Roma Tre

Dottorato di ricerca in Matematica

XXXVIII Ciclo

Automation of Differential Cryptanalysis through Constraint Programming and Machine Learning

Candidato:

Rocco Brunelli

Supervisore:

Prof. Marco Pedicini

Co-Supervisore:

Prof. Loïc Rouquette

Coordinatore:

Prof. Alessandro Giuliani

Anno Accademico 2025/2026

Abstract

The security evaluation of block ciphers relies on cryptanalytic techniques that are increasingly difficult to apply manually as designs grow in complexity and the number of rounds increases. This motivates the development of automatic and systematic tools that can explore large cryptanalytic search spaces, reduce human bias, and provide meaningful security bounds. This thesis investigates two complementary approaches to the automation of cryptanalysis.

First, it focuses on the automation of differential cryptanalysis and its extension to advanced attack models based on this cryptanalysis through Constraint Programming (CP). We extend the Tagada, a CP framework, originally designed for differential attacks, to support boomerang and sandwich-style distinguishers. By modeling the composition of upper and lower differential trails in a unified optimization framework, we enable the automatic construction and evaluation of boomerang distinguishers, reaching the state-of-art results and discovering new ones in a more general approach. This contributes toward a more systematic and tool-assisted assessment of differential security.

Second, this thesis studies the automation of cryptanalysis through machine learning. Due an unbalanced distribution, Neural distinguishers have shown strong empirical performance in distinguishing reduced-round block ciphers from random permutations, but they operate largely as black boxes and rely on implicit feature extraction. To make this process more systematic and interpretable, we introduce a generic feature-engineering technique based on partial decryption, which injects structural information about the cipher into neural distinguishers. This approach improves performance, enhances transparency, and clarifies the link between learned representations and classical differential properties.

Overall, this work shows how optimization-based methods and machine-learning-based methods can be used as complementary tools for the automated analysis of block cipher security, helping bridge the gap between classical and data-driven cryptanalysis.

Acknowledgments

It is my pleasure to thank everyone who helped make this thesis possible. First and foremost, I would like to thank my advisors, Marco Pedicini and Loïc Rouquette, for their constant support throughout my PhD studies.

Marco, when I started this journey, I knew nothing about cryptography. Thanks to you, I had the opportunity to work on a different and, surprisingly, very fascinating side of mathematics. You gave me the chance to travel and meet inspiring people from all over the world. I was a bit lost in the first period, thanks your suggestions and your patience I could continue to work and find new interesting challenges. Your support during all this period is something I will take care. I have said it before and I say it again: thank you.

Loïc, after your talk at my university, I started asking you many questions, and you always took the time to reply. That was very kind of you, and I truly appreciated your continuous support and your wise advice. Sorry (but not sorry) if I bothered you. Thanks to let me work with you and to give me new opportunities.

A special thanks goes to the CEO Emiliano Betti and Professor Marco Cesati for providing the funding for my scholarship and for supporting my research. I would also like to thank the whole Epigenesys team for welcoming me into their daily work environment. In particular, Alessio and Giulio, thank you for making everything easier and for giving me the support I needed.

I would like to thank Professor Marine Minier for giving me the opportunity to join the Inria laboratory in Nancy during my visiting period. During this time, I was able to work on the results presented in the first part of this thesis.

I am also grateful to Emanuele Bellini for proposing that I work on the project that led to my first publication and to the results presented in the second part of this thesis. Thank you also for continuing to involve me in new challenges and projects. I also thank the TII team with whom I collaborated, in particular Anna Hambitzer and Juan Grados.

A special thanks to David Gerault. David, you carefully followed this thesis and were always available to guide me whenever I had doubts, both about the project and about my future decisions.

Finally, thanks to the XXXVIII Mathematics PhD class: Simone, Martina, Lorenzo, and Mario. Thank you for sharing with me both uncertainties and daily life in the office, and for brightening my days with small talks. I would also like to thank the always-present PhD students—Ilaria, Michele B., Nicoletta, Elisa, and Wei and to all the PhD students, PostDoc and researcher I met during my journey: Dionisis, Kostas, Ioannis, Fabio, Anna, Thierno, Leonardo, Lorenzo and Enrico.

Grazie Mamma, Papà, Elisa, gli amici di Piazza Bologna, gli amici della sezione I e Maddalena, anche se non lo mostro ogni giorno, vi sono e sarò sempre grato del silenzioso supporto che mi date.

Rocco Brunelli
2025/2026

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

In the digital era, cryptography has become an indispensable component of modern information systems. Secure communications, electronic commerce, cloud services, and critical infrastructures all rely on cryptographic algorithms to guarantee confidentiality, integrity, and authenticity of data. The pervasive use of digital technologies has made the security of cryptographic primitives a fundamental requirement for both economic and societal stability.

Among the different branches of cryptography, symmetric-key cryptography plays a central role. Thanks to its high efficiency, it is the preferred solution for the encryption of large volumes of data and constitutes the core of many cryptographic protocols. Despite its apparent simplicity, the design of secure symmetric primitives remains a challenging task, as even minor structural weaknesses can lead to severe vulnerabilities.

For this reason, the rigorous analysis of symmetric algorithms and the development of systematic cryptanalytic techniques are essential steps in the assessment of modern cryptographic systems.

Motivation and Context

Block ciphers are a fundamental building block of modern cryptography and play a central role in the protection of digital communications, storage systems, and critical infrastructures. Their security is not only a theoretical requirement but a practical necessity, as weaknesses in widely deployed primitives can lead to severe real-world consequences. For this reason, the design and evaluation of symmetric-key algorithms has been, and continues to be, an active area of research in cryptography.

Among the many techniques developed to analyze the security of block ciphers, *differential cryptanalysis* has established itself as one of the most powerful and influential. Since its introduction, it has deeply shaped both the design principles of modern ciphers and the methodology used to assess their resistance against statistical attacks. Differential cryptanalysis studies how differences in the input propagate through the cipher and how these differences affect the distribution of output differences. By identifying high-probability differential characteristics, an attacker can distinguish a cipher from a random permutation or, in some cases, recover secret key material.

However, as cipher designs have grown in complexity and the number of rounds has increased, the manual construction and analysis of differential characteristics has become increasingly difficult. The search space of possible trails grows exponentially with the number of rounds, and even expert-driven analysis becomes error-prone and time-consuming. This has motivated the development of automatic tools and optimization-based approaches that can systematically explore the space of characteristics and provide provable guarantees on their optimality.

At the same time, recent advances in machine learning and artificial intelligence have opened new perspectives in cryptanalysis. Neural networks have been successfully applied to distinguish reduced-round versions of block ciphers from random permutations, giving rise to the emerging field of *neural cryptanalysis*. These approaches challenge traditional assumptions and raise new questions about the nature of distinguishers, feature extraction, and the interplay between classical and data-driven techniques.

This thesis lies at the intersection of these research directions. Its goal is to investigate automatic methods for differential cryptanalysis, extend them to advanced attack models such as the boomerang attack, and explore their interaction with modern neural distinguishers. By combining formal optimization techniques with machine learning-based approaches, we aim to provide a unified framework for the systematic analysis of symmetric ciphers.

Part I: Automatization through Constraint Programming

The starting point of this work is a detailed study of iterated block ciphers and the principles of differential cryptanalysis, presented in Chapter 2. We review the mathematical foundations of differences, differential probabilities, and characteristics, and discuss how these concepts are used to evaluate the resistance of a cipher against statistical attacks. Particular attention is devoted to the notion of optimal differential trails and to the challenges posed by large design spaces.

Building on this foundation, Chapter 3 formulates the search for optimal differential characteristics as an optimization problem. By modeling the propagation of differences through the cipher as a set of constraints, the problem can be expressed in terms of constraint programming (CP), or related formal frameworks. This approach allows the use of powerful off-the-shelf solvers to automatically find characteristics with minimal weight or maximal probability.

The use of automatic tools provides several advantages. First, it reduces the reliance on heuristic, manual analysis and enables systematic exploration of large search spaces. Second, it allows the comparison of different designs under a unified methodology. Finally, it offers formal guarantees on the optimality of the differential characteristics, which is crucial when drawing security conclusions.

Nevertheless, the automatic search for differential characteristics also presents significant challenges. The size of the resulting optimization problems can be enormous, and naive formulations quickly become infeasible. A central theme of this thesis is therefore the design of efficient encodings and the integration of domain-specific knowledge into general-purpose solvers.

While classical differential cryptanalysis focuses on single trails spanning the entire cipher, more advanced attack models consider the composition of multiple characteristics. One of the most prominent examples is the *boomerang attack*, which combines two short differential trails to attack a larger number of rounds.

Chapter 4 presents one of the main contributions of this thesis: the integration of the boomerang attack into an automatic search framework. We extend an existing tool, Tagada, to support the systematic construction and evaluation of boomerang distinguishers. This requires a careful modeling of the interaction between the upper and lower trails, as well as the constraints imposed by their composition.

The automatic treatment of boomerang attacks significantly increases the expressive power of the framework. It allows the exploration of attack scenarios that would be extremely difficult to analyze manually and provides a unified view of classical differential and boomerang-style attacks. Moreover, it highlights the importance of tool-supported analysis in modern cryptography, where the complexity of both designs and attacks continues to grow.

Part II: Automatization through Machine Learning

In the second part of the thesis, we shift our focus to the emerging field of neural cryptanalysis. Chapter 6 reviews the seminal work on neural distinguishers and discusses how deep learning techniques can be used to detect subtle statistical deviations in

cipher outputs. Unlike classical distinguishers, which rely on explicitly constructed features such as differences or linear masks, neural networks can learn complex, non-linear representations directly from data.

This raises several fundamental questions. What kind of information do neural distinguishers exploit? How do they relate to classical differential characteristics? And can traditional cryptanalytic insights be used to guide and improve data-driven approaches?

To address these questions, Chapter 7 introduces a novel feature engineering technique based on *generic partial decryption*. The idea is to use partial knowledge of the cipher structure to transform raw data into more informative representations before feeding it to a neural network. This approach bridges the gap between classical and neural cryptanalysis by explicitly injecting domain knowledge into the learning process.

Our results show that carefully designed features can significantly improve the performance and interpretability of neural distinguishers. At the same time, they provide new insights into the relationship between differential properties and learned representations.

Contributions and Organization of the Thesis

The main contributions of this thesis can be summarized as follows:

- A systematic formulation of differential characteristic search as an optimization problem, together with efficient encodings suitable for automatic solvers.
- The extension of an automatic framework to support boomerang attacks, enabling the unified analysis of advanced differential techniques.
- An in-depth study of neural cryptanalysis and the proposal of generic partial decryption as a feature engineering method for neural distinguishers.
- An experimental evaluation that highlights the connections between classical cryptanalysis and modern machine learning-based approaches.

The remainder of the thesis is organized as follows. Chapter 2 introduces iterated ciphers and the principles of differential cryptanalysis. Chapter 3 presents the optimization-based formulation and the use of automatic tools. Chapter 4 extends this framework to boomerang attacks. Chapter 6 reviews neural cryptanalysis, and Chapter 7 introduces generic partial decryption as a bridge between classical and neural methods.

By combining formal optimization techniques with data-driven approaches, this thesis aims to contribute to a deeper understanding of modern cryptanalysis and to provide practical tools for the systematic evaluation of symmetric cryptographic primitives.

Publications and Collaborations.

- The results presented in Chapter 4 are currently under submission. This is a joint work with professor Marine Minier, Université de Lorraine, CNRS, Inria, LORIA, Nancy, France, and professor Loïc Rouquette, EPITA Research Laboratory (LRE), 4/16 rue Voltaire, Le Kremlin-Bicêtre F-94270, France. I started this work during my visiting period at the laboratories of Inria, in Nancy and finished once returned in Rome.
- The results presented in Chapter 7 have been published [BBG+25]. This is a joint work with professor Marco Pedicini, Università di Roma Tre, and Emanuele Bellini, David Gerault, Anna Hambitzer, Technology Innovation Institute, Abu Dhabi, UAE.

Notation

During the thesis we use the following notation:

Symbol	Description
$\oplus$	is the bit-wise XOR
$\cdot \lll \alpha$	is the left rotation of the vector $\cdot$ of α position
$\cdot \ggg \alpha$	is the right rotation of the vector $\cdot$ of α position
$\odot$	is the bit-wise AND
$\wedge$	is the AND between two bits
$\|\|$	is the concatenation of two vectors
$\neg x$	is the negation of x, with $x \in \{0, 1\}$
$\boxplus_n$	is the modular addition modulo 2^n, if n is clear from the contest it is omitted
$\boxminus_n$	is the modular subtraction modulo 2^n, if n is clear from the contest it is omitted
$[a, b]$	is the set of the integer between a and b, $\{x \in \mathbb{Z} : a \leq x \leq b\}$
$[a, b]$	is the set of the integer between a and b, $\{x \in \mathbb{Z} : a < x \leq b\}$
$\mathbb{F}_2^n$	is a Galois Field of characteristic 2 with n elements, e.g. $\mathbb{Z}_2^n \simeq \{0, 1\}^n$
$\|\| \cdot \|\|_p$	is the L_p norm of a vector, $\|\|v\|\|_p = (\sum_i \|v_i\|^p)^{\frac{1}{p}}$
$\mathbb{P}_{\mathcal{X}}$	is the probability distribution over the set $\mathcal{X}$, if the set is clear from the contest it is omitted
$\mathbb{P}(A\|B)$	is the condition probability distribution of A given the event B

A little remark on the way we write the integers. Let's take an example. Let $w \in [0, 2^4 - 1]$, be $w = 11$. Then, the **hexadecimal representation** is:

```
0xb
```

The **bit representation**, where for bit we mean a value that can be or 0 or 1, instead is:

$$(1, 1, 0, 1) \in \{0, 1\}^4$$

where the first bit is the *most significant bit*. We call a 4-bit string a **nibble** and a 8-bit string a **byte**.

Part I

Automatization through Constraint Programming

Chapter 2

Iterated Ciphers and Differential Cryptanalysis

The design and security analysis of symmetric cryptographic primitives constitute a cornerstone of modern information security. Symmetric-key cryptography represents one of the fundamental pillars of contemporary cryptographic systems: in this setting, the same secret key is shared between the communicating parties and is used for both encryption and decryption. Owing to their efficiency and simplicity of implementation, symmetric algorithms are the primary choice for protecting large amounts of data in practical applications [MvOV96].

Symmetric ciphers are commonly divided into two main families: stream ciphers and block ciphers. Stream ciphers generate a pseudorandom keystream that is combined with the plaintext, typically by a bitwise exclusive-or operation. Block ciphers, on the other hand, operate on fixed-size blocks of data and transform each plaintext block into a ciphertext block of the same length under the control of a secret key. Due to their versatility and robustness, block ciphers have become the dominant primitive in modern symmetric cryptography.

A way to build modern block ciphers is like *iterated ciphers*. In this design paradigm, the encryption function is obtained by the repeated application of a simple round function. Each round performs a structured transformation of the internal state, and the security of the cipher relies on the cumulative effect of many rounds rather than on the complexity of a single operation [Sti06]. This modular approach enables designers to reason about security in a compositional manner and to construct ciphers whose strength emerges from the interaction of simple components.

The conceptual foundations of iterated constructions can be traced back to Claude Shannon's seminal work [Sha49], in which he introduced the principles of *confusion* and *diffusion*. Confusion aims at obscuring the relationship between the secret key and the ciphertext, while diffusion ensures that small changes in the plaintext or in the key rapidly spread across the entire internal state. In modern designs, these principles are realized through the careful combination of non-linear and linear layers.

A typical round function is composed of three main ingredients. The substitution layer introduces non-linearity by means of small lookup tables known as substitution boxes, or S-boxes, which provide resistance against linear and differential attacks. The diffusion layer spreads the influence of each input bit over many output bits, usually through linear transformations such as permutations or matrix multiplications. Finally, a key addition layer injects round-dependent key material, preventing an attacker from isolating the effect of the fixed transformations from that of the secret key. This modular structure not only leads to efficient hardware and software implementations, but also provides a natural framework for cryptanalysis, as the propagation of information through each layer can be modeled precisely.

The security of an iterated block cipher depends critically on the quality of its internal components and on the number of rounds. Poorly designed S-boxes, insufficient diffusion, or weak key schedules may introduce statistical biases that accumulate over several rounds and eventually lead to practical attacks. For this reason, modern cipher designs are typically accompanied by extensive cryptanalytic evaluation and by conservative choices of security margins. Prominent examples include the Data Encryption Standard (DES) and the Advanced Encryption Standard (AES), which has become the de facto standard for symmetric encryption worldwide [DR06].

Furthermore, the design and security analysis of symmetric cryptographic primitives constitute a cornerstone of modern information security. This chapter provides a comprehensive overview of the theoretical and practical frameworks necessary to understand the architecture of modern block ciphers and the cryptanalytic techniques used to evaluate their resistance against statistical attacks.

We begin this chapter in **Section 2.1** by formalizing the notation and exploring the main design paradigms used to construct these functions: the Substitution-Permutation Network (SPN), the Feistel Network, and the Add-Rotate-Xor (ARX) architecture. To provide concrete examples, **Section 2.2** details the specifications of the specific lightweight primitives analyzed in this work, including the Feistel-based Simon, Warp, Twine, Simeck, and Speck families, as well as SPN designs like Aradi and Skinny.

Having established the structural foundations, we turn our attention to security evaluation. **Section 2.3** introduces **Differential Cryptanalysis**, a powerful statistical attack proposed publicly by Biham and Shamir in 1990 [BS91]. This technique, which examines how input differences propagate through the cipher's rounds, revolutionized

the field, requiring the development of formal security bounds. We discuss the theoretical underpinnings of this method, including the concept of *Markov Ciphers* and the *Stochastic Equivalence Hypothesis*, which allow us to model the propagation of differences as a probabilistic chain. We also touch upon **Truncated Cryptanalysis** in **Section 2.4**, a variant introduced by Knudsen in 1995 [Knu95] that relaxes the constraints of specific bit-level differences to bundles of differences.

However, as block ciphers increased in complexity and number of rounds, finding a single high-probability differential characteristic that covers the entire encryption process became increasingly difficult. To address this limitation, David Wagner introduced the **Boomerang Attack** in 1999 [Wag99], enabling cryptanalysts to combine two shorter, high-probability differentials into a distinguisher for the full cipher. **Section 2.5** trace the extensive evolution of this technique, covering the *Amplified Boomerang* [KKS00] and the *Rectangle Attack* (Biham et al.) [BDK01], culminating in the **Sandwich Attack** framework proposed by Dunkelman et al. [DKS14]. This framework remodels the cipher into two distinct differential parts separated by a transition layer, or "switch".

Finally, **Section 2.5.4** presents the most recent advances in the analysis of this transition layer. We discuss the unification of boomerang switches through the **Boomerang Connectivity Table (BCT)**, introduced by Cid et al. at Eurocrypt 2018 [CHP+18]. We conclude by examining the latest generalizations of this concept—the Upper, Lower, and Extended BCTs (UBCT, LBCT, EBCT)—which account for the complex dependencies between the upper and lower differential trails, providing a more accurate estimation of the attack's success probability and correcting previous assumptions of independence.

2.1 Iterated Ciphers

Most modern block ciphers incorporate a sequence of permutation and substitution operations. A commonly used design is based on an **iterated cipher**. An iterated cipher requires the specifications of a **round function** or a **primitive** and a **key schedule**, and the encryption function of a plaintext proceed through n similar **rounds**. Let K be a random binary key of some specified length k ($K \in \{0,1\}^k$). K is used to build n **round keys** (or **subkeys**), which are denoted by $K^1, \ldots, K^n$. The list of round keys, $(K^1, \ldots, K^n)$, is derived by an algorithm called the **key schedule**. The key schedule is constructed from K, the **master key**, that is **private**, using a fixed public algorithm. The round function, say g, takes two inputs: a round key K^r and a current **state**, denoted by w^{r-1}. The next state is defined as $w^r = g(w^{r-1}, K^r)$ where w^0 is the plaintext and w^n is the ciphertext. Decryption is only possible if the function g is bijective. This is true when the key is fixed and then there exists a function g^{-1} such that:

$$g_y^{-1}\left(g_y(w)\right) = w$$

SPN Cipher. A special class of iterated ciphers are the Substitution Permutation Network (SPN) ciphers.

Definition 2.1 (SPN state)**.** The state $w \in \{0,1\}^n$, with n called the **block length**, of a SPN cipher, is split in t parts of s bits denoted by $w_{<1>}||\cdots||w_{<t>}$ where each part has the same length. The substring of the state $w_{<i>} \in \{0,1\}^s$ (with $s \cdot t = n$) are called **words** or **blocks of the SPN state**. Furthermore, the SPN at round r and the i-th word of the state at the same round are denoted respectively by

$$w^r \quad \text{and} \quad w^r_{<i>}$$

A SPN is built from two components:

$$\pi_S : \{0,1\}^s \to \{0,1\}^s \quad \text{and} \quad \pi_P : \{1,\ldots,n\} \to \{1,\ldots,n\}$$

where π_S is a bijective non-linear function called **S-Box** and it is applied to the blocks of the state ($\pi_S(w_{<i>})$), and π_P is a permutation used to permute the n bits by changing their order. In order to obtain the state for the next round, it is usually applied a function that involves both the state and the round key called **round key mixing**.

Example 2.2 (Toy Example [Sti06])**.** Let's take $s = t = 4$. The total length of the bit input string of $s \cdot t = 16$, and a block is of length 4. The block of 4 bits is written in hexadecimal notation (i.e. $0 = (0,0,0,0)$, $1 = (0,0,0,1)$, $\ldots$, $f = (1,1,1,1)$). So, we define the S-Box operation π_S as:

w	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$\pi_S(w)$	e	4	d	1	2	f	b	8	3	a	6	c	5	9	0	7

While the permutation π_P as:

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\pi_P(x)$	1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	16

The round key mixing is the XOR with the all state of 16 bits with a round key of 16 bits. We take as a master key $K \in \{0,1\}^{32}$ and as a round key K^r we use 16 consecutive bits starting from K_{4r-3}. We suppose the master key is:

$$K = \texttt{0011 1010 1001 0100 1101 0110 0011 1111}$$

we have the following round keys $K^1 = \texttt{0011 1010 1001 0100}$, $K^2 = \texttt{1010 1001 0100 1101}$, ecc. The round function and the encryption function are defined as:

$$f_r(w,K) = P(S((w \oplus K^r))) \quad E_R(x) = f_R(f_{R-1}(\cdots(f_1(x))))$$

where $S(x) = (\pi_S(x_{<1>}), \pi_S(x_{<2>}), \pi_S(x_{<3>}), \pi_S(x_{<4>}))$, $P(x) = (x_{\pi_P(1)}, \ldots, x_{\pi_P(16)})$

and R is the a priori selected number of rounds. If we take as input the plaintext $p = x^0 = \texttt{0010 0110 1011 0111}$ we have after one round: $x^1 = \texttt{0010 1110 0000 0111}$ and so on.

ARX Cipher. [YZS+15, RB08, SM87] Another special class of iterated ciphers is the so called ARX cipher, where ARX denotes the operations used for the encryption are the

- (Modular) Addition, $\boxplus : \mathbb{Z}_n \times \mathbb{Z}_n \to \mathbb{Z}_n$,
- (Left or Right Circular) Rotation, $\ggg$: $\{0,1\}^n \to \{0,1\}^n$,
- XOR with the key: $\oplus : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$

In this case the non-linear layer is composed by the Modular Addition and for the decryption it is used the modular subtraction.

Feistel Cipher. [YZS+15, Cop94, Bir11] A **Feistel cipher** is a cipher which divides the states into two halves of equal length, say L^i and R^i. The round function g is in the form of: $g(L^{i-1}, R^{i-1}, K^i) = (L^i, R^i)$ where:

$$L^i = R^{i-1}$$
$$R^i = L^{i-1} \oplus f(R^{i-1}, K^i)$$

The round function f is a composition of different functions, according to which property of security a client wants to reach. Also, make notice of the Generalized Feistel Structure i.e. an SPN cipher that applies the non-linear function only to half of the words of the state as a Feistel cipher.

2.2 Description Cipher

In this section, we provide the specifications of the cryptographic primitives analyzed throughout this thesis. Our focus is primarily on **Lightweight Block Ciphers**, a category of algorithms designed to operate efficiently in resource-constrained environments such as IoT devices. These primitives serve as the testing ground for the cryptanalytic techniques and automated tools developed in the subsequent chapters. Specifically, we investigate the security of these ciphers against Neural Distinguishers and automated differential attacks:

- **Speck** (Section 2.2.2): This ARX-based cipher serves as the primary benchmark in **Chapter 6**, where we analyze the performance of the neural distinguishers originally introduced by Gohr [Goh19] and the subsequent automated frameworks.

- **Simon** and **Simeck** (Section 2.2.1): These Feistel-based families are the main targets of our novel *Generic Partial Decryption* framework presented in **Chapter 7**. We demonstrate how feature engineering can improve neural distinguishers for these ciphers compared to state-of-the-art methods.
- **Aradi** (Section 2.2.3): This recent SPN cipher is a critical case study in **Chapter 7**. We present the first practical neural key recovery attack on 5 rounds of Aradi and show how neural analysis can reveal weaknesses exploitable by classical attacks.
- **WARP, TWINE, and SKINNY** (Sections 2.2.4, 2.2.5, 2.2.6): We also describe these prominent lightweight ciphers (Generalized Feistel Structures and SPN) to provide a comprehensive overview of modern design paradigms. These are frequently used as benchmarks in the literature of automated cryptanalysis (e.g., to test the TAGADA tool discussed in Chapter 4).

The following subsections outline the round functions without the key schedules necessary to understand the attacks implemented later in this work.

2.2.1 Simon and Simeck

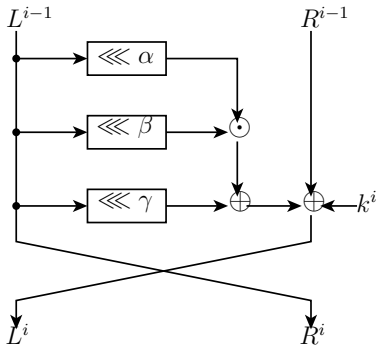

FIGURE 2.1 Round function of SIMON and SIMECK. SIMON has rotational constants: $\alpha = 1$, $\beta = 8$ and $\gamma = 2$, while SIMECK has $\alpha = 0$, $\beta = 5$ and $\gamma = 1$

SIMON $n/2n$ [BSS+13] and SIMECK $n/2n$ [YZS+15] are families of lightweight block ciphers, in particular they are Feistel cipher. Indeed, the state is split into two blocks of equal length of n bits: $w = L||R$, i.e. $L = w_{<0>}$ and $R = w_{<1>}$. The state is updated using the round key k^i as follows:

$$\begin{aligned} L^i &\leftarrow \left[(L^{i-1} \lll \alpha) \odot (L^{i-1} \lll \beta)\right] \oplus (L^{i-1} \lll \gamma) \oplus R^{i-1} \oplus k^i \\ R^i &\leftarrow L^{i-1} \end{aligned} \tag{2.1}$$

The differences on these two ciphers relies on the key schedules that we don't write there. The possible variants for these are $n = 32, 48, 64$. During this work, we focus

on the variants using a 32-bit state and a 64-bit key, SIMON 32/64 and SIMECK 32/64. The first uses the rotational constants $\alpha = 1$, $\beta = 8$ and $\gamma = 2$, while for SIMECK 32/64 $\alpha = 0$, $\beta = 5$ and $\gamma = 1$.

2.2.2 Speck

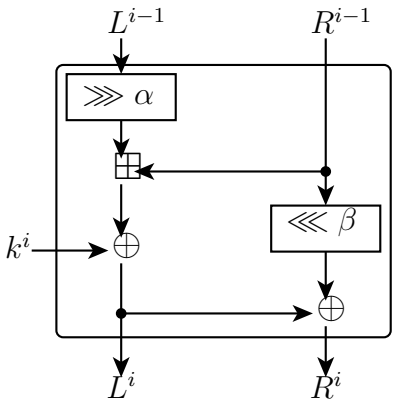

FIGURE 2.2 SPECK round function.

SPECK $n/2n$ is part of the family of lightweight Feistel block cipher designed, together with SIMON, in [BSS+13]. The possible variants we can see are: $n = 32, 48, 64$. We focus on the variant using a 32-bit state and a 64-bit key, which we shall denote as SPECK 32/64. During encryption, the state is split into two equal-sized words, which are then processed by a Feistel-like round function depicted in Figure 2.2 for both the ciphers. The rotational constants are $\alpha = 7$ and $\beta = 2$. The round function is defined like that:

$$L^i = [(L^{i-1} \gg \alpha) \boxplus R^{i-1}] \oplus k^i$$
$$R^i = (R^{i-1} \ll \beta) \oplus [(L^{i-1} \gg \alpha) \boxplus R^{i-1}] \oplus k^i$$

2.2.3 Aradi

The block cipher ARADI [GMW24] was published by NSA researchers without a security analysis, and very few public independent analyses have been performed (e.g. [BRRT24, BFG+24, ADG24]).

ARADI [GMW24] is a substitution-permutation network (SPN), taking as input a 128-bit block and a 256-bit key designed for low-latency applications.

The round function updates the state $(w||x||y||z)$ represented as four 32-bit words

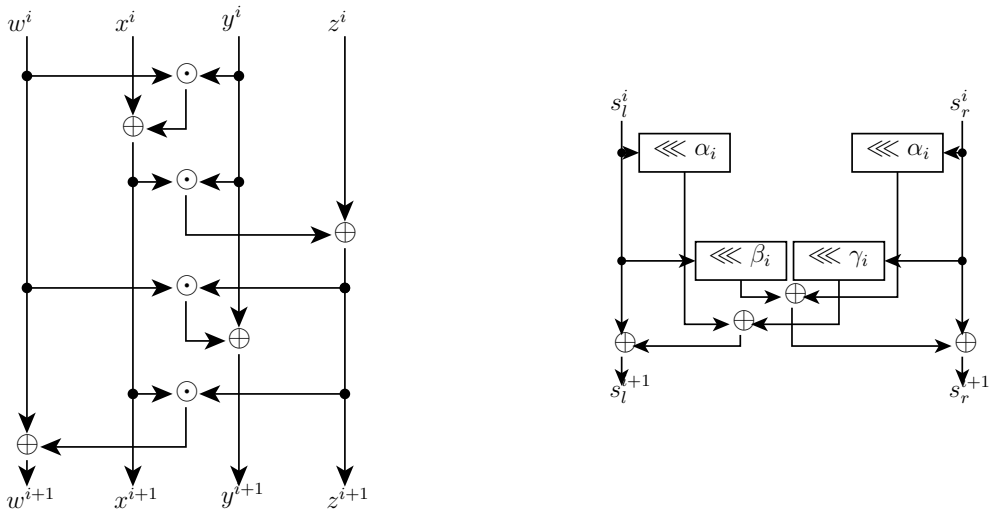

FIGURE 2.3 On the left the S-box π and on the right the component function L_i of the linear layer Λ_i in ARADI.

through a column-wise 4-bit S-box π followed by a row-wise linear layer Λ_i :

$$\Lambda_i(w||x||y||z) := (L_i(w)||L_i(x)||L_i(y)||L_i(z)),$$

$$L_i(s) = L_i(s_l||s_r) := (s_l \oplus (s_l \lll \alpha_i) \oplus (s_r \lll \gamma_i)||s_r \oplus (s_r \lll \alpha_i) \oplus (s_l \lll \beta_i))$$

If we denote by $\tau_v(s)$, the bitwise XOR of s with v, then the ARADI encryption function has the form:

$$\tau_{k^{16}} \circ (\Lambda_{15} \circ \pi \circ \tau_{k^{15}}) \circ \cdots \circ (\Lambda_1 \circ \pi \circ \tau_{k^1}) \circ (\Lambda_0 \circ \pi \circ \tau_{k^0})$$

where k^i denotes the round key, we consider it composed of four words of 32-bits $k^i = k_w^i||k_x^i||k_y^i||k_z^i$ corresponding to the four words of the block.

2.2.4 Warp

It receives a 128-bit plaintext and a 128-bit master key and then performs 40 full rounds plus one partial round (without nibble permutation) to produce a 128-bit ciphertext. Employing a 32-branch generalized Feistel structure (GFS), WARP aims at providing 128-bit security in the single-key setting while achieving a small footprint. The internal state of WARP can be represented as $w = w_{<0>}|| \cdots ||w_{<31>}$, where $w_{<i>} \in \{0,1\}^4$. WARP splits the 128-bit master key K into two 64-bit halves, $K = K^0||K^1$ and $K^j = K^j_{<0>}|| \ldots ||K^j_{<15>}$ with $j \in \{0,1\}$ and $K^j_{<i>} \in \{0,1\}^4$. $K^{(r-1) \bmod 2}$ is used as the round-key in the r-th round. The round function of WARP applies the same 4-bit S-

box and round-key addition to one of each two consecutive nibbles of the internal state. Afterwards, a permutation π is applied to the nibbles of the state. We refer to design paper [BBI+21] for a full specification.

Algorithm 1 `warp`. R is the number of round, rc_0^r and rc_1^r are round constant to XOR with the 4-bit block.

for $r = 1, \ldots, R+1$ **do**
  for $i = 0, \ldots, 15$ **do**
    $w^r_{<2i+1>} \leftarrow S(w^r_{<2i>}) \oplus w^r_{<2i+1>} \oplus K^{(r-1) \bmod 2}_{<i>}$
  end for
  $w^r_{<1>} \leftarrow w^r_{<1>} \oplus rc_0^r$
  $w^r_{<3>} \leftarrow w^r_{<3>} \oplus rc_1^r$
  $w^{r+1}_{<0>} || \ldots || w^{r+1}_{<31>} \leftarrow w^r_{<\pi(0)>} || \ldots || w^r_{<\pi(31)>}$
end for

2.2.5 Twine

Twine is a 64-bit block cipher with 80 or 128-bit key. We write Twine-80 or Twine-128 to denote the key length [KSMM12]. This cipher uses a Type-2 generalized Feistel structure. The state of Twine is divide into 16 nibbles (i.e. 16 words of 4-bit). The encryption function works very similar to Warp but in this case we don't have the round constants and the the S-Box is applied to the XOR of the even stat with the round key rk^r, see algorithm 2.

Algorithm 2 `twine`. R is the number of round and rk^r indicates the round key.

for $r = 0, \ldots, R$ **do**
  for $i = 0, \ldots, 7$ **do**
    $w^r_{<2i+1>} \leftarrow S(w^r_{<2i>} \oplus rk^r) \oplus w^r_{<2i+1>}$
  end for
  $w^{r+1}_{<0>} || \ldots || w^{r+1}_{<31>} \leftarrow w^r_{<\pi(0)>} || \ldots || w^r_{<\pi(31)>}$
end for

2.2.6 Skinny

SKINNY is a family of very lightweight tweakable block ciphers designed by Beierle et al. and presented at CRYPTO'16 [BJK+16]. The round function of SKINNY follows

a classical SPN structure and a non-MDS binary matrix is used for the MixColumns operation. The matrix has a low weight (only half of the coefficient are non-zero) and the tweakey is xored to the two first rows only. It enciphers blocks of length $n = 64$ or $n = 128$ bits seen as a 4×4 matrix of cells. SKINNY has three main tweakey size versions: the tweakey size can be equal to $t = 64$ or 128 bits, $t = 128$ or 256 bits, and $t = 192$ or 384 bits, and we denote $z = n/t$ the tweak size to the block size ratio. The number of rounds is directly derived from the z value: between 32 rounds for the 64/64 version up to 56 for the 128/384 version. SKINNY64 (128) applies first the 4 (8) bit S-Boxes to each cell of the matrix (`SB`). After, a XOR with some constants (`AC`) and with the Tweak (`ART`). Then, for each row is applied a circular rotation to the left of a value of bits according the row (`SR`). Finally, it is applied a Galois Multiplication with a matrix for each column of the state (`MC`), see Figure 2.4. We refer the interested reader to [BJK+16] for more details.

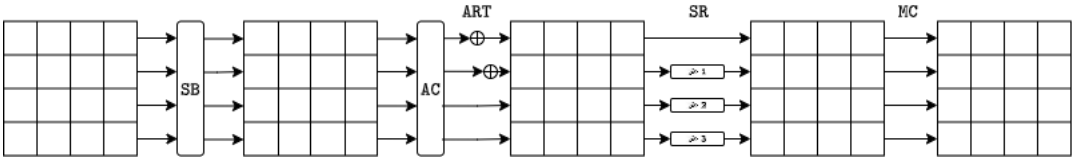

FIGURE 2.4 `skinny`.

2.3 Differential Cryptanalysis

Differential cryptanalysis, the study of how modifications of the inputs propagate to the output, is one of the most important techniques for establishing bounds on the resistance of block ciphers. While it was first publicly presented by Biham and Shamir in 1990 [BS91], it is believed to have been known and used as early as 1974 [Cop94].

Definition 2.3 (Perfect Secrecy)**.** According to [Sha49] a cryptosystem has **perfect secrecy** if

$$\mathbb{P}(x|y) = \mathbb{P}(x)$$

for all possible x in the plaintext set, y in the ciphertext set. That is, the a posteriori probability that the plaintext is x, given the ciphertext y is observed, is identical to the a priori probability that the plaintext is x. That's the same of showing [Sha49]:

$$\mathbb{P}(y|x) = \mathbb{P}(y)$$

In this setting, the aim of Differential Cryptanalysis is to analyze the propagation of the state differences during the encryption process in order to find an underlying distribution in the ciphertext/plaintext set.

Differential Cryptanalysis 2.4. Given two inputs, $x, x^* \in \mathcal{X}$, the encryption function $E_K : \mathcal{X} \to \mathcal{Y}$, and the corresponding output, $y = E_K(x), y^* = E_K(x^*)$, $y, y^* \in \mathcal{Y}$, we would like to estimate

$$\mathbb{P}_{\mathcal{Y}}(y \oplus y^* = \delta_{out} | x \oplus x^* = \delta_{in}).$$

Usually, a round function is said to be **cryptographic weak** if, given *few* triples (δ, y, y^*), it is feasible to determine the subkey K^r (this is an unformal definition, it is not clear what feasible means from the contest).

2.3.1 Differential Cryptanalysis on Iterated ciphers.

In this section, we use the follow notation:

$$y(i) = f_i(\cdots(f_1(x, K))) \quad \text{and} \quad y(0) = x$$

where f_i is the the round function applied at the round i. Furthermore, when we refer to the term *difference* it means the *bit-wise XOR*.

We adopt the formalism of [LMM91]:
Definition 2.5 (i-round Differential)**.** An **i-round differential** is a couple $(\delta_{out}, \delta_{in})$, where δ_{in} is the difference plaintexts x and x^*, i.e. $x \oplus x^* = \delta_{in}$ and where δ_{out} is a possible difference for the resulting i-th round outputs $y(i)$ and $y^*(i)$, i.e. $y(i) \oplus y^*(i) = \delta_{out}$. The **probability of an i-round differential** $(\delta_{in}, \delta_{out})$ is

$$\mathbb{P}_{\mathcal{Y}}(y(i) \oplus y^*(i) = \delta_{out} | x \oplus x^* = \delta_{in})$$

when the plaintext x and the subkeys $K^1, \ldots, K^i$ are independent and uniformly random.
Definition 2.6 (i-round Differential Trail)**.** A **i-round differential trail** or **characteristic** is a sequence of $i+1$ differences through the i round functions $(\delta_0, \delta_1, \ldots, \delta_i)$ where $\delta_0 = x \oplus x^*$ is the difference of the plaintexts and $\delta_j = y(j) \oplus y^*(j)$ for $j = 1, \ldots, i$, is the difference of the output states after each round. The **probability of an i-round differential trail** $(\delta_0, \delta_1, \ldots, \delta_i)$ is

$$\mathbb{P}_{\mathcal{Y}}(y(i) \oplus y^*(i) = \delta_i | x \oplus x^* = \delta_0, y(1) \oplus y^*(1) = \delta_1, \ldots, y(i-1) \oplus y^*(i-1) = \delta_{i-1})$$

An other notation for a i-round differential trail is:

$$\delta_0 \xrightarrow{E_K} \delta_i$$

or $\delta_0 \to \delta_i^*$ if the encryption function is clear from the context but the presence of the intermediate states is implied even if they are not explicitly written.

A classical differential attack on an r-round iterated cipher is divided into three step:

1. Find a $r-1$ round differential $(\delta_{in}, \delta_{out})$ with maximum, or nearly, maximum probability;
2. Choose a plaintext x uniformly at random and compute x^* such that $x^* = x \oplus \delta_{in}$; submit x and x^* under the actual (unknown) key K to the encryption for r rounds; from the ciphertexts $y(r)$ and $y^*(r)$ assign a score to all the possible subkeys K^r corresponding to the last round such that the difference at round $r-1$ is δ_{out} $(y(r-1) \oplus y^*(r-1) = \delta_{out})$;
3. Repeat step 2 until we have a significant score for a candidate subkey $\tilde{K}^r$

All the steps are done with two implicit hypothesis.

Hypothesis of Stochastic Equivalence. *For an $r-1$ round differential $(\delta_{in}, \delta_{out})$:*

$$\mathbb{P}(y(r-1) \oplus y^*(r-1) = \delta_{out} | x \oplus x^* = \delta_{in}) \approx$$
$$\mathbb{P}(y(r-1) \oplus y^*(r-1) = \delta_{out} | x \oplus x^* = \delta_{in}, K^1 = \omega_1, \ldots, K^{r-1} = \omega_{r-1})$$

for almost all the subkeys values $(\omega_1, \ldots, \omega_{r-1})$.

In other words, we are saying that if we pick the key uniformly at random than the probability to propagate a difference doesn't depend on the key but only by the differences. The second hypothesis, called the **Wrong Key Randomization Hypothesis**, was introduced in [HKM95] for linear cryptanalysis. Later, [BG11] applied it also for differential cryptanalysis.

Wrong Key Randomization Hypothesis 1. *For a $r-1$-differential $(\delta_{in}, \delta_{out})$*

$$\mathbb{P}\left(f_k^{-1}(E_{K^*}(x)) \oplus f_k^{-1}(E_{K^*}(x \oplus \delta_{in})) = \delta_{out}\right) = \begin{cases} p & \text{if } k = k^* \\ \frac{1}{2^m - 1} & \text{if } k \neq k^* \end{cases}$$

where p is a probability different from the uniform distribution and m is the length of last subkey $k \in \{0,1\}^m$.

In other words, if we decrypt the last round using the wrong key the probability of the $r-1$-differential is the uniform probability. This hypotesis is said to be wrong because it is not actually the uniform distribution and doesn't take in account other relevant elements [ABR20]. Under these assumptions, in [LMM91] is defined a lower bound for the complexity to recover the subkey.

Theorem 2.7 ([LMM91])**.** *Suppose the hypothesis of stochastic equivalence is true,*

then, in an attack by differential cryptanalysis:

$$Complexity(r) \geq \frac{2}{p_{max} - \frac{1}{2^m - 1}}$$

where $p_{max} = \max_{\delta_{in}} \max_{\delta_{out}} \mathbb{P}(y(r-1) \oplus y^*(r-1) = \delta_{out} | x \oplus x^* = \delta_{in})$ *and* m *is the block length. Furthermore, if* $p_{max} \approx \frac{1}{2^m - 1}$ *then a differential cryptanalysis attack doesn't succeed.*

It makes sens to describe how an actual classical attack works to evaluate the probability of a differential.

Step 1. Classical Search of Differential. In order to search for differential, we have to divide into two cases. In the first case, we want to analyze the propagation of the difference through one round of the iterated cipher, in the second case we try to analyze the propagation though multiple rounds.

For the first case, to evaluate the difference propagation probability after one round of an iterated cipher:

$$\mathbb{P}(y(1) \oplus y^*(1) = \delta_{out} | x \oplus x^* = \delta_{in})$$

We split the round function into

(i) **linear** (as permutation or rotational shift or XOR with a constant),
(ii) **key mixing** (as XOR with the key, but in general again a linear one),
(iii) **non-linear** (as S-Box or Modular Addition or AND).

. Let f be a linear function, than we can have two cases:

1. $f(x) \oplus f(x^*) = f(x \oplus x^*) = f(\delta_{in})$ (e.g. $f(x) = x \ggg \alpha$), since δ_{in} and δ_{out} are fixed and f is a function that leads to the same output, we have

$$\mathbb{P}(y(1) \oplus y^*(1) = \delta_{out} | x \oplus x^* = \delta_{in}) =$$

$$\mathbb{P}(f(x) \oplus f(x^*) = \delta_{out} | x \oplus x^* = \delta_{in}) =$$

$$\mathbb{P}(f(\delta_{in}) = \delta_{out} | x \oplus x^* = \delta_{in}) = \begin{cases} 1 & \text{if } \delta_{out} = f(\delta_{in}) \\ 0 & \text{otherwise} \end{cases}$$

2. $f(x) \oplus f(x^*) = \delta$ (e.g. $f(x) = x \oplus c$) where c is a constant and we can do the same step as before but at end picking $\delta_{in} = \delta_{out}$.

So, in both of them the propagation of the difference is deterministic, i.e. it doesn't

depend by the choice of x and x^*.

In the case of non-linear functions we have to rely to other methods. The most common tool in the SPN cipher is the **Differential Distribution Table**.

Definition 2.8 (Differential Distribution Table)**.** Let $\pi_S : \{0,1\}^l \to \{0,1\}^m$ be an S-Box, $\delta \in \{0,1\}^l$ and $\Delta \in \{0,1\}^m$ be the entries of the table, then we can define the **Differential Distribution Table** (DDT) as:

$$\mathcal{D}_{\pi_S}(\delta, \Delta) = \left|\left\{x \in \{0,1\}^l \,|\, \pi_S(x) \oplus \pi_S(x \oplus \delta) = \Delta\right\}\right|$$

The **Propagation Ratio** is:

$$\mathcal{R}(\delta, \Delta) = \frac{\mathcal{D}_{\pi_S}(\delta, \Delta)}{2^l}$$

Then, the maximum value in the DDT is called **Differential Uniformity**.

Remark. *If we find an $x \in \{0,1\}^l$ such that $\pi_S(x) \oplus \pi_S(x \oplus \delta) = \Delta$ and let $x^* = x \oplus \delta$ i.e. $\pi_S(x) \oplus \pi_S(x^*) = \Delta$, then also for x^* we have $\pi_S(x^*) \oplus \pi_S(x^* \oplus \delta) = \pi_S(x^*) \oplus \pi_S(x) = \Delta$. It means that if the value of DDT is different from 0 then it is a multiple of 2.*

Example 2.9 (DDT)**.** Let's take the S-Box at 4 bit, $\pi_S : \{0,1\}^4 \to \{0,1\}^4$ presented in the round function in Example 2.2:

w	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$\pi_S(w)$	e	4	d	1	2	f	b	8	3	a	6	c	5	9	0	7

Taking, in hexadecimal again, $\delta = 1$ and $\Delta = 3$ we have:

$$(\pi_S(x) \oplus \pi_S(x \oplus 1) = 3 \iff x = 6 \vee x = 7) \implies \mathcal{D}_{\pi_S}(1, 3) = 2$$

So, the Propagation Ratio of the differential $(1, 3)$ is $\frac{2}{2^4} = 0.125$. The complete DDT is presented in Table 2.1.

Table 2.1 Differential Distribution Table of Example 2.9

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	0	0	0	2	0	2	4	0	4	2	0	0
2	0	0	0	2	0	6	2	2	0	2	0	0	0	0	2	0
3	0	0	2	0	2	0	0	0	0	4	2	0	2	0	0	4
4	0	0	0	2	0	0	6	0	0	2	0	4	2	0	0	0
5	0	4	0	0	0	2	2	0	0	0	4	0	2	0	0	2
6	0	0	0	4	0	4	0	0	0	0	0	0	2	2	2	2
7	0	0	2	2	2	0	2	0	0	2	2	0	0	0	0	4
8	0	0	0	0	0	0	2	2	0	0	0	4	0	4	2	2
9	0	2	0	0	2	0	0	4	2	0	2	2	2	0	0	0
a	0	2	2	0	0	0	0	0	6	0	0	2	0	0	4	0
b	0	0	8	0	0	2	0	2	0	0	0	0	0	2	0	2
c	0	2	0	0	2	2	2	0	0	0	0	2	0	6	0	0
d	0	4	0	0	0	0	0	4	2	0	2	0	2	0	2	0
e	0	0	2	4	2	0	0	0	6	0	0	0	0	0	2	0
f	0	2	0	0	6	0	0	0	0	4	0	2	0	0	2	0

From now, we consider the values in the DDT as their propagation ratio, hence we divided them for 2^l. So, the probability to propagate the difference though one round of the cipher depends on the DDT of the non-linear layer.

In order to propagate the difference through multiple rounds, it is necessary to separate two approaches. The first one, the most common used during a differential attack is based on the Markov assumption of independency between each round and the second one analyze the dependencies between the states of each round. For the latter, a lot of research has been done to evaluate the exact maximum expected differential probability in particular for AES, e.g. [HLL$^+$00, KMT01, PSC$^+$02, PSLL03, DR06, KS07, CR15] and for Feistel or MISTY networks, e.g. [NK93, Mat96], in which researches suppose that all state bits are xored by subkeys which are assumed to be chosen independently and uniformly at random, or [CLN$^+$17] in which in contrary focuses on determining the exact probability of a differential characteristic when the key is fixed, or [Sku23] that prove a new upper bound for SPN and Feistel Cipher. We focus on the former method in which we make use of the Markov Assumption.

Remark. *Let's recall that a sequence of discrete random variable $v_0, \ldots, v_r$ is a **Markov Chain** if, for $0 \leq i < r$ (where $r = \infty$ is allowed), it holds:*

$$\mathbb{P}(v_{i+1} = \beta_{i+1} | v_i = \beta_i, v_{i-1} = \beta_{i-1}, \ldots, v_0 = \beta_0) = \mathbb{P}(v_{i+1} = \beta_{i+1} | v_i = \beta_i)$$

*Furthermore, a Markov chain is called **homogeneous** if $\mathbb{P}(v_{i+1} = \beta | v_i = \alpha)$ is independent of i for all β and α.*

Definition 2.10 (Markov Cipher)**.** An iterated cipher with round function $y = f(x, K)$ is a **Markov cipher** if there is a group operation "+" (in our case is the XOR $\oplus$) for defining differences, we denote with $\Delta y = y + y^*$ such that, for all choices of α and β (where $\alpha, \beta \neq 0$ (where for 0 we mean the neutral element of the group where we are defining the operation +), we have that

$$\mathbb{P}(\Delta y = \beta | \Delta x = \alpha, x = \gamma)$$

is *independent* of γ when the subkey K is uniformly random, or, equivalently, if we have that

$$\mathbb{P}(\Delta y = \beta | \Delta x = \alpha, x = \gamma) = \mathbb{P}(\Delta y(1) = \beta | \Delta x = \alpha)$$

for all choices of γ when he subkey K is uniformly random.

So, to evaluate the probability of multiple rounds we refer to another theorem showed in [LMM91].

Theorem 2.11 ([LMM91])**.** *If an r-round iterated cipher is a Markov cipher and the r round keys are independent and uniformly at random, then the sequences of differences $\Delta x = \Delta y(0), \Delta y(1), \ldots, \Delta y(r)$ is an homogeneous Markov chain. Moreover, this Markov chain is stationary if Δx is uniformly distributed over the non-neutral elements of the group.*

Hence, for a Markov cipher with independent and uniformly random subkeys, the probability of an r-round differential characteristic is given by the Chapman-Kolmogorov equation [HMK81] for a Markov chain:

$$\mathbb{P}(\Delta y(1) = \beta_1, \ldots, \Delta y(r) = \beta_r | \Delta x = \beta_0) = \prod_{i=1}^{r} \mathbb{P}(\Delta y(1) = \beta_i | \Delta x = \beta_{i-1}) \tag{2.2}$$

It follows that the probability of an r-round differential (δ_0, δ_r) is:

$$\mathbb{P}(\Delta y(r) = \delta_r | \Delta x = \delta_0) = \sum_{\delta_1} \sum_{\delta_2} \cdots \sum_{\delta_{r-1}} \prod_{i=1}^{r} \mathbb{P}(\Delta y(1) = \delta_i | \Delta x = \delta_{i-1}) \tag{2.3}$$

where the sums are over all possible values of differences between distinct elements, i.e., over all group elements excepting the neutral element. The difference between Equation 2.2 and 2.3 is about the specifications of the all the state differences during the iterated cipher for r rounds, in the first we specify every difference (differential characteristic) while in the second we just specify the beginning and the end difference (differential).

Example 2.12 (Toy Cipher search differential)**.** Let's take again the SPN cipher introduced in Example 2.2 and his DDT (Table 2.1). Then, following the round function:

- Take $\delta_0 = (\texttt{0000 1011 0000 0000}) = \texttt{0x0b00}$;
- In the key mixing operation the difference remain the same;
- To analyze the propagation probability through the S-Box for the second block, we check the row of `0xb` and we can notice that the highest value of the DDT is in the second column and the Propagation Ratio is $\frac{1}{2}$. So now, the intermediate state after the S-Box is `0x0200` with a probability of 0.5;
- The last layer is the permutation that change the state in $\delta_1 = \texttt{0x0040}$.

We have found a 1 round differential $(\delta_0, \delta_1) = (\texttt{0x0b00}, \texttt{0x0040})$, with probability $\frac{1}{2}$. Continuing this process for another round, and taking the highest value of DDT it is possible to find another 1 round differential $(\delta_1, \delta_2) = (\texttt{0x0040}, \texttt{0x0220})$ with probability $\frac{3}{8}$. The total probability of the differential characteristic $(\delta_0, \delta_1, \delta_2) = (\texttt{0x0b00}, \texttt{0x0040}, \texttt{0x0220})$ is $\frac{1}{2} \times \frac{3}{8} = \frac{3}{16}$, while the probability of the differential (δ_0, δ_2) is

$$\mathbb{P}(\Delta y(2) = \delta_2 | \Delta x = \delta_0) = \\ = \sum_{\delta_1} \left(\mathbb{P}(\Delta y(1) = \delta_1 | \Delta x = \delta_0)\right) \left(\mathbb{P}(\Delta y(1) = \delta_2 | \Delta x = \delta_1)\right)$$

As shown in the example, it is always almost impossible to actually evaluate the probability of the differential since the state are composed to many evaluations (in the example 2^{16} possible elements). Nevertheless, using Equation 2.2 and 2.3 we can notice that the first one is a lower bound for the last one.

$$\mathbb{P}(\Delta y(r) = \beta_r | \Delta x = \beta_0) \geq \mathbb{P}(\Delta y(1) = \beta_1, \ldots, \Delta y(r) = \beta_r | \Delta x = \beta_0)$$

It implies, that if we are able to find a differential characteristic with sufficiently high probability, then the differential itself will have an high probability. Usually, differential probabilities are approximated with the probability of their best differential characteristic:

$$\mathbb{P}(\Delta y(r) = \beta_r | \Delta x = \beta_0) \approx \mathbb{P}(\Delta y(1) = \beta_1, \ldots, \Delta y(r) = \beta_r | \Delta x = \beta_0) \tag{2.4}$$

i.e., it is enough to show weakness in a cipher just finding a differential characteristic with high probability. Hence, when we talk about the probability of a differential characteristic it refers as well to the probability of its differential.

