding = Same, Activation = ReLU
Batch Normalization	-
Average Pooling	Pool length = 2
Flatten	-
Fully-connected	Neurons = 4096
Fully-connected	Neurons = 4096
Output	Softmax: 256 classes

Table 6: Network hyperparameters for $\text{CNN}_{\text{bnreg}}$.

Layer type	Hyperparameters
Trace input	1400
Convolution 1D	Filter = 64, Filter length = 11, Padding = Same, Activation = ReLU
Batch Normalization	-
Average Pooling	Pool length = 2
Convolution 1D	Filter = 128, Filter length = 11, Padding = Same, Activation = ReLU
Batch Normalization	-
Average Pooling	Pool length = 2
Convolution 1D	Filter = 256, Filter length = 11, Padding = Same, Activation = ReLU, $L_2 = 0.2$, Dropout = 0.5
Batch Normalization	-
Average Pooling	Pool length = 2
Convolution 1D	Filter = 512, Filter length = 11, Padding = Same, Activation = ReLU, $L_2 = 0.3$, Dropout = 0.6
Batch Normalization	-
Average Pooling	Pool length = 2
Convolution 1D	Filter = 512, Filter length = 11, Padding = Same, Activation = ReLU, $L_2 = 0.3$, Dropout = 0.7
Batch Normalization	-
Average Pooling	Pool length = 2
Flatten	-
Fully-connected	Neurons = 4096
Fully-connected	Neurons = 4096
Output	Softmax: 256 classes