Jump, It Is Easy: JumpReLU Activation Function in Deep Learning-based Side-channel Analysis

Abraham Basurto-Becerra[1], Azade Rezaeezade[2], and Stjepan Picek[1]

[1] Radboud University, The Netherlands
[2] Delft University of Technology, The Netherlands

Abstract. Deep learning-based side-channel analysis has become a popular and powerful option for side-channel attacks in recent years. One of the main directions that the side-channel community explores is how to design efficient architectures that can break the targets with as little as possible attack traces, but also how to consistently build such architectures. In this work, we explore the usage of the JumpReLU activation function, which was designed to improve the robustness of neural networks. Intuitively speaking, improving the robustness seems a natural requirement for side-channel analysis, as hiding countermeasures could be considered adversarial attacks.
In our experiments, we explore three strategies: 1) exchanging the activation functions with JumpReLU at the inference phase, training common side-channel architectures with JumpReLU, and 3) conducting hyperparameter search with JumpReLU as the activation function. While the first two options do not yield improvements in results (but also do not show worse performance), the third option brings advantages, especially considering the number of neural networks that break the target. As such, we conclude that using JumpReLU is a good option to improve the stability of attack results.

Keywords: Side-Channel Analysis · Deep Learning · JumpReLU.

1 Introduction

Side-channel analysis (SCA) is a class of cryptanalytic attack techniques that exploit the physical implementation of cryptographic algorithms to extract sensitive information, such as cryptographic keys. Differing from traditional cryptanalysis that focuses on the mathematical properties of algorithms, side-channel attacks leverage unintentional information leakage from a device's physical characteristics during operation. These attacks can exploit different sources of leakage, including power consumption [18], electromagnetic emissions [11], timing [19], or even sound [12]. If a device is not sufficiently resistant to SCA, an adversary can measure its leakage and use statistical analysis methods to recover secrets.

SCA attacks can be divided into two categories [26]:

1. Non-profiling: also called direct attacks, where the adversary collects measurements from the device under attack and uses statistical methods to infer

the secret information. Examples of this attack category include simple analysis, differential analysis [18], and mutual information analysis [13].
2. Profiling: also called two-stage attacks, the adversary has a clone (or very similar) device to the device to be attacked. Then, the attacker uses this clone device to perform multiple cryptographic executions with different keys and inputs to create a training set and learn a statistical model from side-channel leakages. Next, in the attack phase, this model is used against the target device. If the model provides satisfactory generalization, the attacker can recover secrets from the target device. The most well-known and powerful profiling method, from an information-theoretic perspective, is the template attack [4]. It is based on the Bayesian rule and the assumption that the measurements are mutually independent among the features given the target class [1, 4].

In recent years, deep learning-based approaches have emerged as a powerful alternative that can often surpass template attacks in performance [25]. In particular, multilayer perceptron (MLP) and convolutional neural networks (CNNs) have been widely explored and shown to be effective in recovering secrets from implementations protected with countermeasures [2,34]. Finding high-performing neural network architectures is frequently challenging as it involves selecting from a long list of hyperparameters. From these hyperparameters, in the side-channel analysis domain, the activation function is normally selected to be either ReLU or SeLU. ReLU is the most commonly used activation function, mainly due to its simplicity and effectiveness. Despite research on novel activation functions specifically designed for side-channel analysis, such as the proposal by Knežević et al. [17], ReLU remains a standard choice. However, this does not mean that more suitable activation functions cannot be designed.

In the field of adversarial learning, Erichson et al. [8] presented the JumpReLU activation function. The authors describe it as a very simple and inexpensive strategy that can be used to "retrofit" a previously trained network to improve its resilience to adversarial attacks (i.e., attacks on machine learning that aim to cause misclassification of the machine learning algorithm). In deep learning side-channel analysis, traces capture noise, and implementation countermeasures can be seen as a manipulation to deceive the model into making incorrect predictions. Thus, the natural questions that arise are: Can JumpReLU be used to improve model accuracy in the SCA domain? Can JumpReLU be used as in [8] on already-trained models to enhance their accuracy? Which threshold value should be used? Can we expect randomly generated models to perform better using JumpReLU instead of ReLU and SeLU?

In this paper, we provide answers to these questions by performing a comprehensive list of experiments with the well-known SCA datasets.
In summary, our main contributions are:

1. We verify whether JumpReLU can be used in already-trained SCA models to increase their performance.
2. We show how performance is affected when an existing architecture is taken and its activation function is replaced with JumpReLU.

3. We provide experimental results of which threshold range provides the best performance.
4. We provide a benchmark between random models using ReLU, SeLU, and JumpReLU.