In the same paper [LMM91] it is worth to mention as well the last theorem regarding the security of a round function.

Theorem 2.13 ([LMM91])**.** *For a Markov cipher of block length m with independent and uniformly random round subkeys, if the semi-infinite Markov chain* $\Delta x =$

$\Delta y(0), \Delta y(1), \ldots$ *has a steady-state probability distribution, i.e., if there is a probability vector* $(p_1, \ldots, p_M)$ *such that, for all* α_i, $\lim_{r\to\infty} \mathbb{P}(\Delta y(r) = \alpha_j | \Delta x = \alpha_i) = p_j$, *then this steady state distribution must be the uniform distribution, i.e.,*

$$\lim_{r\to\infty} \mathbb{P}(\Delta y(r) = \alpha_j | \Delta x = \alpha_i) = \frac{1}{2^m - 1}$$

for every differential (α, β), *so that every differential is roughly equally likely after sufficiently many rounds. Furthermore, if we assume additionally that the hypothesis of stochastic equivalence holds for this Markov cipher, then, for almost all subkeys, this cipher is secure against a differential cryptanalysis attack after sufficiently many rounds.*

Step 2. Classical Key Recovery. In an r-round iterated cipher, the key schedule algorithm is employed to generate r round keys (or subkeys). It is, as well the round function, an invertible function. This implies that if we recovered one subkey we are able to recover all the subkeys used in the previous rounds. Suppose we have found a $r-1$-round differential (α, β) with a certain probability $p_{\alpha,\beta}$, and we would kike to extend the attack trying to actual recover some (hopefully all) bits of a subkey. In the Classical Key Recovery Algorithm, we would like to recover the unknown key $\tilde{K}$:

- Pick two plaintext with difference α;
- Encrypt for r rounds these two plaintexts using the unknown key;
- Pick all the *"right"* possible subkey for the last round, i.e. our guesses;
- Decrypt with our guess the encrypted ciphertext and if the difference is β then we increment the score of that subkey.

In order to the attack to be successful, the expected number of plaintext pairs N_p is required to be:

$$N_p > \frac{c}{p_{\alpha,\beta}}$$

Despite 50 years of research, new insights on differential cryptanalysis are still presented regularly at major security conferences. For instance, a new generic differential key recovery algorithm was presented at Eurocrypt 2024 [BDD+24]. These advances often go hand in hand with the improvement of tools. In particular, the search for differential characteristics has been made significantly easier in recent years through automated search tools based on SAT [SNC09], SMT, MILP [MWGP12], or CP [GMS16]. The ability to easily obtain, analyze, or count differential characteristics with given properties lets the cryptographer focus on novel analysis techniques rather than the tedious implementation of dedicated search algorithms. However, difficult instances keep challenging these modern tools despite the constant evolution of techniques [SWW21]; as a result, finding accuracy rate differential bounds is still an open problem for many primitives.

These challenges become even more acute when investigating variants of differential cryptanalysis, such as multiple differential cryptanalysis [BG11], where the cryptographer considers not one but many of the possible output differences. To this day, it is extremely difficult to evaluate the resistance of a block cipher to such attacks. Another variant of a classical differential attack are the Boomerang Attack, tackled in section 2.5 or the Truncated Cryptanalysis, discussed in section 2.4.

2.4 Truncated Cryptanalyis

The Truncated Cryptanalysis is a type of Differential cryptanalysis introduced in [Knu95]. The idea is in the middle between a $r-$round differential and an $r-$round differential characteristic. In particular, we can see the $r-$round differential characteristic as a specific sequence from the initial difference δ_{in} to the output difference δ_{out}, while the differential is the set of all the possible differences sequences from δ_{in} to δ_{out}, then a truncated characteristic is the set of all the sequences that follow a specific path of active bits. The former definition of a truncated differential captures this idea.

Definition 2.14 (Abstraction)**.** Let $w^j = w^j_{<1>}||\cdots||w^j_{<t>}, v^j = v^j_{<1>}||\cdots||v^j_{<t>} \in \{0,1\}^n$, with $w_{<i>}$, $v_{<i>} \in \{0,1\}^s$, be the states of an SPN cipher after j round and f_j is the round function. Furthermore, let δ^j be the differences after each round i.e. $\delta^j = w^j \oplus v^j$, notice that we can split the differences in words as well $\delta^j = \delta^j_{<1>}||\cdots||\delta^j_{<t>}$ where $\delta^j_{<i>} = w^j_{<i>} \oplus v^j_{<i>}$. Then, we say that a word of the difference is **active** at the round j, if at least one of its bit is active (or if at least one element of the bit string is 1).

$$\delta^j_{<i>} \text{ is active} \quad \iff \quad \delta^j_{<i>} \in (0, 2^s - 1]$$

We can call this process as the **Abstraction** of the difference. Furthermore, we can define the **abstracted** or **truncated** difference $\Delta : \underbrace{\{0,1\}^s \times \cdots \times \{0,1\}^s}_{t-times} \to \{0,1\}^t$ as $\Delta = (\Delta(\delta_{<1>}), \ldots, \Delta(\delta_{<t>}))$:

$$\Delta(\delta_{<i>}) = \begin{cases} 1 & \text{if } \delta^j_{<i>} \in (0, 2^s - 1] \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

Definition 2.15 (i-round Truncated Differential Trail)**.** A **i-round truncated differential trail** or **characteristic** for an SPN cipher is a sequence of $i+1$ truncated differences through the i round functions $(\Delta^0, \Delta^1, \ldots, \Delta^i)$ where Δ^j are the truncated differences for $j = 1, \ldots, i$ rounds of the cipher.

Let δ^0 and δ^1 be the input and the output difference through one round of an SPN

cipher and the probability of the differential characteristic is

$$\mathbb{P}(y(1) \oplus y(1)^* = \delta^1 | x \oplus x^* = \delta^0) = \prod_{i=1}^{t} \mathbb{P}(y(1)_{<i>} \oplus y(1)^*_{<i>} = \delta^1_{<i>} | x_{<i>} \oplus x^*_{<i>} = \delta^0_{<i>}) \tag{2.6}$$

where t is the number of words of the same length $\{0,1\}^s$ composing a state and $y(j)$ represent the output after j rounds.

In the case of the truncated differences, to analyze the probability of the propagation we can notice that, if the truncated input difference is 0, since the S-Box is a bijective function, then the difference is not propagated in the output with probability one:

$$\mathbb{P}(\Delta^1_{<i>} = 0 | \Delta^0_{<i>} = 0) = 1$$

If the input difference is one then it is propagated in the output difference with a certain probability according to the difference that allows the propagation in that round. To approximate this probability, it is usually applied an upper bound using the largest value in the DDT related to the specific S-Box S:

$$\mathbb{P}(\Delta^1_{<i>} = 1 | \Delta^0_{<i>} = 1) = \max_{\alpha,\beta} \mathcal{D}_S(\alpha, \beta)$$

Then, to estimate the propagation probability through the all word state, we should multiply the probability where the input and output word are 1. We can use the following formula:

$$\prod_{i=1}^{t} \mathbb{P}(\Delta^1_{<i>} = 1 | \Delta^0_{<i>} \in \{0,1\}) = \prod_{i=1}^{t} \max_{\alpha,\beta} (\mathcal{D}_S(\alpha, \beta))^{\Delta^0_{<i>}}$$

where $\Delta^0_{<i>}$ works as the indicator function, it is 1 if the word is active, furthermore, we are counting one active S-Box otherwise is 0 and we don't count that S-Box. Since the propagation probability of the differences is 1 of the linear layer and using the Markov assumptions, it is possible to propagate this idea in order to approximate the **probability of a r-round truncated characteristic**:

$$p_{trunc} \approx \left(\max_{\alpha,\beta} \mathcal{D}(\alpha, \beta) \right)^{\#NL} \tag{2.7}$$

where $\#NL$ indicates the number of non-linear layer active, i.e. with input truncated difference equal to 1. The use of truncated characteristics helps to give an upper bound

for the probability i.e.

$$p_{diff} \leq p_{trunc} \tag{2.8}$$

and this implies that if we are not able to find a truncated characteristics than we know for sure that we are not able to instantiate a differential characteristic.

2.5 Boomerang Attack

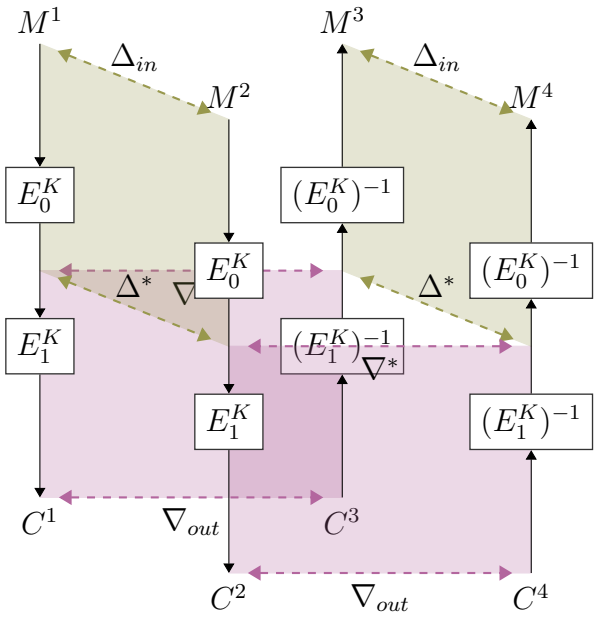

FIGURE 2.5 Boomerang Attack. $M^1 = M, M^2 = M \oplus \Delta_{in}, C^1 = E_K(M^1), C^2 = E_K(M^2), C^3 = C^1 \oplus \nabla_{out}$ and $C^4 = C^2 \oplus \nabla_{out}$.

The differential attack is not effective on some block ciphers when increasing the number of rounds [Vau98]. Thus, in 1999, David Wagner in [Wag99] introduced boomerang attacks. In this kind of attacks, an attacker tries to construct a distinguisher on a block cipher E where, after having ciphered a pair of messages with a difference Δ_{in}, the attacker shifts the ciphertexts from a difference ∇_{out} and hopes that the boomerang returns, i.e. that the two deciphers have a difference Δ_{in}, see Figure 2.5. In other words, the attacker aim for the following Boomerang relation:

$$E^{-1}(E(M) \oplus \nabla_{out}) \oplus E^{-1}(E(M \oplus \Delta_{in}) \oplus \nabla_{out}) = \Delta_{in} \tag{2.9}$$

that happens with a high probability, as far as possible from the uniform probability. To be more specific, the attack consider four plaintexts M^1, M^2, M^3, M^4 along with their respective ciphertexts C^1, C^2, C^3, C^4. Let $E(\cdot)$ represent the encryption function

and decompose the cipher into

$$E = E_1 \circ E_0$$

where E_0 represent the first part of the cipher, with r_0 rounds of E and E_1 the last part, with r_1 rounds such that $r_0 + r_1 = r$. [Wag99] used a differential (Δ_{in}, Δ^*) of E_0 and a differential characteristic (∇^*, ∇_{out}) for E_1. The pair M^1, M^2 is the input of differential of E_0 and the outputs of E_0 using the pairs M^1, M^3 and M^2, M^4 is a suitable input for the differential of E_1. Then, M^3 and M^4 create a pair to use for the differential (Δ_{in}, Δ^*) of E_0:

$$\begin{aligned} E_0(M^3) \oplus E_0(M^4) &= E_0(M^1) \oplus E_0(M^2) \oplus E_0(M^1) \oplus E_0(M^3) \oplus E_0(M^2) \oplus E_0(M^4) \\ &= E_0(M^1) \oplus E_0(M^2) \oplus E_1^{-1}(C^1) \oplus E_1^{-1}(C^3) \oplus E_1^{-1}(C^2) \oplus E_1^{-1}(C^4) \\ &= \Delta^* \oplus \nabla^* \oplus \nabla^* = \nabla^* \end{aligned}$$

A **right quartet** is a quartet where all four characteristics, for $E_0, E_0^{-1}, E_1^{-1}, E_1$, hold simultaneously. In other words, the attacker tries to combine a first differential that happens for the first rounds, say E_0, and a second covering the final rounds, say E_1. Thus, the cipher E is entirely covered by the distinguisher cut into two parts E_0 and E_1 ($E = E_1 \circ E_0$). The main objective is to focus on the intermediate condition to be satisfied, i.e., letting $X^i = E_0(M^i)$, we should have:

$$X^1 \oplus X^2 = \Delta^* \quad \text{and} \quad X^1 \oplus X^3 = \nabla^* \quad \text{and} \quad X^2 \oplus X^4 = \nabla^* \tag{2.10}$$

Definition 2.16 (Boomerang Differentials)**.** Let $E(\cdot)$ be the encryption function with r rounds, E_0 be the first part of the cipher, with r_0 rounds and E_1 the last part, with r_1 rounds such that $r_0 + r_1 = r$ such that $E = E_1 \circ E_0$. Then:

- a $r_0 + r_1$**-round boomerang differential** is a couple (Δ, ∇), where Δ is the difference plaintexts of E_0 and where ∇ is the difference ciphertexts of E_1 and the condition 2.10 holds;
- a $r_0 + r_1$**-round boomerang differential trail** or **characteristic** is a quartet $(\Delta_{in}, \Delta^*, \nabla^*, \nabla_{out})$ such that (Δ_{in}, Δ^*) is a differential for E_0 and (∇^*, ∇_{out}) is a differential for E_1^{-1} and the condition 2.10 holds.

Remark. *Since the encryption function is an invertible function, the probability of a differential is the same of its inverse i.e.:*

$$\mathbb{P}^E(\Delta y(i) = \Delta_{out} | \Delta x = \Delta_{in}) = \mathbb{P}^{E^{-1}}(\Delta y(i) = \Delta_{in} | \Delta x = \Delta_{out})$$

where the superscript of $\mathbb{P}^{(\cdot)}$ *indicate the encryption function. This holds as well for the differential characteristics.*

With this remark and the assumption of Markov hypothesis, we can observe that the

probability of a $(\Delta_{in}, \Delta^*, \nabla^*, \nabla_{out})$ $(r_0 + r_1)$-boomerang differential characteristic is:

$$\underbrace{\mathbb{P}^{E_0}(\Delta y(r_0) = \Delta^* | \Delta x = \Delta_{in})}_{M^1, M^2} \cdot \underbrace{\mathbb{P}^{E_0}(\Delta y(r_0) = \Delta^* | \Delta x = \Delta_{in})}_{M^3, M^4} \cdot$$
$$\underbrace{\mathbb{P}^{E_1^{-1}}(\Delta y(r_1) = \nabla^* | \Delta x = \nabla_{out})}_{C^1, C^3} \cdot \underbrace{\mathbb{P}^{E_1^{-1}}(\Delta y(r_1) = \nabla^* | \Delta x = \nabla_{out})}_{C^2, C^4} =$$
$$\mathbb{P}^{E_0}(\Delta y(r_0) = \Delta^* | \Delta x = \Delta_{in})^2 \cdot \mathbb{P}^{E_1^{-1}}(\Delta y(r_1) = \nabla^* | \Delta x = \nabla_{out})^2$$

It is possible to approximate the **boomerang success probability** p_{boom}, in other words the probability of $(r_0 + r_1)$-*boomerang* differential, as:

$$\begin{aligned} p_{boom} &= \sum_{\Delta^*} \mathbb{P}^{E_0}(\Delta y(r_0) = \Delta^* | \Delta x = \Delta_{in})^2 \sum_{\nabla^*} \mathbb{P}^{E_1^{-1}}(\Delta y(r_1) = \nabla^* | \Delta x = \nabla_{out})^2 \\ &\geq \mathbb{P}^{E_0}(\Delta y(r_0) = \Delta^*_{fix} | \Delta x = \Delta_{in})^2 \cdot \mathbb{P}^{E_1^{-1}}(\Delta y(r_1) = \nabla^*_{fix} | \Delta x = \nabla_{out})^2 \\ &\geq \mathbb{P}(\Delta_{in} \xrightarrow{E_0} \Delta^*_{fix})^2 \cdot \mathbb{P}(\nabla_{out} \xrightarrow{E_1^{-1}} \nabla^*_{fix})^2 \end{aligned}$$

In the common attacks that involve the boomerang attack, it is unfeasible to evaluate the actual probability of a boomerang differential. Then, like the case of differential we are using the probability of the boomerang differential characteristic to approximate the actual probability (even if it is a lower bound, like we have done in Equation 2.4):

$$\begin{aligned} p_{boom} &\approx \mathbb{P}^{E_0}(\Delta y(r_0) = \Delta^* | \Delta x = \Delta_{in})^2 \cdot \mathbb{P}^{E_1^{-1}}(\Delta y(r_1) = \nabla^* | \Delta x = \nabla_{out})^2 \\ &\approx \mathbb{P}(\Delta_{in} \xrightarrow{E_0} \Delta^*)^2 \cdot \mathbb{P}(\nabla_{out} \xrightarrow{E_1^{-1}} \nabla^*)^2 = p^2 q^2 \end{aligned} \tag{2.11}$$

where p and q represent the probability of the *partial* differential characteristics of respectively r_0 rounds of $\Delta_{in} \xrightarrow{E_0} \Delta^*$ and r_1 rounds of $\nabla_{out} \xrightarrow{E_1^{-1}} \nabla^*$.

A classical boomerang attack to recover the right quartets (i.e. to create a boomerang distinguisher) is:

1. Find a boomerang differential (Δ, ∇) with high probability $p^2 q^2$;
2. Pick a random plaintext M^1 and let $M^2 = M^1 \oplus \Delta$;
3. Encrypt the plaintexts: $C^1 = E^K(M^1)$ and $C^2 = E^K(M^2)$ and denote $C^3 = C^1 \oplus \nabla$ and $C^4 = C^2 \oplus \nabla$;
4. Decrypt the ciphertexts: $M^3 = D^K(C^3)$ and $M^4 = D^K(C^4)$, where $D = E^{-1}$;
5. If $M^3 \oplus M^4 = \Delta$ it means you have found a right quartet.

The question which rises is what is the fraction of right quartet among all quartets. In that case,the number of required plaintexts and ciphertexts to obtain at least one right quartet is approximately

$$\frac{k}{p^2}\frac{c}{q^2} = \frac{h}{p^2q^2}$$

where the first term came from the data required to search the differential $\Delta_{in} \xrightarrow{E_0} \Delta^*$ and the differential $\Delta^* \xrightarrow{E_0^{-1}} \Delta_{in}$ and the second one with E_1 and (∇_{out}, ∇^*). The data/time complexity of constructing a boomerang distinguisher is in

$$O\left(\frac{1}{p^2q^2}\right)$$

The Boomerang Attack as presented in this section was developed during the years to obtain higher probability in less computational time. Now, we present the evolution of the Boomerang Attack until the attack we implemented in our work.

2.5.1 Amplified Boomerang Attack

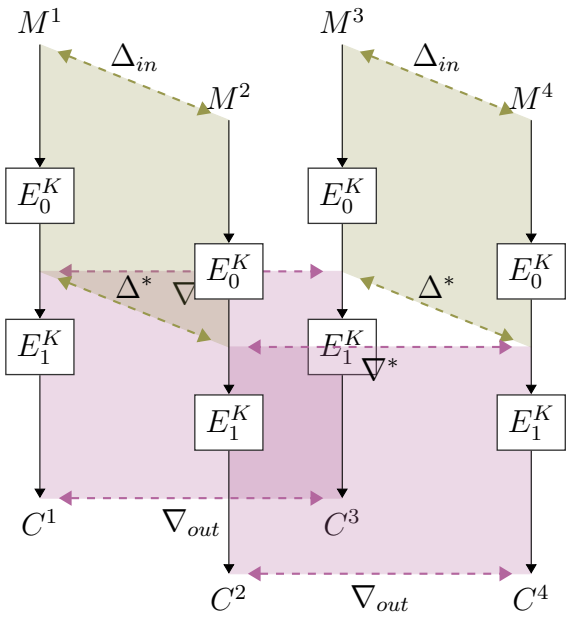

FIGURE 2.6 Amplified Boomerang Attack. $M^1 = M, M^2 = M \oplus \Delta_{in}, C^1 = E_K(M^1), C^2 = E_K(M^2), C^3 = C^1 \oplus \nabla_{out}$ and $C^4 = C^2 \oplus \nabla_{out}$. Compared to the Figure 2.5 it is not present the decrypted function $(E_1^K)^{-1}$. To denote this, the arrows also indicate the encryption direction that, in this case, are all in the "forward" direction.

Many generalizations have followed to reduce the requirement of a decryption oracle, where the decryption oracle is the request to know the decrypted plaintext knowing the ciphertext but without knowing the key used in the decryption process. [KKS00] proposed the **amplified boomerang attack** where the data are only considered in the encrypting direction as shown in Figure 2.6. In this attack, the targeted cipher can be distinguished from a random permutation by querying enough quartets(M^1, M^2, M^3, M^4) with $M^1 \oplus M^2 = \Delta_{in}$ and $M^3 \oplus M^4 = \Delta_{in}$ and verifying whether the corresponding ciphertexts (C^1, C^2, C^3, C^4) satisfy $C^1 \oplus C^3 = \nabla_{out}$ and $C^2 \oplus C^4 = \nabla_{out}$.

In this case, to evaluate the probability, [KKS00], tries to approximate the probability of the boomerang:

$$\mathbb{P}(\Delta_{in} \xrightarrow{E_0} \Delta^*_{fix})^2 \cdot \sum_{\nabla^*} \mathbb{P}(\nabla^* \xrightarrow{E_1} \nabla_{out})^2 \tag{2.12}$$

2.5.2 Rectangle Attack

This attack was further refined by [BDK01] and called **rectangle attack**. In their work, they made two improvements. The first extends the approximation of the boomerang probability, since the probability could increase if we not consider a differential with high probability but also many differentials with small probabilities:

$$\sum_{\Delta^*} \mathbb{P}(\Delta_{in} \xrightarrow{E_0} \Delta^*)^2 \cdot \sum_{\nabla^*} \mathbb{P}(\nabla^* \xrightarrow{E_1} \nabla_{out})^2$$

Then, they switch the focus on the conditions to a right quartet, i.e. they can extend the search of right quartets even with different differentials for (M^1, M^2) and for (M^3, M^4). So, the approximation of the boomerang probability is:

$$\sum_{\Delta^*} \left(\mathbb{P}(\Delta^1_{in} \xrightarrow{E_0} \Delta^*) \mathbb{P}(\Delta^2_{in} \xrightarrow{E_0} \Delta^*) \right) \cdot \sum_{\nabla^*} \left(\mathbb{P}(\nabla^* \xrightarrow{E_1} \nabla^1_{out}) \mathbb{P}(\nabla^* \xrightarrow{E_1} \nabla^2_{out}) \right)$$

We can notice that again the condition 2.10 is satisfied, since we are requiring that:

$$\begin{aligned} M^1 \oplus M^2 = \Delta^1_{in} &\xrightarrow{E_0} \Delta^* = X^1 \oplus X^2 \\ M^3 \oplus M^4 = \Delta^2_{in} &\xrightarrow{E_0} \Delta^* = X^3 \oplus X^4 \\ X^1 \oplus X^3 = \nabla^* &\xrightarrow{E_1} \nabla^1_{out} = C^1 \oplus C^3 \end{aligned}$$

So, to close the boomerang, we have to check that

$$X^2 \oplus X^4 = \nabla^*$$

but that is for free since the other two conditions on the X^i and so, we can take the highest differential characteristic that lead to $X^2 \oplus X^4 = \nabla^* \xrightarrow{E_1} \nabla^2_{out} = C^2 \oplus C^3$. It is possible to be even more general and use this approximation:

$$\sum_{\Delta_1^*, \Delta_2^*} \left(\mathbb{P}(\Delta^1_{in} \xrightarrow{E_0} \Delta_1^*) \mathbb{P}(\Delta^2_{in} \xrightarrow{E_0} \Delta_2^*) \cdot \sum_{\nabla^*} \mathbb{P}(\nabla^* \xrightarrow{E_1} \nabla_{out}) \mathbb{P}(\nabla^* \oplus \Delta_1^* \oplus \Delta_2^* \xrightarrow{E_1} \nabla_{out}) \right) \tag{2.13}$$

where we have the right quartets, with the conditions:

$$M^1 \oplus M^2 = \Delta^1_{in} \xrightarrow{E_0} \Delta_1^* = E_0(M^1) \oplus E_0(M^2)$$
$$M^3 \oplus M^4 = \Delta^2_{in} \xrightarrow{E_0} \Delta_2^* = E_0(M^3) \oplus E_0(M^4)$$

If $\nabla^* = E_0(M^1) \oplus E_0(M^3)$ we have that

$$\nabla^* \oplus \Delta_1^* \oplus \Delta_2^* = E_0(M^2) \oplus E_0(M^4)$$

in this way the condition 2.10 is satisfied. Then, we ask that those two input of the E_1 differentials are going in the same output $\nabla_{out} = C^1 \oplus C^3 = C^2 \oplus C^4$.

If we have m pairs with difference Δ_{in}, a fraction of about p of them satisfies the differential characteristic for E_0. Thus, we have about mp pairs with output difference Δ^* in the input of E_1, giving $\binom{mp}{2}$ (all possible pairs without repetition and without the order matters, i.e. a simple combination) quartets consisting of two such pairs. Assuming that the intermediate encryption values distribute uniformly over all the possible values, then with probability 2^{-n} we can get $E_0(M^1)$ and $E_0(M^3)$ such that its XOR is ∇^*, but once it occurs we have for free another pair ($E_0(M^2)$,$E_0(M^4)$). Again, we can switch the order, i.e. using $E_0(M^3)$ with $E_0(M^2)$, so we have two ways to use them as a quartet with a probability of 2^{-n+1}. Hence, we have $\binom{mp}{2} \cdot 2^{-n+1}$ possible quartets which may satisfy our requirements. Each of the pairs satisfy the second characteristic E_1 with probability q, but since we have two of them the right possible quartets given from these are:

$$\binom{mp}{2} \cdot 2^{-n+1} \cdot q^2 = \frac{mp(mp-1)}{2} \cdot 2^{-n+1} \cdot q^2 \approx m^2 2^{-n} \cdot (pq)^2$$

For a random permutation, starting with the fixed difference Δ_{in}, the expected number of quartets is $\binom{m}{2} 2^{-n} \cdot 2^{-n} \approx m^2 2^{-2n}$. Hence, if

$$m^2 2^{-n} \cdot (pq)^2 > m^2 2^{-2n} \iff (pq)^2 > 2^{-n}$$

It means that we could count more right quartet that uniformly random noise. Hence, we can distinguish E from a random permutation if

$$p^2q^2 \gg 2^{-n}$$

To produce $2^n(pq)^{-2}$ quartets of ciphertexts, we need $2^{\frac{n}{2}}(pq)^{-1}$ plaintext pairs, so we have to encrypt $4 \cdot 2^{\frac{n}{2}}(pq)^{-1}$ different chosen plaintexts. Although we have to check $O(2^n(pq)^{-2})$ ciphertext quartets, by using a hash table the time complexity of a rectangle distinguisher can be reduced to $O(2^{\frac{n}{2}}(pq)^{-1})$. Thus, the data and time complexity of a rectangle distinguisher is in

$$\mathcal{O}(2^{\frac{n}{2}}(pq)^{-1})$$

2.5.3 Sandwich Attack

Dunkelman *et al.* [DKS14] considered another variant of the Boomerang Attack: **the Sandwich Attack**, where the cipher is cut into three (and no more two) parts $E = E_1 \circ E_m \circ E_0$ to capture what is happening in the middle, as shown in Figure 2.7

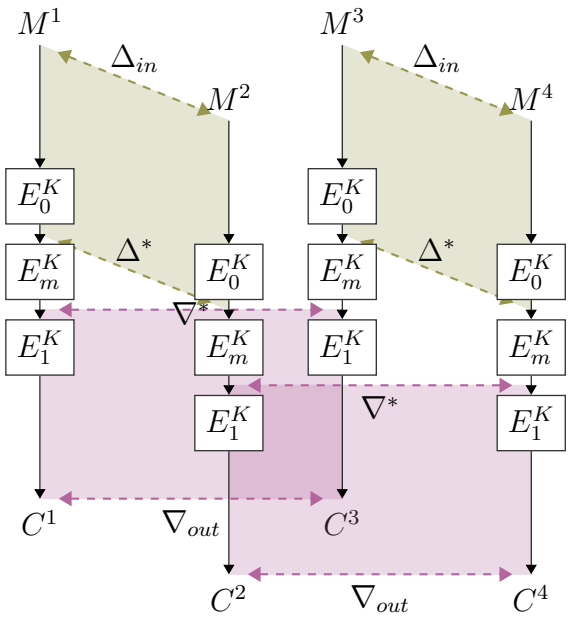

FIGURE 2.7 Sandwich Attack. $M^1 = M, M^2 = M \oplus \Delta_{in}, C^1 = E_K(M^1), C^2 = E_K(M^2), C^3 = C^1 \oplus \nabla_{out}$ and $C^4 = C^2 \oplus \nabla_{out}$.

In this case, we add one ore multiple round in the middle part so that the condition

2.10 change in this way:

$$X^1 \oplus X^2 = \Delta^* \ \wedge \ Y^1 \oplus Y^3 = \nabla^* \ \wedge \ Y^2 \oplus Y^4 = \nabla^* \tag{2.14}$$

where $X^i = E_0(M^i)$ and $Y^i = E_m(X^i)$.

Definition 2.17 (Rectangle Differential)**.** Let $E(\cdot)$ be the encryption function with r rounds, E_0 be the first part of the cipher, with r_0 rounds, E_m the middle part, with r_m rounds and E_1 the last part, with r_1 rounds such that $r_0 + r_m + r_1 = r$ such that

$$E = E_1 \circ E_m \circ E_0$$

Then,

- An $r_0 + r_m + r_1$**-round sandwich differential** is a couple (Δ, ∇), where Δ is the difference plaintexts of E_0 and where ∇ is the difference ciphertexts of E_1 and the middle part is connected according 2.14.
- A $r_0+r_m+r_1$**-round sandwich differential trail** or **characteristic** is a quartet $(\Delta_{in}, \Delta^*, \nabla^*, \nabla_{out})$ such that (Δ_{in}, Δ^*) is a differential for E_0 and (∇^*, ∇_{out}) is a differential for E_1^{-1} and the middle part is connected according 2.14.

The probability of the sandwich differential trail, using the Markov Assumption, can be approximate as:

$$p_{sandwich} \approx \mathbb{P}(\Delta_{in} \xrightarrow{E_0} \Delta^*)^2 \cdot \mathbb{P}(\nabla^* \xrightarrow{E_1} \nabla_{out})^2 \cdot r(\Delta^*, \nabla^*) = p^2 q^2 r \tag{2.15}$$

where p and q are, respectively, the differential characteristic probability for E_0 and E_1, while r, called the **penalty** probability, is:

$$\begin{aligned} r &= r(\Delta^*, \nabla^*) \\ &= \mathbb{P}(X^3 \oplus X^4 = \Delta^* | X^1 \oplus X^2 = \Delta^* \wedge Y^1 \oplus Y^3 = \nabla^* \wedge Y^2 \oplus Y^4 = \nabla^*) \\ &= \mathbb{P}\left(E_m^{-1}(E_m(X^1) \oplus \nabla^*) \oplus E_m^{-1}(E_m(X^2) \oplus \nabla^*) = \Delta^* | X^1 \oplus X^2 = \Delta^*\right) \end{aligned} \tag{2.16}$$

Without any further assumptions on E_m, the penalty probability should be expected near 2^{-n} (for an n-bits block iterated cipher). However, as observed in [BK09, BDCD03], in some cases E_0 and E_1 can be chosen such that the probability r is 1. In their work, this is treated as a trick allowing to increase the probability of the distinguisher.

The probability of the sandwich differential can be obtain making the sums over Δ^*

and ∇^* in 2.15,

$$\sum_{\Delta^*,\nabla^*} \mathbb{P}(\Delta_{in} \xrightarrow{E_0} \Delta^*)^2 \cdot \mathbb{P}(\nabla^* \xrightarrow{E_1} \nabla_{out})^2 \cdot r(\Delta^*, \nabla^*)$$

But as we already saw for the differentials, the sum is unfeasible to evaluate in practice, so when we talk about the probability of a sandwich differential we mean the probability of its trail even if it is just a lower bound. Following the same steps we have done to find the right quartets for the amplified boomerang, we have that the possible right quartets in the sandwich attack is:

$$\binom{mp}{2} \cdot 2^{-n+1} \cdot r' \cdot q^2 \approx m^2 p^2 2^{-n} q^2 r'$$

where

$$r' = \mathbb{P}(Y^2 \oplus Y^4 = \nabla^* | X^1 \oplus X^2 = \Delta^* \text{ and } Y^1 \oplus Y^3 = \nabla^* \text{ and } X^1 \oplus X^3 = \Delta^*)$$

In this case we are looking for shorter differential in E_0 or/and E_1 i.e. the probability p or/and q is smaller. Notice that $r' = r$ when we have Feistel Cipher.

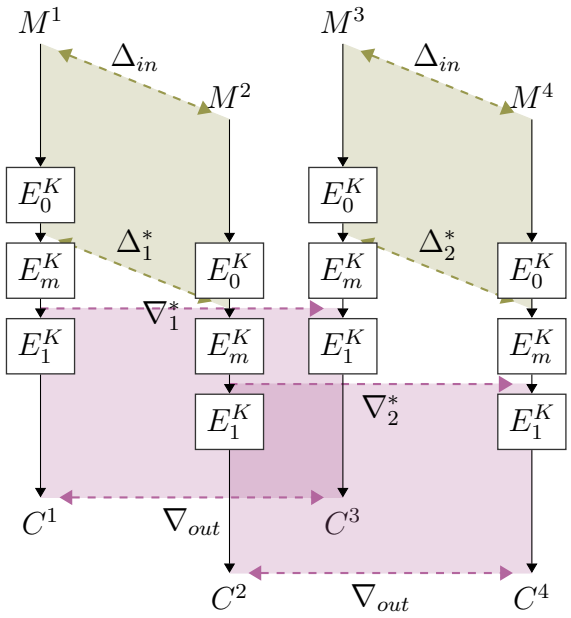

FIGURE 2.8 General Sandwich Attack. $M^1 = M, M^2 = M \oplus \Delta_{in}, C^1 = E_K(M^1), C^2 = E_K(M^2), C^3 = C^1 \oplus \nabla_{out}$ and $C^4 = C^2 \oplus \nabla_{out}$.

Since the intermediate differences Δ^* and ∇^* can take arbitrary values in the sandwich

distinguisher 2.15, we can consider the clustering effect (see Figure 2.8). Therefore, a more accurate formula to compute the probability of sandwich distinguisher is:

$$\sum_{\Delta_1^*,\Delta_2^*,\nabla_1^*,\nabla_2^*} \mathbb{P}(\Delta_{in} \xrightarrow{E_0} \Delta_1^*)\mathbb{P}(\Delta_{in} \xrightarrow{E_0} \Delta_2^*)\mathbb{P}(\nabla_1^* \xrightarrow{E_1} \nabla_{out})\mathbb{P}(\nabla_2^* \xrightarrow{E_1} \nabla_{out}) \cdot r$$
$$r = r(\Delta_1^*, \Delta_2^*, \nabla_2^*, \nabla_2^*)$$
$$= \mathbb{P}\left(E_m^{-1}(E_m(X^1) \oplus \nabla_1^*) \oplus E_m^{-1}(E_m(X^2) \oplus \nabla_2^*) = \Delta_2^* | X^1 \oplus X^2 = \Delta_1^*\right)$$

Remark. *There are also many results of impossibility [Mur11, Kir15, BK09] that induced a full reconsideration of what is a Boomerang attack where the independence of the rounds could no more be considered.*

2.5.4 BCT, UBCT, LBCT and EBCT

We denote the middle part, i.e. the link between the E_0 and E_1 characteristics as **Boomerang Switch**, [BK09, BDCD03, Mur11]. We can have different types of Boomerang Switches (Ladder, S-Box, Feistel), later we identify a way to unify them. The seminal work proposed at Eurocrypt 2018 [CHP+18] introduced the Boomerang Connectivity Table (BCT) framework to analyze E_m when it is composed of one SPN round.

Definition 2.18 (Boomerang Connectivity Table)**.** Let $\pi_S : \{0,1\}^m \to \{0,1\}^m$ be the non-linear (invertible) function, $\Delta^* \in \{0,1\}^m$ and $\nabla^* \in \{0,1\}^m$ be the entries of the table, then we can define the **Boomerang Connectivity Table** (BCT) as:

$$\mathcal{B}_{\pi_S}(\Delta^*, \nabla^*) = \left|\left\{x \in \{0,1\}^m | \pi_S^{-1}\left(\pi_S(x) \oplus \nabla^*\right) \oplus \pi_S^{-1}\left(\pi_S(x \oplus \Delta^*) \oplus \nabla^*\right) = \Delta^*\right\}\right|$$

The BCT represents the observations by [Mur11] and [BK09] in a unified manner.

- **Incompatibility.** $(\Delta^*, \nabla^*)$is incompatible when the corresponding entry in the BCT is 0.
- **Ladder Switch.** It corresponds to the first row $\Delta^* = 0$ and the first column $\nabla^* = 0$ of the BCT, in which either one of the input or output difference is zero, while the other is non-zero, gives probability 1.
- **S-box Switch.** It corresponds to the claim that a DDT entry with non-zero value v would imply that the corresponding BCT entry is v. While this is correct in some cases, [CHP+18] show that the value of the BCT can in fact be larger than v owing to the new switching effect. However, at least the effect of S-box switch is always guaranteed.

As shown in [WP19] this approach could be useless in the case of many rounds in the middle. Then, many papers generalizing this approach to more rounds [WP19, SQH19,

DDV20a] or to different ciphers [BHL+20, WWS23] have finally emerged. To denote the next tables, we use the notation of [DDV20a].

Definition 2.19 (UBCT, LBCT, EBCT)**.** Let $\pi_S : \{0,1\}^m \to \{0,1\}^m$ be the non-linear (invertible) function, we can define the variants of the BCT.

Let $\Delta^*_{in}, \Delta^*_{out} \in \{0,1\}^m$ and $\nabla^* \in \{0,1\}^m$ be the entries of the table, then **Upper Boomerang Connectivity Table** (UBCT) as:

$$\mathcal{U}_{\pi_S}(\Delta^*_{in}, \Delta^*_{out}, \nabla^*) = \left| \left\{ x \in \{0,1\}^m \middle| \begin{matrix} \pi_S^{-1}\left(\pi_S(x) \oplus \nabla^*\right) \oplus \pi_S^{-1}\left(\pi_S(x \oplus \Delta^*_{in}) \oplus \nabla^*\right) = \Delta^*_{in} \\ \pi_S(x) \oplus \pi_S(x \oplus \Delta^*_{in}) = \Delta^*_{out} \end{matrix} \right\} \right|$$

Let $\Delta^* \in \{0,1\}^m$ and $\nabla^*_{in}, \nabla^*_{out} \in \{0,1\}^m$ be the entries of the table, then **Lower Boomerang Connectivity Table** (LBCT) as:

$$\mathcal{L}_{\pi_S}(\Delta^*, \nabla^*_{in}, \nabla^*_{out}) = \left| \left\{ x \in \{0,1\}^m \middle| \begin{matrix} \pi_S^{-1}\left(\pi_S(x) \oplus \nabla^*_{out}\right) \oplus \pi_S^{-1}\left(\pi_S(x \oplus \Delta^*) \oplus \nabla^*_{out}\right) = \Delta^* \\ \pi_S(x) \oplus \pi_S(x \oplus \nabla^*_{in}) = \nabla^*_{out} \end{matrix} \right\} \right|$$

Let $\Delta^*_{in}, \Delta^*_{out} \in \{0,1\}^m$ and $\nabla^*_{in}, \nabla^*_{out} \in \{0,1\}^m$ be the entries of the table, then **Extended Boomerang Connectivity Table** (EBCT) as:

$$\mathcal{E}_{\pi_S}(\Delta^*_{in}, \Delta^*_{out}, \nabla^*_{in}, \nabla^*_{out}) = \left| \left\{ x \in \{0,1\}^m \middle| \begin{matrix} \pi_S^{-1}\left(\pi_S(x) \oplus \nabla^*_{out}\right) \oplus \pi_S^{-1}\left(\pi_S(x \oplus \Delta^*_{in}) \oplus \nabla^*_{out}\right) = \Delta^*_{in} \\ \pi_S(x) \oplus \pi_S(x \oplus \Delta^*_{in}) = \Delta^*_{out} \\ \pi_S(x) \oplus \pi_S(x \oplus \nabla^*_{in}) = \nabla^*_{out} \end{matrix} \right\} \right|$$

These tables underline the possibility to propagate the input difference for more rounds. In particular, the second request of the UBCT regards the possibility to propagate the difference also for the upper part in order to continue the multiple rounds. In the Feistel ciphers, [BHL+20] noticed that, fixed the input difference $\Delta^* = \Delta^*_L || \Delta^*_R$ and the output difference $\nabla^* = \nabla^*_L || \nabla^*_R$ for the condition 2.14, given F as round function and X^i be the inputs of the middle round and Y^i be the respectively output of the middle round ($Y^i = Y^i_L || Y^i_R = F(X^i_L) \oplus X^i_R || X^i_L$), in the left part the difference is propagated for free, i.e.:

$$\begin{aligned} X^3_L \oplus X^4_L &= (X^3_L \oplus X^1_L) \oplus (X^1_L \oplus X^2_L) \oplus (X^2_L \oplus X^4_L) \\ &= (Y^3_R \oplus Y^1_R) \oplus (X^1_L \oplus X^2_L) \oplus (Y^2_R \oplus Y^4_R) \\ &= \nabla^*_R \oplus \Delta^*_L \oplus \nabla^*_R \\ &= \Delta^*_L \end{aligned}$$

where the second equality comes from the structure of the Feistel cipher (the right part of one round output correspond to the left part of the input). In the right part, it is

possible to notice that

$$Y_L^i = F(X_L^i) \oplus X_R^i \implies X_R^i = Y_L^i \oplus F(X_L^i)$$

and

$$\begin{aligned} X_L^i = Y_R^i &\implies X_L^1 \oplus X_L^3 = Y_R^1 \oplus Y_R^3 = \nabla_R^* = Y_R^2 \oplus Y_R^4 = X_L^2 \oplus X_L^4 \\ &\implies \begin{cases} X_L^3 = X_L^1 \oplus \nabla_R^* \\ X^4 = X_L^2 \oplus \nabla_R^* = X_L^1 \oplus \Delta_L^* \oplus \nabla_R^* \end{cases} \end{aligned}$$

Then,

$$\begin{aligned} X_R^3 \oplus X_R^4 =& (F(X_L^3) \oplus Y_L^3) \oplus (F(X_L^4) \oplus Y_L^4) \\ =& F(X_L^1 \oplus \nabla_R^*) \oplus Y_L^3 \oplus F(X_L^1 \oplus \Delta_L^* \oplus \nabla_R^*) \oplus Y_L^4 \\ =& F(X_L^1 \oplus \nabla_R^*) \oplus Y_L^1 \oplus \nabla_L^* \oplus F(X_L^1 \oplus \Delta_L^* \oplus \nabla_R^*) \oplus Y_L^2 \oplus \nabla_L^* \\ =& F(X_L^1 \oplus \nabla_R^*) \oplus F(X_L^1) \oplus X_R^1 \oplus F(X_L^1 \oplus \Delta_L^* \oplus \nabla_R^*) \oplus F(X_L^2) \oplus X_R^2 \\ =& F(X_L^1 \oplus \nabla_R^*) \oplus F(X_L^1) \oplus X_R^1 \oplus F(X_L^1 \oplus \Delta_L^* \oplus \nabla_R^*) \oplus F(X_L^1 \oplus \Delta_L^*) \oplus X_R^1 \oplus \Delta_R^* \\ =& F(X_L^1 \oplus \nabla_R^*) \oplus F(X_L^1) \oplus F(X_L^1 \oplus \Delta_L^* \oplus \nabla_R^*) \oplus F(X_L^1 \oplus \Delta_L^*) \oplus \Delta_R^* \end{aligned}$$

Then,

$$X_R^3 \oplus X_R^4 = \Delta_R^* \iff F(X_L^1 \oplus \nabla_R^*) \oplus F(X_L^1) \oplus F(X_L^1 \oplus \Delta_L^* \oplus \nabla_R^*) \oplus F(X_L^1 \oplus \Delta_L^*) = 0$$

Since in the round function we have usually one non-function, called π_S, it is possible to rewrite the previous condition for the Feistel ciphers as:

$$\pi_S(x) \oplus \pi_S(x \oplus \nabla^*) \oplus \pi_S(x \oplus \Delta^*) \oplus \pi_S(x \oplus \Delta^* \oplus \nabla^*) = 0$$

This leads to the definitions in [BHL+20] of these new tables:
Definition 2.20 (FBCT, FUBCT, FLBCT, FEBCT)**.** Let $\pi_S : \{0,1\}^m \to \{0,1\}^m$ be the non-linear function. Let $\Delta^* \in \{0,1\}^m$ and $\nabla^* \in \{0,1\}^m$ be the entries of the table, then we can define the **Feistel Boomerang Connectivity Table** (FBCT) as:

$$\mathcal{FB}_{\pi_S}(\Delta^*, \nabla^*) = |\{x \in \{0,1\}^m | \pi_S(x) \oplus \pi_S(x \oplus \nabla^*) \oplus \pi_S(x \oplus \Delta^*) \oplus \pi_S(x \oplus \Delta^* \oplus \nabla^*) = 0\}|$$

Let $\Delta_{in}^*, \Delta_{out}^* \in \{0,1\}^m$ and $\nabla^* \in \{0,1\}^m$ be the entries of the table, then **Feistel**

Upper Boomerang Connectivity Table (FUBCT) as:

$$\mathcal{FU}_{\pi_S}(\Delta^*_{in}, \Delta^*_{out}, \nabla^*) = \left| \left\{ x \in \{0,1\}^m \middle| \begin{matrix} \pi_S(x) \oplus \pi_S(x \oplus \nabla^*) \oplus \pi_S(x \oplus \Delta^*_{in}) \oplus \pi_S(x \oplus \Delta^*_{in} \oplus \nabla^*) = 0 \\ \pi_S(x) \oplus \pi_S(x \oplus \Delta^*_{in}) = \Delta^*_{out} \end{matrix} \right\} \right|$$

Let $\Delta^* \in \{0,1\}^m$ and $\nabla^*_{in}, \nabla^*_{out} \in \{0,1\}^m$ be the entries of the table, then **Feistel Lower Boomerang Connectivity Table** (FLBCT) as:

$$\mathcal{FL}_{\pi_S}(\Delta^*, \nabla^*_{in}, \nabla^*_{out}) = \left| \left\{ x \in \{0,1\}^m \middle| \begin{matrix} \pi_S(x) \oplus \pi_S(x \oplus \nabla^*_{out}) \oplus \pi_S(x \oplus \Delta^*) \oplus \pi_S(x \oplus \Delta^* \oplus \nabla^*_{out}) = 0 \\ \pi_S(x) \oplus \pi_S(x \oplus \nabla^*_{in}) = \nabla^*_{out} \end{matrix} \right\} \right|$$

Let $\Delta^*_{in}, \Delta^*_{out} \in \{0,1\}^m$ and $\nabla^*_{in}, \nabla^*_{out} \in \{0,1\}^m$ be the entries of the table, then **Feistel Extended Boomerang Connectivity Table** (FEBCT) as:

$$\mathcal{FE}_{\pi_S}(\Delta^*_{in}, \Delta^*_{out}, \nabla^*_{in}, \nabla^*_{out}) = \left| \left\{ x \in \{0,1\}^m \middle| \begin{matrix} \pi_S(x) \oplus \pi_S(x \oplus \nabla^*_{out}) \oplus \pi_S(x \oplus \Delta^*_{in}) \oplus \pi_S(x \oplus \Delta^*_{in} \oplus \nabla^*_{out}) = 0 \\ \pi_S(x) \oplus \pi_S(x \oplus \Delta^*_{in}) = \Delta^*_{out} \\ \pi_S(x) \oplus \pi_S(x \oplus \nabla^*_{in}) = \nabla^*_{out} \end{matrix} \right\} \right|$$

The tables presents different properties showed in [BHL+20, WWS23].

Chapter 3

Optimization Problem and Automatic Tools

As discussed in Chapter 2, the security evaluation of modern block ciphers against differential cryptanalysis requires the identification of differential characteristics with high probabilities [BS91]. However, as cryptographic designs have become more complex to resist such attacks, the manual search for these characteristics has become a tedious, error-prone, and often infeasible task. Consequently, the field has shifted towards **automated cryptanalysis**, leveraging powerful solvers to evaluate security bounds efficiently.

This chapter explores the mathematical foundations and software tools that enable this automation. Historically, the move toward automated search began with Matsui's seminal work in 1994 [Mat94], where he proposed an iterative algorithm to find the best differential characteristic for the DES cipher [Cop94]. This approach was essentially based on the Markov Independence Hypothesis. Matsui's method was later refined by Biryukov *et al.* [BN10], who introduced a branch-and-bound search algorithm, and by Fouque *et al.* in 2013 [FJP13], who modeled the search as a shortest path problem in a directed acyclic graph.

Today, the state-of-the-art approach involves modeling the cryptographic primitive and the attack as a standard optimization problem. In **Section 3.1**, we provide the basic definitions of these problems:

- **SAT** (Satisfiability);
- **SMT** (Satisfiability Modulo Theories);

- **MILP** (Mixed Integer Linear Programming);

and then give more details on:

- **CP** (Constraint Programming) and **COP** (Constraint Optimization Problems).

Building on these solvers, several automated tools and libraries have been developed to assist cryptanalysits. **Section 3.2** reviews prominent libraries such as `CryptoSMT` [Köl20], which searches for differential and linear trails using SAT/SMT solvers, and the more recent Claasp (Cryptographic Library for the Automated Analysis of Symmetric Primitives)[BGG+24], which offers a modular approach to generating models for various solvers.

Finally, **Section 3.3** is dedicated to Tagada (Tool for Automatic Generation of Abstraction-based Differential Attacks) [LDLS21, DDG+24]. We detail its methodology, which splits the search into two steps: finding an optimal truncated differential on an abstracted model of the cipher, and subsequently instantiating it with specific differences. We explain how Tagada utilizes Directed Acyclic Graphs (DAGs) to represent the cipher and employs logic modelization to translate cryptographic operators into constraints solvable by Minizinc.

3.1 Optimization Problem

The core of automated cryptanalysis lies in the ability to translate the algebraic and non-linear properties of a block cipher into a mathematical model that can be interpreted by a solver. To find the best differential characteristic, we effectively transform a cryptanalytic problem into a *Constraint Satisfaction Problem* (CSP) or a *Constraint Optimization Problem* (COP). In a CSP, the goal is simply to find a feasible assignment of variables (e.g., a valid differential trail) that satisfies all the rules of the cipher. In a COP, we go a step further: among all valid assignments, we seek the one that minimizes or maximizes a specific objective function, such as minimizing the number of active S-boxes or maximizing the probability of the characteristic. In this section, we provide the theoretical background for the four main paradigms used to model these problems in modern cryptography:

- **SAT (Boolean Satisfiability):** Problems defined purely in propositional logic, ideal for bit-level analysis.
- **SMT (Satisfiability Modulo Theories):** An extension of SAT that allows for more expressive constraints, such as bit-vector arithmetic, which naturally maps to word-oriented ciphers.
- **MILP (Mixed Integer Linear Programming):** A framework for optimization where constraints are linear inequalities and variables are integers, widely used to bound the number of active non-linear components.

- **CP (Constraint Programming):** A declarative paradigm that focuses on the relationships between variables and uses constraint propagation to prune the search space efficiently.

Understanding the strengths and limitations of each approach is crucial for selecting the appropriate tool for a given cryptographic primitive.

3.1.1 SAT

A Satisfiability (SAT) problem is defined as:

- Let a set of boolean variable $\{X_i\}_i$ i.e. $X_i \in \{0,1\}$, with 0 it is indicate the boolean value $false$ and with 1 the boolean value $true$;
- Given a boolean/logic function or formula $F(X_1, \ldots, X_n)$ e.g. $F(X_1, X_2, X_3) = (X_1 \vee X2) \wedge (\neg X_3)$. F could be a composition of different boolean functions e.g. $F(X_1, \ldots, X_n) = F_1(X_1, \ldots, X_n) \wedge F_2(X_1, \ldots, X_n) \cdots \wedge F_m(X_1, \ldots, X_n)$;
- Then, the aim is to find a valid instantiation $\{\bar{X}_i\}_i$ for the formula F such that $F(\bar{X}_1, \ldots, \bar{X}_n) = 1$ i.e. it is true, e.g. a solution for the F before is $(\bar{X}_1, \bar{X}_2, \bar{X}_3) = (0,1,1)$, in this case $(0,1,1)$ *satisfies* F.

The SAT Problem was proven to be NP-hard problem [Coo71].

3.1.2 SMT.

A Satisfaiability Modulo Theory (SMT) problem is an extension of a SAT problem. In particular, in the SAT problem we ask if an instantiation of the variables is true or false for a logic formula, while in the SMT problem we can consider theories other than the boolean logic. In other words, a SMT problem satisfy if a logic formula with arithmetic or, defined according the chosen theory, constrains is satisfiable, i.e., if there exist a model that let it be true in a certain theory.

- Fix a theory $\mathcal{T}$, e.g., the arithmetic on the integer;
- Let a set of variables $\{X_i\}_i$ be in theory i.e. $X_i \in \mathbb{Z}$;
- Let $F(X_1, \ldots, X_n) = F_1(X_1, \ldots, X_n) \wedge F_2(X_1, \ldots, X_n) \cdots \wedge F_m(X_1, \ldots, X_n)$ be the logic formula to satisfy, with F_j arithmetic formulas in the theory e.g. $F(X_1, X_2) = (X_1 > 3) \wedge (X_2 = X_1 + 2) \wedge (X_2 \leq 10)$;
- Then, a valid instantiation is $(\bar{X}_1, \ldots, \bar{X}_n)$ such that $X_i \in \mathbb{Z}$ and $F(\bar{X}_1, \ldots, \bar{X}_n) = true$ and we can say that $(\bar{X}_1, \ldots, \bar{X}_n)$ satisfies F e.g. $(\bar{X}_1, \bar{X}_2) = (4,6)$ satisfies F.

In this case, we cannot affirm if the SMT problems are NP-hard or not since it depends on the theory we are using, e.g. if $\mathcal{T}$ is the logic theory than it is an NP problem, but if we are using the arithmetic on $\mathbb{R}$ than it is a P problem.

