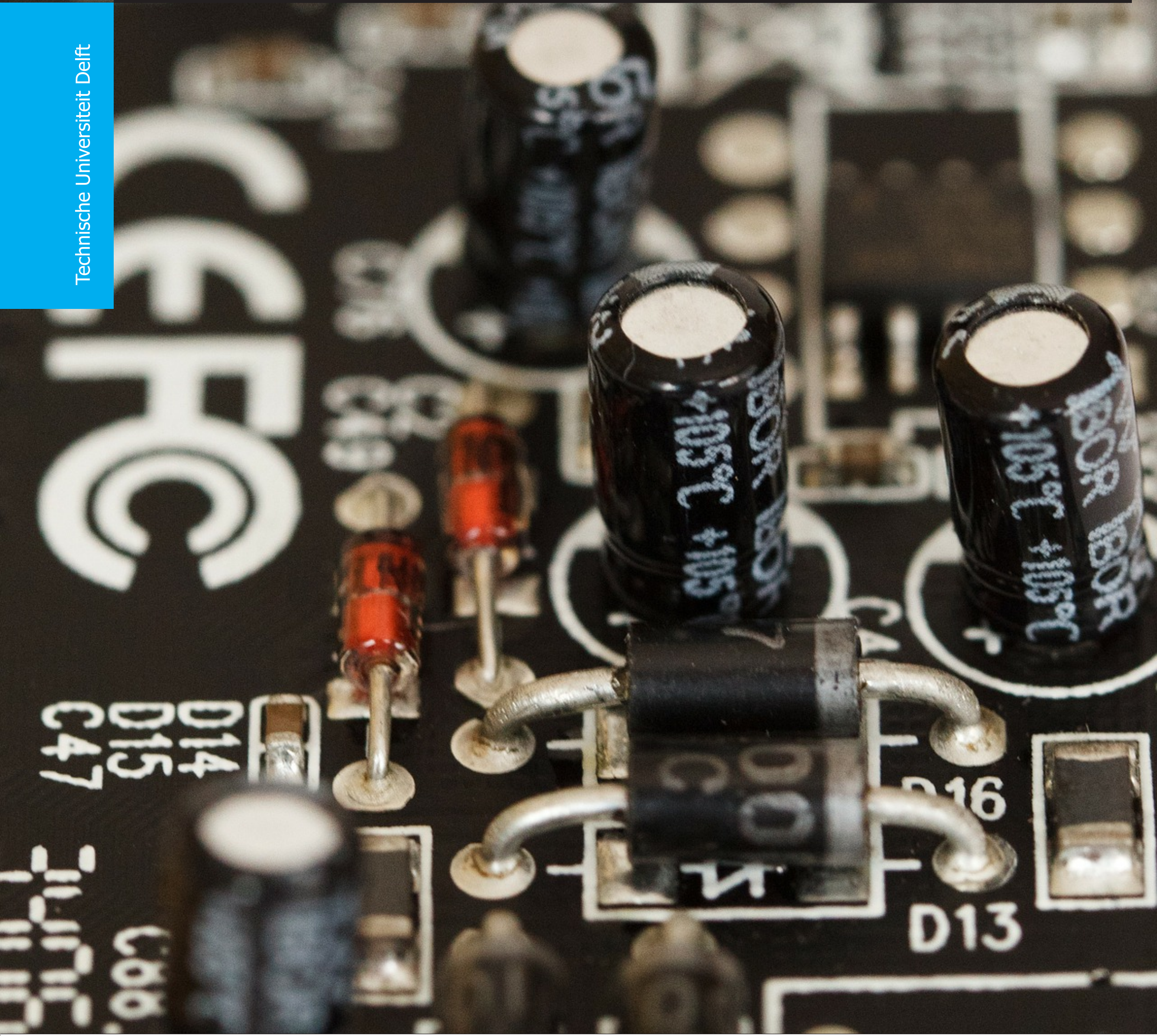
Towards Understanding of Deep Learning in Profiled Side-Channel Analysis

Similarity of predictors measured and explained

Daan van der Valk

Technische Universiteit Delft



TU Delft Delft University of Technology

Challenge the future

Towards Understanding of Deep Learning in Profiled Side-Channel Analysis

Similarity of predictors measured and explained

by

Daan van der Valk

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Thursday August 22nd, 2019 at 10:00 AM.

Supervisor:	Dr. S. Picek,	TU Delft
Thesis committee:	Prof. dr. P. H. Hartel,	TU Delft
	Dr. J. Urbano,	TU Delft
	Prof. dr. L. Batina,	Radboud University

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Side-channel attacks (SCA) aim to extract a secret cryptographic key from a device, based on unintended leakage. Profiled attacks are the most powerful SCAs, as they assume the attacker has a perfect copy of the target device under his control. In recent years, machine learning (ML) and deep learning (DL) techniques have become popular as profiling tools in SCA. Still, there are many settings for which their performance is far from expected. In such occasions, it is very important to understand the difficulty of the problem and the behavior of the learning algorithm. To that end, one needs to investigate not only the performance of machine learning but also to provide insights into its explainability.

In this work, we look at various ways to explain the behavior of ML and DL techniques. We study the bias–variance decomposition, where the predictive error in various scenarios is split in bias, variance and noise. While the results shed some light on the underlying difficulty of the problem, existing decompositions are not tuned for SCA. We propose the Guessing Entropy (GE) bias–variance decomposition, incorporating the domain-specific GE metric in a tool to analyze attack characteristics. Additionally, we show the relation between the mean squared error and guessing entropy. Our experiments show this decomposition is a useful tool in trade-offs such as model complexity.

To dive deeper into the inner representations of neural networks (NNs), we use Singular Vector Canonical Correlation Analysis (SVCCA) to compare models used in SCA. We find that different datasets, or even leakage models, are represented very differently by neural networks. We apply SVCCA to a recent portability study, which shows one should be careful to overtrain their networks with too much data.

Finally, do we even need complicated neural networks to conduct an efficient attack? We demonstrate that a small network can perform much better by mimicking the outputs of a large network, compared to learning from the original dataset.

Preface

Writing this thesis report has been both a pleasure and a nightmare. In the past year, I have enjoyed the supported of many fellow students, friends and family members. I would like to mention a couple of them.

I am grateful for Jay Kim's help with the TU Delft cluster, so its GPUs could do the heavy lifting required in this kind of research. Thanks to the quiet, near-anonymous, Chinese girl who has (literally) very patiently opened the door for me in my first weeks in NTU's office.

This project would have been impossible without the support of the Justus and Louise van Effen Foundation, whose Research Grant allowed my exchange to NTU. I hope they use the pictures I sent them wisely.

Thanks to all nice colleagues at PACE lab at the NTU, for making the exchange both productive and enjoyable. First and foremost, I would like to Shivam Bhasim for both his enlightening supervision, and the questionable but hilarious tours in Singapore. Johannes, Romain, Alexandre and of course Rico have made it a fantastic three months overseas.

I will forever be grateful to my parents for supporting me all the way through my studies. Also, the emotional and practical support from Lotte during my master was amazing. Thanks to Fons, Floor, Marius, Rico, Claudio, Julia and Lotte for proofreading this thesis report and putting up with the bad puns during the tryouts for my defense.

Last but not least, a big thank you to Stjepan. In all the meetings in person and on Skype, some at ridiculous times, he was somehow always full of fresh ideas and sharp comments; I couldn't wish for a better supervisor.

Daan van der Valk
Delft, August 2019

Contents

1 Introduction

In recent decades, information and communication technology (ICT) has interconnected people and organizations all over the world. For critical infrastructure, such as online banking and digital communication, our society heavily relies on cryptography. The *Internet of Things* (IoT) is a contemporary paradigm concerning the inter-networking of all kind of small electronic equipment, such as sensors, smart cards, microcontrollers, and mobile devices. Typically, they have limited storage and computational capacities. We depend on such devices for making payments, travel, identification, access control, monitoring, etc., and predictions indicate that the IoT will continue to grow in the coming years. Therefore, the cryptographic mechanisms on these constrained devices must be both secure and efficiently implemented [1].

The Advanced Encryption Standard (AES) was announced in 2001 as the new federal standard as a block cipher by the U.S. National Institute of Standards and Technology [2]. At the time of writing, AES is widely adopted and no efficient key-recovery attack on full AES is known. However, side-channel analysis (SCA) focuses not only on the observed communication (e.g., plaintext and ciphertext messages), but also use unintended leakage over one or multiple channels. For example, power consumption or electromagnetic emission of the chip may disclose some information about the computed encryption or decryption. There have been many successful side-channel attacks on different AES implementations [3].

A categorization of side-channel attacks on AES is shown in Figure 1.1. Simple and differential power analysis (SPA and DPA) were proposed in 1999 [4]. These rely on visual inspection and statistical analysis, respectively, of power consumption measurements from an implementation of DES, the predecessor of AES.

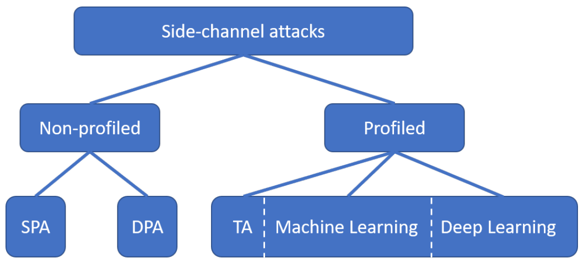


Figure 1.1: Hierarchy of side-channel attacks

With the growth of IoT and increasing security awareness, cryptographic research has led to countermeasures to harden devices against side-channel attacks. These can be divided into two categories: masking and hiding. Masking randomizes the relation between sensitive data and leakage, usually by introducing other variables. Hiding pursues feature-wise decorrelation: per feature, there should be a random or constant measurement. This is commonly implemented by a change in the encryption algorithm, by introducing random delays, or by changing the order of operations.

The side-channel community moved forward and found *profiled* attacks can be more powerful than SPA and DPA. Such attacks are split up in a profiling phase, where the patterns of traces from a clone device are captured in a model, and an attack phase in which the target device is attacked. A prevalent category of such profiled attacks are the so-called Template Attacks (TA), which are the most powerful attacks from an information theoretic perspective [5]. It is based on the Bayes theorem and considers first-order dependence among the different points of interest in the trace. Typically, noise is assumed to follow a multivariate normal distribution. So, the profiling phase in TA consists of estimating the mean vector and a covariance matrix to describe the leakage per class. These leakage classes are typically devised to profile for one byte of the key at the time, resulting in 256 classes for all possible bytes between `0x00` and `0xFF`.

The profiled setting can be translated to a typical machine learning (ML) classification problem. An algorithm, referred to as a *classification technique*, captures the patterns of some *training dataset* and saves them in a model. The classifier tries to separate the classes based on the data, and is typically evaluated on other data in the *test set*. In profiled side-channel attacks, the profiling traces are used as the training set and the traces from the target device are used as the test set. Numerous machine learning techniques have been used to attack AES implementations, such as the Naive Bayes' and Random Forest classifiers [6].

Deep learning, a bundle of techniques that employ artificial neural networks (NNs), has grown in popularity in the last decade. In recent years, deep learning has been shown very effective in SCA, even against protected implementations. In particular convolutional neural networks outperform other profiling techniques when they are provided with large enough datasets. However, neural networks are seen as black-box algorithms: it is unclear *why* and *how* they are so good in the classification problems they deal with. This complicates both the work of attackers and defenders in a practical setting. For attackers, there is no clear-cut methodology to find the optimal hyperparameters of a neural network: the model architecture (e.g., the numbers of layers, neurons per layer, kernel size, number of kernels and activation functions), learning rate, loss function, etc. Also, defenders cannot test every possible configuration of NNs to evaluate if their countermeasures are effective against any (convolutional) neural network. On a more abstract level, we miss a good understanding of why these deep learning techniques are so successful.

In a realistic scenario, an attacker would profile a certain device, but attack a different device with a different key. In literature, this is often simplified to splitting a dataset from a single device (with a single key) set in two: training and testing data. Recent works have shown that this simplification paints a too optimistic view of the attack [7–9]. This *portability* issue draws the question on what effect a change of device and/or key has on the attack model.

Neural networks come in all kinds and shapes. Large models, with multiple layers and many parameters, seem to be better in learning complicated patterns. However, there is little theoretical foundation for this: a small network should be able to approximate the same function. *Mimicking* such a large network may be the key for a small network to conduct a successful attack.

We identify the following problems:

- How do neural networks compare to template attacks and machine learning-based attacks? Do they share the same characteristics and trade-offs?
- Why do some network architectures work better than others?
- How can we effectively defend against neural networks used for SCA?
- How can we compare different NNs and measure to what extend they still learn the same patterns or intermediate representations, even when looking at different datasets (e.g., with/without countermeasures)?
- As devices have slightly different physical characteristics [7–9], how can we generalize a classifier trained for one device to work for another (portability)?

These are challenging questions, given the fundamental block box working of deep learning. Answers to them would help both sides of the cat-and-mouse game of SCA. On the attack side, a better understanding of why and how attacks work helps in breaking implementations, On the other hand, chip manufacturers and developers can use this knowledge to improve their implementations. Also, our work on NN analysis and comparison may be further extended by the deep learning community. In Chapter 3, after the discussion of related work, these problems are made more concrete and the research questions are stated.

This remainder of this report is structured as follows: Chapter 2 provides a background on machine learning, deep learning, AES, and side-channel analysis. In Chapter 3, relating state-of-the-art research is discussed, and the research question is reformulated and split into sub-questions. In the following chapters, we present the research conducted in this project, starting with Chapter 4 on bias–variance analysis of different classifiers. Next, the further analysis and comparison of neural networks is presented in Chapter 5, where we also look into the portability issue. Chapter 6 presents a first demonstration of mimicking in SCA. Finally, conclusions and recommendations are presented in Chapter 7.

2 Background

This chapter highlights the background of this research project. It excludes state-of-the-art research, which is discussed in next Chapter 3. First, machine learning and deep learning principles are explained. Then, we make the switch to cryptography, in particular AES. Finally, these areas come together in profiled side-channel analysis.

2.1. Machine Learning

Machine learning (ML) algorithms build mathematical models based on sample data to perform specific tasks without using explicit instructions, instead relying on patterns in the data. They are used in many different settings, such as image recognition, clustering, and spam filtering [10]. Supervised learning is the subset of ML tasks where a model is built to map an input to an output, based on some *training set* with example input-output pairs. When there is a finite number of possible outputs, the problem is called a classification problem, and the predictive models are called *classifiers*. After a classifier's learning phase, ideally it would correctly determine the class labels for new, unseen instances. A good score for the test set requires a classifier to effectively generalize from the training data.

The input typically consists of multiple numbers, referred to as *features* or *points of interest* (POIs). One individual input-output mapping is called a *labeled object*. When the object only includes the measurements, without the label, it is called a *feature vector*. We denote a feature vector with n features as $\mathbf{x} = (x_1, \ldots, x_n)$. A single (labeled) object is referred to as a *sample*, or *trace* in the side-channel domain. A trace only includes the class label y in context of the training set. The number of samples in the training set are denoted as N, and the test test contains M samples.

The true relation between a feature vector $\mathbf{x}$ and its label y is the mapping $f(\mathbf{x}) = y$. A machine learning algorithm aims to capture this function as well as possible, resulting in a model that performs some function $\hat{f}(\mathbf{x}) = \hat{y}$. Some models estimate the class distributions by learning a (fixed) number of *parameters*, which are embedded in the model and learned during the training algorithm. Some classifiers can be configured before training. To avoid confusion, such settings are called *hyperparameters* (e.g., number of trees, see Section 2.1.1).

2.1.1. Classifiers

Numerous classification techniques have been invented and used for side-channel analysis. This section provides a theoretical background on those used in this work: decision trees, random forest, naive Bayes, and the linear and quadratic discriminant.

Decision Tree

A decision tree is a simple classifier consisting of choices based on observations about a sample's features (represented in the branches), to predict its class (represented in the leaves). Decision trees are generally seen as white-box classifiers [11]. Those of moderate depth can be visualized in a human-readable format to explain the decision making.

The tree is built in a greedy manner: based on some splitting criteria, the training set is divided in each branch. This process is repeated recursively, until a node either consists of samples which all have the same class, separation cannot be further improved, or the maximum depth of the tree (a hyperparameter) is reached. Decision trees are deterministic; for the same settings and dataset, they will result in the same predictive function.

The splitting condition determines how the splits are made. A popular metric is called the *Gini impurity*, which is minimized over all possible splits. A split will result in regions where samples fall into. Such a subspace in a node m is defined as region R_m, consisting of N_m samples. We define the proportion of observations of a class k in node m as follows:

$$p_{mk} = 1/N_m \sum_{x_i \in R_m} I(y_i = k) \tag{2.1}$$

Then, the Gini impurity represents the probability of misclassifying a random sample from the dataset, if it was randomly labeled according to the dataset's class distribution:

$$H(X_m) = \sum_k p_{mk}(1 - p_{mk}) \tag{2.2}$$

The Gini impurity is minimized by the learning algorithm; it is zero in the optimal case when every region contains only samples of a single class. Optimized versions of the decision tree learning algorithm have runtime $\mathcal{O}(n \cdot N \cdot \log N)$, with n the number of features and N the number of samples.

Random Forest

The random forest (RF) algorithm is an ensemble decision tree learner [12], consisting of k decision trees. Contrary to individual decision trees, random forests are non-deterministic: they introduce two sources of randomness in the training phase. For each tree, a sample of the training set is drawn with replacement – this is a bootstrap sample, see Section 2.1.3. Also, instead of splitting a node according to the best split among all features, it is selected as the best split among a random subset of the features. Random forests generally suffer less from overfitting and variance compared to a single decision tree [12].

In this work, we use the Scikit-learn [13] implementation of the random forest. Instead of class voting as proposed by Breiman [12], the trees' decisions are combined by averaging their probabilistic predictions, as this typically results in better performance[1]. The default configuration was used: the Gini criterion for splitting and $\sqrt{n}$ features are randomly selected for each branch. There is one hyperparameter that we varied: the number of trees; see chapter 4 for details. Training runtime has complexity $\mathcal{O}\left(k\left(n \cdot N \cdot \log N\right)\right)$ with k the number of trees.

Naive Bayes Classifier

Naive Bayes (NB) is based on Bayes' rule, which finds the conditional probability of class y when observing feature vector $\mathbf{x} = (x_1, \ldots, x_n)$ as follows:

$$p(y \mid \mathbf{x}) = \frac{p(y)\; p(\mathbf{x} \mid y)}{p(\mathbf{x})} \tag{2.3}$$

Typically, the probabilities $p(y)$, $p(\mathbf{x} \mid y)$, $p(\mathbf{x})$ are unknown, but are estimated based on the training data. Naive Bayes is not a single classifier, but rather a family of classifiers that make the (naive) assumption that features $x_1, \ldots, x_n$ are independent. Although this assumption rarely holds, it allows to separate the features' conditional probabilities per class $p(\mathbf{x} \mid y)$ in Eq. 2.3:

$$p(y \mid x_1, \ldots x_n) = \frac{p(y)\; \prod_{i=1}^{n} p(x_i \mid y)}{p(x_1, \ldots, x_n)} \tag{2.4}$$

[1]Scikit-learn developers (2019). Ensemble methods. *Scikit-learn documentation*. Retrieved May 22, 2019, from https://scikit-learn.org/stable/modules/ensemble.html#random-forests.

This assumption simplifies and accelerates training tremendously, as it scales linear with the number of features. The classifiers using this rule are different in how they estimate the distribution of $p(x_i \mid y)$. In this research, we follow literature and use the Gaussian Naive Bayes classifier [14], which assumes a Gaussian distribution for all features:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{2.5}$$

Training the Naive Bayes classifier is very efficient and has only $\mathrm{O}(N \cdot n)$ time complexity. For some datasets in SCA (e.g., masked implementations) the correlation between features reveal important patterns [15]. Consequently, highly-correlated features in the measurements will be ignored as such by Naive Bayes and can reduce the number of successful predictions. Another limitation is that Naive Bayes is better in label prediction than probability estimation [16]. In SCA, we typically use guessing entropy for attacking difficult data (see Section 2.4.1), which employs class probability estimates.

Linear and Quadratic Discriminant Analysis

Just like Gaussian Naive Bayes, linear discriminant analysis (LDA) and quadratic discriminant analysis (QDA) assume a class-wise Gaussian distribution for each feature and attempts to capture data characteristics using Bayes' rule (Equation 2.3). However, they are more flexible than Naive Bayes, as it does not assume features to be independent. Phrased differently, Gaussian Naive Bayes assumes the covariance matrix of each class to be diagonal, contrary to LDA and QDA.

In LDA and QDA, the probability $P(\mathbf{x} \mid y)$ is modeled as a multivariate Gaussian distribution with density:

$$P(\mathbf{x} \mid y = k) = \frac{1}{(2\pi)^{n/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu_k)^t \Sigma_k^{-1} (\mathbf{x} - \mu_k)\right) \tag{2.6}$$

where n is the number of features. This distribution is approximated by estimating a class means μ_k and covariance matrix Σ_k for each class k.

LDA assumes that all classes have an equal covariance:

$$\Sigma_k = \Sigma \quad \text{for every class } k \tag{2.7}$$

As a result, LDA classification decision boundaries are linear. In side-channel analysis, LDA is known as *pooled (covariance)-Template Attack*, as the covariance matrixes are pooled among classes.[2]

QDA is similar to LDA, but drops assumption 2.7. QDA is therefore more flexible than LDA, as it can better cope with differently shaped classes. However, as the covariance matrix of each class is estimated separately, it requires more training examples to make good predictions. In side-channel analysis, QDA is commonly referred to as Template Attack [18]. LDA and QDA both have time complexity $\mathrm{O}(N \cdot n^2)$. Figure 2.1 depicts the difference between Gaussian NB, LDA, and QDA for an artificial two-dimensional problem.

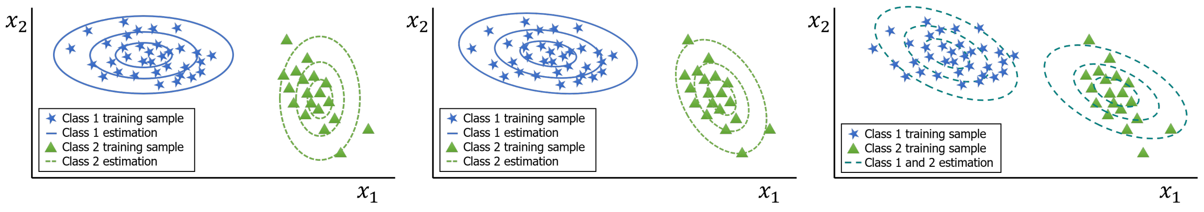
2.1.2. Data Normalization

Some classifiers are sensitive to scaled or shifted data [19]. This means that features with more extreme values will be more influential in the classifier's decision, even if they say little about the actual class. Also, for some techniques it takes longer to learn patterns if very high or low values are present [19, 20]. To address this issue, data is often pre-processed. This is commonly referred to as *data normalization* or *scaling*. In this work, features were normalized by subtracting the mean $\bar{x}_i$ and dividing by the standard deviation σ_i:

$$x_i' = \frac{x_i - \overline{x_i}}{\sigma_i} \quad \text{for every feature } i = 1, \ldots, n \tag{2.8}$$

After scaling, we have new mean $\overline{x_i'} = 0$ and standard deviation $\sigma_i' = 1$ for each feature i.

[2]Note that the SCA community has no uniform definition on how to implement a pooled TA attack, e.g. whether the covariance matrix should be scaled [17]. We use the sklearn's LDA classifier with an SVD-based solver, which scales the variance.



(a) Gaussian Naive Bayes (b) QDA (Template Attack) (c) LDA (pooled Template Attack)

Figure 2.1: Three classifiers that assume a class-wise multivariate Gaussian distribution, for a mock problem with two-dimensional data distributed over two classes. (a) shows Naive Bayes, where a different covariance matrix is estimated per class. As features are assumed to be independent, the covariance matrix is diagonal. In the graphical depiction, this means the class estimation will never be 'rotated', but rather stretched. (b) shows the classical template attack (TA), in machine learning known as quadratic discriminant analysis (QDA). Here, a different covariance matrix is estimated for each class, and assumptions are made about feature-wise dependence. From the theoretical point of view, with an unlimited number of training data, this is the most powerful attack for data that follows a multivariate normal distribution. Finally, (c) shows the pooled TA, also known as linear discriminant analysis (LDA), in which one covariance matrix is estimated for all classes. For a small sample size or high number of features, this can lead to better predictions than QDA.

2.1.3. Evaluation

The performance of a classifier is often evaluated on data not used for training, called the *test set*. A simple metric for a classifier's performance is *accuracy*, the percentage of test samples that are assigned the correct class:

Definition 2.1.1 *The accuracy of a model indicates which fraction of the data is labeled correctly, i.e.*

$$accuracy = \frac{\textit{number of correct predictions}}{\textit{number of predictions}}$$

However, just splitting the data in a training and test set is not enough for classifiers that can be configured with hyperparameters (e.g., the number of trees in a random forest). When multiple models are compared using their performance on the test set, selecting the model with the highest score may create a bias towards the test set. In that case, the estimated performance is too optimistic. Therefore, it is common to use another evaluation set: the *validation set*. The validation set is used to find the best configuration for the model. Then, the test set is only used at the very end to objectively estimate the performance for unseen data.

Some classification techniques directly attempt to maximize the accuracy. Others make use of a different function that expresses how good or bad the model fits the data. Such a function is generally an *error* or *loss function*. For each sample, the loss can be computed, and a learning technique attempts to minimize the average loss over the training set. A popular error function is the $0-1$ *loss*:

Definition 2.1.2 *The* $0-1$ *error of a predicted class* $\hat{y}$*, compared to the true class* y*, equals:*

$$L(y,\hat{y}) = \begin{cases} 0, & \textit{if } \hat{y} = y \\ 1, & \textit{otherwise.} \end{cases}$$

Note that $0-1$ loss has perfect negative correlation with accuracy: the mean $0-1$ loss reflects the fraction of incorrectly labeled objects, instead of the correctly labeled ones.

Another popular function is the (mean) squared error.

Definition 2.1.3 *The squared error of a prediction* $\hat{y}$*, compared to the true value* y*, equals:*

$$L(y,\hat{y}) = (y-\hat{y})^2$$

Note that the squared error can both be applied when the output is a real number (regression) or categorical (classification). In the classification setting used in this work, the prediction $\hat{y}$ and true value y are hot-encoded vectors. That means a vector with the length $n_{classes}$ expresses the class i

by the position of a 1 entry in $\hat{y}_i$, with the other entries 0. For example, when having nine classes (0–8), class 0 would be represented by the hot-encoded vector $y = (1,0,0,0,0,0,0,0,0)^\mathsf{T}$. Notice that prediction $\hat{y}$ may contain a probability assigned to every class, with $\hat{y}_i \in [0,1]$ for every class i and $\sum_i \hat{y}_i = 1$.

Another popular choice in deep learning is *categorical cross entropy*, also called the *log loss*. Again, we express the prediction for a sample x in vector $\hat{y}$, with entry $\hat{y}_i$ corresponding to the probability of class i:

Definition 2.1.4 *The categorical cross entropy of prediction $\hat{y}$, compared to the true value y, equals:*

$$L(y,\hat{y}) = -\sum_i y_i \log \hat{y}_i$$

In addition to the different loss functions, we use one other metric to evaluate the performance of the attack: guessing entropy, which is introduced in Section 2.4.1. When searching for the best machine learning algorithm, it is good to keep in mind the ”No Free Lunch” theorem. With this theorem, Wolpert [21] showed that there exists no universally superior machine learning technique. So, instead of looking for the absolute best learning algorithm, we should try to find the algorithm that performs well for a certain type of problem [20].

Bootstrapping

Bootstrapping is a sampling technique based on drawing with replacement, proposed by Efron [22]. It has many applications in statistics, as it allows for estimation of measures of accuracy, such as bias and variance. As input, the algorithm takes original dataset X, which has N samples (e.g., $X = (x_1, \ldots, x_N)$). Then, a bootstrap sample consists of N^* samples of X randomly drawn with replacement: $X^* = (x_1^*, \ldots, x_{N^*}^*)$. By convention, the new sample has the size of the original dataset: $N^* = N$. For a large enough N, this means the probability of bootstrapping to the exact same dataset is extremely small: some traces will appear multiple times, and others not at all, in X^*.

To distinguish between different bootstrapped sets, we refer to such sets as $X^{1*} = (x_1^{1*}, \ldots, x_{N^*}^{1*})$, $X^{2*} = (x_1^{2*}, \ldots, x_{N^*}^{2*})$, and so on. To estimate classifier's characteristics, we employ bootstrapping to create different training sets. To allow this scenario, we made the following changes to Efron's algorithm:

- As a classifier is trained using one bootstrap sample, we require all classes to be present in every bootstrap sample. Formally, with $y(x_j)$ indicating the true class label of sample x_j, we require all classes to be represented in the set $\{y(x_1^{i*}), \ldots, y(x_{N^*}^{i*})\}$ for each bootstrap sample X^{i*}. We redraw until this condition is satisfied. For QDA (see Section 2.1.1), we require a minimum of two samples per class to estimate the correlation between features (covariance matrix).
- In some experiments, we measure the effect of having smaller/larger training sets. There, the size of each bootstrap N^* is varied and not fixed to N.

The procedure is described in Algorithm 1.

Algorithm 1: Generating l bootstrapped sets of size N^* from dataset X.

```
for $i = 1$ to $l$ do
  repeat
    for $j = 1$ to $N^*$ do
      $x_j^{i*} \leftarrow$ random sample from $X$
    end for
  until $\{y(x_1^{i*}), \ldots, y(x_{N^*}^{i*})\}$ contains all classes
  $X^{i*} = (x_1^{i*}, \ldots, x_{N^*}^{i*})$
end for
return $X^{1*}, \ldots, X^{l*}$
```

2.1.4. Complexity, Bias, and Variance

Classifiers can have different complexities. For example, a random forest with five decision trees is more complex than one of these trees individually, and a random forest with ten trees is even more complex. However, it is harder to compare the complexity of two entirely different classifiers, e.g. random forest versus QDA.

Apart from changing hyperparameters, models can also get more complex when the feature vectors grow. Although adding more features can help classification, as more data is included, many classifiers suffer from the *curse of dimensionality*. This means that learning meaningful patterns gets increasingly difficult as more features are included. This can happen due to various reasons: it's harder to estimate distributions or covariance matrices, most of the data is in the rim, larger parts of the feature space are not covered, and noise can be increased [19, 23].

Vladimir Vapnik and Alexey Chervonenkis defined a precise measure for a classifier's complexity: the VC-dimension [24]. Informally, with k classes, the VC-dimension represents the largest number of vectors h that can be separated in all k^h possible ways. The VC-dimension of many classifiers cannot be calculated precisely, so in practice often only a theoretical bound can be found [25, 26]. Using conservative assumptions (e.g., worst-case separability or configuration), these bounds are typically very loose and are therefore not practical in most deep learning contexts [20].

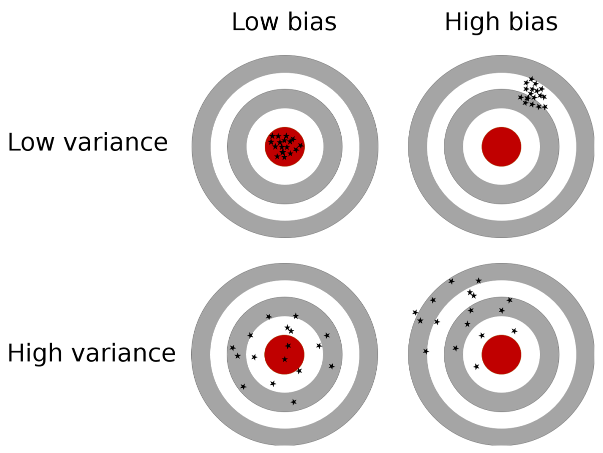


Figure 2.2: Bias and variance: ideally, a classifier has low bias and low variance. Bias indicates how accurate the predictions are on average, compared to the actual labels. Variance marks the how widely the predictions are spread among different instances.