2 Background

2.1 Deep Learning SCA

Deep learning techniques, particularly neural networks, enhance SCA by eliminating the need for extensive pre-processing or manual feature engineering [3,15]. By training models on large datasets of captured leakages, attackers can improve their ability to recover secret keys even from countermeasure-protected (masking [20] and hiding [21]) devices, making this approach a growing concern [10].

Deep learning-based SCA (DLSCA) is a profiling attack defined as a classification problem. The output classes are specified using the target intermediate variable and leakage model. In the DLSCA profiling step, a set $\mathcal{X}$ contains N_p measurements along with their counterparts plain/ciphertexts and keys collected from the clone device and is used to train the deep neural network. In the attack phase, the trained model is used to classify N_a measurements from the target device. The output probabilities of the neural network for each class are used to rank the most probable key. For each key candidate, a score S_k is calculated using the formula:

$$S_k = \sum_{i=1}^{N_a} \log p(x_i, c_j). \tag{1}$$

Here, N_a is the number of measurements in the attack set, and $p(x_i, c_j)$ represents the probability that a measurement x_i belongs to class c_j. The keys are then sorted according to their scores. The key with the highest score is considered the most likely key used for the cryptographic operation on the target device. The guessing entropy (GE) and the required number of attack traces (NT) can be defined using the score vector in Eq. (1). Suppose the correct key ($k^\star$) used in the cryptographic operation is ranked at the position r^{th} among all possible keys. This position is called the rank of ($k^\star$). The guessing entropy is the average rank of $k^\star$ in multiple experiments. The required number of attack traces (NT) is the average minimum number of measurements needed for the model to place $k^\star$ in the first position (where $GE = 0$) [30].

2.2 Datasets

For this work, two well-known publicly available datasets were used, namely the ASCAD dataset [2] in its fixed-key and variable-key variants. These datasets were captured from EM measurements while executing a software-protected AES implementation running on an 8-bit AVR architecture microcontroller ATMega8415. The implementation is protected with the first-order masking, where

the masks for the first two bytes in the AES state during the first round are fixed to 0, i.e., are not protected. As such, the target byte for this work is always the third byte. For the fixed-key version, the 700 points of interest preselected by the dataset authors are used. The dataset consists of a total of 60000 traces: 50000 training traces and 10000 test traces. For the variable-key version, the 1400 points of interest preselected by the dataset authors are used. The dataset consists of a total of 300000 traces: 200000 training traces and 100000 test traces.

2.3 Activation Functions

Activation functions are an essential component of neural networks, introducing non-linearity. They enable the learning of complex patterns and relationships in data. Without activation functions, a neural network would be equivalent to a single-layer model, regardless of the number of layers, limiting its ability to capture complicated dependencies. Activation functions help determine how neurons respond to inputs and control the flow of gradients during backpropagation, affecting convergence speed and overall training stability [6].

ReLU The Rectified Linear Unit (ReLU) activation function is a widely used non-linear function in artificial neural networks [22, 29], particularly in deep learning models. It is defined by:

$$g(z) = max\{0, z\}. \tag{2}$$

ReLU is the default recommendation in modern neural networks [14].

SeLU The Scaled Exponential Linear Units (SeLU) [16] activation function is defined by:

$$f(x) = \lambda \begin{cases} x, & \text{if } x > 0. \\ \alpha(e^x - 1), & \text{if } x \leq 0, \end{cases} \tag{3}$$

with $\alpha \approx 1.6733$ and $\lambda \approx 1.0507$.

JumpReLU The JumpReLU [9] (Jump Rectified Linear Unit) activation function is a variant of the standard ReLU function with a jump discontinuity, yielding piece-wise continuous functions. It introduces a positive threshold parameter κ such that the neuron remains inactive until its input exceeds κ. The magnitude of the jump κ is a parameter the user must define.

$$J(z) = zH(z - \kappa) = \begin{cases} 0, & \text{if } z \leq \kappa. \\ z, & \text{if } z > \kappa. \end{cases} \tag{4}$$

Here, H denotes the discrete Heaviside unit step function. The JumpReLU activation function introduces robustness and an additional amount of sparsity,

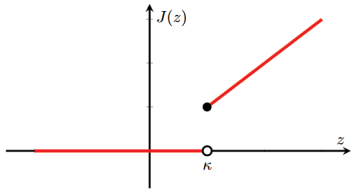


Fig. 1: JumpReLU activation function.

controlled via the jump value κ. Thus, JumpReLU suppresses small positive signals. We depict the operation of the JumpReLU activation function in Figure 1.

When presented, JumpReLU was proposed as a strategy to improve resilience to adversarial attacks. Moreover, it was concluded that it can be used in models already trained using ReLU. In this scenario, it provides a trade-off between robustness and classification accuracy, which the user can control in a post-training stage by tuning the threshold value κ.

Another application where JumpReLU has shown improved results versus other activation functions is in sparse autoencoders for language models. Here, JumpReLU enables more faithful reconstructions than competing methods, such as Gated or TopK Sparse AutoEncoders [27].