3.1.3 MILP.

Mixed Integer Linear Programming is a constraint problem of linear optimization where some of the unknown variables are integers and the constraints are linear inequalities. In other words, we are trying to solve this problem:

$$\min_{x \in S} c^T x \quad \text{or} \quad \max_{x \in S} c^T x \quad \text{where } S = \left\{ x \in \mathbb{R}^n \;\middle|\; \begin{cases} Ax \leq b \\ x_i \in \mathbb{Z} \quad \forall i \in \mathcal{I} \end{cases} \right\}$$

where $c \in \mathbb{Z}^n, b \in \mathbb{Z}^m$, $A \in \mathbb{Z}^{m \times n}$ and $\mathcal{I} \subseteq \{1, \ldots, n\}$ is a subset of the indices. MILP is proven to be an NP-hard problem [GJ90].

3.1.4 Constraint Programming

Constraint Programming (CP) regards the field of all the previous problem we explained before. A Constraint Satisfaction Problem (CSP) is a tuple (X, D, C), where:

- $X = \{x_1, \ldots, x_n\}$ is the set of variables e.g. $X = \{x, y, z\}$;
- $D = \{d_1, \ldots, d_n\}$ is the set of domains i.e. each d_i contains all the possible values for x_i e.g. $d_x = d_y = d_z = \{0, 1\}$;
- $C = \{c_1, \ldots, c_m\}$ is the set of the constraints e.g. $c : x + y = z$.

Definition 3.1 (Constraint)**.** Let X be a set of variable and D a set of the domains of the variables in X, $S = (x_{i_1}, \ldots, x_{i_k})$, called the **scope**, be a subset of $X \times \cdots \times X$, and $R \subseteq d_{i_1} \times \cdots d_{i_k}$, called the **relation**, be a subset of $D \times \cdots \times D$. Then, a **constraint** c is a pair (S, R). The **arity** $\mathcal{A}$ of a constraint is the size of its scope i.e. $\mathcal{A} = len(S)$.

Example 3.2 (Simple Constraint)**.** Taking the example of before $S = (x, y, z)$ and $R = \{(0, 0, 0), (1, 0, 1), (0, 1, 1)\}$. Then, we can say that a tuple $\tau \in d_{i_1} \times \cdots d_{i_k}$ *satisfies* c if and only if $\tau \in R$ e.g. $\tau \in \{0, 1\} \times \{0, 1\} \times \{0, 1\}$ satisfies $x + y = z$ if and only if $\tau \in \{(0, 0, 0), (1, 0, 1), (0, 1, 1)\}$.

There are two ways to describe a constraint, one is the **extensional representation** i.e. in which we describe all the allowed tuples e.g. $((x, y, z), \{(0, 0, 0), (1, 0, 1), (0, 1, 1)\})$; the other one is the more compactly description: **intensional representation** e.g. $x, y, z \in \{0, 1\}, x + y = z$.

A constraint c with scope S is determined by a function

$$F_c : S \to \{false, true\}$$

The **satisfying tuples** are exactly those that give 1 i.e. *true*, indeed can see that in

the example of before

$$F_c(x, y, z) = \begin{cases} 1 & \text{if } x + y = z \text{ or if } c \text{ is satisfied} \\ 0 & \text{otherwise of if } c \text{ is not satisfied} \end{cases}$$

hence, $F_c(0, 0, 0) = true$.

Given a CP with variables $X = x_1, x_2, ..., x_n$, domains $D = d_1, d_2, ..., d_n$ and constraints C, let

$$I = (x_1 \to v_1, \ldots, x_n \to v_n)$$

be an **instantiation** of values such that v_i is the value chosen for the variable x_i. Let $Y \subseteq X$ be $\{x_{i_1}, \ldots, x_{i_k}\}$. Then, the **projection** of I on Y is

$$proj_Y(I) = (x_{i_1} \to v_{i_1}, \ldots, x_{i_k} \to v_{i_k})$$

An instantiation is **valid** if

$$v_i \in D_i \forall i \tag{3.1}$$

Moreover, if I is valid and $Y = X$ then I is **total** otherwise **partial**.

A total instantiation I is **consistent** if $\forall c_i = (S_i, R_i)$, the projection of I on S_i belongs to R_i.

Definition 3.3 (Solution)**.** A **solution** is a total and consistent instantiation. In other words, it is an instantiation such that:

- Domains are respected: $v_i \in d_i \forall i$ and
- The instantiation I satisfies all constraints in C.

i.e. $F_{c_i}(v_{i_1}, \ldots, v_{i_k}) = true \ \forall c_i \in C$.

Remark. *SAT is a special case of CP that only accepts Boolean variables and Boolean formulae. Those restrictions allow to use dedicated algorithms such as the DPLL [DP60, DLL62] and the CDCL [SS97, MSS99] algorithms which provide very good performances. In differential cryptanalysis, SAT is efficient to compute truncated differential characteristics since they contain a lot of Boolean variables and formulae. The non-Boolean part of the model can be translated into a Boolean one by introducing temporary variables and transforming constraints into Boolean formulae. Even if this can generate an exponential number of variables, it scales pretty well in practice.*

Once the CP has been formally defined, solvers may have two goals:

1. to answer if the CP has a solution or not, which is called a decision problem;
2. to return a solution for the given CP;

In order to find a CP solution, the usual strategy is to apply an algorithm of Branch and Propagate. An iterative algorithm that search the solution continuing instantiating an allowed value for a random variable (see Algorithm 3). The first phase is the choice of

Algorithm 3 `branch_and_propagate`

Require: CP problem (X, D, C) a partial solution $I = (x_{i_1} \to v_{i_1}, \ldots, x_{i_k} \to v_{i_k})$

```
1: fn branch_and_propagate(X, D, C, I)
2: if len(I) = len(X) (I is a complete instantiation) then
3:     return I
4: else
5:     select an unsigned varibale x_i ∈ X
6:     for v_i ∈ D(x_i) do
7:         (X', D', C', I') ← propagate(X, D, C, I ∪ (x_i → v_i))
8:         sol ← branch_and_propagate(X', D', C', I')
9:         if sol ≠ ∅ then
10:            return sol
11:        end if
12:    end for
13:    return ∅
14: end if
```

the variable i.e. branching, and the second phase is the extension of the solution i.e. propagation.

Definition 3.4 (GAC)**.** A constraint $c = (S, R) \in C$ is called **local consistent** or **Generalized Arc Consistent** (**GAC**) [R94] if and only if $\forall x_{i_t} \in S$, $\forall v \in D_{i_t}$, $\forall x_{i_j} \in S \backslash \{x_{i_t}\}$ $\exists v_{i_j} \in D_{i_j}$ such that the partial instantiation $(x_{i_1} \to v_{i_1}, \ldots, x_{i_t} \to v, \ldots, x_{i_k} \to v_{i_k})$ has support in R, i.e. $(v_{i_1}, \ldots, v, \ldots, v_{i_k}) \in R$ or in other words if and only if

$$F_c(v_{i_1}, \ldots, v, \ldots, v_{i_k}) = true$$

If the constraint is local consistent with the instantiation then we are able to propagate the instantiation of that variable. Furthermore, if every constraint $c \in C$ is local consistent with every instantiation, then the instantiation is called **consistent** [MM88] and it is a **solution**.

To propagate an instantiation, Constraint Propagators are algorithms that restrict the domains of variables to enforce a certain form of local consistency or to detect inconsistencies early. They act as the software component connecting the expressiveness of the model with the actual search for solutions. More effective filtering makes it possible to shrink variable domains earlier in the search, which in turn prunes the search tree more efficiently. However, in general, stronger filtering comes at the cost of increased computation time. Therefore, it is important to find a balance between filtering strength and overall solving time. A common strategy to improve filtering is to design constraints with a richer level of knowledge, which is often achieved through the use of **global constraints**. Those are constraint with dynamic arity i.e. they are defined for different dimension of the scope S e.g. `all_different` [R94], `abstract_XOR` [Rou22] or `setSum` [BHHW04]. To check if it is possible to find a solution i.e if the problem is satisfiable `all_different` has polynomial complexity, while the other two are $\mathcal{NP}$-complete.

3.1.5 Constraint Optimization Problem

Constraint Optimization Problem (**COP**) is a CP problem in which we are interested to find the best solution according an **objective function**. Indeed, a COP is a tuple (X, D, C, f) where X, D, C are defined as in the CP problem and f is the **objective function** we want to optimize e.g.:

$$\max_{x \in X} \quad \text{or} \quad \min_{x \in X} f(x_1, \ldots, x_n) \quad \text{such that } x \text{ respect } C$$

Definition 3.5 (Optimal Solution)**.** A solution for the CP $= (X, D, C)$ is an **optimal solution** for the COP $= (X, D, C, f)$ if it maximize or minimize $f(x)$.

Regarding the Branch and Bound algorithm, the problem solving is done in the same way as for the branch and propagate algorithm but adds a cutting technique allowing to prune the search tree earlier and thus limit the number of states to visit. To illustrate how the algorithm works, let us assume that the objective function f must be minimized. The search begins by identifying an initial solution s_0. If no solution exists, the process terminates immediately. This first solution provides an upper bound on the optimal solution s^*, meaning that $f(s^*) \leq f(s_0)$. Once the initial solution has been found, the algorithm continues exploring the search space. At each decision point, a bounding function is used to determine whether the current partial assignment can still lead to a solution better than the best one discovered so far. Whenever a new solution is obtained, the upper bound is updated accordingly, and the search proceeds. This process repeats until no further improving solutions can be identified. At that point, the most recently found solution is guaranteed to be the best possible one.
Remark. *MILP is a special case of COP where the variables are Integer variables, constraints are restricted to linear inequalities and the objective function f is linear. As with SAT, the type of variables and the restriction of constraints allow the use of*

dedicated algorithms such as Cutting-plane [Gom58], the Branch and Bound [LD10, LW66] or the Branch and cut [PR91] methods.

3.2 Automatic Tools

As we saw in section 2.3, every cryptanalysis involves a lot of tedious and rigorous work according the specific cipher in order to find the best characteristic to use. Then, starting from the 1994 work of Matsu [Mat94] the focus on the standard attacks moves to the automatization of the search of such characteristics. The basic idea is, indeed, based on the Markov Independence Hypothesis, i.e., in his work [Mat94], Matsui tried to find the best differential characteristic at round n in an iterative way knowing the best differential characteristic at round $n-1$. His idea was further developed by Biryukov *et al.* [BN10]. They first thought to use a model, a research tree, to make easier the research of differential characteristics of an SPN cipher optimizing the idea of Matsui. In 2013, Foque *et al.* in [FJP13] continued to this idea but representing the cipher as a directed byparted acyclic graph where the nodes are composed by a key and a state and to find the best differential characteristic corresponds to find the shortest path. Finally, we reached the idea of the modern Automated Tools.

Automated tools to support cryptanalysts have become a cornerstone for the design of new primitives. Over time, such tools were made more generic and gathered into libraries. The linear trails library [DEM15] is dedicated to the search for linear characteristics on SPN ciphers. ARX toolkit [Leu12, Leu13] and YAARX [VLoASL24] focus on ARX ciphers, the former testing conditions for trails to be possible, and the latter performing various analysis techniques on the components.

On the algebraic cryptanalysis side, the Automated Algebraic Cryptanalysis tool [Sta10a] tests properties of block and stream ciphers; in particular, it evaluates the randomness of a cipher through Maximum Degree Monomial tests [Sta10b]. Autoguess [HE21] is a tool to automate the technique guess-and-determine. This technique involves making a calculated guess of a subset of the unknown variables, which enables the deduction of the remaining unknowns using the information obtained from the guessed variables and some given relations. In order to automate this technique, SAT/SMT, MILP, and Gröbner basis solvers are used and several new modeling techniques to exploit these solver proposed. For instance, the authors of the library introduce new encodings in CP and SAT/SMT to solve the problem of determining the minimal guess, i.e., the subset of guessed variables from which the remaining variables can be deduced. Autoguess also allows to automate the key-bridging technique. This technique is utilized in key recovery attacks on block ciphers, wherein the attacker seeks to determine the minimum number of sub-key guesses needed to deduce all the involved sub-keys through the

key schedule. The significant contribution of this work lies in integrating key-bridging techniques into tools that were previously only capable of searching for distinguishers. As a result, these enhanced tools can now be utilized as fully automated methods for recovering keys.

CryptoSMT [Köl20] is the first large-scale solver-based library dedicated to cryptanalysis. Based on SMT and SAT solvers, it provides an extensive toolkit, permitting the search for optimal differential and linear trails, the evaluation of the probability of a differential, the search for hash function preimages, and secret key search. Another SMT-based library, based on ArxPy [RLA17] is the CASCADA framework [RR22], which also implements techniques to search for rotational-XOR differentials, impossible-rotational-XOR, but also related-key impossible-differentials, linear approximations, and zero-correlation characteristics. The generated SMT models are expressed through the theory of bit-vectors [BFT16], and follow the general methodology of Mouha and Preneel [MP13] for differential properties, Sasaki's [ST17] technique for impossible differentials, an SMT-based miss-in-the middle search for related-key impossible differentials of ARX ciphers [ARS+22a], and a novel method proposed for zero-probability global properties. If a search can not use the previous methods, then a generic method, based on the constructions of statistical tables, such as the Differential Distribution Table (DDT), is used. Depending on the sizes of the inputs of the block cipher, these generic models could be costly, so they also proposed heuristic models by relaxing the accuracy of their properties; they called them weak models. Finally, their framework implements methods to check the properties mentioned above experimentally.

For Automated Tools we intend the tools that modelize the operators that composed an iterated cipher using logic operators or integer equations/inequalities. This creates another problem to solve: **Satisfaiability** (SAT), **Satisfaiability Modulo Theories** (SMT) **Constrain Programming** (CP) problem and **Mixed Integer Linear Programming** (MILP) problem.

Claasp. Bellini *et al.* introduced Claasp (Cryptographic Library for the Automated Analysis of Symmetric Primitives) [BGG+24]. The library is designed to be modular, extendable, easy to use, generic, efficient and fully automated. It is an extensive toolbox gathering state-of-the-art techniques aimed at simplifying the manual tasks of symmetric primitive designers and analysts. CLAASP is built on top of Sagemath and is open-source under the GPLv3 license. The central input of CLAASP is the description of a cryptographic primitive as a list of connected components in the form of a directed acyclic graph. From this representation, the library can automatically:

1. generate the Python or C code of the primitive evaluation function;
2. execute a wide range of statistical and avalanche tests on the primitive;

3. generate SAT, SMT, CP and MILP models to search, for example, differential and linear trails;
4. measure algebraic properties of the primitive;
5. test neural-based distinguishers.

CLAASP can reproduce many of the results that were obtained in the literature and even produce new results [BGG+24].

3.3 Tagada

Tagada [LDLS21, DDG+24] (Tool for Automatic Generation of Abstraction-based Differential Attacks) is a tool which generates Minizinc [NSB+07] models for the search for differential primitives on word-based SPN ciphers, such as SKINNY or WARP. The search for such differential is typically divided into two steps:

1. Find the Truncated Differential with the highest probability (section 2.4);
2. Instantiate the Truncated Differential found at the step 1 with the best Differential (i.e. the one with the greatest probability).

Tagada generates models that implement the two steps, including optimizations based on inferred equalities through XOR operators, in order to drastically reduce the number of incorrect solutions to be passed to step 2. Such constraints are deduced naturally from a Directed Acyclic Graph (DAG) representation of the cipher under study. The genericity of Minizinc models enables solving with a range of CP, SAT and SMT solvers, in particular, the ones participating in the MiniZinc competition, that provide an interface to MiniZinc. On the other hand, solver-specific optimizations and perks are abstracted away by the Minizinc interface, compared to models developed in the native language of a solver.

3.3.1 Representation of the cipher

Tagada represents the cipher as graphs. In particular, since it works with SPN ciphers, let $w = w_{<1>}||\dots||w_{<t>}$ be an SPN state such that the words $w_{<i>} \in \{0,1\}^s$ have all the same length s. Then, Tagada computes a Directed Acyclic Graph (DAG) that contains two different kinds of vertices denoted P and O, respectively:

- each vertex in P corresponds to a parameter;
- each vertex in O corresponds to an operator.

Arcs connect operators to their input and output parameters: the predecessors (resp. successors) of an operator o are denoted $pred(o)$ (resp. $succ(o)$) and they correspond to input (resp. output) parameters, i.e. $succ(o) = o(pred(o))$. Some input parameters

have no predecessor in the DAG. These input parameters either correspond to words that are resulting from the plaintext or the key, or to constant values. The set of input parameters that are constant values is denoted C (see Figure 3.1).

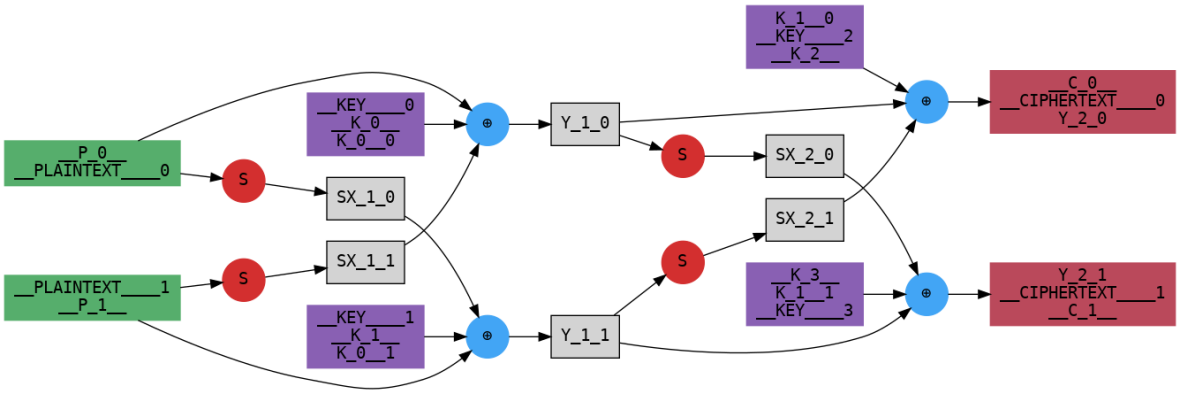

FIGURE 3.1 DAG representation of a 2-round Toy Cipher, see Example 3.6. In this case the set of the operators are $O = \{S, \oplus\}$ and the set of the parameters (all variables) are: $P = \{p_{<0>}, p_{<1>}, k^0_{<0>}, k^0_{<1>}, sx^1_{<0>}, sx^1_{<1>}, y^1_{<0>}, y^1_{<1>}, k^1_{<0>}, k^1_{<1>}, sx^2_{<0>}, sx^2_{<1>}, c_{<0>}, c_{<1>}\}$.

In order to perform the search of the differentials, TAGADA need to create the DAGs. Given the iterated function of the cipher, it is possible to create the DAG described before, called the **Specification DAG**. Each node P contains

- The domain of the word e.g. for WARP it is a word of 4 bits so the domain is $[0, 2^4 - 1]$ and
- The type i.e. if it is a constant or a variable.

Each node O contains:

- The input of the transition;
- The output of the transition;
- The type of the function i.e linear or non-linear and
- The name of the function e.g. X-or, rotation etc.

Let's describe a toy cipher useful to understand how a Specification DAG.

Example 3.6 (Toy Cipher)**.** A state of the Toy Cipher is composed by two nibble $p^0 = p^0_{<0>} || p^0_{<1>}$ with $p^0_{<i>} \in \{0,1\}^4$. The round function first apply an S-Box S of 4 bits,

$$sx^1_{<i>} = S(w^0_{<i>})$$

after a round key mixing XOR with the round key, the initial state and the permuted output of the S-Box:

$$y^1_{<0>} = k^0_{<0>} \oplus sx^1_{<1>} \oplus p^0_{<0>} \quad y^1_{<1>} = k^0_{<1>} \oplus sx^1_{<0>} \oplus p^0_{<1>}$$

We repeat this operation to obtain the ciphertexts. First, the S-Boxes:

$$sx^2_{<i>} = S(y^1_{<i>})$$

Finally,

$$c_{<0>} = y^2_{<0>} = k^1_{<0>} \oplus sx^2_{<1>} \oplus y^1_{<0>} \quad c^1_{<1>} = y^2_{<1>} = k^1_{<1>} \oplus sx^2_{<0>} \oplus y^1_{<1>}$$

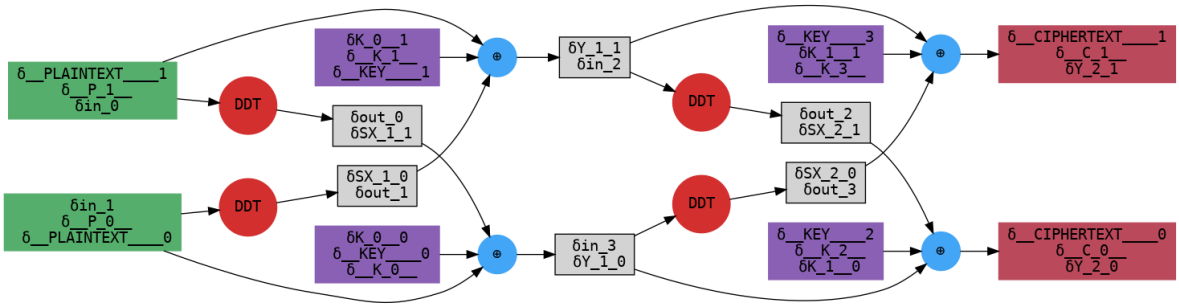

FIGURE 3.2 Differential DAG representation of a 2-round Toy Cipher.

To perform the search, we need the Differential DAG Figure 3.2 and the Truncated DAG 3.3. To be more accurate we denote with P_S and O_S, P_D and O_D, P_T and O_T, the set of vertices and operators respectively of the Specification, Differential and Truncated DAG. TAGADA creates this DAG in order from the previous one. First, the differential one. We divide the nodes in P_S into two possible case. Let $x \in P_S$

1. If x is a variables, then $x \to \delta x$ and $\delta x \in P_D$. So, we treat him as a differential variable in the Differential DAG;
2. If x is a constant, then we can have two possible cases according to which node operator $o \in O_S$ are the input. Let $y \in P_S$ be the other input and $k \in P_S$ be the output to that function $(o(y, x) = k)$
   (a) If $o = \oplus$, since:

$$(y \oplus x) \oplus ((y \oplus \delta) \oplus x) = \delta \implies \delta \xrightarrow{\oplus x} \delta$$

   Then, the constant x disappears together with that operator and $\delta y = \delta k \in P_D$ in the Differential DAG.
   (b) If o in any other linear operator (ecxept for the bit-wise OR), since:

$$o(y, x) \oplus o((y \oplus \delta), x) = o(\delta, x) \implies \delta \xrightarrow{o(\cdot, x)} o(\delta, x)$$

   Then, the constant remain the same $x \in P_D$ as well the operator $o \in O_D$.

For the other operators that don't involve the constants we have two possible cases.

Let $o \in O_S$

1. If o is linear

$$o(x) \oplus o(x \oplus \delta) = o(x \oplus x \oplus \delta) = o(\delta) \implies \delta \xrightarrow{o(\cdot)} o(\delta)$$

   Then, $o \in O_D$.
2. If o is non-linear, i.e. it is an S-Box, TAGADA replace it with its DDT. So, $o \in O_S$ then $DDT_o \in O_D$.

The last DAG we need in TAGADA for the search for differential is the Truncated DAG. This is the process of abstraction we have described in section 2.4. To build the last graph we start from the Differential DAG. The nodes in P_T represent the abstraction of the words. In particular, let $\delta x_{<i>} \in P_D$ be the i-th word of the difference state then, we abstract the variable using the function Δ Equation 2.5:

$$\delta x_{<i>} \to \Delta(\delta x_{<i>})$$

We obtain $\Delta(\delta x_{<i>}) \in P_T$ and it represents the abstraction of that word:

$$\Delta(\delta x_{<i>}) = \begin{cases} 1 & \text{if } \delta(x_{<i>}) \in (0, 2^s - 1] \\ 0 & \text{if } \delta(x_{<i>}) = 0 \end{cases}$$

The nodes in O_D, that represents the operators, are divided into two categories:

- The non-linear operator, i.e. the DDT of the S-Box, is replaced with its **maximum activation value**: its greatest value, in order to evaluate the probability for the truncated (as we have done in Equation 2.7).
- The linear operators are replaced with the truth table that modelize their function in order to compile the CP problem (see subsection 3.3.2)

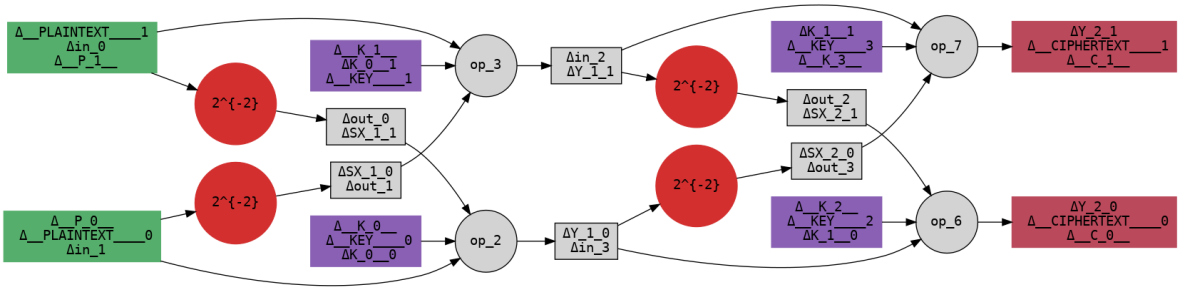

FIGURE 3.3 Truncated DAG representation of a 2-round Toy Cipher.

3.3.2 Logic Modelization of the Operators.

With the term **Logic Modelization** we refer to the translation of the operators as constrains in the COP problem. From the DAGs we have created, TAGADA creates a COP problem with boolean constraint so the domains are all $\{0, 1\}$.

Linear Operator. Let's take the bit-wise XOR operator:

$$x_1 \oplus x_2 = x_3$$

with $x_1, x_2, x_3 \in \{0, 1\}$. The tuples allowed by this constrains are defined by extensional representation as

$$(x_1, x_2, x_3) \in \{(1, 0, 1), (0, 1, 1), (0, 0, 0), (1, 1, 0)\}$$

In the case of abstracted variable the situation could change, indeed in this case where we represent the word XOR operation as

$$\Delta(\delta x_1) \oplus \Delta(\delta x_2) = \Delta(\delta x_3),$$

we have

$$\Delta(\delta x_i) = 1 \iff \delta(x_i) \in (0, 2^s - 1]$$

So, the tuples allowed are

$$(\Delta(\delta x_1), \Delta(\delta x_2), \Delta(\delta x_3)) \in \{(1, 0, 1), (0, 1, 1), (0, 0, 0), (1, 1, 0), (1, 1, 1)\}$$

The first two tuples $(1, 0, 1), (0, 1, 1)$ are allowed since if just one input word has a difference and the second doesn't it implies that the output has a difference, the third tuple $(0, 0, 0)$ is allowed since if there is no difference in both the inputs then the output doesn't have any difference. If both the inputs of the XOR does have a difference:

$$(\Delta(\delta x_1) = 1, \Delta(\delta x_2) = 1)$$

then. we can have two options:

1. The difference is different:

$$\delta x_1 \neq \delta x_2 \implies \delta x_1 \oplus \delta x_2 = \delta x_3 \neq 0$$

Then,

$$\delta x_3 \in (0, 2^s - 1] \iff \Delta(\delta x_3) = 1$$

;

2. It is the same difference:

$$\delta x_1 = \delta x_2 \implies \delta x_1 \oplus \delta x_2 = \delta x_3 = 0 \implies \Delta(\delta x_3) = 0$$

So, to express the possible tuples allowed, the constrain set in TAGADA is:

$$\begin{aligned}(\neg\Delta(\delta x_1) \wedge \neg\Delta(\delta x_2) \wedge \neg\Delta(\delta x_3)) \vee \\ (\Delta(\delta x_1) \wedge \Delta(\delta x_3)) \vee (\Delta(\delta x_1) \wedge \Delta(\delta x_2)) \vee (\Delta(\delta x_2) \wedge \Delta(\delta x_3))\end{aligned}$$

The codification in the Truncated DAG is expressed through an **abstract table** in which all the possible values are listed (see the table below).

$\Delta(\delta x_1)$	$\Delta(\delta x_2)$	$\Delta(\delta x_1) \oplus \Delta(\delta x_2)$
0	0	0
0	1	1
1	0	1
1	1	0
1	1	1

If we consider the XOR with three inputs as well, following the same idea of before, the boolean constrain is set as:

$$\begin{aligned}(\neg\Delta(\delta x_1) \wedge \neg\Delta(\delta x_2) \wedge \Delta(\delta x_3) \wedge \neg\Delta(\delta x_4)) \vee \\ (\Delta(\delta x_3) \wedge \Delta(\delta x_4)) \vee (\Delta(\delta x_2) \wedge \Delta(\delta x_4)) \vee (\Delta(\delta x_1) \wedge \Delta(\delta x_4)) \vee \\ (\Delta(\delta x_2) \wedge \Delta(\delta x_3)) \vee (\Delta(\delta x_1) \wedge \Delta(\delta x_3)) \vee (\Delta(\delta x_1) \wedge \Delta(\delta x_2))\end{aligned}$$

The first line allows the tuples where all the variables are 0, the second line refers to the tuples where just one of the word is active and so the output should be, and the last line concerns the possibility to have couples of input bit words to be active that could imply anything for the output word. In total we have, $1 + 3 + 3$ formula to form a **Disjoint Normal Formula**. In this case tuples allowed for $\Delta(\delta x_1) \oplus \Delta(\delta x_2) \oplus \Delta(\delta x_3) = \Delta(\delta x_4)$ are presented in the table below:

$\Delta(\delta x_1)$	$\Delta(\delta x_2)$	$\Delta(\delta x_3)$	$\Delta(\delta x_1) \oplus \Delta(\delta x_2) \oplus \Delta(\delta x_3)$
0	0	0	0
0	0	1	1
0	1	0	1
1	0	0	1
0	1	1	0
0	1	1	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

In general, TAGADA is able to menage as well the bitwise XOR of n variables using in the truncated DAG again the abstracted table and for the constrain the translation using $1 + n + \binom{n}{2}$ AND formulas to create an DNF.

Every linear operator implemented in TAGADA creates in the truncated DAG the abstracted table in which every possible tuple is presented and then, translated as a DNF formula for the related CP problem.

Non-Linear Operator. Instead, for the non-linear layer implementation on the COP problem, the only constrain defined in this case is related to the property of the SPN S-box, i.e. they are bijective. Given $\Delta(\delta x_1)$ and $\Delta(\delta x_2)$ such that $x_2 = \pi_S(x_1)$ i.e. x_2 is the ouput of the S-Box applied to x_1, then, the boolean condition created from this layer is:

$$\Delta(\delta x_1) = \Delta(\delta x_2)$$

In other words, if the input word is active so it is the output word. The operators implemented inside TAGADA are presented in the Table 3.1.
Remark. *TAGADA allows also the Conjunctive Normal Formula as well. To translate from the abstract table rapresentation to those formulas it is used ESPRESSO [MSBV93].*

3.3.3 Step 1.

In the first step of TAGADA, the objective is to find the best truncated differential. In order to find those truncated differential we define a COP problem. The idea of TAGADA is to automatic generate this problem just by the description of the Truncated DAG. The objective function is

$$f_w(x) = \sum_{i,j} w^j_{<i>} \Delta(\delta x^j_{<i>}) \tag{3.2}$$

Table 3.1 Supported operators with corresponding notations and step support.

Notation	Operator	Step 1 support	Step 2 support
$=$	Equal	✓	✓
LFSR	Linear Feedback Shift Register	✓	✓
$A, B \to (A, B)$	Split	✓	✓
$(A, B) \to AB$	Concat	✓	✓
$\ll, \gg$	Left (Right) Shift	✓	✓
$\lll, \ggg$	Left (Right) Circular Shift	✓	✓
$\&K$	Bitwise AND with Constant	✓	✓
$\oplus$	N-ary Bitwise XOR	✓	✓
$\otimes_K$	GF Multiplication with Constant	✓	✓
$\odot_K$	GF Matrix Multiplication with Constant	✓	✓
T	Linear Lookup Table	✓	✓
DDT	Differential Distribution Table	✓	✓

where

- $x^j_{<i>}$ represent the i-th word of the input state of an S-Box at round j;
- $\Delta(\delta x^j_{<i>})$ represent the abstraction of the difference of such word;
- $w^j_{<i>}$ represent the minus logarithm in base 2 of the maximum activation value of the S-Box applied to the i-th word at round j:

$$w^j_{<i>} = -\max \mathcal{D}_{\pi^j_{S,<i>}}$$

with $\pi^j_{S,<i>}$ the S-Box applied at the round j to the word i-th word (usually, in the SPN cipher the S-Box used in the round function is the same so that value is always the same).

Since the aim is to maximize the truncated probability 2.7, and remarking that the value of the DDT is a multiple of 2, we want to maximize the approximation:

$$\begin{aligned} p_{trunc} &= \prod_{i,j} \max \mathcal{D}_{\pi^j_{S,<i>}} \Delta(\delta x^j_{<i>}) \\ \log_2(p_{trunc}) &= \log_2 \left(\prod_{i,j} \max \mathcal{D}_{\pi^j_{S,<i>}} \Delta(\delta x^j_{<i>}) \right) \\ \log_2(p_{trunc}) &= \sum_{i,j} \log_2 \left(\max \mathcal{D}_{\pi^j_{S,<i>}} \right) \Delta(\delta x^j_{<i>}) \end{aligned}$$

Another reason to use the logarithm in base 2 is that we can sometimes work with very

large integer, sometimes in the order of 2^{16} and the dedicated solver could not handle such big number.

Using the logarithm the maximization of the objective doesn't change, but if we add also a minus then:

$$\begin{aligned}\min\left(-\log_2(p_{trunc})\right) &= \min\left(-\sum_{i,j}\log_2\left(\max \mathcal{D}_{\pi^j_{S,<i>}}\right)\Delta(\delta x^j_{<i>})\right)\\ &= \min\left(\sum_{i,j}-\log_2\left(\max \mathcal{D}_{\pi^j_{S,<i>}}\right)\Delta(\delta x^j_{<i>})\right)\\ &= \min\left(\sum_{i,j} w^j_{<i>}\Delta(\delta x^j_{<i>})\right)\end{aligned}$$

The aim is then to minimize the objective function:

$$\min \sum_{i,j} w^j_{<i>}\Delta(\delta x^j_{<i>}) \tag{3.3}$$

In the COP problem (X, D, C, f_w) of the step 1, we have defined

- The objective f_w in Equation 3.3;
- The set of variables is $X = \{\Delta(\delta x^j_{<i>}) | \delta x^j_{<i>} \text{ is a variable}\}_{i,j}$;
- The set of domains is $\{0, 1\}$ for each variable;

We miss just the constraints. To define the constrains it is possible, from the Truncated DAG, to easily define those constrains using the Logic Modelization of the operators (subsection 3.3.2). The COP problem is then written in Minizinc (see Algorithm 4).

Minizinc allows us to solve the CP problem using different already existed dedicated solver such as PICAT-SAT [ZK16] or Or-Tools [CDF+23]. From the step 1, Tagada is able to find which S-Boxes are active and estimate the probability of the truncated characteristic. Hence, we are able to find an upper bound for the differential, see Equation 2.8.

Algorithm 4 `minizinc_search_truncated_toy_cipher`

```
 1: var bool : var_0;
 2: var bool : var_1;
 3: var bool : var_2;
 4: var bool : var_3;
 5: var bool : var_4;
 6: var bool : var_5;
 7: var bool : var_6;
 8: var bool : var_7;
 9: var bool : var_8;
10: var bool : var_9;
11: var bool : var_10;
12: var bool : var_11;
13: var bool : var_12;
14: var bool : var_13;
15: constraint var_0 == var_1;
16: constraint var_2 == var_3;
    constraint (((¬(var_1) ∧ ¬(var_4) ∧ ¬(var_3) ∧ ¬(var_9))
17:                 ∨(var_3 ∧ var_9) ∨ (var_4 ∧ var_9) ∨ (var_1 ∧ var_9)
                    ∨(var_4 ∧ var_3) ∨ (var_1 ∧ var_3) ∨ (var_1 ∧ var_4)));
    constraint (((¬(var_3) ∧ ¬(var_6) ∧ ¬(var_1) ∧ ¬(var_8))
18:                 ∨(var_1 ∧ var_8) ∨ (var_6 ∧ var_8) ∨ (var_3 ∧ var_8)
                    ∨(var_6 ∧ var_1) ∨ (var_3 ∧ var_1) ∨ (var_3 ∧ var_6)));
19: constraint var_7 == var_8;
20: constraint var_5 == var_9;
    constraint (((¬(var_8) ∧ ¬(var_10) ∧ ¬(var_9) ∧ ¬(var_11))
21:                 ∨(var_9 ∧ var_11) ∨ (var_10 ∧ var_11) ∨ (var_8 ∧ var_11)
                    ∨(var_10 ∧ var_9) ∨ (var_8 ∧ var_9) ∨ (var_8 ∧ var_10)));
    constraint (((¬(var_9) ∧ ¬(var_12) ∧ ¬(var_8) ∧ ¬(var_13))
22:                 ∨(var_8 ∧ var_13) ∨ (var_12 ∧ var_13) ∨ (var_9 ∧ var_13)
                    ∨(var_12 ∧ var_8) ∨ (var_9 ∧ var_8) ∨ (var_9 ∧ var_12)));
23: constraint (sum([var_3, var_1, var_4, var_6, var_10, var_12])) > 0;
24: var 0..800 : obj_200;
25: constraint obj_200 == sum([var_1, var_3, var_8, var_9]);
26: var 0..800 : obj = 200 ∗ obj_200;
27: constraint obj >= 0;
    solve ::int_search([var_1, var_3, var_8, var_9], smallest, indomain_min, complete)
28:         minimize obj_200;
```

3.3.4 Step 2

The Step 2 of TAGADA is the instantiation process and it uses the output of Step 1 and the Differential DAG in order to find the actual differential: if the word of a state is active, we would like to instantiate a difference for the difference. In particular, TAGADA defines a new COP problem to solve. The objective function in this case is:

$$f(x) = \sum_{i,j} -\log_2 \left(\mathcal{D}_{\pi_S} \left(\delta(x^j_{<i>}) \right) \right) \Delta(\delta x^j_{<i>}) \tag{3.4}$$

The aim is to minimize this function. To define the COP $=(X, D, C, f)$, we need also:

- The set of variables: $X = \{\delta(x^j_{<i>}) | x^j_{<i>} \text{ is a variable}\}_{i,j}$;
- The domains of the variables are the same (since in the SPN cipher the length of the word is fixed for every state) and they depends from the length s of the word of the cipher: $D(\delta x^j_{<i>}) = [0, 2^s - 1] \subset \mathbb{N}$,

To define the set of the constraint we are able to define the modelization of the operators quite easily using the Differential DAG. The first contrain derive directly from the definition of the abstraction:

$$\begin{aligned} \Delta(\delta x^j_{<i>}) = 0 &\implies \delta(x^j_{<i>}) = 0 \\ \Delta(\delta x^j_{<i>}) = 1 &\implies \delta(x^j_{<i>}) \in (0, 2^l - 1] \end{aligned}$$

this process is called the **realization** of the abstract values. To modelize the propagation of the other operators, TAGADA use the CHOCO library [PFL16]. [DDG+24] combine the two steps introducing a new one optimized algorithm 5. It uses one extra function: `Step1_Next_Possible_UB`$(G_T, seen, UB)$. It simply increases the exponent (decreases the probability) by the activation of one S-Box.

Algorithm 5 `Tagada`. The *seen* set is used to avoid to pick the same truncated differential again.

Require: Truncated DAG G_T, Differential DAG G_D

Initialize $LB \leftarrow 0$, $UB \leftarrow 1$, `best` $\leftarrow$ *null*, *seen* $\leftarrow \{\}$
$\texttt{sol}_1 \leftarrow \texttt{Step1}(G_T, seen, UB)$ ▷ Find the best truncated, see section 3.3.3, under the probability UB.
$UB \leftarrow \mathbb{P}(\texttt{sol}_1)$
while $LB < UB$ **do**
    $seen \leftarrow seen \cup \texttt{sol}_1$
    $\texttt{sol}_2 \leftarrow \texttt{Step2}(G_D, \texttt{sol}_1, LB)$ ▷ Search the best instantiation see section 3.3.4.
    $LB \leftarrow \mathbb{P}(\texttt{sol}_2)$
    if $LB < UB$ **then**
        $\texttt{sol}_1 \leftarrow \texttt{Step1\_Next}(G_T, seen, UB)$ ▷ Search another truncated differential not in *seen* and with probability fixed at UB.
        if $\texttt{sol}_1$ is *null* **then**
            $UB \leftarrow \texttt{Step1\_Next\_Possible\_UB}(G_T, seen, UB)$
            if $LB \geq UB$ **then**
                `break`
            end if
            $\texttt{sol}_1 \leftarrow \texttt{Step1}(G_T, seen, UB)$
        end if
        $UB \leftarrow \mathbb{P}(\texttt{sol}_1)$
    end if
end while

Chapter 4

Integrating Boomerang Attack into Tagada

This chapter presents the primary contribution of this thesis: the integration of Boomerang Cryptanalysis into the automated tool Tagada. As explored in Chapter 2, the Boomerang Attack allows cryptanalysts to distinguish a block cipher from a random permutation by combining two high-probability differential characteristics that do not cover the entire cipher individually. While Chapter 3 introduced Tagada as a tool for finding standard differential trails using Constraint Programming (CP), extending it to support boomerang distinguishers introduces significant structural complexities that this chapter addresses.

We adopt the **Sandwich Attack** framework, arguably the most refined theoretical model for boomerang analysis. In this model, the cipher E is decomposed into three distinct sub-components: the upper trajectory E_0, the lower trajectory E_1, and a middle transition component E_m. The validity of the distinguisher is determined by the overall probability formula $P = p^2q^2r$, where p and q represent the probabilities of the characteristics for E_0 and E_1 respectively, and r quantifies the probability of their successful interaction through E_m.

Section 4.1 details the methodology for partitioning the cipher's Specification DAG (Directed Acyclic Graph) into these three required sub-graphs. A major theoretical and practical challenge discussed in this section is the automated modeling of the decryption process required for the lower trail E_mE_1. Unlike SPN ciphers, where decryption is often a straightforward inversion of components, Feistel networks and complex Key Schedules rely on structural properties rather than functional invertibility. We propose a novel

algorithmic approach that generates decryption constraints by traversing the graph backwards—solving for inputs given outputs—without the need to explicitly construct a separate inverse graph.

The search strategy itself is divided into two sequential optimization phases, reflecting the architecture of TAGADA:

- **Step 1 (Section 4.2):** We model the search for truncated differentials as a Constraint Optimization Problem (COP). The objective function is designed to minimize the cost (log-probability) of active S-boxes across both the encryption path (E_0) and the decryption path (E_1), while simultaneously ensuring consistency in the shared middle segment (E_m). This step provides the abstract structure of the attack.
- **Step 2 (Section 4.3):** This phase focuses on instantiating the abstract solution with concrete difference values. The primary goal here is to maximize the interaction probability r. Since evaluating r for all possible pairs is computationally infeasible, we introduce a heuristic filtering algorithm. This algorithm leverages the **Boomerang Connectivity Table (BCT)** and clustering effects to identify the most promising candidate pairs for experimental verification.

Finally, **Section 4.4** validates the effectiveness of this extension. We present experimental results obtained on several block ciphers, comparing the characteristics found by our automated tool against those identified by existing state-of-the-art libraries, demonstrating that our generalized approach yields competitive and often equivalent results without requiring manual cipher-specific adaptations.

The Tagada Notation. A block cipher E_K is a function that takes as input a binary sequence K of size k bits, called key (also secret key or master key), a binary sequence M of size n bits, called plaintext or message, and returns a binary sequence C of size n bits, called ciphertext. More formally:
Definition 4.1 (Block Cipher)**.**

$$E_K(M) = E(K, M) : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n \text{ and } C = E_K(M).$$

Block ciphers are in the family of symmetric ciphers. To do this, E_K should be associated with a decryption function, called D_K. This decryption function takes as input a K, C pair and returns M. More formally:
Definition 4.2 (Associated Deciphering function of a Block Cipher)**.**

$$D_K(C) = D(K, C) = E_K^{-1}(C) : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n \text{ with}$$
$$M = D_K(C) \text{ and } \forall K, D_K(E_K(M)) = M.$$

Definition 4.3 (Associated Specification DAG of a Block Cipher)**.**

$SG = (SG_S, SG_F, SG_T, SG_P, SG_K, SG_C)$ where,

$$
\begin{aligned}
SG_S &= \{s \mid s \text{ is a state}^1 \text{ of the cipher}\} \\
SG_F &= \{f \mid f \text{ is a function of the cipher}\} \\
SG_T &= \\
\{((x_1,...,x_n), f, (y_1,...,y_m)) &\in ({SG_S}^n) \times SG_F \times ({SG_S}^m) \mid (y_1,...,y_m) = f(x_1,...,x_n)\} \\
SG_P &= \{p \mid p \text{ is a plaintext word of the cipher}\} \\
SG_K &= \{k \mid k \text{ is a primary key word of the cipher}\} \\
SG_C &= \{c \mid c \text{ is a ciphertext word of the cipher}\}
\end{aligned}
$$

For a given transition (or arc) $t \in SG_T$ such as $t = ((x_1,...,x_n), f, (y_1,...,y_m))$, we define its inputs as: $t_X = (x_1,...,x_n)$, its outputs as: $t_Y = (y_1,...,y_m)$ and its function as: $t_f = f$.

For a given state s, we define $\texttt{pred}(SG, s)$ the set of predecessors transitions of s. More formally: $\texttt{pred}(SG, s) = \{t \in SG_T, s \in t_Y\}$.

4.1 Splitting the Graph

After giving the notation, we describe how to cut the graph in Tagada.

4.1.1 Splitting the block cipher E

The computation of boomerang distinguishers can be done by using several techniques. In this paper we use the same construction as in [HNE22]. The boomerang distinguisher is then represented by two differential characteristics, an upper characteristic, which represents the first part of the encryption process, and a lower characteristic, which represents the second part of the encryption process. The two characteristics are overlapping on a given number of rounds, in the middle, this section is called E_m (see Figure 4.1).

The probability of the boomerang distinguisher can be derived from the probability of

[1] A state is either a plaintext word, a key word, an internal state (*i.e.* a temporary word created during the encryption process), a ciphertext word or a constant.

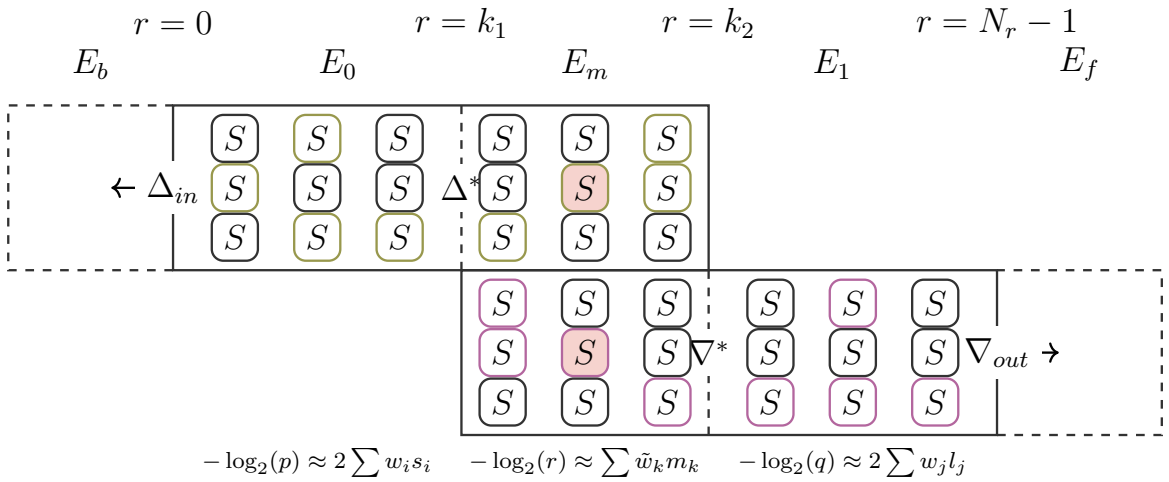

FIGURE 4.1 Boomerang cryptanalysis construction. The upper characteristic is composed of three parts: E_b, E_0 and E_m. The lower characteristic is also composed of three parts: E_m, E_1 and E_f where the boomerang distinguisher is computed on E_0, E_1 and E_m whereas the extended rounds for the attack are E_b and E_f. The cut of the three parts E_0, E_m and E_1 are done on rounds k_1 and k_2. The probability is approximated by evaluating the number of active S-Boxes in E_0 ($\sum s_i$), E_1 ($\sum l_j$) and the number of common active S-Boxes in E_m ($\sum m_k$). The weights w_i, w_j and w_k of those sums are defined in Section 4.2.

both upper and lower characteristics with the boomerang approximation:

$$p^2 q^2 r. \tag{4.1}$$

To do so, we generate two characteristics, one characteristic for E_0 and E_m, called the upper characteristic, and one characteristic for E_m and E_1 called the lower characteristic. Both E_0 and E_1 can be computed by usual differential models. The problem is E_m which links the two characteristics together in order to build the boomerang.

4.1.2 Creating E_0, E_m and E_1 automatically.

In order to build a valid boomerang distinguisher we need to split the cipher into three parts E_0, E_m and E_1. In TAGADA, ciphers are described as graphs and the library does not handle "rounds". In order to cut the graph in three parts we rest on the variables names in order to retrieves the end of each parts. In TAGADA the input and output of round functions are designated as X_r. By convention, X_r represents the set of output states for round r and, simultaneously, the set of input states for round $r + 1$. As previously established in Figure 4.1, the final round of E_0 is k_1, the final round of E_m is k_2, and the final round of E_1 is $N_r - 1$. Consequently, the set of output states of E_0 is X_{k_1}, the set of output states E_m is X_{k_2} and the set of output states of E_1 is the

ciphertext SG_C. Having defined these output variables, the subsequent step involves retrieving the variables and transitions pertinent to each part. This is achieved by recursively defining the subgraph extraction process using Algorithm 6. For a given output variables O and a specified set of allowed transitions $T_{allowed}$, the algorithm identifies all predecessor states of O exclusively using the transitions within $T_{allowed}$.

The individual subgraphs are then formally defined as follows:

$$
\begin{aligned}
(S_{E_0}, F_{E_0}, T_{E_0}) &= \texttt{extract\_boomerang\_part}(SG, SG_T, X_{k_1}) \\
(S_{E_m}, F_{E_m}, T_{E_m}) &= \texttt{extract\_boomerang\_part}(SG, SG_T \setminus T_{E_0}, X_{k_2}) \\
(S_{E_1}, F_{E_1}, T_{E_1}) &= \texttt{extract\_boomerang\_part}(SG, SG_T \setminus (T_{E_0} \cup T_{E_1}), SG_C)
\end{aligned}
$$

These definitions extract the subgraphs E_0, E_m, and E_1 based on their respective output states (X_{k_1}, X_{k_2}, and SG_C) while successively restricting the set of available transitions (SG_T) to ensure the parts are disjoint in terms of transitions.

Notice that, we can directly construct the concatenated subgraph E_0E_m by invoking the function:

$$\texttt{extract\_boomerang\_part}(SG, SG_T, k_2)$$

Similarly, the subgraph E_mE_1 can be constructed by calling

$$\texttt{extract\_boomerang\_part}(SG, SG_T \setminus T_{E_0}, SG_C)$$

and then the subgraph E_m is the intersection between those two subgraphs.

4.2 Step 1: Computing Truncated Differentials Distinguishers

We recall that the Tagada Step 1 works on an abstraction where Boolean variables represent difference values. The aim of the Step 1 is to minimize the number of Boolean variables equal to 1 when entering the S-boxes.

Model the COP problem. To model our COP problem we follow the general structure used to find truncated differential characteristics of [DDG+24] with several adaptations inspired by [HNE22]. First, we create the binary variables defining sets for the nodes in the E_0E_m, E_mE_1 and E_m graphs. As in [DDG+24] each variable is 1 if the state is active (i.e. it contains a difference) otherwise it is 0.

Algorithm 6 `extract_boomerang_part`

Require: A Specification Graph SG, a set $T_{allowed}$ of allowed transitions and a set of output nodes O.

Ensure: A triplet (S, F, T) defined as follows: S is the set of nodes that can reach an output node in O using only transitions from the allowed set $T_{allowed}$. T is the minimal subset of $T_{allowed}$ that contains all the transitions necessary to connect the nodes in S to the output nodes in O. Finally, F is the set of functions utilized by the transitions in T.

1: $S \leftarrow O$
2: $T \leftarrow \emptyset$
3: $F \leftarrow \emptyset$
4: `updated` $\leftarrow$ `true`
5: **while** `updated` **do**
6: `updated` $\leftarrow$ `false`
7: **for** each node s in S **do**
8: **for** each transition $t \in \texttt{pred}(SG, s)$ where $t \in T_{allowed} \wedge t \notin T$ **do**
9: $S \leftarrow S \cup t_I$
10: $F \leftarrow F \cup \{t_f\}$
11: $T \leftarrow T \cup \{t\}$
12: `updated` $\leftarrow$ `true`
13: **end for**
14: **end for**
15: **end while**
16: **return** (S, F, T)

The objective is to minimize the function:

$$2 \underbrace{\sum_i w_i s_i}_{\text{Upper Prob: } -\log_2(p)} + 2 \underbrace{\sum_j w_j l_j}_{\text{Lower Prob: } -\log_2(q)} + \underbrace{\sum_k \tilde{w}_k m_k}_{\text{Middle Prob: } -\log_2(r)} \tag{4.2}$$

Let's analyze each term:

- $\sum_i w_i s_i$ contains the weights w_i that corresponds to the minus logarithm in base 2 of the maximum activation probability inside the DDT, $s_i \in \{0, 1\}$ are the variables related to the input state of an S-box in E_0; and it is multiplied by 2 since in the computation of the total probability of Equation 4.1 we have p^2;
- $\sum_j w_j l_j$ is equivalent to $\sum_i w_i s_i$ but for E_1;
- $\sum_k \tilde{w}_k m_k$ contains the weights $\tilde{w}_k$ that corresponds to the minus logarithm in base 2 of the maximum activation probability inside the BCT (see definition

2.18). $m_j \in \{0,1\}$ are the variables related to the activation of an S-box in E_m. A specific condition is imposed for the activation of an S-Box within E_m. An S-Box in E_m is defined as active if and only if it is active in both the upper characteristic E_0E_m and in the lower characteristic E_mE_1.

To model a boomerang, we firstly define two COPs: one that reflects the E_0E_m characteristic and another that reflects the E_mE_1 characteristic. For the E_0E_m characteristic, we need to model the encryption process, while for the E_mE_1 characteristic, we need to model the decryption process. In this section, we demonstrate how to model the E_0E_m characteristic and how to adapt the computation to model the E_mE_1 characteristic. Finally, we show how to connect the middle part of the upper characteristic with the middle part of the lower characteristic and create the full COP.