Therefore, we avoid general comparison and rather look at complexity in a more practical setting: using known models and varying their number of parameters by changing their hyperparameters (e.g., the number of trees). Fine-tuning a model complexity using hyperparameters is an important task in finding a good classifier [27]. When a model is too simple, it may underfit: not capturing the underlying structure in the data properly. When a model is too complex, it may overfit: following the structure of the training data too precisely, lacking a generalization of the underlying structure.

Bias and Variance

To make these terms more precise, it is useful to decompose the loss (see Section 2.1.3) in bias and variance. The *bias* of a model comes from erroneous assumptions in the learning algorithm: how "wrong" the prediction is on average. A high bias typically reflects underfitting: the learning algorithm is too simple to properly learn the underlying pattern. The *variance* is an error from sensitivity to (small) changes in the training set. It indicates how different realizations of the same learning algorithm vary for a given point. A common graphical depiction of bias and variance interplay is given in Figure 2.2.

Often, models with low bias are more complex (e.g., a high number of trees in a random forest), so they can accurately represent the training set. In contrast, models with higher bias tend to be relatively simple (such as a single decision tree or naive Bayes), but may produce lower variance predictions when applied beyond the training set. This trade-off in the choice of the model complexity is called the *bias–*

variance dilemma [19]. Figure 2.3 depicts this trade-off: the optimal model complexity has the lowest total error, based on both bias and variance.

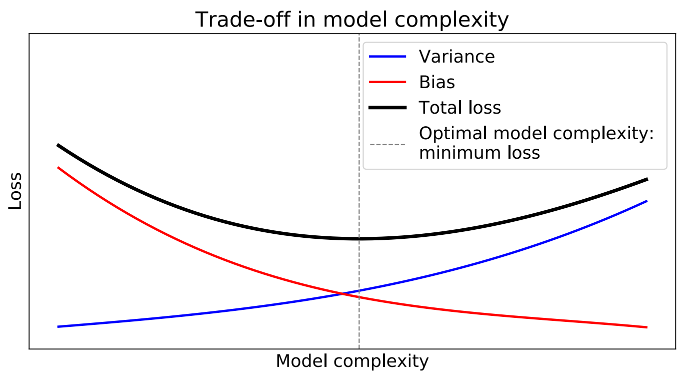


Figure 2.3: The complexity trade-off between bias and variance. Total loss is the sum of bias and variance. The model with optimal complexity is the one that minimizes the total loss.

Bias–Variance Decomposition

The loss function determines how to decompose the loss into bias and variance. Let y denote the true value of a predicted label for a certain test example x. Then, the loss function $L(y,\hat{y})$ measures the cost of predicting $\hat{y}$ when the true label equals y. The aim is to minimize the loss, i.e., to find the models that minimize the average loss over all examples X.

The decomposition of two loss functions are used in this work: (mean) squared error and $0-1$ loss. We repeat the definitions from 2.1.3 here:

Definition *The squared error of a prediction $\hat{y}$, compared to the true value y, equals:*

$$L(y,\hat{y}) = (y-\hat{y})^2$$

Definition *The $0-1$ error of a prediction $\hat{y}$, compared to the true value y, equals:*

$$L(y,\hat{y}) = \begin{cases} 0, & \textit{if } \hat{y} = y \\ 1, & \textit{otherwise.} \end{cases}$$

Domingos [28] defined a loss decomposition for $0-1$ loss, which we will use to analyze the classifiers in side-channel attacks. We introduce several building blocks, then show the decomposition itself.

Definition 2.1.7 *The optimal prediction of a sample x is the prediction y^* which minimizes $E_y[L(y,y^*)]$, over all possible values y, weighted by their probabilities given x.*

More intuitively, based on a sample x, the optimal prediction y^* is the best prediction that a model can make – even when it is not the correct label y. This accounts for Bayes' error: when classes overlap, classification errors are unavoidable, but y^* is the best label as it is weighted over the probabilities of x. When sample x has class overlap with a more likely class (based on the feature vector), this difference between the true value y and the optimal prediction y^* is called *noise*:

Definition 2.1.8 *The noise of a sample x is $N(x) = E_y[L(y,y^*)]$.*

The noise term is also called Bayes' error or the unavoidable error. It is independent of the classification technique. When there is no class overlap for x, Bayes' error for that sample is 0 and $y^* = y$.

A machine learning model aims to learn the mapping $f(x) = y^*$ as well as possible. Let D be a set of training sets. Because a learning algorithm produces a model based on the training set it has been given, we average the error over all training sets D. Thus, we want to minimize the expected loss $E_{D,y}[L(y,\hat{y})]$ for our model.

Definition 2.1.9 *The main prediction of a sample x for a loss function L and training sets D is $y_m^{L,D} = argmin_{y'} E_D[L(\hat{y}, y')]$.*

Put in words, the main prediction $y_m^{L,D}$ is the 'average' prediction, based on all training sets D and a loss function L. When D and L are unambiguous, we denote the main prediction as y_m. If L is squared loss, the main prediction is the mean of the predictions; in $0-1$ loss, it is the most frequent prediction among all classifiers trained on the different sets in D. Using the main prediction, we can now define the bias and loss of a classification technique.

Definition 2.1.10 *The bias of a predictive model on sample x is $B(x) = L(y^*, y_m)$.*

The bias indicates the loss penalty between the main prediction, with respect to the optimal prediction.

Definition 2.1.11 *The variance of a predictive model on sample x is $V(x) = E_D[L(y_m, \hat{y})]$.*

The variance indicates the average loss penalty between the predictions $\hat{y}$ and the main prediction y_m.

As we have defined the bias, variance and noise of a sample, we can decompose the loss incurred by a single prediction $\hat{y}$.

Definition 2.1.12 *The loss $E_{D,y}[L(y, \hat{y})]$ for a prediction $\hat{y}$ can be decomposed into bias, variance and noise as follows:*

$$\begin{aligned} E_{D,y}[L(y,\hat{y})] &= c_1 E_y[L(y,y^*)] + L(y^*, y_m) + c_2 E_D[L(y_m, \hat{y})] \\ &= c_1 N(x) + B(x) + c_2 V(x) \end{aligned}$$

where c_1 and c_2 are multiplicative factors that differ depending on the loss function.

In the case of squared loss, then $c_1 = c_2 = 1$.

In $0-1$ loss, Domingos cleverly defined c_1 and c_2 such that c_1 accounts for Bayes' error and variance, keeping the expected value of the loss intact. c_1 accounts for Bayes' error, caused by a mismatch between the optimal prediction and true label: $c_1 = P_D(\hat{y} = y^*) - P_D(\hat{y} \neq y^*) P_t(\hat{y} = y | y^* \neq y)$.

When a prediction $\hat{y}$ in the $0-1$ multiclass setting different from the main prediction y_m, this can be either good or bad. When the main prediction is correct, i.e., $y_m = y^*$, a different prediction will be wrong by definition: so, the variance increases the total loss. However, when the main prediction is incorrect, a different prediction $\hat{y}$ will reduce loss only when it is the optimal prediction, so $\hat{y} = y^*$. This reasoning leads to the following multiplicative factor c_2:

$$c_2 = \begin{cases} 1, \text{ if } \ y_m = y^* \\ -P_D(\hat{y} = y^* | \hat{y} \neq y_m), \text{ otherwise.} \end{cases}$$

This means that we can distinguish between the unbiased variance V_u and biased variance V_b. When the main prediction is optimal ($y_m = y^*$), all predictions deviating from the main prediction contribute to the loss in V_u. Alternatively, if the main prediction is non-optimal ($y_m \neq y^*$), only the predictions indicating the right class count towards reducing the loss via the biased variance V_b.

For a more detailed description and proofs, we refer to the works of Domingos [28, 29]. We will come back to the bias–variance decomposition in Chapter 4, where the relation between mean squared error (MSE) and guessing entropy (GE) is shown. There, we explore the bias and variance of several classifiers used in SCA in detail.

2.2. Deep Learning

Deep learning (DL) refers to machine learning techniques that employ artificial neural networks (NNs). Such networks consists of *neurons*, each of which computes a certain function based on its inputs. The simplest and most widespread forms are called *feed-forward networks*, in which there are no cycles or loops among neuron connections. Typically, neurons are organized in layers. The first layer does not compute a function, but delivers the data samples: each feature is represented by one neuron in the *input layer*. A layer in which every neuron is connected with every neuron of the previous and/or next layer is called *fully connected*. Finally, the last layer is called the *output layer*. In classification tasks, each class is represented by one neuron in the output layer.

A simple network is shown in Figure 2.4. This kind of structure is referred to as *multi-layer perceptron* (MLP): it consists of an input layer, one or multiple *hidden layers*, and an output layer. All hidden layers and the output layer are fully connected.

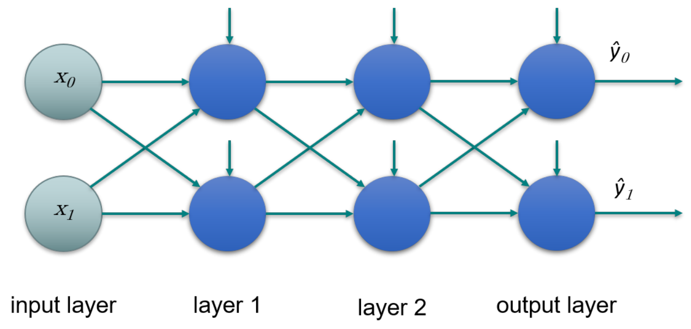


Figure 2.4: A simple multi-layer perceptron: one input layer, two hidden layers, and one output layer, each consisting of two neurons

Deep learning is an emerging subject and consists of many techniques, combining neurons in many different ways. This background chapter focuses on the basics, which we use in our research. The rest of this section is organized as follows: section 2.2.1 explains the computational power of neurons and popular techniques to achieve non-linear outputs, Section 2.2.2 discusses the training of networks using backpropagation, loss functions, and optimizers. Strategies to prevent overfitting are described in section 2.2.3, and several kinds of layers (e.g., convolutional) are introduced in section 2.2.4. Lastly, some deep learning architecture design paradigms are discussed in section 2.2.5.

2.2.1. Neurons and Activation Functions

The elementary unit of a neural network is a neuron. It computes a function σ, and forwards its output to the next layer. The general form of its function is defined as:

$$\sigma(w_1 \cdot x_1 + \cdots + w_q \cdot x_q + b) \tag{2.9}$$

where x_i represents its ith input, w_i the weight of that input, m the number of inputs the neuron receives, and b the *bias* of the neuron–not to be confused with the bias in Section 2.1.4. The weights and bias are called the *parameters* of the neuron: they will be tweaked to optimize the network's output. Figure 2.5 shows a single neuron with three inputs.

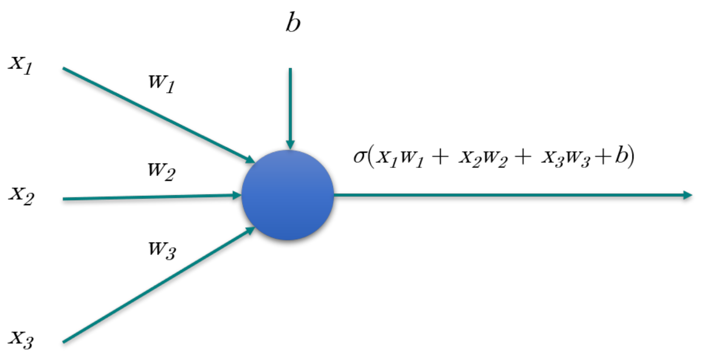


Figure 2.5: A neuron, outputting the result of a function based on three weighted inputs and its bias.

The function σ is called the neuron's *activation function*. The input of the activation function is the weighted sum of the inputs, plus the bias. This is linear; thus, non-linear behavior of the neural network is induced by the activation function. In this work, only two activation functions are used:

- *Rectified linear unit (ReLU)* which outputs its input if it is positive, zero otherwise:
$$\sigma(x) = max(0, x) \tag{2.10}$$
- *Softmax*, also called the normalized exponential function, is used for the output layer. It normalizes the output of all neurons such that each component will be in the interval $[0, 1]$, and their sum is equal to one:
$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \text{ for } i = 1, \ldots, k \text{ and } \mathbf{z} = (z_1, \ldots, z_k) \in \mathbb{R}^k \tag{2.11}$$
with k classes, each represented by one neuron.

The neurons are combined to form a network like the multi-layer perceptron (MLP) shown in Figure 2.4. The input layer is the representation of the feature vectors: the measurements in SCA. For each feature, there is one neuron in the input layer. These are fed into the hidden layers, which typically use the ReLu activation function. Finally, the output layer represents the estimated probability per class, using Softmax activation to ensure all probabilities are in the interval $[0, 1]$ and sum to one.

2.2.2. Training

The learning algorithm in deep learning is backpropagation, which tries to minimize some loss function. This error function defines some distance between the correct label and outputs predicted by the network. The backpropagation algorithm makes use of the gradient of the loss function with respect to the neurons' weights. As the gradient expresses in which direction these parameters should be changed (e.g., increase or decrease) to reduce the loss, this can be used to update the weights. This is done in iterations, which are also referred to as *epochs*.

There are several methods that describe how to update the weights precisely. An algorithm to update the neurons' weights is called an *optimizer*. The stochastic gradient descent (SGD) was introduced in 1951, updating the weights simply in the direction of their respective gradient [30]. The *learning rate* was used to influence the learners speed. When the learning rate is too low, the weights are updated only slightly every step, therefore taking a long time to converge to a local minimum. If the learning rate is too large, the weights will be updated too heavily and oscillate throughout the steps, mostly updating the weights with a large gradient.

To avoid this oscillation and still quickly converge, several modifications to the SGD algorithm have been proposed. The *Adam* optimizer combines several existing tricks (AdaGrad, RMSProp), using adaptive learning rates per parameter [31]. Adam is considered state-of-the art and preferred over other methods [32], and therefore used in this work.

Then, the question remains which error to minimize. Two error functions are popular in deep learning: the mean squared error (MSE) and categorical crossentropy (CCE). We have introduced those losses in section 2.1.3. We show the relation between MSE and Guessing Entropy in Section 4.2. However, categorical crossentropy is a more popular choice in classification problems. Therefore, this work uses both MSE and CCE as loss functions for training neural networks.

As computing the gradient involves (many) simple operations, the back-propagation algorithm can be highly parallelized. The explosive growth in computational power since the beginning of this century has allowed for training of increasingly big architectures [20].

2.2.3. Regularization

Neural networks, especially those with a larger number of parameters, are prone to overfitting [20, 33]. In such networks, the model weights are expressive enough to describe the training data exactly. This is known as *memorization* – in contrast to the desired *generalization*, where the model also performs well on new data from the test set. *Regularization* techniques attempt to reduce overfitting, by adding or removing information (e.g., noise), penalizing complexity in the learning algorithm, or stopping the training phase when overfitting is suspected to happen. The following subsections describe arguably the most common techniques: dropout, L2-regularization, and noise addition. Notice that batch normalization layers (see Section 2.2.4) also have a small regularization effect.

Dropout

Dropout is a popular regularization method, based on dropping out neurons randomly during the training phase [34]. At each iteration, a fraction p of neurons are randomly selected and kept in place; the other $1-p$ are temporarily removed. For the next iteration, the removed neurons are re-added and a new set of neurons is randomly selected for temporary removal.

During training, the neurons' weights are updated and neurons get specific roles in the context of the network. As neurons increasingly rely on these specializations – called complex co-adaptations – they can lead to a fragile model. It will fit tightly to the training data and hardly generalize to the underlying patterns. When using dropout, these specialized neurons will be dropped once in a while, and other neurons will also capture the representations needed to make the predictions. As a result, the specific weights of neurons are less influential in the network's predictions, so the overfitting is reduced. In literature, it is common to keep a $p = 0.5$ fraction of neurons [34, 35], which we follow in this work.

L2-Regularization

As explained in the previous section, deep learning relies on gradient descent to minimize some loss function. Some regularization techniques add another term to the total loss: some measure of the model complexity. So, instead of only minimizing some loss function $L_{train}(\mathbf{w})$, another term is added to express model complexity:

$$J(\mathbf{w}) = L_{train}(\mathbf{w}) + \lambda\Omega(\mathbf{w}) \tag{2.12}$$

where $\Omega(\mathbf{w})$ indicates a penalty function based on the model weights $\mathbf{w}$, and some constant λ indicates how heavy this penalty should contribute to the total loss. Choosing weights $\mathbf{w}$ by minimizing the total loss $J(\mathbf{w})$ will result in a tradeoff between fitting the training data and having a simple model.

Several penalties $\omega(\mathbf{w})$ have been proposed. A popular choice is L^2-regularization, also known as weight decay, which simply employs the L^2 norm as penalty function:

$$J(\mathbf{w}) = L_{train}(\mathbf{w}) + \lambda\mathbf{w}^\top\mathbf{w} \tag{2.13}$$

It has been applied in many machine learning techniques and is a popular choice for neural network regularization [20, 36].

Data Augmentation and Noise Addition

Data augmentation refers to generating new samples by adding information to the original data. When the training set is small, similar samples generated using existing data and randomness may help the learning process. Also, when additional domain knowledge is available, it may be incorporated in the samples.

Alternatively, noise maybe randomly added to the normal samples during every training iteration. This can improve generalization and performance on the test set [37]. Noise drawn from a Gaussian distribution centered around 0 seem to provide the best results [38]. Adding noise to training data and L^2-regularization is strongly related and can be, in fact, reduced to each other [39]. Comparable with other regularization techniques, this randomization can avoid overfitting to the training samples, and force the classifier's model to be more generalized.

2.2.4. Layers

Apart from fully connected layers, many other combinations of neurons can be used as a layer in a neural network. This section presents the techniques that are relevant in this work.

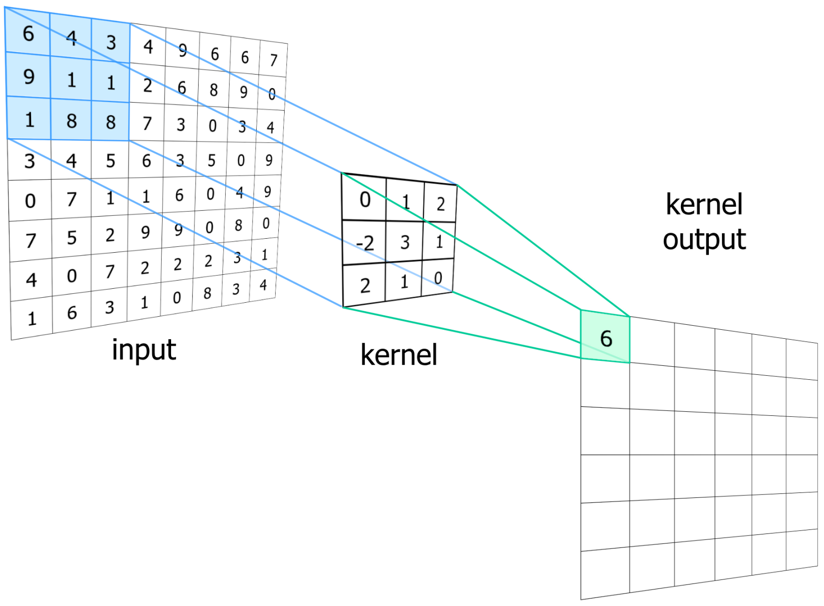


Figure 2.6: A convolutional layer used in image recognition. On the left, we see the input data, with the current *receptive field* highlighted. In the middle, we see the *kernel* with its trained parameters (weights) and on the right, we see the convolutional layer output for this kernel. The output is the weighted sum of inputs from the receptive field: $0 \cdot 6 + 1 \cdot 4 + 2 \cdot 3 + (-2) \cdot 9 + 3 \cdot 1 + 1 \cdot 1 + 2 \cdot 1 + 1 \cdot 8 + 0 \cdot 8 = 6$.

Convolution

A *convolutional* layer consists of learnable filters, also called *kernels*. A filter is a set of weights, just like a normal neuron, but only considers one subset of the input at the time: this part of the input is called the *receptive field*. The kernel size specifies the dimensions of the receptive field, defining how many parameters a filter embeds. For this specific subset of the input, the kernel's output will be, as before, the weighted sum of the inputs plus the bias. Again, an activation function – typically ReLu – is applied to achieve non-linearity. Each filter is convoluted across the input, with a particular step size in every dimension. This step size is called the *stride*.

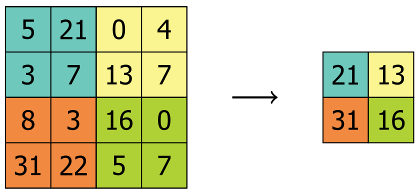
Figure 2.6 shows a filter in action: here, an 8 x 8 input is processed by a 3 x 3 filter. As the stride is one, the resulting output has dimensions 6 x 6 per kernel. Instead of learning a mapping between all inputs and (many) neurons, convolutional layers only learn the filter weights. Therefore, these layers allow for dealing with high-dimensional data with relatively few parameters. Due to their popularity, networks containing such layers are called *convolutional neural networks* (CNN).

Pooling

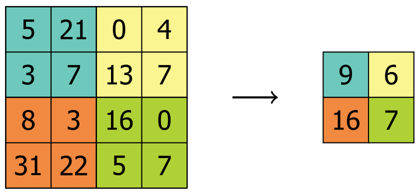
A *pooling* layer learns no parameters, but rather slides through its input data and combines subsets of it by applying some function. Its most common forms are *max pooling* and *average pooling*, where respectively the maximum or average value over the input subset is outputted. Max pooling seems favorable in many scenarios and is used more commonly [35, 40, 41].

Often, the activated outputs of a convolutional layer are pooled. Convolutional layers output the precise positions of the patterns they activate on, so small movements of these patterns will result in a different output. Pooling layers reduce this variance. Figure 2.7 shows a pooling layer with pool sizes 2 x 2, and the outputs of max and average pooling on a 4 x 4 input.

Pooling can also be used to reduce model complexity. By decreasing the output size, the next layer needs less parameters to be trained. The depth, which is defined by the number of filters in the previous layer, remains unchanged: the output of each filter is pooled separately.



(a) Max pooling.



(b) Average pooling.

Figure 2.7: A max pooling and average pooling layer, applied to a 4 x 4 input with pool size 2 x 2.

Batch Normalization

Batch normalization (BN) is a technique to normalize the activations of the previous layer per batch [42]. It normalizes to a mean of zero and standard deviation of one. It can be used after the input layer (producing similar results to feature scaling) or after the activations of deep layers. Batch normalization improves the learning speed and performance of neural networks.

The previous layer's m-dimensional outputs $x = (x^{(1)}, ..., x^{(m)})$ are normalized by subtracting the batch mean μ_B and dividing by the batch standard deviation σ_B (plus a small $\epsilon > 0$ to avoid dividing by zero):

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \epsilon}} \tag{2.14}$$

for all dimensions $k \in [1, m]$ and all batch samples $i \in [1, |B|]$. $\mu_B^{(k)}$ and $\sigma_B^{(k)^2}$ are the mean and variance per dimension.

After this transformation, the weights in the next layer are no longer optimal. To account for this, a batch normalization layer has two parameters. After the normalization in Eq. 2.14, these outcomes are scaled and shifted according to γ and β, respectively:

$$y_i = \gamma \hat{x}_i + \beta \tag{2.15}$$

where y_i denotes the final output of the BN layer, given x_i as inputs from the previous layer. Parameters γ and β are trained using backpropagation, together with the other parameters of the network. In addition to improving the learning speed, batch normalization has a slight regularization effect. As it computes the scaling parameters batch-wise, these will vary slightly among different batches.

2.2.5. Design Strategies

Deep learning has been applied in many fields, notably computer vision [43]. Since computing power allows for the training of large convolutional neural networks, they have consistently beaten other image recognition classifiers [20]. As the research in computer vision has brought many advances to deep learning, we mention a notable CNN architecture that provided the fundamentals of architectures in different areas. It is infeasible to try all possible architectures and see which one performs best, when using a large number of parameters. Therefore, it is convenient to be able to build on the works of others and try to apply proved design principles. The "no free lunch" theorem also applies to deep learning: an architecture optimized for one problem will not be universally superior as a classification technique.

VGG is a network proposed by Simonyan and Zisserman [41], named after the Visual Geometry Group at the University of Oxford. It has the following design principles:

- Small kernel size: 3 x 3 for every layer;
- Max pooling with 2 x 2 windows, with stride 2. It is applied 5 times, after some convolutional layers (grouped together in blocks);
- Increasing number of filters per layer: doubled after every max pooling layer;
- Convolutional blocks are added until the spatial dimension is reduced to 1;

- After the last max pooling layer, the output is flattened and followed by two fully connected layers with 4096 neurons each;
- After the fully connected layers is the output layer;
- The convolutional and fully connected layers use ReLu activations, the output layer uses Softmax to normalize the predictions.

The VGG design principles have also been used successfully using 1-dimensional convolution, for example in speech recognition [44].[3]

2.3. Cryptography

Cryptography aims to provide secure means of communication in the presence of adversaries. An important goal, that of *information privacy*, is that two parties can exchange messages privately, so that the adversary learns nothing about what was communicated. Since the Mid-20th century, cryptography has advanced rapidly, stimulated by the need to communicate privately in the Second World War [45].

Cryptographic primitives, in which both parties know and use the same key, are known as *symmetric-key algorithms*. Symmetric-key algorithms which take a fixed-length bitstring as input are called *block ciphers*. Although block ciphers are typically computationally efficient, they require both parties to know the same key. So, before using a block cipher, the key needs to be exchanged securely between the parties. In some settings, like the Internet, this is solved by using *hybrid cryptosystems*: an *asymmetric algorithm* (or *public key algorithm*) establishes a first means of secure communication, helping the parties to agree on a key for the block cipher. Then, the faster block cipher is used to encrypt the subsequent communication.

An important desideratum in cryptography is *Kerkhoffs's principle*, which states that the a cryptographic system should be secure, even if everything about the system is public knowledge, except the key. This law applies to this work as well: the cryptographic algorithms under attack are public knowledge. The goal of an attack is to retrieve the secret key.

2.3.1. Advanced Encryption Standard

One of today's most used block ciphers is the Advanced Encryption Standard (AES). De facto, IoT devices are equipped with the AES block cipher to secure their communications with the outside world. After a public competition and selection process, a subset of the Rijndael block cipher was selected as the standard by the U.S. National Institute of Standards and Technology [2]. As input, the algorithm takes message blocks of 128 bits, and a key of either 128, 192, or 256 bits. The key size used can be indicated after the algorithm's name, e.g. AES-256. The block size of the outputted ciphertext is 128 bits. A block can be seen as a 4×4 matrix with 8-bit entries which AES operates on. There are four primitive operations: *AddRoundKey*, *SubBytes*, *ShiftRows*, and *MixColumns*. After the first round the key is added, several rounds of SubBytes, ShiftRows, MixColumns, and AddRoundKeys are executed. Depending on the key size of 128, 192, or 256 bits, these operations are repeated in 10, 12, or 14 rounds, respectively. In the last round, the MixColumns operation is omitted. Table 2.1 presents an overview of the AES operations.

So far, the best known key-recovery attacks on full AES have time complexity $2^{126.01}$and $2^{254.27}$ for AES-128 and AES-256, respectively [46]. These are negligible improvements over brute-force attacks which try every single possible key, so an attack would take billions of years on current and anticipated hardware. In conclusion, when AES and its key generation are implemented correctly, it is assumed to be secure against black box attacks.

Memory and computation power are limited in typical IoT devices. AES allows for efficient implementations; only 768 bytes of code suffice for AES-128 on a 8-bit architecture [47]. These implementations typically rely on the (inverse) S-Box in a lookup table in memory, and simple byte operations to conduct

[3]Note that the original VGG design was published before Batch Normalization (BN) hads been invented. Using BN, VGG's performance can be further improved, an example application is listed here: `https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c`

the other steps in the algorithm. After each step, in particular the S-Box operation, the intermediate result is stored in memory.

Operation	Description	Repetition
AddRoundKey	XOR with round key 1	
SubBytes	S-Box substitution	9, 11, or 13 rounds
ShiftRows	Cyclic shift of rows	
MixColumns	Mixing operation on each column	
AddRoundKey	XOR with round key 2, 3, …	
SubBytes	S-Box substitution	
ShiftRows	Cyclic shift of rows	
AddRoundKey	XOR with final round key	

Table 2.1: AES operations

2.4. Side-Channel Attacks

At the time of writing, AES is widely adopted and no efficient key-recovery attack on full AES is known. However, side-channel analysis (SCA) focuses not only on the observed communication (e.g., plaintext and ciphertext messages), but also on unintended leakage over one or multiple channels. For example, power consumption or electromagnetic leaks of the chip may enclose some information about the computed encryption or decryption. There have been many successful side-channel attacks on different AES implementations [3, 48].

Kocher *et al.* [4] introduced simple and differential power analysis (SPA and DPA) in 1999. The power consumption of a circuit implementing AES-predecessor DES was measured during encryption. SPA relies on visual inspection of the measurements, which can provide information about the operation as well as the used key. As the measurements may also correlate with key material in a more subtle manner, manual inspection can be hard. DPA relies on statistical analysis of the traces, filtering out some of the noise, and providing a more sophisticated attack. The first complete side-channel attacks using electromagnetic radiation were demonstrated in 2001 [49, 50].

2.4.1. Profiled Analysis

The side-channel community moved forward and found that *profiled* attacks are more powerful than SPA and DPA. Profiled side-channel analysis represents the most powerful side-channel attack as we assume an attacker is in possession of a clone device which he uses to obtain knowledge about the device while running. This is known as the profiling phase, during which a model is build. Later, in the attack phase, that knowledge is used to mount the attack on a target device.

Chari *et al.* [5] introduced the *template attack* (TA), which profiles measurements for some intermediate value associated to a plaintext byte and key byte. It assumes the traces follow a multivariate Gaussian distribution per class. If this assumption holds, and an unlimited number of profiling traces is available, Template Attacks are the optimal attack from the information theoretic perspective. This profiling technique is equivalent to Quadratic Discriminant Analysis (QDA) as introduced in Section 2.1.1.

In the last decade, machine learning in profiling side-channel analysis transitioned from an exotic approach to a well-established technique for powerful analysis. The first results with machine learning were promising but template attack, as the most powerful profile from the information theoretic point of view [5], was the first choice for most of the SCA community. As researchers started to explore more diverse attack scenarios, they recognized settings where machine learning could perform even better than template attacks.

Recently, deep learning started attracting more attention in numerous research domains as well as in SCA. The results with deep learning were very interesting: such techniques could outperform all other methods in numerous scenarios. Most notably, protected implementations have been broken where other attack techniques are much less effective [51, 52]. Yet, despite all these positive results, there is still much to be done and, especially, understood.

For instance, deep learning techniques have many hyper-parameters one needs to tune, which is a computationally intensive process with no guarantee that the best hyper-parameters are obtained. Comparing this with the simplicity of a template attack that has no hyper-parameters to tune makes the situation even more complex. Indeed, we are aware of powerful SCA techniques, but often we do not know how to employ them in the best possible way. Naturally, this is not something encountered only in SCA: other domains suffer from the same problems and also rely on extensive experimental procedures in order to obtain the best results possible.

A natural question is whether there are techniques that provide us with better insight into the performance of machine and deep learning. While such insights are far from providing a complete picture, they can give some useful results and improve the process, which in turn can yield a more powerful attack. Additionally, we can increase the problem understanding and its difficulty for a certain machine learning algorithm.

We consider a scenario where a powerful attacker has a clone device that is of the same type as the device to be attacked. Then, he can use that clone device to obtain knowledge he later uses to attack the target device. Let calligraphic letters ($\mathcal{X}$) denote sets, capital letters (X) denote random variables over $\mathcal{X}$, and the corresponding lowercase letters (x) denote their realizations. The symbol $E_z[]$ denotes the expected value over z, and symbol P_z the probability over z.

From a profiling device, the attacker is able to obtain a set of N profiling traces T_p in order to characterize the leakage. By combining the profiling traces and a known secret key k_p^*, the attacker can calculate the leakage model $Y(T_p, k_p^*)$. Since the attacker has N profiling traces, he is able to obtain N pairs (x_i, y_i) to build the attack model f. Here, y_i denotes the label corresponding to trace x_i; we omit the index i when possible. Additionally, each trace x consists of n points of interests (features).