3 Related Work

Since the work of Maghrebi et al. [20], where the first published results of deep learning techniques applied to the domain of side-channel analysis are presented, multiple research works have been published. Those works showcase the importance of hyperparameter tuning and how identifying high-performing models can be challenging due to the large number of hyperparameters that must be considered. As such, multiple works considered how to find high-performing neural network architectures efficiently, see, e.g., [24, 30, 32–34].

Regarding the impact of activation functions, Benadjila et al. [2] have studied the effect of the activation function on the performance of the neural network, and they reported that their best results were obtained with ReLU, tanh, and softsign (a variation of tanh), but they chose ReLU due to its state-of-the-art results and because its computation time is lower than the other two functions. Knežević et al. [17] used evolutionary algorithms to evolve new activation functions for side-channel analysis. Their experiments with the ASCAD database showed that this approach is highly effective compared to results obtained with standard activation functions and that it can match the state-of-the-art results from the literature. The authors evaluated two leakage models (the Hamming Weight (HW) and Identity (ID) models) and MLP and CNN architectures. Moreover, Do et al. [5] analyzed the effect of the activation function in MLP- and CNN-based deep learning models for non-profiled side-channel analysis. For their MLP-based models, using the ELU activation function provided

better performance than ReLU in fighting against noise generation-based hiding countermeasures.

4 Experimental Setup

4.1 Neural Network Topologies

We use two well-known neural network topologies that are common choices in DLSCA: **Multilayer Perceptron (MLP)** is a basic type of neural network with an input layer, one or more hidden layers, and an output layer. The input layer receives training data that passes through fully connected hidden layers before reaching the output. In classification tasks, the output layer represents different classes. MLP learns patterns in the data by adjusting its weights using gradient descent and backpropagation.

Convolutional Neural Network (CNN) is another type of neural network designed to process structured data. It includes convolutional layers that detect important features using small filters (kernels). These layers are followed by activation functions and pooling layers, which help reduce complexity while keeping essential information. Finally, one or more fully connected layers process the extracted features for the final output.

4.2 Datasets and Leakage Model

In our experiments, we consider two publicly available datasets: **ASCAD-Fixed** and **ASCAD-Variable** and their **desynchronized 50 and 100** versions.[3] Both datasets are provided using measurements from a software implementation of AES-128. The implementation is protected with Boolean masking. Since the first and second bytes are masked with zero, and the sensitive variable leaks in the first order, we target the third byte. The target platform is an 8-bit AVR microcontroller (ATmega8515), and the measurements are the electromagnetic emanations (EM) from the target [2]. In ASCAD-Fixed, there are 50000 traces for training, all with the same key, and 10000 traces for attack. The traces have 700 features, an interval that includes most leaky time samples considering the Sbox output as a sensitive variable.
In ASCAD-Variable, there are 200000 traces for training with the key changing randomly for every trace and 100000 traces for attack (we use 100000 traces for training and 10000 traces for attack from this dataset). The traces have 1400 features. The key is fixed when measuring the attack traces for both datasets. In the desynchronized versions of both datasets, each trace is shifted using a random variable of 0 to $N^{[0]}$, where $N^{[0]} = 50$ for ASCAD-Fixed and ASCAD-Variable denotes desynchronization of 50, and $N^{[0]} = 100$ for ASCAD-Fixed and ASCAD-Variable denotes desynchronization of 100.

[3] The datasets can be found here.

We use the Identity (ID) leakage model, where we assume that the exact value of the sensitive variable is leaking. Using divide-and-conquer[4] strategy and targeting the Sbox output (8 bits) of the first round as the sensitive variable, there are 256 possible classes with the ID leakage model.

4.3 Analysis Methodology

This study examines how the use of JumpReLU affects the performance of DLSCA. To clarify this, we investigate two different scenarios.

1. We explore whether substituting the activation function in well-known architectures with JumpReLU can potentially enhance their performance. This is explored in two settings:
   - Substitution of the activation function on an already trained model, i.e., JumpReLU is used only at the inference phase.
   - Substitution of the activation function and training of the model, i.e., JumpReLU is employed during both the training and inference phases.
2. We investigate whether the average performance of random CNN and MLP models for DLSCA is better using JumpReLU compared to the other activation functions.