4.2.1 Modelling E_0E_m

To define the COP, we need as before the (X, D, C, f). The objective is related to the problem in its entirely and it is given Equation 3.3. The focus is then moved to find (X, D, C).

Given the triplet $(S_{E_0E_m}, F_{E_0E_m}, T_{E_0E_m})$ that represents the boomerang part E_0E_m, the set of variables X can be defined as the set of truncated differences of the states involved: $X = \{\Delta_s \mid s \in S_{E_0E_m}\}$. Here, Δ_s represents the difference abstraction of state s, see Equation 2.5. Consequently, the domain of each variable Δ_s is fixed to a binary set $\{0,1\}$, reflecting the presence (1) or absence (0) of a difference in that state.

The set of constraints C is derived directly from the transition function $T_{E_0E_m}$. In the analysis of symmetric ciphers, the operators are broadly categorized into non-linear and linear components.

Non-Linear Operator. For the non-linear operators (e.g., S-boxes), the constraint is governed by the property $S(x) \oplus S(x \oplus 0) = 0$. This formula signifies a key dependency: for any S-box application, the output difference of the S-box is active (non-zero) if and only if its corresponding input difference is also active (non-zero). This dependency is formally modeled by the following logical:

$$\forall t \in T_{E_0E_m} \text{ where } t_f \text{ is an S-Box,}$$
$$\text{let } (\Delta_{in}) = t_I \text{ and } (\Delta_{out}) = t_O, \qquad \Delta_{in} = \Delta_{out}$$

Linear Operator. Linear operators are modeled as in [DDG+24]. For each linear operator f, an abstract table α_f is computed from the operator and posted as a constraint table. This abstract table is generated by brute-forcing all possible pairs of inputs, which limits the abstraction to 32-bit operators. More formally:

$$\forall t \in T_{E_0E_m} \text{ where } t_f \text{ is linear,}$$
$$\text{let } \alpha_{t_f} = \texttt{abstract\_table}(t_f), \qquad t_I||t_O \in \alpha_{t_f}.$$

To achieve the optimal differential characteristic for E_0E_m, we aim to select the solution with the highest probability. Consequently, the activation function is the one described in Equation 4.2.

$$\texttt{minimize } obj = 2\sum_i w_i s_i$$

To remove the trivial solution which always occurs with no difference at all –i.e., where two equivalent plaintexts are always ciphered to the same ciphertext– we add an additional constraint that force at least one difference in the input. This difference may be either in the plaintext or in the master key for what we call related-key attacks, or only in the plaintext for single-key attacks.

$$\sum_{\Delta_s \in SG_P \cup SG_K} \Delta_s > 0$$

4.2.2 Modelling E_mE_1.

In the cipher E_mE_1, the decryption direction must be modeled. However, since the encryption process is bijective, we can also model the cipher in the decryption direction. The objective function for E_mE_1 is also the one described in Equation 4.2.

$$\texttt{minimize } obj = 2\sum_i w_j l_j$$

Here, we also want to eliminate the trivial solution that always results in no difference at all. However, for the lower trail, we want to ensure that there is at least a difference

in the ciphertext or the key schedule for related-key attacks, or only in the ciphertext for single-key attacks.

$$\sum_{\Delta_s \in SG_C \cup SG_k} \Delta_s > 0$$

The calculation of key schedule states is explained shortly.

Linking E_m. The segment E_m is analyzed in two distinct contexts: the end of the first path E_0E_m and the start of the second path E_mE_1. To ensure path consistency and unify the two analyses, the respective Constraint Satisfaction Problems are merged. Furthermore, linking constraints are introduced to enforce correspondence between the two instances of E_m. Specifically, using the technique proposed in [HNE22], the S-Boxes in the upper E_m instance are directly linked to the corresponding S-Boxes in the lower E_m instance. An S-Box within the shared E_m is defined as active if and only if both of its linked counterparts (the S-Boxes in the upper path and the S-Boxes in the lower path) are simultaneously active.

We defined the shared S-Box activity variable m for the segment E_m. For an S-Box $s^{\uparrow}$ in the upper trail and its corresponding lower S-Box $s^{\downarrow}$, the activity is constrained by the logical conjunction:

$$\forall s^{\uparrow}, s^{\downarrow} \in E_m : m = s^{\uparrow} \wedge s^{\downarrow}$$

This constraint enforces that the S-Box at this position in the shared segment is considered active only when both the upper and lower paths exhibit activity.

The goal is to determine the optimal differential characteristic by maximizing the expected probability of the boomerang return. This is achieved through the following objective function

$$\texttt{minimize}\ obj = \sum_k \tilde{w}_k m_k$$

where m_k are the S-Box activity variables and $\tilde{w}_k$ are weights derived from the BCT table.

Enhancing the Probability Evaluation of E_m. A significant distinction exists between the model presented in [HBS21] (focused on SPN ciphers) and that of [HNE22] (focused on Feistel Networks): the latter introduces additional constraints on the shared segment E_m.

These constraints are designed to enforce a probability-one transition (i.e., preventing cancellations) across the S-Boxes in both the upper and lower differential trails. Formally, this mandates that once a word becomes active, it cannot be deactivated by subsequent XOR operations involving other variables. Although the theoretical justification for this modification remains underexplored, the model incorporating these constraints yield probabilities that are empirically closer to the experimental evaluations.

A notable complexity in implementing this approach lies in its automation: the model requires a directional representation of the cipher, with E_m represented in the encryption direction for the upper path and the decryption direction for the lower path.

The differential propagation with probability-one constraints are manifested in two distinct ways, dependent on the type of cryptographic primitive considered. For S-Boxes: The constraints remain intrinsically tied to the characteristic of the S-box itself. Specifically, the activation of an output difference is directly conditioned upon the activation of a specific input difference. For Linear Operators: The propagation rule is more systematic. The generic formula dictates that if any input difference to the linear function is active (non-zero), all output differences resulting from that input is considered active (non-zero). More formally:

$$\forall t \in T_{E_0 E_m} \text{ where } t_f \text{ is linear},$$
$$\forall \Delta_o \in t_o, \Delta_o = \bigvee_{\Delta_i \in t_I} \Delta_i$$

The integration of the decryption function, which we model as the lower portion of the E_m graph, presents significant conceptual challenges. The primary difficulty lies in the fact that a simple inversion of the encryption graph is not universally applicable. Decryption strategies differ fundamentally according to the underlying cryptographic architecture, mainly between Substitution-Permutation Networks (SPNs) and Feistel networks.

In the case of SPNs, the design relies almost exclusively on the use of bijective functions (S-boxes, bit permutations). Consequently, obtaining the decryption function is conceptually straightforward: it requires the inversion of each component function (e.g., using the inverse S-Box) and the reversal of the direction of the arcs in the data flow graph.

In contrast, Feistel networks do not require their round functions f to be bijective.

Invertibility is an emergent property of the architecture itself. Let the round function of a Feistel network be defined as:

$$L_{i+1} = f(L_i, K_i) \oplus R_i$$
$$R_{i+1} = L_i$$

The decryption process is derived by rearranging the terms, without needing to invert f:

$$L_i = R_{i+1}$$
$$R_i = f(L_i, K_i) \oplus R_{i+1}$$

Beyond structural inversion, two major components of a cipher exhibit asymmetric behavior when transitioning to decryption.

The first challenge concerns the key schedule. Whether for encryption or decryption, this process is fundamentally unidirectional: it derives the round keys (derived keys) from the master key. The subgraph representing the key schedule must therefore not be inverted when modeling decryption; it is reused as is. Only the application of the round keys to the data path may follow a reversed order (e.g., from RK_n to RK_1).

The second difficulty lies in the operation of adding the key to the text, usually performed via an exclusive-OR (XOR) operation. For encryption, we have:

$$C = P \oplus K$$

For decryption, the involutive property of XOR is used:

$$C \oplus K = P \tag{4.3}$$

This is not a functional inversion (XOR is its own inverse) but a reorganization of

the operands. From a graphical modeling perspective, the operation node (XOR) is preserved, but its input connections (the ciphertext C and the key K) are reconfigured to produce the plaintext P. To correctly model the decryption process, a system must algorithmically differentiate between components. We employ the following formal strategies to: (1) detect the key schedule subgraph, (2) correctly identify and invert key-mixing operations, and (3) properly reverse the remaining data-path functions based on the cryptographic structure (e.g., SPN or Feistel).

(1). We recursively define the set of elements belonging to the key schedule. An element is considered part of the key schedule if it is either an element of the primary key or is derived exclusively from other key schedule states. Formally, this property is captured by the predicate $\texttt{is\_key\_schedule\_state}(SG, s)$, whose definition relies on two supporting predicates.

A state s is a key schedule element if it is either a primary key element or if it is the output of a key schedule transition:

$$\begin{aligned} &\texttt{is\_key\_schedule\_state}(SG, s) = \\ &\texttt{is\_primary\_key\_element}(SG, s) \vee \\ (\exists t \in \texttt{pred}(SG, s), &\texttt{is\_key\_schedule\_transition}(SG, t)) \end{aligned}$$

The supporting predicates are defined as follows:

- The predicate $\texttt{is\_primary\_key\_element}(SG, s)$ identifies states that are part of the primary key, SG_K. This forms the base case for the recursion.

$$\texttt{is\_primary\_key\_element}(SG, s) = s \in SG_K.$$

- The predicate $\texttt{is\_key\_schedule\_transition}(SG, t)$ identifies transitions that belong to the key schedule. This holds if (1) all of its input states x are already key schedule elements, and (2) at least one input x is a variable state (to exclude operations that only involve constants).

$$\begin{aligned} &\texttt{is\_key\_schedule\_transition}(SG, t) = \\ (\forall x' \in t_X, &\texttt{is\_key\_schedule\_state}(SG, x')) \wedge (\exists x \in t_X, \texttt{is\_variable}(SG, x)) \end{aligned}$$

Using these predicates, we define the Key Schedule associated with the specification

graph SG as the subgraph KS.
Definition 4.4 (Key Schedule)**.**

$$\begin{aligned} KS(SG) &= (S_{KS}, F_{KS}, T_{KS}) \\ S_{KS} &= \{s \in SG_S \mid \texttt{is\_key\_schedule\_state}(SG, s)\} \\ F_{KS} &= \{f \in SG_F \mid \exists t \in T_{KS}, t_f = f\} \\ T_{KS} &= \{t \in SG_T \mid \texttt{is\_key\_schedule\_transition}(SG, t)\} \end{aligned}$$

We define a Key Mixing transition as the step in which key material is integrated with the algorithm's internal data state. This is frequently implemented as an AddRoundKey operation. Formally, a transition t is a key-mixing transition if it has at least one input from the key schedule and at least one input from the data path.

(2) We define a Key Mixing transition as the step in which key material is integrated with the algorithm's internal data state. This is frequently implemented as an AddRoundKey operation. Formally, a transition t is a key-mixing transition if it has at least one input from the key schedule and at least one input from the data path.

$$\begin{gathered} \texttt{is\_key\_mixing\_transition}(SG, t) = \\ (\exists x \in t_X, \texttt{is\_variable}(SG, x) \land \texttt{is\_key\_schedule\_state}(SG, x)) \land \\ (\exists x' \in t_X \setminus x, \texttt{is\_variable}(SG, x') \land \neg\texttt{is\_key\_schedule\_state}(SG, x')) \end{gathered}$$

(3) To generate the constraints for the decryption path, we do not explicitly compute the inverse graph. Instead, we traverse the graph SG and apply constraints based on the classification of each transition f.

- If f is a Key Schedule Transitions ($f \in F_{KS}$: The transition belongs to the keyschedule. The constraint is applied in the forward direction, identical to the encryption path, to preserve the unidirectionnal flow from the primary key to the derived rounds keys.

$$\forall y^L \in f_Y^L \qquad y^L = \max_{x^L \in f_X^L} x^L$$

- if f is Invertible (Bijective): If the operator f is invertible and not a part of the keyschedule (e.g., an S-Box), the constraint for decryption is the inverse of the encryption path constraint.

$$\forall x^L \in f_X^L \qquad x^L = \max_{y^L \in f_Y^L} y^L$$

- If f is a Key Mixing Transition: If f is a key mixing operator, it is typically an involution operation like XOR. We do not invert the function itself. Instead, we algebraically rearrange the constraint to solve the data-path input, given the key-path input and the output.
  Let $f_{X_w}^L$ be the set of input variables that are known (computable without this transition) and $f_{X_u}^L$ be the set of input variables that are unknown (computable only with this transition), such that $f_X^L = f_{X_w}^L \cup f_{X_u}^L$ and $f_{X_w}^L \cap f_{X_u}^L = \emptyset$. For an XOR-based operation, we have:

$$\bigoplus_{x_w \in f_{X_w}^L} x_w \oplus \bigoplus_{x_u \in f_{X_u}^L} x_u = f_Y^L$$

  We then rewrite the constraint to solve for the unknown inputs x_u:

$$\forall x_u \in f_{X_u}^L \qquad x_u = \bigoplus_{x_w \in f_{X_w}^L} x_w \oplus \bigoplus_{y \in f_Y^L} y$$

  For example, consider the round function of Warp:

$$s_{2i+1}^{r+1} = f(s_{2i+1}^r, k^r, s_{2i}^r) = s_{2i+1}^r \oplus k^r \oplus S(s_{2i}^r)$$

  To generate the decryption constraint, we solve the data-path input s_{2i+1}^r:

$$s_{2i+1}^r = s_{2i+1}^{r+1} \oplus k^r \oplus S(s_{2i}^r).$$

Our analysis framework (Tagada) does not explicitly store the inverse functions for operators. Therefore, the key idea of our algorithm is to start from the ciphertexts and traverse the graph visiting all nodes and adding the appropriate linear constraints during the process. The main challenge is to determine which nodes can be visited (i.e., computed) at each step.

We initialize a Boolean flag for each variable, setting it to `true` for ciphertexts and key schedule states (which are considered known) and `false` for all the others. The cipher is considered "inverted" when all plaintext variables nodes are marked `true`. The pseudocode for this process is depicted in Algorithm 7.

Algorithm 7 `inversion`

Require: Truncated Graph G, the set of ciphertext variables `ciphertext` and the set of key schedule elements `key_schedule_states`.

$C \leftarrow$ `ciphertext` $\cup$ `key_schedule_states` ▷ The set of computable elements.
$T \leftarrow$ `transitions`(G)
while T is not empty **do**
    take t from T
    `is_evaluable`, $X_u, X_w, Y \leftarrow$ `can_be_evaluated`(t) ▷ See Algorithm (8)
    if `is_evaluable` **then**
        if $t \in E_m$ and t_f is linear and t is not a key schedule transition **then**
            Add constraints that model propagation with a probability of one.
        end if
        $C \leftarrow C \cup \{X_u\}$
        remove t from T
    end if
end while

The `inversion` algorithm (Algorithm 7) is guaranteed to terminate due to the invertibility of the encryption function (as per Definition 4.2), which ensures all nodes are reachable. The `can_be_evaluated` helper function in its basic form is described in Algorithm 8.

4.2.3 The Output

The output of the Step 1 is a file JSON that includes:

- the upper characteristic E_0E_m with the minus logarithm in base 2 of its truncated probability, called upper probability and computed by the active S-box in E_0;
- the lower characteristic E_mE_1 with the minus logarithm in base 2 of its truncated probability, called lower probability and computed by the active S-box in E_1;
- r the minus logarithm in base 2 of the E_m truncated probability, computed by the common active S-boxes in E_0E_m and E_mE_1;
- the minus logarithm in base 2 of the total probability (4.1).

Remark. *Observe that the middle part in E_0E_m could be different from the one in E_mE_1 but the important part is to notice when both of them are simultaneously active.*

Algorithm 8 `can_be_evaluated`

Require: A transition t, the set of variables that can already be calculated C.

$X_w \leftarrow \emptyset$
$X_u \leftarrow \emptyset$
for each input variable x in t_X **do**
    if $x \in C$ **then**
        $X_w \leftarrow X_w \cup \{x\}$
    else
        $X_u \leftarrow X_u \cup \{x\}$
    end if
end for
$Y \leftarrow \emptyset$
for each output variable y in t_Y **do**
    if $y \in C$ **then**
        $Y \leftarrow \{y\}$
    else
        $X_u \leftarrow \{y\}$
    end if
end for
`is_evaluable` $\leftarrow (\#X_u = 1)$
return `is_evaluable`, X_u, X_w, Y

4.3 Step 2: Computing Boomerang Probability

The probability evaluation of a boomerang attack, in its "sandwich" configuration (see subsection 2.5.3), is based on the analysis of the three distinct components of the cipher, the initial part E_0, the middle part E_m and the final part E_1. The overall success probability of the attack is estimated using the formula : $P = p^2q^2r$, an approach detailed in [HNE22]. The components of this formula are defined as follows:

- p represents the probability of the differential characteristic for the upper part, E_0.
- q represents the probability of the differential characteristic for the lower part, E_1.
- r is an experimentally evaluation that quantifies the probability of compatibility between the two differential characteristic through the middle part, E_m.

To compute this probability, we must consider several scenarios, depending on whether the components E_0, E_m and E_1 are empty.

Algorithm 9 `EvaluateEm`

Require: The number of keys to try N_K, the number of quartets to try for each key N_Q, the input difference of E_m β, the output difference of E_m γ, and the partial cipher block E with it associated decryption function D.

counter $\leftarrow 0$
repeat N_K **times**
  $K \leftarrow$ `random_value`()
  repeat N_Q **times**
    $M^1 \leftarrow$ `random_value`()
    $M^2 \leftarrow M^1 \oplus \alpha$
    $(C^1, C^2) \leftarrow (E_K(M^1), E_K(M^2))$
    $(C^3, C^4) \leftarrow (C^1 \oplus \beta, C^2 \oplus \beta)$
    $(M^3, M^4) \leftarrow (D_K(C^3), D_K(C^4))$
    if $M^3 \oplus M^4 = \alpha$ **then**
      counter $\leftarrow$ *counter* $+ 1$
    end if
  end repeat
end repeat
return $-\log_2(\frac{counter}{N_K N_Q})$

Exclusion of the Case Where E_m is empty. Within the strict framework of the "sandwich" attack, the middle component E_m must be non-empty by definition. A configuration where the size of E_m is null would not correspond to this attack model, but rather to earlier boomerang attack constructions. Such models are considered inadequate for our analysis. Therefore, we dismiss this case and require the middle part to be always non-empty.

4.3.1 Analysis of E_0 and E_1 Configurations.

We distinguish three main cases based on the structure of the E_0 and E_1 parts.

Case 1: E_0 and E_1 are non-empty. This is the standard configuration for the sandwich attack. The analysis proceeds as follows:

1. Evaluation of p and q: We employ standard techniques of Tagada for finding differential trails and the optimal differential characteristics for E_0 and E_1. The associated probabilitites p and q are then completely determined.
2. Experimental Estimation of r: The probability r is estimated empirically. More precisely, the output difference of the characteristic over E_0 and the input difference of the characteristic over E_1 are fixed. Experiments are conducted on E_m to count the number of valid pairs that verify the boomerang and thereby derive a

robust estimate of r. The input difference of r is set to the output difference of E_0 and the output difference of r is set to the input difference of E_1.

Case 2: E_0 and E_1 are empty. In this configuration, the attack is reduced to the analysis of the middle part E_m. The boomerang probability is therefore entirely contained within the probability r, while p and q are, by definition, equal to 1. Since the characteristics for E_0 and E_1 are empty, the input and output differences of E_m are unconstrained (in value). The estimation of r is thus performed by using randomly differences for the active input and output word differences.

Case 3: E_0 is empty or E_1 is empty. This asymmetric case occurs when one of the two outer parts is empty.

- If E_0 is empty (resp. E_1), its probability p (resp. q) is equal to 1. It is therefore not necessary to constraint the active input (resp. output) word differences of the middle part.
- The analysis focuses on finding an optimal differential characteristic for the non-empty part E_0 (resp. E_1), which gives us the probability p (resp. q).
- For the experimental estimation of r, since the part corresponding to the empty characteristic has no fixed differential values, random values are used for the actives bits in the input (if E_0 is empty) or the output (if E_1 is empty) of the middle part E_m.

The details of the two sub-cases are given in Appendix B of [BMR26].

4.3.2 Step 2: Optimization

This section details the optimization process for the second step of the boomerang attack. The primary task in Step 2 is to select concrete instantiations of the differential characteristics found for E_0 and E_1. The optimization goal is to make the selection in a way that maximizes the final probability of the distinguisher, $P = p^2q^2r$.

The key of this optimization lies in the compatibility factor, r. While the probabilities p and q are fixed by the chosen characteristics, a single truncated characteristic can often be realized by several different concrete instantiations. Each instantiation can alter the input/output differences for the middle part, E_m, which directly impacts the value of r.

Our goal is to find the specific pair of instantiations for E_0 and E_1 that yields the highest possible r. However, exhaustively testing every combination is computationally infeasible due to the huge time requirements.

To solve this efficiently, we use a heuristic-based filtering algorithm (see Algorithm 12).

This algorithm cleverly prunes the search space by identifying the most promising pairs of differences for E_m and by removing the pairs that lead to have the same E_m values. It operates using a score function, which ranks pairs based on their likelihood of producing a high r value.

This scoring approach is guided by:

- the "cluster effect" noted in [HBS21]: difference pairs that appear frequently across many instantiations tend to be the most effective for the distinguisher and
- the "BCT effect" (see subsection 4.3.3)

By focusing our experimental evaluation only on these high-scoring pairs, we can efficiently find a near-optimal configuration.

If is worth noticing that the filter can be disabled by setting the score threshold to $\omega = -1$, which would cause the algorithm to evaluate all possible pairs.

4.3.3 Heuristic on the Optimization

As we mentioned before to analyze the probability of the propagation of the boomerang differentials, i.e. the boomerang switch, for one round in the middle, in 2018 it was introduced the BCT (Boomerang Connectivity Table) [CHP+18]. After that, there were introduced other tables in the case of multiple round as we saw in subsection 2.5.4. In Section 4.3.2, we introduced the filtering Algorithm 12 in which we didn't define the score function. Here, we have some heuristically proposals that reduce the computational time and find the same optimal solution. Supposing, that we have a state of n words than:

$$score(s^{\uparrow}, s^{\downarrow}) = score(\texttt{output}(s^{\uparrow}), \texttt{input}(s^{\downarrow})) = \prod_{i=1}^{n} \prod_{j=1}^{n} f(\texttt{output}(s^{\uparrow})_i, \texttt{input}(s^{\downarrow})_j)$$

For the Warp and Twine, since they are both generalized Feistel cipher we set

$$f(x, y) = \mathcal{FB}_S(x, y)$$

where S is the S-Box regarding the specific cipher and $\mathcal{FB}$ is the Feistel Boomerang Connectivity Table (see Definition 2.20).

For Skinny, instead we set:

$$f(x, y) = \mathcal{B}_S(x, y)$$

where S is the Skinny S-Box and $\mathcal{B}$ is the Boomerang Connectivity Table (see Definition 2.18).

We keep the pair with the score greater of a certain threshold α:

$$f(x, y) \geq \alpha$$

Since the values of tables are greater than 0 when we set $\alpha = 0$, it means we are not applying any further filtration. If α is not mentioned in the tables of the results it means that it is set to 0.

4.3.4 The Output

The output of the Step 2 is a file JSON that includes:

- the upper differential characteristic E_0 with the minus logarithm in base 2 of its probability p, called upper probability and computed by the active S-box in E_0;
- the lower differential characteristic E_1 with the minus logarithm in base 2 of its probability q, called lower probability and computed by the active S-box in E_1;
- r the minus logarithm in base 2 of the E_m truncated probability, computed by the distinguisher as described at the beginning of the Section (2.5.3);
- the minus logarithm in base 2 of the total probability (4.1).

4.4 Results

There exist two major tools for computing boomerang distinguishers: the one[2] [HBS21] and the one[3] by [DDV20b]. In this Section, we compare our automatic tool with existing ones that are available in line.

Note that, as we could not exhibit the exact behaviour of what is happening in E_m, our results could differ from the ones of [HBS21] and of [DDV20b]. The way, we choose the boomerang inside E_m and the way we compute the "best" probability are empirical. Note also that a boomerang distinguisher is pertinent as soon as its probability is under the size of the plaintext and/or the key (to mount the corresponding attack). This is why we only give results below this limit, except in the case of SKINNY-128 for 14 rounds, where we hope to reach the limit, but ultimately do not.

Our main results for various ciphers are given in Table 4.1. Note that, as we could not exhibit the exact behaviour of what is happening in E_m, our results could differ from the ones of [HBS21, DDV20b, LMR22, HNE22]. All experiments were executed on the *Grid5000* "gros" cluster, using a node equipped with four **Intel Xeon Gold 6240L**

[2]see `https://github.com/hadipourh/Boomerang`.

[3]see `https://gitlab.inria.fr/pderbez/boomerangskinny`.

Table 4.1 Results for boomerang distinguishers for the three considered ciphers on Rnds rounds. The overall probability is given by p^2q^2r. The time is given in minutes. $(*)^a$ means that we have a score greater or equal than 0.1 while $(*)^b$ that the score is greater or equal than 0.5. For the results of [DDV20b], we provided two probabilities: (1) the probability computed without an extra step used to improve the probability accuracy and (2) the original paper results. The execution time with the † marker represents the time to compute the probability (1).

Cipher	Rnds	p	q	r	p^2q^2r	Reference	Time (in minutes)
SKINNY-64	11				$2^{-80.0}$ ($2^{-59.23}$)	[DDV20b]	161m †
	11	$2^{-8.00}$	$2^{-8.00}$	$2^{-24.00}$	$2^{-56.00}$	Our Work	62m
SKINNY-128	13				2^{-122} ($2^{-112.53}$)	[DDV20b]	700m †
	13	$2^{-22.00}$	$2^{-22.00}$	$2^{-24.00}$	$2^{-112.00}$	Our Work	713m
	14				2^{-158} ($2^{-128.52}$)	[DDV20b]	547m †
	14	$2^{-22.00}$	$2^{-42.00}$	$2^{-24.00}$	$2^{-152.00}$	Our Work	3600m
TWINE	15	2^{-8}	2^{-8}	$2^{-19.03}$	$2^{-51.03}$	[HNE22]	1m
	15	$2^{-4.00}$	$2^{-8.00}$	$2^{-24.00}$	$2^{-48.00}$	Our Work	271m
	16	2^{-8}	2^{-8}	$2^{-26.04}$	$2^{-58.04}$	[HNE22]	305m
	16	$2^{-8.00}$	$2^{-8.00}$	$2^{-24.00}$	$(2^{-56.00})^b$	Our Work	371m
	16	$2^{-8.00}$	$2^{-8.00}$	$2^{-23.00}$	$(2^{-55.00})^a$	Our Work	3180m
WARP	21	2^{-10}	2^{-19}	$2^{-26.55}$	$2^{-84.55}$	[HBS21]	1m
					2^{-104}	[LMR22]	1270m
	21	$2^{-14.00}$	$2^{-20.00}$	$2^{-23.00}$	$2^{-91.00}$	Our Work	105m
	22	2^{-16}	2^{-19}	$2^{-26.55}$	$2^{-96.55}$	[HBS21]	180m
					2^{-120}	[LMR22]	3014m
	22	$2^{-16.00}$	$2^{-20.00}$	$2^{-24.00}$	$2^{-96.00}$	Our Work	361m
	23	$2^{-24.00}$	$2^{-20.00}$	$2^{-27.59}$	$2^{-115.59}$	[HBS21]	563m
	23	$2^{-16.00}$	$2^{-22.00}$	$2^{-24.00}$	$2^{-100.00}$	Our Work	2400m

CPUs (18 cores per CPU, 2.60 GHz). Unless otherwise specified, we used a single CPU (18 cores) for our experiments. The complete results will be available soon.

4.5 Conclusions and Open Questions

In this thesis, we have seen how to extend Tagada, originally dedicated to differential cryptanalysis, in order to support boomerang-style attacks through the integration of the sandwich framework. We showed how to model and combine the different components of a boomerang distinguisher in a unified and automated way, and how to leverage this model to search for new distinguishers and to improve existing ones. Our experiments confirm that the resulting framework is able to recover known results and to find

new meaningful distinguishers, suggesting that automated tools can play a central role in the analysis of advanced differential attacks.

It is of crucial importance for the symmetric cryptographic community to develop tools that allow for automatic and systematic bounds on the security of primitives. In this respect, the extension of Tagada to boomerang and sandwich attacks represents a first step toward a more comprehensive automated evaluation of differential security.

Despite these encouraging results, a number of open questions remain.

- *Integration of related-key variants.* In this work, we focused on the single-key setting. An important extension consists in integrating the related-key variant, i.e. injecting the difference also in the round keys, into the framework, in order to automatically search for and evaluate related-key boomerang distinguishers.
- *Beyond distinguishers.* At present, the framework is limited to the construction of distinguishers. Integrating key-recovery attacks on top of these distinguishers, and automating the full attack pipeline, remains an open problem and next step to do.
- *Completeness of the attack model.* The current modeling assumes idealized conditions for the composition of the upper and lower trails and for the treatment of dependencies between rounds. Understanding how much these assumptions affect the tightness of the resulting bounds, and how to refine the model to obtain more accurate estimates, is an open direction.
- *Scalability and performance.* While the approach scales well to several lightweight block ciphers, it remains to be explored how it behaves on larger designs or in settings where only very weak boomerang biases are available.
- *Toward tight automatic security bounds.* Ultimately, a major open goal is to move from the automatic generation of distinguishers to a framework that provides tight and reliable security bounds for symmetric primitives, combining differential, boomerang, and related-key analyses in a unified way.

We believe that addressing these questions will be key to turning automated boomerang cryptanalysis into a practical tool for the systematic and reliable evaluation of symmetric-key designs.

Algorithm 10 `Tagada Boomerang Case1`.

Require: The Specification Graph of the cipher: SG, the Truncated Differential Graph of the full cipher: ΔG, the Differential Graph of E_0: $\delta G^{\uparrow}$ and the Differential Graph of E_1: $\delta G^{\downarrow}$.

$seen \leftarrow \{\}$
$(p^2q^2r_{min}, p^2q^2r_{max}) \leftarrow (0, 1)$
$({s_2}^{\uparrow *}, {s_2}^{\downarrow *}) \leftarrow (None, None)$
$s_1 \leftarrow \texttt{Step1}(\Delta G, seen, p^2q^2r_{max})$
$p^2q^2r_{max} \leftarrow \texttt{TruncatedBoomerangProbability}(s_1)$
while $p^2q^2r_{min} < p^2q^2r_{max}$ **do**
    $E_0 \leftarrow \texttt{UpperCharacteristic}(s_1)$
    $p, {S_2}^{\uparrow} \leftarrow \texttt{Step2}(\delta G^{\uparrow}, E_0)$ ▷ Returns all the possible differentials of E_0 with the highest probability p.
    $E_1 \leftarrow \texttt{LowerCharacteristic}(s1)$
    $q, {S_2}^{\downarrow} \leftarrow \texttt{Step2}(\delta G^{\downarrow}, E_1)$ ▷ Returns all the possible differentials of E_1 with the highest probability q.
    $seen \leftarrow \texttt{UpdateSeen}({S_2}^{\uparrow}, E_0, {S_2}^{\downarrow}, E_1, seen)$ ▷ See Algorithm (11)
    $S_2 \leftarrow \texttt{Filtering}({S_2}^{\uparrow}, {S_2}^{\downarrow}, \alpha)$ ▷ See Algorithm (12)
    $(\bar{r}, {s_2}^{\uparrow}, {s_2}^{\downarrow}) \leftarrow (0, None, None)$
    for each pair $({s'_2}^{\uparrow}, {s'_2}^{\downarrow}) \in S_2$ **do**
        $\bar{r}' \leftarrow \texttt{EvaluateEm}({s'_2}^{\uparrow}, {s'_2}^{\downarrow}, SG)$ ▷ See Algorithm (9)
        if $p^2 \cdot q^2 \cdot \bar{r} < p^2 \cdot q^2 \cdot \bar{r}'$ **then**
            $(\bar{r}, {s_2}^{\uparrow}, {s_2}^{\downarrow}) \leftarrow (\bar{r}', {s'_2}^{\uparrow}, {s'_2}^{\downarrow})$
        end if
    end for
    if $p^2q^2r_{min} < p^2 \cdot q^2 \cdot \bar{r}$ **then**
        $(p^2q^2r_{min}, {s_2}^{\uparrow *}, {s_2}^{\downarrow *}) \leftarrow (p^2 \cdot q^2 \cdot \bar{r}, {s_2}^{\uparrow}, {s_2}^{\downarrow})$
    end if
    if $p^2q^2r_{min} < p^2q^2r_{max}$ **then**
        $s_1 \leftarrow \texttt{NextTruncatedDifferential}(\Delta G, seen, p^2q^2r_{max})$
        if s_1 is `None` **then**
            $p^2q^2r_{max} \leftarrow \texttt{NextTheoreticalUpperBound}(\Delta G, p^2q^2r_{max})$
            if $p^2q^2r_{min} < p^2q^2r_{max}$ **then**
                break
            end if
            $s_1 \leftarrow \texttt{Step1}(\Delta G, seen, p^2q^2r_{max})$
            $p^2q^2r_{max} \leftarrow \texttt{TruncatedBoomerangProbability}(s1)$
        end if
    end if
end while
return $({s_2}^{\uparrow *}, {s_2}^{\downarrow *}, p^2q^2r_{min})$

Algorithm 11 `UpdateSeen`

Require: The set of all upper instantiations: ${S_2}^{\uparrow}$ and their associated truncated differential characteristic ${s_1}^{\uparrow}$, the set of all lower instantiations: ${S_2}^{\downarrow}$ and their associated truncated differential characteristic ${s_1}^{\downarrow}$, the set of already seen solutions: *seen*.

if ${S_2}^{\uparrow} \neq \emptyset$ and ${S_2}^{\downarrow} \neq \emptyset$ **then**
    $seen \leftarrow seen \cup \{({s_1}^{\uparrow}, {s_1}^{\downarrow})\}$ ▷ We eliminate the couple.
else if ${S_2}^{\uparrow} \neq \emptyset$ and ${S_2}^{\downarrow} = \emptyset$ **then**
    $seen \leftarrow seen \cup \{{s_1}^{\downarrow}\}$ ▷ We eliminate just the lower solutions.
else if ${S_2}^{\uparrow} \neq \emptyset$ and ${S_2}^{\downarrow} \neq \emptyset$ **then**
    $seen \leftarrow seen \cup \{{s_1}^{\uparrow}\}$ ▷ We eliminate just the upper solutions.
else
    $seen \leftarrow seen \cup \{{s_1}^{\uparrow}\}$
    $seen \leftarrow seen \cup \{{s_1}^{\downarrow}\}$
end if
return `seen`

Algorithm 12 `Filtering`

Require: The set of all upper instantiations: ${S_2}^{\uparrow}$, the list of all lower instantiations ${S_2}^{\downarrow}$, the threshold probability: ω.

function UniqueBy(S, f)
Returns a new set containing only unique elements from S, based on the identifier extracted by function f.
Require: A set S, and a key extraction function $f : S \to ID$.
Ensure: A set $S' \subseteq S$ containing the unique elements of S with respect to the identifiers returned by f.
    $S' \leftarrow \emptyset$ ▷ The set to store filtered unique items.
    $ID_{seen} \leftarrow \emptyset$ ▷ The set of identifiers already encountered.
    for each element $s \in S$ **do**
        $id \leftarrow f(s)$
        if $id \notin ID_{seen}$ **then**
            $ID_{seen} \leftarrow ID_{seen} \cup \{id\}$
            $S' \leftarrow S' \cup \{s\}$
        end if
    end for
    return S'
end function

${S'_2}^{\uparrow} \leftarrow$ UniqueBy(${S_2}^{\uparrow}, output$)
${S'_2}^{\downarrow} \leftarrow$ UniqueBy(${S_2}^{\downarrow}, input$)
$result = \{(s^{\uparrow}, s^{\downarrow}) \in {S'_2}^{\uparrow} \times {S'_2}^{\downarrow} \mid score(s^{\uparrow}, s^{\downarrow}) \geq \omega\}$
return *result*

Part II

Automatization through Machine Learning

Chapter 5

Introduction to Machine Learning

The ability of artificial systems to acquire knowledge directly from data, rather than relying on hard-coded rules, represents one of the most significant shifts in the history of computer science. The difficulties faced by systems relying on manual programming suggest that AI systems need the ability to acquire their own knowledge by extracting patterns from raw data. This capability is known as **Machine Learning (ML)**. As a field of Artificial Intelligence (AI), ML focuses on the automatization of the learning process; specifically, we aim to avoid developing programs by hand, preferring instead that algorithms learn by themselves what to do.

In 1959, Arthur Samuel described Machine Learning as the "field of study that gives computers the ability to learn without being explicitly programmed" [Sam59]. He concluded that programming computers to learn from experience should eventually eliminate the need for much of the detailed programming effort required by traditional approaches. Decades later, Tom Mitchell provided a more formal definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [Mit97].

Historically, the pursuit of machine intelligence dates back to Alan Turing's seminal paper [Tur50], which introduced a benchmark for demonstrating machine intelligence indistinguishable from a human being. Deep learning, specifically, emerged from attempts to model biological learning, leading to the development of **Artificial Neural Networks (ANNs)** [Hin91]. Early work during the cybernetics era introduced linear models such as the McCulloch–Pitts neuron [MP43], the Perceptron [Ros58], and ADALINE [WH60]. Despite their influence, these linear approaches were limited—notably failing to represent the XOR function—which led to significant criticism and a decline

in neural network research [MP69].

Research revived in the 1980s with the rise of **connectionism** [RMtPRG86, MRH04], which emphasized distributed representations and demonstrated the effectiveness of back-propagation for training deep networks [HRWR02, LSF87]. Further progress in the 1990s advanced neural sequence modeling [Hoc91, BSF94]. The "third wave" began in 2006 with the successful training of deep belief networks [HOT06] and theoretical analyses highlighting the importance of depth [BL07, DB11]. This era established the term **Deep Learning**, marking the point where deep neural networks began to surpass alternative AI methods.

We believe a good book that provides the theoretical framework necessary to understand, implement, and optimize these systems is "Deep Learning" by I. Goodfellow et al. [GBC16]. This chapter provides the necessary elements from [GBC16] to develop our work.

We begin in **Section 5.1** by defining the essential mathematical tools—derivatives, gradients, Hessian matrices, and Jacobians—required to navigate the parameter space of learning models.

In **Section 5.2**, we formalize the statistical principles of learning. We define the fundamental goal of **generalization**, which distinguishes learning from pure optimization, and introduce the **Maximum Likelihood Estimation (MLE)** framework. This section also addresses critical concepts such as the data generating distribution, the bias-variance trade-off in estimators, and the role of hyperparameters and validation sets in preventing overfitting.

Section 5.3 delves into the algorithms that power the training process. Since analytical solutions are rarely available for deep models, we rely on iterative optimization. We explore **Stochastic Gradient Descent (SGD)** [RM51] and its importance in handling large datasets. We then examine advanced techniques designed to accelerate convergence, including **Momentum** [Pol64], Nesterov's acceleration [SMDH13], and adaptive learning rate methods such as **AdaGrad**, **RMSProp**, **ADAM** [KB14], and **AMSGRAD** [RKK19]. Strategies to stabilize training, such as **Batch Normalization** [IS15] and Curriculum Learning [YLC+09], are also discussed.

Building on these optimization foundations, **Section 5.4** introduces **Deep Feedforward Networks** (or MLPs), the quintessential deep learning models designed to approximate functions. We discuss their structural components, including the choice of cost functions (e.g., Cross-Entropy) and activation functions (e.g., ReLU, Sigmoid), and detail the **Back-Propagation** algorithm [RHW86b] used to compute gradients efficiently. To ensure these models generalize well, **Section 5.5** covers regularization techniques such as parameter norm penalties (L_1, L_2) and Early Stopping.

Finally, we explore specialized architectures tailored for specific data modalities. **Sec-**

tion 5.6 presents **Convolutional Neural Networks (CNNs)** [LBH+98], which leverage sparse interactions, parameter sharing, and equivariant representations to process grid-like data such as images. We also cover variants like Dilated CNNs and pooling mechanisms. Concluding the chapter, **Section 5.7** introduces **Recurrent Neural Networks (RNNs)** [RHW86b], architectures designed with feedback loops to process sequential data and capture long-term dependencies.

5.1 Definitions

We give some useful definition and remarks to understand the theoretical part for the rest of the chapter

Definition 5.1 (Derivative)**.** Suppose we have a function $y = f(x)$ with $f : \mathbb{R} \to \mathbb{R}$. Then,

$$f'(x) := \lim_{\Delta \to 0} \frac{f(x + \Delta) - f(x)}{\Delta}$$

is called **derivative** of that function f. It is also denoted as $\frac{dy}{dx}$.

The first derivative specifies how to scale a small change in the input in order to obtain the corresponding change in the output: $f(x+\epsilon) \approx f(x)+f'(x)\epsilon$. A function $f : \mathbb{R} \to \mathbb{R}$ is said to be **derivable** if there exist and it is finite $f'(x)\ \forall x \in D \subseteq \mathbb{R}$.

Definition 5.2 (Critical Points)**.** Let $f : D \to \mathbb{R}$, with $D \subseteq \mathbb{R}$ be a derivable function. Then, $x^* \in D$ is said to be a **critical point** or a **stationary point** if

$$f'(x) = 0$$

A critical point is said to be a **local minimum** if it is a point where $f(x)$is lower than at all neighboring points, while it is said to be a **local maximum** if it is a point where $f(x)$ is higher than at all neighboring points. Some critical points are neither maxima nor minima. These are known as **saddle points**. A point that obtains the absolute lowest value of $f(x)$ is a **global minimum**.

Definition 5.3 (Gradient)**.** Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function with multiple inputs:

$$f(x_1, \ldots, x_n) = y$$

Then, the **partial derivative** in x_i is denoted by:

$$\frac{\partial}{\partial x_i} f(x)$$

It is the derivative of f in respect of x_i considering the other variables as constants.

The **gradient** $\nabla_x f(x)$ is the vector containing all of the partial derivatives:

$$\nabla_x f(x) = \left(\frac{\partial}{\partial x_1} f(x), \ldots, \frac{\partial}{\partial x_n} f(x) \right)^T$$

Definition 5.4 (Directional Derivative)**.** The **directional derivative** in direction u is the derivative of the function $f(x + \alpha u)$ with respect to α evaluated at $\alpha = 0$. Using the chain rule:

$$\left(\frac{\partial}{\partial \alpha} f(x + \alpha u) \right)_{\alpha=0} = \left(u^T \nabla_x f(x + \alpha u) \right)_{\alpha=0} = u^T \nabla_x f(x)$$

Definition 5.5 (Second Derivative)**.** Let $f : \mathbb{R} \to \mathbb{R}$ be a derivable function. Then, the **second derivative** is the derivative of the derivative and it is denoted by

$$f''(x) := (f'(x))' \quad \text{or} \quad \frac{d^2}{dx^2} f = \frac{d}{dx} \left(\frac{d}{dx} f \right)$$

The second derivative tells us how the first derivative changes as we vary the input. This is important because it tells us whether a gradient step causes as much of an improvement as we would expect based on the gradient alone. We can think of the second derivative as measuring **curvature**.

Definition 5.6 (Hessian)**.** Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function with multiple inputs. In this case, to denote the second derivative we use the **Hessian matirx** $H(f)(x)$:

$$H(f)(x)_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(x) = \frac{\partial}{\partial x_i} \left(\frac{\partial}{\partial x_j} f(x) \right)$$

It is important to notice that the differential operators are commutative is commutative where the second partial derivatives are continuous:

$$\frac{\partial^2}{\partial x_i \partial x_j} f(x) = \frac{\partial^2}{\partial x_j \partial x_i} f(x)$$

hence, $H_{i.j} = H_{j,i}$, the Hessian matrix is a **symmetric** matrix.

Definition 5.7 (Eigenvector)**.** Let $A \in \mathbb{R}^{n \times n}$ be a quadratic matrix. The **eigenvector** of A are the vectors $v \in \mathbb{R}^n$ such that:

$$Av = \lambda v$$

$\lambda \in \mathbb{R} \setminus \{0\}$ is said to be an **eigenvalue**.

t is possible to use the second derivative also to determine if the critical points are minimum, maximums or saddle points, this is called the **second derivative test**. In

the case of $f : \mathbb{R} \to \mathbb{R}$:

- if $f''(x) > 0$ and $f'(x) = 0$ then x is a local minimum;
- if $f''(x) < 0$ and $f'(x) = 0$ then x is a local maximum,
- if $f''(x) = 0$ and $f'(x) = 0$ we don't know.

In the case of $f : \mathbb{R}^n \to \mathbb{R}$, denoting with $\underline{0} = (0, \dots, 0) \in \mathbb{R}^n$ the null vector:

- if $\nabla_x f(x) = \underline{0}$ and H is positive defined i.e. $v^T H v > 0$ for all $v \in \mathbb{R}^n$ (or if all the eigenvalues are positive), then $x \in \mathbb{R}^n$ is a local minimum;
- if $\nabla_x f(x) = \underline{0}$ and H is negative defined i.e. $v^T H v < 0$ for all $v \in \mathbb{R}^n$ (or if all the eigenvalues are negative), then $x \in \mathbb{R}^n$ is a local maximum;
- if $\nabla_x f(x) = \underline{0}$ and H has at least one positive and one negative eigenvalues, then $x \in \mathbb{R}^n$ is a saddle point;
- if $\nabla_x f(x) = \underline{0}$ but H has all the non-zero eigenvalues positive (or negative) and at least one eigenvalue zero then the test is inconclusive.

Definition 5.8 (Orthogonal)**.** Two vectors $v_1, v_2 \in \mathbb{R}^n$ are said to be **orthogonal** if

$$v_1^T v_2 = 0$$

Furthermore, they are said to be **orthonormal** if they are orthogonal and

$$||v_1||_2 = ||v_2||_2 = 1$$

Definition 5.9 (Basis)**.** A set of $\{v_1, \dots, v_m\}$ where $v_i \in \mathbb{R}^n$ is said to be a **basis** if they are linearly independent and for all $x \in \mathbb{R}^n$ we can write

$$x = \sum_{i=1}^{m} c_i v_i$$

with $c_i \in \mathbb{R}$.

Definition 5.10 (Orthonormal)**.** A matrix $A \in \mathbb{R}^{n \times n}$ is said to be an **orthogonal** matrix if:

$$A^{-1} = A^T$$

Furthermore, it is an **orthonormal** matrix if all the rows (and the columns) are an orthonormal basis.

Definition 5.11 (Diagonalizable)**.** Let $A \in \mathbb{R}^{n \times n}$ be a matrix. A is said to be **diagonalizable** if there exists $D \in \mathbb{R}^{n \times n}$ invertible such that:

$$\Sigma = D^{-1} A D$$

where $\Sigma \in R^{n \times n}$ is a diagonal matrix.

Remark. *It is proven in the Spectral theorem of Hilbert (for a clear explanation check [Rud91]) that in the Real space, every symmetric matrix can be diagonalizable by an*

orthogonal transformation. Furthermore, it exists an orthonormal basis of eigenvectors in which the matrix becomes diagonal with the eigenvalues on the diagonal: there exist a matrix Q such that

$$D = Q^T A Q$$

where D is a diagonal matrix with the eigenvalues $\lambda_i \in \mathbb{R}$ on the diagonal and Q is composed by the eigenvectors that form an orthogonal basis.

Definition 5.12 (Jacobian)**.** Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be a function such that: $(y_1, \ldots, y_n) = f(x_1, \ldots, x_m) = (f_1(x_1, \ldots, x_m), \ldots, f_n(x_1, \ldots, x_m))$. Then, the **Jacobian matrix** is the matrix $J \in \mathbb{R}^{n \times m}$ with entries

$$J_{i,j} = \frac{\partial}{\partial x_j} f_i(x)$$

In particular, the entry (i, j) is the derivative of the i-th function with respect to the variable x_j.

5.2 Machine Learning

The central challenge in machine learning is that it must perform well on new, previously unseen inputs, not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called **generalization**. Typically, when training a machine learning model:

- We have access to a **training set**, $(X^{(train)}, y^{(train)})$;
- We can compute some error measure on the training set called the **training error** or **loss function** or **cost function**, $\mathcal{L}_{train}(\theta)$;

The training is the process during which we reduce and, hopefully, minimize this training error. So far, what we have described is simply an optimization problem. What separates machine learning from optimization is that we want the **generalization error**, also called the **test error**,

$$\mathcal{L}_{test}(\theta)$$

to be low as well. So, we have to find the value θ such that:

$$\theta^* = \arg\min_{\theta} \mathcal{L}_{test}(\theta)$$

We show an example of how this could work.

Linear Regression. The goal is to build a system that can take a vector $x \in \mathbb{R}^n$ as input and predict the value of a scalar $y \in \mathbb{R}$ as its output. In the case of linear regression, the output is a linear function of the input. Let $\hat{y}$ be the value that our

model predicts y should take on. We define the output to be

$$\hat{y} = w^T x = \sum_{i=1}^{n} w_i x_i$$

where $w \in \mathbb{R}^n$ is a vector of parameters. Parameters are values that control the behavior of the system. In this case, w_i is the coefficient that we multiply by feature x_i before summing up the contributions from all the features. To follow the definition of Mitchell, we need a task, a performance measure and the experience or training.

The Task. Predict y from x by outputting $y = f(x) = w^T x$. The algorithm creates a function $\hat{f}$ such that $\hat{f}_w(x) = \hat{y}$

The performance measure. Given m pairs (x_i, y_i) with $x_i \in \mathbb{R}^n$ and $y \in \mathbb{R}$, we have $(X^{(test)}, y^{(test)})$

$$\mathcal{L}_{test}(w) = \frac{1}{m} ||\hat{y}^{(test)} - y^{(test)}||_2^2$$

where $\hat{y}_i^{(test)} = \hat{f}_w(x_i^{(test)})$.

Training. Given **new** m pairs (x_i, y_i) with $x_i \in \mathbb{R}^n$ and $y \in \mathbb{R}$: $(X^{(train)}, y^{(train)})$, we try to minimize:

$$\mathcal{L}_{train}(w) = \frac{1}{m} ||\hat{y}^{(train)} - y^{(train)}||_2^2$$

We try to find the minimum setting the gradient equal to 0:

$$\nabla_w \mathcal{L}_{train} = 0 \implies w^* = \left((X^{(train)})^T X^{(train)} \right)^{-1} (X^{(train)})^T y^{(train)}$$

The system of equations whose solution is given by the last equation is known as the **normal equations**. Evaluating that equation constitutes a simple learning algorithm:

$$\mathcal{L}_{test}(w^*) = \frac{1}{m} ||\hat{f}_{w^*}(X^{(test)}) - y^{(test)}||$$

Data Distribution. The generalization error is defined as the expected value of the error on a new input. Here, the expectation is taken across different possible inputs, drawn from the distribution of inputs we expect the system to encounter in practice. We typically estimate the generalization error of a machine learning model by measuring its performance on a **test set** of examples that were collected separately from the training set. If the training and the test set are collected arbitrarily, there is indeed little we can

do. The train and test data are generated by a probability distribution over datasets called the **data generating process**. We typically make a set of assumptions known collectively as the **independent identical distribution** (**i.i.d.**) **assumptions**. These assumptions are that the examples in each dataset are independent from each other, and that the train set and test set are identically distributed, drawn from the same probability distribution as each other. This assumption allows us to describe the data generating process with a probability distribution over a single example. The same distribution is then used to generate every train example and every test example.
Definition 5.13 (Distribution)**.** We call that shared underlying distribution the **data generating distribution**, denoted:

$$p_{data}$$

We denote the parametric, (that depends on a parameter θ), the family of probability distribution over the same space indexed by θ with:

$$p_{model}(x, \theta)$$

In other words, $p_{model}(x, \theta)$ maps any configuration x to a real number estimating the true probability $p_{data}(x)$. We denote with:

$$\hat{p}_{data}$$

the **empirical distribution** observed by the data. In other words, it is the distribution we approximate with the training data we have.

Hyperparameters. Most machine learning algorithms have several settings that we can use to control the behavior of the learning algorithm. These settings are called **hyperparameters**. The values of hyperparameters are not adapted by the learning algorithm itself (though we can design a nested learning procedure where one learning algorithm learns the best hyperparameters for another learning algorithm). More frequently, the setting must be a hyperparameter because it is not appropriate to learn that hyperparameter on the training set. This applies to all hyperparameters that control model capacity. If learned on the training set, such hyperparameters would always choose the maximum possible model capacity, resulting in overfitting. To solve this problem, we need a **validation set** of examples that the training algorithm does not observe. The validation set must always come from the training data. In practice, we divide the training data into two separate parts. One part is used to actually learn the model parameters, and we still refer to this subset as the training set, even though it is only a portion of the full data originally available for training. The other part is used as a validation set, whose role is to estimate how well the model is likely to generalize and to help adjust the hyperparameters during development. A common split is to use around 80% of the data for training and 20% for validation.

5.2.1 Estimators.

The field of statistics gives us many tools that can be used to achieve the machine learning goal of solving a task not only on the training set but also to generalize.

Point Estimation. Point estimation is the attempt to provide the single *best* prediction of some quantity of interest. In general the quantity of interest can be a single parameter or a vector of parameters in some parametric model but it can also be a whole function. In order to distinguish estimates of parameters from their true value, our convention is to denote a point estimate of a parameter θ by $\hat{\theta}$. Let $\{x^{(1)}, \ldots, x^{(m)}\}$ be a set of m i.i.d. points. A **points estimator** is any function:

$$\hat{\theta}_m = g(x^{(1)}, \ldots, x^{(m)})$$

The two most used point estimators are:

1. **Bias.** The bias of an estimator is defined as:

$$bias(\hat{\theta}_m) = \mathbb{E}(\hat{\theta}_m) - \theta$$

   where the expectation is over the data (seen as samples from a random variable) and θ is the true underlying value of θ used to define the data generating distribution.
2. **Variance and Standard Error.** Another property of the estimator that we might want to consider is how much we expect it to vary as a function of the data sample. Just as we computed the expectation of the estimator to determine its bias, we can compute its variance. The variance of an estimator is simply the variance $Var(\hat{\theta})$, where the random variable is the training set. Alternately, the square root of the variance is called the standard error, denoted $SE(\hat{\theta})$:

$$SE(\hat{\theta}_m) = \sqrt{Var(\hat{\theta}_m)} = \mathbb{E}(\hat{\theta}_m^2) - \mathbb{E}(\hat{\theta}_m)^2$$

Definition 5.14 (Unbiased Estimator)**.** An estimator $\hat{\theta}_m$ is said to be **unbiased** if $bias(\hat{\theta}_m) = 0$, which implies that $\mathbb{E}(\hat{\theta}_m) = \theta$.