Once this phase is finished, the attacker measures an additional M traces T_a from the device under attack and uses the function f in order to obtain the unknown secret key k_a^*. More specifically, for each attack trace, the attacker uses $f(x)$ to either predict the most likely class $\hat{y}$, or predicts a probability vector for all classes $\hat{\mathbf{p}}(x) = [\hat{p}_0, \ldots, \hat{p}_{|\mathcal{Y}|-1}]$ with $|\mathcal{Y}|$ the number of classes in the leakage model and $\sum_{j=0}^{|\mathcal{Y}|-1} \hat{p}_j = 1$. The outputted classes or class probabilities are used to obtain information about k_a^*. Although it is usually assumed that the attacker has an unlimited number of traces available during the profiling phase, the attacker is bounded due to practical limitations.

When profiling the leakage, one must select the appropriate leakage model, which will determine the number of possible outputs. The Hamming weight (HW) and Hamming distance (HD) leakage models are often used as they have a reduced number of classes, which results in reduced training complexity. $HW(y)$ indicates the number of bits in y that are 1. $HD(y_1, y_2)$ indicates the number of bitwise transitions between y_1 and y_2 (i.e., number of bits flipped from 0 to 1 or vice-versa).

Unfortunately, the classes obtained from those models are imbalanced, which can cause problems for profiling SCAs [53]. A common alternative is to profile directly on the intermediate value, which requires a bigger profiling set but does not suffer from imbalance. When considering AES, HW results in $|\mathcal{Y}| = 9$ classes and intermediate value results in $|\mathcal{Y}| = 256$ classes as the attack is computed byte-wise.

To evaluate the performance of the attack, a common option is to use *guessing entropy* (GE) [54]. The guessing entropy metric gives the average number of key candidates the attacker needs to test to reveal the secret key after conducting a side-channel analysis. More formally, given T_a traces in the attacking phase, an attack outputs a key guessing vector $\mathbf{g} = [g_0, g_1, \ldots, g_{|\mathcal{K}|-1}]$ in decreasing order of probability where $|\mathcal{K}|$ denotes the size of the keyspace. To calculate the key guessing vector $\mathbf{g}$ over a number of samples, we use the following:

$$g_i = \sum_{j=1}^{T_a} \log(\hat{p}_{ij}), \tag{2.16}$$

where $\hat{p}_{ij}$ is the estimated probability for key candidate $i = \{1, \ldots, |\mathcal{K}|\}$ using sample j. Guessing entropy is the average position of k_a^* in $\mathbf{g}$ over a number of experiments. As we will investigate in this paper, guessing entropy is influenced by the profiling and attack sets that the attacker uses:

$$GE(T_p, T_a) = E_{T_p,T_a}[i] \text{ with } i \text{ such that } g_i = k_a^*. \tag{2.17}$$

From the perspective of the attacker, the lower GE the better; when the GE drops to 1, the first key (byte) candidate is the correct key (byte).

2.4.2. Countermeasures

With the growth of IoT and increasing security awareness, cryptographic research has led to countermeasures to harden devices against side-channel attacks. These can be divided in two categories: masking and hiding.

Masking randomizes the relation between sensitive data and leakage. A popular countermeasure in AES is masking the S-Box output by XORing with Boolean values [55]. So, instead of storing the output $\texttt{Sbox}[input_i \oplus roundkey_i]$ for byte i, a mask is added to the output:

$$\texttt{Sbox}' = \texttt{Sbox}[input_i \oplus roundkey_i] \oplus mask_i \tag{2.18}$$

Where the masks are 8-bit values that can be fixed, rotated, or random [3, 55]. When executing the next operation, results are unmasked and processed further. As a result, the device's power consumption or electromagnetic emission will be decorrelated with the S-Box output. However, the masked Sbox-output may still be profiled in a so called *higher-ordered attack*, which incorporates the existence of masks.

Hiding makes the power/radiation random, or constant, per point of interest. By introducing random delays during encryption or decryption, feature-wise leakage is reduced. However, distinctive patterns are still present somewhere in the measurements – and when a learning technique is able to handle this shift (e.g., CNN), those patterns may still be found.

2.4.3. Public Datasets

In our experiments, we use four publicly available datasets corresponding to characteristic cases: measurements of an unprotected implementation with low noise, high noise, and those protected with a random delay and a masking countermeasure. These datasets are commonly used in related work, to benchmarks the performance of attack techniques. All experiments consider the AES algorithm and either the HW leakage model or the intermediate value model. We denote plaintext with PT and ciphertext with CT.

DPAcontest v4

DPAcontest v4 contains measurements of a masked AES-256 software implementation on a smart card [56].[4] Since the masking leaks the first-order information [57], we can consider the mask to be known and turn the implementation into an unprotected scenario. The most leaking operation in this implementation is the processing of the S-Box operation. We attack the first round and our leakage model equals:

$$Y(k^*) = \texttt{Sbox}[PT_1 \oplus k^*] \oplus \underbrace{M}_{\text{known mask}}, \tag{2.19}$$

where P_1 is the first plaintext byte. The measured signal to noise ratio (SNR) equals 5.8577. The measurements have 3 000 features around the S-Box part of the algorithm execution and in total there are 100 000 traces available.

Unprotected AES-128 on FPGA (AES_HD)

The AES_HD consists of measurements of an unprotected AES-128 implementation on a FPGA.[5] The AES algorithm was written in VHDL in a round based architecture, which takes 11 clock cycles for each

[4]The DPAv4 dataset is available at http://www.dpacontest.org/v4/.
[5]The AES_HD dataset is available at https://github.com/AESHD/AES_HD_Dataset.

encryption. The core is wrapped around by a UART module to enable external communication. It is designed to allow accelerated measurements to avoid any DC shift due to environmental variation over prolonged measurements. The total area footprint of the design contains 1 850 LUT and 742 flip-flops. The design was implemented on Xilinx Virtex-5 FPGA of a SASEBO GII evaluation board.

Side-channel traces were measured using a high sensitivity near-field EM probe, placed over a decoupling capacitor on the power line. Measurements were sampled on the Teledyne LeCroy Waverunner 610zi oscilloscope. We attack the register writing in the last round [56]:

$$Y(k^*) = \underbrace{\mathtt{Sbox}^{-1}[CT_i \oplus k^*]}_{\text{previous register value}} \oplus \underbrace{CT_j}_{\text{ciphertext byte}}, \tag{2.20}$$

where CT_i and CT_j are two ciphertext bytes, and the relation between i and j is given through the inverse ShiftRows operation of AES. We use $i = 12$ resulting in $j = 8$ as it is one of the easiest bytes to attack. The model-based SNR has a maximum value of 0.0096. Each trace has 1 250 features, and in total, there are 1 000 000 traces.

Random Delay Countermeasure (AES_RD)

This dataset targets a smartcard with an 8-bit Atmel AVR microcontroller, with an AES-128 implementation using a random delay countermeasure as described by Coron and Kizhvatov [58].[6] Adding random delays to the normal operation of a cryptographic algorithm causes misalignment of important features, which in turns makes the attack more difficult to conduct. As a result, the overall SNR is reduced: it has a maximum value of 0.0556. We attack the first AES key byte targeting the first S-Box operation:

$$Y(k^*) = \mathtt{Sbox}[PT_1 \oplus k^*]. \tag{2.21}$$

Each trace has 3 500 features and in total there are 50 000 traces. Recently, this countermeasure was shown to be prone to deep learning based side-channel [18]. It does not modify the leakage order (like masking). However, since this countermeasure is often applied in commercial products, we use it as a target case study.

ASCAD

Finally, we test our architecture on the recently available ASCAD database.[7] The target platform is an 8-bit AVR microcontroller (ATmega8515) running a masked AES-128 implementation. Measurements are made using electromagnetic emanation [52]. A first order masking scheme is used as countermeasure, protecting the S-Box computation by a 8-bit random mask M.

The dataset follows the MNIST database and provides 60 000 traces, where originally 50 000 traces were used for profiling/training and 10 000 for testing. We use the raw traces and use the pre-selected window of 700 relevant samples per trace corresponding to masked S-Box for the third plaintext byte:

$$Y(k^*) = \mathtt{Sbox}[PT_3 \oplus k^*]. \tag{2.22}$$

The SNR for the ASCAD dataset is ≈ 0.8 under the assumption we know the mask, while it is almost zero with the unknown mask.

2.4.4. Portability Dataset

In a realistic SCA scenario, the profiling phase is conducted with a device under the control of the attacker: when desired, multiple keys can be used to learn a model. The attack phase would involve a different device with a different (fixed) key. This practical challenge is commonly known as *portability*, by Bhasin *et al.* [7] defined as "*[..] all settings where an attack has no access to traces from the device under attack to conduct a training but only to traces from a similar or clone device, with uncontrolled variations*". Up to recently, this issue has drawn limited attention from the SCA community. In 2012, Elaabid and Guilley [59] described some portability issues with PCA-based TAs on an AIC implementing

[6]The AES_RD dataset is available at `https://github.com/ikizhvatov/randomdelays-traces`.
[7]The ASCAD dataset is available at `https://github.com/ANSSI-FR/ASCAD`.

DES. Their attacks were less effective both in a setting with a chip running on reduced voltage, as well as a scenario where the chip had been aging for four years. As a solution, they proposed realignment in time and normalization in amplitude, with respect to the original template. This combination solved the portability issue in their scenarios.

Several works on portability of AES have been made public in mid-2019. They all involve a deep learning attack, using power measurements of unprotected AES-128, in the value model (256 classes). Das *et al.* [8] used Atmega 8-bit microcontrollers, with four devices for training and four for testing. Their attack involved an MLP with two deep layers of 200 neurons, plus batch normalization and dropout. They demonstrated that adding traces from more different devices in the profiling phase has a positive effect on the attack performance. The authors extended their work by measuring on 30 devices, and also used a CNN to attack in addition to the MLP [9]. Pre-processing the MLP attack with dynamic time warping to align the traces, followed by PCA-based feature extraction, massively improved their test accuracy on different devices. In a similar setting, Bhasin *et al.* [7] attacked an unprotected AES-128 implementation on a AVR Atmega328p microcontroller. They took a more methodological approach on the portability issue, proposing the 'Multiple Device Model' (MDM) which also involves combining different devices to be used in the profiling phase. They explored the effects of portability for several classifiers, including an MLP and CNN. They explored the following scenarios:

- Same device and key. This was used a baseline, as publications often evaluate their attacks in this scenario.
- Same device, different key: in this scenario, the training data is based on different keys. Their results show a large negative effect on classical ML techniques (i.e., NB, RF) attacks, but the deep learning techniques are able to cope with this change quite well.
- Same key, different device: again, this change has a large negative impact on classical ML approaches, but the MLP and CNN only had a mild penalty.
- Different devices with different keys: in the most realistic setting, the profiling phase is conducted using a device having a certain key, targeting another device with a different key in the attack. Here, all techniques suffer from a large penalty. Although the MLP and CNN still outperform the other classifiers, they need much more attack traces to find the correct key byte compared to the first scenario.

Interestingly, their results also indicate that adding more data is not beneficial in a portability setting: overfitting on traces from one device will lead to a penalty in performance when evaluating for another device. Only the (larger) CNN profited from a large training set [7].

Despite the slightly different approaches, these papers point in the same direction: profiling measurements from multiple devices improves the attack on another device. This raises the following question: to what extend to neural networks learn the same representation, when they are trained on measurements from different devices?

Data Structure

Bhasin *et al.* [7] were kind enough to share their datasets to be used in this work. This allowed us to compare neural network representations in a portability setting, looking at networks trained on measurements of the same AES implementation, but different devices and/or keys. The power measurements are taken on an implementation of AES-128 on an AVR ATmega 328p 8-bit microcontroller. Each trace consists of 600 features. Each set contains 50 000 measurements, of which (at most) 40 000 samples are used for training and 10 000 samples for testing.

We use three sets from this project, number 1, 2, and 3. Sets 1 and 2 have been taken on two different devices, where the same key is used. Sets 2 and 3 have been taken on the same device, but using different keys for encryption. In Chapter 5, we use the portability datasets to see to which extend NNs trained on them learn the same representation.

3 Related Work

In this chapter, we discuss scientific publications in the relevant domains of profiled SCA and deep learning. First, we discuss notable attacks on AES side-channel datasets, on which the model architectures in this work are based. We continue with research analyzing classifier's behavior in SCA, and recent developments in research involving the portability issue. Next, two 'hot topics' in Deep Learning are discussed: the comparison between deep neural networks (DNNs), and the mimicking of large networks by smaller architectures. Finally, findings from these state-of-the-art literature are used to place this work in context and we reformulate the research questions.

3.1. Machine Learning in Side-Channel Analysis

In the last decade or, machine learning techniques transitioned from exotic side-channel analysis techniques to a standard for conducting profiling side-channel attacks. Various techniques have been successful in SCA, such as Random Forest [60, 61], Support Vector Machines [62, 63], multilayer perceptrons (MLPs) [53, 64], and convolutional neural networks (CNNs) [18, 35, 51]. Note that the last two techniques belong to the deep learning domain and they gained a significant attention in the last few years. Those works usually concentrate on the empirical evaluation of machine learning techniques; some also compare with template attacks.

To the best of our knowledge, the most effective use of deep learning in profiled SCA make use of noise addition to VGG-like networks [35]. More specifically, it describes four model architectures and training settings for the public datasets described in Section 2.4.3. Like in VGG, they employ a large number of convolutional layers with a kernel size of three, and an increasing number of filters. The noise addition happens after batch normalization of the input layer, and is drawn from a normal distribution with mean zero and standard deviation α[1]:

$$X^* = BN_0(X) + \Psi, \qquad \Psi \sim N(0, \alpha) \tag{3.1}$$

Adding noise significantly reduces overfitting and boosts the performance of the attacks. As we attempt to answers questions about state-of-the-art profiled SCA, the network designs in this work are heavily inspired by [35].

3.1.1. Analysis of Classification Characteristics

Although machine learning techniques have been widely successful in SCA, efforts directed towards explaining the machine learning performance or providing a more general understanding of it are more sparse [65].

Up to now, in the SCA community, the bias-variance decomposition attracted only moderate attention. Lerman, Bontempi, and Markowitch introduced the bias-variance decomposition in SCA as

[1]In an earlier version of the paper, they drew from the standard normal distribution and α was a coefficient of this noise term: $X* = BN_0(X) + \alpha\Psi, \qquad \Psi \sim N(0,1)$. Although not equivalent, this led to roughly the same results.

they explored it as a metric in profiling attacks [66]. They considered template attack and stochastic attack in several scenarios like the number of profiling traces and the number of informative points in traces, evaluating using the loss expressed in bias and variance. Also, the bias-variance decomposition has been used as a tool for leakage profiling in order to improve the performance of the evaluation process [67]. They concluded that their technique can be useful in practice when the noise is not too high. While these examples gave promising results, we believe that the bias-variance decomposition can help much more in SCA. There has been no research into the influence of model complexity, nor has bias-variance been connected with common SCA metrics like guessing entropy.

We extended existing works by investigating the influence of different factors (e.g., model complexity of neural networks). Also, this work proposes the guessing entropy-decomposition and shows the relation between mean squared error and guessing entropy.

A first work on attribution for deep neural networks used in side-channel analysis was presented by Hettwer, Gehrer, and Güneysu [68]. There, several methods are proposed for image visualization to reveal Point of Interests (POIs). These are based on methodologies that have been established in the deep learning community, in particular computer vision. To determine and visualize which features where most influential in a (C)NN, they applied Saliency Maps [69], LRP [70], and occlusion [71]. Although these methods can be used as distinguishers, they are not suited to compare among networks or understand their inner representations.

3.2. Deep Learning Representations

In recent years, efforts have been made to better explain predictions of black-box techniques, typically focusing on insight in prediction decisions. This includes rule extraction, visual representations, feature importance, sensitivity analysis and activation maximization [11].

In the side-channel domain, we have the distinctive challenges around countermeasures and portability. Therefore, it is important to consider different kinds of data; e.g., protected vs. unprotected implementations, device A vs. device B. So, instead of just focusing to explain the classifiers' predictions, it is also useful to compare the representations learned by different models. To the best of our knowledge, two peer-reviewed methodologies have been devised to compare the inner representations of neural networks, which we will introduce in the next subsection.

3.2.1. Comparison of Neural Networks

Raghu *et al.* [72] proposed *Singular Vector Canonical Correlation Analysis* (SVCCA) to compare two layers in a network. It uses the *activation vectors* of the neurons: the outputs after the activation function is applied. A single layer j is represented by a $N \times c_j$ matrix, with c_j the number of neurons in the layer. For each neuron, the output is stored for every single sample in the dataset.

The matrices representing layers are the inputs of SVCCA, which consists of two steps. First, both matrices are reduced using Singular Vector Decomposition (SVD), to clear out noise from neurons that produce (near) constant outputs. Then, the two resulting matrices are compared using Canonical Correlation Analysis (CCA). CCA finds the linear transformations for both layers, so that their values are maximally correlated. This results in an ordered set of SVCCA components. The pairwise correlation between the components, as well as their values for individual samples, can be used to analyze the layers' representations.

In their paper, the authors only consider the neurons' outputs based on known data – so, output for artificial data is not considered [72]. In their subsequent research, Raghu and her colleagues further developed and tested their methodology [73]. The SVCCA methodology is discussed in detail in Chapter 5.

An alternative approach for layer-wise comparison was proposed in [74]. Similar to SVCCA, it is based on neuron outputs. These activations are computed for real data and many other similar examples where noise is added using PCA. However, the curvature-based approach has some downsides in comparison to SVCCA: for each data sample, 10 648 artificial samples are generated and used – providing both a conceptual disadvantage of relying on augmented data, and makes both memory and run time very expensive. Also, as it relies on complex mathematical concepts (Riemannian manifolds and curvatures), interpreting the outcomes is difficult.

3.2.2. Mimicking

The complexity of a neural network is often measured in the number of trainable weights. When increasing the depth of the model, by adding more layers, a more complicated input-output mapping $\hat{f}(\mathbf{x}) = \hat{y}$ can be learned. Indeed, in some cases larger (convolutional) networks are able to fit patterns when small networks are unable to do so [20]. Although the advantage of deep models have been demonstrated empirically for various problems, there is little theoretical work supporting this advantage. On the contrary, the *universal approximation theorem* states that an MLP with a single hidden layer can approximate any continuous function, given enough neurons [75]. ReLu-networks consisting of a single hidden layer with $n+4$ neurons have been proven to be universal approximators [76]. Despite the enormous potential of narrow networks, it appears very difficult to train them. Our learning algorithms seem to favor deeper networks to learn complicated functions.

Mimicking is a technique where a classifier learns from the outputs of another classifier. So, the *learner network* uses a modified version of the training set: instead of using the original labels, the predictions of the *teacher network* are used to learn the function. This means that the teacher's mistakes are passed on to the learner. Ba and Caruana [77] demonstrated the effectivity of mimicking a large network for speech (TIMIT) and image (CIFAR-10) datasets. Using narrow networks, training on the teacher network's outputs led to a much better performance than on the original labels. However, Urban *et al.* [78] showed that the properties of convolutional layers are harder to mimic.

To the best of our knowledge, mimicking has not been applied in SCA. When having similar performance, a small learner network is preferable to a (much) larger network. In this setting, the profiling phase would consist of two parts: the training of a large teacher network, and the mimicking of it by a smaller network. Then, the attack phase would employ the learner network, and would either a) use simpler and cheaper hardware: the use of specialized equipment such as graphics cards could be avoided; or b) be much faster when using the same hardware. This had led us to explore mimicking in a scenario where a large network normally performs better than a small one.

3.3. Research Questions

In this chapter, we have discussed recent research in both deep learning and side-channel analysis. Based on these developments, and the uncertainties that remain, we reformulate our research questions here:

1. Is there a relation between common loss functions used in machine and deep learning and the guessing entropy metric from SCA?
2. Can we change the bias-variance decomposition to reflect SCA metrics?
3. When using machine and deep learning techniques, which parameters can be modified effectively to reduce bias and/or variance?
4. In which settings can we use SVCCA be used to compare NNs and what can we learn from its application in SCA?
5. Do neural networks trained for similar data, using different devices and/or keys, learn the same representation?
6. How can we effectively ”port” a neural network to handle different devices?
7. Can SCA functions learned by large neural networks be mimicked by smaller networks?

4 Bias–Variance

In this chapter, classification techniques for profiled SCA are analyzed. First, we investigate the connection between the mean squared error (MSE) loss function and guessing entropy (GE). We apply Domingos' bias–variance decomposition as introduced in Section 2.1.4, to investigate the loss of classifiers when changing the model complexity, number of features, or training set size. Next, we propose the Guessing Entropy Decomposition, which is tailored for SCA, as it incorporates multiple traces in evaluating attacks. For the same experiments, the GE Decomposition is shown and the results are analyzed.

The *'Bias-variance Decomposition in Machine Learning-based Side-channel Analysis'* paper is based on the results of this chapter.[1]

4.1. Experimental Setup

The experiments were conducted on four datasets: DPAv4, AES_RD, AES_HD and ASCAD, as discussed in Section 2.4.3. Six techniques are used to conduct the attacks: template attack (TA), pooled TA, Gaussian Naive Bayes (NB), random forest, multilayer perceptron (MLP), and convolutional neural network (CNN). These classification techniques are described in detail in sections 2.1.1 and 2.2. This part discusses data pre-processing, the use of bootstrapping, and hyper-parameter tuning for the architectures of the MLP and CNN.

4.1.1. Data Pre-Processing

The datasets described in section 2.4.3 have various numbers of features and data formats. In order to compare the classifiers, we have pre-processed the data to a unified format. First, all features were normalized as described in Section 2.1.2. This means the feature-wise mean is 0 and standard deviation is 1.

Classifiers suffer from the curse of dimensionality [23]. In particular, for the random forest and MLP classifiers, a smaller number of features is beneficial in order to improve the classification score. For each dataset, we selected 200 features for the HW model and 200 features for the intermediate value model, based on the Pearson correlation.

Pearson correlation coefficient measures linear dependence between two variables, x and y, in the range $[-1,1]$, where 1 is the total positive linear correlation, 0 is no linear correlation, and -1 is the total negative linear correlation. Pearson correlation for a sample of the entire population is defined as

[1]Daan van der Valk and Stjepan Picek (2019). Bias-variance Decomposition in Machine Learning-based Side-channel Analysis. *Cryptology ePrint Archive*. https://eprint.iacr.org/2019/570.

input	
Fully connected: 200 neurons	*repeat 0–5 times*
output	

Table 4.1: Architecture of MLP network. The fully connected layers have ReLu activations, while the output layer is normalized using Softmax

such [79]:

$$Pearson(x,y) = \frac{\sum_{i=1}^{N}((x_i - \bar{x})(y_i - \bar{y}))}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{N}(y_i - \bar{y})^2}}. \tag{4.1}$$

We have calculated Pearson correlation for the target class variables HW and intermediate value, which consists of categorical values that are interpreted as numerical values. The features are ranked in descending order of the coefficients' absolute values. Therefore, the most strongly correlated features (either positive or negatively correlated) are selected.

Note, in order to use the convolutional shift of CNNs, we do not require feature selection. Consequently, all features are used as input for the CNNs.

4.1.2. Bootstrap

Bootstrapping is used to randomize the training data that was given to the learning algorithm, as defined in Algorithm 1. To get an accurate depiction of the estimated parameters (i.e., loss components), a high number of bootstraps is required. For each setting with random forest classifiers, we took $l = 100$ bootstraps. For neural networks, this number of experiments is unfeasible: for each time a parameter is changed, l networks need to be trained. Therefore, we ran experiments with MLPs and CNNs using $l = 10$ bootstraps.

4.1.3. Deep Learning Architectures

As described in sections 2.2 and 2.2.4, multilayer perceptron (MLP) and convolutional neural network (CNN) are feed-forward neural network paradigms. Despite their popularity in recent years, there is no clear-cut methodology to select a specific architecture for a certain classification problem. Therefore, we employ two network structures (one MLP and one CNN) that were shown to be effective in literature [35, 52].

We designed an MLP network based on the work of Prouff *et al.* [52], in which the hidden layers consist of 200 neurons each, and use the rectified linear unit (ReLU) activation. The output layer consists of either 9 or 256 neurons, representing the classes in the HW and intermediate value, respectively, and is normalized using the Softmax activation. We investigated the influence of complexity by varying the number of hidden layers from 0 to 5. For the experiments varying the training data size in terms of features and traces, we used a fixed size of two hidden layers.

For the CNN, we base our architecture on a VGG-like design [35]. This style is based on multiple convolutional blocks, with convolutional layers with an increasing number of filters (8, 16, 32, 64 (twice), 128 (twice)). The filter size is 3, with strides 1. After the convolutional layer(s) in a convolutional block, we employ Batch Normalization. To allow for a large number of experiments, we modified this design by adding max pooling layers after each convolutional layer. This decreases the intermediate dimensions, and thus the total number of parameters that must be optimized in the network. After the last convolutional block, the output is flattened and fed into a fully connected layer of 512 neurons, before reaching the output layer. To avoid overfitting, we employ dropout before the fully connected layer and the output layer. ReLu activation is applied for all convolutional layers and fully connected layer, while the output layer is again normalized using Softmax. Table 4.2 shows the complete architecture.

In our experiments on model complexity, we start without any convolution, then add one convolutional block at a time. For all other experiments, we use the model involving 2 convolutional blocks.

input	
BN	
Convolution of 8 filters	*Convolution block 1*
BN	
Max Pooling	
Convolution of 16 filters	*Convolution block 2*
BN	
Max Pooling	
Convolution of 32 filters	*Convolution block 3*
BN	
Max Pooling	
Convolution of 64 filters	*Convolution block 4*
Convolution of 64 filters	
BN	
Max Pooling	
Convolution of 128 filters	*Convolution block 5*
Convolution of 128 filters	
BN	
Max Pooling	
Flatten	
Dropout (50%)	
Fully connected: 512 neurons	
Dropout (50%)	
Output	

Table 4.2: Architecture of CNN network. Each convolutional layer has kernel size 3 and strides 1. Each max-pooling layer has pool size 2 and strides 2. The convolutional and fully connected layers have ReLu activation; the output layer uses Softmax.

When training the neural networks, we attempt to minimize the mean squared error as loss. We employ a batch size of 64, and train for 50 epochs using the Adam optimizer with a learning rate of 0.0001.

4.2. Connection between Mean Squared Error and Guessing Entropy

In this section, show how mean squared loss and guessing entropy are connected. While the $0-1$ loss function is addressed in related works, see e.g., [66, 67], to the best of our knowledge, the mean squared loss (MSE) was not investigated up to now in the context of SCA and bias–variance decomposition. We give additional information on the decomposition of error for MSE and then elaborate on why MSE is relevant in the SCA context due to its connection to guessing entropy. The squared loss function equals

$$L(y,\hat{y}) = (y-\hat{y})^2 \tag{4.2}$$

and it can be averaged over N examples to arrive to the mean squared loss (alternatively, we can see this as the expected value of the set of examples). By substituting the $E_{D,y}[L(y,\hat{y})]$ from Eq. 2.1.12 with the squared loss function, we derive the expression:

$$E_{D,y}[(y-\hat{y})^2] = c_1 N(x) + B(x) + c_2 V(x). \tag{4.3}$$

As already stated, coefficients c_1 and c_2 are one for the squared loss function. The optimal prediction y^* is $E_y[y]$ and the main prediction y_m is $E_D[\hat{y}]$. Finally, Eq. (4.3) becomes:

$$E_{D,y}[(y-\hat{y})^2] = E_y[(y-E_y[y])^2] + (E_y[y]-E_D[\hat{y}])^2 + E_D[(E_D[\hat{y}]-\hat{y})^2]. \tag{4.4}$$

Note that the bias term $(E_y[y]-E_D[\hat{y}])^2$ is squared so it is also often called bias squared. Still, we follow Domingos' notation and refer to it as bias [28].

Next, we derive the connection between the MSE and guessing entropy. The mean squared error loss function is commonly used for training feed-forward neural networks[2]. Computing the mean of Definition 2.1.3, we derive the MSE:

Definition 4.2.1 *The squared error of a prediction $\hat{y}$, compared to the true value y, equals: The mean squared error over a dataset with N samples, based on a prediction $\hat{\mathbf{p}}_i$ and true value $\mathbf{p}_i$ per sample $i = 1,2,\ldots,N$, equals:*

$$MSE = \frac{1}{N}\sum_{i=0}^{N-1} (\mathbf{p}_i - \hat{\mathbf{p}}_i)^2$$

For clarity, $\mathbf{p}_i$ denotes the hot-encoding of the correct class label y, and $\hat{\mathbf{p}}_i$ the predicted probabilities by some classifier, for a trace i. For the intermediate value model, there are 256 classes, so $\mathbf{p}_i$ and $\hat{\mathbf{p}}_i$ are 256-dimensional vectors with their entries non-negative and having a sum equal to 1. For example, if a trace i has the true class 0 and the classifier assigns equal probabilities to classes 0 and 1 (and zero probability for all other classes), we have:

$$\mathbf{p}_i = \begin{bmatrix} p_{i,0} \\ p_{i,1} \\ p_{i,2} \\ \vdots \\ p_{i,255} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \qquad \hat{\mathbf{p}}_i = \begin{bmatrix} \hat{p}_{i,0} \\ \hat{p}_{i,1} \\ \hat{p}_{i,2} \\ \vdots \\ \hat{p}_{i,255} \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.5 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.5}$$

Notice that the class label represents the leakage of i according to the leakage model. For trace i with plaintext PT_i, one particular class label $\hat{y}$ corresponds to a key guess: by inverting the leakage

[2]A more common loss function is categorical cross-entropy since it gives less emphasis to incorrect predictions and training can be more efficient due to better weight adjustments during backpropagation. Still, we consider it as a very interesting choice since 1) there is a connection between guessing entropy and MSE, 2) with MSE, there is an easy decomposition into bias and variance, and 3) the results with MSE will often not differ much from those obtained with the cross-entropy loss function.

model, the attack finds key guess $k = Y^{-1}(PT, \hat{y})$. Note Y^{-1} is bijective: when the plaintext is fixed, only one key guess refers to one leakage class. Thus, we can decompose the MSE into a term for each key guess:

$$MSE = \frac{1}{N}\sum_{i=0}^{N-1}(\mathbf{p}_i - \hat{\mathbf{p}}_i)^2 \tag{4.6}$$

$$= \frac{1}{N}\sum_{i=0}^{N-1}\begin{bmatrix} p_{i,0} - \hat{p}_{i,0} \\ \vdots \\ p_{i,255} - \hat{p}_{i,255}\end{bmatrix}^2 \tag{4.7}$$

$$= \frac{1}{N}\sum_{i=0}^{N-1}\sum_{j=0}^{255}(p_{i,j} - \hat{p}_{i,j})^2 \tag{4.8}$$

$$= \frac{1}{N}\sum_{j=0}^{255}\sum_{i=0}^{N-1}(p_{i,j} - \hat{p}_{i,j})^2 \tag{4.9}$$

$$= \frac{1}{N}\sum_{k=0}^{255}\left(\sum_{i=0}^{N-1}(p_{i,Y^{-1}(PT_i,j)} - \hat{p}_{i,Y^{-1}(PT_i,j)})^2\right) \tag{4.10}$$

$$= \sum_{k=0}^{255} MSE(k) \tag{4.11}$$

In the final derivation 4.11, we have $MSE(k) = \frac{1}{N}\sum_{i=0}^{N-1}(p_{i,Y^{-1}(PT_i,j)} - \hat{p}_{i,Y^{-1}(PT_i,j)})^2$ being the attribution to the MSE of one particular key guess. Notice that for the correct key k^* the corresponding label $p_{i,Y^{-1}(PT_i,j)} = 1$ for all i, and for all other keys $Y_{i,Y^{-1}(PT_i,j)} = 0$ for all i. Consequently, when a classifier is trained to minimize mean squared error, this means it also minimizes the average key rank of the correct key. In this sense, MSE optimizes for guessing entropy.