In the following, we introduce the steps required for each scenario.
In the **first scenario**, we would like to see if replacing the activation function of well-known architectures with JumpReLU results in improving the performance of those models. To verify this, our methodology is straightforward. We take the following steps:

- **Acquiring baseline models:** We start by training ten models using the architectures reported in previous works. Training is done using identical hyperparameters and datasets to ensure comparable results. The models we used are listed in Table 1. We call them "baseline models". The activation functions in these architectures are either ReLU or SeLU, but none use JumpReLU. The works listed in Table 1 reported one or more CNN and/or MLP neural networks for at least one of the datasets in Section 4.2. Table 1 shows what kind of model has been reported for which dataset in each work.
- **Record the baseline model's performance:** After training each model and using that model to attack, we use guessing entropy and the required number of traces (NT) to report that model's performance for the target dataset the model was developed for.
- **Re-training models with JumpReLU:** Once more, we need to train the baseline models. This time, we keep all the hyperparameters as before and only replace the models' activation function with JumpReLU. As mentioned in Section 2.3, JumpReLU has its own hyperparameter, κ, which should be tuned for each neural network. To do this tuning, we examine five thresholds

[4] Divide-and-conquer strategy is a strategy to recover a long key by retrieving its smaller parts separately

Table 1: The list of works/architectures we consider in our experiments. We depict $*$ if the work reported a well-performing CNN and $\bullet$ if the work reported a well-performing MLP.

Covered datasets	ASCAD-Fixed			ASCAD-Variable		
	$N^{[0]} = 0$	$N^{[0]} = 50$	$N^{[0]} = 100$	$N^{[0]} = 0$	$N^{[0]} = 50$	$N^{[0]} = 100$
[2]	$*, \bullet$	$*, \bullet$	$*, \bullet$	$*, \bullet$	$*, \bullet$	$*, \bullet$
[34]	$*$	$*$	$*$			
[32]	$*$	$*$	$*$			

for each baseline model. We train each model with five different values for κ, $\kappa \in \{0.001, 0.002, 0.003, 0.004, 0.005\}$. The tuned model with the best performance is denoted the "JumpReLU-equivalent model".
- **Report the JumpReLU-equivalent models performance:** The last step is to report the performance of the models with the JumpReLU activation function and compare it with the performance of the original ones. The hypothesis is to verify whether replacing previously found good models' activation functions with JumpReLU improves their performance.

In the **second scenario**, we would like to see if using JumpReLU can improve the performance of the models on average. To evaluate this, we compare the average performance of a number of neural networks using two common activation functions (ReLU and SeLU being randomly selected for each neural network combination of hyperparameters) against their performance when their activation function is fixed to JumpReLU. The methodology for comparing performance is described in the following steps.

- **Acquiring baseline models:** We generate 500 neural networks of the specific topology (MLP or CNN) using a random search. The searching ranges for hyperparameters of MLP and CNN are listed in Table 2. These ranges are chosen based on those reported in the previous works [7, 23, 34]. Since those 500 neural networks are generated randomly, many of them cannot decrease GE. Then, we only select the models that reached $GE = 0$ within 4000 attack traces as the "baseline models".
- **JumpReLU equivalent models:** With the same topologies generated for the baseline models we verify if by using JumpReLU would have provided better results for this random model, and validate against five different values for κ, $\kappa \in \{0.001, 0.002, 0.003, 0.004, 0.005\}$.
- **The average performance of baseline models:** We use "AVERAGE_GE" and "AVERAGE_NT" to represent the average performance of baseline models. The "AVERAGE_GE" is the average over GE that MLP or CNN baseline models can reach in an attack set with 4000 attack traces. The "AVERAGE_NT" is the average over the required number of attack traces that MLP or CNN baseline models need to reach $GE = 0$.

Table 2: Searched range of MLP and CNN hyperparameters. For both MLP and CNN dense layers, we used the ranges shown in *Dense layers* part of Table 2.

Hyperparameters	**Range**
MLP dense layers	
Number of neurons	[10, 30, 50, 70, 90, 120, 150, 200, 250, 300, 400, 500]
Number of layers	[2, 8], step = 1
CNN convolution and dense layers	
Number of neurons in dense layer	[50, 100, 150, 200, 300, 400, 500]
Number of dense layers	[2, 4], step = 1
Number of convolution layers	[2, 4], step = 1
Kernel size	[4, 20], step = 2
Number of filters	[4, 24], step = 4
i^{th} layer filter size	$((i-1)^{th} filter\_size)^2$
Pooling	"Average", "Max"
Pooling size	[2, 10], step = 2
Pooling stride	[2, 10], step = 2
Learning hyperparameters	
Optimizer	"Adam"
Weight initialization	"random_uniform", "he_uniform", "glorot_uniform",
Activation function	"ReLU", "SeLU"
Batch size	$[128, 256, 512]$
Learning rate	$[1e-3, 5e-4, 1e-4, 5e-5, 1e-5]$
Epochs	100

– **Re-training models with JumpReLU:** In step one, we generated two pools of MLP and CNN neural networks, each pool with 500 different models. Now, we replace the activation function of those models with JumpReLU and re-train them for $\kappa \in \{0.001, 0.002, 0.003, 0.004, 0.005\}$ jumping thresholds. Then, we consider the best-acquired performance as the performance of that model when using JumpReLU. Again, the considered metrics are GE and NT. The tuned baseline model with the best performance is called the "JumpReLU-equivalent model."
– **The average performance of JumpReLU-equivalent models:** The AVERAGE_GE and the AVERAGE_NT are calculated as performance metrics for the models that reached $GE = 0$ in each pool. We compare the AVERAGE_GE and AVERAGE_NT of the baseline and the JumpReLU-equivalent models. This way, the influence of using JumpReLU on DLSCA performance can be observed.