It is important to underline also the behavior of an estimator as the amount of training data grows. In particular, we usually wish that, as the number of data points m in our dataset increases, our point estimates converge to the true value of the corresponding parameters. More formally,

Definition 5.15 (Consistency)**.** Let $x^{(1)}, \ldots, x^{(m)} \sim p(\theta)$ be random values i.i.d. and

$\hat{\theta}_m$ be the estimation of θ. If:

$$\lim_{m\to\infty} \mathbb{P}(|\hat{\theta}_m - \theta| > \epsilon) = 0, \quad \forall \epsilon > 0$$

Then, $\hat{\theta}_m$ is called **consistent**.

Maximum Log-Likelihood Estimation. Consider a set of m examples $X = \{x^{(1)}, \ldots, x^{(m)}\}$ drawn independently from the true but unknown data generating distribution $p_{data}(x)$. The **maximum likelihood estimator** θ_{ML} for θ is then defined as

$$\theta_{ML} = \arg\max_{\theta} p_{model}(X, \theta) = \arg\max_{\theta} \prod_{i=1}^{m} p_{model}(x^{(i)}, \theta)$$

This product over many probabilities can be inconvenient for a variety of reasons. For example, it is prone to numerical underflow. Hence, we can apply the logarithm that doesn't change the maximum sign and convert into addition the multiplication. So, we obtain the **maximum log-likelihood estimator**:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(x^{(i)}, \theta)$$

Because the arg max does not change when we rescale the cost function, we can divide by m to obtain a version of the criterion that is expressed as an expectation with respect to the empirical distribution $\hat{p}_{data}$ defined by the training data:

$$\theta_{ML} = \arg\max_{\theta} \frac{1}{m} \sum_{i=1}^{m} \log p_{model}(x^{(i)}, \theta) = \arg\max_{\theta} \mathbb{E}_{x\sim\hat{p}_{data}} \log p_{model}(x, \theta) \tag{5.1}$$

One way to interpret maximum likelihood estimation is to view it as minimizing the dissimilarity between the empirical distribution $\hat{p}_{data}$ defined by the training set and the model distribution, with the degree of dissimilarity between the two measured by the **Kullback Leibler (KL) divergence** [KL51]. The KL divergence is given by

$$D_{KL}(\hat{p}_{data}||p_{model}) = \mathbb{E}_{x\sim\hat{p}_{data}} \left(\log \hat{p}_{data}(x) - \log p_{model}(x)\right)$$

The maximum log-likelihood estimator can be generalized to the case where our goal is to estimate a conditional probability $\mathbb{P}(y|x, \theta)$ in order to predict y given x. This is actually the most common situation because it forms the basis for most supervised learning. If X represents all our inputs and Y all our observed targets and those examples are assumed to be i.i.d., then the conditional maximum log-likelihood estimator

is:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{m} \log \mathbb{P}(y^{(i)}|x^{(i)}, \theta)$$

The maximum likelihood estimator has the property of consistency and under appropriate conditions,

- The true distribution p_{data} must lie within the model family $p_{model}(\cdot, \theta)$. Otherwise, no estimator can recover p_{data};
- The true distribution p_{data} must correspond to exactly one value of θ. Otherwise, maximum likelihood can recover the correct p_{data} , but is not able to determine which value of θ was used by the data generating processing.

Beyond the maximum likelihood estimator, there exist several other inductive principles that yield consistent estimators. However, consistent estimators can vary in their **statistical efficiency**, meaning that some estimators achieve lower generalization error with a fixed number of training samples or, equivalently, require fewer samples to reach a desired level of accuracy. Statistical efficiency is mainly analyzed in the **parametric setting**, where the objective is to estimate unknown parameters rather than entire functions, assuming the true parameters are identifiable. Closeness to the true parameter is commonly measured using the expected mean squared error between the estimated and true parameter values, with the expectation taken over m training samples. This error decreases as m increases, and for sufficiently large m, the Cramér–Rao lower bound [Rao21, Cra46] establishes that no consistent estimator can achieve a lower mean squared error than the maximum likelihood estimator, since it is itself consistent. Consequently, due to its consistency and statistical efficiency, maximum likelihood estimation is widely regarded as a preferred approach in machine learning.

5.3 Optimization Algorithm

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways. Machine learning usually acts indirectly. In most machine learning scenarios, we care about some performance measure P, that is defined with respect to the test set and may also be intractable. We therefore optimize P only indirectly. We reduce a different cost function $\mathcal{L}(\theta)$ in the hope that doing so improves P . This is in contrast to pure optimization, where minimizing $\mathcal{L}$ is a goal in and of itself. Optimization algorithms for training deep models also typically include some specialization on the specific structure of machine learning objective functions. Typically, the cost function or **empirical risk** can be written as an average over the training set, such as:

$$\mathcal{L}(\theta) = \mathbb{E}_{(x,y)\sim\hat{p}_{data}} L(f(x;\theta), y) \tag{5.2}$$

where L is the per example loss function, $f(x, \theta)$ is the predicted output when the input is x, $\hat{p}_{data}$ is the empirical distribution. Equation 5.2 defines an objective function with respect to the training set. We would usually prefer to minimize the corresponding objective function where the expectation is taken across the data generating distribution p_{data} rather than just over the finite training set, in other words the **risk**:

$$\mathcal{L}^*(\theta) = \mathbb{E}_{(x,y)\sim p_{data}} L(f(x;), y)$$

The goal of a machine learning algorithm is to reduce the expected generalization error given by $\mathcal{L}^*$. We emphasize here that the expectation is taken over the true underlying distribution p_{data}. If we knew the true distribution $p_{data}(x, y)$, risk minimization would be an optimization task solvable by an optimization algorithm. However, when we do not know $p_{data}(x, y)$ but only have a training set of samples, we have a machine learning problem. The simplest way to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set. This means replacing the true distribution $p(x, y)$ with the empirical distribution $\hat{p}(x, y)$ defined by the training set:

$$\mathbb{E}_{x,y\sim\hat{p}_{data}(x,y)}[L(f(x, \theta), y] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)}, \theta), y^{(i)}) \tag{5.3}$$

where m is the number of training examples. The training process based on minimizing this average training error is known as **empirical risk minimization**. In this setting, machine learning is still very similar to straightforward optimization. Rather than optimizing the risk directly, we optimize the empirical risk, and hope that the risk decreases significantly as well.

Constraint Optimization. Constrained optimization aims to find minima or maxima of an objective function subject to constraints that define a feasible set S, whose elements are called *feasible points*. A common strategy is to enforce constraints through norm bounds or projection-based methods, where gradient descent steps are followed by a projection back onto the feasible region, or are restricted to remain feasible via line search techniques. Efficiency can be improved by projecting the gradient onto the tangent space of the feasible set before taking optimization steps [Ros60].

An alternative approach transforms the constrained problem into an unconstrained one by reparameterizing the feasible set. While effective in specific cases, this strategy is problem-dependent and lacks generality. A comprehensive and systematic framework for constrained optimization is provided by the **Karush–Kuhn–Tucker (KKT)** theory [Kar14, KT51]. In this framework, the feasible set S is defined using equality constraints $g^{(i)}(x) = 0$ and inequality constraints $h^{(j)}(x) \leq 0$. For each constraint, a corresponding *KKT multiplier* is introduced, leading to the definition of the *generalized*

Lagrangian:

$$\mathcal{L}(x, \lambda, \alpha) = f(x) + \sum_i \lambda_i g^{(i)}(x) + \sum_j \alpha_j h^{(j)}(x).$$

The constrained optimization problem can then be reformulated as a min–max problem over the generalized Lagrangian, ensuring that infeasible points are excluded and that the optimal solution over the feasible set is preserved. Inequality constraints play a crucial role: a constraint is said to be *active* at the solution if it holds with equality, while inactive constraints do not affect the optimal point and correspond to zero-valued KKT multipliers.

The optimality of a solution is characterized by the **KKT conditions**, which are necessary (but not always sufficient) for optimality. These conditions require stationarity of the generalized Lagrangian, feasibility of both the primal variables and the multipliers, and *complementary slackness*, which enforces that each inequality constraint is either active or has a zero multiplier. Together, these properties provide a powerful and general tool for analyzing and solving constrained optimization problems.

Minibatch. A key characteristic of machine learning optimization is that the objective function decomposes as a sum over training examples, making it possible to estimate expectations using only subsets of the data. This structure allows optimization algorithms to update parameters using approximate estimates of the objective and its gradient, computed from small randomly sampled batches of training examples. Approximating gradients in this way significantly reduces computational cost and can accelerate convergence, especially when redundancy exists within the training data. Optimization algorithms that use the entire training set are called **batch** or **deterministic** gradient methods, because they process all of the training examples simultaneously in a large batch. Optimization algorithms that use only a single minibatch at a time are sometimes called **stochastic** or sometimes **online** methods. The term online is usually reserved for the case where the examples are drawn from a stream of continually created examples rather than from a fixed-size training set over which several passes are made. The choice of minibatch size is influenced by several practical and theoretical considerations: Larger minibatches provide more accurate gradient estimates, although the improvement diminishes as batch size increases. Extremely small batches tend to underutilize multicore hardware, motivating the use of a minimum batch size to ensure computational efficiency. Memory constraints also limit batch size, since parallel processing of examples requires memory that scales linearly with the number of samples in the minibatch. Hardware characteristics further affect this choice, as certain architectures—particularly GPUs—often achieve better performance with specific batch sizes, commonly powers of two. Finally, small minibatches can introduce a regularizing effect [WM03] by injecting noise into the learning process, which may improve generalization, though at the cost of increased variance in gradient estimates, reduced learning rates, and longer overall training time. With some datasets

growing rapidly in size, faster than computing power, it is becoming more common for machine learning applications to use each training example only once or even to make an incomplete pass through the training set. When using an extremely large training set, overfitting is not an issue, so underfitting and computational efficiency become the predominant concerns. See also [BB07] for a discussion of the effect of computational bottlenecks on generalization error, as the number of training examples grows.

5.3.1 Gradient Descent Algorithm.

Optimization algorithms that use only the gradient, such as gradient descent, are called **first-order optimization algorithms**. Optimization algorithms that also use the Hessian matrix, such as Newton's method, are called **second-order optimization algorithms** [NW06]. The **gradient descent algorithm** [L.47] is an optimization technique created to find the local (or, hopefully, global) minimum of $f(x)$ under certain hypothesis. In order to minimize f, we would like to find the direction in which f decreases the fastest. We can do this using the directional derivative:

$$\min_{u,u^T u=1} u^T \nabla_x f(x) = \min_{u,u^T u=1} ||u||_2 ||\nabla_x f(x)||_2 \cos\theta$$

where the equality is the definition of scalar product and θ is the angle between u and the gradient. Since $||u||_2 = 1$ and we are minimizing in respect to u it means that we have to search for θ such that $\cos\theta$ is minimum and it happens when u point in the opposite direction of $\nabla_x f(x)$. We can decrease f by moving in the direction of the negative gradient. This is known as the method of **steepest descent** or **gradient descent**. Starting from a point $x' \in \mathbb{R}^n$, then we move to the second one x'' according the formula:

$$x'' = x' - \epsilon \nabla_x f(x')$$

where ϵ is called **learning rate** and it is a positive scalar determining the size of the step. We can choose ϵ in several different ways. A popular approach is to set ϵ to a small constant. Sometimes, we can solve for the step size that makes the directional derivative vanish. Another approach is to evaluate

$$f(x' - \epsilon \nabla_x f(x'))$$

for several values of ϵ and choose the one that results in the smallest objective function value. This last strategy is called a **line search**. Steepest descent converges when every element of the gradient is zero (or, in practice, very close to zero).

To be more accurate, it is possible to also use the **second derivative**. Because the

Hessian matrix is real and symmetric, we can decompose it into a set of real eigenvalues and an orthogonal basis of eigenvectors. The second derivative in a specific direction represented by a unit vector d is given by $d^T H d$. When d is an eigenvector of H, the second derivative in that direction is given by the corresponding eigenvalue:

$$d^T H d = d^T \lambda d = \lambda d^T d = \lambda ||d||_2^2 = \lambda$$

For other directions of d, the directional second derivative is a weighted average of all of the eigenvalues, with weights between 0 and 1, and eigenvectors that have smaller angle with d receiving more weight. The maximum eigenvalue determines the maximum second derivative and the minimum eigenvalue determines the minimum second derivative. The (directional) second derivative tells us how well we can expect a gradient descent step to perform. We can make a second-order Taylor series approximation [Apo67] to the function $f(x)$ around the current point $x^{(0)}$:

$$f(x) \approx f(x^{(0)}) + (x - x^{(0)})^T (\nabla_x f(x^{(0)})) + \frac{1}{2}(x - x^{(0)})^T (H(f)(x^{(0)}))(x - x^{(0)})$$

If we use a learning rate of ϵ, then the new point x is given by $x^{(0)} - \epsilon \nabla_x f(x^{(0)})$. Substituting this into our approximation, we obtain:

$$f(x^{(0)} - \epsilon \nabla_x f(x^{(0)})) \approx$$
$$f(x^{(0)}) - \epsilon (\nabla_x f(x^{(0)}))^T (\nabla_x f(x^{(0)})) + \frac{1}{2}\epsilon^2 (\nabla_x f(x^{(0)}))^T (H(f)(x^{(0)}))(\nabla_x f(x^{(0)}))$$

There are three terms here: the original value of the function, the expected improvement due to the slope of the function, and the correction we must apply to account for the curvature of the function. When $(\nabla_x f(x^{(0)}))^T (H(f)(x^{(0)}))(\nabla_x f(x^{(0)}))$ is zero or negative, the Taylor series approximation predicts that increasing ϵ forever decreases f forever. In practice, the Taylor series is unlikely to remain accurate for large ϵ, so one must resort to more heuristic choices of ϵ this case. When $(\nabla_x f(x^{(0)}))^T (H(f)(x^{(0)}))(\nabla_x f(x^{(0)}))$ is positive, solving for the optimal step size that decreases the Taylor series approximation of the function the most yields to:

$$\epsilon^* = \frac{(\nabla_x f(x^{(0)}))^T (\nabla_x f(x^{(0)}))}{(\nabla_x f(x^{(0)}))^T (H(f)(x^{(0)}))(\nabla_x f(x^{(0)}))}$$

In the worst case, when $\nabla_x f(x^{(0)})$ aligns with the eigenvector of H corresponding to the maximal eigenvalue λ_{max}, then this optimal step size is given by $\frac{1}{\lambda_{max}}$ To the extent that the function we minimize can be approximated well by a quadratic function, the eigenvalues of the Hessian thus determine the scale of the learning rate.

Choosing an appropriate step size is difficult because it must be small enough to ensure stability but large enough to allow progress; Newton's method addresses this by using second-order (Hessian) information to adapt the update and, for quadratic functions, reach the minimum in a single step. In deep learning, assuming functions (or their derivatives) are Lipschitz continuous provides theoretical guarantees by ensuring that small input changes lead to bounded output changes, improving optimization stability.

5.3.2 Stochastic Gradient Descent

Nearly all of deep learning is powered by one very important algorithm: **stochastic gradient descent** or **SGD** [RM51]. Stochastic gradient descent is an extension of gradient descent algorithm we just saw. It was first applied in the Neural Network in 1991 [Bot91]. The cost function used by a machine learning algorithm often decomposes as a sum over training examples of some per-example loss function. For example, the negative conditional log-likelihood of the training data can be written as Equation 5.3. For these additive cost functions, gradient descent requires to compute:

$$\nabla_\theta \mathcal{L}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$$

The computational cost of this operation is $O(m)$. As the training set size grows to billions of examples, the time to take a single gradient step becomes prohibitively long. The insight of stochastic gradient descent is that the gradient is an expectation. The expectation may be approximately estimated using a small set of samples. Specifically, on each step of the algorithm, we can sample a **minibatch** of examples $\mathbb{B} = \{x^{(1)}, \ldots, x^{(m')}\}$ drawn uniformly from the training set. The minibatch size m' is typically chosen to be a relatively small number of examples, ranging from 1 to a few hundred. Crucially, m' is usually held fixed as the training set size m grows. We may fit a training set with billions of examples using updates computed on only a hundred examples. The estimate of the gradient is formed as:

$$g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$$

using examples from the minibatch $\mathbb{B}$. The stochastic gradient descent algorithm then follows the estimated gradient downhill:

$$\theta \leftarrow \theta - \epsilon g$$

where ϵ is the learning rate. Gradient descent was historically considered slow and unreliable, especially for non-convex optimization, but it is now known to be highly effective in training modern machine learning models, often reaching very low cost values even without guarantees of optimality. Stochastic gradient descent (SGD) plays a central role beyond deep learning, particularly for training large models on massive datasets, as its per-update cost is independent of the training set size m. Although the number of updates needed for convergence grows with m, SGD typically reaches the model's optimal test performance before observing all training examples, implying an asymptotic training cost of $O(1)$ with respect to m. Moreover, SGD relies on unbiased gradient estimates computed from minibatches sampled i.i.d. from the data distribution, enabling efficient and scalable optimization (Algorithm 13).

Algorithm 13 `SGD_training_iteration_k`

Require: Learning rate ϵ_k, Initial parameter θ

1: **while** stopping criterion not met **do**
2: Sample a minibatch of m' examples from the training set $\{x^{(i)}\}_{i=1}^{m}$ with corresponding targets $y^{(i)}$
3: Compute the gradient estimate: $\hat{g} \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)}, \theta)y^{(i)})$
4: Apply the update $\theta \leftarrow \theta - \epsilon_k \hat{g}$
5: **end while**

A crucial hyperparameter of stochastic gradient descent is the learning rate, which must decrease over time and is therefore denoted by ϵ_k. This requirement arises because SGD relies on noisy gradient estimates whose variance does not vanish at a minimum, unlike batch gradient descent, which can use a fixed learning rate. Convergence of SGD is guaranteed if the learning rate satisfies the **Robbins-Monro conditions** [RM51]:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \quad \wedge \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty.$$

In practical implementations, the learning rate is often decayed linearly until a predefined iteration τ according to

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau, \qquad \alpha = \frac{k}{\tau}, \tag{5.4}$$

and then kept constant for the remainder of training. The choice of the learning rate schedule is typically guided by empirical evaluation through learning curves of the objective function.

Momemtum. An improvement regards the method of **momentum** [Pol64]. Formally, the momentum algorithm introduces a variable v that plays the role of velocity, it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient. In the momentum learning algorithm, we assume unit mass, so the velocity vector v may also be regarded as the momentum of the particle. A hyperparameter $\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay. The update rule is given by:

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(x^{(i)}, y^{(i)}, \theta) \right)$$
$$\theta \leftarrow \theta + v$$

The velocity v accumulates the gradient elements $\nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(x^{(i)}, y^{(i)}, \theta) \right)$. The larger α is relative to ϵ, the more previous gradients affect the current direction.

A variant of the momentum [SMDH13] was inspired by Nesterov's accelerated gradient method [Nes83, Nes14]. The update rules in this case are given by:

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}, \theta + \alpha v), y^{(i)}) \right)$$
$$\theta \leftarrow \theta + v$$

where the parameters α and ϵ play a similar role as in the standard momentum method. The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum the gradient is evaluated after the current velocity is applied. Thus one can interpret Nesterov momentum as attempting to add a correction factor to the standard method of momentum.

Parameter Initialization Strategies. Training algorithms for deep learning models are iterative and highly sensitive to the choice of parameter initialization. The initial point can affect convergence, numerical stability, convergence speed, the final cost, and the generalization performance of the model. Despite its importance, initialization remains poorly understood, and modern strategies are largely heuristic. Designing effective initializations is challenging because properties that are desirable at initialization may not be preserved during training, and initialization choices that favor optimization may negatively impact generalization.

Common heuristics focus on controlling the scale of the initial weights. A simple ap-

proach initializes weights of a layer with m inputs by sampling from

$$U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$$

while the normalized initialization proposed by [GY10] samples from $U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$. This latter method balances the variance of activations and gradients across layers under simplifying linear assumptions, and has been shown to perform well in practice despite the presence of nonlinearities.

Adaptive Learning Rates The learning rate is a critical hyperparameter in neural network training, strongly affecting performance and often difficult to set. Momentum can partially mitigate sensitivity to different directions in parameter space but introduces another hyperparameter. An alternative is to use separate learning rates for each parameter and adapt them automatically during training. Early methods such as the **delta-bar-delta** algorithm [Jac87] increase a parameter's learning rate when its gradient retains the same sign and decrease it when the gradient changes sign, though this applies primarily to full-batch optimization.

Modern incremental approaches include **AdaGrad** [DHS11], which scales each parameter's learning rate inversely proportional to the square root of the sum of its historical squared gradients. This allows greater progress along gently sloped directions and slower updates along steep directions. While AdaGrad has desirable theoretical properties in convex settings, in deep learning it can excessively reduce effective learning rates early in training, making it suitable for some models but not universally optimal.

RMSProp. The **RMSProp** algorithm [TH12] modifies AdaGrad to better handle non-convex optimization by replacing the accumulated squared gradients with an exponentially weighted moving average. While AdaGrad can reduce the learning rate too quickly before reaching a locally convex region, RMSProp discards long-past gradient history, allowing rapid convergence once a convex-like structure is encountered. This moving average introduces a new hyperparameter, ρ, controlling its time scale. Empirically, RMSProp is effective and widely used for training deep neural networks, often in combination with Nesterov's momentum.

ADAM. [KB14] is yet another adaptive learning rate optimization algorithm. The name ADAM derives from the phrase adaptive moments. In the context of the earlier algorithms, it is perhaps best seen as a variant on the combination of RMSProp and momentum with a few important distinctions. First, in ADAM, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. The use of momentum in combination

with rescaling does not have a clear theoretical motivation. Second, ADAM includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin. RMSProp also incorporates an estimate of the (uncentered) second-order moment, however it lacks the correction factor. Thus, unlike in ADAM, the RMSProp second-order moment estimate may have high bias early in training. ADAM is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.

AMSGRAD. [RKK19] introduce in 2019 a modification in the ADAM algorithm called AMSGRAD, see Algorithm 14. AMSGRAD uses a smaller learning rate in comparison to ADAM and yet incorporates the intuition of slowly decaying the effect of past gradients on the learning rate as long as the change in the inverse of learning rate of the adaptive method with respect to the time is positive semidefinite. The key difference of AMSGRAD with ADAM is that it maintains the maximum of all the second moment until the present time step and uses this maximum value for normalizing the running average of the gradient instead of the estimation of the second moment in ADAM. By doing this, AMSGRAD results in a non-increasing step size and avoids the pitfalls of ADAM and RMSPROP. To gain more intuition for the updates of AMSGRAD, it is instructive to compare its update with ADAM and ADAGRAD. Suppose at particular time step t and coordinate i, we have the second moment at the previous time greater than the square of the gradient at this time in this coordinate, then ADAM aggressively increases the learning rate, however, as we have seen in the previous section, this can be detrimental to the overall performance of the algorithm. On the other hand, ADAGRAD slightly decreases the learning rate, which often leads to poor performance in practice since such an accumulation of gradients over a large time period can significantly decrease the learning rate. In contrast, AMSGRAD neither increases nor decreases the learning rate and furthermore, decreases the second moment which can potentially lead to non-decreasing learning rate even if gradient is large in the future iterations.

5.3.3 Batch Normalization

Batch normalization [IS15] is a method for stabilizing and accelerating the training of deep neural networks by adaptively reparametrizing the network, rather than being an optimization algorithm itself. Deep networks require coordinated updates across many layers, which can cause instability when gradient updates assume other layers remain constant. Batch normalization standardizes the activations of each layer by replacing a minibatch of activations H with

$$H' = \frac{H - \mu}{\sigma + \delta},$$

Algorithm 14 `SGD_Amsgrad`. The operation $\diamondsuit$ indicate the multiplication elementwise.

Require: Step size ϵ, Initial parameter θ, Exponential decay rates for moment estimates $\rho_1, \rho_2 \in [0, 1)$ (suggested defaults: 0.9 and 0.999 respectively), small constant δ (for numerical stabilization, suggested default: 10^{-8}

Initialize first and second moment variables $s = 0, r = 0$
Initialize time step $t = 0$
while stopping criterion not met **do**
    Sample a minibatch of m' examples from the training set $\{x^{(i)}\}_{i=1}^{m}$ with corresponding targets $y^{(i)}$
    Compute the gradient estimate: $\hat{g} \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)}, \theta)y^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1)\hat{g}$
    Update biased second moment estimate: $r \leftarrow \max(\rho_2 r + (1 - \rho_2)\hat{g}\diamondsuit\hat{g}, r)$
    Correct bias in first moment: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$
    Compute update: $\Delta\theta = -\epsilon\frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$
    Apply update: $\theta \leftarrow \theta + \Delta\theta$
end while

where μ and σ are the mean and standard deviation of each unit in the minibatch, with $\delta > 0$ added to σ for numerical stability. Furthermore, suppone the size of the minibatch is N, then we can estimate the empirical mean $\hat{\mu}$ and the empirical variance $\hat{\sigma}$:

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i \quad \hat{\sigma} = \sqrt{\frac{1}{N}\sum_i (x_i - \hat{\mu})^2}$$

So, the new sample is:

$$x' = \frac{x - \hat{\mu}}{\hat{\sigma} + \delta}, \quad \forall x \in H$$

Gradients are back-propagated through this normalization, preventing updates that simply increase the mean or variance of activations. At test time, running averages of μ and σ are used to allow single-example evaluation.

To preserve expressive power, batch normalization introduces learnable parameters γ and β, transforming H' to $\gamma H' + \beta$, which allows the network to represent the same functions while improving learning dynamics. This reparametrization decouples the mean and variance of a layer from the parameters of preceding layers, simplifying gradient-based optimization. Batch normalization can be applied to inputs or hidden layers

and generally standardizes only the mean and variance, leaving nonlinear relationships between units intact. Inspired by [DSPK15], this approach has become a practical and widely used technique in training deep networks.

In convolutional networks, described in section 5.6, it is important to apply the same normalizing μ and σ at every spatial location within a feature map, so that the statistics of the feature map remain the same regardless of spatial location.

5.3.4 Continuation Methods and Curriculum Learning

Optimization can also be facilitated through **continuation methods**, which aim to replace a difficult objective $\mathcal{L}(\theta)$ function with a sequence of progressively harder ones $\{\mathcal{L}^{(0)}, \ldots, \mathcal{L}^{(n)}\}$. These cost functions are designed to be increasingly difficult, with $\mathcal{L}^{(0)}$ being fairly easy to minimize, and $\mathcal{L}^{(n)}$, the most difficult, being $\mathcal{L}(\theta)$, the true cost function motivating the entire process. When we say that $\mathcal{L}^{(i)}$ is **easier** than $\mathcal{L}^{(i+1)}$, we mean that it is well behaved over more of θ space. Classical continuation methods, which predate their application to neural networks, typically operate by smoothing or "blurring" the objective function,

$$\mathcal{L}^{(i)}(\theta) = \mathbb{E}_{\theta' \sim \mathcal{N}(\theta', \theta\sigma^{(i)})} \mathcal{L}(\theta')$$

thereby reducing non-convexity and improving tractability [MW06]. These approaches are closely related to simulated annealing, which injects noise into the parameter updates to encourage exploration of the optimization landscape [KGV83]. More recently, continuation methods have gained renewed attention in machine learning and artificial intelligence, with comprehensive reviews highlighting their effectiveness across a variety of applications [MI15]. Traditionally, continuation methods were motivated by the desire to escape poor local minima and recover global optima in highly non-convex problems. By convolving the original objective with a smoothing kernel, the resulting blurred objective can become approximately convex, allowing optimization to proceed more reliably. However, this strategy is not universally successful: some functions remain non-convex regardless of the degree of smoothing, while in other cases the minimum of the smoothed objective may track toward a suboptimal local minimum of the original function. Moreover, even when continuation methods are theoretically applicable, the number of intermediate objectives required can make the approach computationally expensive. Although local minima are no longer considered the primary obstacle in neural network optimization, continuation methods remain valuable because they improve other aspects of the optimization landscape. Smoother objectives can reduce flat regions, decrease the variance of gradient estimates, and improve the conditioning of the Hessian matrix, thereby enhancing the effectiveness of local descent methods.

Curriculum learning can be interpreted as a practical and data-driven instance of continuation methods [YLC+09]. Instead of modifying the objective function directly, curriculum learning structures the training process so that simpler examples are presented first, with progressively more complex examples introduced over time. This strategy has been shown to accelerate learning and improve generalization across a wide range of tasks in natural language processing and computer vision. Empirical evidence suggests that stochastic curricula, which maintain a mixture of easy and difficult examples while gradually increasing the proportion of harder ones, are particularly effective for training recurrent neural networks to capture long-term dependencies [ZS14]. These findings are consistent with observations from human and animal learning, where instruction typically proceeds from simple, prototypical cases to more challenging examples. Overall, many challenges in neural network optimization stem from the global structure of the loss function rather than from inaccuracies in local gradient estimates. Consequently, a dominant strategy for improving optimization is to initialize parameters in regions of the parameter space that are connected to good solutions by short, well-behaved paths. Architectural design choices and continuation-based training strategies both serve this goal by reshaping the optimization landscape so that local descent methods can more effectively discover high-quality solutions.

Other possible optimization strategies Many optimization techniques are not exactly algorithms, but rather general templates that can be specialized to yield algorithms, or subroutines that can be incorporated into many different algorithms.

Coordinate Descent. More generally, block coordinate descent refers to minimizing with respect to a subset of the variables simultaneously. The term coordinate descent is often used to refer to block coordinate descent as well as the strictly individual coordinate descent. Coordinate descent makes the most sense when the different variables in the optimization problem can be clearly separated into groups that play relatively isolated roles, or when optimization with respect to one group of variables is significantly more efficient than optimization with respect to all of the variables.

Polyak Averaging. Polyak Averaging [PJ92] consists of averaging together several points in the trajectory through parameter space visited by an optimization algorithm. If t iterations of gradient descent visit points $\theta^{(1)}, \ldots, \theta^{(t)}$, then the output of the Polyak averaging algorithm is $\hat{\theta}^{(t)} = \frac{1}{t} \sum_i \theta^{(i)}$. On some problem classes, this approach has strong convergence guarantees. When applied to neural networks, its justification is more heuristic, but it performs well in practice. The basic idea is that the optimization algorithm may leap back and forth across a valley several times without ever visiting a point near the bottom of the valley. The average of all of the locations on either side should be close to the bottom of the valley though.

Meta Algorithm. When directly training a complex model on a difficult task is challenging, it can be effective to first train simpler models or to train on simpler tasks before moving to the final task. These strategies are collectively called **pretraining**. **Greedy** algorithms solve each component of a problem in isolation; while the combination of individually optimal components is not guaranteed to be globally optimal, greedy solutions are computationally efficient and often acceptable. Greedy solutions can also serve as initialization for joint optimization, improving convergence and solution quality. In deep learning, **greedy supervised pretraining** [BLP+07] breaks a supervised learning problem into simpler supervised tasks, training subsets of layers sequentially to provide better guidance for intermediate layers. Pretraining can improve both optimization and generalization, and has been applied successfully in multiple contexts [SZ14, YWD11, YCL+14, RYK+14].

Architectural Design for Easier Optimization. Many advances in deep learning optimization have come from designing model architectures that are inherently easier to optimize, rather than from developing new optimization algorithms. Modern networks typically combine linear transformations with differentiable activation functions that maintain significant gradients, facilitating gradient flow across layers. Architectures using ReLUs, maxout units, or LSTM cells rely more on linear components than earlier sigmoid-based models, improving alignment between local gradients and global progress. Additional strategies address the vanishing gradient problem. Skip connections [SGS15] or auxiliary output heads [SLJ+14, LXG+14] shorten the effective distance between parameters and the output, providing stronger gradient signals to early layers. Auxiliary classifiers can be removed after training, enabling joint optimization of all layers while ensuring informative gradients throughout the network. These architectural innovations have been crucial to the practical trainability of deep models.

5.4 Deep Feedforward Networks

Deep feedforward networks, also often called **feedforward neural networks**, or **multilayer perceptrons** (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f^*. A feedforward network defines a mapping $y = f(x, \theta)$ and learns the value of the parameters θ that result in the best function approximation. These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. Feedforward networks are of extreme importance to machine learning practitioners. They form the basis of many important commercial applications.

Feedforward neural networks are called **networks** because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together.

For example, we might have three functions $\{f^{(i)}\}_{i=1}^{3}$ connected in a chain, to form

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer, and so on. The overall length of the chain gives the **depth** of the model. It is from this terminology that the name **deep learning** arises. The final layer of a feedforward network is called the **output layer**.

During neural network training, we drive $f(x)$ to match $f^*(x)$. The training data provides us with noisy, approximate examples of $f^*(x)$ evaluated at different training points. Each example x is accompanied by a label $y \approx f^*(x)$. The training examples specify directly what the output layer must do at each point x; it must produce a value that is close to y. The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data does not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of $f^*(x)$. Because the training data does not show the desired output for each of these layers, these layers are called **hidden layers**.

Each hidden layer of the network is typically vector-valued. The dimensionality of these hidden layers determines the **width** of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of the layer as consisting of many **units** that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value.

The idea of using many layers of vector-valued representation is drawn from neuroscience. The choice of the functions $f^{(i)}(x)$ used to compute these representations is also loosely guided by neuroscientific observations about the functions that biological neurons compute. However, modern neural network research is guided by many mathematical and engineering disciplines, and the goal of neural networks is not to perfectly model the brain. It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function. Feedforward networks are the application of the principle to learning deterministic mappings from x to y that lack feedback connections.

Historical notes. Feedforward neural networks are efficient nonlinear function approximators optimized via gradient-based methods, reflecting centuries of progress in function approximation and calculus. The chain rule, central to back-propagation, dates

to the 17th century [Lei76, L'H96], while gradient descent emerged in the 19th century [L.47]. Early perceptrons of the 1940s were limited to linear mappings, whose inability to learn XOR led to skepticism in the 1970s–1980s. Advances in multilayer networks and practical gradient computation—using dynamic-programming forms of the chain rule for control and sensitivity analysis [Kel60, BD62, Dre62, Dre73, BHS79, Lin76]—enabled back-propagation training [Wer70, Lec85, Par85, RHW86b, RHW86a, HMR86].

Although the feedforward architecture has remained largely unchanged since the 1980s, performance improved through larger datasets, better hardware, and methodological innovations. Notably, cross-entropy replaced mean squared error for sigmoid and softmax outputs, and piecewise linear activations replaced sigmoids, following early work by Fukushima [Fuk75, Fuk80] and later empirical validation [JKRL09, GBY10]. Between 2006 and 2012, pretraining was considered necessary, but since 2012 feedforward networks reliably perform across tasks. The use of supervised learning to aid unsupervised objectives has also become common. Modern feedforward networks have evolved from a mistrusted concept into a central tool for probabilistic modeling [GMD+13], yet their potential continues to motivate improvements in architecture and optimization.

Choice of Cost Function The choice of the cost function is a crucial aspect of deep neural network design and largely follows the same principles used for other parametric models. Most modern neural networks are trained using **maximum likelihood**, leading to the use of the **negative log-likelihood**, or equivalently the **cross-entropy**, as the cost function:

$$\mathcal{L}(\theta) = -\mathbb{E}_{x,y\sim\hat{p}_{data}} \log p_{model}(y|x).$$

The specific form of the loss depends on the assumed output distribution; for example, a Gaussian output leads to the **mean squared error** objective, independently of the form of the predictor $f(x;\theta)$. Deriving cost functions from probabilistic models simplifies design and often yields gradients that are more stable and informative, avoiding issues caused by saturating nonlinearities.

Cross-entropy has the distinctive property of often lacking a finite minimum, as models can assign arbitrarily high probability or density to training targets. In many settings, however, the goal is not to learn the full conditional distribution but rather a specific statistic of y given x. From this perspective, learning can be viewed as selecting a function rather than just parameters, and the loss becomes a **functional**. Using tools from the **calculus of variations**, minimizing mean squared error yields a predictor of the conditional mean, while minimizing **mean absolute error** yields the conditional median. Despite this interpretability, these losses often perform poorly with gradient-based optimization, which explains why cross-entropy remains the preferred choice even when full density estimation is unnecessary.

5.4.1 Output Units

The choice of output units is tightly coupled with the choice of cost function, most commonly the **cross-entropy** derived from **maximum likelihood**. Given hidden features $h = f(x; \theta)$, the output layer transforms these features to define a conditional distribution over the target variable.

Linear output units implement an affine transformation $\hat{y} = W^T h + b$ and are typically used to predict the mean of a Gaussian distribution. In this case, maximum likelihood reduces to minimizing the **mean squared error**. Linear units do not saturate and thus are well suited to gradient-based optimization, though they are usually insufficient to parametrize constrained quantities such as covariance matrices.

For binary outputs, the appropriate probabilistic model is the **Bernoulli** distribution. Instead of thresholded linear units, which lead to vanishing gradients, **sigmoid** output units are used:

$$\hat{y} = \sigma(w^T h + b), \quad \text{where } \sigma(x) = \frac{1}{1 + \exp(-x)} \tag{5.5}$$

σ is called the **logistic sigmoid function**. Combined with maximum likelihood, this yields a loss expressed via the **softplus** function, which avoids gradient saturation when predictions are incorrect and ensures effective learning.

For discrete variables with more than two outcomes, the **softmax** function generalizes the sigmoid to the **Multinoulli** distribution. A linear layer produces unnormalized log-probabilities, or **logits**, which are exponentiated and normalized by softmax. The corresponding log-likelihood has favorable optimization properties because the logarithm undoes the exponential, preventing vanishing gradients even for highly confident but incorrect predictions. Squared error and other losses that do not invert the softmax nonlinearity perform poorly in this setting [Bri90].

Softmax activations can saturate when differences between logits are large, making numerical stability a critical concern. Stable implementations exploit the invariance:

$$softmax(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{5.6}$$

to avoid **underflow** and **overflow**. While softmax is formally overparametrized, this rarely poses practical difficulties, and the simpler formulation is usually preferred.

Beyond standard linear, sigmoid, and softmax outputs, neural networks can represent a wide range of output distributions. The **maximum likelihood** framework provides a unified principle: the network outputs parameters of a conditional distribution $p(y|x; \theta)$, and the loss is the negative log-likelihood. This approach naturally extends to modeling uncertainty, such as learning input-dependent (**heteroscedastic**) variance in Gaussian

regression using constrained outputs (e.g., via the **softplus**).

For multimodal targets, **mixture density networks** [JJNH91, Bis94] model $p(y|x)$ as a mixture of Gaussians, with mixture weights, means, and covariances predicted by the network. Although powerful, these models can be numerically unstable and often require heuristics such as gradient clipping [UML14]. Despite these challenges, they have been successfully applied to tasks such as speech generation [Sch99] and motion modeling [Gra13]. More complex output structures can still be addressed within the maximum likelihood framework, typically requiring more advanced architectures such as recurrent neural networks.

5.4.2 Hidden Units

After addressing design choices common to gradient-based parametric models, a key issue specific to feedforward neural networks is the choice of **hidden units**. The design of hidden unit activation functions remains an active research area with few definitive theoretical guidelines. In practice, **rectified linear units** (**ReLU**) are an effective default choice, although many alternatives exist and their relative performance is difficult to predict a priori. As a result, selecting hidden units typically relies on empirical trial and error validated on held-out data.

Despite the use of gradient-based optimization, several popular activation functions are not differentiable everywhere. For instance, ReLU is non-differentiable at zero. In practice, this does not hinder learning because neural network training rarely converges to exact minima with zero gradients. Most activation functions have well-defined left and right derivatives, and implementations usually select one-sided derivatives. Numerical noise in digital computation further reduces the practical relevance of theoretical non-differentiability. Consequently, non-differentiability can usually be safely ignored.

Most hidden units compute an affine transformation followed by an element-wise nonlinearity:

$$z = W^T x + b, \quad h = g(z),$$

and differ primarily in the choice of the activation function g.

Rectified Linear Units (**ReLU**) use

$$g(z) = \max\{0, z\} \tag{5.7}$$

They are easy to optimize because they behave similarly to linear units while avoiding saturation for active inputs. Their gradients are large, consistent, and free of second-order effects. A common practice is to initialize biases to small positive values to ensure early activation. A limitation of ReLU is the absence of gradient when the unit

is inactive. Several generalizations address this issue by introducing a nonzero slope for negative inputs:

$$h_i = \max(0, z_i) + \alpha_i \min(0, z_i).$$

These include **absolute value rectification** ($\alpha = -1$), **leaky ReLU** [Maa13], and **parametric ReLU (PReLU)** [HZRS15b].

Maxout units [GWFM+13] further generalize ReLU by outputting the maximum over a group of linear responses. Maxout units effectively *learn the activation function itself* and can approximate any convex function with sufficient capacity. They subsume ReLU and its variants but require more parameters and often stronger regularization [CSL13]. Their redundancy can improve robustness and reduce catastrophic forgetting [GMD+13]. More generally, ReLU-like units are motivated by the principle that models closer to linearity are easier to optimize, a principle that also underlies architectures such as **LSTM** networks.

Before ReLU, most networks used **sigmoid** or **tanh** activations. These functions saturate over much of their domain, leading to vanishing gradients and difficult optimization. As a result, their use as hidden units in feedforward networks is now discouraged. When sigmoidal units are required, **tanh** typically performs better due to its closer resemblance to the identity function near zero. Sigmoidal units remain useful in recurrent networks, probabilistic models, and autoencoders, where piecewise linear units may be unsuitable.

A wide variety of other hidden units have been explored. In practice, many differentiable activation functions perform similarly, and new ones are typically adopted only when they demonstrate clear advantages. **Linear hidden units** can be useful within nonlinear networks to impose low-rank structure and reduce parameters. **Softmax** units may serve as hidden units in specialized architectures, acting as soft switches. Other examples include **radial basis function (RBF)** units, which are difficult to optimize due to saturation, and the **softplus** function [DBB+00], a smooth ReLU approximation that is generally outperformed by ReLU in practice [GBY10]. The **hard tanh** activation [Col04] provides a bounded piecewise linear alternative.

Overall, while a small set of activation functions dominates current practice, the design of hidden units remains an open and empirically driven area of research, and many effective alternatives may still be undiscovered.

5.4.3 Architecture Design

A fundamental design choice in neural networks concerns the **architecture**, namely the number of units and the pattern of connections among them. Most architectures are organized into **layers** arranged in a chain, where each layer computes a nonlinear transformation of the previous one. In such feedforward architectures, the primary

structural choices are the **depth** of the network and the **width** of each layer. While even a single hidden layer is sufficient to fit the training data, deeper networks often require fewer parameters and generalize better, at the cost of increased optimization difficulty. In practice, selecting an appropriate architecture relies on empirical experimentation guided by validation performance.

Linear models are easy to optimize due to the convexity of many associated loss functions, but they can represent only linear mappings. Feedforward neural networks overcome this limitation through hidden layers, providing a universal framework for approximating nonlinear functions. The **universal approximation theorem** [HSW89, Cyb89] states that a network with a linear output layer and at least one hidden layer using a **squashing** activation function can approximate any Borel measurable function to arbitrary accuracy, given sufficiently many hidden units. This result extends to broader classes of activation functions, including rectified linear units [LLPS93], and even to approximating derivatives of functions [HSW90]. However, universality guarantees representational capacity only, not learnability or generalization.

Although a shallow network can represent any function, it may require an infeasibly large number of hidden units and may fail to generalize. In the worst case, the number of required units grows exponentially with input dimensionality [Bar93]. Deeper architectures can represent certain families of functions far more efficiently, reducing both model size and generalization error. Results from circuit complexity and neural network theory show that functions efficiently representable by deep models may require exponentially many units in shallow ones [Hås86, HG91, HMP$^+$93, MSS94, Maa06]. In particular, deep rectifier networks can represent functions with a number of linear regions that grows exponentially with depth [MPCB14], highlighting the expressive advantage of depth.

Beyond representational efficiency, choosing a deep architecture also encodes prior assumptions about the structure of the target function. Deep models reflect the belief that the function of interest can be decomposed into a composition of simpler functions or interpreted as a multi-step computation. From a representation learning perspective, this corresponds to discovering hierarchical factors of variation or intermediate computational states.

While neural networks are often described as simple chains of layers, practical architectures are far more diverse. Modern designs frequently include non-sequential connections, such as **skip connections**, which facilitate gradient flow and improve optimization in deep networks. Another key architectural decision concerns connectivity patterns: fully connected layers connect all units across layers, whereas **sparse connectivity** restricts interactions to subsets of units, reducing parameters and computational cost while exploiting problem structure. Overall, architecture design extends well beyond depth and width, encompassing connectivity and information flow, and remains largely driven by empirical insights tailored to specific tasks and data.

5.4.4 Back-Propagation.

When a feedforward neural network maps an input x to an output $\hat{y}$, information flows forward through the layers, producing a scalar cost $\mathcal{L}(\theta)$. While computing analytical gradients is straightforward, evaluating them naively is computationally expensive. The **back-propagation algorithm** [RHW86b], often called **backprop**, efficiently computes these gradients by propagating information backward from the cost.

Back-propagation refers only to gradient computation, not to the learning algorithm itself, which is typically stochastic gradient descent. Moreover, it is not limited to neural networks: it can compute derivatives of any differentiable function. In learning, the most common quantity of interest is $\nabla_\theta \mathcal{L}(\theta)$, but back-propagation can compute more general derivatives such as Jacobians (see Definition 5.12). We focus on the common case where the function has a single scalar output. We formalize back-propagation using **computational graphs**, which represent variables as nodes and operations as directed edges.

Definition 5.16 (Computational Graph)**.** Let $G = (V, E)$ be an directed graph with vertices set V and edges set E. G is a **computational graph** if V contains the variables e.g. scalar, vector, matrix, tensor or even variable of another type and E contains the operations i.e. a function of one or more variable. If a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y. We sometimes annotate the output node with the name of the operation applied, and other times omit this label when the operation is clear from context.

Each operation is assumed to produce a single output, without loss of generality. Back-propagation relies on the chain rule, which is efficiently applied by traversing the computational graph backward.

Chain Rule 5.17. *Let x be a real number, and let f and g both function from $\mathbb{R}$ to $\mathbb{R}$. Suppose $y = g(x)$ and $z = f(g(x)) = f(y)$. Then, the **chain rule** states that:*

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Generalizing for $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$. If $x \in \mathbb{R}^m$, $y = g(x) \in \mathbb{R}^n$ and $z = f(y) \in \mathbb{R}$, then:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

In vector notation:

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z$$

where $\frac{\partial y}{\partial x} \in \mathbb{R}^{n\times m}$ is the Jacobian matrix of g.

Back-propagation repeatedly applies Jacobian–gradient products along graph edges.

Definition 5.18 (Tensor)**.** A **tensor** is an array with more than two axis. For example $A \in \mathbb{R}^{n \times m \times p}$. The notation for the tensor is $\mathbf{A}$.

The algorithm generalizes naturally to tensors. Gradients with respect to tensors are treated analogously to vector gradients by indexing all tensor entries.

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial \mathbf{Y}_j} \tag{5.8}$$

A naive application of the chain rule would involve many repeated subexpressions, potentially leading to exponential computational cost. Back-propagation avoids this by storing intermediate gradients and reusing them.

Back-propagation constructs a reverse graph in which each node computes the derivative of the final scalar output with respect to the corresponding forward node:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i \text{ s.t. } u^{(j)} \text{ is an input of } f^{(i)}} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}} \tag{5.9}$$

The computational cost of back-propagation scales linearly with the number of edges in typical neural network graphs, making it far more efficient than naive differentiation.

Algorithm 15 `basic_forward_propagation_NN`.

Require: The input x, target output y, Network Depth l, weight matrices of the model $W^{(i)}, i \in \{1, \ldots, l\}$, biases parameters of the model $b^{(i)}, i \in \{1, \ldots, l\}$

1: $h^{(0)} \leftarrow x$
2: **for** $k = 1, \ldots, l$ **do**
3: $\quad a^{(k)} \leftarrow b^{(k)} + W^{(k)} h^{(k-1)}$
4: $\quad h^{(k)} \leftarrow f(a^{(k)})$
5: **end for**
6: $\hat{y} \leftarrow h^{(l)}$
7: **return** $L(\hat{y}, y)$

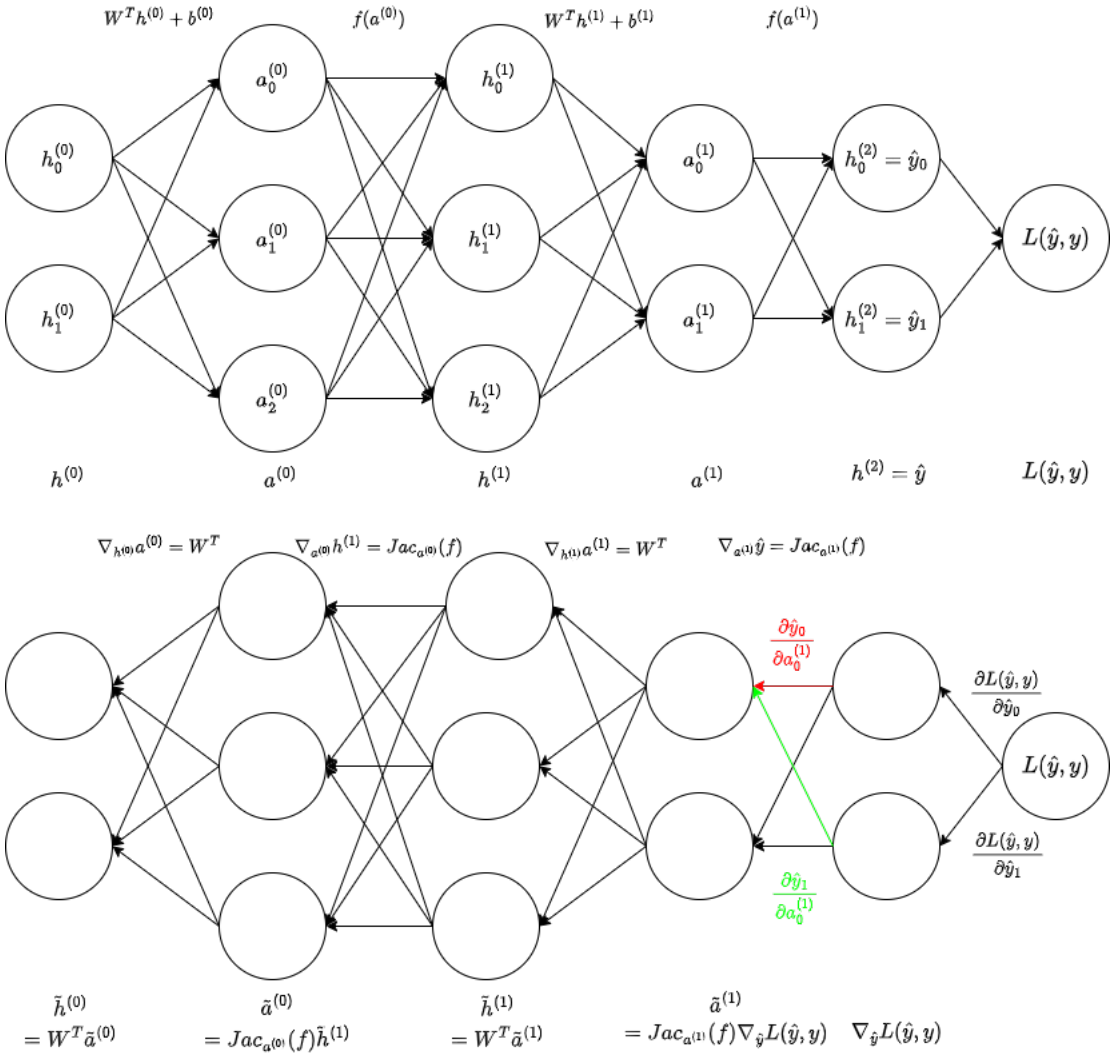

FIGURE 5.1 Example MLP Neural Network. In the top image is presented the forward propagation, while in the bottom image the backward propagation.

Algorithm 16 `basic_backward_propagation_NN`.

Require: The input x, target output y, Network Depth l, weight matrices of the model $W^{(i)}, i \in \{1, \dots, l\}$, biases parameters of the model $b^{(i)}, i \in \{1, \dots, l\}$

$J \leftarrow$ `basic_forward_propagation_NN(x, y, l, b, W)`
$g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{y}, y)$
for $k = l, l-1, \dots, 1$ **do**
    Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation it could be element-wise multiplication if f is element-wise (for the sack of simplicity we write the element-wise operation but, as we see in Figure 5.1, it could be a matrix vector product):
    $g \leftarrow \nabla_{a^{(k)}} J = g \odot f'(a^{(k)})$
    Compute gradient on weights and biases (it is possible to use those computation in algorithm 14 or in any other update of the weights):
    $\nabla_{b^{(k)}} J \leftarrow g$
    $\nabla_{W^{(k)}} J \leftarrow g\ (h^{(k-1)})^T$
    Propagate the gradients w.r.t. the next lower-level hidden layer's activations
end for

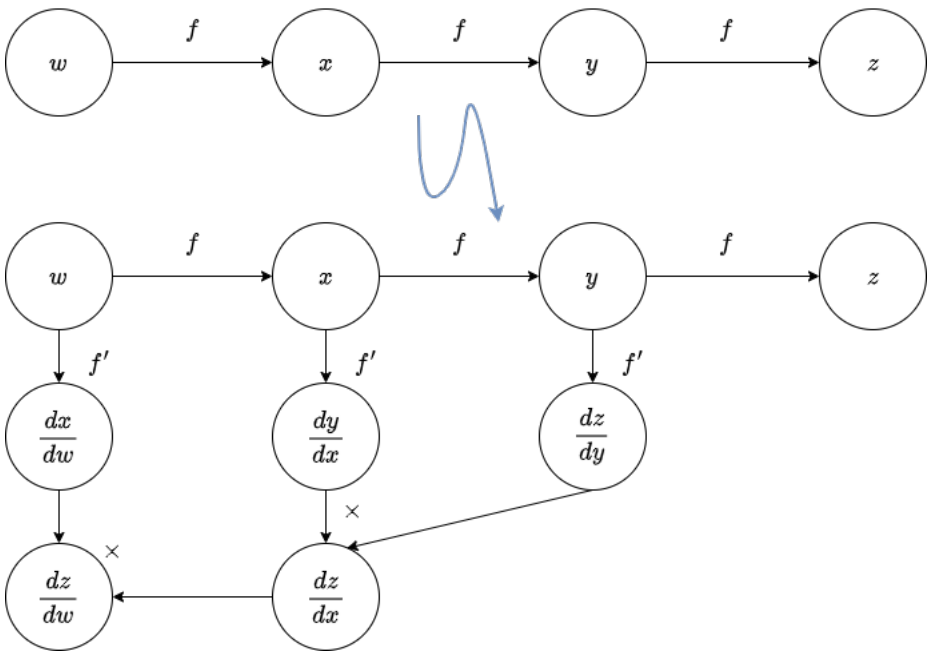

FIGURE 5.2 Example symbol-to-symbol approach.