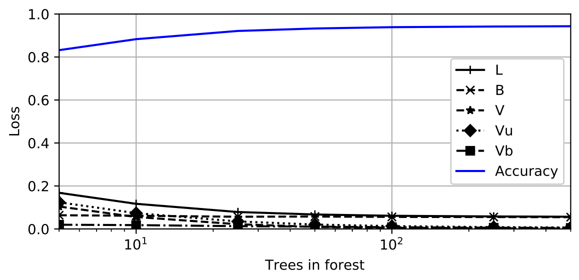
4.3. Bias–Variance Decomposition Results

Here, we present results for the bias–variance decomposition for model complexity, number of features, and number of measurements. In the plots for Domingos' bias–variance decomposition, the total 0–1 loss is indicated L, and decomposed into bias B and variance V. The variance itself is the sum of two parts: the unbiased variance V_u and the biased variance V_b. See Section 2.1.4 for a comprehensive explanation on Domingos' bias–variance decomposition. To help the readers, we also show the average accuracy as a more common metric to evaluate classifiers. As some attacks are either trivial (DPAv4) or fail completely, only characteristic cases are shown here. We concentrate on model complexity, as hyperparameter tuning is a hard problem, and these are not considered before in related works.

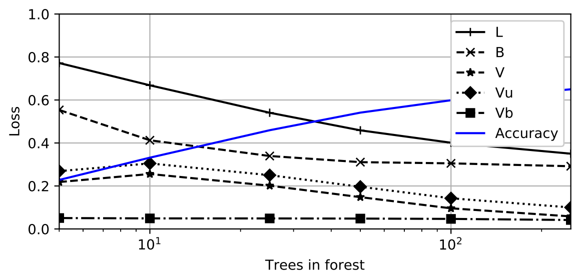
4.3.1. Model Complexity

When discussing the model complexity, we consider the number of trees for random forest, the number of hidden layers for MLP, and the number of convolutional blocks for CNN. While there are other hyper-parameters we could investigate, we consider these to be the core ones and consequently, the most relevant ones to investigate.

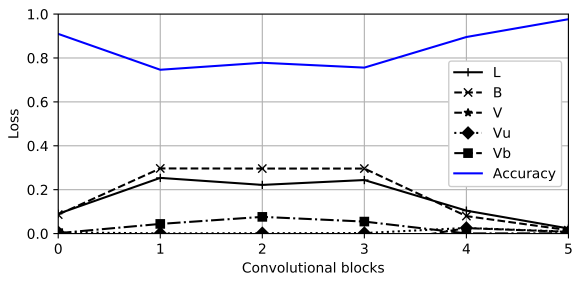
The first scenario we consider is DPAv4, as depicted in Figure 4.1. For random forest and the HW model, we see that even a small number of trees is enough for low loss (and high accuracy). Still, when increasing the number of trees, loss additionally reduces due to a reduction in variance (unbiased variance). For the value model, the decrease in bias is significant with the increase in the



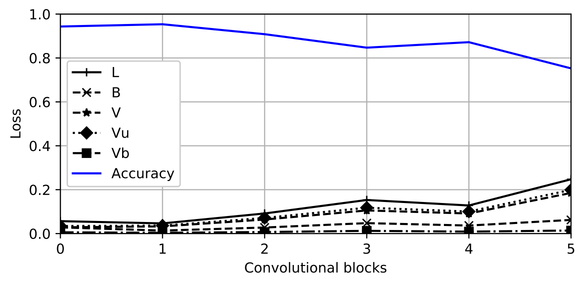
(a) Random forest, HW model, different number of trees.



(b) Random forest, value model, different number of trees.

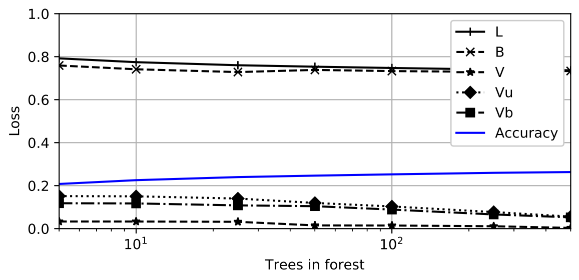


(c) CNN, HW model, different number of convolutional blocks.

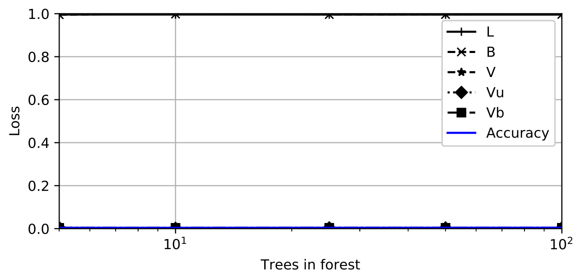


(d) CNN, value model, different number of convolutional blocks.

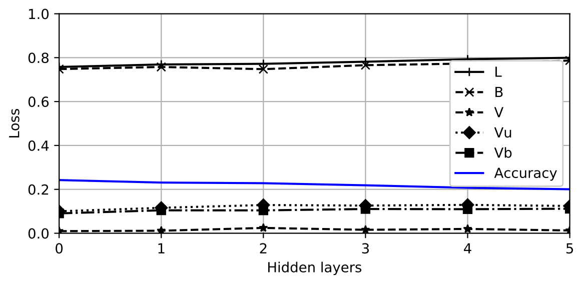
Figure 4.1: DPAv4 dataset results for model complexity.



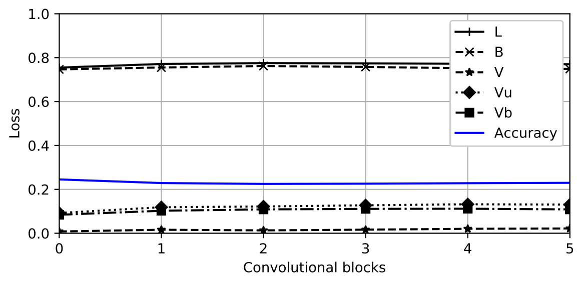
(a) Random forest, HW model, different number of trees.



(b) Random forest, value model, different number of trees.



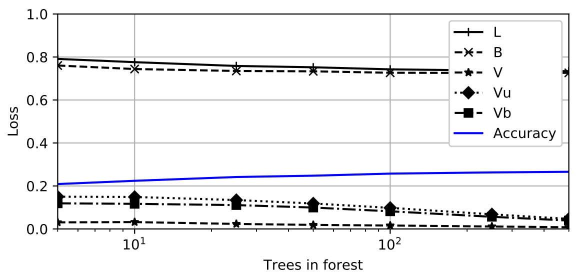
(c) MLP, HW model, different number of hidden layers.



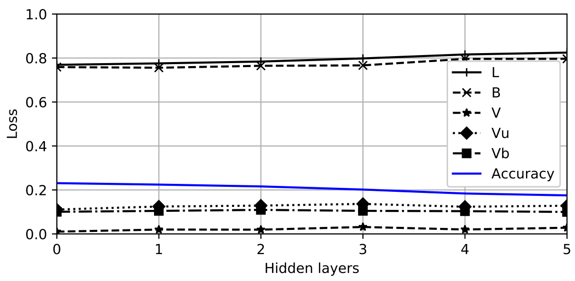
(d) CNN, HW model, different number of convolutional blocks.

Figure 4.2: AES_HD dataset results for model complexity.

number of trees, which indicates that more trees are needed to capture the complexity of data and to avoid underfitting. Besides bias, we can also notice a decrease in the unbiased variance. For CNN and HW model, we see first an increase in bias with increasing complexity. Note, even when there are no convolutional blocks, the CNN still includes a fully connected layer. Then, the bias becomes stable until we increase the number of convolutional blocks to more than 3 when bias starts to increase significantly. Notice the region between 1 and 3 convolutional layers where loss is lower than bias due to the increase in the biased variance. For the value model, loss increases with adding convolutional layers as we increase the unbiased variance, which is indicating that our model overfits and cannot

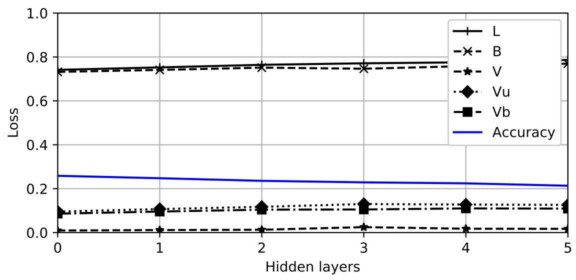


(a) Random forest, HW model, different number of trees.

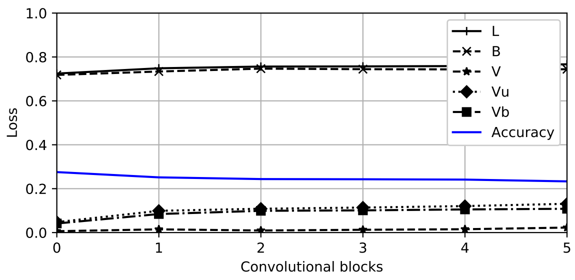


(b) MLP, HW model, different number of hidden layers.

Figure 4.3: AES_RD dataset results for model complexity.



(a) MLP, HW model, different number of hidden layers.



(b) CNN, HW model, different number of convolutional blocks.

Figure 4.4: ASCAD dataset results for model complexity.

adapt to new, previously unseen measurements.

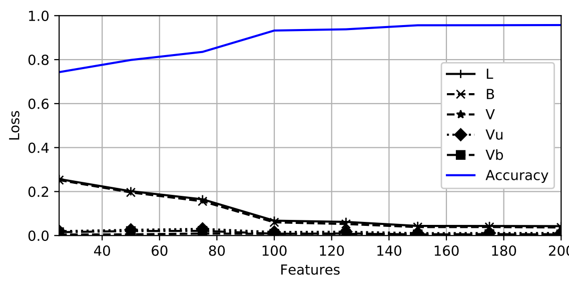
For the AES_HD dataset (Figure 4.2), we see that with added trees for random forest, loss slowly decreases due to the increase in bias. This indicates the dataset is difficult for this classifier and we require a much larger number of trees to prevent underfitting. The scenario for random forest and value model is interesting as here we cannot discern much from the bias–variance decomposition. Indeed, bias is very high and our model does not learn anything about data, so, naturally, variance must be low. This is also a common result we obtain for most of the scenarios with the value model. Both MLP and CNN in HW model exhibit similar behavior: almost constant loss and bias regardless of adding model complexity. Biased and unbiased variance are almost the same and they slowly increase. This means our models overfit. When considering accuracy, we also see it does not change. Upon closer inspection, we observe that these classifiers constructed models that classify all measurements into HW 4, the class with the largest population.

Next, in Figure 4.3, we depict results for the AES_RD dataset. Using more trees, random forest is able to improve performance by reducing bias. For MLP, increasing complexity has a counterproductive effect, increasing bias. This means that random forest benefits from added model complexity while for MLP, using more hidden layers means it overfits.

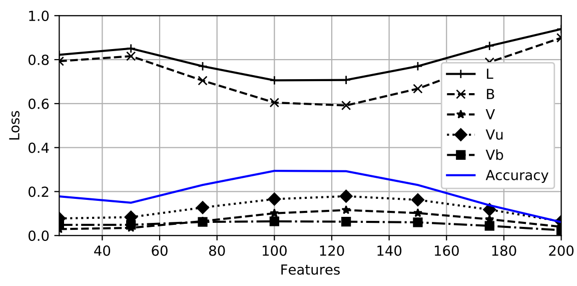
Finally, in Figure 4.4, we depict results for the ASCAD dataset. We consider the HW model and MLP and CNN architectures. Their performance is very similar (differing in the loss value only slightly) and with added model complexity, variance increases due to overfitting.

In general, we see that adding complexity often manages to reduce loss by reducing bias. Variance is usually low, except when we use complex architectures and investigate easy datasets (DPAv4). This indicates that our considered models mostly suffered from underfitting. With these conclusions, we are able to obtain insights into the behavior of various classifiers and adapt them to reach better performance. Thus, the bias–variance decomposition is useful for adjusting the architecture of machine learning algorithms in SCA. Still, there is a number of scenarios where the bias–variance decomposition does not help: if a dataset is very difficult for a certain classifier, we see a very high bias (and low variance) but increasing the model complexity does not improve performance. Consequently, for such cases, we can conclude our models underfit (which is relevant), but unfortunately, we obtain no insight on whether more complex models help at all.

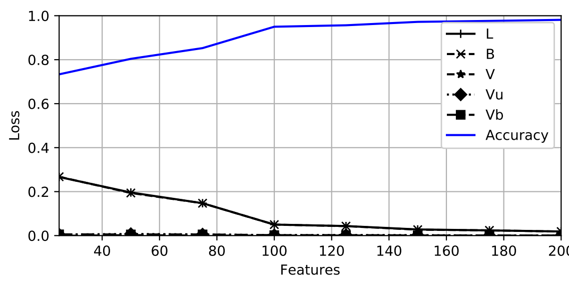
4.3.2. Number of Features



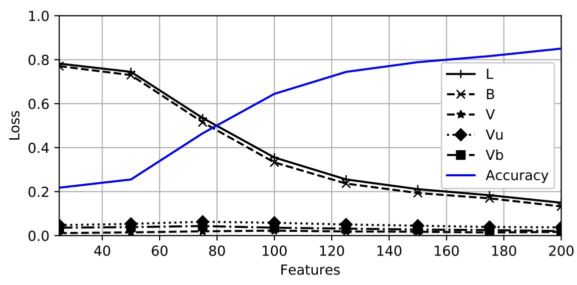
(a) Template Attack, HW model.



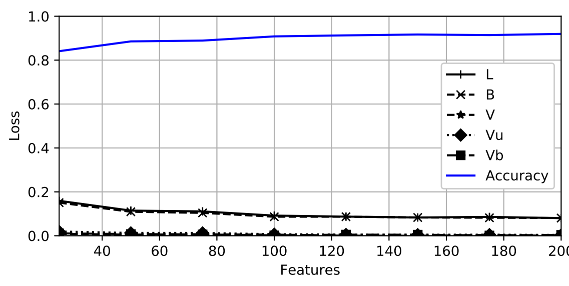
(b) Template Attack, value model.



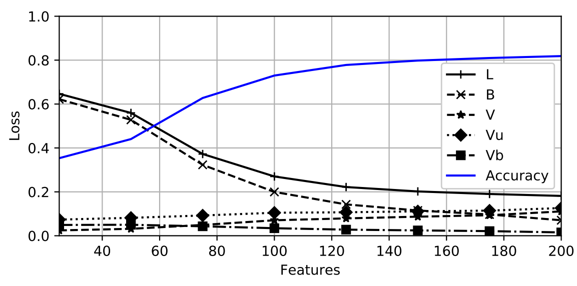
(c) Pooled Template Attack, HW model.



(d) Pooled Template Attack, value model.

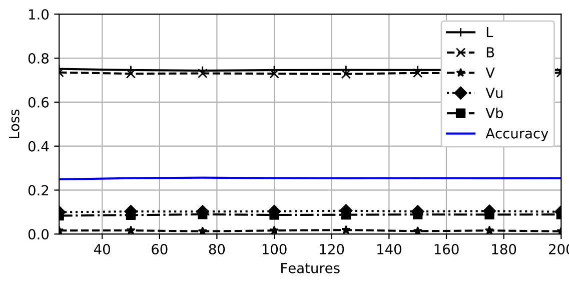


(e) MLP, HW model, 2 hidden layers

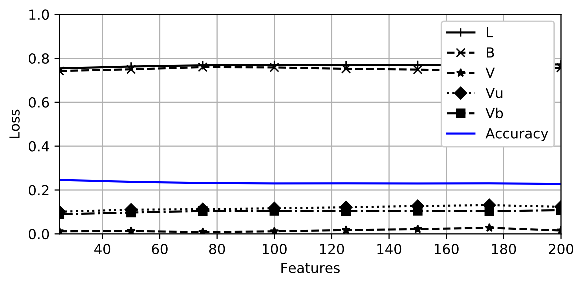


(f) MLP, value model, 2 hidden layers

Figure 4.5: DPAv4 dataset results for different numbers of features.



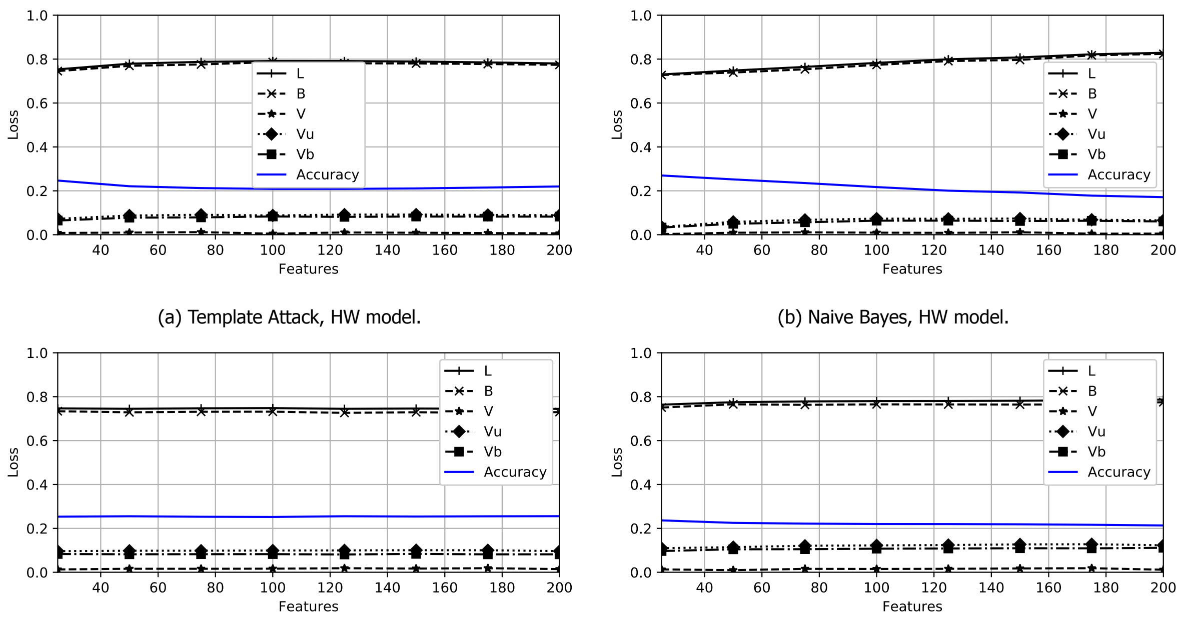
(a) random forest, HW model, 100 trees.



(b) MLP, HW model, 2 hidden layers.

Figure 4.6: AES_HD dataset results for different numbers of features.

In Figure 4.5, we give results for (pooled) TA and MLP attacking DPAv4, when considering a different number of features. Interestingly enough, we see a similar behavior for all techniques in the HW model: adding features brings only a small improvement in performance due to a decrease in bias. In particular, MLP is able to find solid patterns with only 25 features. The TA and pooled TA seem to perform near-optimal with 100 features, but accuracy steadily increases when going up to 200 features. Variance is almost at 0, which indicates this dataset does not overfit. In general, this is as good a performance as one could expect. For the value model, we see that adding features improves the performance significantly as bias reduces for the pooled TA and MLP. There is a point around 100 features when variance starts to increase, but that is negligible when compared to the bias decrease. As the value model is more complex than the HW model, more features help to capture the relations in data and reducing underfitting. Still, at 200 features bias and variance are quite close and one could expect that by adding more features we would soon arrive at a point where the model would start to overfit. We observe this with the Template Attack in Figure 4.5b. There, the optimal number of features is around 100-120. Using more/less features leads to over/underfitting, resulting in a high bias. As TA estimates a separate covariance matrix for each of the 256 classes, the pooled TA (Figure 4.5d) is clearly a better choice here.



(a) Template Attack, HW model.

(b) Naive Bayes, HW model.

(c) Random forest, HW model, 100 trees.

(d) MLP, HW model, 2 hidden layers.

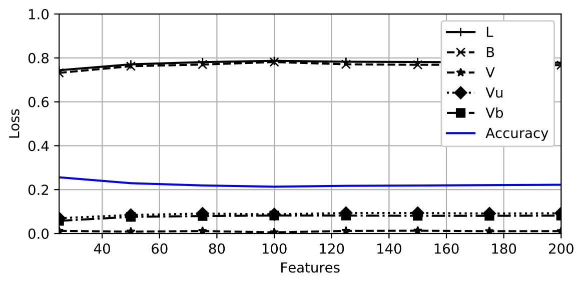
Figure 4.7: AES_RD dataset results for different numbers of features.

Figure 4.6 depicts results for AES_HD. Despite using two very different classifiers (that use different loss functions), the results are similar. Adding more features is not beneficial. Our model seems unable to learn a pattern from the data, so adding more features does not bring stronger models. Similar results are already seen in other scenarios and are common when considering datasets with countermeasures or a large amount of noise.

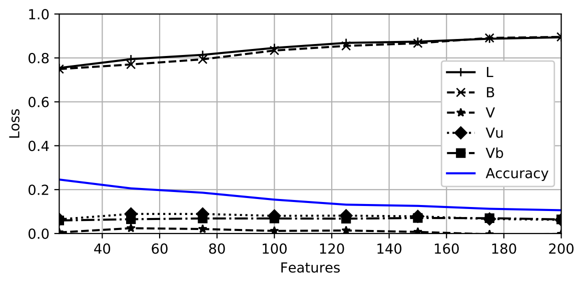
Next, AES_RD exhibits almost identical behavior as seen in Figure 4.7 for RF and MLP. Unfortunately, on the basis of these results, it is difficult to say whether added features would help for AES_HD and AES_RD. For AES_RD, we even observe a negative effect of adding features for the TA and Gaussian NB: the Naive Bayes' classifier even performs worse than random (majority class) guessing.

Finally, for the ASCAD dataset as depicted in Figure 4.8 we clearly see the curse of dimensionality in the Template Attack and Naive Bayes, for which both bias and variance increase when adding features. Using random forest, there is only a small decrease in bias when adding features. For MLP, no conclusion can be given besides the fact that there is a small increase in variance.

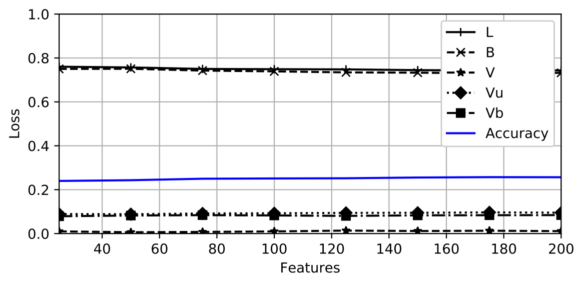
The benefit of added features is much less clear than the benefit of model complexity. This is not so surprising because it is to be expected that relevant information is included in the most important features. When the datasets are difficult to classify, then the performance is low as there is actually no learning happening so, naturally, one cannot expect benefit from introducing additional information in the form of added features.



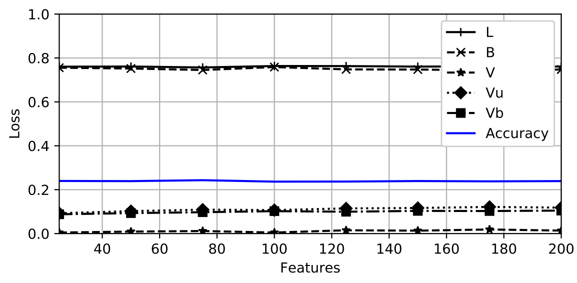
(a) Template Attack, HW model.



(b) Naive Bayes, HW model.



(c) Random forest, HW model, 100 trees.



(d) MLP, HW model, 2 hidden layers.

Figure 4.8: ASCAD dataset results for different numbers of features.

4.3.3. Training Set Size

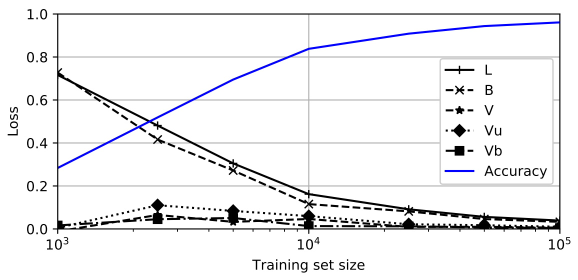
In Figure 4.9, we give results for DPAv4 when considering various training set sizes. Again, we see the advantages of pooling the covariance matrixes in the Template Attack, both in the HW and value model. For the classical TA, we see that adding more training traces eventually reduces loss because of a drop in bias, even if there is a small (initial) increase in variance. In the pooled TA, value model (Figure 4.9d), we see an interesting effect of adding more training samples: bias decreases to a stable level, reached around 10,000 training samples. However, variance decreases even further when adding more samples and slowly approaches 0. This means that a larger training set will lead to more similar models, even if their average prediction is not improved.

For CNN (Figures 4.9e) in the HW model, we see that as the training set size increases, loss decreases due to a decrease in bias. Then, around 10^4 training set size, we see that bias becomes larger than loss. This behavior occurs due to an insufficient model complexity to model the data when there is a large number of measurements. Variance, on the other hand, slowly decreases with the increase in the training set size. For the value model, we require much more data to build a strong model. This is clear due to the significant drop in bias when more than 10^4 measurements are used. With more measurements, variance also increases but it stays below bias. This indicates we require more complex model and more data for this scenario to improve even further.

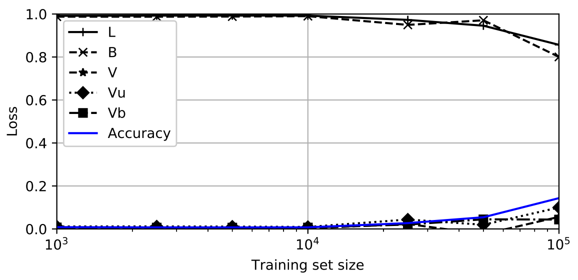
All the other datasets show similar behavior (AES_HD in Figure 4.10, AES_RD in Figure 4.11, and ASCAD in Figure 4.12). In these cases, adding measurements does not decrease loss and it appears there is no benefit from more measurements. Naturally, the situation is not so simple as here we also need to consider what the model complexities are and how these can benefit from extra measurements.

While it seems intuitive that the additional training measurements must help in the machine learning-based SCAs, we actually see that it is often not possible to see the benefit of it if we do not also increase the model complexity. Only for the easiest dataset (DPAv4), there is a clear advantage of added measurements. For all other scenarios, we see a constant behavior with high bias, which means that all those models underfit.

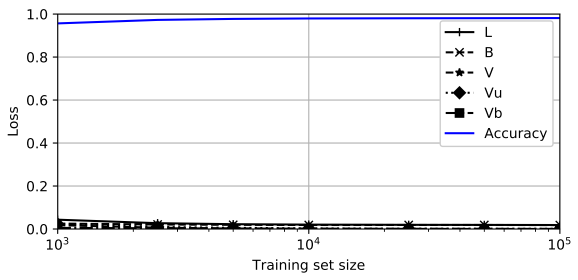
The bias–variance decomposition gives us insights on the behavior of various machine learning-based side-channel attacks. At the very minimum, we are able to discern whether the loss comes from bias or variance (neglecting the noise part). From there, we are able to estimate whether our models underfit or overfit, which then indicate how to improve the performance. In many cases, we



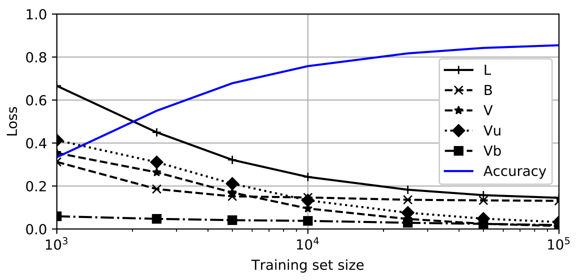
(a) Template Attack, HW model.



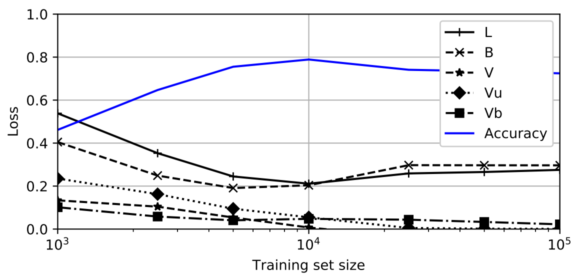
(b) Template Attack, value model.



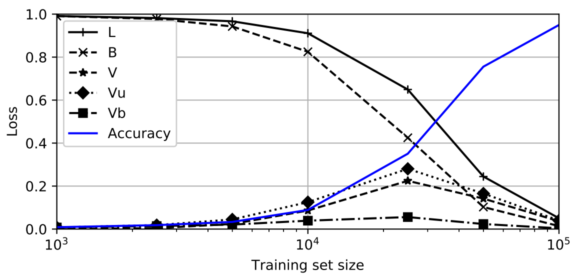
(c) Pooled Template Attack, HW model.



(d) Pooled Template Attack, value model.

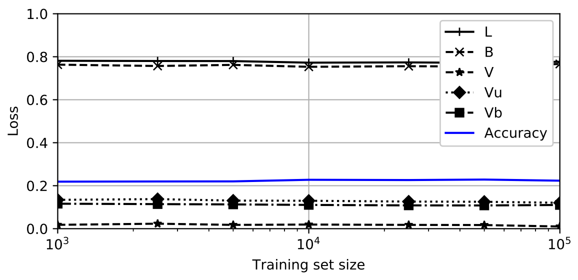


(e) CNN, HW model, 2 convolutional blocks.

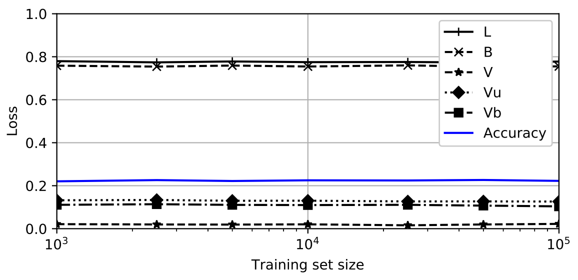


(f) CNN, value model, 2 convolutional blocks.

Figure 4.9: DPAv4 dataset results for different training set sizes.



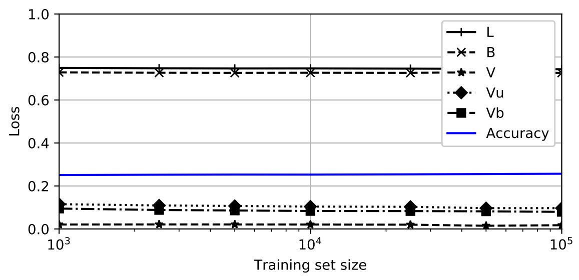
(a) MLP, HW model, 2 hidden layers.



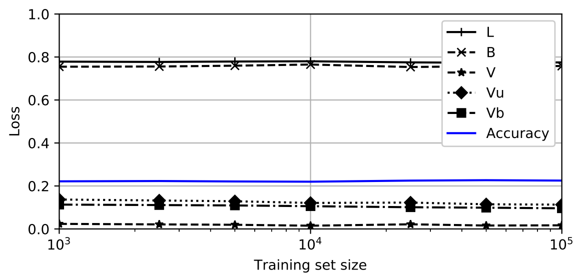
(b) CNN, HW model, 2 convolutional blocks.

Figure 4.10: AES_HD dataset results for different training set sizes.

can obtain even more information as we see the best trade-off between bias and variance. There are also many cases where the bias–variance decomposition is not helping; for instance, when the loss does stays constant throughout the experiment. In this case, bias is usually high while variance is low, which means our model did not learn. Then, we do not have any indication whether the change in parameters (e.g., number or features or measurements) help or not. Such scenarios usually occur either when the noise is high or there are countermeasures. Our results are also in accordance with findings in [67], despite the fact that the authors considered different datasets and machine learning algorithms.