5 Experimental Results

5.1 Baseline Models

Retrofit To answer if the behavior observed by Erichson et al., where JumpReLU can be used to "retrofit" already trained models, is also true for SCA, we used two architectures described in [2]: MLP_{best} and CNN_{best}. For both architectures, ten

models were trained for each dataset with each $N^{[0]}$ using its original activation function (ReLU), and at the inference moment, ReLU and JumpReLU were used. As there is no previous data on which threshold range κ would provide the best performance, three different sets of values were tested $S_a = \{0.001k \mid k \in [1..9]\}$, $S_b = \{0.01k \mid k \in [1..9]\}$ and $S_c = \{0.1k \mid k \in [1..9]\}$.

MLP_{best}-based models only achieved generalization on the ASCAD-Fixed dataset with $N^{[0]} = 0$. All CNN_{best}-based models achieved generalization on both datasets and with the three evaluated $N^{[0]}$ values.

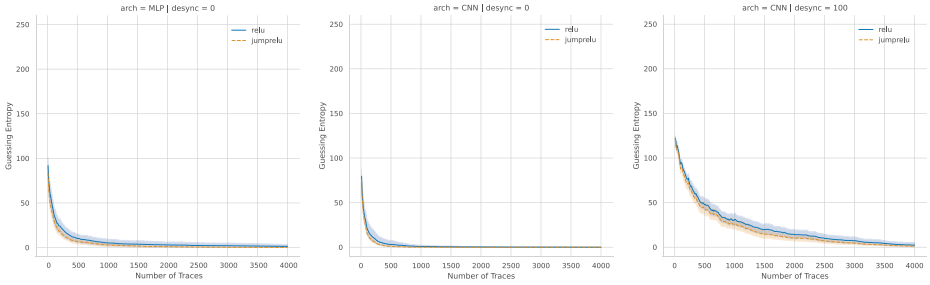
The experimental results show that using JumpReLU at inference time with the already-trained models did not provide any relevant performance variation. When the value of κ was from S_a or S_b, the performance of the models was almost identical to that of ReLU; for κ values in S_c equal to or greater than 0.5, the performance was even slightly worse than with ReLU. These results are consistent across both datasets and the three $N^{[0]}$ corresponding values.

Substitution Having seen that simply using JumReLU at inference time does not provide the desired performance gain, we investigate the impact of substituting the original activation function with JumpReLU. Again, we used MLP_{best} and CNN_{best} based models, but this time, these are trained using JumpReLU. Concerning model generalization, the results are the same as described in the previous experiments. However, in this scenario, the value of κ has a more pronounced impact on the performance of the models. Here, κ values from S_a provide a slight improvement for some models (see Figures 2 and 3), while values from S_b generally provide worse performance, and finally, values from S_c mostly lead to models not generalizing. Note that the figures depict the average behavior in darker lines and the standard deviation by shaded area in the same color.

With the information that JumpReLu can provide a small performance gain on architectures that already have shown good performance, we proceed to evaluate the performance of other architectures that do not use the ReLU function but SeLU. The tested architectures are from [34] and [32]. Both works used a technique called One Cycle Policy [31] to choose the learning rate hyperparameter. However, as we want to isolate the effect of substituting the activation function to JumpReLU, this technique is not used in our experiments, and only three different learning rate values were used: $\{0.0005, 0.00025, 0.0001\}$. The experimental results show that for these architectures, SeLU presented the best results, and JumpReLU did not provide any improvement.

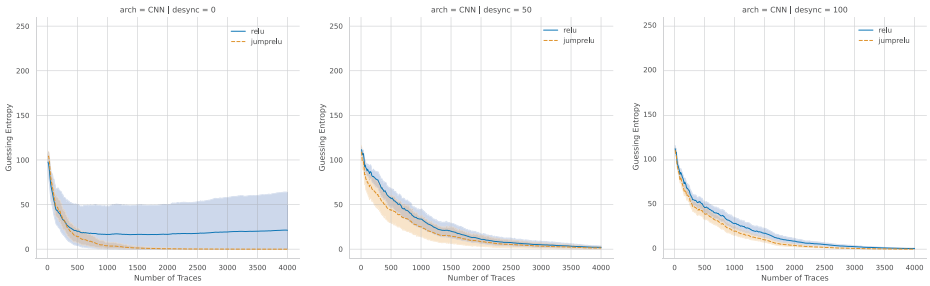
5.2 Random Models

Our observations show that substituting the activation function of existing well-performing architectures with JumpReLU cannot provide a significant advantage. Thus, the remaining scenario where JumpReLU is considered is to find new well-performing architectures. We explore this scenario by performing a random model search.