Back-propagation can be implemented either as **symbol-to-number** differentiation or as **symbol-to-symbol** differentiation. The latter constructs a new computational

graph for the gradients, enabling higher-order differentiation and flexible execution [BBB+10, BLP+12, AAB+16]. Each operation provides a local `bprop` rule implementing a Jacobian–vector product, allowing the back-propagation algorithm to remain generic. Back-propagation was introduced to avoid the exponential cost that arises from repeatedly evaluating identical subexpressions in the naive application of the chain rule. While a direct expansion of the chain rule can lead to an exponential number of terms due to the many possible paths in a deep computational graph,

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\text{path } (u^{(\pi_1)}, \ldots, u^{(\pi_t)}), \text{ from } \pi_1 = j \to \pi_t = n} \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}$$

back-propagation avoids this explosion by reusing intermediate results. Assuming each node of the computational graph corresponds to a single operation, computing gradients in a graph with n nodes requires at most $O(n^2)$ operations and stored values, since the graph contains at most $O(n^2)$ edges. In practice, most neural network graphs are approximately chain-structured, resulting in a much more favorable $O(n)$ complexity. Back-propagation achieves this efficiency by storing partial derivatives at each node and reusing them whenever needed, rather than recomputing them along multiple paths. This table-filling strategy, in which gradients are computed once and propagated backward through the graph, can be viewed as an instance of **dynamic programming**.

5.5 Regularization

Machine learning algorithms must generalize well to unseen data. A common challenge is overfitting, where a model learns patterns specific to the training set rather than the true underlying structure. **Regularization** encompasses strategies that improve generalization and reduce test error, often by constraining parameter values or adding penalties to the objective function. These constraints can encode prior knowledge, favor simpler models, or make underdetermined problems solvable. Statistically, regularization trades increased bias for reduced variance, achieving a balance that improves test performance. In deep learning, large and expressive networks are commonly used; proper regularization prevents overfitting and enables reliable predictions in real-world applications.

5.5.1 Parameter Norm Penalties

Regularization often limits model capacity by adding a parameter norm penalty $\Omega(\theta)$ to the objective function $\mathcal{L}$, yielding

$$\tilde{\mathcal{L}}(\theta, X, y) = \mathcal{L}(\theta, X, y) + \alpha\Omega(\theta),$$

where $\alpha \geq 0$ controls the penalty strength. In neural networks, the penalty is usually applied only to the weights w, leaving biases unregularized.

The most common penalty is the L_2 norm, or **weight decay**, with $\Omega(\theta) = \frac{1}{2}||w||_2^2$. The gradient update becomes

$$w \leftarrow (1 - \epsilon\alpha)w - \epsilon\nabla_w \mathcal{L}(w, X, y),$$

shrinking weights multiplicatively at each step. Approximating $\mathcal{L}$ quadratically around its minimum w^*:

$$\hat{\mathcal{L}}(w) = \mathcal{L}(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*),$$

the regularized minimum is

$$\tilde{w} = (H + \alpha I)^{-1} H w^* = Q(\Lambda + \alpha I)^{-1}\Lambda Q^T w^*,$$

where $H = Q\Lambda Q^T$ is the Hessian decomposition. Weight decay shrinks components along directions with small eigenvalues, preserving directions that strongly reduce the objective.

For linear regression with $\mathcal{L}(w) = (Xw - y)^T(Xw - y)$, adding L_2 regularization gives

$$\tilde{\mathcal{L}}(w) = (Xw - y)^T(Xw - y) + \frac{1}{2}\alpha w^T w \implies w = (X^T X + \alpha I)^{-1} X^T y,$$

effectively adding α to the diagonal of $X^T X$ and shrinking weights on features weakly correlated with the target.

L_1 regularization, defined by

$$\Omega(\theta) = ||w||_1 = \sum_i |w_i|,$$

similarly scales with α:

$$\tilde{\mathcal{L}}(w, X, y) = \mathcal{L}(w, X, y) + \alpha||w||_1 \implies \nabla_w \tilde{\mathcal{L}}(w, X, y) = \nabla_w \mathcal{L}(w, X, y) + \alpha\text{sign}(w),$$

inducing sparsity and enabling **feature selection**, as in LASSO [Tib18].

5.5.2 Early Stopping

When training large models with sufficient capacity to overfit, we often observe that training error decreases steadily, while validation error eventually rises. To obtain a model with better generalization, we can restore the parameters corresponding to the lowest validation error. Each time the validation error improves, we store a copy of the model parameters; training terminates when no improvement occurs for a pre-specified number of iterations. This strategy, known as **early stopping**, is widely used in deep learning due to its effectiveness and simplicity. Conceptually, early stopping can also be viewed as an efficient hyperparameter selection method, where the number of training steps acts as a tunable hyperparameter.

5.6 Convolutional Neural Network

Convolutional neural networks (CNNs) have been pivotal in deep learning, bridging neuroscience insights with practical machine learning applications. Early successes include LeCun's CNN for automated check reading at AT&T in the 1990s [LBH+98], later adopted by NEC, and Microsoft's OCR and handwriting systems [SSP03]. For a historical overview, see [LKF10], while analytical foundations are discussed in [Lin76]. CNNs excel in grid-structured data, particularly 2D images, and their early success is attributed to computational efficiency: fewer parameters and operations allowed extensive experimentation and tuning, unlike fully connected networks which faced adoption barriers largely due to expectations rather than capability.

Convolutional networks [LeC89] specialize neural networks for grid-like data (e.g., time series or images) by replacing standard matrix multiplication with convolution. Convolution computes a weighted sum of inputs, emphasizing nearby elements via a kernel w:

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da$$

In practice, data are discrete, giving the **discrete convolution**:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{+\infty} x(a)w(t-a)$$

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the **input** and the second argument (in this example, the function w) as the **kernel** or the **filter**. The output is sometimes referred to as the **feature map**. The entries of the filters are called **channels** and the entries of the matrix (i, j) are called **width** (to not confuse with the width of a Neural

Network) and **height**. For 2D images I and kernels K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Convolution is commutative:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

but neural network libraries often implement **cross-correlation** (see Figure 5.3) (no kernel flipping):

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

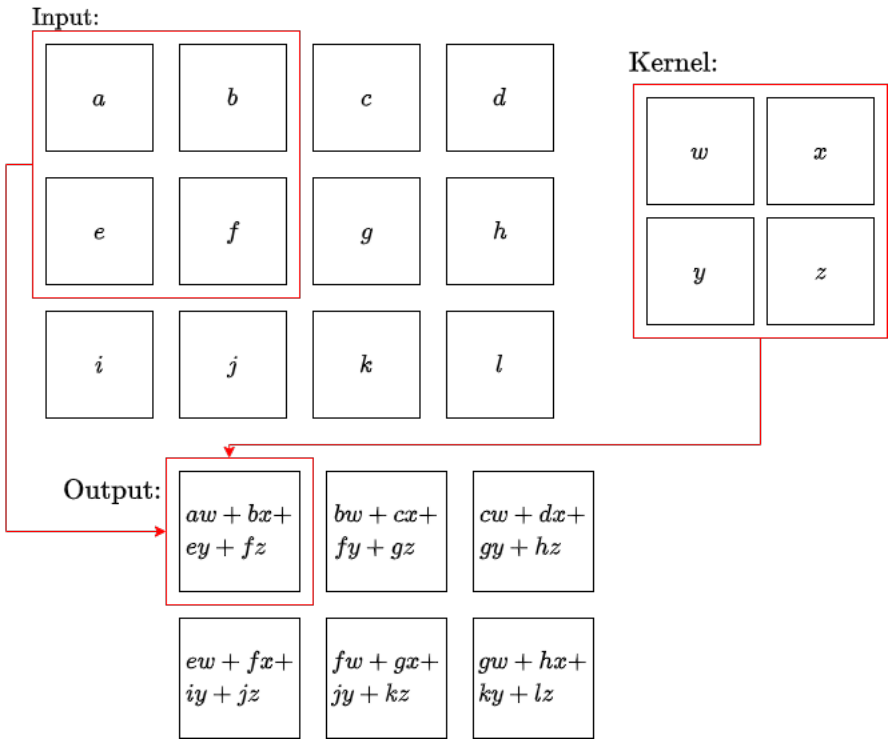

FIGURE 5.3 Example 2-D convolution graph.

It is possible also to use more filters (see Figure 5.4), to obtain:

$$S(i,j,t) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n,t)$$

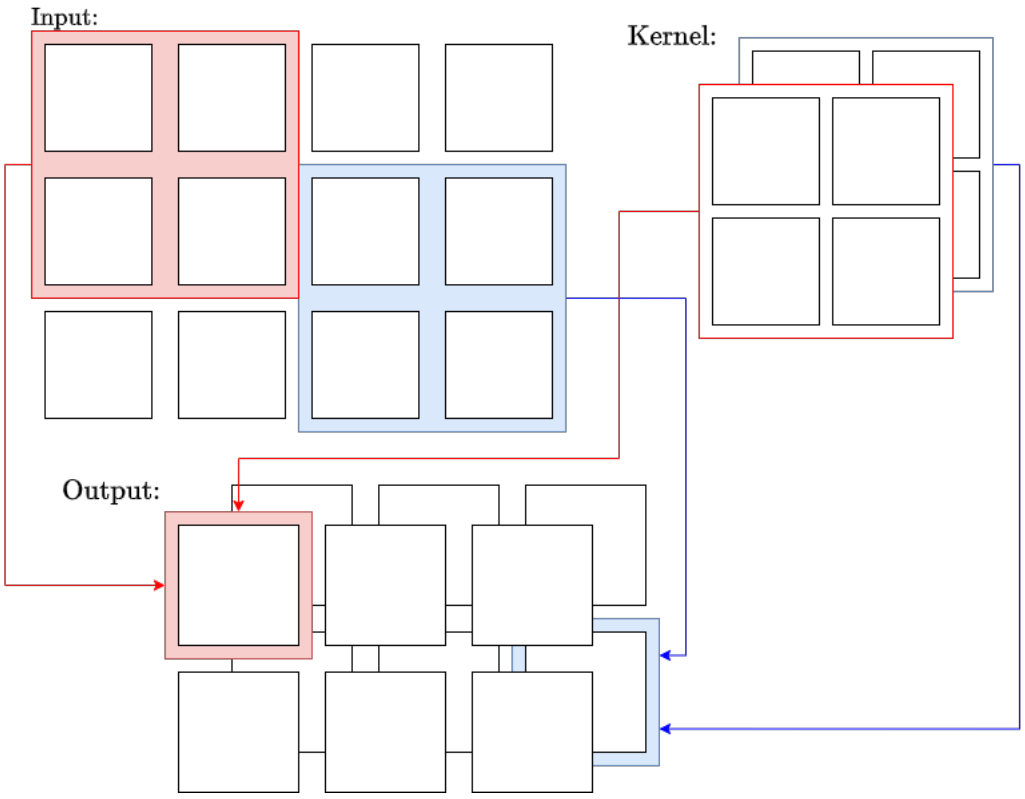

FIGURE 5.4 Example 2-D convolution graph with two filters.

Discrete convolution can be represented as matrix multiplication with structured sparsity. In 1D, the matrix is **Toeplitz**; in 2D, it is a **doubly block circulant** matrix. Each row (or block) is a shifted version of the previous, reflecting the shared weights of the kernel. Although CNNs often employ further optimizations for efficiency, theoretically any algorithm that works with matrix multiplication can handle convolution without modification.

Motivation Convolution leverages three key ideas that improve efficiency and performance in machine learning systems:

- **Sparse interactions**. Unlike traditional layers where every output unit interacts with every input unit via $m \times n$ parameters and $O(m \times n)$ runtime, convolution uses small kernels with $k \ll m$, leading to $k \times n$ parameters and $O(k \times n)$ operations. This reduces memory and computation while allowing complex interactions to be built from simple, local building blocks.
- **Parameter sharing**. Each kernel parameter is reused across input positions, or equivalently, the network has **tied weights**. Forward propagation runtime remains $O(k \times n)$, but the number of parameters drops to k, dramatically improving statistical efficiency and reducing storage.

- **Equivariant representations**. Convolutional layers are **equivariant to translation**: if the input shifts, the output shifts correspondingly. Formally, if g translates the input, then $f(g(x)) = g(f(x))$. This property ensures that features detected in one region of the input are detected in all other regions, which is particularly useful in images or time series. Note, however, that convolution is not naturally equivariant to scale or rotation; additional mechanisms are required for those transformations.

5.6.1 Dilated CNN

Dilated CNN. In [YK15] it was introduced the concept of **Dilated CNN**. The [YK15] refers to two-dimensional image data, however, a prominent example that uses dilated convolutions and deals with long, as well as short-range dependencies on one-dimensional temporal data is [VdoDZ+16]. A dilated convolution uses a dilation rate above one. Therefore, instead of learning a filter function between bits 1 and 2, a convolutional layer with dilation rate 3 can learn a filter function between bits 1 and 4 (see Figure 5.5). In general,

$$S_d(i,j,t) = (I * K)(i,j) = \sum_m \sum_n I(i + m \cdot d, j + n \cdot d) K(m,n,t)$$

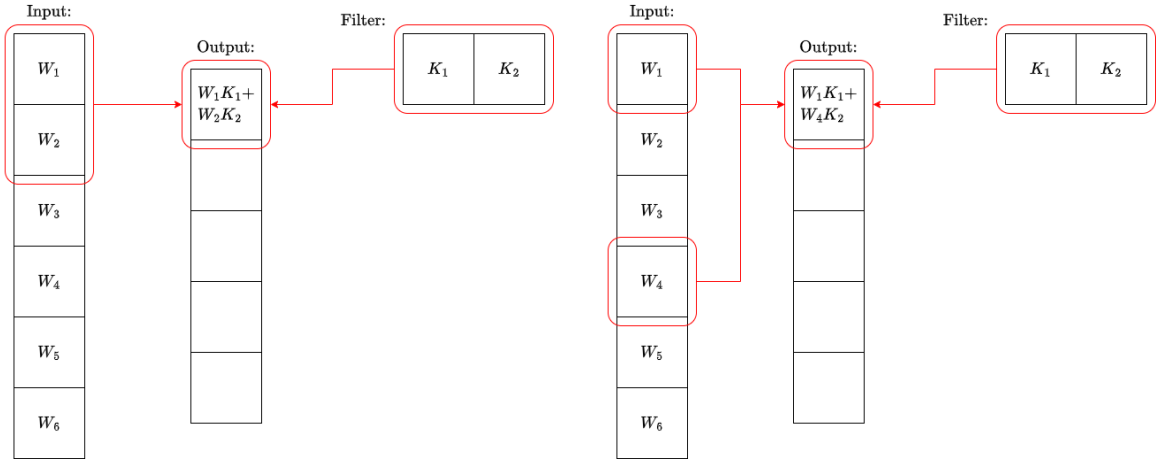

FIGURE 5.5 1D Dilated CNN. On the left the dilatation rate is 1 while on the right is 3.

5.6.2 Pooling

A typical convolutional layer has three stages:

1. **1st**. Perform several convolutions in parallel to produce linear activations.
2. **2nd**. Apply a nonlinear activation function (e.g., ReLU), sometimes called the **detector** stage.
3. **3rd**. Apply a **pooling function** to summarize local outputs.

A pooling function replaces outputs at each location with a summary of nearby outputs. For example, **max pooling** [ZC88] computes:

$$S(i,j) = \max_{m,n}(I(i-m, j-n)K(i,j))$$

Other options include average pooling, L_2 pooling, or weighted averages. Pooling introduces approximate **translation invariance**: small input shifts do not strongly affect most pooled outputs. This improves statistical efficiency when exact feature location is less important than feature presence.

Pooling also allows fewer units than detectors, e.g., by summarizing regions spaced k pixels apart, reducing computation and memory for subsequent layers. It is particularly useful for handling variable-sized inputs, as pooling can adjust offsets to produce a fixed-size summary for the classification layer.

Theoretical work guides pooling choices [BPL10], and more advanced approaches include dynamic pooling per input [BRPL11] or learning a single pooling structure applied to all inputs [JHD12].

5.6.3 Variants

In neural networks, convolution usually differs slightly from the standard discrete convolution in mathematical literature. Practically, convolution is applied in parallel with multiple kernels, allowing each layer to extract many features at many locations. Inputs are typically grids of vector-valued observations, such as color images with RGB channels. In multilayer networks, the output of one layer—often a 3-D tensor with channel and spatial indices—serves as input to the next layer. Software implementations usually include a batch dimension, forming 4-D tensors, but we omit it here for clarity.

Multi-channel convolutions are generally not commutative unless the number of input and output channels matches. CNN implementations also typically support **stride**, which downsamples the output to reduce computational cost, and **zero padding**, which allows control of the spatial size of the output independently of the kernel size. Zero padding prevents rapid shrinking of the spatial dimensions, preserving the expressive power of the network.

Beyond forward convolution, CNN implementations require operations for back-propagation:

computing gradients with respect to kernels and inputs. These operations may involve the transpose of the convolution operation, especially for strided convolutions, and are necessary for training networks with multiple hidden layers or reconstruction tasks [SVLD91, Goo10]. Additionally, biases can be shared across outputs or learned separately for each spatial location. Separate biases may slightly reduce statistical efficiency but allow the model to adjust for spatial differences in input statistics, for example when using implicit zero padding.

Efficient Convolution Algorithm. Convolution can be accelerated by using Fourier transforms or exploiting **separable kernels**, where a d-dimensional kernel is expressed as the outer product of d vectors. Separable kernels reduce runtime and storage from $O(w^d)$ to $O(w \times d)$, since multidimensional convolution can be replaced by d one-dimensional convolutions. Efficient convolution algorithms, including approximate methods, are crucial in practice, especially for large networks and deployment-focused applications.

5.7 Recurrent Neural Network

Recurrent neural networks (RNNs) are designed to process sequential data [RHW86b]. Unlike convolutional networks for grid-structured inputs, RNNs handle sequences $x^{(1)}$, $\ldots, x^{(\tau)}$, allowing variable-length sequences and long-term dependencies. A key feature is parameter sharing across time, enabling generalization to unseen sequence lengths. One-dimensional convolution over time forms the basis of time-delay networks [LH88, WHH$^+$89, LWH90], but these remain shallow; RNNs propagate information recursively using shared parameters, producing a deep computational graph.

A computational graph formalizes computations from inputs and parameters to outputs and loss. Unfolding a recurrent computation creates a chain-structured graph, making parameter sharing explicit. For a dynamical system:

$$s^{(t)} = f(s^{(t-1)}, \theta)$$

unfolding for $\tau = 3$ gives:

$$s^{(3)} = f(f(s^{(1)}, \theta), \theta)$$

Similarly, with external input $x^{(t)}$:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}, \theta)$$

The unfolded graph illustrates information flow forward and backward through time.

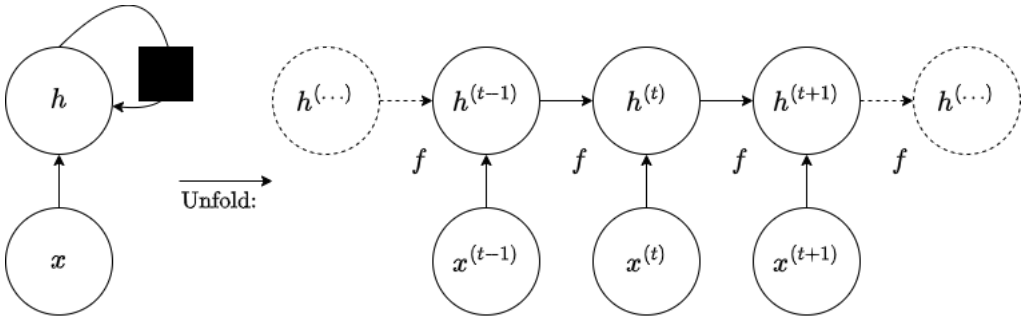

FIGURE 5.6 A recurrent network with no outputs. Unfolding maps the recurrent circuit to a computational graph of repeated components, whose size depends on sequence length.

The unfolded recurrence can be written as:

$$h^{(t)} = g^{(t)}(x^{(t-1)}, \ldots, x^{(1)}) = f(h^{(t-1)}, x^{(t)}, \theta)$$

with two advantages: the model always has the same input size, and the same transition function f with shared parameters applies at every time step. This enables learning a single model for sequences of any length and generalizing to unseen lengths with fewer examples.

RNN architectures include:

- Output at each time step with hidden-to-hidden recurrence;
- Output at each step with recurrence from outputs to next hidden units;
- Full sequence read-in with a single output at the end.

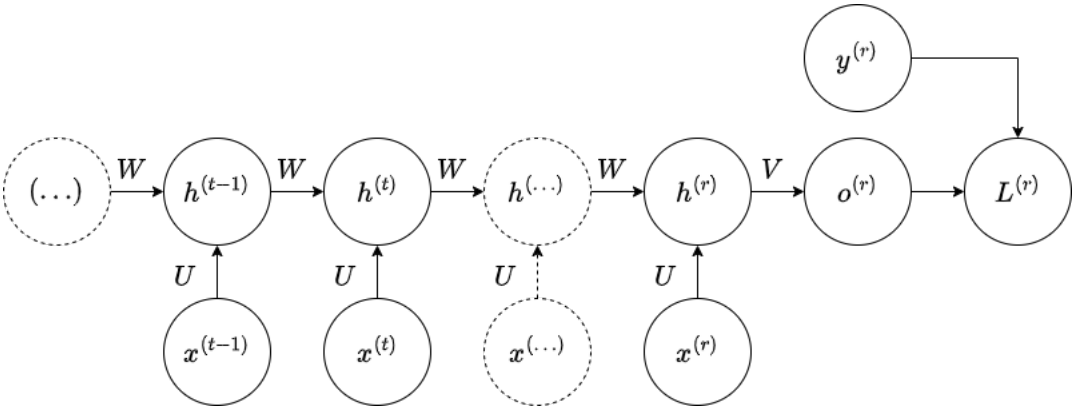

FIGURE 5.7 Time-unfolded recurrent neural network producing a single output at the end of the sequence.

For the last type (Figure 5.7), input-to-hidden connections are parametrized by U, hidden-to-hidden by W, and hidden-to-output by V. The network summarizes the

sequence into a fixed-size representation. Gradients on the output $o^{(t)}$ can be computed via backpropagation from downstream modules. For a deeper analysis of other RNN types, see Chapter 10 of [GBC16].

Chapter 6

Neural Cryptanalysis

The seminal work of Aron Gohr [Goh19] presents a novel intersection between cryptanalysis and machine learning, demonstrating how deep learning techniques can significantly enhance classical cryptanalytic methods. The work focuses on the lightweight block cipher Speck 32/64 and investigates its security under round-reduced conditions, i.e., versions of the cipher with fewer than the full number of rounds. Traditionally, distinguishing attacks and key search strategies in symmetric cryptography have relied on statistical models such as differential or linear cryptanalysis, which exploit predictable patterns in how input differences propagate through cipher rounds. Gohr's contribution challenges this conventional wisdom by showing that deep residual neural networks—a class of deep learning models—can act as powerful cryptographic distinguishers that outperform standard techniques in certain scenarios. Specifically, the study demonstrates that these neural distinguishers achieve a much lower mean key rank in a chosen-plaintext attack on nine-round Speck 32/64 than comparable classical distinguishers. Beyond constructing machine-learning-based distinguishers, the paper introduces a Bayesian optimization-based key search policy which, when combined with neural models, reduces the effective security of 11-round Speck 32/64 to approximately 38 bits—a notable improvement over earlier results in the literature. Additionally, the author demonstrates that deep learning models can automatically discover effective input differences without relying on human cryptanalytic insight, highlighting the adaptability of learned cryptanalytic features. This research is part of a growing body of work applying machine learning to cryptographic security, illustrating how modern data-driven approaches can uncover subtle structures in cipher outputs that are invisible to purely mathematical or statistical analyses.

The convergence of Deep Learning (DL) and Cryptanalysis represents one of the most intriguing frontiers in modern cryptographic research. While the previous chapters

explored classical statistical attacks—such as Differential and Linear Cryptanalysis—where the cryptanalyst manually identifies predictable patterns in the cipher's structure, this chapter investigates a paradigm shift: the use of artificial neural networks to automatically learn these patterns from data.

This domain, often referred to as **Neural Cryptanalysis**, gained significant traction with the seminal work of Aron Gohr in 2019 [Goh19]. As detailed in the opening of this chapter, Gohr's research challenged the conventional wisdom by demonstrating that deep Residual Neural Networks (ResNets) could distinguish round-reduced versions of the lightweight block cipher Speck 32/64 from random permutations with higher accuracy than state-of-the-art classical distinguishers. This was a pivotal moment, proving that deep learning models can act as powerful *cryptographic distinguishers* by uncovering subtle correlations in the ciphertext that are often invisible to purely mathematical analyses.

The chapter is structured to guide the reader through the evolution and mechanics of this approach. We begin in **Section 6.1** by establishing the baseline: the **Classical Distinguisher**. Here, we define the theoretical limits of differential attacks using the difference distribution table (DDT) and the concept of a perfect differential distinguisher based on likelihood ratios. This serves as a benchmark to evaluate the performance of machine learning models.

Section 6.2 delves into the core contribution: the **Neural Distinguisher**. We dissect the architecture proposed by Gohr, which utilizes a specialized input representation and a specific depth of residual blocks to "learn" the differential propagation of the cipher. We discuss the rationale behind the network design, including the use of initial 1D convolutions to model bit-sliced operations and the training strategy involving Cyclic Learning Rates and Curriculum Learning to tackle higher numbers of rounds.

However, a distinguisher alone is not an attack. **Section 6.3** bridges the gap between distinguishing and breaking a cipher. It details the methodology for converting the neural distinguisher into a practical **Key Recovery Attack** on 9-round Speck 32/64. This involves complex techniques such as prepending differential transitions, using neutral bits to boost the signal, and employing a Bayesian optimization strategy to rank key candidates efficiently.

Section 6.4 addresses a major limitation of early neural cryptanalysis: the need for cipher-specific tuning. We introduce **AutoND** (Automatic Neural Distinguisher), a framework proposed by Bellini et al.[BGH+23]. This section describes a generic pipeline that uses evolutionary algorithms to automatically search for optimal input differences—maximizing a defined "bias score"—and trains a generalized network architecture called **DBitNet**. This represents the current state-of-the-art, moving towards fully automated, "push-button" tools for evaluating the security of cryptographic primitives against AI-driven attacks.

To further extend the scope of neural distinguishers beyond the single-pair setting, **Section 6.5** introduces the **Multipair-Distinguisher**, an enhanced approach that aggregates information from multiple plaintext–ciphertext pairs to amplify the distinguishing signal and improve the reliability of neural-based attacks against round-reduced ciphers.

Finally, **Section 6.6** surveys additional advances in Neural Cryptanalysis drawn from the broader literature, with a particular focus on the results reported in [GLN22], highlighting how neural distinguishers and automated pipelines have been generalized to other ciphers and attack scenarios.

6.1 Classical Distinguisher [Goh19]

Applying this theoretical results with the differential properties analysis showed in [LM02], Gohr creates a classical distinguisher. To distinguish between examples of real ciphertext pairs and examples generated at random, he assumed that random ciphertext pair differences are distributed according to the uniform distribution on nonzero ciphertext blocks. He classifies an observed output difference as real if the predicted probability of observing it p_δ in the real distribution is greater than the uniform one:

$$p_\delta > \frac{1}{2^{32}-1}$$

and as random otherwise. This exploits the non-uniformity of the output difference distribution perfectly if his prediction of this distribution does not contain errors. This classical distinguisher observes a ciphertext pair that has been generated from a known input difference (but unknown input plaintext pairs) and try to guess based on the ciphertext difference whether the observed pair has been generated by reduced Speck encryption or randomly chosen. It is clear that one could improve on this by considering not only the difference data for an observed ciphertext pair, but the entire data observed. However, this is more difficult, because calculating the full distribution of ciphertext pairs for the real distribution is not feasible.

6.1.1 Perfect Differential Distinguisher

He has developed a perfect distinguisher for the 5 round-reduced Speck 32/64. Given an observed ciphertext pair $C := (C_0, C_1)$ and an input difference Δ, we denote with the likelihood

$$\mathbb{P}_{real}(C)$$

the probability that C is actually encrypted by a pair of fixed input difference Δ. We would observe (C_0, C_1) under the real distribution for a block cipher E of block size b

and key size k given uniformly random key and plaintext data is given by

$$\mathbb{P}_{real}(C) = \frac{N}{2^{(b+k)}}$$

where N is the number of key and plaintext pairs (K, P) such that

$$E_K(P) = C_0 \quad \text{and} \quad E_K(P \oplus \Delta) = C_1$$

On the other hand,

$$\mathbb{P}_{random}(C) = \frac{1}{2^{2b} - 2^{b+1} + 1} \approx \frac{1}{2^{2b} - 2^b}$$

For perfect classification we need to determine whether

$$\mathbb{P}_{real}(C) > \mathbb{P}_{random}(C)$$

or in othwe word whether

$$N_{keys}(C) > \frac{2^{b+k}}{2^{2b} - 2^b} \approx 2^{k-b}$$

or not.

6.2 Neural Distinguisher [Goh19]

Motivation [Goh19]. Recent advances in deep learning have led to significant breakthroughs across a variety of challenging tasks, ranging from machine translation [WSC+16, BCB14] and autonomous driving [CSKX15] to achieving superhuman performance in several abstract board games [CS15, SHM+16, SHS+17]. In the field of cryptography, practical applications of machine learning have so far mainly focused on side-channel analysis [MPP16, PHG17, PSH+18]. At a more theoretical level, the strong connection between cryptography and machine learning has long been acknowledged, as discussed for example in existing surveys of the area [Riv91]. Many cryptographic problems can be naturally formulated as learning problems, and cryptographic hardness assumptions can even be used to construct distributions that are intentionally difficult to learn. Nevertheless, despite this close conceptual relationship, relatively little work has investigated machine-learning-based approaches to cryptanalysis. This paper helps bridge this gap by being the first to demonstrate that neural networks can be used to con-

struct attacks that are highly competitive with the published state of the art against a round-reduced version of a modern block cipher.

6.2.1 Gohr Neural Network

Instead of search for classical distinguisher, Gohr developed a Neural Network that label a pair of ciphertexts $C = (C_0, C_1)$ as random or encrypted. In particular, the Neural Network gives us a probability:

$$\mathcal{NN}(C_0, C_1) = p \in [0, 1] \in \mathbb{R}$$

that the pair is actualy encrypted or not:

$$\mathcal{ND}(C_0, C_1) = \begin{cases} 1 & \text{if } p > \tau \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

for a certain threshold τ where the label 1 means that the pair is actually encrypted by an encryption function E_K with key K:

$$C_0 = E_K(P) \quad \text{and} \quad C_1 = E_K(P \oplus \delta)$$

while the label 0 that is a random bit string. In other words, to be real encrypted the pair should be a probability distribution far from a certain distribution τ. The Neural Network used in [Goh19] is shown in Figure 6.3.

Input Representation Since he worked on SPECK 32/64, a pair (C_0, C_1) of ciphertexts that cipher can be written as a sequence of four sixteen-bit words (w_0, w_1, w_2, w_3), mirroring the word-oriented structure of the cipher. In our networks, the w_i are directly interpreted as the row-vectors of a 4×16-matrix and the input layer consists of 64 units likewise arranged in a 4×16 array.

$$(w_0^{(0)}, \ldots, w_0^{(15)}, w_1^{(0)}, \ldots, w_3^{(15)}) \xrightarrow{\texttt{reshape}} \begin{bmatrix} w_0^{(0)} & w_0^{(1)} & \cdots & w_0^{(15)} \\ w_1^{(0)} & w_1^{(1)} & \cdots & w_1^{(15)} \\ w_2^{(0)} & w_2^{(1)} & \cdots & w_2^{(15)} \\ w_3^{(0)} & w_3^{(1)} & \cdots & w_3^{(15)} \end{bmatrix}$$

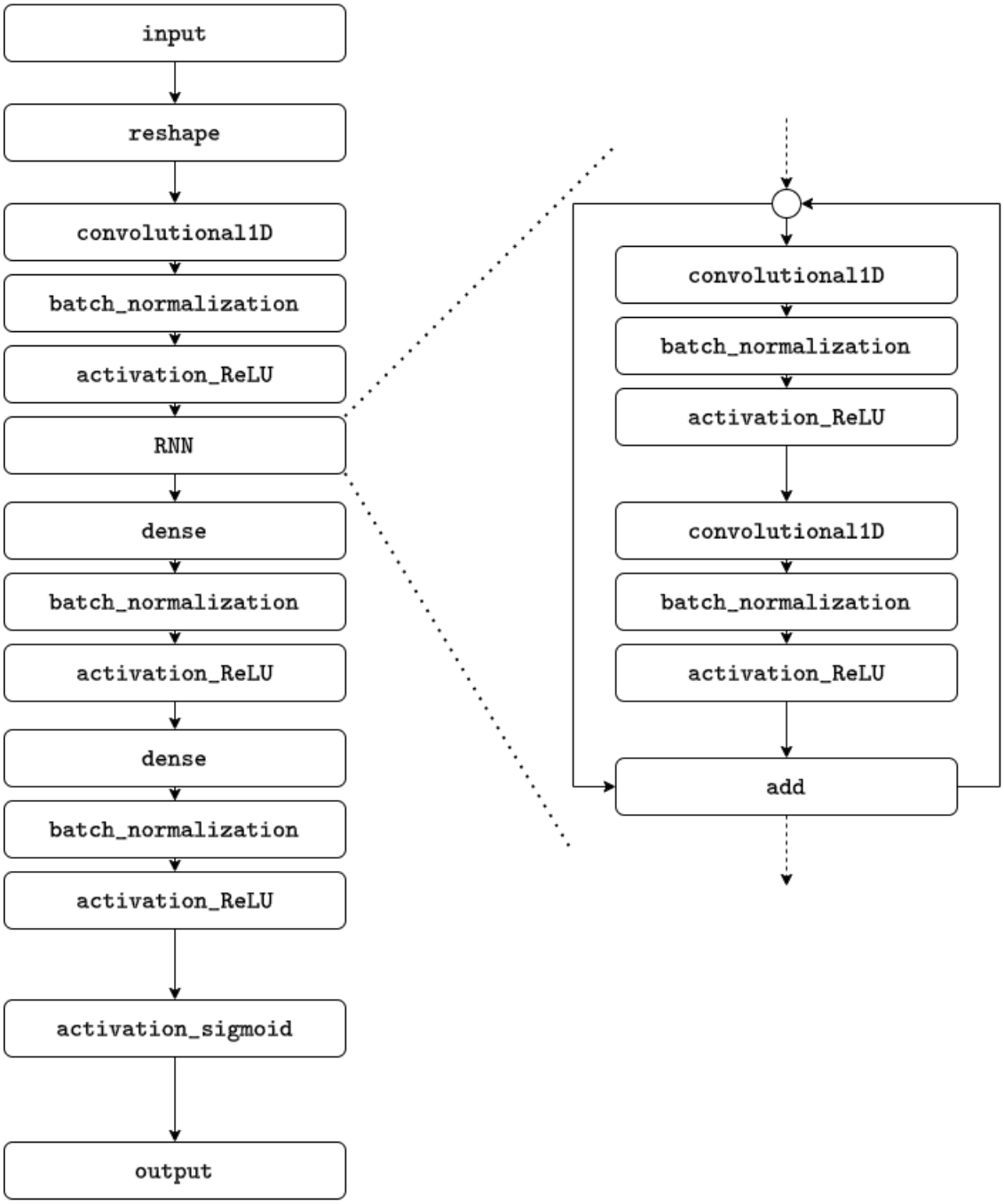

FIGURE 6.1 Neural Distinguisher of Gohr.

Initial Convolution. The input layer is connected in channels- first mode to one layer of bit-sliced, e.g. width 1, convolutions with 32 output channels (see section 5.6):

$$\begin{bmatrix} w_0^{(0)} & w_0^{(1)} & \cdots & w_0^{(15)} \\ w_1^{(0)} & w_1^{(1)} & \cdots & w_1^{(15)} \\ w_2^{(0)} & w_2^{(1)} & \cdots & w_2^{(15)} \\ w_3^{(0)} & w_3^{(1)} & \cdots & w_3^{(15)} \end{bmatrix} \xrightarrow{\texttt{conv1D}} \begin{bmatrix} c_0^{(0)} & c_0^{(1)} & \cdots & c_0^{(15)} \\ c_1^{(0)} & c_1^{(1)} & \cdots & c_1^{(15)} \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ c_{31}^{(0)} & c_{31}^{(1)} & \cdots & c_{31}^{(15)} \end{bmatrix}$$

where:

$$c_i^{(j)} = \sum_{t=1}^{4} w_t^{(j)} k_i^{(t)} + \lambda ||k||_2$$

(see Figure 6.2), with the last term a L_2 norm as regularizer (see section 5.5). Batch normalization (see subsection 5.3.3) is applied to the output of these convolutions.

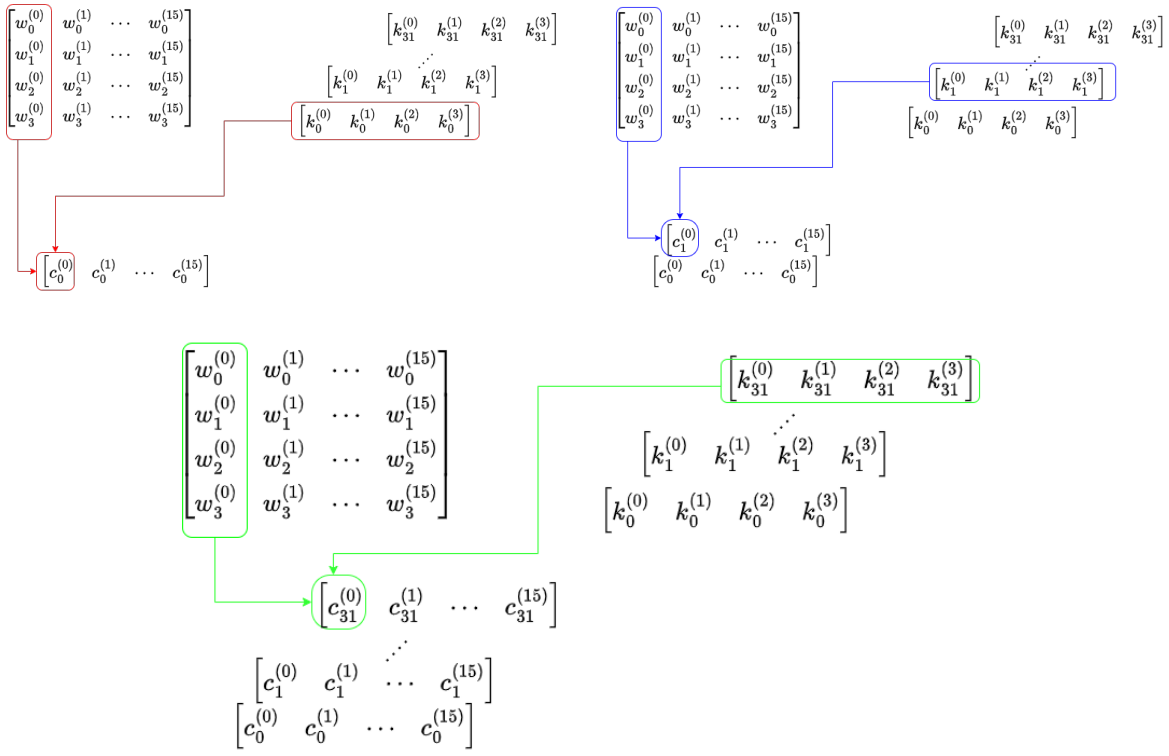

FIGURE 6.2 1D Convolutional.

$$\begin{bmatrix} c_0^{(0)} & c_0^{(1)} & \cdots & c_0^{(15)} \\ c_1^{(0)} & c_1^{(1)} & \cdots & c_1^{(15)} \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ c_{31}^{(0)} & c_{31}^{(1)} & \cdots & c_{31}^{(15)} \end{bmatrix} \xrightarrow{\texttt{batch\_norm}} \begin{bmatrix} b_0^{(0)} & b_0^{(1)} & \cdots & b_0^{(15)} \\ b_1^{(0)} & b_1^{(1)} & \cdots & b_1^{(15)} \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ b_{31}^{(0)} & b_{31}^{(1)} & \cdots & b_{31}^{(15)} \end{bmatrix}$$

where

$$b_i = \frac{c_i - \hat{\mu}}{\hat{\sigma}}$$

Finally, an activation layer with activation function ReLU, Equation 5.7, are applied to the outputs of batch normalization and the resulting 32×16 matrix is passed to the main RNN.

$$\begin{bmatrix} b_0^{(0)} & b_0^{(1)} & \cdots & b_0^{(15)} \\ b_1^{(0)} & b_1^{(1)} & \cdots & b_1^{(15)} \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ b_{31}^{(0)} & b_{31}^{(1)} & \cdots & b_{31}^{(15)} \end{bmatrix} \xrightarrow{\texttt{ReLU}} \begin{bmatrix} d_0^{(0)} & d_0^{(1)} & \cdots & d_0^{(15)} \\ d_1^{(0)} & d_1^{(1)} & \cdots & d_1^{(15)} \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ d_{31}^{(0)} & d_{31}^{(1)} & \cdots & d_{31}^{(15)} \end{bmatrix}$$

where

$$d_i = \mathrm{ReLU}(b_i)$$

RNN of Convolutional Blocks. The Gohr Network implements a RNN (see section 5.7) with convolutional block. Each convolutional block consists of two layers of 32 filters. Each layer applies first the convolutions, then a batch normalization, and finally a rectifier layer. At the end of the convolutional block, a skip connection then adds the output of the final rectifier layer of the block to the input of the convolutional block and passes the result to the next block.

Prediction Head. The prediction head consists of three hidden layers and one output unit. The first and second layer are densely connected layers with 64 units. The first and the second of these layers is followed by a batch normalization layer and a rectifier layer of 64 units each. The final layer consists of a single output unit using a sigmoid activation Equation 5.5:

$$\mathcal{NN}(C_0, C_1) = \texttt{sigmoid}(h)$$

where h is the output of the last ReLU activation layer. Then,

$$\mathcal{ND}(C_0, C_1) = \begin{cases} 1 & \text{if } \mathcal{NN}(C_0, C_1) > \tau \\ 0 & \text{otherwise} \end{cases}$$

Rationale. The use of the initial width-1 convolutional layer is intended to make the learning of simple bit-sliced functions such as bit-wise addition easier. The number of filters in the initial convolution is meant to expand the data to the format required by the residual tower. The choice of the input channels is motivated by a desire to make the word-oriented structure of the cipher known to the network. The use of a densely connected prediction head reflects the fact that for a nontrivial number of rounds, Gohr does not expect the input data to show strong spatial symmetries, so any attempt to extract local features from the data using a spatially symmetric pooling layer of some sort is probably futile. The size of the layers was determined by experiment, although we tried only a few settings. The depth of the residual tower was chosen so as to allow for integration of input data over the whole input string within the convolutional layers. However, even a design with just one residual block achieves reasonably good (clearly superior to a purely differential distinguisher) results.

Basic Training. Pipeline Training was run for 200 epochs on the dataset of size 10^7. The datasets were processed in batches of size 5000. The last 10^6 samples were withheld for validation. Optimization was performed against Mean Square Error Loss plus a small penalty based on L_2 weights regularization (with regularization parameter $\epsilon = 10^{-5}$):

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i(\theta) - y_i)^2 + \epsilon||\theta||_2 \tag{6.2}$$

where $\hat{y}(\theta)_i = f(x_i, \theta)$ is the output of the $\mathcal{NTD}$. The choice of this function is driven by [GLN22]. Indeed, given the probability p_δ for every output difference $\delta \in \{0,1\}^n \setminus \{(0,\ldots,0)\}$ (and a fixed input difference) a distinguisher could just use:

$$p_\delta > \frac{1}{2^n - 1}, \quad \tau = \frac{1}{2^n - 1}$$

as a decision rule, i.e. the ciphertext-pair is coined to be long to the encryption distribution if the probability of the observed ciphertext difference is greater in the encryption case than the probability of the same difference to appear in the random case. With this, the accuracy of such a distinguisher would be:

$$\frac{1}{2}\sum_{p_\delta > \frac{1}{2^n-1}} p_\delta + \frac{1}{2}\sum_{p_\delta \leq \frac{1}{2^n-1}} \frac{1}{2^n - 1} \tag{6.3}$$

assuming that both, the encryption and the random case, appear with probability $\frac{1}{2}$, as it is the case for training the neural network (and evaluating its accuracy). Note that if both cases are equiprobable, this is actually the best that can be done if only the ciphertext-differences of single samples are considered. Interestingly, this accuracy is basically the same as the mean absolute distance of the ciphertext-difference distri-

bution and the uniform one [GLN22]. This means that maximizing the accuracy of such a distinguisher (over the chosen input difference) may not lead to the same input difference as if one tries to maximize the probability of a differential, as the former maximizes the mean absolute distance, while the later maximizes the maximal distance. It was used the ADAM algorithm with default parameters in Keras [C+15].

Learning Rate. A cyclic learning rate schedule was used, setting the learning rate (Equation 5.4) l_i for epoch i to:

$$l_i = \alpha + \frac{n - i \bmod n + 1}{n} \cdot (\beta - \alpha), \text{ with } \alpha = 10^{-4}, \beta = 2 \cdot 10^{-3} \text{ and } n = 9$$

The networks obtained at the end of each epoch were stored and the best network by validation loss was evaluated against a test set of size 10^6 not used in training or validation.

Choice of the input difference. The choice in the input difference used for the different rounds of the curriculum learning is based by the study of the difference propagation based on [BG11]. Indeed, he switched differences according to the round to help the training of the SPECK distinguisher. In particular, [Goh19] used a curriculum learning (see subsection 5.3.4) to train the $\mathcal{ND}$. For 8 rounds, the training scheme described above fails, i.e. the model does not learn to approximate any useful function. He still succeeded in training an 8-round distinguisher slightly superior to the difference distribution table by using several stages of pre-training. First, he retrained the best seven-round distinguisher to recognize 5-round SPECK 32/64 with the input difference

$$\texttt{0x8000}, \texttt{0x840a}$$

This was done on 10^7 examples for ten epochs with a batch size of 5000 and a learning rate of 10^{-4}. Then, he trained the distinguisher so obtained to recognize 8-round SPECK with the input difference

$$\texttt{0x0040}, \texttt{0x0000}$$

by processing 10^9 freshly generated examples once with batch size 10000, keeping the learning rate constant. Finally, learning rate was dropped twice to 10^{-5} and finally to 10^{-6} after processing another 10^9 fresh examples each, again with a batch size of 1000.

Results. [Goh19] obtained better results than classical pure differential distinguisher with specific modification according the round he was attacking. Overall, the neural distinguishers from 5 to 8 rounds achieve higher accuracies than classical differential baselines across all tasks, though they do not reach the accuracy of key-search-based distinguishers. Training and validation losses were close at the end of training, indicat-

ing only mild overfitting. Algorithm 1 of [Goh19] applied to the seven-round problem slightly improved prediction accuracy, correctly predicting 62.7% of the test samples. A mild key-dependency was observed when encryption used fixed keys, consistent with previous SPECK studies; for example, the true positive rate of the seven-round distinguisher varied between 49.7% and 57.1% across 100 random keys. Empirical testing was performed on large test sets (∼500,000 positive and negative examples per set). Classical differential distinguishers rely on the full difference distribution table of SPECK 32/64 under the Markov assumption, while neural distinguishers solve the same distinguishing task.

6.3 Key Recovery [Goh19]

We can extend all of the distinguishers here discussed by one round at no additional cost by using the fact that the first subkey addition happens after the first application of nonlinearity in SPECK. An adversary in the chosen-plaintext setting can easily inject plaintext differences of their choosing into the output of the first round of SPECK. A simple attack on 9-round SPECK can then be performed as follows:

1. Request encryptions for n chosen plaintext pairs (P, P') with fixed input difference $\texttt{0x0040}, \texttt{0x0000}$. Obtain the corresponding ciphertext pairs (C, C'), using the right (unknown) key K^*.
2. For each value of the final subkey K, decrypt the ciphertext under K to get C^k . Let δ^K be the difference of the ciphertext pair $(C^K, (C')^K)$.
3. Use a 7-round differential distinguisher to get scores Z_i^K for each partially decrypted ciphertext pair i:

$$Z_i^K = \mathcal{NN}((C^K, (C')^K)_i)$$

4. For each K, combine the scores Z^K into one score v^K:

$$v^K = \sum_{i=1}^{n} \log_2 \left(\frac{Z_i^K}{1 - Z_i^K} \right) \tag{6.4}$$

5. Sort the keys in descending order according to their score v^K

Then, it is possible to extend this attack using the neural 7-round distinguisher to a 9-round distinguisher by prepending a two-round differential transition: $\delta_{in} \to \delta_0$

The probability, fixing $\delta_0 = \texttt{0x0400}, \texttt{0000}$, is passed as desired with a probability of about 1/64. The 9-round distinguisher is then extended by following the previous steps.

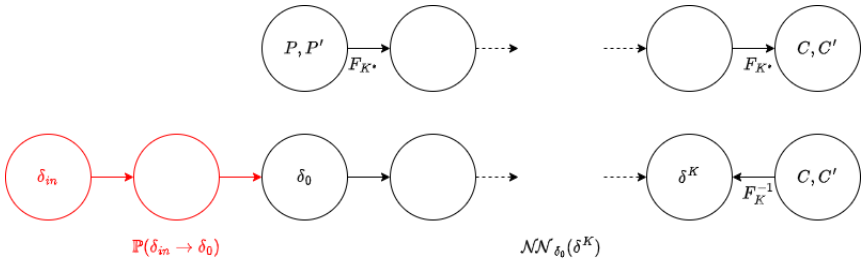

FIGURE 6.3 Prepending two-round.

The signal from this distinguisher is rather weak. Gohr therefore boost it by using k (probabilistic) neutral bits (see Definition 7.5,7.6) to create from each plaintext pair a plaintext structure consisting of 2^k plaintext pairs that are expected to pass the initial two-round differential together. For each plaintext structure, he decrypts the resulting ciphertexts under all final subkeys and rank each partially decrypted ciphertext structure using our neural distinguisher. If the resulting score is beyond a threshold c_1, we attempt to decrypt another round and grade the resulting partially-decrypted ciphertexts using a six-round neural distinguisher. A key guess is returned if the resulting score for the partially decrypted ciphertext structure then exceeds another threshold c_2.

Bayesian Optimization. Bayesian optimization [PGCP99] is a method that is commonly used for the optimization of black box functions f that are expensive to evaluate. In [Goh19], he employs Bayesian optimization to design an effective key-search strategy for reduced-round SPECK. The proposed strategy significantly decreases the number of trial decryptions required by the baseline attack, at the expense of an additional but moderately costly optimization phase. The core idea is that the expected output of the distinguisher under an incorrect key depends on the bit-wise difference between the tested key and the true secret key. This **wrong-key response profile** can be learned through a preliminary precomputation phase. Based on the responses observed from a set of trial decryptions, the optimization procedure then selects new key candidates to test, choosing them so as to maximize the likelihood of the observed distinguisher outputs.

Results. The details of the other optimization steps are detailed in [Goh19]. An attack is considered successful if the last round key is recovered exactly and the second round key differs from the true key by at most two bits. Under this criterion, the attack succeeds in 521 out of 1000 trials, corresponding to a success rate of about 52%. In all successful cases, the error in the second round key is confined to at most two bits in the most significant nibble. This performance is comparable to previous work, which reports a success rate of approximately 55% under the same data complexity. On average, the proposed attack is expected to require around 2145 chosen plaintexts to achieve success.

6.4 AutoND [BGH+23]

AutoND is the first automatized pipeline introduced in [BGH+23] to create a $\mathcal{ND}$ with a general approach. In this paper, they proposed a step forward towards the fully automated route, through a generic pipeline: suitable input difference δ_{in} candidates are obtained through an evolutionary algorithm, and are used to train DBitNet, a fully generic neural network that requires no tuning nor human input. The neural distinguishers obtained through their pipeline are competitive with, and sometimes better than, specialized approaches on the ciphers for which they were designed. With this work, they aim to provide a basis on which other researchers can improve neural cryptanalysis, and apply it to more ciphers, without the burden of optimizing the neural distinguisher itself.

6.4.1 Choiche of Input Difference [BGH+23]

The generic pipeline to find an input difference is based on a meta optimizier defined in Algorithm 17. This optimizer operates on the premise that neural distinguishers detect truncated differentials from earlier rounds [BGPT21]. In particular, [BGH+23] conjecture that Benamira et al.'s conclusions generalize to other ciphers, and that high biases in individual bits are a good approximation for the presence of high probability truncated differentials, which are otherwise difficult to find in a generic way. If this conjecture is correct, then highly biased difference bits at round r should lead to good neural distinguishers at round $r+\theta$ through differential-linear properties. Therefore, it is possible to assume a good input difference for neural distinguisher is one for which high biases exist in the difference bits of the higher rounds. This assumption is verified empirically via experiments, as the neural distinguishers usually cover several rounds past the highest round where a bias was detected. They focus on the problem of finding the optimal input difference (for neural distinguishers) cryptographically, under the assumption that this input difference maximizes the bias of intermediate difference bits. More formally, we assume that a good input difference for neural distinguishers is one that maximizes a bias score, defined as:

Definition 6.1 (Bias Score)**.** Let $r \in \mathbb{Z}^+$ be the number of rounds, $E^r : \mathbb{F}_2^n \times \mathbb{F}_2^k \to \mathbb{F}_2^n$ be a block cipher with block length n and key length k, and $\delta_{in} \in \mathbb{F}_2^n$ be the input difference. The **exact bias score** for δ, $b(\delta)$ is the sum of the biases of each bit position j in the output difference:

$$b_r(\delta) = \frac{1}{n}\sum_{j=0}^{n-1}\left|\frac{1}{2} - \frac{\sum_{x\in\mathbb{F}_2^n, K\in\mathbb{F}_2^k\setminus\{0\}}(E_K^r(x)\oplus E_K^r(x\oplus\delta))_j}{2^{n+k}}\right| \tag{6.5}$$

The exact bias score cannot be computed for practical ciphers, as it requires enumerating all keys and plaintexts. On the other hand, [BGH+23] proposed to use an approximation, obtained from a limited number of samples t:

Algorithm 17 `evolutionary_optimizer` [BGH+23]. It starts from an initial population of random input differences, and improves the population iteratively by deriving new candidates from known ones (using a mutation probability M), ranking them through their bias score, and allowing the best ones to move to the next generation. The algorithm stops if no input difference scores higher than a threshold T_b. In practice, the initial population contains 1024 differences, the 32 best ones are kept at each generation, and we set $M = 1$, $t = 104$, and $T_b = 0.01$.