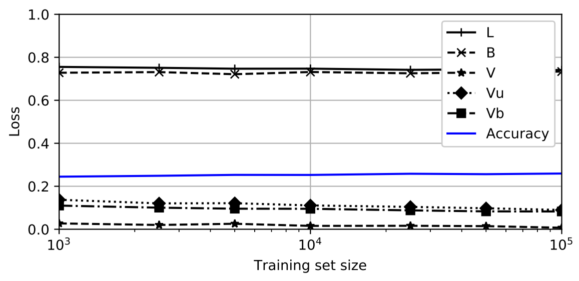


(a) Random forest, HW model, 100 trees.

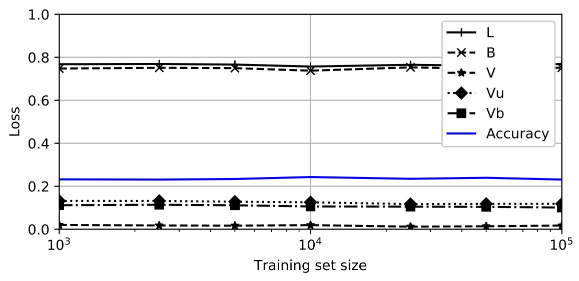


(b) CNN, HW model, 2 convolutional blocks.

Figure 4.11: AES_RD dataset results for different training set sizes.



(a) Random forest, HW model, 100 trees.



(b) MLP, HW model, 2 hidden layers.

Figure 4.12: ASCAD dataset results for different training set sizes.

Additionally, in the value model, loss and bias are usually very high and we do not see much change in those values. On the other hand, in the HW model, the results are more revealing although we cannot directly connect the bias performance with the performance of side-channel attack (as for instance measured with guessing entropy).

4.4. Guessing Entropy Bias–Variance Decomposition

As discussed in the previous section, while the bias–variance decomposition can give us valuable information about the performance of machine learning-based SCAs, it also has some drawbacks. Domingos' bias–variance decomposition has a pitfall in the side-channel domain: the metric is based on a decomposition of individual measurements in a test set. This means only changes in the training data can be used to research a classifier's characteristics. It is closely related to the classification accuracy, only taking the assigned class label for each attack trace into account. In SCA, a single trace is typically not enough to find the correct key. Therefore, the predicted class probabilities from the test set are combined. In some cases, this means the key can be determined even with a fairly low accuracy score. As Domingos' bias–variance decomposition is not useful in this case, we inspect the bias and variance of the classifiers' guessing entropy. This allows for analysis of a more practical setting: a scenario in which multiple traces are used to guess the key byte.

From the perspective of the attack, guessing entropy indicates the work to be done. In the ideal case, the guessing entropy is 0: the best-ranked key guess as attack output is the correct key byte. As there are 256 possible key bytes, the worst possible GE would be 255. Notice that if a classifier consistently predicts the right class label for each trace, both the GE and the (Domingos') loss would be 0. If the correct label is consistently ranked second, the Domingos' bias would be maximal (i.e., 1) and classification accuracy would be zero percent. However, when combining multiple outputted probabilities per class, the guessing entropy may drop after only a couple of traces.

In this section, we evaluate the guessing entropy for different classifiers. We consider the same scenarios as described in the previous section, but now they include the size of the attack set. For each variable we investigate (i.e., number of traces, number of features, complexity), we generate guessing entropy between 0 and 5 000 attack traces. Then, we define the *guessing entropy bias* as the expected guessing entropy based on profiling set T_p and attack set T_a:

$$B_{GE}(T_p,T_a) = E[GE(T_p,T_a)]. \tag{4.12}$$

Next, we define *guessing entropy variance* as the expected deviation from the GE mean:

$$V_{GE}(T_p,T_a) = E\Big[\big|GE(T_p,T_a) - E\left[GE(T_p,T_a)\right]\big|\Big]. \tag{4.13}$$

Note that we follow Domingos in using the standard deviation (instead of the standard deviation squared) to define variance. Now, we can define a simple decomposition of guessing entropy loss:

$$L = B(T_p,T_a) + V(T_p,T_a) - V(T_p,T_a) = B + V_b - V_u. \tag{4.14}$$

This decomposition is simple in the sense that the loss is equal to the bias, but allows us to analyze the guessing entropy, taking into account both the average score and the deviations from it. As we show in this section, this gives new insight on the performance of classifiers used in SCA. The 3D graphs in this section show the bias in colored surface plots, while the biased (V_b) and unbiased (V_u) variance are shown as grey surface plots added and subtracted from the bias, respectively.

We depict the results for the model complexity, different number of features and different number of training measurements. Note that for the model complexity, we depict all results, while for the other two scenarios, we select a few characteristic cases. We emphasize that differing from the bias–variance decomposition, guessing entropy bias–variance decomposition is able to offer insights for all considered scenarios.

4.4.1. Model Complexity

In Figure 4.13, we show the guessing entropy decomposition for different classifiers for the DPAv4 dataset. For the simplest dataset, we see no influence of model complexity to the guessing entropy: typically, only a handful (< 5) of traces suffice to find the correct key.

For the AES_RD dataset, we see in Figure 4.15 that again it is easier to attack using the HW model, compared to intermediate values. As convolutional layers are invariant to shifts in input data, the CNN classifier has a clear advantage for this dataset. Indeed, we see that the only successful attack utilizes a CNN. In this case, a higher complexity is clearly beneficial, as a small number of convolutional blocks is not sufficient to reach a guessing entropy of zero – but still outperforms the MLPs and random forests. For the other classifiers, we see no substantial influence of complexity on the guessing entropy bias and variance.

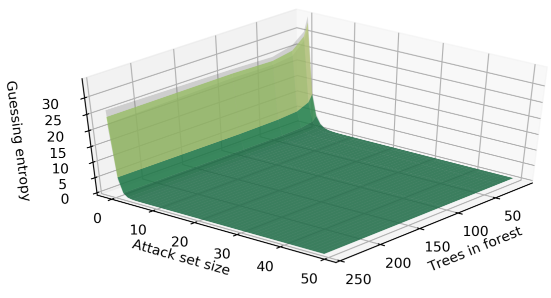
Finally, Figure 4.16 shows the results for the ASCAD dataset. Similar to the AES_HD dataset, we observe that the random forest classifier benefits from higher complexity, while MLP and CNN benefit from a low complexity. Those classifiers perform best when having no hidden layers or no convolutional blocks, respectively. When increasing complexity, these models immediately start to overfit and require more traces to find the secret key. For more than two hidden layers in MLP, or more than three convolutional blocks in CNN, these networks are unable to reach a GE of zero, even using 5 000 attack traces. In the intermediate value model, only MLP consistently attacks the data successfully. Both the random forest and CNN have a high bias and variance in this scenario.

4.4.2. Training Set Size

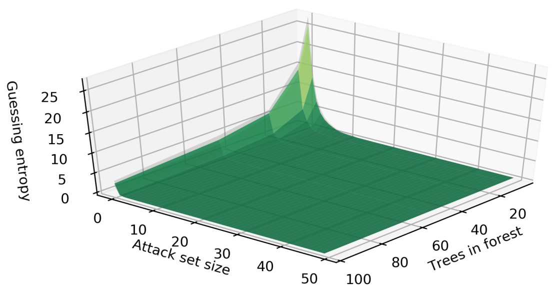
Figure 4.17 shows the relation between training set size for a CNN attack on the AES_RD dataset. It is clear that the CNN needs a large dataset to conduct a successful attack.

Figure 4.18 shows the influence of the number of training traces used to attack the ASCAD dataset. In this scenario, having more traces clearly benefits the attacks in the HW model. All classifiers are able to determine the correct key byte when given 25 000 or more traces for training.

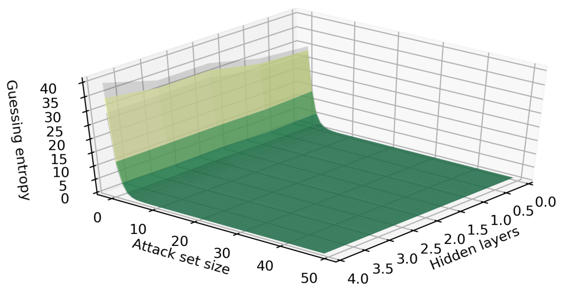
In the more difficult scenario of attacking the intermediate value we see a positive correlation between training set size and guessing entropy for the neural networks. The MLP classifier is able to find the key byte when using the maximum number of traces, while the CNN achieves a guessing



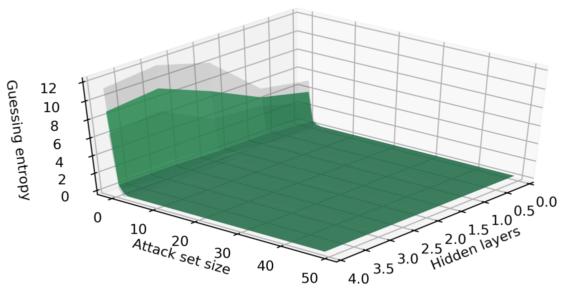
(a) Random forest, HW model, different number of trees.



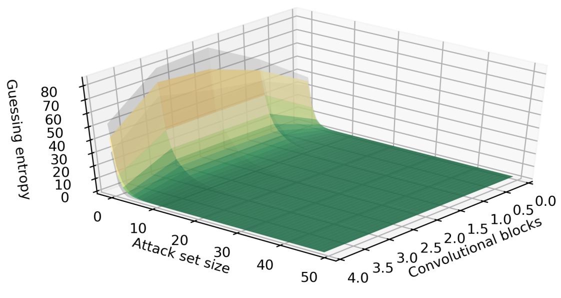
(b) Random forest, value model, different number of trees.



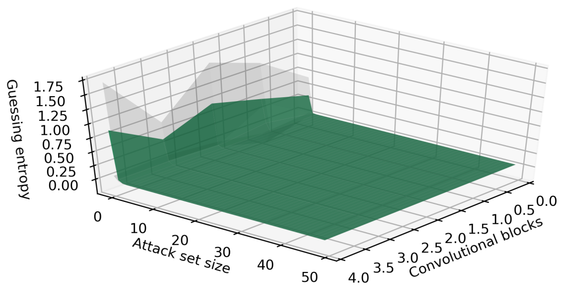
(c) MLP, HW model, different number of hidden layers.



(d) MLP, value model, different number of hidden layers.



(e) CNN, HW model, different number of convolutional blocks.



(f) CNN, value model, different number of convolutional blocks.

Figure 4.13: Guessing entropy decomposition, DPAv4 dataset results for model complexity.

entropy of 30. For the random forest classifier, adding more traces does not seem to help: even for the largest training set, it performs only slightly better than random guessing.

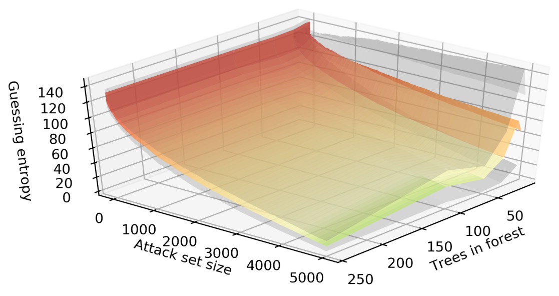
Note that we omit results for other scenarios. In these settings, there was no significant relation between the training set size and the classifier's guessing entropy: either the classifier was able to attack successfully or not.

4.4.3. Number of Features

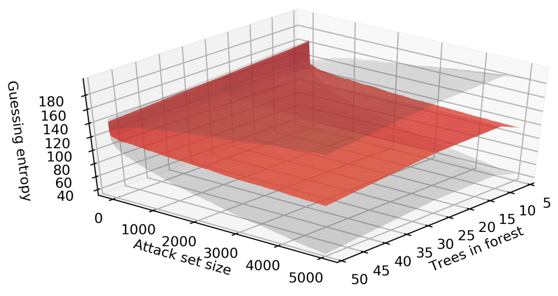
Figure 4.19 shows the influence of the number of features used to attack the AES_HD dataset in the HW model. We observe an interesting difference between TA and pooled TA: while they begin similar, the GE diverges when adding more features. Classical TA suffers from the curse of dimensionality, and is unable to derive good patterns when the input vector grows too large. On the contrary, more features are beneficial in pooled TA, as less attack traces are required to conduct a successful attack. For the random forest and MLP classifier, we observe a small influence on the guessing entropy: selecting less features seems to benefit the GE slightly.

The effect of the number of features to attack ASCAD in the HW model is shown in Figure 4.20. Clearly, adding more features is highly beneficial in this scenario: when using the maximum number of features (200), the MLP attack is successful. The random forest also profits from more features, reaching a GE of ~ 5 using 200 features and the full attack set.

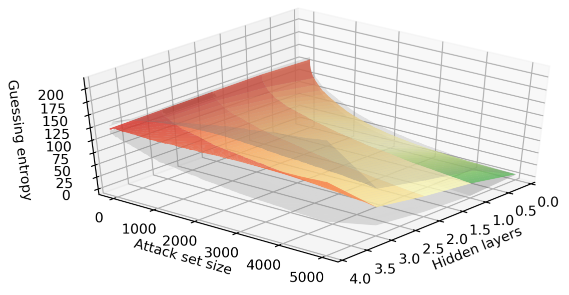
For the other datasets, we see no relation between number of features and the guessing entropy



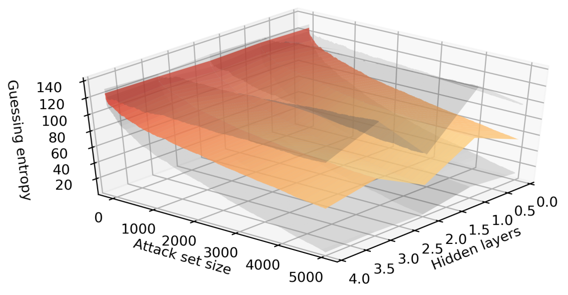
(a) Random forest, HW model, different number of trees.



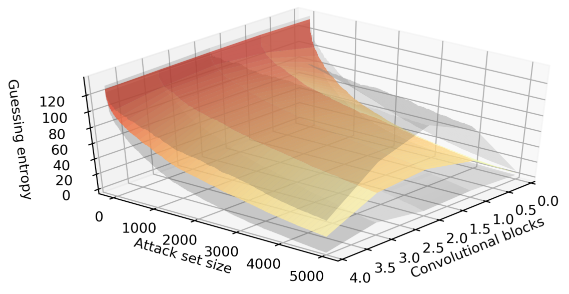
(b) Random forest, value model, different number of trees.



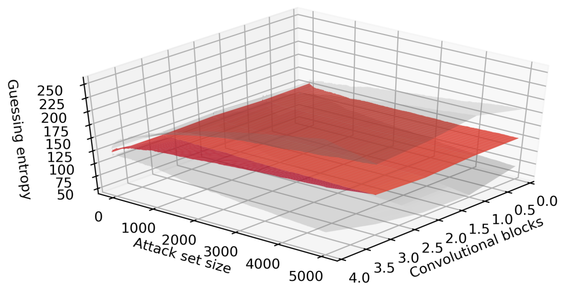
(c) MLP, HW model, different number of hidden layers.



(d) MLP, value model, different number of hidden layers.



(e) CNN, HW model, different number of convolutional blocks.

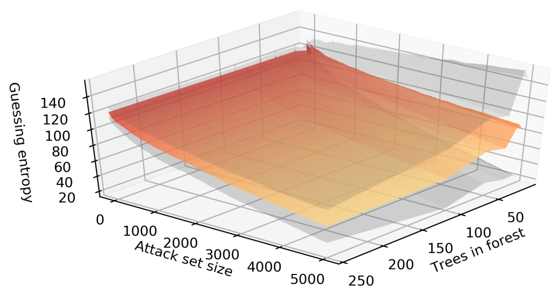


(f) CNN, value model, different number of convolutional blocks.

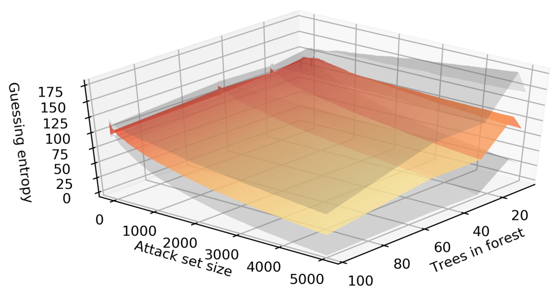
Figure 4.14: Guessing entropy decomposition, AES_HD dataset results for model complexity.

for the random forest and MLP classifiers. In DPAv4, the smallest number of features (25) is already enough to retrieve the key byte. Differing from that, both classifiers are unable to break AES_RD, independent of the number of features, because of the shift in correlated value due to the random delay countermeasure.

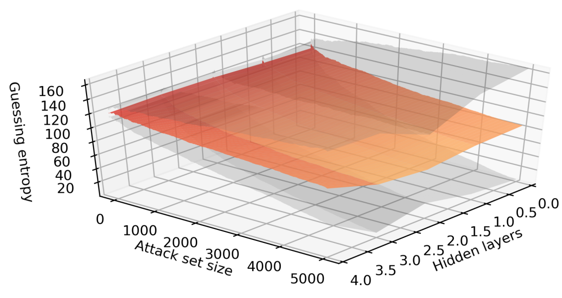
Comparing the Guessing Entropy and Domingos' Bias–Variance decompositions depends on the scenario. For simple problems (e.g., DPAv4 dataset), both decompositions show the bias and variance in a meaningful way. This is true for any case where the accuracy score is reasonably high. When accuracy is low, Domingos has a clear disadvantage: if accuracy is low throughout different scenarios, the graphs are flat and show no influence of the parameter (i.e., complexity, size of training set, number of features). In this case, guessing entropy decomposition is still able to distinguish these scenarios when there is a difference in outputted class probabilities, leading to a better key guess. For example, in Figure 4.12, we observe little change in Domingo's loss for the ASCAD dataset attacked with different training set sizes. Figure 4.18 shows there is a clear relation between the training set size and the performance of the attack. Consequently, the guessing entropy decomposition is arguably a better method for evaluating classifiers' characteristics for difficult scenarios in SCA.



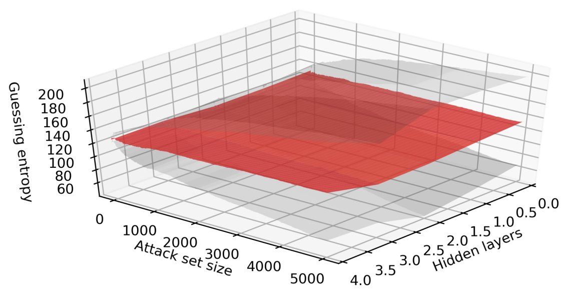
(a) Random forest, HW model, different number of trees.



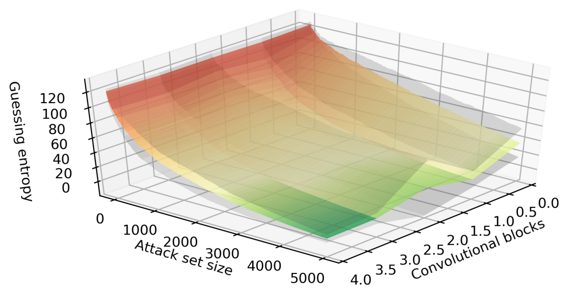
(b) Random forest, value model, different number of trees.



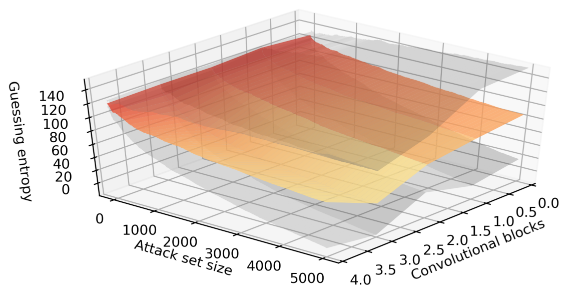
(c) MLP, HW model, different number of hidden layers.



(d) MLP, value model, different number of hidden layers.

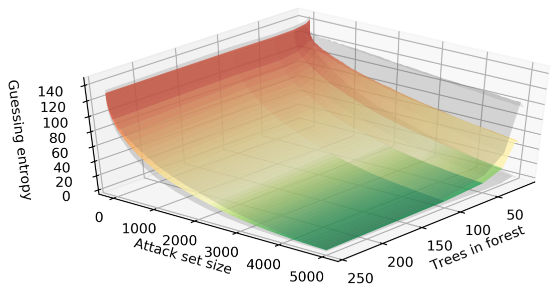


(e) CNN, HW model, different number of convolutional blocks.

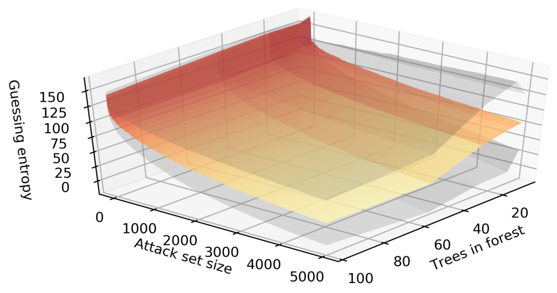


(f) CNN, value model, different number of convolutional blocks.

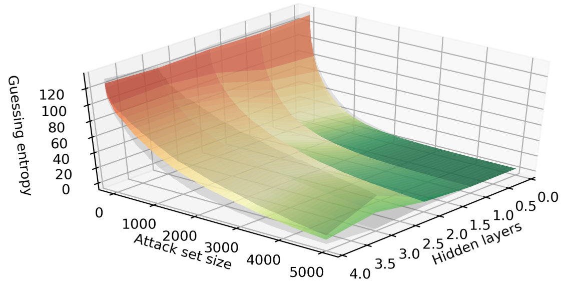
Figure 4.15: Guessing entropy decomposition, AES_RD dataset results for model complexity.



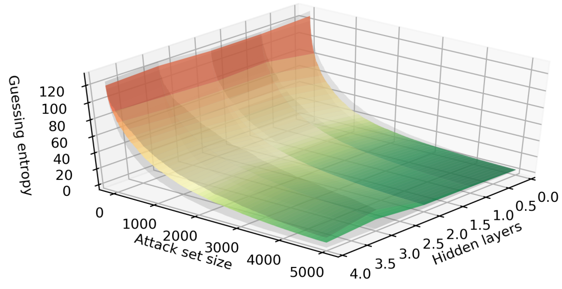
(a) Random forest, HW model, different number of trees.



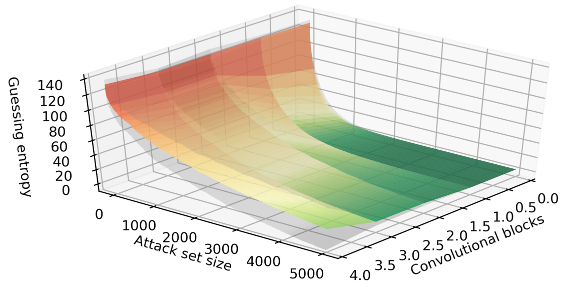
(b) Random forest, value model, different number of trees.



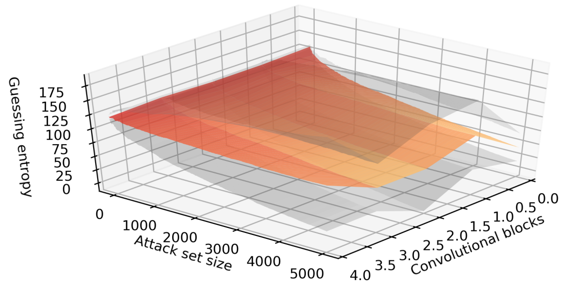
(c) MLP, HW model, different number of hidden layers.



(d) MLP, value model, different number of hidden layers.

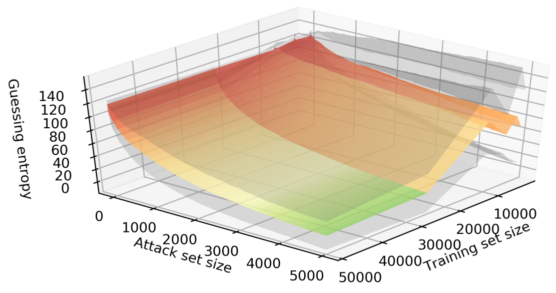


(e) CNN, HW model, different number of convolutional blocks.

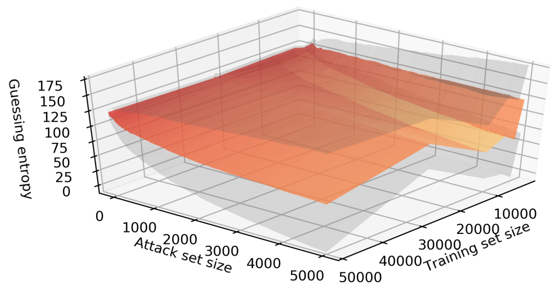


(f) CNN, value model, different number of convolutional blocks.

Figure 4.16: Guessing entropy decomposition, ASCAD dataset results for model complexity.

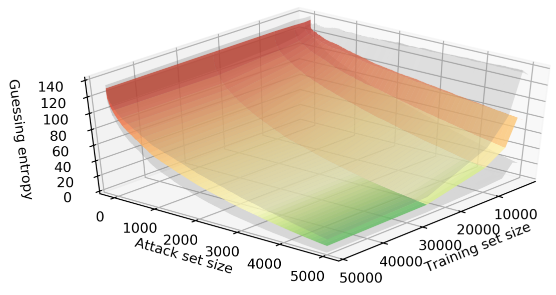


(a) CNN, HW model, 2 convolutional block.

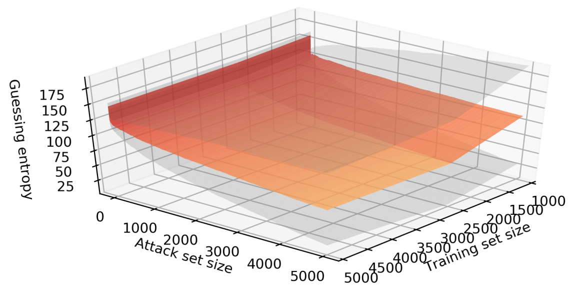


(b) CNN, value model, 2 convolutional block.

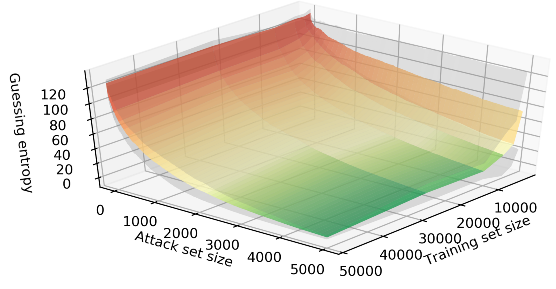
Figure 4.17: Guessing entropy decomposition, AES_RD dataset results for different training set sizes.



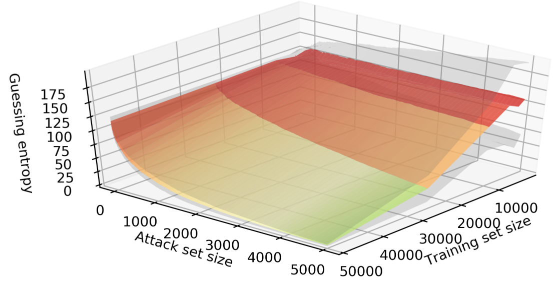
(a) Random forest, HW model, 100 trees.



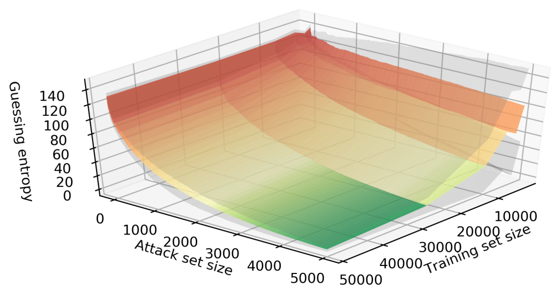
(b) Random forest, value model, 100 trees.



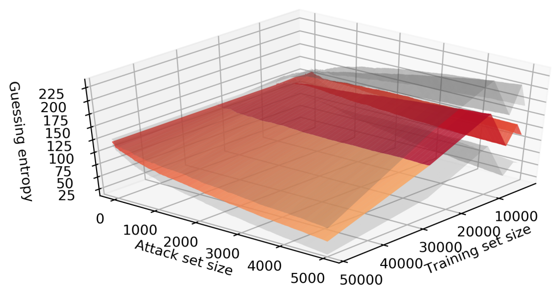
(c) MLP, HW model, 2 hidden layers.



(d) MLP, value model, 2 hidden layers.

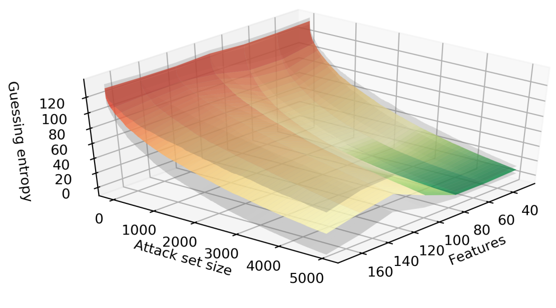


(e) CNN, HW model, 2 convolutional blocks

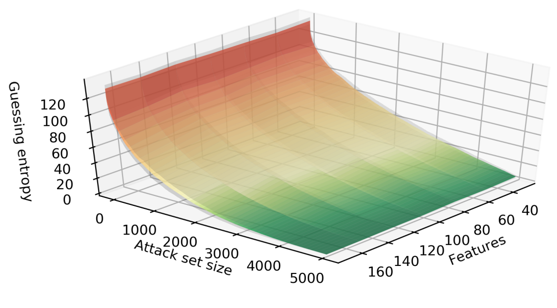


(f) CNN, value model, 2 convolutional blocks

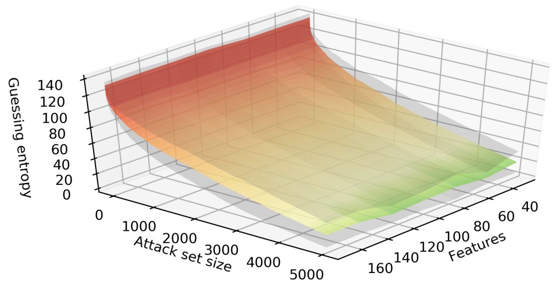
Figure 4.18: Guessing entropy decomposition, ASCAD dataset results for different training set sizes.



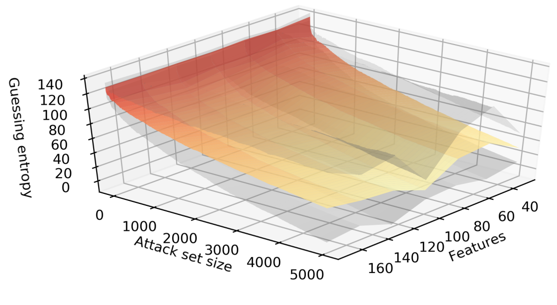
(a) Template Attack, HW model.



(b) Pooled Template Attack, HW model.

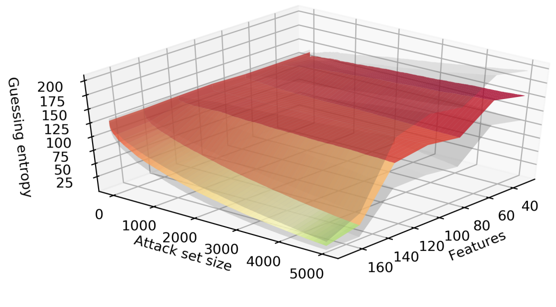


(c) Random forest, 100 trees, HW model.

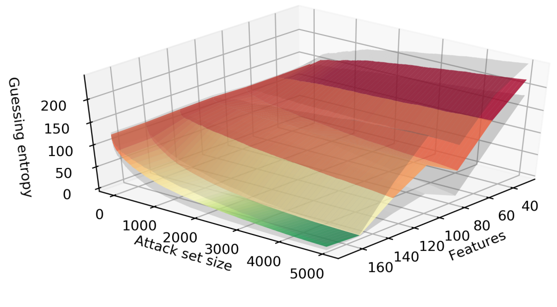


(d) MLP, 2 hidden layers, HW model.

Figure 4.19: Guessing entropy decomposition, AES_HD dataset results for different numbers of features.



(a) Random forest, HW model, 100 trees.



(b) MLP, HW model, 2 hidden layers.