(a) MLP; $N^{[0]}$=0; κ=0.001 (b) CNN; $N^{[0]}$=0; κ=0.004 (c) CNN; $N^{[0]}$=100; κ=0.003

Fig. 2: ASCAD-Fixed observed performance improvement.



(a) CNN; $N^{[0]}$=0; κ=0.001 (b) CNN; $N^{[0]}$=50; κ=0.001 (c) CNN; $N^{[0]}$=100; κ=0.001

Fig. 3: ASCAD-Variable observed performance improvement.

ASCAD-Fixed The results for the ASCAD-Fixed dataset are given in Table 3. For MLP-based models, $GE = 0$ was only reached with $N^{[0]} = 0$, with 163 models using ReLU and 104 using SeLU. CNN-based models did reach $GE = 0$ for all $N^{[0]}$, with 68 and 52 models for ReLU and SeLU activation functions, respectively. Note that we do not show any results with MLP and desynchronization as we could not find any architecture breaking the target with the given number of attack traces.

For JumpReLU, we see that both MLP and CNN architectures reach good results. More precisely, for CNNs, the threshold variations do not cause many differences, and all settings are stable: a similar number of models breaking the target and a similar minimal number of traces to break the target. Moreover, we observe that breaking a synchronized target is relatively simple, while after added desynchronization, the task becomes significantly more difficult (with only a handful of architectures breaking the target). An additional observation is that CNN with SeLU activation function actually reaches a better best result (only 2690 attack traces needed vs. 3990 for the case with JumpReLU). Still, CNNs with SeLU and ReLU do not manage to break the unsynchronized version with the 100 desynchronization level at all, showcasing that JumpReLU provides more robustness. On the other hand, for MLP, we see that using JumpReLU al-

lows for the majority of the tested models to break the target, and the best results are comparable with the SeLU case (180 vs. 190 attack traces). To conclude, JumpReLU allows more architectures to break the target, with minimal influences from the selected threshold level, and allows comparable results on the minimal number of attack traces required to break the target.

Table 3: Experimental results for the ASCAD-Fixed dataset. The column Models denotes the number of models that reached GE equal to 0. $N^{[0]}$ denotes the desynchronization rate, Arch denotes the architecture type, and AF denotes the activation function used (in the last layer is always softmax).

$N^{[0]}$	**Arch**	**AF**	**Threshold**	**Models**	**Avg NT**	**Min NT**
0	cnn	SeLU	-	52	2202.7	460
0	cnn	ReLU	-	68	2682.8	570
0	cnn	JumpReLU	0.001	118	2289.2	440
0	cnn	JumpReLU	0.002	116	2430.6	390
0	cnn	JumpReLU	0.003	119	2457.7	430
0	cnn	JumpReLU	0.004	115	2437.2	420
0	cnn	JumpReLU	0.005	117	2494.5	490
50	cnn	SeLU	-	3	3123.3	2690
50	cnn	ReLU	-	1	4000.0	4000
50	cnn	JumpReLU	0.002	1	3990.0	3990
100	cnn	JumpReLU	0.001	2	3980.0	3970
100	cnn	JumpReLU	0.002	2	3860.0	3720
100	cnn	JumpReLU	0.005	2	3915.0	3830
0	mlp	ReLU	-	163	705.1	210
0	mlp	SeLU	-	104	902.0	180
0	mlp	jumpReLU	0.001	344	732.8	220
0	mlp	JumpReLU	0.002	344	754.6	210
0	mlp	JumpReLU	0.003	334	745.7	210
0	mlp	JumpReLU	0.004	337	729.6	230
0	mlp	JumpReLU	0.005	334	724.0	190

ASCAD-Variable The results for the ASCAD-Variable dataset are given in Table 4. With this dataset, for all $N^{[0]}$ values, at least one model reached $Ge = 0$. Considering JumpReLU, we can again observe that the threshold level does not play a significant role. Moreover, as before, JumpReLU allows more architectures to break the target than ReLU and SeLU. Still, with MLP-based models with desynchronization levels of 50 and 100, we break the target using SeLU, while we cannot do it with JumpReLU. However, since there is only one such architecture, it is difficult to assess the relevance of such a result. For CNNs without desynchronization, we also observe that JumpReLU reduces the minimal number of attack traces, giving additional advantage to the usage of JumpReLU.