`starting_population` $\leftarrow$ [`RandomInt`$(0, 2n1)$ `for` 1024 `times`]
Sort `starting_population` by $\tilde{b}^t(\cdot)$ in descending order
`current_population` $\leftarrow$ first 32 elements of `starting_population`
`score` $\leftarrow$ max value of $\tilde{b}^t$ among the starting differences
$r \leftarrow 0$
while `score` $< T_b$ **do**
    $r \leftarrow r + 1$
    for iteration$= 0, \ldots, 50$ **do**
        `candidates` $\leftarrow$ [ ]
        for $i = 0, \ldots, 32$ **do**
            for $j = i + 1, \ldots, 32$ **do**
                if `Random_Float`$(0, 1) < M$ **then**
                    $m \leftarrow 1$
                else
                    $m \leftarrow 0$
                end if
                `app` $\leftarrow$ `current_population`$_i \oplus$ `current_population`$_j \oplus (m \ll$ `Random_Int`$(0, n1))$
                `candidates` $\leftarrow$ `candidates` $\cup$ `app`
            end for
        end for
        Evaluate $S_{\delta,r}$ for all δ in `candidates`
        Sort `candidates` by $S_{\delta,r}$ in descending order
        `current_population` $\leftarrow$ first 32 elements of `candidates`
    end for
    `score` $\leftarrow$ max value of $S_{\delta,r}$ among the `current_population`
end while
return `current_population`, r

Algorithm 18 `evolutionary_optimizer_fixed_R`. It starts from an initial population of random input differences, and improves the population iteratively by deriving new candidates from known ones (using a mutation probability M), ranking them through their bias score, and allowing the best ones to move to the next generation. The algorithm stops if no input difference scores higher than a threshold T_b. In practice, the initial population contains 1024 differences, the 32 best ones are kept at each generation, and we set $M = 1$, $t = 104$.

Require: The number of rounds R

`starting_population` $\leftarrow$ [`RandomInt`$(0, 2n1)$ `for` 1024 `times`]
Sort `starting_population` by $\tilde{b}^t(\cdot)$ in descending order
`current_population` $\leftarrow$ first 32 elements of `starting_population`
`score` $\leftarrow$ max value of $\tilde{b}^t$ among the starting differences
for $= 1, \ldots, R$ **do**
    for iteration$= 0, \ldots, 50$ **do**
        `candidates` $\leftarrow$ [ ]
        for $i = 0, \ldots, 32$ **do**
            for $j = i + 1, \ldots, 32$ **do**
                if `Random_Float`$(0, 1) < M$ **then**
                    $m \leftarrow 1$
                else
                    $m \leftarrow 0$
                end if
                `app` $\leftarrow$ `current_population`$_i \oplus$ `current_population`$_j \oplus (m \ll$ `Random_Int`$(0, n1))$
                `candidates` $\leftarrow$ `candidates` $\cup$ `app`
            end for
        end for
        Evaluate $S_{\delta,r}$ for all δ in `candidates`
        Sort `candidates` by $S_{\delta,r}$ in descending order
        `current_population` $\leftarrow$ first 32 elements of `candidates`
    end for
end for
return `current_population`

Definition 6.2 (Approximated Bias Score)**.** Let $r \in \mathbb{Z}^+$ be the number of rounds, $E^r : \mathbb{F}_2^n \times \mathbb{F}_2^k \to \mathbb{F}_2^n$ be a block cipher with block length n and key length k, and $\delta_{in} \in \mathbb{F}_2^n$ be the input difference. The **approximated bias score** for δ, $\tilde{b}^t(\delta)$, is the sum of the biases of each bit position j in the output difference, computed for t samples:

$$\tilde{b}_r(\delta) = \frac{1}{n} \sum_{j=0}^{n-1} \left| \frac{1}{2} - \frac{\sum_{i=0}^{t-1} (E_{K^i}^r(x^i) \oplus E_{K^i}^r(x^i \oplus \delta))_j}{t} \right| \tag{6.6}$$

The conjecture used in [BGH+23] is the followed:

Conjecture. *Input differences δ that reach the most rounds with a neural distinguisher have a high bias score $b(\delta)$. We further assume that $\tilde{b}^t(\delta)$ is a good estimation of $b(\delta)$.*

Since the optimizer is heuristic, some good differences may have been identified in a subset of the rounds only; we therefore rerun the scoring procedure for the union of these lists, to obtain, for each difference δ_i, R bias scores $\tilde{b}^t_{\delta_i,r}$. Then, we evaluate the final score:

$$S_{\delta_i,R} = \sum_{r=1}^{R} \tilde{b}^t_{\delta_i,r} \tag{6.7}$$

According to Equation 6.7, the optimizer sorts the differences according the total score they have obtained until that point.

Notice also that the round at which a difference is evaluated is an important parameter. We can have two possible cases. In the original word, the most relevant round is not known in advance, then, they run the optimizer iteratively from round 1 to round $R + 1$, where no bias score above the threshold T_b is returned, obtaining R lists of 32 differences Δ_r for $r \in [1, R]$. In this way, they didn't just identify the theoretical optimal input differences but also the round (see Algorithm 17). In the second case, we know the round R we would like to attack, so we lunch the optimizer for that number (see Algorithm 18).

6.4.2 DBitNet

At FSE 2024, DBitNet was introduced as a "cipher-agnostic" architecture [BGH+23]. This architecture avoids cipher-specific components, employs a simplistic design, and achieves state-of-the-art results across various primitives without requiring any modifications or hyperparameter adjustments. The input of DBitNet is the same of the $\mathcal{ND}$ of Gohr (see Equation 6.1). In particular, it is a pair (C, C') with block length n.ì;

$$(w_0^{(0)}, w_0^{(1)}, \ldots, w_0^{(n-1)}, w_1^{(0)}, \ldots, w_1^{(n-1)})$$

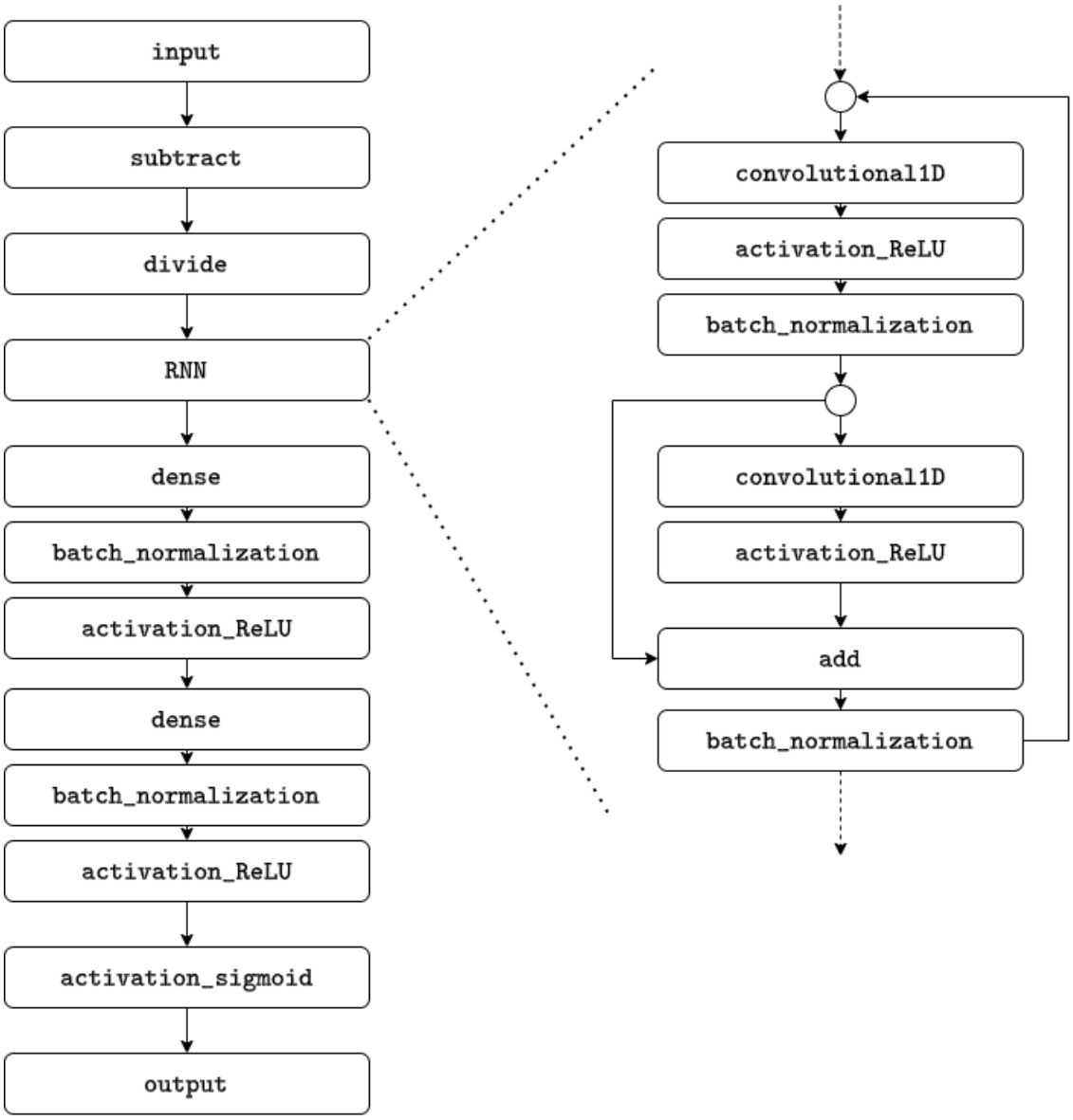

FIGURE 6.4 Neural Distinguisher DBITNET.

Gohr's neural distinguisher is immensely successful as a distinguisher for SPECK. However, [BGH+23] identified a range of hyperparameters that need tuning for application to new ciphers the most important among them again being the input reshaping.

Reshape. The input reshaping serves to investigate dependencies of far-apart bits as well as neighboring bits in the 64-bit input: for example, the bit-slicing filter may learn functions between bits (1, 17, 33, 49) while the following $k = 3$ filter may learn functions between neighboring bits $(1, 2, 3)$. In this way, near and long-range dependencies among the bits can be learned. Therefore, the input reshaping can potentially be avoided, given another, more generic way to investigate near, as well as long-range dependencies.

Normalization. It is applied a normalization of the values from the range $[0, 1]$ to $[-1, 1]$. In particular:

$$a_i^{(j)} = \frac{w_i^{(j)} - 0.5}{0.5}, \quad \forall i = 0, 1 \ \forall j = 0, \ldots, n-1$$

Rotational For DBitNet. The idea in the RNN of DBITNET is to use the Dilated CNN (see subsection 5.6.1) to analyze the long and shor-term dependencies among the bits. Indeed they implement two Dilated 1D CNN. One 1D Convolutional layer is used to analyze the long-term dependencies among the bits, indeed it has a variational dilatation rate according to Formula (Equation 6.8),

$$dr_0 = \left\lfloor \frac{\text{input size}}{2} - 1 \right\rfloor \quad dr_i = \left\lfloor \frac{r_{i-1} + 1}{2} - 1 \right\rfloor \tag{6.8}$$

while the second 1D Convolutional layer analyze the short-term dependencies with dilatation rate fixed to 1. Both of them the filters has dimension 2 i.e. they are analyzing the dependencies in pair of bits.

It is possible to notice that using a large dilatation rate imply a shrink in the dimension of the output. This shrinking of the neuronal width dimensionality is also encountered in popular image detection networks like ResNet [HZRS15a]. As "compensation" the number of channels is increased: In ResNet 34 for example the image size is halved from 224 pixels to 112, to 56, to 28 pixels, and so on while the number of channels increases from the 3 red-green-blue channels to 64, and 128. [BGH+23] follow a similar tactic and increase the number of channels with each dilational convolution. They start with 32 filters, identical to Gohr, in the first convolutional layer. Whenever the neuronal width is halved, we add 16 filters, resulting in

$$32 + i \times 16$$

filters in the i-th dilated convolution.

The first Convolutional layer has a Batch Normalization layer, while after the second there is first an addition layer with the output of the first layer and then a Batch Normalization layer. This convolutional layer are repeated for

$$\lfloor \log_2(\texttt{input\_size}) - 3 \rfloor$$

iterations. In particular, taking the input of SPECK Neural Distinguisher (C, C') we have in total 64 bits, hence we repeat this operation for 3 times.

Prediction Head. The prediction head consists of three hidden layers and one output unit. Defined as in the Gohr net in subsection 6.2.1.

6.4.3 Training Pipeline for AutoND [BGH+23]

Algorithm 19 `Generic_Training_AutoND`.

Require: Oracle access to an r-round block cipher $E_k(\cdot)$, where $P \in \mathcal{P}$, $k \in \mathcal{K}$, and $C \in \mathcal{C}$ such that $C = E_k(P)$.

$\mathcal{ND} \leftarrow$ randomly initialize the neural network (cf. subsection 7.1.4)
$(R, \delta) \leftarrow$ `evolutionary_optmizer`
Select starting round $r_0 < R$ and input difference δ
▷ Execute curriculum learning starting from round r_0 (cf. subsection 7.1.5)
$A_{\text{val}} \leftarrow 1.0$
$r \leftarrow r_0$
while $A_{\text{val}} > 0.50 + 10\sigma \wedge r < R$ **do**
    $\mathcal{D}_{\text{train}} \leftarrow \text{GENERATEDATASET}(r, N_{\text{train}}, \delta)$
    $\mathcal{D}_{\text{val}} \leftarrow \text{GENERATEDATASET}(r, N_{\text{val}}, \delta)$
    Train $\mathcal{ND}$ on $\mathcal{D}_{\text{train}}$
    $A_{\text{val}} \leftarrow$ accuracy of $\mathcal{ND}$ on $\mathcal{D}_{\text{val}}$ ▷ Evaluate neural distinguisher
    $r \leftarrow r + 1$ ▷ Progress to next round r
end while
return $\mathcal{ND}$

Pipeline Training was run for 40 epochs on the dataset of size 10^7. The datasets were processed in batches of size 5000. The last 10^6 samples were withheld for validation. Optimization was performed against Mean Square Error Loss plus a small penalty based on L_2 weights regularization (with regularization parameter $\epsilon = 10^{-5}$) as in Equation 6.2.

The training pipeline is a curriculum learning, described in Section 5.3.4. First, they found the input difference δ and the target round R with the optimizer. Then, they don't start the training of the Neural Distinguisher starting directly from that round R, but instead the training starts from some previous rounds $r_0 < R$. Once the best model is found for that r_0, they continue to progressively train the Neural Distinguisher for $r_0 + 1, r_0 + 2, \ldots, R$ until they arrive to R (see Algorithm 19). There isn't a theory for the best starting round r_0 that leads to the best accuracy, so, they decide to start the training from a round r_0 where we know the Neural Distinguisher achives an accuracy above 90%. In the general approach, if the Neural Distinguisher reach an accuracy around 50%, it is not useful, since it doesn't distinguish anymore from random and encrypted pairs. Hence, another control is applied to make sure to have always an accuracy above a certain threshold. The important difference from the other works in this field, is that we doesn't change the difference during the training, but we keep the one found by the optimizer during all the process.

Learning Rate. In this simple pipeline for DBitNet, they avoid a learning rate schedule, as well as any manual variation of the standard optimizer settings as follows: ADAM is known as one of the most advanced optimizers, however, it has been observed to fail to converge to an optimal solution [RKK19]. Such convergence failure may make it necessary to find an optimal learning rate schedule manually. For our purposes of a generic application to a range of new target ciphers, such a manual choice should be avoided. As an alternative to either the manual mitigation of the convergence issue or an automated hyperparameter tuning of the learning rate, they used the AMSGRAD optimizer 14 [RKK19].

Polish Step. Furthermore, they improve the (already competitive) results they have obtained with a simple training pipeline by applying the simple polishing pipeline. Once we reach the target round, we retrain 5 times the best model at the same round using new and freshly dataset. At a batch size of 10.000, we use the AMSGRAD optimizer, decreasing the (constant) learning rate at each iteration, from 10^4 to 10^5 to 10^6. The three learning rates, smaller than the AMSGRAD optimizer's default value of 10^3, ensure the final convergence to an optimal solution for features that are not present in many batches. The polishing step was only added to show that also some of the most elaborate and successful training pipelines can be replaced with our automated training pipeline. Five fresh datasets (with 10^6 samples in each) are generated for the final accuracy evaluation. The expected and observed standard deviation is 0.0005 as explained in the following.

Good Neural Distinguisher. The predictions of neural distinguishers can be modeled as binomial experiments with n trials, and two equiprobable outcomes, random or not random. Since $n = 10^7$ for training and 10^6 for validation, the expected mean and

standard deviation of a distinguisher making random prediction are

$$\mu = 0.5 \cdot n, \quad \sigma = \sqrt{\frac{n}{4}}$$

or, as a percentage,

$$\sigma_{\%} = \frac{1}{\sqrt{2n}}$$

We consider the validation successful if the validation accuracy (percentage of correct guesses) exceeds ten standard deviations, i.e., $A_{\text{not random}} > 50.5\%$.

6.4.4 Results

For each target cipher, the experiments start from the input differences identified by the evolutionary optimizer. Depth-1 Gohr networks and DBitNet distinguishers are trained to separate ciphertext pairs generated from chosen plaintext differences from random pairs using a simple training pipeline. The results show the highest number of rounds reached and the corresponding best accuracy, comparing training with 10 and 40 epochs per round. In most cases, increasing the number of epochs leads to improved performance, demonstrating the benefit of longer training. The obtained distinguishers enable key-recovery extensions, allowing additional rounds to be prepended to the attack. For Simon 32/64, the results are comparable to the previous work of [Goh19], while relying on a significantly simpler and less computationally intensive training pipeline. For Simon 64/128, one additional round is achieved compared to prior results that use more ciphertext pairs, whereas specialized approaches tailored to Simon can still outperform this method for a shlight 10% [HRCF21]. Several promising input differences reported in earlier work also appear among the optimizer's solutions, though they were not further analyzed. Notably, for Simon 128/256, a new 20-round distinguisher is obtained with an accuracy of 0.5057. There are other ciphers results presented in [BGH+23].

6.5 Multipair-Distinguisher

Until now, we have discussed about single pair distinguisher i.e. a Neural Network that takes in input a single pair (C, C') and classify it as random or an actual encrypted pair, see 6.1. For multi-pairs distinguisher we refer to the Neural Networks that takes in input different pairs $(C_1, C'_1, C_2, C'_2, \ldots, C_n, C'_n)$ and it gives the label 1 if they are all encrypted or 0 if none of them is. In particular, fixed an input difference Δ and taking plaintext pairs (P_i, P'_i) such that $P_i \oplus P'_i = \Delta$ we denote with E_K the encryption

function ($C_i = E_K(P_i)$), then:

$$\mathcal{ND}(C_1, C'_1, C_2, C'_2, \ldots, C_n, C'_n) = \begin{cases} 1 & \text{if } E_K^{-1}(C_i) \oplus E_K^{-1}(C'_i) = \Delta \ \forall i \\ 0 & \text{otherwise} \end{cases}$$

6.5.1 [LLS+24]

[LLS+24] continues the work inspired by [Goh19]. In particular, they analyze the decryption part of SIMECK and SIMON (see Figure 2.2.1 for better clarification). So, they notice that, knowing the ciphertext pair at i-th round $C^i = L^i||R^i$ and $(C')^i = (L')^i||(R')^i$, it is possible to deterministically, i.e. without key dependencies, recover the differece at the round $i-1$ (see Equation 7.3 and 7.2) $\Delta^{i-1} = \Delta_L^{i-1}||\Delta_R^{i-1}$ but it is not possible to push more this approach for the right part (see Equation 7.4). From this observations, they defines a new data format to give as input to Neural Distinguisher:

$$(\Delta_L^i, \Delta_R^i, C_L, C_R, C'_L, C'_R, \Delta_R^{i-1}, \tilde{\Delta}_R^{i-2})$$

where i denotes the round. Furhtermore, they define a multi-pair neural distinguisher that takes in input s pairs defined as before. Let

$$\begin{aligned} A_1 &= (\Delta_L^i, \Delta_R^i, C_L, C_R, C'_L, C'_R, \Delta_R^{i-1}, \tilde{\Delta}_R^{i-2})^1 \\ A_2 &= (\Delta_L^i, \Delta_R^i, C_L, C_R, C'_L, C'_R, \Delta_R^{i-1}, \tilde{\Delta}_R^{i-2})^2 \\ &\vdots \\ A_s &= (\Delta_L^i, \Delta_R^i, C_L, C_R, C'_L, C'_R, \Delta_R^{i-1}, \tilde{\Delta}_R^{i-2})^s \end{aligned}$$

be the 8 pairs where i is the number of rounds and $E_K(C_i) = P_i$, then:

$$\mathcal{ND}_{LLS}(A_1, \ldots, A_s) = \begin{cases} 1 & \text{if } E_K^{-1}(C) \oplus E_K^{-1}(C') = \Delta \text{ with } (C, C') \in A_j \ \forall j \\ 0 & \text{otherwise} \end{cases}$$

In other words, if all the pairs of ciphertexts are actually encrypted with the same key from an input pair of plaintexts with fixed input difference.

Basic Training. [LLS+24] relied on specific input differences for the training. In particular, to train by 9 to 11 rounds of SIMON 32/64, they use the input difference

```
0x00000040
```

For the 12-round distinguisher, the best 10-round distinguisher is retrained on the input difference

`0x04400100`

Then, the best network of the first stage is retrained with the input difference

`0x00000040`.

The authors adopt a similar pipeline for SIMECK 32/64, in order to help the training process in the curriculum learning setting (see subsection 5.3.4).

They construct the dataset based on the above steps and set $s = 8$. In the basic training process, the size of the training set is $2 \cdot 10^7$, and the test set is $2 \cdot 10^6$. Meanwhile, there is an independent key used for each sample. Therefore, the training set has $2 \cdot 10^7$ corresponding random keys, and the test set has $2 \cdot 10^6$ corresponding random keys.

The analysis of this Neural Network is beyond the scope of this thesis but for more reading [LLS+24]. The results of this paper are further discuss later in comparison with our results section 7.2.

6.6 Other results in Neural Cryptanalysis [GLN22]

A sample from the encryption distribution is of the form

$$(c_1, c_2) = (E(k, p), E(k, p \oplus \Delta))$$

for $k \in \{0,1\}^{n_k}$ and $p \in \{0,1\}^{n_p}$ drawn independently and uniformly at random, which has to be distinguished from the two ciphertexts being drawn independently and uniformly at random. If $E(k, .)$ is bijective, choosing p uniform at random is the same as choosing c_1 or c_2 uniform at random. Hence, looking only at c_1 (or only at c_2) there is no difference to the uniform distribution. So,a distinguisher needs to look at (parts of) c_1 and (parts of) c_2 at once in order to distinguish the encryption distribution from the uniform one. Knowing that c_1 and c_2 are uniform distributed also tells us that the marginal distribution of each bit is a Bernoulli-distribution with probability $\frac{1}{2}$ for it to be 1 (or 0). Obviously, adding independent key bits from a key chosen uniform at random removes any potential dependency between bits.

Lemma 6.3 ([GLN22])**.** *Let $X_1, \ldots, X_m, Y_1, \ldots, Y_l$ and $K_1, \ldots, K_l$ be Bernoulli-distributed random variables. Let further (for every i) K_i be 1 with probability $\frac{1}{2}$ and independent of $X_1, \ldots, X_m, Y_1, \ldots, Y_l$ and $K_1, \ldots, K_{i1}, K_{i+1}, K_l$. Then*

$$\mathbb{P}[X_i = x_i \forall i, Y_j = y_j \oplus K_j \forall j] = 2^l \mathbb{P}[X_i = x_i \forall i]$$

By interpreting the X_i as bits from c_1 and Y_i as bits from c_2 this means that if we add independent key bits (more precisely key bits that are independent of the bits from c_1 and c_2 we are looking at) to c_2 we would get

$$\mathbb{P}[X_1 = x_1, \ldots, X_m = x_m, Y_1 = y_1 \oplus K_1, \ldots Y_l = y_l \oplus K_l] = \\ 2^l \mathbb{P}[X_1 = x_1, \ldots, X_m = x_m] = 2^{-m-l}$$

i. e. the bits would look uniform distributed. But there is also a different interpretation. If we divide the bits from c_1 and c_2 that we are looking at in a group of bits that get added by the same key bits (or no key bits at all), i.e. we have dependencies between the key bits that get added to the bits, and interpret them as $X_1, \ldots, X_m$ (after key addition), and interpret the rest of the bits, i.e. the ones where we add independent key bits, as $Y_1 \oplus K_1, \ldots, Y_l \oplus K_l$, we can see that we only need to consider the group corresponding to the dependent key bits.

While it is clear that it is possible to calculate the DDT knowing the probability distribution of (c_1, c_2), it is interesting to note that the reverse is also true under some conditions.

Lemma 6.4 ([GLN22])**.** *Let $X_1, \ldots, X_m$ and $Y_1, \ldots, Y_m$ be Bernoulli-distributed random variables and $(z_1, \ldots, z_m)$, $(w_1, \cdots, w_m)$ as well as $(\delta_1, \ldots, \delta_m)$ be elements of $\{0,1\}^m$. Let further*

$$\mathbb{P}[V_j = v_j | X_i = v_i, Y_i = v_i \oplus \delta_i \forall i \neq j] = \frac{1}{2} \tag{6.9}$$

hold for all $(v_1, \ldots, v_m) \in \{0,1\}^m$ and all $V_j \in \{X_j, Y_j\}$. Then we have that:

$$\mathbb{P}[X_i = z_i, Y_i = z_i \oplus \delta_i \forall i] = \mathbb{P}[X_i = w_i, Y_i = w_i \oplus \delta_i \forall i]$$

i. e. the probability is independent of the concrete choice of $z_1, \ldots, z_m$.

For us this means that as long as the hypothesis of this Lemma holds (Eq. 6.9) the probability distribution does only depend on the difference between c_1 and c_2 and can therefore be calculated using the DDT. This can easily be seen by expressing the probability of a difference as the sum of every possible bit configuration

$$\mathbb{P}[X_i = Y_i \oplus \delta_i \forall i] = \sum_{(z_i, \ldots, z_m)} \mathbb{P}[X_i = z_i, Y_i = z_i \oplus \delta_i \forall i]$$

Since the lemma tells us that all configurations have the same probability, calculating them from the probability of the difference is just a matter of dividing by the number of possible configurations. Hence, if we want to see whether it could be possible to learn additional features aside from the ones provided by the DDT, we can check if Eq. 6.9

does not hold. For example, let us assume we have a cipher with round function of the form $R : \{0,1\}^n \to \{0,1\}^n \to \{0,1\}^n$,

$$R(k, x) = f(x) \oplus k$$

for some $f : \{0,1\}^n \to \{0,1\}^n$ and independent round key k. Then we know from Lemma 6.4 that Eq. 6.9 holds. Therefore, we cannot expect to learn additional information from the distribution of ciphertext-pairs than we could already learn from the DDT. More general, [GLN22] has shown the following.

Theorem 6.5 ([GLN22])**.** *If for a (bijective) cipher there exist a transformation of the ciphertexts independent of the key such that the transformed ciphertexts have the from $c' \oplus k'$, where k are independent key bits, then the encryption distribution and the ciphertext-difference distribution (where the difference is calculated after applying the key independent transformation) contain the same information.*

Hence, in order to show that the encryption distribution does not provide any additional information to the DDT we have to show that we can represent ciphertexts as $c' \oplus k'$ for independent key bits k'.

6.6.1 Create a Multi-Pair from a Single-Pair Distinguisher

One current trend in the vicinity of machine learning assisted cryptanalysis is the use of multiple ciphertext-pairs per sample. While this could be a valid approach to improve upon using single pairs due to possible dependencies between the pairs, and especially could help to surpass classical distingiushers even for ciphers where this is not possible using single pairs [GLN22], current literature fails to put the results into the right perspective, which we would like to chance. For this, it is important to note that differential-neural distinguishers, like the one from [Goh19], actually produces a score between 0 and 1 for each sample, representing its confidence for the sample to belong to the encryption class (instead of the random class). To evaluate the accuracy, we reduce this confidence into a hard decision by simply comparing it to 0.5, i.e. we loose information. But if we simply interpret each score as the probability of the sample to belong to the encryption class, and if we assume independence, we can actually make use of the whole score.

Corollary 6.6 ([GLN22])**.** *Let us assume that we have m samples, each drawn from the same distribution (encryption or random). Let us denote the probability that the i-th samples is an encryption samples as p_i. If the probabilities $p_1, \ldots, p_m$ are independent, then the probability that all m samples are drawn from the encryption distribution is:*

$$\frac{1}{1 + \prod_{i=1}^{m} \frac{1-p_i}{p_i}} \tag{6.10}$$

Proof. The problem we are looking at here is constructing a multi-pair distinguisher from a single-pair distinguisher. We make the simplifying assumption that our single-pair distinguisher is perfectly calibrated, i.e. that we can interpret the distinguisher outputs as probabilities for the observed pair to be real, where a prior probability of $\frac{1}{2}$ is assumed for both cases, meaning that we test samples that are a priori equally likely to be from the random and the real distribution (see Equation 6.3).

We also assume that these distinguisher outputs can be treated as independent. In this context, this is essentially another perfect-calibration assumption: given m ciphertext pairs c_i that are each individually real/random and which have the pairwise distinguisher scores

$$\mathcal{NN}(c_i) = p_i$$

We can build a distinguisher that assigns a probability to each sequence of real/random labels for these samples by multiplying the appropriate combination of p_i and $1 - p_i$ values as needed. "Independence" of scores then can be defined to mean that this distinguisher for these multi-labels is perfectly calibrated, at least as long as there is no repetition of samples.

Now when we build a multi-pair distinguisher, we know that the samples in a set of samples that we are looking at are either all real or all random. It does this by basically computing the probability of

$$P_{real} = \mathbb{P}(c_i \text{ is a real pair } \forall i) \quad \text{and} \quad P_{rand} = \mathbb{P}(c_i \text{ is a random pair } \forall i)$$

Furthemore, it is possible to notice that:

$$P_{real} = \prod_{i=1}^{m} p_i \quad \text{and} \quad P_{random} = \prod_{i=1}^{m} (1 - p_i)$$

Then, from the definition of conditional probability, in this case the probability of all observed samples being real under the condition that either all samples are real or they are random (as is the case when we use a multi-pair distinguisher on a tuple of ciphertext pairs), and the fact that the denominator in this case is two disjoint events, and hence just the sum of those two probabilities, we compute the result as:

$$\frac{P_{real}}{P_{real} + P_{random}} = \frac{1}{1 + \frac{P_{rand}}{P_{real}}} = \frac{1}{1 + \prod_{i=1}^{m} \frac{1-p_i}{p_i}}$$

□

Note that a perfectly calibrated distinguisher is not necessarily anywhere near optimal. So there is no guarantee of optimality for the combined-response construction (even if optimal pairwise distinguishers are used), and no requirement of optimality on the single-pair base distinguishers. In practice, the neural distinguishers we use, isn't

perfectly calibrated, but quite well calibrated, and the combined-response construction performs fairly well if good pairwise distinguishers are used. Corollary 6.6 then gives a way to combine the predictions of the single-pair distinguishers into one score for the full tuple that is still perfectly calibrated, under the assumptions mentioned above

Note that this was already implicitly used in [Goh19], as a ranking of m ciphertext-pairs based on this combined probability is the same as one by Equation 6.7, which is how the keys are ranked in [Goh19] when using multiple ciphertext-pairs during trial decryption. In other words, [Goh19] already used multi-pair distinguishers, even if only implicitly. Using this rule of combining probabilities/distinguisher responses, we are able to explicitly turn a distinguisher for one ciphertext-pair into one for an arbitrary amount of ciphertext-pairs.

A comparison with results from the literature shows that the claimed benefits of multi-pair distinguishers are largely absent. In particular, the multi-pair approach of [CSYY21] performs worse than their single-pair distinguishers, while [ZWw22] achieve only limited improvements, smaller than reported and restricted to specific cases such as 7–9 rounds of Speck 32/64 and 12 rounds of Simon 32/64. Although our distinguishers were not available for comparison, evaluations against published methods such as [Goh19], or against the authors' own single-pair distinguishers, would have been desirable.

Benamira et al. [BGPT21] similarly claim improvements over [Goh19] by combining multiple single-pair scores, but this overlooks the fact that such score combinations are already inherent in the key-recovery attacks of [Goh19], resulting in no actual improvement. While training neural distinguishers with multiple ciphertext pairs may help reveal weak distinguishers that are otherwise hidden by evaluation noise, as illustrated by our results for 66-round KATAN [DCDK09], fair comparisons are essential. In particular, the number of plaintext–ciphertext pairs per sample should be kept constant, since increasing the number of samples only improves accuracy estimation and does not affect the intrinsic distinguishing power.

Chapter 7

Generic Partial Decryption as Feature Engineering for Neural Distinguishers

In the previous chapter 6, we introduced the fundamental tools of modern symmetric cryptanalysis and progressively moved from classical differential techniques to learning-based distinguishers built with deep neural networks. In particular, we studied how differential properties can be exploited to construct effective input representations for neural distinguishers, and we showed that an appropriate choice of features is often as critical as the network architecture itself. These results highlighted that the success of neural cryptanalysis does not solely depend on the learning model, but also on how cryptanalytic structure and prior knowledge are embedded into the data presented to the network.

Within this context, several recent works have suggested that partially decrypting ciphertexts before feeding them to a neural distinguisher can significantly improve its performance [LLS+24, BGPT21]. Such approaches intuitively aim at "peeling off" a small number of rounds in order to amplify exploitable statistical biases or to better align the data with the underlying differential behavior of the cipher. However, these techniques are typically introduced in a heuristic and cipher-specific manner, and a systematic treatment of partial decryption as a general feature-engineering mechanism is still missing.

This chapter builds on the cryptanalytic framework and neural methodologies devel-

oped earlier in this thesis and addresses this gap. We focus on a concrete methodological question: how to systematically exploit partial decryption as a form of feature engineering for neural distinguishers.

We introduce the notion of *generic partial decryption* and show how it can be used to transform raw ciphertexts into representations that better expose round-dependent statistical biases. From this perspective, partial decryption is not merely a cipher-specific preprocessing trick, but a general tool that can be integrated into modern neural pipelines to enhance learning efficiency and distinguishing power. This abstraction allows us to place several previously proposed heuristics within a common conceptual framework and to reason about their design trade-offs in a principled way.

The remainder of the chapter is organized as follows.

1. We introduce the automated Generic Partial Decryption pipeline (section 7.1), and obtain competitive neural distinguishers for Simon, Aradi, and Simeck. Our multi-pair neural distinguishers cover 12 rounds for both Simon and Simeck with an accuracy of 0.5156, slightly surpassing the state-of-the-art results reported in [LLS+24]. Generic Partial Decryption significantly improves the 5-round accuracy for Aradi, increasing it from approximately 60% to 77% in the single-pair setting, and 100% when in the multi-pair setting.
2. We enhance the explainability of previous results achieved using partial decryption by comparing the outcomes across four configurations: *i)* using a simple ciphertext pair as a sample, *ii)* incorporating multiple ciphertext pairs per sample, *iii)* increasing the amount of training data, and *iv)* applying partial decryption. This approach enables a clear differentiation of the individual contributions to the overall accuracy of the neural distinguisher. The results are reported in section 7.2.
3. Using our 4-rounds neural distinguishers, we present the first practical neural key recovery attack on 5 rounds of Aradi, with data complexity 2^9 (for 75% success rate), and time complexity dominated by 2^{40} neural distinguisher evaluations. We extend this attack to 7 rounds, albeit with non-practical time complexity. More importantly, for the first time, we leverage information learned from the neural distinguisher to build a classical key recovery attack. For 5 rounds, this attack has data complexity comparable to the neural one, and time complexity 2^{42}. These attacks are discussed in section 7.3.

7.1 Design of the Generic Partial Decryption Pipeline

To design the Generic Partial Decryption pipeline, presented in [BBG+25], we begin by outlining the strategy for generic partial decryption in subsection 7.1.1. This strategy enables the derivation of various potential data formats for the neural distinguisher.

From these, we select four data formats for detailed investigation, as described in subsection 7.1.2. After determining the input data format, the next critical decisions involve selecting the input difference (subsection 7.1.3) and choosing the appropriate neural network architecture (subsection 7.1.4). Furthermore, we aim to simplify the complex training pipelines commonly employed in prior works by reducing the amount of training data required, leading to our streamlined training pipeline design (subsection 7.1.5). To address concerns regarding the use of multiple pairs, we choose to train single-pair distinguishers, which can also be evaluated on multiple pairs for comparison with existing literature, as detailed in subsection 7.1.6.

7.1.1 Our Strategy for Generic Partial Decryption

The specific features utilized by the $\mathcal{ND}$ to make predictions remain largely unknown. It has been established that these features are mostly differential in nature, but a small non-differential component is shown in [Goh19], and the presence of differential-linear features has been suggested in [BGPT21]. Various improvements, such as designing new $\mathcal{ND}$ architectures or altering the input data format, have been proposed to enhance $\mathcal{ND}$ performance. However, these enhancements are typically tailored to specific ciphers or those sharing particular characteristics, complicating the understanding of how and why the $\mathcal{ND}$ models the cipher characteristics effectively. In [Goh19], the key recovery strategy uses the response profile of the $\mathcal{ND}$ when the last round is decrypted with an incorrect key. In [BGPT21], the authors use a similar technique to build equivalent classical distinguishers. In addition, they observe that the right part of the Speck 32 state at the previous round can be computed without knowing the key and propose to use it as an additional feature for the neural distinguisher. In [BGlL$^+$22], the authors extend that notion to the Simon 32/64 cipher family and compare different types of related feature engineering. The concept of Partial Decryption was introduced in [LLS$^+$24], where the authors investigated whether augmenting input data with additional features could enhance $\mathcal{ND}$ accuracy. They also designed $\mathcal{ND}$ architectures specifically suited for targeted tasks. According to [LLS$^+$24], their approach improved accuracy and opened avenues for further extensions. (see subsection 6.5.1).

Remark. *Partial Decryption cannot be universally applied to all symmetric ciphers due to variations in their structural properties and key scheduling algorithms. For ciphers like Simeck, where the differential propagation is relatively straightforward and key-dependent differentials can be deterministically derived by setting specific portions of the key to a fixed value (e.g., zero), partial decryption becomes feasible. However, many ciphers employ more complex key schedules or nonlinear operations, such as substitution-permutation networks (SPNs) or Feistel structures with intricate mixing layers, which obscure the differential relationships across rounds. Additionally, certain ciphers rely on strong diffusion properties and nonlinearity to prevent deterministic recovery of intermediate states. These design choices, aimed at enhancing cryptographic strength, inherently limit the practicality of partial decryption techniques, as they reduce*

the availability of predictable patterns in the cipher's behavior that could be exploited.

Our primary goal is to develop a general understanding of the features captured by the $\mathcal{ND}$ to enhance interpretability. Building on the work in [LLS+24], analyzing the differentials in earlier rounds could yield better input features for training $\mathcal{ND}$ models. However, if partial decryption is limited to the specific cipher under analysis, it demands significant manual effort from the cryptanalyst, thereby reducing the suitability of the $\mathcal{ND}$ as a generic cryptanalytic tool. The propagation of a differential through a cipher typically gains more entropy at each round until full diffusion is achieved. Therefore, the difference at round $i-1$ normally holds more useful information than the difference at round i for a distinguisher. Given knowledge of the round function of a cipher, it is often possible to derive partial information about the difference in the previous rounds. For Feistel-like ciphers such as Simon and Speck, part of the previous round state can be directly obtained. For SPNs, such as Aradi, truncated differential information on the previous round can be recovered (in particular, the activity pattern of the S-Boxes). In addition, high probability differential transitions, by definition, hold for a large portion of the key space so that partial decryption with an incorrect key has a high chance of preserving them. While such information is trivial for the cryptographer to obtain, it would be unreasonable to expect the neural distinguisher to learn it on its own. For instance, in the case of Simon, learning the round function to retrieve the previous round difference appears difficult for a neural distinguisher. By construction of Feistel-like ciphers, it is trivial for a cryptographer to retrieve information on the previous round difference; on the other hand, if the round function is sufficiently complex, it becomes arbitrarily hard for a neural network with black-box access to do so. A cryptographer's natural approach would be to observe which parts of the previous rounds' differences can be recovered with a high probability and use them only. This is also the approach taken by [LLS+24]. On the other hand, following the goal to limit human input to the mere specification of the cipher, we would like to derive such information automatically in a generic fashion. More specifically, we provide partial decryption with a random key as a feature and let the neural distinguisher find the relevant parts and discard the less relevant ones. Without loss of generality, we use the subkey 0 in our experiments rather than a random one; intuitively, the exact choice has no impact as long as the expected Hamming distance with the actual key is $\frac{n}{2}$. The number of rounds for which such partial decryption holds information varies depending on the cipher under consideration; for instance, in [GLN22], the authors use information from many previous rounds in their analysis of KATAN. In this paper, for the sake of confirming the benefit of partial decryption as a feature, we restrict ourselves to 2 rounds of partial decryption. This captures the relevant information for most block ciphers while keeping the size of the dataset and experimental time reasonable. We expect our conclusions to hold when including the partially decrypted difference in all rounds, but it may scale poorly for specific examples such as KATAN, where dozens of rounds would make the sample size unpractical.

We extend the concept of *Partial Decryption*, originally tailored to a specific cryptographic primitive, to the broader framework of *Generic Partial Decryption*. This approach is formalized as follows:

Definition 7.1 (Encryption and Decryption Function)**.** Let $\mathcal{P}$ the plaintext set, $\mathcal{C}$ the ciphertext set, $\mathcal{K}$ the key space, f the encryption round function and g the decryption round function such that:

$$f : \mathcal{P} \times \mathcal{K} \to \mathcal{C}, \quad f_k^i(C^{i-1}) := f(k_i, C^{i-1}) = C^i$$
$$g : \mathcal{C} \times \mathcal{K} \to \mathcal{P}, \quad g_k^i(C^i) := g(k_i, C^i) = C^{i-1}.$$

where k_i is the i-th round key, derived from the master key $k \in \mathcal{K}$.
Then, we can define the *encryption function* and the *decryption function* as:

$$E_k^n(C^0) = f_k^{n-1} \circ \cdots \circ f_k^0(C^0) = C^n, \quad D_k^n(C^n) = g_k^0 \circ \cdots \circ g_k^{n-1}(C^n) = C^0,$$

such that:

$$f_k^i \circ g_k^i(C) = C \quad \text{and} \quad g_k^i \circ f_k^i(P) = P \quad \text{with } i = 1, \ldots, n.$$

Definition 7.2 (k-Partial Differential)**.** Let $(C^i, (C')^i) = (E_k^i(P), E_k^i(P'))$ be a pair of the encryption function outputs after i rounds. Then, the *differential* at the i-th round is:

$$\Delta^i = E_k^i(P) \oplus E_k^i(P') = C^i \oplus (C')^i.$$

We call the **1-Partial Differential** at round $i-1$ and the **2-Partial Differential** at round $i-2$:

$$\tilde{\Delta}^{i-1} = g(0, C^i) \oplus g(0, (C')^i), \quad \tilde{\Delta}^{i-2} = g\left(0, g(0, C^i)\right) \oplus g\left(0, g(0, (C')^i)\right) \tag{7.1}$$

where 0 is the round key (i.e. $k_i = 0$). More in general, the **k-Partial Differential** is:

$$\tilde{\Delta}^{i-k} = \underbrace{g(0, g(0, \ldots, g(0, C^i))}_{k-times} \oplus \underbrace{g(0, g(0, \ldots, g(0, (C')^i))}_{k-times}.$$

In the Generic Partial Decryption technique, the ciphertexts are decrypted with round key 0. In contrast to the approach in [LLS+24], we do not select which parts of the resulting words are kept. For the remainder of this paper, we focus exclusively on the application of $\tilde{\Delta}^{i-1}$ and $\tilde{\Delta}^{i-2}$.

7.1.2 Choice of the Data Format

Based on our Generic Partial Decryption strategy, several choices of data formats are possible. We focus on three specific formats:

Given an input difference δ and a pair of plaintexts $(P, P \oplus \delta)$, we denote ciphertext pairs encrypted with the same key as $(C, C') = (E_k^r(P), E_k^r(P \oplus \delta))$ for some $k \in \mathcal{K}$.

Simple Pair (SP). The first data format, SP, is the most commonly used in the literature; It provides a baseline to evaluate whether adding additional features can improve training:

$$(C^r, (C')^r).$$

Last Differential, Simple Pair, 1-Partial Differential and 2-Partial Differential (LD, SP, 1PD, 2PD). The second format, (LD, SP, 1PD, 2PD), is closely related to options recently investigated in [LLS+24] and incorporates both ciphertext pairs and partial differentials:

$$(\Delta^r, C^r, (C')^r, \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2}).$$

Last Differential, 1-Partial Differential and 2-Partial Differential (LD, 1PD, 2PD). The third format (LD, 1PD, 2PD) is unique in that it does not utilize ciphertext pairs, relying solely on differential values. This format serves to evaluate the influence of non-differential features on the accuracy of the $\mathcal{ND}$:

$$(\Delta^r, \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2}).$$

The first of our data formats, SP, is the most commonly adopted in the literature, and we need it to understand if adding features could help the training or not. The second format (LD, SP, 1PD, 2PD) is close to recently investigated options [LLS+24]. It is commonly assumed, following the conclusions of [Goh19], that the ciphertexts themselves are needed to reach optimal accuracy; our last format (LD, 1PD, 2PD) aims at testing this hypothesis.

7.1.3 Choice of Input Differences

The choice of the input difference used in a neural distinguisher has been shown to have a significant impact on the number of distinguished rounds and the accuracy [BGPT21]. A good input difference for a neural distinguisher is not necessarily the input difference for the best classical differential trail. It suffices to observe that differential trails for a few rounds have relatively low probability (*e.g.,* 2^{-9} for 5 rounds of Speck), whereas neural distinguishers distinguish based on a single pair. Therefore, the property that they identify has to occur with significantly higher probability than a classical trail, and properties such as multiple differentials and differential-linear play an important role.

It is commonplace in neural distinguishers research to pick an input difference with a low Hamming Weight, as it is expected to propagate slowly and enable longer neural distinguishers. These input differences are often chosen from intermediate rounds of known trails for the cipher under study, as is the case in [LLS+24] (see subsection 6.5.1). The choice of the input differential is crucial for the training of $\mathcal{ND}$. These choices come from the literature, and they are specific for those ciphers. This inevitably makes the whole pipeline much more elaborate and difficult to generalize. On the other hand, the AutoND pipeline, introduced at FSE 2024, the authors of [BGH+23] introduced an evolutionary optimizer (see Algorithm 17) for identifying suitable input differences for neural distinguishers in a generic manner (see subsection 6.4.1).

Given this approach, the existing evolutionary optimizer appears well-suited for application to partially decrypted input data formats.

7.1.4 Choice of the Neural Network Architecture

For a comprehensive overview of the neural network architectures for cryptanalysis, we refer readers to the recent systematization of knowledge [GHHP24]. To motivate our choice of neural network architecture, we provide the following summary: Multilayer perceptrons (MLPs) have been widely explored as a basic yet lightweight architecture, for example in [BR21, YK21, ZZY+21, BBCD22, ERP22]. While MLPs are efficient and easy to implement, their performance is often surpassed by more advanced architectures. The majority of subsequent works build upon Gohr's original neural network design, which has been adapted and refined in various studies, among others [BGPT21, HRCF21, SZM21, TH21, WW21, BBCD22, LCLH22, LTZ22, WTZ+22]. Although Gohr's network can be adapted to new ciphers, this process necessitates careful tuning of numerous hyperparameters [GLN22]. Advanced components, such as attention mechanisms [DCC23], LSTMs [BBCD22, SSL+22], GoogLeNet-inspired Inception modules [ZWC23, ZLWL23, BLYZ23], and DenseNets [SM23], have also been explored. While these approaches demonstrate strong results for specific ciphers, they are often highly cipher-specific and are accompanied by significantly increased training times.

Given our goal of developing generic applicability, we focus on two specific architectures: DBitNet, which emphasizes generalization across ciphers, and GohrAMS, a variant of Gohr's network that incorporates AMSGRAD learning rate adaptation also introduced in [BGH+23].

The Neural Networks we consider are listed in the followin paragraphs.

GohrAMS. The Neural Network proposed in [Goh19] is defined in subsection 6.2.1. We use the AMSGRAD optimizer [RKK19] instead of the original learning rate scheduler, as it was shown in [BGH+23] to achieve equivalent performance without the need

for manual fine-tuning of the original cyclic learning rate schedule, see Figure 6.3.

DBitNet. DBitNet, Figure 6.4, introduced in [BGH+23], leverages dilated convolutions to capture both short- and long-range dependencies within cipher inputs, avoiding the need for explicit input reshaping. The network employs a sequence of convolutional layers with varying dilation rates. As the neuronal width decreases with each dilated step, the number of filters increases incrementally to compensate for the reduced spatial dimensions. This design ensures a balance between dimensionality reduction and feature extraction, enabling the network to achieve effective representation and generalization across a variety of ciphers (see subsection 6.4.2).

Table 7.1 Comparison of hyperparameters between DBitNet and GohrAMS.

Hyperparameter	DBitNet	GohrAMS
Input size	n	n
Initial filters	32	32
Kernel size	2	3
Dilation rate	Variable according Formula (Equation 6.8)	Not used
Residual blocks depth	Based on number of dilation rates	1
Regularization	10^{-5}	10^{-5}
Dense layers (head)	$d_1 = 256$, $d_2 = 64$	$d_1 = 64$, $d_2 = 64$
Final activation	Sigmoid	Sigmoid
Optimizer	AMSGRAD[a]	AMSGRAD[a]

[a] [RKK19].

7.1.5 Choice of the Training Pipeline for AutoND [BGH+23]

The training pipeline is known to be determinant when targeting higher rounds, for which the signal of the distinguisher becomes weaker. In [Goh19], the 8-round distinguisher was obtained by retraining the best distinguishers starting from likely differences in the middle rounds and increasing the amount of training data by a factor of 100 (see subsection 6.2.1). Similarly, in [LLS+24], the 12-round Simon 32/64 distinguisher is built by iteratively retraining the best distinguishers on curated data, with a dedicated cyclic learning rate (see subsection 6.5.1. The authors adopt a similar pipeline for Simeck 32/64. They aim to achieve similar accuracies without using such dedicated techniques using the description in subsection 6.4.3.

7.1.6 Choice of Single- or Multi-Pair Distinguisher

The [LLS+24]'s $\mathcal{ND}$s are multi-pair Distinguishers trained with samples composed of 8 pairs, increasing the amount of data required for training and testing (see subsec-

tion 6.5.1). On the other hand, [GLN22] proposes a method to create a multi-pair distinguisher from a single-pair distinguisher (see 6.6.1). The algorithm, takes as input m ciphertext pairs, either all real or all random, but otherwise assumed to be independent. The $\mathcal{ND}$ is assumed to be perfectly calibrated so that the pairwise scores $\mathcal{NN}(c_i) = p_i$ can be interpreted as probabilities. Under these assumptions, the Multi-Pair distinguisher derives the probabilities aggregates them as the combined response score in equation 6.10.

In [BBG+25], we follow subsection 6.6.1:

1. We validate our $\mathcal{ND}$ for each format on 10^6 pairs; these datasets are denoted $(X_j)_j$.
2. We split the dataset according the labels; (X_j^1) are the real samples and (X_j^0) the random ones. Let $N^t = \#(X_j^t)$, be the cardinality of the set for $t \in \{0, 1\}$.
3. We obtain the respective scores of the samples as $p_j^t = \mathcal{NN}(X_j^t)$ with $t \in \{0, 1\}$.
4. We divide the corresponding (p_j^t) into m subgroups and apply Equation 6.10. We obtain in total $M^t = \lfloor N^t/m \rfloor$ prediction for $t \in \{0, 1\}$.
5. Each M^t subsample is classified based on its combined response score.
6. The accuracy is given from the ratio of the right-labeled subsamples over the total subsamples we have.

7.1.7 Final Pipeline for Generic Partial Decryption

Based on the previously presented considerations, we arrive at the following pipeline for generic partial decryption.

The main algorithm, see Algorithm 20, trains a neural distinguisher $\mathcal{ND}$ to identify the highest non-random round of a block cipher implementation $E_k(\cdot)$. It adopts a curriculum learning approach, beginning from an initial round r_0 and progressively training the $\mathcal{ND}$ on datasets generated for increasing rounds. The training process continues until the validation accuracy drops below a predefined threshold (defined as before subsection 7.1.5, ten standard deviations, 10σ, above the random accuracy of fifty percent). We are using again $N_{\text{val}} = 10^6$ resulting in $\sigma = 0.0005$. Datasets for each round are created using the chosen data format in the `GeneratePDDataset` algorithm that we have discussed in subsection 7.1.2, see Algorithm 21.

The `GeneratePDDataset` sub-algorithm constructs a labeled dataset $\mathcal{D}$ for a given round r. It generates N samples through the following steps:

(i) Randomly select plaintexts $P \in \mathcal{P}$ and keys $k \in \mathcal{K}$.
(ii) Assign a random label $y \in \{0, 1\}$.
(iii) Compute ciphertext pairs (C, C') using the r-round encryption function $E_k^r(\cdot)$ on $(P, P \oplus \delta_0)$.

Algorithm 20 `Generic_Partial_Decryption`.

Require: Oracle access to an r-round block cipher $E_k(\cdot)$, where $P \in \mathcal{P}$, $k \in \mathcal{K}$, and $C \in \mathcal{C}$ such that $C = E_k(P)$, dataformat according one of subsection 7.1.2 `data_format`

$\mathcal{ND} \leftarrow$ randomly initialize the neural network (cf. subsection 7.1.4)
$(R, \delta) \leftarrow$ `evolutionary_optmizer`
Select starting round $r_0 < R$ and input difference δ
▷ Execute curriculum learning starting from round r_0 (cf. subsection 7.1.5)
$A_{\text{val}} \leftarrow 1.0$
$r \leftarrow r_0$
while $A_{\text{val}} > 0.50 + 10\sigma \wedge r < R$ **do**
    $\mathcal{D}_{\text{train}} \leftarrow$ `GeneratePDDataset`$(r, N_{\text{train}}, \delta,$ `data_format`$)$
    $\mathcal{D}_{\text{val}} \leftarrow$ `GeneratePDDataset`$(r, N_{\text{val}}, \delta,$ `data_format`$)$
    Train $\mathcal{ND}$ on $\mathcal{D}_{\text{train}}$
    $A_{\text{val}} \leftarrow$ accuracy of $\mathcal{ND}$ on $\mathcal{D}_{\text{val}}$ ▷ Evaluate neural distinguisher
    $r \leftarrow r + 1$ ▷ Progress to next round r
end while
return $\mathcal{ND}$

Algorithm 21 `Generic_Partial_Decryption_Dataset`.

Require: Round r, number of samples N, and input difference δ, dataformat according subsection 7.1.2 `data_format`.