Figure 4.20: Guessing entropy decomposition, ASCAD dataset results for different numbers of features.

5 Common Knowledge

In this chapter, we describe our work comparing neural network layers. We find a measure for the 'common knowledge' that may be shared among NNs in SVCCA. In most applications, only the output layer is analyzed: a network's ability to classify samples is evaluated on its outputs. The intermediate representation of the deep layers is typically ignored. However, this may reveal useful information about the network's interpretation of the data. A better understanding of the deep layers may be used to improve the usage of neural networks in side-channel analysis.

The method used for comparison is Singular Vector Canonical Correlation Analysis (SVCCA), which we formalize and describe in Section 5.1. We establish a baseline of this technique in Section 5.2, and show where it can and cannot be applied. Finally, we apply SVCCA to the portability datasets in Section 5.3.

5.1. Implementation of SVCCA

Raghu *et al.* [72] proposed Singular Vector Canonical Correlation Analysis to compare two layers in a network, based on the neurons' activation outputs. It uses the following definitions:

Definition 5.1.1 *A **neuron** i is defined by the output it generates over a dataset $X = x_1, \dots, x_N$. The ith neuron of layer l is represented by $\mathbf{z}_i^l = (z_i^l(x_1), \dots, z_i^l(x_N))$. Here, $z_i^l(x_j)$ indicates the output (a single number) of the neuron for data sample x_j. Thus, a neuron is a vector in $\mathbb{R}^N$.*

Such an output is also called an *activation vector*: for each neuron, it stores the outcome after the activation function is applied.

Definition 5.1.2 *A **layer** j is defined as the subspace spanned by its neurons, so a subspace in $\mathbb{R}^{N \cdot c_j}$. It is constructed as a $N \times c_j$ matrix, where c_j is the number of neurons in layer j. So, layer j is defined as $l_j = \mathbf{z}_1^{l_j}, \dots, \mathbf{z}_{c_j}^{l_j}$.*

Based on this definition, the SVCCA algorithm compares two such layers l_1 and l_2. It operates on two matrices, each having an entry per neuron per data sample. Notice that the layers can have a different number of neurons, but there should be an equal number N samples used to compare the layers. After the layers have been trained and their outputs have been stored as l_1 and l_2, the SVCCA procedure for layer comparison works as follows:

1. **Singular Value (SV) decomposition** of both layers separately. For both layers l_1 and l_2, their singular value (SV) decompositions are computed and outputted as $l_1' \subset l_1$ and $l_2' \subset l_2$. This transformation represents the same data in another form: matrixes l_1' and l_2' will still have N rows, but contain $L_1' \leq c_1$ and $L_2' \leq c_2$ columns, respectively.

   With this transformation, a preset percentage of the variance is explained. After this step, two reduced subspaces $l_1' \subset l_1$ and $l_2' \subset l_2$ are used as inputs for the next step.

2. **Canonical Correlation Analysis (CCA)** computes the linear transformations on l_1', l_2' to maximize correlation. These operations can be defined as matrices W_X and W_Y to operate on l_1' and l_2' respectively. The outputted subspaces $\tilde{l}_1 = W_X l_1'$ and $\tilde{l}_2 = W_Y l_2'$ are maximally correlated. So, the algorithm returns the following outputs:
   - CCA components: the number of components is $min(L_1', L_2')$, the smallest dimension of the SVD-reduced layers. For each component, we have:
     - The value of the CCA component for both of the networks, for each data sample in the comparison dataset. $o_m^i(x_j)$ denotes the value of the ith component for model m for data sample x_j;
     - the correlations $corrs = \rho_1, \ldots, \rho_{min(L_1', L_2')}$ which indicate how well each component correlates between both layers.
   - To express the output of SVCCA in a single metric, the SVCCA similarity $\overline{\rho}$ represents how well the representations of two layers are aligned with each other:

$$\overline{\rho} = \frac{1}{\min(m_1, m_2)} \sum_i \rho_i \tag{5.1}$$

Note that the first step of the algorithm, Singular Value Decomposition, is the backbone of Principal Component Analysis (PCA), which is commonly used in machine learning for data reduction. The SVCCA steps are depicted in Figure 5.1. The produced correlations $corrs$ and the average correlation $\overline{\rho}$ will be used as metric to evaluate common knowledge, expressing the similarity between the layers' representations.

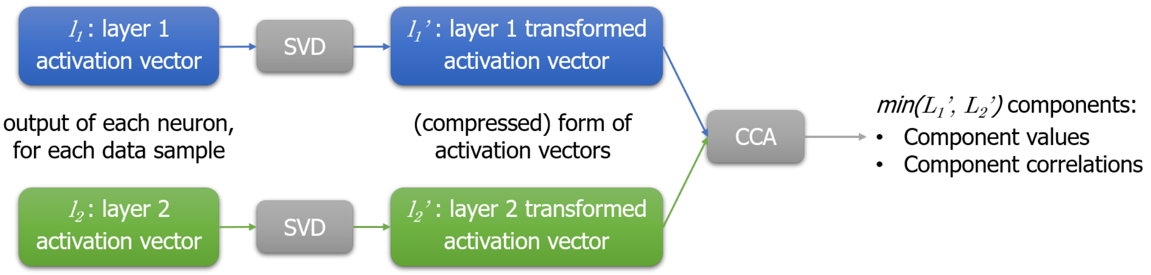

l_1: layer 1 activation vector
SVD
l_1': layer 1 transformed activation vector
output of each neuron, for each data sample
(compressed) form of activation vectors
CCA
$min(L_1', L_2')$ components:
• Component values
• Component correlations
l_2: layer 2 activation vector
SVD
l_2': layer 2 transformed activation vector

Figure 5.1: Overview of the operations in SVCCA. As input, it takes the activation vectors from both layers, containing each neuron's output for all samples in the dataset. The singular vector decomposition (SVD) is applied to both layers individually, resulting in a (reduced) matrix per layer. These matrices are compared using Canonical Correlation Analysis (CCA), which is another linear transformation that maximizes correlation between both layers. Both the correlations (from high to low correlation) and the values of the components, per sample in the dataset, are outputted.

5.2. Establishing a Baseline

In this section, the SVCCA methodology is tested to compare small MLPs with (nearly) the same architecture. We first describe the various datasets that are used to train these models. Then, we show the results of comparing networks in four scenarios: A) a single network compared with multiple others, B) comparing models against a copy that was differently initialized, C) the effect of changing the size of the comparison dataset and D) the effect of changing the comparison dataset itself.

As the main reference, the DPAv4 dataset with unmasked labels is used, as defined in Equation 2.19. This dataset is an easy target, even when using a relatively small network. We use 3 000 features covering the S-Box operation. When the leakage model is ambiguous, the dataset is marked as "DPAv4 (unmasked)". We also introduce several other datasets to set up a baseline comparison.

We also consider the attack on DPAv4, where masks are ignored (i.e., considered unknown). Then, Eq 2.19 changes to:

$$Y(k^*) = \texttt{Sbox}[PT_1 \oplus k^*], \tag{5.2}$$

and we simply classify by looking at the first S-Box output. This is referred to as "DPAv4 (ignoring masks)".

5.2.1. Other Training Datasets

To get a clearer picture of what SVCCA outcomes mean, we compare DPAv4-based models with several other datasets. These include a real dataset from another field (computer vision), a set of generated 'side-channel' measurements, and completely random data.

CIFAR-10

We have chosen the CIFAR-10 dataset[1] as reference problem from another field. It consists of 50 000 training and 10 000 test images. They are 32 x 32 pixels in 3 channels (colors); for this comparison we 'flatten' those to obtain 3 076 features. CIFAR-10 is an interesting dataset for this comparison, as:

- It is from a completely different field, so no patterns are expected to overlap between DPAv4 and CIFAR-10;
- a neural network can be built with a very similar architecture: it has 3 076 features and ten classes. This is quite close to the 3 000 measurements around the S-Box computation of DPAv4, where we can select nine classes (HW model);
- It was also used in the SVCCA paper [72] as a baseline.

Except for data normalization (see Section 2.1.2), the original dataset is used.

Generated Traces

To further compare with similar data, we have generated a random dataset that is close to common theoretical assumptions about side-channel measurements:

- There are 3 000 features: 2 900 are drawn completely random, from the standard normal distribution. The other 100 are semi-random:
  - For all classes, a class mean in computed for each of the 100 semi-random features.
  - For these 100 features, for some sample i, feature j is drawn like this: $x_{i,j} = 0.5 \cdot N(0,1) + 0.5 \cdot \mu_{k,j}$ where k is the class of sample i and $\mu_{k,j}$ indicates this class k's mean for feature j.
- The columns are shuffled randomly.

Although completely artificial, this dataset follows the Gaussian noise assumption. In the theoretical setting with an unlimited number of traces, the QDA classifier (template attack) will be the optimal classifier.

Random 'Outputs'

Finally, instead of computing activation vectors from some neural network layer, a matrix of the same size is randomly generated. As this has no relation with Deep Learning representations whatsoever, we expect to see no 'common knowledge', as this randomness does not contain any knowledge. All entries are randomly drawn from the standard normal distribution ($\sim N(0,1)$).

5.2.2. Experimental setup

Every model is an MLP with one hidden layer, consisting of 100 neurons that have ReLu activation. The next layer was the output layer, having either 9 (DPAv4 HW, Generated), 10 (CIFAR-10), or 256 (DPAv4 value) neurons. The output layer uses Softmax activation. The models were trained minimizing the categorical crossentropy loss, using the Adam optimizer, and 50 epochs were carried out. All models were trained on 25 000 samples.

Note that all networks had 3 000 inputs, except for those trained on CIFAR-10 which had 3 076. To adjust for this small mismatch, comparison data was either padded with zeros at the end (when there

[1]The CIFAR-10 dataset is available at `https://www.cs.toronto.edu/~kriz/cifar.html`.

Network 1	Network 2 Dataset	Indexes	Model	Scenario
DPAv4 (unmasked), HW model, trained on indexes 0–25000	DPAv4 (unmasked)	0–25000	HW	A_1 Same network
		0–25000	HW	A_2 Different initialization
		25000–50000	HW	A_3 Different part of dataset
		0–25000	value	A_4 More classes (value model)
	DPAv4 (ignoring masks)	0–25000	HW	A_5 Different leakage model: ignoring masks
		0–25000	value	A_6 Different leakage model: ignoring masks
	CIFAR-10	0–25000	–	A_7 Different dataset: CIFAR-10 images
	Generated	0–25000	–	A_8 Different dataset: generated traces
	No network outputs, but 'activation vector' randomly drawn from the standard normal distribution			A_9 Comparison with random data

Table 5.1: Baseline experiment A: a single network trained on the DPAv4 dataset in the HW model compared with itself and several other classifiers. The comparisons were based on the networks' deep layers' outputs, using all 100 000 traces in the DPAv4 dataset.

were not enough features) or cropped at the end (when there were too much features). The models generally performed well after training, with close to 100% accuracy for the DPAv4 and generated sets, and roughly 50% accuracy for the CIFAR-10 dataset. The comparison was conducted between the only hidden layer of the models.

5.2.3. Single Network vs Others

For each of the scenarios described in the previous sections, an MLP was randomly initialized and trained. To show the effect of changing training and comparison data, we only present the SVCCA comparisons for characteristic scenarios.

Table 5.1 shows the experimental setup for the first set of experiments A. It compares a model trained on the first 25 000 DPAv4 samples, with labels as described by the unmasked leakage model (Eq. 2.19, in HW) with itself and several other networks. The resulting SVCCA correlations are shown in Figure 5.2. For each comparison, the entire DPAv4 dataset is used to generate the models' activation vectors.

The blue line indicates the comparison with the exact same network's outputs (A_1). As expected, it shows a perfect correlation ($\rho_i = 1$ for all i). Next, we see a comparison with a MLP trained on exactly the same data, but with a different random initialization of the model's weights before training (A_2, orange line, $\overline{\rho} = 0.5646$). The green line (A_3) compares with yet another network, trained on the next 25 000 DPAv4 samples, showing a slightly lower correlation ($\overline{\rho} = 0.5404$).

A slight modification of the base network's problem is the switch from HW to intermediate value labels (A_4, $\overline{\rho} = 0.4162$). Based on the same data, a network can also learn the original S-Box output (i.e., Eq 5.2) in the HW (A_5, $\overline{\rho} = 0.3992$) or value model (A_6, $\overline{\rho} = 0.3287$). Although these DPAv4-related models still show some similarity with the base network, we observe a stronger similarity between the base network and some unrelated networks. In particular, we see a slightly higher correlation with the CIFAR-10 network (A_7, $\overline{\rho} = 0.4429$) compared to the aforementioned networks. Roughly on the same levels as the others, we see the similarity with the model trained on generated data (A_8, $\overline{\rho} = 0.4031$).

Finally, as expected, we see almost no correlation with a completely random vector drawn from the standard normal distribution (A_9, $\overline{\rho} = 0.0271$).

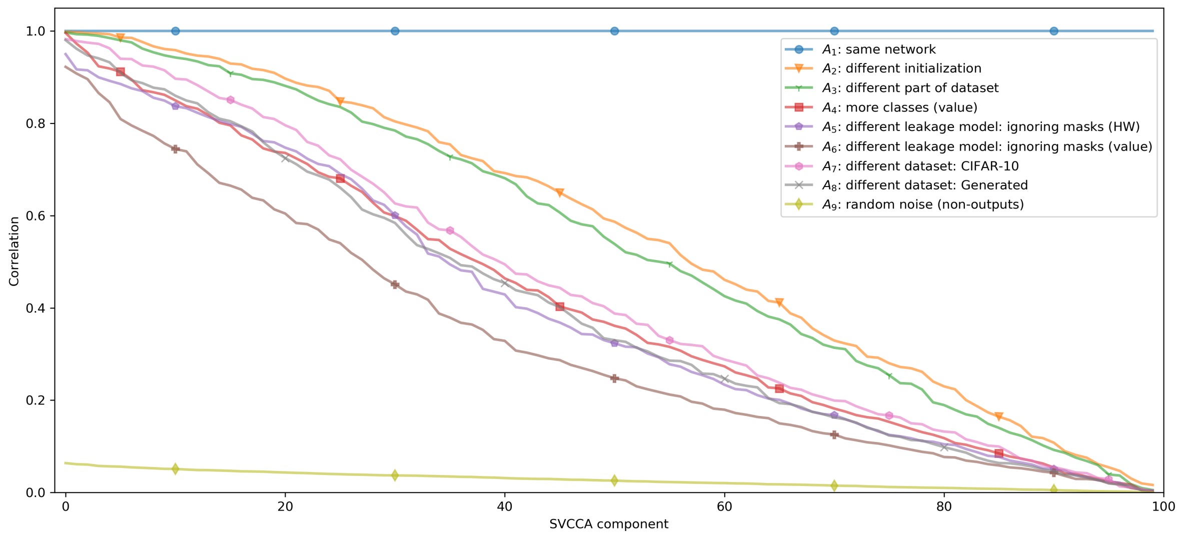


Figure 5.2: Results for baseline experiment A, the SVCCA comparison between a model trained on DPAv4 (HW) and several other models. A detailed description of these settings is listed in Table 5.1.

Label-Based Inspection

To further investigate how the SVCCA outcomes behave in this scenario, Figure 5.3 shows the values of the first component for each of the scenarios in Figure 5.2 – except the first one, as identical models result in identical outcomes. Along the x-axis, 100 samples are randomly selected for all 9 classes in the HW model. The y-axis shows the first SVCCA component value per data sample. Notice that SVCCA is purely based on the activation vectors, in this case of the hidden layer. SVCCA is not based on class labels, but interestingly enough, we can clearly see a relation between the class label and the first component value for some scenarios.

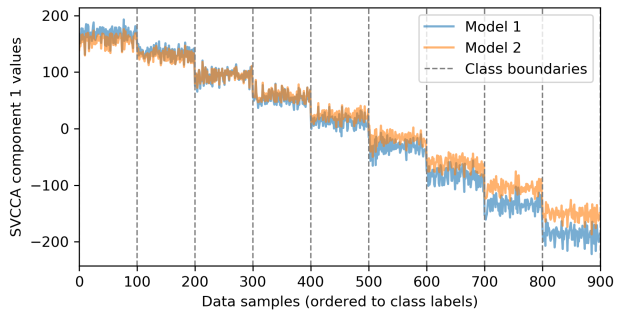
To make this more precise, we need some definition for the relation between a SVCCA component and the class label. For this, a simple correlation metric suffices:

Definition 5.2.1 *The **class-correlation** of some experiment E for model $m \in 1,2$ for an ordered list of class labels Y is $\rho^{E}_{Y,m}$. It equals the Pearson correlation[2] between the first SVCCA component, based on experiment E, and the class labels Y, for one of the two compared models m.*

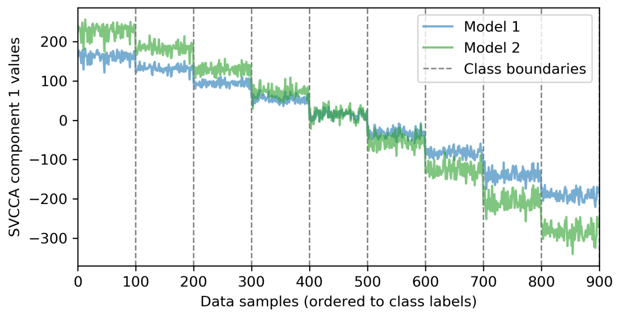
In this case, Y are the labels of DPAv4 where the known mask is removed (Eq. 2.19), in the HW model. We observe that when different MLPs are trained on exactly the same data, there is a very high correlation with the class label for both networks (Figure 5.3a, $\rho^{A_2}_{Y,1} = -0.9824$, $\rho^{A_2}_{Y,2} = -0.9827$). This means that the strongest pattern (i.e., first component) that SVCCA finds among the deep layer's outputs, are highly related with the class labels learned by the networks. When training on similar, but not identical data, we see a comparable situation (Figure 5.3b, $\rho^{A_3}_{Y,1} = -0.9826$, $\rho^{A_3}_{Y,2} = -0.9820$). In the next scenario, we compare with a model trained with identical data, but taking intermediate values (0–255), instead of HW, as labels. Notice that these 256 classes are 'encapsulated' in the 9 HW-based classes. For example, value '42' always maps to Hamming Weight '3'. Figure 5.3c shows the result: again, we see observe a high correlation between the first component values for both models, and the HW class label $\rho^{A_4}_{Y,1} = -0.9895$, $\rho^{A_4}_{Y,2} = -0.9885$.

In the other scenarios of Figure 5.2, we see lower correlations between the first SVCCA component and labels. When comparing with a model trained for DPAv4 HW labels while ignoring the mask (Equation 5.2), we see no significant relation between the component values and the class labels (Figure 5.3d, $\rho^{A_5}_{Y,1} = 0.0371$, $\rho^{A_5}_{Y,2} = 0.0356$). Clearly, the most similar patterns in these layers say nothing meaningful about the samples' classes.

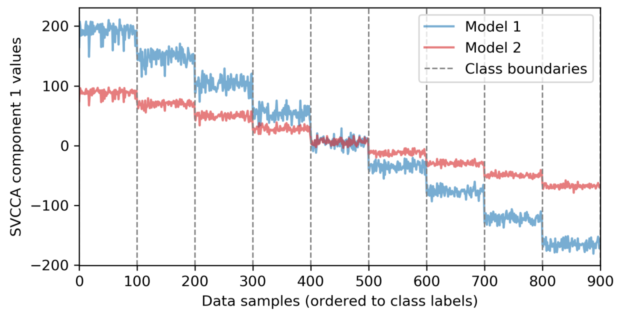
[2]Pearson correlation is defined in Equation 4.1.



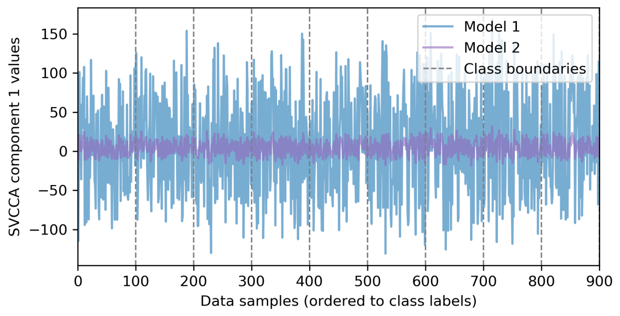
(a) A_2: Different initialization.



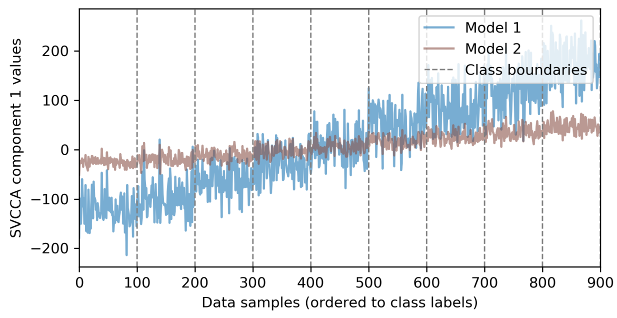
(b) A_3: Different part of dataset.



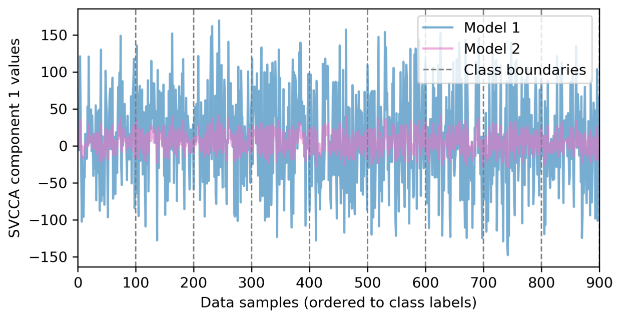
(c) A_4: More classes (value model).



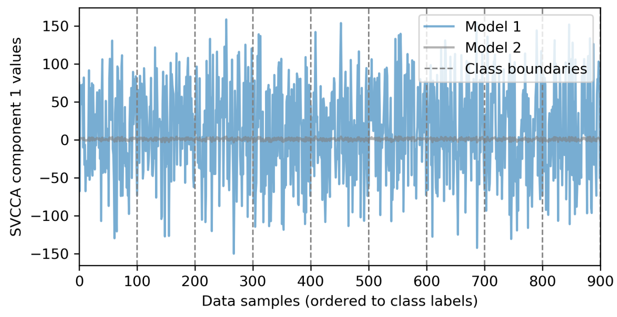
(d) A_5: Different leakage model: ignoring masks (HW model).



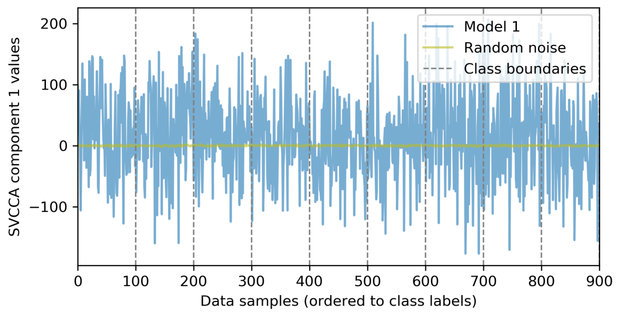
(e) A_6: Different leakage model: ignoring masks (value model).



(f) A_7: Different dataset: CIFAR-10 images.



(g) A_8: Different dataset: generated traces.



(h) A_9: Random noise, instead of a network's outputs.

Figure 5.3: First SVCCA component for comparisons as described in Table 5.1 and shown in Figure 5.2. Along the x-axis, data samples are sorted according to class label; for each class, 100 samples were randomly selected. The y-axis indicates the value of SVCCA component 1 for these samples. When training in the same leakage model (whether it's HW or value), a strong correlation between the class label and SVCCA component is observed.

In conclusion, for models trained on similar data and the same leakage, we see their internal representation is extremely similar. It does not matter whether intermediate value or HW labels are used. Although SVCCA is not provided with class labels, the underlying patterns it finds show that the MLPs have an extremely similar internal representation, aiming for a high class separability.

Based on the outcomes of experiment A, we can draw the following conclusions:

- When comparing networks trained on HW labels to those on intermediate value labels, the inner

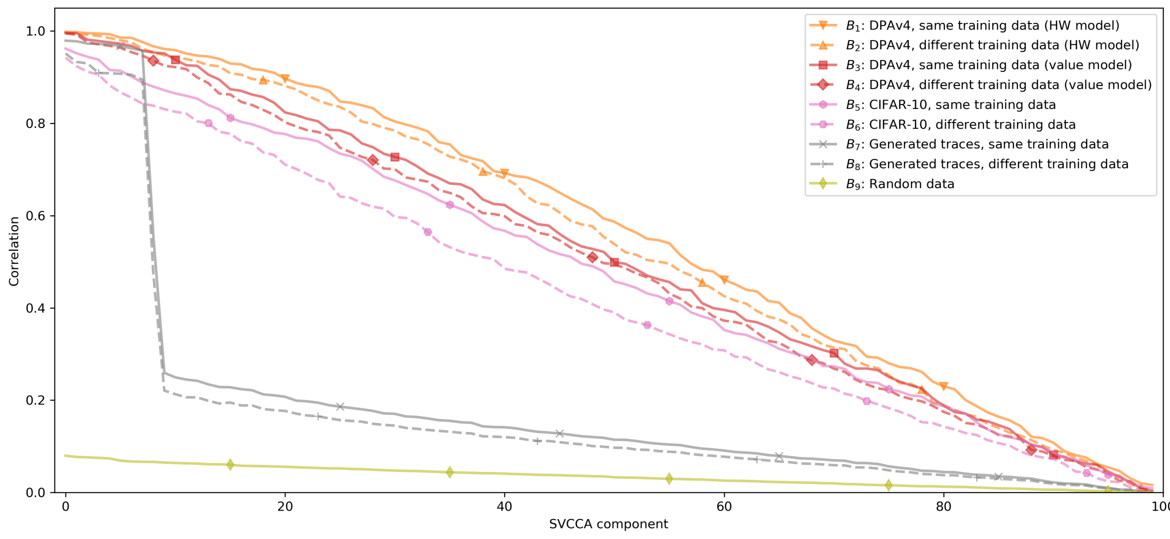


Figure 5.4: Results for baseline experiment B. It includes comparisons between different instances (randomizations) that either use exactly the same, or very similar, training data.

representation can be very similar;

- SVCCA can find relations which are apparently meaningless; e.g., it can find correlation between an SCA dataset (DPAv4) and an image dataset (CIFAR-10);
- When changing the leakage model of the labels, i.e., ignoring the masks or not, the correlation between models drops heavily. In our case, it was a similar relation as that with the image dataset. This suggests SVCCA is not a good tool for comparing arbitrary SCA datasets;
- Despite the fact that SVCCA is independent from class labels, its components can be highly correlated with the labels.

5.2.4. Networks Trained on the Same Dataset

Figure 5.4 shows the results of experiment B, comparisons between pairs of neural networks that were trained on the same or similar data. Three datasets were used to train the networks on: DPAv4 in both the HW (B_1, B_2) and value model (B_3, B_4), CIFAR-10 (B_5, B_6) and the generated traces (B_7, B_8). Finally, we compare two randomly drawn matrices from the standard normal distribution (B_9).

Pairwise, the comparison was made in two settings: NNs trained on exactly the same data (full lines), and trained on different parts of the same dataset (dashed lines). For the generated dataset (B_7), we observe something interesting: there is a sudden drop in the correlation after the 8th principal component. We see $\rho_8 = 0.9579, \rho_9 = 0.5554, \rho_{10} = 0.2593$. This may be explained by the class labels: there are 9 classes with distinguishable class means around the 100 useful features. The deep layer may simply focus on these class means without outputting anything else.

Although there are slight variations, the other NN comparisons all show the same behavior. Consistently, there is a lower correlation when using different data, compared to identical data. There seems to be some difference between datasets: apparently, the variance between the functions learned for DPAv4 are lower than those for CIFAR-10. Again, we observe almost no correlation when comparing with the baseline of random data (B_9, $\overline{\rho} = 0.0347$) instead of activation vectors.

5.2.5. Comparison Data

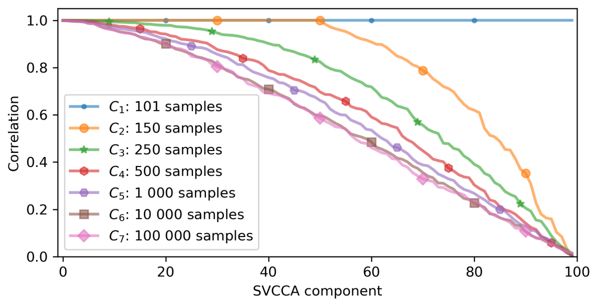
Apart from the networks' training inputs, the comparison data also influences the compared layers' activation vectors. A layer is completely defined by its `input⟶output` mapping, which is given by its weights and biases. Raghu *et al.* [72] argue that comparing outputs for all possible inputs is both infeasible and irrelevant:"Our primary interest is not in the neuron's response to random data, but rather in how it represents features of a specific dataset (e.g., natural images). Therefore, in this study we take a *neuron's representation to be its set of responses over a finite set of inputs* — those drawn from some training or validation set" [72]. However, this poses some practical challenges: it doesn't tell us *how much* comparison data we need, nor which data to use when the dataset is small (e.g. matrix operations do not converge) or when networks are trained on different datasets.

In the final two baseline experiments, we research the choice of comparison set(s): the dataset that is fed into both neural networks to produce the activation vectors. Experiment C, shown in Figure 5.5a, shows the relation between comparison set size and resulting SVCCA correlations. We compare two neural networks trained on DPAv4 with HW labels, that have been initialized differently before training. Note that for 100 neurons, SVCCA needs at least 101 data samples to converge. We observe that a small comparison set will sketch a too optimistic view of the correlation among models: using 101 comparison samples, a perfect correlation is found: $\overline{\rho} = 1.0000$ (blue line). When increasing the comparison set slightly to 150 (orange, $\overline{\rho} = 0.8184$), 250 (green, $\overline{\rho} = 0.7073$), 500 (red, $\overline{\rho} = 0.6351$) or 1 000 (purple, $\overline{\rho} = 0.6039$) samples, the maximized correlations steadily decrease. Using 10 000 samples (brown, $\overline{\rho} = 0.5704$) gives almost the same result as using all 100 000 samples (pink, $\overline{\rho} = 0.5646$) from the original DPAv4 dataset.

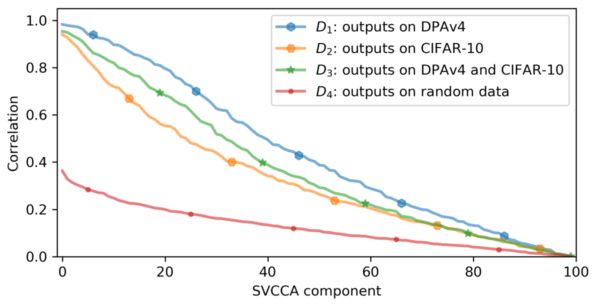
In conclusion, using a too small comparison dataset will result in SVCCA finding very high correlation among models, yielding little meaningful results. Note that these results should be considered in relation to the other dimension of the activation: the number of neurons whose output is considered, is fairly low at 100.

Finally, Figure 5.5b shows the comparison of a model trained for DPAv4 (HW labels) with a model trained for CIFAR-10, using different comparison sets. The choice of comparison set has a large influence on the relations found by SVCCA. The blue line for experiment D_1 indicates the comparison using the full DPAv4 dataset; in this case, we see a high overall correlation ($\overline{\rho} = 0.4429$). Comparing using the model outputs on CIFAR-10 data (D_2, orange line) paints a less optimistic view, with $\overline{\rho} = 0.3310$. In D_3, the layers' outputs of both networks were combined to compute the comparison. Based on D_1 and D_2, it is unsurprising that the correlations found are somewhere in between (D_3, green line, $\overline{\rho} = 0.3745$). Finally, computing activation vectors based on random data drawn from the standard normal distribution (D_4, red line, $\overline{\rho} = 0.1233$) results in only a very low relation between both networks.