Table 4: Experimental results for the ASCAD-Variable dataset. The column Models denotes the number of models that reached GE equal to 0. $N^{[0]}$ denotes the desynchronization rate, Arch denotes the architecture type, and AF denotes the activation function used (in the last layer is always softmax).

$N^{[0]}$	**Arch**	**AF**	**Threshold**	**Models**	**Avg**	**Min**
0	cnn	SeLU	-	17	2956.5	470
0	cnn	ReLU	-	29	2656.6	350
0	cnn	JumpReLU	0.001	71	2349.4	170
0	cnn	JumpReLU	0.002	70	2527.4	210
0	cnn	JumpReLU	0.003	69	2463.2	300
0	cnn	JumpReLU	0.004	69	2505.8	290
0	cnn	JumpReLU	0.005	66	2378.6	240
50	cnn	SeLU	-	1	3800.0	3800
50	cnn	JumpReLU	0.001	1	2930.0	2930
50	cnn	JumpReLU	0.003	1	3980.0	3980
100	cnn	SeLU	-	1	4000.0	4000
100	cnn	JumpReLU	0.003	1	4000.0	4000
0	mlp	ReLU	-	75	2333.2	710
0	mlp	SeLU	-	42	1670.7	340
0	mlp	JumpReLU	0.001	143	2153.7	420
0	mlp	JumpReLU	0.002	132	2249.5	480
0	mlp	JumpReLU	0.003	131	2346.5	600
0	mlp	JumpReLU	0.004	136	2515.8	560
0	mlp	JumpReLU	0.005	128	2333.0	700
50	mlp	SeLU	-	1	3990.0	3990
100	mlp	SeLU	-	1	4000.0	4000

6 Conclusions and Future Work

This paper investigates how the JumpReLU activation function can improve the performance of DLSCA models. Finding performant models remains a significant challenge for DLSCA, and the process is still largely reliant on the designer's expertise and the computing resources at their disposal, as these factors determine how much experimentation and fine-tuning can be conducted within a given timeframe. Finding one model that shows good performance with a given combination of hyperparameters does not imply that small changes to any specific hyperparameter will lead to a predictable improvement or degradation in performance. Hyperparameter tuning is often highly context-dependent, with interactions between parameters influencing the overall model behavior. With this in mind and based on our experimental results, we can conclude that JumpReLU can be considered a promising option when constructing new architectures for DLSCA. More precisely, we see especially encouraging results when the JumpReLU is given as one of the options during the hyperparameter tuning. The architectures with it seem to improve the performance from two aspects: 1) more architectures breaking the target and 2) fewer attack traces required to break the target for the most performant architectures.

In future work, we plan to explore whether JumpReLU can bring advantages against other hiding countermeasures like Gaussian noise or jitter. Moreover, JumpReLU showed very good performance when combined with sparse autoencoders [28], which could be another interesting research direction for DLSCA.

References

1. Batina, L., Djukanovic, M., Heuser, A., Picek, S.: It started with templates: the future of profiling in side-channel analysis. Springer (2021)
2. Benadjila, R., Prouff, E., Strullu, R., Cagli, E., Dumas, C.: Deep learning for side-channel analysis and introduction to ascad database. Journal of Cryptographic Engineering **10**(2), 163–188 (2020)
3. Cagli, E., Dumas, C., Prouff, E.: Convolutional neural networks with data augmentation against jitter-based countermeasures: Profiling attacks without preprocessing. In: Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. pp. 45–68. Springer (2017)
4. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: International workshop on cryptographic hardware and embedded systems. pp. 13–28. Springer (2002)
5. Do, N., Hoang, V., Doan, V.S., Pham, C.: On the performance of non-profiled side channel attacks based on deep learning techniques. IET Information Security **17**(3), 377–393 (Dec 2022). https://doi.org/10.1049/ise2.12102, http://dx.doi.org/10.1049/ise2.12102
6. Dubey, S.R., Singh, S.K., Chaudhuri, B.B.: Activation functions in deep learning: A comprehensive survey and benchmark. Neurocomputing **503**, 92–108 (2022)
7. Emmanuel, P., Remi, S., Ryad, B., Eleonora, C., Cecile, D.: Study of deep learning techniques for side-channel analysis and introduction to ascad database. CoRR **53**, 1–45 (2018)