As the authors of [72] claimed, the comparison is influenced by the dataset which is used to compute the activations. When there is a generous amount of data available, we should use it to get a more realistic sense of the relation between layers. Comparisons with only very little data compared to the number of neurons ($C_1 - C_3$) should be avoided.



(a) Comparing using different numbers of DPAv4 samples



(b) Comparing using different datasets

Figure 5.5: Influence of the comparison dataset: the data that is used to generate activation vectors, to represent the layers.

5.3. Portability

In the previous section, we concluded that SVCCA is not a suitable method to compare neural networks trained on entirely different datasets. When studying the portability of models (see Section 2.4.4), however, we deal with very similar data. In this section, we study whether portability is indeed an issue in deep learning-based SCA: when switching to a copy of the device, is the data represented differently by a neural network? Can we avoid overfitting to a single device?

Bhasin *et al.* [7] propose several model architectures, of which we use two:

- **MLP**: a small multi-layer perceptron with three deep layers, having 50, 25 and 50 neurons. The input layer consists of 50 features, which are selected based on Pearson correlation.
- **MLP2**: a multi-layer perceptron with four deep layers, having 500 neurons each. For this architecture, all 600 features are used.

Both networks aim to optimize the categorical cross-entropy, with a learning rate of 0.001. Following [7], we train those networks either using 10 000 or 40 000 training samples. In their work, they observed a better generalization for both architectures when there were fewer training samples used.

Note that the implementation is unprotected and the intermediate value model (256 classes) is used to generate the labels. Three datasets are used (1, 2, 3). Note the relation between the datasets:

- Dataset 1 and 2 are taken from different devices, but have the same key;
- Dataset 1 and 3 are taken from different devices that use different keys;
- Dataset 2 and 3 are taken from the same device, using different keys.

Bhasin *et al.* [7] based the MLP and MLP2 architectures on tuning results for 50 and 600 features, respectively. They were chosen based on their 'general' performance, without finetuning for a specific dataset. As we would like to study the effects of optimizing for a single device, we created an optimized architecture for dataset 1. We conducted a grid search with 0–5 hidden layers, having 100/200/300/400/500 neurons per layer, for learning rates 10^{-lr} with $lr = 2, 3, 4, 5$. The model was trained on 40 000 samples and uses all 600 features. Based on validation accuracy of 5000 samples, we propose the following architecture optimized for dataset 1:

- **Optimized**: a multi-layer perceptron with two hidden layers, having 300 and 100 neurons. It is trained using the Adam optimizer, with a learning rate of 0.001.

Although a CNN architecture is proposed in [7], we do not apply the SVCCA analysis on it, due to practical limitations. Although the number of parameters in convolutional layers is low, they produce massive activation vector. The CNNs' first layers' outputs are roughly 750 times larger than those of the MLP, taking 28.4 GB to store a single convolution activation vector on disk. Although the analysis of CNN is definitely interesting, we leave this for future work.

We first train MLP and MLP2 networks based on only one dataset (i.e., dataset 1, 2, or 3), and see how closely those datasets are represented in the hidden layers. We also compare with the optimized architecture for dataset 1.

Then, we use the Multiple Device Model (MDM) to train networks on two datasets, and verify that they generalize better with fewer training data from each set.

5.3.1. Training using Single Dataset

In this section, we compare MLP, MLP2 and optimized models trained on a single dataset. Note that the MLP and optimized architectures are much smaller than the MLP2 architecture. In particular, the MLP networks may have more difficulty in learning complicated functions compared to MLP2. However, a smaller network also reduces the risk of overfitting.

Figure 5.6 shows a comparison between the first deep layers of several MLP instances. Comparing networks trained on different datasets (1 vs 2, 1 vs 3, 2 vs 3) seems to result in a homogeneous amount of common knowledge. Also, the training set size seems to have no influence on the correlation between networks. This may be explained by MLP's small architecture: the networks roughly learn the same function which approximates the training data, but do not overfit. We omitted the results for the second and third deep learning in this setting: they produced very similar results.

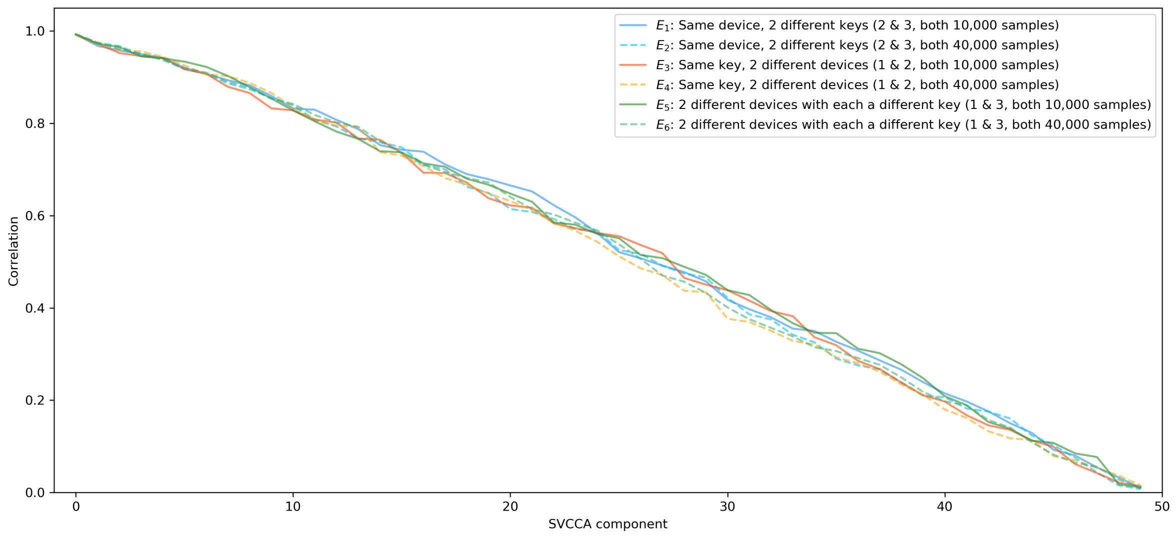


Figure 5.6: Experiment E: comparing **MLP** instances, in the first deep layer (50 neurons). We compare neural networks trained on a single dataset (1, 2, or 3). Independent of the dataset combination that is compared, and whether a large or small dataset is used, we see the same level of similarity.

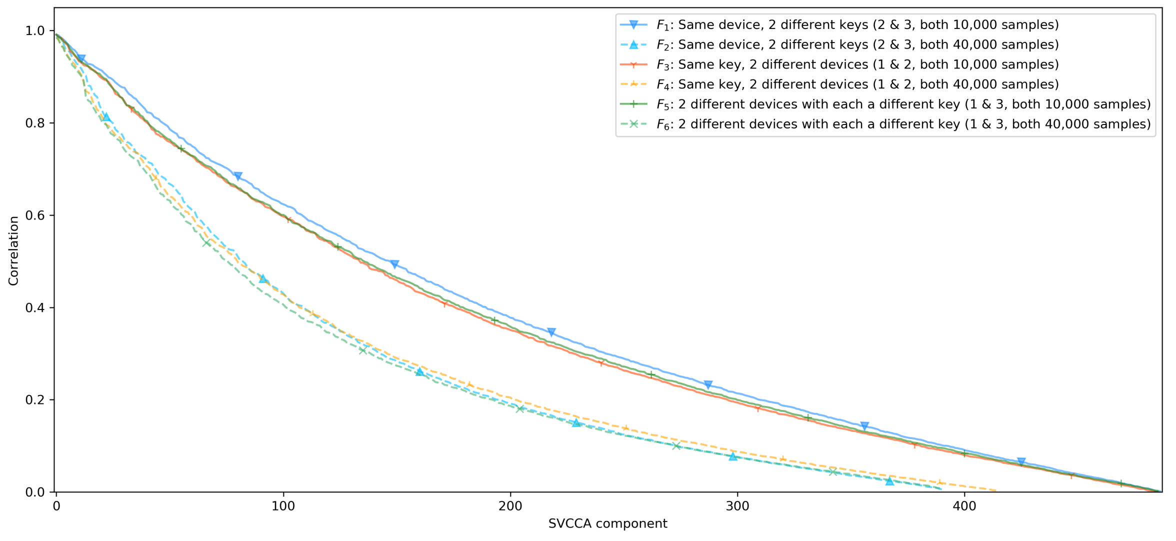


Figure 5.7: Experiment F: comparing **MLP2** instances, in the second deep layer (500 neurons). We compare neural networks trained on a single dataset (1, 2, or 3). Although the combination of datasets is not of significant influence, we observe a much more similar representation in the networks trained on fewer data.

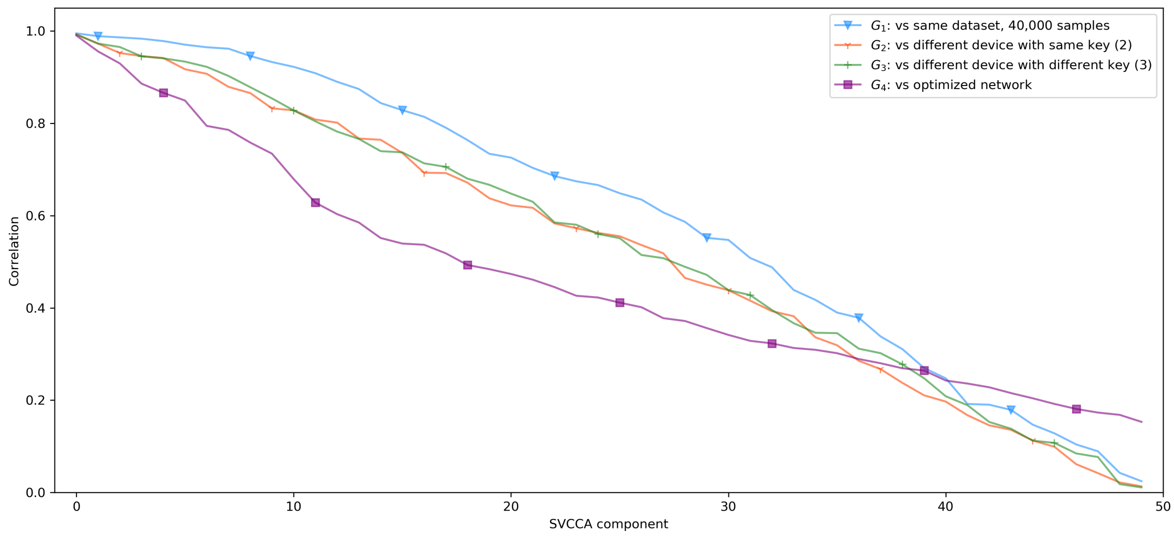


Figure 5.8: Experiment G: comparing **MLP** instances, in the first deep layer (50 neurons). In each comparison, the first model is an MLP trained on 10 000 samples from dataset 1. We compare against MLPs trained on 1) 40 000 traces from dataset 1, 2) 10 000 traces from dataset 2, and 3) 10 000 traces from dataset 3. Finally, 4) compares with the **optimized** architecture which has 100 neurons in its first deep layer, and is trained on 40 000 traces from dataset 1.

For the MLP2 architecture, we observed similar results in the first and the last deep layer. The middle of the network, however, paints a different picture. Figure 5.7 shows the comparison between the second deep layer of the MLP2 networks. Here, we observe a much lower correlation between networks trained on a large dataset (40 000 training samples). This may indicate that with a larger network like MLP2, the neurons fit much more precisely around the training data. Although Bhasin *et al.* [7] report a better performance in these larger networks, their specialization leads to a divergence from models learning other (large) datasets. The third deep layer showed similar results to those shown in Figure 5.7.

Experiment G (Figure 5.8) compares an MLP instance trained on 10 000 samples from dataset 1 with various other models. First, we compare to an MLP trained on very similar data (G_1): 40 000 training samples from dataset 1. There is relatively much common knowledge between those networks. MLPs that were trained on a different device (G_2 and G_3) show a little less common knowledge. Interestingly, the highly optimized network for dataset 1, trained on 40 000 samples, has the least correlation with the base network. This may be a result of specialization and overfitting of the optimized architecture. Figure 5.8 shows the result for the first deep layer; the other layers characterize in the same way.

Similarly to G, experiment H compares a single MLP2 instance to several other models. The results of the second deep layer are shown in Figure 5.9. As the optimized network has only 100 neurons in it's second layer, SVCCA results in 100 components (H_4). Again, we see the optimized network showing much less common knowledge with the model. But experiments G and H show an interesting difference between the small MLP and larger MLP2 architectures. With MLP, we see G_1 (same training set, more samples) results in more similar representation than G_2 and G_3 (different training sets). With MLP2, the correlations found in H_2 and H_3 surpass those from H_1. In other words, a large network has more in common with networks trained on a small set with a different key and/or device, than with its 'specialized' counterpart that uses more samples from the same training set. The other deep layers confirm this finding, except the last deep layer. In the fourth layer, all comparisons result in similar fashion (like those in Figure 5.6).

Finally, we verify our suspicion that adding more training samples leads to a more specialized function in the deep layers. Experiment I (Figure 5.10) repeats the experiment H, but uses the MLP2 trained on 40 000 traces from dataset 1 as the reference model. We observe a stronger relation with the 'simpler' MLP2 model trained on 10 000 samples from the same dataset (I_1). As expected, this results in higher correlations compared to other models trained on large, different dataset (I_2, I_3) and

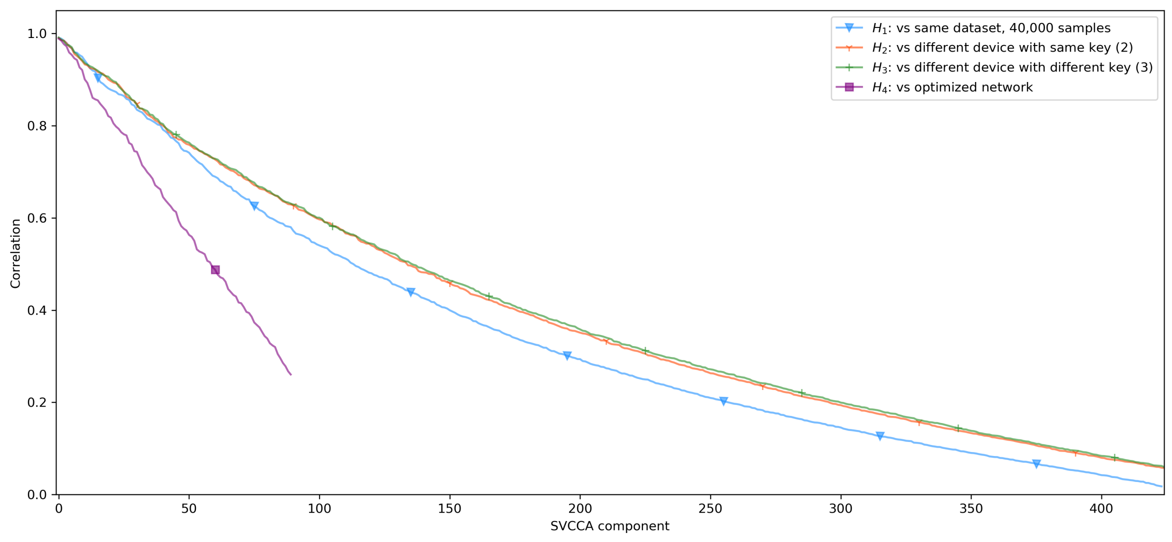


Figure 5.9: Experiment H: comparing **MLP2** instances, in the second deep layer (500 neurons). In each comparison, the first model is an MLP2 trained on 10 000 samples from dataset 1. We compare against MLP2s trained on 1) 40 000 traces from dataset 1, 2) 10 000 traces from dataset 2, and 3) 10 000 traces from dataset 3. Finally, 4) compares with the **optimized** architecture which has 100 neurons in its first deep layer, and is trained on 40 000 traces from dataset 1.

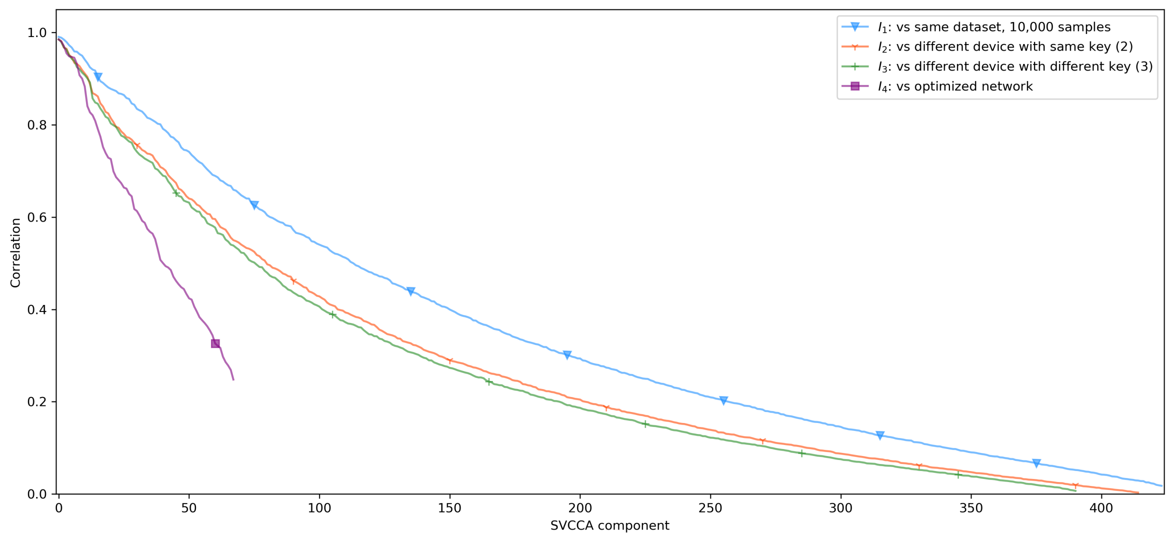


Figure 5.10: Experiment I: comparing **MLP2** instances, in the second deep layer (500 neurons). In each comparison, the first model is an MLP2 trained on 40 000 samples from dataset 1. We compare against MLP2s trained on 1) 10 000 traces from dataset 1, 2) 40 000 traces from dataset 2, and 3) 40 000 traces from dataset 3. Finally, 4) compares with the **optimized** architecture which has 100 neurons in it's first deep layer, and is trained on 40 000 traces from dataset 1.

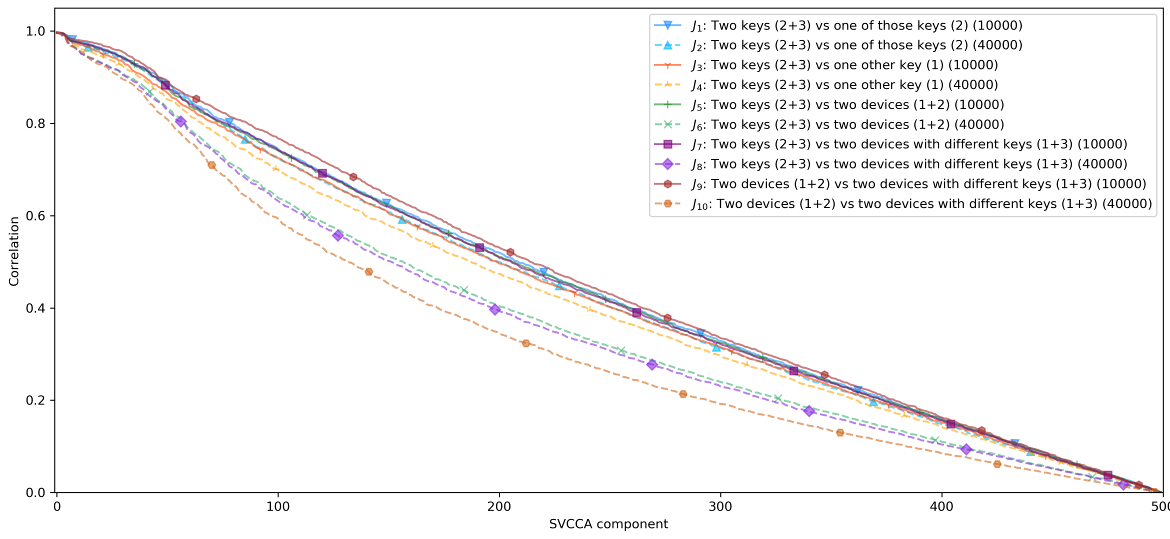


Figure 5.11: Experiment J: comparing **MLP2** instances, in the first deep layer (500 neurons). We compare models trained on two datasets with each other and with those trained on a single dataset. Again, the scenarios seem of little influence on the internal similarity, but the networks trained on a large dataset (40 000 traces) are less similar than those trained on a small dataset (10 000 traces).

the optimized network for dataset 1 (I_4).

5.3.2. Training using Multiple Datasets

Finally, we combine datasets together, so networks learn from two datasets at the same time. This can either involve two different keys on the same device (datasets 2+3), two devices using the same key (datasets 1+2), or two devices having different keys (1+3). One would expect that a network trained on dataset 1 and 3 would generalize best, as both key and device are mixed. We trained networks with the same settings as before, but now using 10 000 and 40 000 training samples *per dataset* from these combinations.

For MLP, training using multiple datasets gave us little information. Just like experiment E (Figure 5.6), all correlations were roughly the same. Again, we assume this architecture is too small to specialize for a specific key and/or device. Therefore, the results on the MLP architecture were omitted.

Figure 5.11 (experiment J) shows the result for the first deep layer of MLP2 networks. Like with MLP, we see no significant difference between the combinations. Training with datasets 1 and 3 does not increase the common knowledge, compared to the combinations of sets (1, 2) or (2, 3). However, we do observe the same phenomenon showed in the previous section: the size of the training sets has a large influence. This explains the findings of [7] in their Multiple Device Model (MDM): when testing on another key and/or device, it's better not to overtrain your network. In these settings, MLPs and MLP2s performed better when using a relatively small training set. The final deep layer paints a similar picture as the results in Figure 5.11.

When inspecting the middle layers (the second and third), we see this specialization due to a large training set even better. Figure 5.12 shows the same experiment as J, but compares the second layer rather than the first. A much larger drop in common knowledge is observed for models trained on a large dataset. This shows that specialization particularly takes place in the middle layers.

To summarize:

- There is some common knowledge across networks that were trained on very similar data. We observe a similar level across the portability scenarios; it doesn't matter much whether the device, key, or both, are changed;
- When looking from a portability perspective, one should be careful not to overtrain his NN with

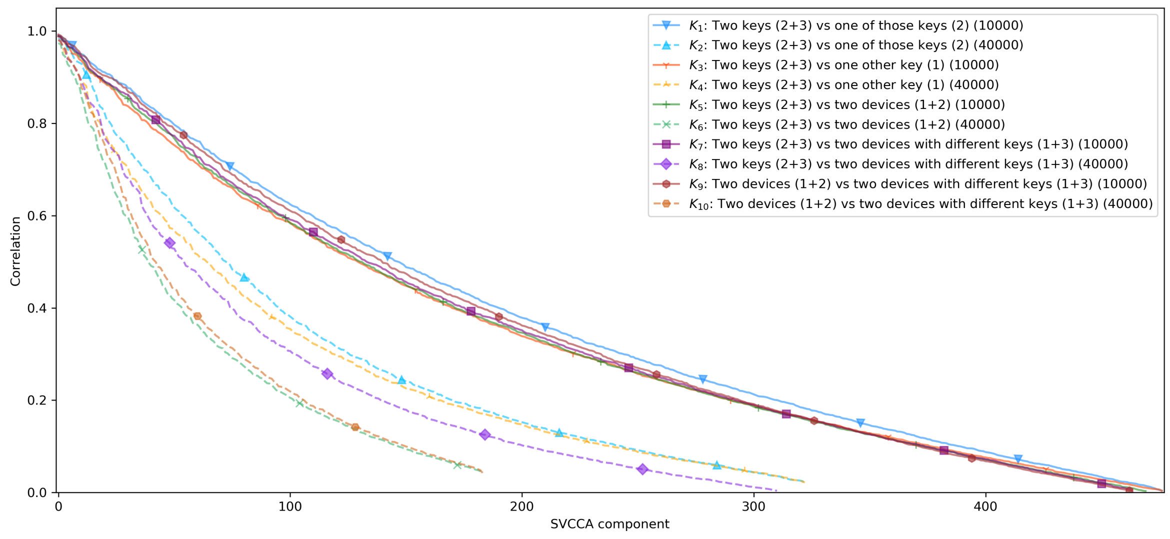


Figure 5.12: Experiment K: again comparing **MLP2** instances, but in the second deep layer (500 neurons). More extreme than in experiment J, we observe a large influence of the training set size. With a larger training set, particularly for a cross-device setting, the similarity decreases.

too much data. The SVCCA correlations decrease when networks are trained with more data, particularly when using a highly tuned model. This is in accordance with [7], which suggests a smaller training set (10 000 traces) generalizes better than a large on (40 000 traces) for the MLP and MLP2 architectures;

- SVCCA indicates that the middle layers (e.g., deep layers 2 and 3 in MLP2) specialize more than the first and last deep layer.

6 Mimicking

In this chapter, we explore mimicking: imitating a neural network by another one. As explained in Section 3.2.2, a network with one single hidden layer with ReLu activations can approximate any function. However, training such a narrow network is typically difficult. Literature has shown that narrow networks can benefit from mimicking a more complex network, compared to learning the original class labels [77, 78].

The findings in Chapter 4 show that a larger complexity is not always beneficial when attacking side-channel datasets. Figure 4.15e shows that the AES_RD dataset is best attacked with a fairly complex CNN, in the HW model. Based on this result, this chapter demonstrates the use of mimicking for the AES_RD dataset. From the different models trained in Chapter 4, the best model was selected based on guessing entropy: a CNN with 5 convolutional blocks. The architecture is listed in Table 4.2. From this single model, predictions were generated for the entire AES_HD dataset.

In next following Section 6.1, we describe how the experiments were conducted. Then, results are shown in Section 6.2.

6.1. Experimental Setup

We tried mimicking the big CNN with various architectures. All these networks share a common structure: there is only one fully connected layer, which has ReLu activation. All output layers consist of 9 neurons (HW model) and use Softmax activation. The number of neurons in the hidden layer was set at 100, 500, 1000. We also tried to mimic using shallow CNNs: between the input and fully connected layer, a convolutional layer was introduced. Both the MSE and CCE loss functions were tried; as categorical cross-entropy consistently scored better, we only show the results with that error. Note that the big CNN used for the original attack was optimized for MSE. We used the Adam optimizer with learning rate 0.001.

Only characteristic settings are shown here. We evaluate the performance of mimicking for the following scenarios:

1. **MLP-500**: an MLP using all 3500 features, having one fully connected layer with 500 neurons. It has 1 755 009 trainable parameters.
2. **MLPr-100**: an MLP using 200 features selected on Pearson correlation. It has 100 neurons in the fully connected layer and only 21 009 trainable parameters.
3. **MLPr-1000**: an MLP using 200 features selected on Pearson correlation. It has 1000 neurons in the fully connected layer and 210 009 trainable parameters.
4. **CNN-8-3-500**: a CNN using all 3500 features. It has one convolutional layer with 8 filters of size 3, followed by a fully connected layer with 500 neurons. It has 13 997 041 trainable parameters.

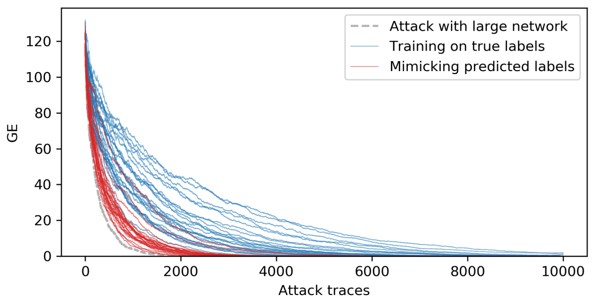
5. **CNN-32-3-100**: a CNN using all 3500 features. It has one convolutional layer with 32 filters of size 3, followed by a fully connected layer with 100 neurons. It has 11 194 737 trainable parameters.

Note that the original model had 24 981 949 trainable parameters. Per setting, 25 models were trained on the original labels, and 25 models were trained on the predictions of the teacher CNN from Chapter 4.

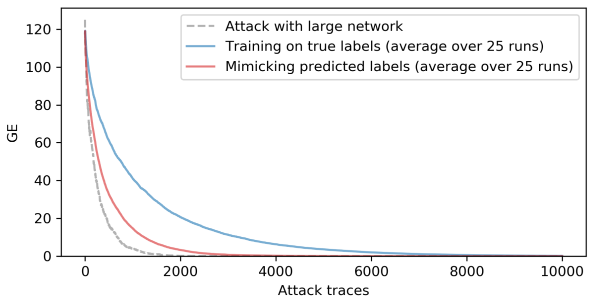
6.2. Results

In this section, we discuss the results of the mimicking attacks on AES_RD. For each setting, we show the guessing entropy of all 25 runs, as well as the average over those runs. The dashed grey line indicates the performance of the teacher model.

Figure 6.1 shows the performance of the **MLP-500** models. Here, we see a clear advantage of mimicking: on average, the networks that mimic the CNN reach a guessing entropy of 0 after roughly 3000 traces. When training on original labels, it takes 7500 attack traces on average to find the correct key byte. This shows mimicking is clearly favorable for this architecture. Although its attack is not as effective the teacher's attack, its architecture is almost 14 times smaller in terms of parameters.



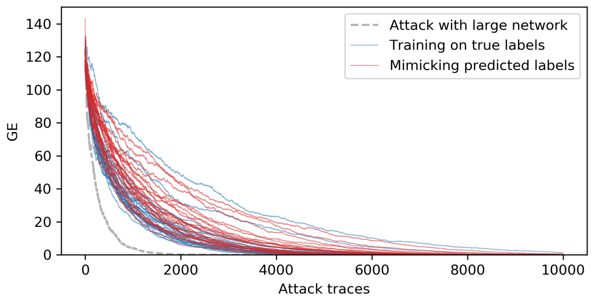
(a) Performance of all attacks



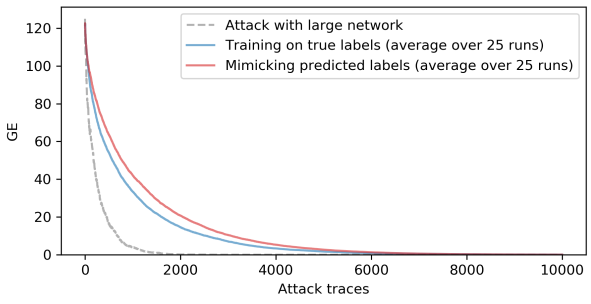
(b) Performance, averaged

Figure 6.1: **MLP-500**: MLPs using all 3500 features, having one fully connected layer with 500 neurons.

Figures 6.2 and 6.3 show the results for the reduced MLPs, which use only 200 features. In particular **MLPr-100** is extremely fast, having only 21 009 trainable parameters. However, for both scenarios, there seems to be no advantage of mimicking. For **MLPr-100**, the attack using the original labels is slightly better on average. For **MLPr-1000**, the guessing entropy of both strategies is almost equal; mimicking performs only slighly better because one network doesn't converge properly.



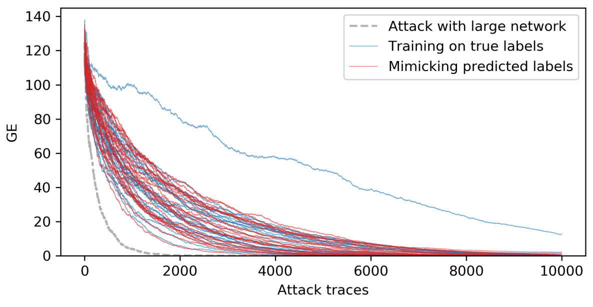
(a) Performance of all attacks



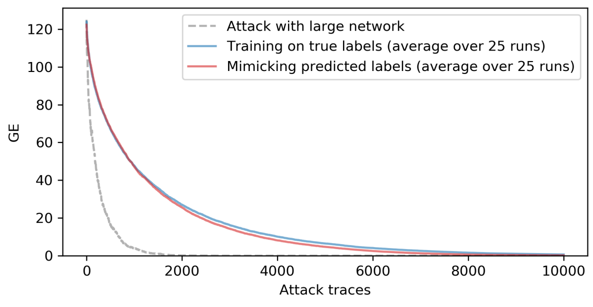
(b) Performance, averaged

Figure 6.2: **MLPr-100**: MLP using 200 features, having one fully connected layer with 100 neurons.

Finally, we look at the narrow CNNs. Subfigures 6.4a and 6.5a indicate that not all narrow CNNs converge in this configuration. This applies to both network designs, for both learning strategies. This makes the results somewhat ambiguous. When looking at the average GE scores naively, this suggests **CNN-8-3-500** (Figure 6.4b) benefits from mimicking, but **CNN-32-3-100** (Figure 6.5b) does not. However, the non-converging networks give a large penalty to both averages.



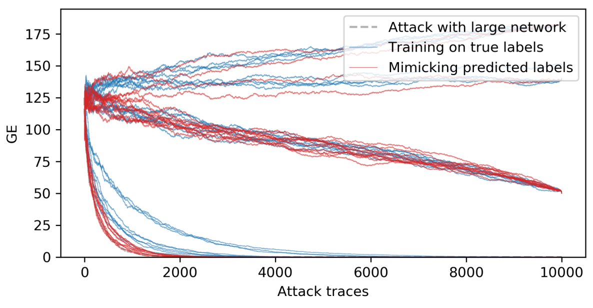
(a) Performance of all attacks



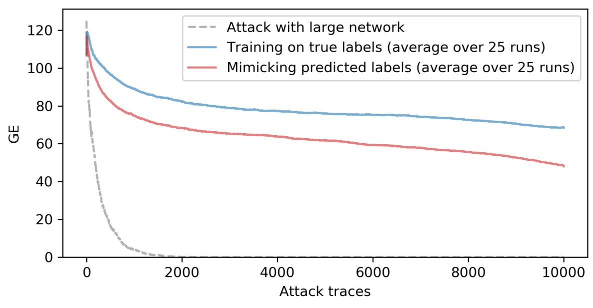
(b) Performance, averaged

Figure 6.3: **MLPr-1000**: MLP using 200 features, having one fully connected layer with 1000 neurons.

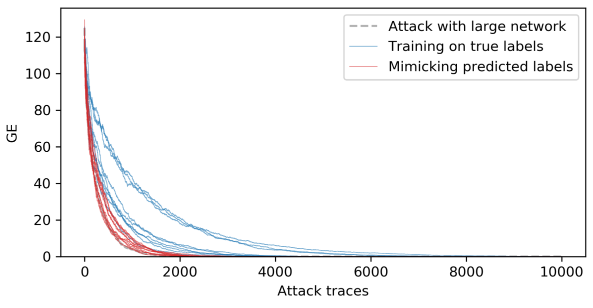
To paint a more realistic view on how the attacks perform, we have made a selection in the models: those with a GE higher than 50, after 5000 attack traces, are considered non-converging. They are deleted from the results in Figures 6.4c and 6.5c. The average of the converging networks (Subfigures 6.4d and 6.5d) shows a much more promising view of mimicking. Of the narrow CNNs that converge, all of them perform reasonable, but the mimicking networks consistently score better than the ones training on original data. The mimicking models perform almost as good as the teacher network.



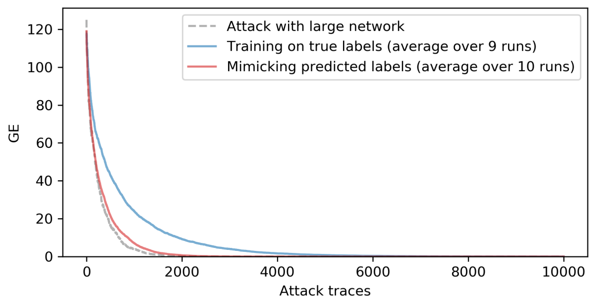
(a) Performance of all attacks



(b) Performance, averaged



(c) Performance of the attacks that converge

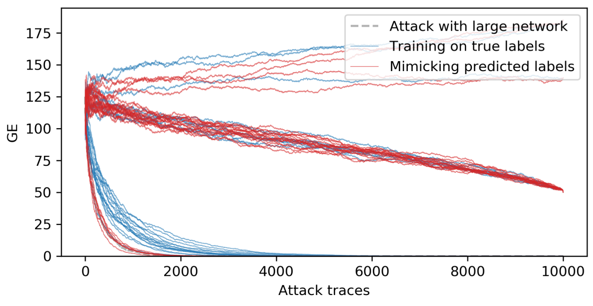


(d) Performance, averaged, of attacks that converge

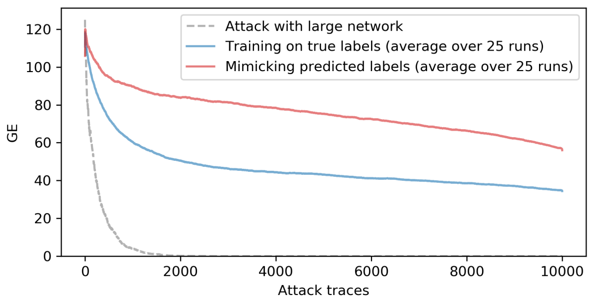
Figure 6.4: **CNN-8-3-500**: CNN using all 3500 features. It has one convolutional layer with 8 filters of size 3, followed by a fully connected layer with 500 neurons.

In conclusion, we demonstrated the potential of mimicking for attacks on the AES_HD dataset. An MLP with 500 neurons in a single hidden layer, using all features, performs consistently better when mimicking a teacher network, compared to original labels. It does so with roughly 14 times less parameters than the teacher network. For narrow MLPs that use only a selected of the features, there seems to be no added value from mimicking. Narrow CNNs that mimic may perform amazingly, approaching the teacher network's results, but are more sensitive to random initialization and have the risk of non-convergence.

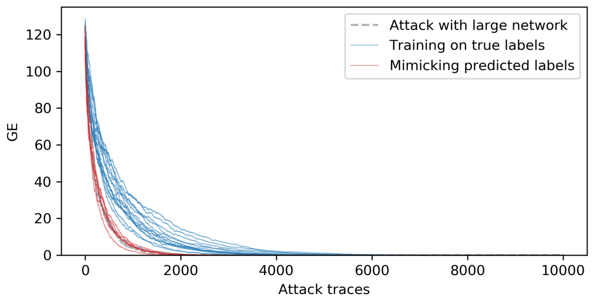
Mimicking may be a useful tool for model compression, where a simple model can be used after profiling. In SCA, this means an attacker could leave his heavy GPU at home after the profiling phase, and use more lightweight hardware to conduct the attack phase in the field. Alternatively, using the same hardware, the predictions of the attack set could be produced much faster.



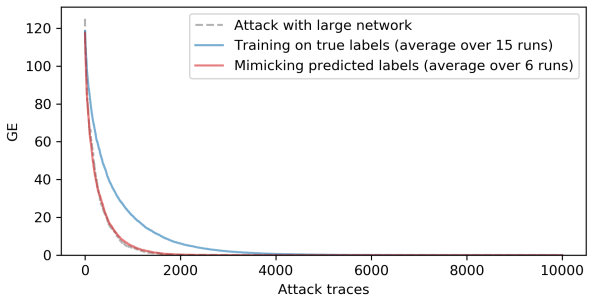
(a) Performance of all attacks



(b) Performance, averaged



(c) Performance of the attacks that converge



(d) Performance, averaged, of attacks that converge

Figure 6.5: **CNN-32-3-100**: CNN using all 3500 features. It has one convolutional layer with 32 filters of size 3, followed by a fully connected layer with 100 neurons.

Instead of evaluating only guessing entropy, attacks could also be ranked according to their efficiency – taking into account the required time to generate prediction of the attack set. Future work could include objective measurements on attack efficiency using limited hardware to make NN predictions. Finally, Ba and Caruana [77] proposed to introduce a small linear layer (no activation function) between the inputs and the fully connected layer; this decreases the number of parameters when the input dimensionality is high. As using a selected number of features in MLPr-100 and MLPr-1000 did not perform well, this technique may be a solution to achieve mimicking with much smaller networks.

7 Conclusions

Side-channel attacks (SCA) use unintended leakage, in order to extract sensitive information from a device. State of the art attacks commonly employ machine learning and deep learning techniques to profile some device, in order to attack a target device. However, there is a lack of understanding on why and how these techniques perform well, and in what settings. In this study, we have focused on profiled attacks on implementations of the Advanced Encryption Standard (AES).

In this chapter, we discuss the results from the previous parts of this report, guided by the research questions posed in Section 3.3. We make various suggestions for future research and summarize our contributions.

In Chapter 4, we have considered the bias–variance decomposition and how it can be used in profiled side-channel attacks.

RQ1*: Is there a relation between common loss functions used in machine and deep learning and the guessing entropy metric from SCA?*

A1: Yes, we showed that minimizing the mean squared error (MSE) is equivalent with minimizing the guessing entropy (GE), based on the fact that every key guess can be attributed in the MSE. This is very important since MSE is a commonly used loss function in machine and deep learning.

However, the categorical cross-entropy (CCE) is more popular choice in most deep learning classification tasks. Therefore, it would be interesting to connect the CCE with GE in a similar way.

Next, we conducted a detailed analysis where we consider different training set sizes, numbers of features, and model complexities. The results we obtained suggest that the bias-variance decomposition is able to provide significant insights, especially when the noise is not high and/or there are no countermeasures. In other, more difficult settings, the bias-variance decomposition gives only very general information on whether the model underfits or overfits but not on how to solve that problem, nor how many traces would be required to succeed in retrieving the secret key.

RQ2*: Can we change the bias-variance decomposition to reflect SCA metrics?*

A2: Yes. We propose the Guessing Entropy bias–variance decomposition, accounting for the GE and its deviation. As it is common in SCA to combine the predictions for multiple traces, this decomposition drops the individual attribution of a sample and looks at the entire dataset at the same time.

The Guessing Entropy bias–variance decomposition has provided us with more information on the behavior of machine learning-based SCA. Finally, we experimentally showed how this tool can be used to interpret attacks and discern potential problems in machine learning attacks.

RQ3*: When using machine and deep learning techniques, which parameters can be modified effectively to reduce bias and/or variance?*

A3: We have studied the effect of several conditions and its impact on the various classifiers. Essentially, adding more training data is typically helpful, especially for difficult datasets. Adding more features is not necessarily helpful: for each technique, the curse of dimensionality must be looked at carefully. In accordance to previous works, we see the pooled TA performs much better than naive TA when adding features. More interestingly, complexity must be carefully tuned for each dataset. While adding more layers certainly helps in attacks on the AES_RD dataset, for others complexity may not help (or even worsen) the attack. For random forest, we see increasing complexity is always helps.

In our experimental setting, the number of repetitions for the bootstrap algorithm plays an important role. If there are not enough repetitions, the variance will not be correctly estimated. At the same time, using deep learning techniques can be computationally demanding even if we need to run them only once. In the setting where we need multiple repetitions, it is not easy to decide the trade-off between the investment in the complexity of the machine learning models and the number of experimental repetitions. In our future work, we plan to investigate the influence of the bootstrap parameters on the reliability of the results. Here, we are especially interested in investigating whether datasets with countermeasures need more repetitions.

We continued in Chapter 5 with a stronger focus on deep learning techniques. Using SVCCA, a recently proposed method to compare neural network layers, we have analyzed the difference between networks used for SCA.

RQ4: *In which settings can we use SVCCA be used to compare NNs and what can we learn from its application in SCA?*

A4: We have observed that networks trained for near-identical problems have similar inner representations, including comparisons where one network is trained for HW labels and the other for intermediate values. However, when comparing among different leakage models (i.e., unmasking or ignoring masks), models in SCA have not more in common than models for entirely different problems (e.g., the CIFAR-10 image database). This shows that different attacks require different neural networks: learning on a simple dataset and attack a (different) difficult dataset will not work. Although this result is somewhat disappointing from the attacker's point of view, we consider this a contribution.

RQ5: *Do neural networks trained for similar data, using different devices and/or keys, learn the same representation?*

A5: There certainly is some common internal representation for models that were trained on very similar data. In a setting were data has the same shape and the leakage model remains unchanges, we observe common knowledge between all networks, including those trained on measurement from a different device and/or key.

RQ6: *How can we effectively "port" a neural network to handle different devices?*

A6: Recent publications suggest that training on multiple devices helps to get a more generalized the model [7–9]. Based on our findings, it is important to avoid overtraining by adding too much training samples. This is in accordance to recent related work [7]. Finally, we show that a model's specialism is mostly developed in its middle layers.

Although the SVCCA method is not suitable for comparing NNs trained on dissimilar datasets, it can shed a light on the mysteries on NNs' inner representations. In future work, we would like to explore the similarity of convolutional outputs in side-channel attacks.

RQ7: *Can SCA functions learned by large neural networks be mimicked by smaller networks?*

Yes, we have demonstrated this in Chapter 6. Attacks on AES_HD with small networks can benefit from mimicking a large teacher CNN's outputs. For a small MLP with a single layer of 500 neurons, models that mimic consistently perform better than their counterparts trained on regular labels. When using a narrow CNN, it is more difficult to find a configuration where the models always converge. For converging models, we have observed that narrow CNNs mimicking a large teacher CNN perform outstanding and even approach the performance of the teacher network.

In future work, a more thorough evaluation of (mimicking) networks could include the runtime of the attack set predictions. Also, techniques like an extra linear layer after the inputs may speed up mimicking even further [77].

7.1. Summary of Contributions

The main contributions of this work are the following:

- We have shown that minimizing the mean squared error (MSE) is equivalent to minimizing the Guessing Entropy (GE);
- We propose the Guessing Entropy bias–variance decomposition, a tool to analyze the characteristics of classifiers used in SCA;
- We apply the Guessing Entropy bias–variance decomposition to various public datasets and show the influence of the number of training samples, number of features, and the model complexity, on the attack performance;
- To the best our knowledge, we are the first to establish a baseline for SVCCA based on inter-dataset comparisons and different class types;
- We have shown the effect of overtraining and optimizing in the portability context, where specialization is shown to occur mostly in the middle of the network;
- To the best of our knowledge, we are the first to demonstrate mimicking in SCA. We use mimicking to improve attacks on AES_RD with small architectures.

Bibliography

[1] L. Malina, J. Hajny, R. Fujdiak, and J. Hosek, *On perspective of security and privacy-preserving solutions in the internet of things,* Computer Networks **102**, 83 (2016).

[2] NIST, *Advanced Encryption Standard (AES),* (2001).

[3] L. Tawalbeh, T. F. Al-Somani, A. Tawalbeh, a. A. Tawalbeh, and H. Houssain, *Review of Side Channel Attacks and Countermeasures on ECC, RSA, and AES Cryptosystems,* Journal of Internet Technology and Secured Transactions (JITST) **5**, 515 (2016).

[4] P. Kocher, J. Jaffe, and B. Jun, *Differential Power Analysis,* in *Advances in Cryptology — CRYPTO' 99*, edited by M. Wiener (Springer Berlin Heidelberg, Berlin, Heidelberg, 1999) pp. 388–397.

[5] S. Chari, J. R. Rao, and P. Rohatgi, *Template Attacks,* in *Cryptographic Hardware and Embedded Systems - CHES 2002* (Springer, 2002) pp. 12–28.

[6] B. Hettwer, S. Gehrer, and T. Güneysu, *Applications of machine learning techniques in side-channel attacks: a survey,* Journal of Cryptographic Engineering (2019).

[7] S. Bhasin, D. Jap, A. Chattopadhyay, S. Picek, A. Heuser, and R. Ranjan Shrivastwa, *Mind the Portability: A Warriors Guide through Realistic Profiled Side-channel Analysis,* CoRR (2019).

[8] D. Das, A. Golder, J. Danial, S. Ghosh, A. Raychowdhury, and S. Sen, *X-DeepSCA: Cross-Device Deep Learning Side Channel Attack,* in *Proceedings of the 56th Annual Design Automation Conference 2019 on - DAC '19*, Vol. 1 (ACM Press, New York, New York, USA, 2019) pp. 1–6.

[9] A. Golder, D. Das, J. Danial, S. Ghosh, S. Sen, and A. Raychowdhury, *Practical Approaches Towards Deep-Learning Based Cross-Device Power Side Channel Attack,* arXiv e-prints (2019), arXiv:arXiv:1907.02674 .

[10] P. P. Shinde and S. Shah, *A Review of Machine Learning and Deep Learning Applications,* Proceedings - 2018 4th International Conference on Computing, Communication Control and Automation, ICCUBEA 2018 , 1 (2019).

[11] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, *A Survey of Methods for Explaining Black Box Models,* ACM Computing Surveys **51**, 1 (2018).

[12] L. Breiman, *Random Forests,* Machine Learning **45**, 5 (2001).

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, *Scikit-learn: Machine Learning in Python,* Journal of Machine Learning Research **12**, 2825 (2011).

[14] S. Picek, A. Heuser, and S. Guilley, *Template attack versus Bayes classifier,* Journal of Cryptographic Engineering **7**, 343 (2017).

[15] J. Pan, F. Zhang, K. Ren, and S. Bhasin, *One Fault is All it Needs: Breaking Higher-Order Masking with Persistent Fault Analysis,* in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (IEEE, 2019) pp. 1–6.

[16] H. Zhang, *The Optimality of Naive Bayes,* in *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2004*, Vol. 2 (2004) pp. 562–567.

[17] O. Choudary and M. G. Kuhn, *Efficient Template Attacks,* in *Smart Card Research and Advanced Applications*, edited by A. Francillon and P. Rohatgi (Springer International Publishing, 2014) pp. 253–270.

[18] E. Cagli, C. Dumas, and E. Prouff, *Convolutional neural networks with data augmentation against jitter-based countermeasures: Profiling attacks without pre-processing,* Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **10529 LNCS**, 45 (2017).

[19] S. Theodoridis and K. Koutroumbas, *Pattern Recognition* (Elsevier, 2009).

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016).

[21] D. H. Wolpert, *The Lack of a Priori Distinctions between Learning Algorithms,* Neural Computation **8**, 1341 (1996).

[22] B. Efron, *Bootstrap Methods: Another Look at the Jackknife,* Ann. Statist. **7**, 1 (1979).

[23] R. E. Bellman, *Dynamic Programming* (Dover Publications, Incorporated, 2003).

[24] V. N. Vapnik and A. Y. Chervonenkis, *On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities,* in *Measures of Complexity* (Springer International Publishing, Cham, 2015) pp. 11–30.

[25] M. A. Carter and M. E. Oxley, *Evaluating the Vapnik–Chervonenkis dimension of artificial neural networks using the Poincaré polynomial,* Neural Networks **12**, 403 (1999).

[26] A. K. Ghosh and P. Chaudhuri, *On Maximum Depth and Related Classifiers,* Scandinavian Journal of Statistics **32**, 327 (2005).

[27] M. Claesen and B. De Moor, *Hyperparameter Search in Machine Learning,* in *The XI Metaheuristics International Conference* (OALib, 2015) pp. 10–14, arXiv:1502.02127 .

[28] P. Domingos, *A Unified Bias-Variance Decomposition and its Applications,* Science , 231 (2000).

[29] P. Domingos, *A Unified Bias-Variance Decomposition for Zero-One and Squared Loss,* Seventeenth National Conference on Artificial Intelligence , 564 (2000).

[30] H. Robbins and S. Monro, *A Stochastic Approximation Method,* The Annals of Mathematical Statistics **22**, 400 (1951).

[31] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization,* International Conference for Learning Representations , 1 (2014), arXiv:1412.6980 .

[32] S. Ruder, *An overview of gradient descent optimization algorithms,* CoRR **abs/1609.0** (2016), arXiv:1609.04747 .

[33] R. Caruana and S. Lawrence, *Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping,* Advances in neural information processing systems. , 402 (2001).

[34] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Improving neural networks by preventing co-adaptation of feature detectors,* CoRR **abs/1207.0** (2012), arXiv:1207.0580 .

[35] 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,* Cryptology ePrint Archive: Report 2018/1023 , 1 (2018).

[36] A. Y. Ng, *Feature selection, L1 vs. L2 regularization, and rotational invariance,* in *Twenty-first international conference on Machine learning - ICML '04* (ACM Press, New York, New York, USA, 2004) p. 78.

[37] J. Sietsma and R. J. Dow, *Creating artificial neural networks that generalize,* Neural Networks **4**, 67 (1991).

[38] S. Rifai, X. Glorot, Y. Bengio, and P. Vincent, *Adding noise to the input of a model trained with a regularized objective,* CoRR **abs/1104.3** (2011), arXiv:1104.3250 .

[39] C. M. Bishop, *Training with Noise is Equivalent to Tikhonov Regularization,* Neural Computation **7**, 108 (1995).

[40] D. Scherer, A. Müller, and S. Behnke, *Evaluation of pooling operations in convolutional architectures for object recognition,* in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 6354 LNCS (2010) pp. 92–101.

[41] K. Simonyan and A. Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition,* in *3rd International Conference on Learning Representations, {ICLR} 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015).

[42] S. Ioffe and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,* in *Proceedings of the 32nd International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 37, edited by F. Bach and D. Blei (PMLR, Lille, France, 2015) pp. 448–456.

[43] M. Vogt, *An Overview of Deep Learning and Its Applications,* in *Fahrerassistenzsysteme 2018*, edited by T. Bertram (Springer Fachmedien Wiesbaden, Wiesbaden, 2019) pp. 178–202.

[44] T. Kim, J. Lee, and J. Nam, *Sample-Level CNN Architectures for Music Auto-Tagging Using Raw Waveforms,* ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings **2018-April**, 366 (2018), arXiv:arXiv:1710.10451v2 .

[45] R. L. Rivest, *Cryptography,* in *Handbook of Theoretical Computer Science*, edited by J. van Leeuwen (Elsevier, Amsterdam, 1990) algorithms ed., Chap. 13, pp. 717–755.

[46] B. Tao and H. Wu, *Improving the Biclique Cryptanalysis of AES,* in *Information Security and Privacy*, edited by E. Foo and D. Stebila (Springer International Publishing, Cham, 2015) pp. 39–56.

[47] J. Daemen and V. Rijmen, *AES Proposal: Rijndeal*, Tech. Rep. (National Institute of Standards and Technology, 2003).

[48] T.-H. Le, C. Canovas, and J. Clédière, *An overview of side channel analysis attacks,* in *Proceedings of the 2008 ACM symposium on Information, computer and communications security - ASIACCS '08* (ACM Press, New York, New York, USA, 2008) p. 33.

[49] K. Gandolfi, C. Mourtel, and F. Olivier, *Electromagnetic Analysis: Concrete Results,* in *Cryptographic Hardware and Embedded Systems - CHES* (2001) pp. 251–261.

[50] J.-J. Quisquater and D. Samyde, *ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards,* in *Proceedings of the International Conference on Research in Smart Cards: Smart Card Programming and Security*, E-SMART '01 (Springer-Verlag, London, UK, UK, 2001) pp. 200–210.

[51] H. Maghrebi, T. Portigliatti, and E. Prouff, *Breaking cryptographic implementations using deep learning techniques,* in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10076 LNCS (2016) pp. 3–26.

[52] E. Prouff, R. Strullu, R. Benadjila, E. Cagli, and C. Dumas, *Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database,* IACR Cryptology ePrint Archive **2018**, 53 (2018).

[53] 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**, 209 (2018).

[54] F.-X. Standaert, T. Malkin, and M. Yung, *A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks,* in *EUROCRYPT*, LNCS, Vol. 5479 (Springer, 2009) pp. 443–461.

[55] F.-X. Standaert, E. Peeters, and J.-J. Quisquater, *On the masking countermeasure and higher-order power analysis attacks,* in *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II* (IEEE, 2005) pp. 562–567 Vol. 1.

[56] TELECOM ParisTech SEN research group, *DPA Contest (4th edition),* (2011).

[57] A. Moradi, S. Guilley, and A. Heuser, *Detecting Hidden Leakages,* in *ACNS*, Vol. 8479, edited by I. Boureanu, P. Owesarski, and S. Vaudenay (Springer, 2014).

[58] J.-S. Coron and I. Kizhvatov, *An Efficient Method for Random Delay Generation in Embedded Software,* in *Cryptographic Hardware and Embedded Systems - {CHES} 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings* (2009) pp. 156–170.

[59] M. A. Elaabid and S. Guilley, *Portability of templates,* Journal of Cryptographic Engineering **2**, 63 (2012).

[60] L. Lerman, R. Poussier, G. Bontempi, O. Markowitch, and F.-X. Standaert, *Template Attacks vs. Machine Learning Revisited (and the Curse of Dimensionality in Side-Channel Analysis),* in *COSADE 2015, Revised Selected Papers* (2015) pp. 20–33.

[61] A. Heuser, S. Picek, S. Guilley, and N. Mentens, *Side-Channel Analysis of Lightweight Ciphers: Does Lightweight Equal Easy?* in *12th International Workshop, on Radio Frequency Identification and IoT Security, Revised Selected Papers* (2016) pp. 91–104.

[62] A. Heuser and M. Zohner, *Intelligent Machine Homicide - Breaking Cryptographic Devices Using Support Vector Machines,* in *COSADE* (2012) pp. 249–264.

[63] S. Picek, A. Heuser, A. Jovic, S. A. Ludwig, S. Guilley, D. Jakobovic, and N. Mentens, *Side-channel analysis and machine learning: A practical perspective,* in *2017 International Joint Conference on Neural Networks (IJCNN)* (2017) pp. 4095–4102.

[64] A. Heuser, S. Picek, S. Guilley, and N. Mentens, *Lightweight Ciphers and their Side-channel Resilience,* IEEE Transactions on Computers **PP**, 1 (2017).

[65] S. Picek, A. Heuser, C. Alippi, and F. Regazzoni, *When Theory Meets Practice : A Framework for Robust Profiled Side-channel Analysis,* CoRR , 1 (2018).

[66] L. Lerman, G. Bontempi, and O. Markowitch, *The bias–variance decomposition in profiled attacks,* Journal of Cryptographic Engineering **5**, 255 (2015).

[67] L. Lerman, N. Veshchikov, O. Markowitch, and F. X. Standaert, *Start Simple and then Refine: Bias-Variance Decomposition as a Diagnosis Tool for Leakage Profiling,* IEEE Transactions on Computers **67**, 268 (2018).

[68] B. Hettwer, S. Gehrer, and G. Tim, *Deep Neural Network Attribution Methods for Leakage Analysis and Symmetric Key Recovery,* CoRR , 1 (2019).

[69] K. Simonyan, A. Vedaldi, and A. Zisserman, *Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps,* CoRR , 1 (2013), arXiv:1312.6034 .

[70] F. Klauschen, K.-R. Müller, A. Binder, G. Montavon, W. Samek, and S. Bach, *On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation,* Plos One **10**, e0130140 (2015), arXiv:1606.04155 .

[71] M. D. Zeiler and R. Fergus, *Visualizing and understanding convolutional networks,* Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **8689 LNCS**, 818 (2014), arXiv:1311.2901 .

[72] M. Raghu, J. Gilmer, J. Yosinski, and J. Sohl-Dickstein, *SVCCA: Singular Vector Canonical Correlation Analysis for Deep Learning Dynamics and Interpretability,* in *Advances in Neural Information Processing Systems 30*, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Curran Associates, Inc., 2017) pp. 6076–6085.

[73] A. S. Morcos, M. Raghu, and S. Bengio, *Insights on representational similarity in neural networks with canonical correlation,* in *32nd Conference on Neural Information Processing Systems (NIPS 2018)*, Nips (Montréal, 2018) arXiv:1806.05759 .

[74] T. Yu, H. Long, and J. E. Hopcroft, *Curvature-based Comparison of Two Neural Networks,* CoRR (2018), arXiv:1801.06801 .

[75] G. Cybenko, *Approximation by superpositions of a sigmoidal function,* Mathematics of Control, Signals, and Systems **2**, 303 (1989).

[76] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang, *The Expressive Power of Neural Networks: A View from the Width,* in *Advances in Neural Information Processing Systems 30*, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Curran Associates, Inc., 2017) pp. 6231–6239.

[77] J. Ba and R. Caruana, *Do Deep Nets Really Need to be Deep?* in *Advances in Neural Information Processing Systems 27*, edited by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Curran Associates, Inc., 2014) pp. 2654–2662.

[78] G. Urban, K. J. Geras, S. E. Kahou, O. Aslan, S. Wang, R. Caruana, A. Mohamed, M. Philipose, and M. Richardson, *Do Deep Convolutional Nets Really Need to be Deep (Or Even Convolutional)?* ICLR (2016), 10.1038/nature14539, arXiv:1603.05691 .

[79] G. James, D. Witten, T. Hastie, and R. Tibsihrani, *An Introduction to Statistical Learning*, Springer Texts in Statistics (Springer New York Heidelbert Dordrecht London, 2001).

A
List of Symbols

$(\mathbf{x}, y)$	combination of feature vector (trace) $\mathbf{x}$ and corresponding label y, based on the leakage model. When needed, including index: $(\mathbf{x_1}, y_1)$
$\hat{y}$	a prediction for y, based on x, by some classifier
n	number of points of interest (features) in a trace, so $\mathbf{x} = (x_1, \dots, x_n)$
T_p	set of profiling traces
N	length of profiling set
k_p^*	key of profiling set
T_a	set of attack traces
k_a^*	key of attack set
M	length of attack set T_a
$y = f(\mathbf{x})$	True mapping between between label y and $\mathbf{x}$.
$\hat{y} = \hat{f}(\mathbf{x})$	Attack model, computing a prediction $\hat{y}$ for a sample $\mathbf{x}$
$E_z[]$	expected value over z
p_z	probability over z
$Y(T_p, k_p^*)$	Leakage model, based on profiling traces and profiling key
$Y^{-1}(PT, \hat{y})$	Inverse leakage model: get a key guess based on plaintext and leakage class
$\mathcal{K}$	key space (length $\lvert\mathcal{K}\rvert$)
$\mathbf{g}$	key guessing vector, $\mathbf{g} = [g_0, g_1, \dots g_{\lvert\mathcal{K}\rvert-1}]$
g_i	estimated probability for key candidate i, $g_i = \sum_{j=1}^{T_a} \log(\hat{p}_{ij})$
$\hat{p}_{ij}$	estimated probability for key candidate i using sample j

B

List of Abbreviations

Abbreviation	Explained in	Full term
AES	2.3.1	Advanced encryption standard
CCE	2.1.3	Categorical cross-entropy
CNN	2.2.4	Convolutional neural network
DL	2.2	Deep learning
DNN	3	Deep neural network
DPA	2.4	Differential power analysis
EM	2.4	Electromagnetic emission
GE	2.4.1	Guessing entropy
HW	2.4.1	Hamming weight
IoT	1	Internet of Things
LDA	2.1.1	Linear discriminant analysis
ML	2.1	Machine learning
MLP	2.2	Multi-layer perceptron
MSE	2.1.3	Mean squared error
NB	2.1.1	Naive Bayes
NN	2.2	Neural network
PoI	2.1	Point of interest
QDA	2.1.1	Quadratic discriminant analysis
RF	2.1.1	Random forest
SCA	2.4	Side-channel analysis, or side-channel attack
SPA	2.4	Simple power analysis
SVCCA	3.2.1	Singular vector canonical correlation analysis
TA	2.4.1	Template attack
VGG	2.2.5	A neural network design, named after the Visual Geometry Group of the University of Oxford

C
Implementation Details

The experiments were run using Python 3.6.8. For the machine learning tasks (i.e., non-deep learning) we have used Scikit-learn 0.20.3. The deep learning experiments were conducted with Keras 2.2.4 using the Tensorflow 1.13.1 GPU-based backend.

Neural networks were trained on TU Delft's High Performance Computing (HPC) cluster, which runs Linux CentOS 7. The cluster nodes are equipped with GTX 1080 Ti graphic cards, which has 11 GB of memory and 3584 processing cores. Tensorflow-gpu 1.13.1 is build against CUDA 10.0, which employs the CUDA Deep Neural Network library (cuDNN) which is optimized for deep learning operations such as forward and backward convolution, pooling, normalization, and activation.