8. Erichson, N.B., Yao, Z., Mahoney, M.W.: Jumprelu: A retrofit defense strategy for adversarial attacks. arXiv preprint arXiv:1904.03750 (2019)
9. Erichson, N.B., Yao, Z., Mahoney, M.W.: JumpReLU: A Retrofit Defense Strategy for Adversarial Attacks (Apr 2019). https://doi.org/10.48550/arXiv.1904.03750, http://arxiv.org/abs/1904.03750, arXiv:1904.03750 [cs]
10. Federal Office for Information Security (BSI): Guidelines for Evaluating Machine-Learning based Side-Channel Attack Resistance. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_46_AI_guide.pdf?__blob=publicationFile&v=6 (02 2024), technical Report AIS 46
11. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: Concrete results. In: Cryptographic Hardware and Embedded Systems—CHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings 3. pp. 251–261. Springer (2001)
12. Genkin, D., Shamir, A., Tromer, E.: Acoustic Cryptanalysis. Journal of Cryptology **30**(2), 392–443 (Apr 2017). https://doi.org/10.1007/s00145-015-9224-2, https://doi.org/10.1007/s00145-015-9224-2
13. Gierlichs, B., Batina, L., Tuyls, P., Preneel, B.: Mutual information analysis: A generic side-channel distinguisher. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 426–442. Springer (2008)
14. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org
15. Kim, J., Picek, S., Heuser, A., Bhasin, S., Hanjalic, A.: Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 148–179 (2019)
16. Klambauer, G., Unterthiner, T., Mayr, A., Hochreiter, S.: Self-normalizing neural networks. Advances in neural information processing systems **30** (2017)
17. Knežević, K., Fulir, J., Jakobović, D., Picek, S., Đurasević, M.: NeuroSCA: Evolving Activation Functions for Side-Channel Analysis. IEEE Access **11**, 284–299 (2023). https://doi.org/10.1109/ACCESS.2022.3232064, https://ieeexplore.ieee.org/document/9998512, conference Name: IEEE Access
18. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) Advances in Cryptology — CRYPTO' 99. pp. 388–397. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
19. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) Advances in Cryptology — CRYPTO '96. pp. 104–113. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
20. Maghrebi, H., Portigliatti, T., Prouff, E.: Breaking cryptographic implementations using deep learning techniques. In: Security, Privacy, and Applied Cryptography Engineering: 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings 6. pp. 3–26. Springer (2016)
21. Masure, L., Belleville, N., Cagli, E., Cornelie, M.A., Couroussé, D., Dumas, C., Maingault, L.: Deep learning side-channel analysis on large-scale traces: A case study on a polymorphic aes. In: European Symposium on Research in Computer Security. pp. 440–460. Springer (2020)
22. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th international conference on machine learning (ICML-10). pp. 807–814 (2010)
23. Perin, G., Picek, S.: On the influence of optimizers in deep learning-based side-channel analysis. In: Selected Areas in Cryptography: 27th International Confer-

ence, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers 27. pp. 615–636. Springer (2021)
24. Perin, G., Wu, L., Picek, S.: Exploring feature selection scenarios for deep learning-based side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 828–861 (2022)
25. Picek, S., Heuser, A., Perin, G., Guilley, S.: Profiled side-channel analysis in the efficient attacker framework. In: International Conference on Smart Card Research and Advanced Applications. pp. 44–63. Springer (2021)
26. Picek, S., Perin, G., Mariot, L., Wu, L., Batina, L.: Sok: Deep learning-based physical side-channel analysis. ACM Computing Surveys **55**(11), 1–35 (2023)
27. Rajamanoharan, S., Lieberum, T., Sonnerat, N., Conmy, A., Varma, V., Kramár, J., Nanda, N.: Jumping Ahead: Improving Reconstruction Fidelity with JumpReLU Sparse Autoencoders (Aug 2024). https://doi.org/10.48550/arXiv.2407.14435, http://arxiv.org/abs/2407.14435, arXiv:2407.14435 [cs]
28. Rajamanoharan, S., Lieberum, T., Sonnerat, N., Conmy, A., Varma, V., Kramár, J., Nanda, N.: Jumping ahead: Improving reconstruction fidelity with jumprelu sparse autoencoders (2024), https://arxiv.org/abs/2407.14435
29. Ramachandran, P., Zoph, B., Le, Q.V.: Searching for activation functions. arXiv preprint arXiv:1710.05941 (2017)
30. Rijsdijk, J., Wu, L., Perin, G., Picek, S.: Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 677–707 (2021)
31. Smith, L.N.: Cyclical learning rates for training neural networks. In: 2017 IEEE Winter Conference on Applications of Computer Vision (WACV). pp. 464–472 (2017). https://doi.org/10.1109/WACV.2017.58
32. Wouters, L., Arribas, V., Gierlichs, B., Preneel, B.: Revisiting a methodology for efficient cnn architectures in profiling attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 147–168 (2020)
33. Wu, L., Perin, G., Picek, S.: I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis. IEEE Trans. Emerg. Top. Comput. **12**(2), 546–557 (2024). https://doi.org/10.1109/TETC.2022.3218372, https://doi.org/10.1109/TETC.2022.3218372
34. Zaid, G., Bossuet, L., Habrard, A., Venelli, A.: Methodology for efficient cnn architectures in profiling attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 1–36 (2020)