$\mathcal{D} \leftarrow \emptyset$
for $j = 1, \ldots, N$ **do**
    $P, k \leftarrow$ random choice of $P \in \mathcal{P}$, $k \in \mathcal{K}$
    $y \leftarrow$ random choice from $\{0, 1\}$ ▷ Randomly choose a label
    $(C, C') \leftarrow (E_k^r(P), E_k^r(P \oplus \delta))$ ▷ Generate ciphertext pair
    $(C, C') \leftarrow$ random pair of bit strings if $y = 0$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{$bit vector according to `dataformat`$\}$ ▷ cf. subsection 7.1.1
end for
return $\mathcal{D}$

(iv) C, C' becomes a random pair of bit string if $y = 0$.

The dataset $\mathcal{D}$ consists of labeled feature vectors according to the chosen data format, which is subsequently used to train and validate the neural distinguisher in the main algorithm. Our parameter choices follow the ones of [BGH+23]: All our $\mathcal{ND}$s are

trained over 40 epochs per round using $N_{\text{train}} = 10^7$ samples, with a batch size of 5000, and validated on $N_{\text{val}} = 10^6$ samples. Therefore, each of the five fresh test sets contains 10^6 samples drawn with equal class priors; the expected imbalance per set is $\leq 0.05\%$. Averaging accuracy over the five sets reduces any residual bias to $\leq 0.02\%$, making the reported mean robust. Our reporting is consistent with prior works such as [BGH+23] and statistically robust. Additionally, we apply the simple polishing step of [BGH+23], where the final network undergoes retraining for three iterations. Each iteration involves repeating the training for one epoch 100 times on 10^7 fresh training samples, resulting in a total training dataset of $100 \times 10^7 = 10^9$ samples per iteration. Since the additional data we are using, we decide to apply the polish step just for the round attacked and not for every step of the curriculum training. With a batch size of 10,000, we use the AMSGRAD optimizer, gradually decreasing the constant learning rate at each iteration from 10^{-4} to 10^{-5} and finally to 10^{-6}. Validation is performed on 10^7 samples at each iteration. Our single-pair classifiers, with $m = 1$, are extrapolated to multi-pair distinguishers ($m = 8$), using the combined response computed as in subsection 7.1.6, to enable fair comparison with the related work. In both cases, the reported accuracies are the average on five fresh testing datasets of size 10^6.

7.2 Neural Distinguishers via Generic Partial Decryption

In the following, we apply our `Generic_Partial_Decryption` (cf. subsection 7.1.7) to SIMON, SIMECK, ARADI, and SPECK. Summarizing the results discussed in detail below, we find that for SIMON and SIMECK, our generic pipeline achieves competitive accuracies compared to the highly specialized approach of [LLS+24]. Moreover, we can explain the accuracies obtained by [LLS+24] by their distinct contributions: the use of multiple pairs, additional data, and the actual partial decryption. Interestingly, for SPECK, the generic partial decryption did not result in improvements, whereas it produced the most significant enhancements for ARADI.

7.2.1 Results on Simon and Simeck

We perform the training starting from round 8 to round 12 with input difference: `0x400` and `0x80` respectively for SIMON 32/64 (Table 7.2) and SIMECK 32/64 (Table 7.3) (for the choice of the input differences cf. subsection 7.1.3). In each table, we present our results for `Generic_Partial_Decryption` applied to SIMON in the DBITNET-columns alongside the results of [LLS+24], which exploit specific features of the SIMON and SIMECK ciphers.

Simon. For SIMON (Table 7.2), in the simple data format (C, C'), DBITNET distinguishes better than random up to 11 rounds; in the multi-pair setting ($m = 8$), the 11 rounds accuracy increases by 4 points, and the 12 rounds accuracy remains borderline, at 2.6σ above randomness. Applying the simple polishing step (cf. subsection 7.1.7) brings the accuracy to 0.5097 (19σ above random). These results indicate that a 12-round distinguisher for SIMON can be achieved with the simple data format by combining *i)* multi-pair evaluation and *ii)* a simple polishing step with additional training data. The remaining gap with the 12-rounds accuracy 0.5152 from [LLS+24] is filled by feature engineering: DBITNET, combined with the `Generic_Partial_Decryption`, achieves slightly higher accuracy (0.5153) using the same feature engineering technique, and up to 0.5156 under the (LD, 1PD, 2PD) format.

Table 7.2 Comparison of our results for SIMON 32/64 with the state-of-the-art. We build neural distinguishers in the standard format and our Generic Partial Decryption format and evaluate the accuracy on a single ciphertext pair ($m = 1$), as well as on multiple ciphertext pairs ($m = 8$) (c.f. subsection 7.1.6). Results for SIMON 32/64.

	[LLS+24] $(\Delta^r, C, C', \Delta_R^{r-1}, \tilde{\Delta}_R^{r-2})$[a]	**DBitNet** (C, C')			**DBitNet** $(\Delta^r, C, C', \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2})$			**DBitNet** $(\Delta^r, \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2})$		
SIMON32/64 Round	$m = 8$ adv.[†]	$m = 1$ Simple	$m = 8$[b] Simple	$m = 8$[b] Pol.	$m = 1$ Simple	$m = 8$[b] Simple	$m = 8$[b] Pol.	$m = 1$ Simple	$m = 8$[b] Simple	$m = 8$[b] Pol.
8	-	0.8367	0.9991	-	0.8417	0.9993	-	0.8408	0.9993	-
9	0.9176	0.6556	0.8892	-	0.6584	0.8938	-	0.6580	0.8933	-
10	0.6975	0.5626	0.6761	-	0.5655	0.6859	-	0.5646	0.6827	-
11	0.5609	0.5150	0.5479	-	0.5177	0.5567	-	0.5173	0.5561	-
12	**0.5152**	0.5004	0.5013	0.5097	0.5022	0.5077	**0.5153**	0.5023	0.5064	**0.5156**

[†] See subsection 7.1.5 [a] Refer to [LLS+24] [b] The $\mathcal{ND}$ is the same of the case $m = 1$ but we change the evaluation step according subsection 7.1.6.

Simeck. For SIMECK (Table 7.3), we observe similar results, unsurprisingly, due to the similarities of the two primitives. However, the key schedule difference between the two primitives does not seem to impact the accuracy of the $\mathcal{ND}$. More importantly, these results show that, contrary to the common-held belief, the presence of the ciphertexts is not always mandatory to reach optimal accuracy; indeed, our best performing format for Simon and Speck, (LD, 1PD, 2PD), only uses differences.

Observation on key dependencies. We try to exploit the features of the inverse round of SIMON and SIMECK. The inverse round of the encryption function 2.1 is defined as:

$$L^{i-1} = R^i$$
$$R^{i-1} = L^i \oplus [(R^i \lll \alpha) \odot (R^i \lll \beta)] \oplus (R^i \lll \gamma) \oplus k^i$$

Knowing outputs at round i, we can recover a differential at round $i-1$ which does not depend on the round key by considering two pairs (L^i, R^i) and $((L')^i, (R')^i)$ encrypted

Table 7.3 Comparison of our results for SIMECK 32/64 with the state-of-the-art. We build neural distinguishers in the standard format and our Generic Partial Decryption format and evaluate the accuracy on a single ciphertext pair ($m = 1$), as well as on multiple ciphertext pairs ($m = 8$) (c.f. subsection 7.1.6). Results of SIMECK 32/64.

	[LLS+24]	**DBitNet**			**DBitNet**			**DBitNet**		
	$(\Delta^r, C, C', \Delta_R^{r-1}, \tilde{\Delta}_R^{r-2})$[a]	(C, C')			$(\Delta^r, C, C', \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2})$			$(\Delta^r, \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2})$		
SIMECK32/64	$m = 8$	$m = 1$	$m = 8$[b]	$m = 8$[b]	$m = 1$	$m = 8$[b]	$m = 8$[b]	$m = 1$	$m = 8$[b]	$m = 8$[b]
Round	adv.[†]	Simple	Simple	Pol.	Simple	Simple	Pol.	Simple	Simple	Pol.
8	-	0.8470	0.9998	-	0.9022	1.0000	-	0.9025	1.0000	-
9	0.9952	0.6819	0.9276	-	0.7075	0.9562	-	0.7074	0.9564	-
10	0.7354	0.5601	0.6832	-	0.5700	0.7127	-	0.5702	0.7133	-
11	0.5646	0.5133	0.5472	-	0.5168	0.5595	-	0.5170	0.5612	-
12	**0.5146**	0.5007	0.5028	0.5109	0.5026	0.5100	**0.5155**	0.5028	0.5090	**0.5156**

[†] See subsection 7.1.5 [a] Refer to [LLS+24] [b] The $\mathcal{ND}$ is the same of the case $m = 1$ but we change the evaluation step according subsection 7.1.6.

using the same key:

$$\begin{aligned} L^{i-1} &= R^i \\ (L')^{i-1} &= (R')^i \end{aligned} \implies \Delta_L^{i-1} = R^i \oplus (R')^i \tag{7.2}$$

$$\begin{aligned} R^{i-1} &= \underbrace{L^i \oplus [(R^i \lll \alpha) \odot (R^i \lll \beta)] \oplus (R^i \lll \gamma)}_{A} \oplus k^i \\ (R')^{i-1} &= \underbrace{(L')^i \oplus [((R')^i \lll \alpha) \odot ((R')^i \lll \beta)] \oplus ((R')^i \lll \gamma)}_{A'} \oplus k^i \end{aligned} \implies \Delta_R^{i-1} = A \oplus A' \tag{7.3}$$

For what concerns differentials at round $i-2$, we are able to recover the left differential, which does not depend on the round key;

$$\begin{aligned} L^{i-2} &= R^{i-1} = A \oplus k^i \\ (L')^{i-2} &= (R')^{i-1} = A' \oplus k^i \end{aligned} \implies \Delta_L^{i-2} = A \oplus (A') = \Delta_R^{i-1}$$

instead, the right differential depends on the key, in fact:

$$\begin{aligned} R^{i-2} &= L^{i-1} \oplus [(R^{i-1} \lll \alpha) \odot (R^{i-1} \lll \beta)] \oplus (R^{i-1} \lll \gamma) \oplus k^{i-1} \\ &= R^i \oplus [((A \oplus k^i) \lll \alpha) \odot ((A \oplus k^i) \lll \beta)] \oplus ((A \oplus k^i) \lll \gamma) \oplus k^{i-1} \\ (R')^{i-2} &= (L')^{i-1} \oplus [((R')^{i-1} \lll \alpha) \odot ((R')^{i-1} \lll \beta)] \oplus ((R')^{i-1} \lll \gamma) \oplus k^{i-1} \\ &= (R')^i \oplus [((A' \oplus k^i) \lll \alpha) \odot ((A' \oplus k^i) \lll \beta)] \oplus ((A' \oplus k^i) \lll \gamma) \oplus k^{i-1} \end{aligned} \tag{7.4}$$

7.2.2 Results on Aradi

We train the neural distinguisher ($\mathcal{ND}$) for Aradi from round 3 to round 5 using the input difference `0x10000000000000000000`. The distinguishers Table 7.4 reach 100% accuracy on 3 and 4 rounds, while the best differential trail over 4 rounds has probability 2^{-32}. Furthermore, with the input difference we used, the best differential trail has probability 2^{-60} (found using CLAASP [BGG+24]). Another previous work [BFG+24, Table 10] achieves 5-round accuracy of 0.5954. This accuracy is close to our results in the (C, C') scenario. In the multi-sample and partially decrypted scenario, we significantly surpass these results, reaching accuracies of up to almost perfect accuracy in round 5. On 5 rounds, the addition of features derived from partial decryption has a considerable positive impact on the accuracy, with DBitNet going from just under 60% accuracy in the (C, C') format to around 77% with the composite format $(\Delta^r, C, C', \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2})$. GohrAMS reaches a similar performance. In contrast, the best 5-round differential trail has probability 2^{-50}; if the input difference is restricted to the one used by our distinguisher, the best trail has probability 2^{-94}. On the other hand, we can observe deterministic truncated differential properties when the input difference is in a single S-Box position. Namely, by construction of the linear layer, such an input difference propagates to 3 non-zero differences after one round with probability 1. At round 2, the truncated propagation is still deterministic, and 9 S-Box positions are active. At round 3, the activity pattern of 22 S-Boxes is non-deterministic, while 10 S-Boxes have a 0 difference with probability 1. This 40-bit property is sufficient to build a strong distinguisher and can be directly checked from the round 4 difference. Indeed, when inverting the last round, the round 4 key addition does not change the difference, and the linear layer propagates it in a deterministic way. The input difference before and after the inversion of S-boxes depends on the round 4 key, but we are only interested in whether they have a zero difference. For bijective S-Boxes, a zero input difference always goes to a 0 output difference and vice versa so that we can observe which S-Box positions are 0 at round 3. Therefore, giving the partial decryption to the neural distinguisher directly gives it a strong property to observe, which it otherwise would have to learn.

Table 7.4 We build neural distinguishers for Aradi in the standard format and our Generic Partial Decryption format and evaluate the accuracy on a single ciphertext pair ($m = 1$), as well as on multiple ciphertext pairs ($m = 8$) (c.f. subsection 7.1.6). Results of Aradi.

	DBitNet (C, C')			**DBitNet** $(\Delta^r, C, C', \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2})$			**GohrAMS** $(\Delta^r, C, C', \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2})$			**DBitNet** $(\Delta^r, \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2})$		
Aradi Round	$m=1$ Simple	$m=8$[a] Simple	$m=8$[a] Pol.	$m=1$ Simple	$m=8$[a] Simple	$m=8$[a] Pol.	$m=1$ Simple	$m=8$[a] Simple	$m=8$[a] Pol.	$m=1$ Simple	$m=8$[a] Simple	$m=8$[a] Pol.
3	1.0000	1.0000	-	1.0000	1.0000	-	1.0000	1.0000	-	1.0000	1.0000	-
4	1.0000	1.0000	-	1.0000	1.0000	-	1.0000	1.0000	-	1.0000	1.0000	-
5	0.5974	0.7533	0.7634	**0.7732**	**0.9984**	**0.9988**	**0.7733**	**0.9988**	**0.9988**	**0.7721**	**0.9988**	**0.9988**

[a] The $\mathcal{ND}$ is the same of the case $m = 1$ but we change the evaluation step according subsection 7.1.6.

Observation on key dependencies. Recalling that the round function updates the state $(w||x||y||z)$ represented as four 32-bit words through a column-wise 4-bit S-box π followed by a row-wise linear layer Λ_i subsection 2.2.3. To find which part of the round function implies a differential that does not depend on the key, we invert one round. Note that the inverse of the linear map Λ_i is itself (in ARADI $2\alpha_i = \beta_i + \gamma_i \implies (L_i \circ L_i)(s) = s$); in fact, by considering

$$(\tau_{k^{i+1}} \circ \Lambda_i \circ \pi \circ \tau_{k^i})^{-1} = \tau_{k^i} \circ \pi^{-1} \circ \Lambda_i \circ \tau_{k^{i+1}} :$$

$$\begin{aligned}\tilde{s}^{i+1} = L_i((s_l^{i+1} \oplus k_{s,l}^{i+1})||(s_r^{i+1} \oplus k_{s,r}^{i+1})) = \\ \Big((s_l^{i+1} \oplus k_{s,l}^{i+1}) \oplus ((s_l^{i+1} \oplus k_{s,l}^{i+1}) \lll \alpha_i) \oplus ((s_r^{i+1} \oplus k_{s,r}^{i+1}) \lll \gamma_i)|| \\ (s_r^{i+1} \oplus k_{s,r}^{i+1}) \oplus ((s_r^{i+1} \oplus k_{s,r}^{i+1}) \lll \alpha_i) \oplus ((s_l^{i+1} \oplus k_{s,l}^{i+1}) \lll \beta_i)\Big)\end{aligned}$$

where s can be one of the words w, x, y or z and $k_s^{i+1} = k_{s,r}^{i+1}||k_{s,l}^{i+1}$ are the round keys at round $i+1$ associated with the word s. Then, we try to recover the input of round i inverting the non-linear S-Box and the application of $\tau_{k^i}^{-1}$:

$$\begin{aligned} w^i \oplus k_w^i &= \tilde{w}^{i+1} \oplus (\tilde{x}^{i+1} \odot \tilde{z}^{i+1}) \\ y^i \oplus k_y^i &= \tilde{y}^{i+1} \oplus ((w^i \oplus k_w^i) \odot \tilde{z}^{i+1}) \\ z^i \oplus k_z^i &= \tilde{z}^{i+1} \oplus ((y^i \oplus k_y^i) \odot \tilde{x}^{i+1}) \\ x^i \oplus k_x^i &= \tilde{x}^{i+1} \oplus ((y^i \oplus k_y^i) \odot (w^i \oplus k_w^i)) \end{aligned}$$

So, we can conclude that for all the blocks, we still have dependencies from the last key that we cannot avoid with the differential.

7.2.3 Results on Speck

We train the $\mathcal{ND}$ for SPECK 32/64 from round 5 to round 8 with input difference `0x400000`. The results are shown in the 7.5. As shown in that table, we observe the opposite behavior compared to ARADI. Specifically, comparing the simple data format (C, C') to the second and third formats that include partial decryption reveals a drop in accuracy for both the single-pair and multi-pair distinguishers. In [Goh19], show that their $\mathcal{ND}$ can distinguish between real pairs (C_0, C_1) and *masked* pairs $(C_0 \oplus m, C_1 \oplus m)$, that are whitened with a random mask m. Since both distributions have the same differential properties $(C_0 \oplus m \oplus C_1 \oplus m = C_0 \oplus C_1)$, they conclude that a small non-differential component is learned by the distinguisher. In SPECK, one of the inputs

to the last modular addition can directly be obtained from the ciphertexts; its knowledge gives additional information on the differential propagation through the modular addition, as analyzed for instance in the case of modular addition with a constant in [ARS+22b]. As such, we expect a neural distinguisher trained on the (PD, 1PD, 2PD) format, without access to the ciphertexts, would perform just as well even without access to the ciphertexts. Indeed, partial decryption with a random key captures the probability distribution skew introduced by the knowledge of one addend. However, our experiments did not confirm that hypothesis, as we observe slightly lower accuracy than in the basic pairs scenario. This could be due to the larger input size, making it more difficult for the ND to locate relevant information. Looking at Table 7.5 we have the opposite observations compared to Aradi. Indeed, comparing the first data format to the second and the last ones, we can notice the drop in the accuracy for both the Single-Pair and Multi-Pair distinguisher.

Table 7.5 We build neural distinguishers for Speck in the standard format and our Generic Partial Decryption format and evaluate the accuracy on a single ciphertext pair ($m = 1$), as well as on multiple ciphertext pairs ($m = 8$) (c.f. subsection 7.1.6). Results of Speck 32/64

Round	[LLS]	This Work										
		DBitNet (C, C')			**DBitNet** $(\Delta^r, C, C', \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2})$			**GohrAMS** $(\Delta^r, C, C', \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2})$			**DBitNet** $(\Delta^r, \tilde{\Delta}^{r-1}, \tilde{\Delta}^{r-2})$		
Speck 32/64 Round	$m=1$ Simple	$m=8$[a] Simple	$m=8$[a] Pol.	$m=1$ Simple	$m=8$[a] Simple	$m=8$[a] Pol.	$m=1$ Simple	$m=8$[a] Simple	$m=8$[a] Pol.	$m=1$ Simple	$m=8$[a] Simple	$m=8$[a] Pol.
5	0.9269	1.0000	-	0.9197	0.9999	-	0.9246	0.9999	-	0.9199	0.9999	-
6	0.7870	0.9898	-	0.7708	0.9859	-	0.7789	0.9879	-	0.7715	0.9859	-
7	0.6142	0.7981	-	0.5976	0.7638	-	0.6124	0.7926	-	0.5986	0.7654	-
8	**0.5106**	**0.5325**	**0.5467**	0.5077	0.5235	0.5366	0.5097	0.5285	0.5449	0.5080	0.5247	0.5366

[a] The $\mathcal{ND}$ is the same of the case $m = 1$ but we change the evaluation step according subsection 7.1.6.

Observation on key dependencies. Recalling the round function of Speck:

$$L^i = [(L^{i-1} \gg \alpha) \boxplus R^{i-1}] \oplus k^i$$
$$R^i = (R^{i-1} \ll \beta) \oplus [(L^{i-1} \gg \alpha) \boxplus R^{i-1}] \oplus k^i$$

So, knowing the ciphertext at round r, it is possible to recover the right part:

$$\begin{aligned} R^{r-1} &= (R^r \oplus L^r) \gg \beta \\ (R')^{r-1} &= ((R')^r \oplus (L')^r) \gg \beta \end{aligned} \implies \Delta_R^{r-1} = R^{r-1} \oplus (R')^{r-1}$$

$$L^{r-1} = ((L^r \oplus k^r) \boxminus R^{r-1}) \ll \alpha$$
$$(L')^{r-1} = (((L')^r \oplus k^r) \boxminus (R')^{r-1}) \ll \alpha$$

The difference on the left part still depends on the key.

7.2.4 Remark and Open Question

From the previous analysis and by examining the round functions of all the ciphers, we note that the linear components—permutation, addition with the round key, and rotation—are generally similar across ciphers. This suggests that the improvements achieved through General Partial Decryption are likely influenced by the diffusion properties of the non-linear operations. For Simon and Simeck, this is attributed to the AND operation; for Aradi, the SBox; and for Speck, the modular addition. We saw that the Partial Differential could sometimes help the improvement of the $\mathcal{ND}$, though we are adding more data to it (and making the training slower). It is not clear a priori whether the application of this technique could lead to better results and if it depends on the architecture of each cipher. More analysis in this road is required.

7.3 Key Recovery Attack via Generic Partial Decryption on Aradi

In [Goh19], neural distinguishers are used to mount a key recovery attack. The attack divides the cipher into three parts: the Prepended n_{PP} rounds, the Neural Distinguisher on n_{ND} rounds, and the Key Recovery part on n_{KR} rounds (see section 7.3).

The Basic Attack. In its most basic form, Gohr's neural key recovery [Goh19] targets $n_{\text{ND}} + n_{\text{KR}}$ rounds. The attacker queries the encryption of plaintext pairs with difference δ. For each candidate subkey in the last n_{KR} rounds, the pairs are decrypted and submitted to the neural distinguisher. Based on the wrong key randomization hypothesis, the distribution induced by incorrect key decryption is assumed to differ from the distribution learned by the neural distinguisher.

Wrong Key Response Profile. Decryption with an incorrect key does not, in practice, induce a random distribution of the pairs for Speck 32. The author of [Goh19] observes that the XOR difference between a wrong key and the actual key influences the scoring output by the neural distinguisher. A **wrong key response profile** quantifies, for each key difference value, the expected distribution of the output of the neural distinguisher. This information helps greatly reduce the number of examined candidate keys, as it selects which direction to walk to get closer to the correct key.

Reaching More Rounds. If there exists an input difference γ, which has probability p to propagate to the $\mathcal{ND}$ input difference δ after n_{PP} rounds, then a pair with input difference γ encrypted for $n_{\text{PP}} + n_{\text{ND}}$ rounds should be classified as real by the n_{ND} rounds neural distinguisher with probability $p \cdot \texttt{TPR} + (1 - p) \cdot \texttt{FPR}$, where TPR and FPR denote respectively the true positive and false positive rates of the neural

distinguisher. For a random permutation, this probability simply becomes FPR. The number of positive predictions can, therefore, be used as a distinguisher for $n_{\text{PP}} + n_{\text{ND}}$ rounds. However, the number of pairs required to reach over 50% confidence with the above distinguisher grows with the square of the inverse of p; to counter that effect, the author of [Goh19] proposes to use **Probabilistic Neutral Bits** (PNBs) to build structures within which all the pairs are expected to behave similarly in the first n_{PP} rounds.

Definition 7.3 (Probabilistic Neutral Bit)**.** Let π_k be a keyed permutation, and $\gamma \to \delta$ be a differential for π_k. A bit i is a **Neutral Bit** for γ, δ if it does not influence the differential. Let x_i^* denote plaintext x with bit position i flipped; if

$$\mathbb{P}(\pi_k(x_i^*) \oplus \pi_k(x_i^* \oplus \gamma) = \delta | \pi_k(x) \oplus \pi_k(x \oplus \gamma) = \delta) = 1 \tag{7.5}$$

then, bit i is neutral. More generally, i is a **Probabilistic Neutral Bit** with probability p if

$$\mathbb{P}(\pi_k(x_i^*) \oplus \pi_k(x_i^* \oplus \gamma) = \delta | \pi_k(x) \oplus \pi_k(x \oplus \gamma) = \delta) = p \tag{7.6}$$

Attacking Large Round Keys. The above attack requires iterating over all possible key error bit vectors. Therefore, little attention has been given to neural key recovery on block ciphers with larger round keys. The main result for that setting [CBSY22] proposes to use different neural distinguishers for different subsets of the key bits. These subsets are identified through the *bit sensitivity test* [CSY23], which identifies *informative bits* of the ciphertext w.r.t the classification by the neural distinguisher. A key recovery using this technique focuses on key bits in the last round key that have a high impact on the corresponding ciphertext bits after partial decryption.

7.3.1 PNKBs and How to Find Them

Aradi uses large (128-bit) round keys. Therefore, we build an alternative key recovery strategy using probabilistic neutral bits in the *decryption direction*. This technique was introduced by Aumasson for differential-linear cryptanalysis against ChaCha [AFK+08]. To distinguish these bits from the neutral bits used in the encryption direction, we call them Probabilistic Neutral Key Bits (PNKBs).

A bit of the last round key is said to be a PKNB when, if it is incorrectly guessed, it does not change the prediction of the neural distinguisher for the corresponding partially decrypted ciphertext pair. In [AFK+08], probabilistic neutral bits are defined as follows for a function $f(k, W)$ of an unknown partial key k and known information W:

Definition 7.4 (Probabilistic Neutral Key Bit (PNKB))**.** The neutrality measure of

the key bit k_i with respect to the function $f(k, W)$ is defined as γ_i, where

$$\gamma_i = 2\mathbb{P}(f(k, W) = f(k_i^*, W)) - 1$$

Furthermore,

$$\mathbb{P}(f(k, W) = f(k_i^*, W)) = \frac{1}{2}(1 + \gamma_i)$$

is the probability (over all k and W) that complementing the key bit k_i does not change the output of $f(k, W)$.

In particular, the singular cases of the neutrality measure are:

- $\gamma_i = 1 \iff \mathbb{P}(f(k, W) = f(k_i^*, W)) = 1$: $f(k, W)$ does not depend on i-th key bit (i.e. it is a neutral bit)
- $\gamma_i = 0 \iff \mathbb{P}(f(k, W) = f(k_i^*, W)) = \frac{1}{2}$: $f(k, W)$ is statistically independent of the i-th key bit (i.e. it is a **significant** bit)
- $\gamma_i = -1 \iff \mathbb{P}(f(k, W) = f(k_i^*, W)) = 0$: $f(k, W)$ linearly depends on the i-th key bit.

In our analysis, given an $r + 1$-round ciphertext pair, we decrypt the last round using a candidate key k and build the candidate decryptions

$$C_0^{r*}(k) = g_{r+1}^k(C_0) \quad \text{and} \quad C_1^{r*}(k) = g_{r+1}^k(C_1)$$

adapting the notations introduced before and where we denote the key-candidate dependencies. From these, we can build the differential at round r and the 1 and 2-Partial Differential:

$$\Delta^{r*}(k) = C_0^{r*}(k) \oplus C_1^{r*}(k), \quad \Delta^{r-1*}(k), \quad \Delta^{r-2*}(k)$$

Then, we set

$$W(k) = (\Delta^{r*}(k), \tilde{\Delta}^{r-1*}(k), \tilde{\Delta}^{r-2*}(k)) \quad \text{and} \quad f(k, W) = \mathcal{NN}(W(k))$$

From this setting, a PNKB is a key bit that does not change the output of the neural distinguisher when guessed incorrectly, in other words when the neutrality is 1. To test whether a key bit b is a PNKB, we build a dataset $\mathcal{D}_{\text{PNKB}}$, in which the random samples are defined as usual and the real samples are built from

$$C_0^* = g_{r+1}^{k^b}(C_0) \quad \text{and} \quad C_1^* = g_{r+1}^{k^b}(C_1)$$

where k^b is the correct round key k with bit b randomized (see Algorithm 22).

Algorithm 22 `Generic_Partial_Decryption_Dataset_for_PNKB`.

Require: Round r, number of samples N, and input difference δ, dataformat (LD, 1PD,2PD) `data_format`, position to randomize b.

$\mathcal{D} \leftarrow \emptyset$
for $j = 1, \ldots, N$ **do**
  $P, k \leftarrow$ random choice of $P \in \mathcal{P}$, $k \in \mathcal{K}$
  $y \leftarrow$ random choice from $\{0, 1\}$ ▷ Randomly choose a label
  $(C, C') \leftarrow (E_k^{r+1}(P), E_k^{r+1}(P \oplus \delta))$ ▷ Generate ciphertext pair
  $k^b \leftarrow (k_0, k_1, \ldots, k_{b-1}, \text{RandomBit}, k_{b+1}, \ldots, k_{n-1})$
  $(C, C') \leftarrow (g_{r+1}^{k^b}(C), g_{r+1}^{k^b}(C'))$
  $(C, C') \leftarrow$ random pair of bit strings if $y = 0$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{$bit vector according to `dataformat`$\}$ ▷ cf. subsection 7.1.1
end for
return $\mathcal{D}$

If the bit is indeed a PNKB, then the accuracy of the neural distinguisher on this modified dataset is equivalent to its accuracy on a normal r round dataset (see Algorithm 23).

Algorithm 23 `Find_PNKB`.

Require: Oracle access to an r-round block cipher $E_k(\cdot)$, where $P \in \mathcal{P}$, $k \in \mathcal{K}$, and $C \in \mathcal{C}$ such that $C = E_k(P)$, dataformat (LP,DP1,DP2) `data_format`, round to attack r, input difference δ, best distinguisher $\mathcal{ND}$ for r rounds, position of the key b to randomize.

$\mathcal{D}_{\text{PNKB}} \leftarrow$ `GeneratePDDataset_for_PNKB`$(r, N_{\text{train}}, \delta,$ `data_format`$, b)$
$\mathcal{D}_{\text{standard}} \leftarrow$ `GeneratePDDataset`$(r, N_{\text{train}}, \delta,$ `data_format`$)$
$A_{\text{PNKB}} \leftarrow$ accuracy of $\mathcal{ND}$ on $\mathcal{D}_{\text{PNKB}}$
$A_{\text{standard}} \leftarrow$ accuracy of $\mathcal{ND}$ on $\mathcal{D}_{\text{standard}}$
if $A_{\text{PNKB}} \approx A_{\text{standard}}$ **then**
  return b
end if
return Not found

Preliminary experiments show that some key bits are naturally almost neutral in our trained $\mathcal{ND}$ for Aradi. However, building a practical attack requires a sufficient number of PNKBs, with neutrality measure as high as possible. We, therefore, retrain our neural distinguishers to reduce their sensitivity to changes in the PNKBs. More specif-

ically, starting from an empty set of PNKBs, and for each last round key bit position i, we retrain our $\mathcal{ND}$ on the corresponding $\mathcal{D}_{\text{PNKB}}$ dataset. If the accuracy remains sufficiently close to the basic $\mathcal{ND}$, we keep bit i as neutral and proceed to the next. If not, bit i is re-set to its correct value, and the next bit is investigated. At the end of this procedure, we have a set of PNKBs, according which round we are attacking, as well as the dual set of significant key bits i.e. where the neutrality is 0.

7.3.2 Rotationally Equivalent $\mathcal{ND}$ for 4-round of Aradi

Before starting with the actual key recovery, we underline another important property we discover in the analysis of the 4-round $\mathcal{ND}$ of Aradi.

Definition 7.5. Rotationally Equivalent Let $\mathcal{ND}_\delta$ be a neural distinguisher trained with input difference $\delta = \delta_{<1>}||\cdots\|\delta_{<t>}$ and accuracy $\mathcal{A}_\delta$. Let $\delta^{\lll\alpha}(j)$ be the original difference with the j-th word rotated of α:

$$\delta^{\lll}(j) = \delta_{<1>}||\cdots||\delta_{<j-1>}||\delta_{<j>} \lll \alpha||\delta_{<j+1>}||\cdots||\delta_{<t>}$$

Also, let $\mathcal{ND}_{\delta^{\lll\alpha}(j)}$ be a neural distinguisher trained with input difference $\delta^{\lll\alpha}(j)$ and accuracy $\mathcal{A}_{\delta^{\lll\alpha}(j)}$. Then, we say that $\mathcal{ND}_\delta$ and $\mathcal{ND}_{\delta^{\lll\alpha}(j)}$ are **rotationally equivalent** if and only if

$$\mathcal{A}_\delta \approx \mathcal{A}_{\delta^{\lll\alpha}(j)}$$

In the case of Aradi (see subsection 2.2.3), the state could be see as 4 words of 32 bits: $x = x_{<1>}||x_{<2>}||x_{<3>}||x_{<4>}$, but since Aradi applies a linear layer through a word divided into left and right part of 16 bits each, we can see the state of Aradi as:

$$x = x_{<1>}||\cdots||x_{<8>} \quad \text{with } x_{<j>} \in \{0,1\}^{16} \tag{7.7}$$

where $x_{<2i-1>}||x_{<2i>}$ is the i-th word of the orginal 4 state word. Let

$$\delta_{aradi,\tau}$$

be an input difference for Aradi with weight 1 in the τ-th word of Equation 7.7. Let $\mathcal{ND}_{\delta_{aradi,\tau}}$ be the neural distinguisher trained with input difference $\delta_{aradi,\tau}$ and accuracy $\mathcal{A}_{\delta_{aradi,\tau}}$. Let

$$\delta^{\lll\alpha}_{aradi,\tau}(\tau)$$

be the difference created by rotating (it is equivalent if left or right rotation but we can suppose, without lose of generality to the left) of α the τ-th word of 7.7. Then, we can train the neural distinguisher $\mathcal{ND}_{\delta^{\lll\alpha}_{aradi,\tau}(\tau)}$ with input difference $\delta^{\lll\alpha}_{aradi,\tau}(\tau)$ and accuracy $\mathcal{A}_{\delta^{\lll\alpha}_{aradi,\tau}(\tau)}$.

Through our experiments, we observe that:

$$\mathcal{A}_{\delta^{\lll\alpha}_{aradi,\tau}(\tau)} \approx \mathcal{A}_{\delta_{aradi},\tau} \quad \forall\alpha = 1,\ldots,16 \quad \forall\tau = 1,\ldots,8$$

In other words, $\mathcal{ND}_{\delta^{\lll\alpha}_{aradi}(\tau)}$ is rotationally equivalent to $\mathcal{ND}_{\delta_{aradi,\tau}}$ for all $\alpha = 1,\ldots 16$ and $\tau = 1,\ldots,8$.

Furthermore, these rotationally equivalent neural distinguishers map equivalent rotated PKNBs. More formally, let

$$\mathcal{I}_{PNKB}(\delta_{aradi,\tau}) = \{i_1,\ldots,i_k\}$$

be the set of index of the PNKBs of $\mathcal{ND}_{\delta_{aradi,\tau}}$. Let

$$\mathcal{B}_\tau = \{j | (\tau-1)\cdot 16 \leq j < \tau\cdot 16, \quad j \in \mathcal{I}_{PNKB}\}$$

be the index in the τ-th word, then we have:

$$\mathcal{I}_{PNKB}(\delta^{<<<\alpha}_{aradi,\tau}(\tau)) = \{[(i-\alpha) \bmod 16\ ] + 16\cdot(\tau-1) | i \in \mathcal{B}_\tau\} \cup (\mathcal{I}_{PNKB}(\delta_{aradi,\tau}) \setminus \mathcal{B}_\tau) \tag{7.8}$$

7.3.3 A Simple 5-rounds Key Recovery

The simplest version of our attack targets 5 rounds of Aradi and uses a 4-round neural distinguisher, trained on input difference with weight 1 found from the optimizer (see subsection 6.4.1):

$$\delta_{aradi} = \texttt{0x10000000000000000000}$$

In this case, this input difference should have the notation $\delta_{aradi,3}$, but we omit the 3 and fix $\tau = 3$ for the rest of the section. This neural distinguisher has over 99% accuracy (see Table 7.4), which enables low data complexity key recoveries. Furthermore, it

exhibits a set of 88 PNKBs, for last round key indices:

$$\begin{aligned}\mathcal{I}_{PNKB}(\delta_{aradi}) = \{ & \\ & 0, 3, 4, 7, 8, 9, 10, 11, 12, 14, 15, 17, 18, 19, 21, 22, 23, 26, 27, \\ & 28, 30, 31, 32, 35, 36, 39, 40, 41, 42, 43, 44, 46, 47, 49, 50, 51, \\ & 53, 54, 55, 58, 59, 60, 62, 63, 64, 67, 68, 71, 72, 73, 74, 75, 76, \\ & 78, 79, 81, 82, 83, 85, 86, 87, 90, 91, 92, 94, 95, 96, 99, 100, \\ & 103, 104, 105, 106, 107, 108, 110, 111, 113, 114, 115, 117, \\ & 118, 119, 122, 123, 124, 126, 127 \\ \} & \end{aligned} \tag{7.9}$$

From this set, we define the set of the rotational equivalent PNKBs. Let $\mathcal{I}_{PNKB}(\delta_{aradi}^{<<<\alpha})$ be the set of the index of the rotated input. Then, since in $\mathcal{I}_{PNKB}(\delta_{aradi})$ (see 7.9) the only index in the left part of the third word are:

$$\mathcal{B} = \{32, 35, 36, 39, 40, 41, 42, 43, 44, 46, 47\}$$

Using 7.8, we obtain:

$$\mathcal{I}_{PNKB}(\delta_{aradi}^{<<<\alpha}) = \{[(i - \alpha) \bmod 16\ ] + 32 | i \in \mathcal{B}\} \cup (\mathcal{I}_{PNKB}(\delta_{aradi}) \setminus \mathcal{B})$$

Basic Key Recovery Strategy. Given the subkey at the 5-th round: $k = (k_1, \ldots, k_{128})$, the basic key recovery strategy is:

- fix the 88 PNKBs to a random value: $\tilde{k} = (\tilde{k}_{i_1}, \ldots, \tilde{k}_{i_{80}})$;
- attack the remaining 40 non-neutral key bits: $\bar{k} = (\bar{k}_{j_1}, \ldots, \bar{k}_{j_{40}}) = k \setminus \tilde{k}$;

We iteratively recover a subset of the bits of $\bar{k}$ using 16 neural distinguishers for the corresponding 16 rotated input differences and PNKB sets. We fix α and let $\mathcal{ND}_\alpha$ be the neural distinguisher trained on the difference $\delta_{aradi}^{\lll\alpha}$ with data format (LD,1PD,2PD).

Let $\mathcal{I} = \{1, \ldots, 128\}$, then, we omit δ_{aradi} in the notation of the PNKBs index set:

$$\mathcal{I}_{PNKB}^{\alpha} := \mathcal{I}_{PNKB}(\delta_{aradi}^{<<<\alpha})$$

Also, we denote with $\bar{\mathcal{I}}^{\alpha}$ the index of the 40 non-neutral key bits:

$$\bar{\mathcal{I}}^{\alpha} = \mathcal{I} \setminus \mathcal{I}_{PNKB}^{\alpha}$$

such that

$$\bar{\mathcal{I}}^{\alpha} \cap \mathcal{I}^{\alpha}_{PNKB} = \emptyset \quad \text{and} \quad \bar{\mathcal{I}}^{\alpha} \cup \mathcal{I}^{\alpha}_{PNKB} = \mathcal{I}$$

The cardinality of these sets are $|\mathcal{I}^{\alpha}_{PNKB}| = 98$ and $|\bar{\mathcal{I}}^{\alpha}| = 40$.

For each iteration, we are using a subset of the index $\bar{I}^{\alpha} \subset \bar{\mathcal{I}}^{\alpha}$ and

$$\bar{k}^{\alpha}_{guess} = (\bar{k}_{guess}[i_1], \ldots, \bar{k}_{guess}[i_t])$$

where $[\cdot]$ indicate the position of the bit and $i_j \in \bar{I}^{\alpha}$.

We create a *total* key guess of $\bar{k}^{\alpha}_{guess}$ by adding random values to the neutral bits $\tilde{k}^{\alpha}_{random}$. Then, we evaluate the score of:

$$k^{\alpha}_{guess} = \tilde{k}^{\alpha}_{random} \cup \bar{k}^{\alpha}_{guess}$$

In total, we should have $2^{|\bar{I}^{\alpha}|}$ possible $\bar{k}^{\alpha}_{guess}$. Using a similar strategy shown in section 7.3. In particular, we set the counter $c_{\bar{k}^{\alpha}_{guess}} = 0$.

1. Create a pairs of plaintext (P, P') such that $P \oplus P' = \delta^{<<<\alpha}_{aradi}(1)$
2. Encrypt this pair to obtain the ciphertext pair for 5 rounds using the secret key K^*: $C = E^5_{K^*}(P)$ and $C' = E^5_{K^*}(P')$
3. For each key guess $\bar{k}^{\alpha}_{guess}$, create a whole subkey k^{α}_{guess}
4. For each k^{α}_{guess}, decrypt for one round the ciphetext pairs: $C^{k^{\alpha}_{guess}} = g_{k^{\alpha}_{guess}}(C)$ and $C'^{k^{\alpha}_{guess}} = g_{k^{\alpha}_{guess}}(C')$
5. Create the data format used for the $\mathcal{ND}_{\alpha}$;
6. Assign a score to k^{α}_{guess} using $\mathcal{NN}_{\alpha}$. Let

$$Z^{k^{\alpha}_{guess}} = \mathcal{NN}_{\alpha}((dataformat)_i)$$

   We increase the counter of $c_{\bar{k}^{\alpha}_{guess}}$ of 1 if $Z^{k^{\alpha}_{guess}} > 0.9$.
7. Repeat the process n times

We notice that some of the non-neutral bits have a strong signal that is reflected in consistently high score in the respective key guess (an high $c_{\bar{k}^{\alpha}_{guess}}$), but on the other hand, some key bits do not exhibit such a clear behavior.

We, therefore, assign a confidence score to each of the guessed key bits and only consider a bit **valid** when the score of the subkey, that it belongs to, is high enough. Let

$$C^{\alpha} = \max_{\bar{k}^{\alpha}_{guess}} c_{\bar{k}^{\alpha}_{guess}}$$

So, the confidence score will be:

$$0.9 \cdot C^{\alpha}$$

From this process, we are able to build a final $k^{\alpha}_{candidate}$ and the score for each bit. First, we build a vector:

$$a^{\alpha}[i] = \frac{1}{2^{|\bar{I}^{\alpha}|}} \sum_{\bar{k}^{\alpha}_{guess}: c_{\bar{k}^{\alpha}_{guess}} > 0.9 \cdot C^{\alpha}} k^{\alpha}_{guess}[i]$$

where k^{α}_{guess} contains the $\bar{k}^{\alpha}_{guess}$ bits. After, we can compute a key candidate $k_{candidate}$ of 128 bits:

$$k^{\alpha}_{candidate}[i] = \begin{cases} 1 & \text{if } a^{\alpha}[i] \geq 0.5 \\ 0 & \text{if } a^{\alpha}[i] < 0.5 \end{cases}$$

The score for the key bit is:

$$s^{\alpha}(j) = \begin{cases} 2 * \left| \frac{1}{2} - a^{\alpha}[j] \right| & \text{if } j \in \bar{I}^{\alpha} \\ 0 & \text{if } j \notin \bar{I}^{\alpha} \end{cases}$$

After that we switch to an another α and so we try to recover another set of the non-neutral key bits.

The choice of the next α is guided by the amount of corresponding **leftover bits**, bits which are part of the input difference's PNKBs and not part of the valid bits set (that is a subset of the non-neutral bit set), in other words the bits in $\mathcal{I}^{\alpha}_{PNKB}$.

To keep the attack practical, we need the set of attacked bits to be as small as possible, so we aim for 20 or fewer leftover bits. The attack continues until the set of valid bits has size 128 (*i.e.,* the entire last round key is recovered), or no new bit is recovered for a threshold number of iterations.

Once we have found 128 bits using the subkey at the 5-th round, to recover the all key (256 key), we repeat the same process with 4 round subkey (other 128 bits) using the 3-round neural distinguisher.

In order to experimentally validate this practical key recovery, the neural network evaluation for 2^{40} ($\bar{\mathcal{I}}^{\alpha} = \bar{I}^{\alpha}$) potential key candidates was an obstacle.

Since Equation 7.9 and 7.8, we can notice that:

$$\bigcap_{\alpha} \mathcal{I}^{\alpha}_{PNKB} \supseteq \mathcal{I}_{PNKB} \setminus \mathcal{B}$$

in other words, the PNKB sets for different distinguishers overlap. Since finding a

score for a key candidate implies that we have found a score for all the PNKBs with indices in $\mathcal{I}^{\alpha}_{PNKB}$ for a fixed α, and more specifically in $\mathcal{I}_{PNKB} \setminus \mathcal{B}$, we are then able to determine bits in positions i such that $i \in \mathcal{I}^{\beta}_{PNKB}$ with $\beta \neq \alpha$. This implies that once a small number of key bits have been recovered, it is usually possible to find an overlapping set with less than 20 key bits still to enumerate. We, therefore, assume preliminary knowledge of 30 key bits in the first explored set for practical validation and successfully recover the remaining 98 bits of the round 5 key.

Results. This attack was performed 25 times, each time with a fresh random key and a random selection of 30 known non-neutral key bits, and successfully recovered the entire last round key 21 times, with a maximum data complexity of 520 pairs, slightly over 2^9 chosen plaintexts. More specifically, each enumeration of a non-neutral key bits set is tested against 10 pairs, and the maximum number of iterations of the enumeration algorithm was 52 (with some PNKB sets attacked several times until a high enough confidence is achieved). It is likely that the data complexity and success rate of the attack can be further improved, for instance by using more pairs at each iteration (therefore increasing the confidence in the corresponding guesses and reducing the number of iterations), or with a smarter exploration of the connections between overlapping PNKB sets.

7.3.4 Extending the Attack

Our attack can be extended to 7 rounds, at the cost of non-practical time complexity, by having the distinguisher start from round 2 instead of round 0 (see Figure 7.1).

This requires retraining a neural distinguisher, as the rotational constants of rounds 2 to 6 are not the same as the ones from rounds 0 to 4. Nonetheless, equivalent distinguishers can be built from round 2, which is consistent with observations on the deterministic propagation of input differences through the cipher.

In addition, we observe that due to the SPN structure of Aradi, the accuracy of neural distinguishers and differential propagation are similar regardless of the row of the Hamming weight 1 input difference (as we saw in 7.8) considering the Aradi state of Equation 7.7 like that:

$$x = \begin{bmatrix} x^{(0)}_{<1>} & \cdots & x^{(15)}_{<1>} & x^{(0)}_{<2>} & \cdots & x^{(15)}_{<2>} \\ x^{(0)}_{<3>} & \cdots & x^{(15)}_{<3>} & x^{(0)}_{<4>} & \cdots & x^{(15)}_{<4>} \\ x^{(0)}_{<5>} & \cdots & x^{(15)}_{<5>} & x^{(0)}_{<6>} & \cdots & x^{(15)}_{<6>} \\ x^{(0)}_{<7>} & \cdots & x^{(15)}_{<7>} & x^{(0)}_{<8>} & \cdots & x^{(15)}_{<8>} \end{bmatrix}$$

Furthermore, truncated differences within a single row share similar properties, meaning that when prepending rounds, we do not need to target a specific bit but a column and

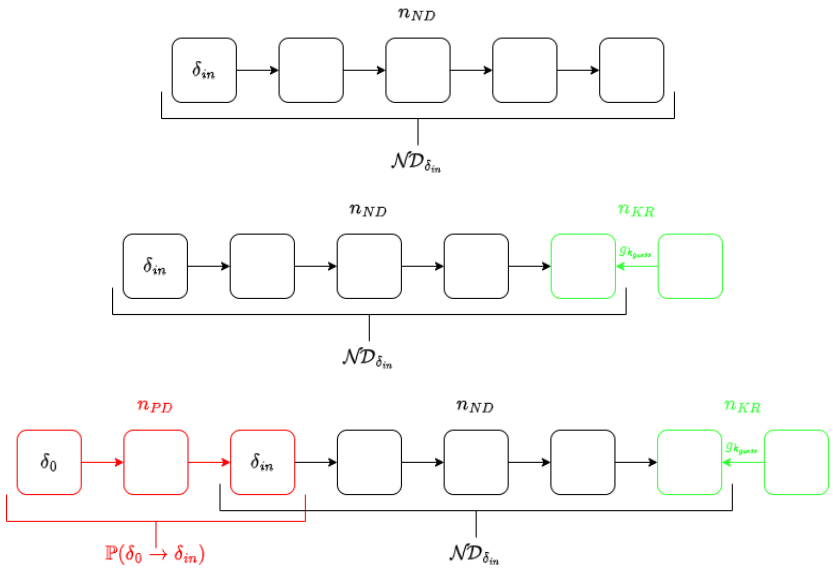

FIGURE 7.1 In the top image, it is presented the n_{ND} rounds (in this case 4) where the $\mathcal{ND}_{\delta_{in}}$, trained on δ_{in} works. In the middle, it is shown how a standard Neural Recovery works, adding n_{KR} rounds (in this case 1). In the bottom image, we preprend n_{PD} round (in this case two) from δ_0 to δ_{in}. Once the differential is found for n_{PD} rounds, it is trained a $\mathcal{ND}_{\delta_{in}}$ with input difference δ_{in} for n_{ND} rounds and then, it is attached other n_{KR} rounds to make the key guess.

benefit from a differential effect.

For Aradi, the best 2-round differential trails that end with a single-column output difference have probability 2^{-24}. We focus, without loss of generality, on the differential transition from $(\texttt{0xca10314}, \texttt{0xca10314}, 0, 0)$ to an output difference where a single active S-Box is present in column 15. In other words:

$$(\texttt{0xca10314}, \texttt{0xca10314}, 0, 0) \to \begin{bmatrix} 0 & \cdots & * & 0 & \cdots & 0 \\ 0 & \cdots & * & 0 & \cdots & 0 \\ 0 & \cdots & * & 0 & \cdots & 0 \\ 0 & \cdots & * & 0 & \cdots & 0 \end{bmatrix} = \Delta$$

where $*$ indicates if the difference is present or not, $*$ are in positions $(1, 15), (2, 15), (3, 15), (4, 15)$ and we ask that just one of them is active. Also, we have:

$$\mathbb{P}[(\texttt{0xca10314}, \texttt{0xca10314}, 0, 0) \to \Delta] = 2^{-20}$$

This truncated differential transition holds with approximate probability 2^{20} (as experimentally verified).

We experimentally find a set of 76 neutral bits for this differential transition, which may enable longer key recoveries. For a 7-round attack, however, we can use significantly less. Remember that our 4-round distinguishers have over 99% accuracy, so the response profile for pairs that do not follow the differential is very distinct.

Therefore, in the attack, we can build small structures of 2^8 pairs; we need on average 2^{20} (due the probability) structures to find one that follows the 2-round differential. Within each structure, we test one pair at a time by testing the corresponding 40 key-bit candidates; the response profile of the correct structure is expected to be the highest.

From there, the attack proceeds as in the previous section, using different neural distinguishers for different key-bit subsets. We expect that with 2^8 pairs per structure, the key recovery procedure requires a single pass for each of the input differences, resulting in a worse-case data complexity of $2^{20} \cdot 2^8 \cdot 16 = 2^{32}$. The time complexity is on average 2^{20} neural distinguisher evaluations per pair, but dominated by the initial 2^{40} neural evaluations for the first PKNB set, yielding a total time complexity of 2^{72} $\mathcal{ND}$ evaluations.

From Neural to Classical Key Recovery. The neural distinguisher findings are helpful in identifying that there is a relevant property for 4 and 5 rounds. However, for cryptographers, knowing there is an attack is not sufficient, and understanding it is crucial. We therefore propose to use the $\mathcal{ND}$ results to mount a classical key recovery. For the 4-round distinguishers, the key property is that a set of 40 bits (10 columns) in round 3 is always 0 due to the slow diffusion of the single active nibble in the input difference. The activity pattern of the S-Boxes is preserved by one round of partial decryption so that decrypting the 4-round pair with a random key is sufficient to observe the 40 zeros pattern for a real pair. On the other hand, for random pairs, it occurs with probability 2^{-40}, so that the corresponding distinguisher is highly accurate: its true positive rate is 1 (as all real pairs follow this pattern), and its false positive rate is $1\text{-}2^{-40}$.

Transforming this distinguisher into a 5-round key recovery requires identifying the key bits involved in the computation of the 40 fixed positions in round 3. Each of these columns depends, by the construction of the linear layer, on 3 columns of the round 4 difference. Since we are only interested in the activity pattern and not the actual value of the S-Box inputs, we do not need the corresponding round 4 key. On the other hand, each of these round 4 columns depends on 3 columns of round 5, for which we do need to obtain the key to get the correct values for the columns of round 3. More specifically, there are 10 sets of 36 key bits, each corresponding to one column, to be recovered.

The probability for an incorrect key guess to result in the expected 4-zeros column pattern is 2^{-4}; after analyzing N pairs, the probability for a wrong key guess to appear

for all N pairs is $2^{-4 \cdot N}$. This figure matches the neural key recovery experiments, where 10 pairs were sufficient for each key guessing phase.

Complexity analysis. The data complexity of the corresponding key recovery is 10 pairs, and the time complexity, $2^{36} \cdot 4 \cdot 10$, rounded up to 2^{42}. In comparison, the first step of the neural key recovery of the previous section requires $10 \cdot 2^{40}$ $\mathcal{ND}$ evaluations, with each $\mathcal{ND}$ evaluation accounting for significantly more than a simple operation, resulting in higher time complexity. On the other hand, the neural key recovery may perform similarly, or better, if retrained for the subsets of 36 non-neutral key bits used in the classical attack; however, we did not perform such experiments, as we find the classical attack more satisfying.

7.4 Conclusions and Open Questions

In this work, we recognize neural distinguishers as a useful tool for identifying differential properties that do not immediately appear from classical differential characteristics. We, therefore, aim to make their application to new primitives easier, building on the generic AutoND framework and making it more competitive with dedicated approaches. In particular, we notice that recent works, such as [LLS+24], break the classical black-box model (where the neural distinguisher is only given ciphertext pairs) to include information on the structure of the cipher, by including features derived from the deterministic (and sometimes probabilistic) propagation of the difference to the previous rounds. We propose Generic Partial Decryption as a way to include such features automatically, and evaluate the resulting framework on Simon, Speck, and Aradi.

Our experiments show that this approach closes most of the performance gap between fully automated neural distinguishers and more hand-crafted feature-engineering pipelines. In particular, we observe that Generic Partial Decryption allows AutoND to recover a large fraction of the accuracy achieved by the state-of-the-art approach of [LLS+24] over the AutoND framework, but that the multi-pair aspect plays the most important role (even though feature engineering helps fill a small gap between the two techniques). In addition, we propose the first neural key recovery on 5 rounds of Aradi, with practical complexity, and propose an equivalent classical attack, informed by the biases exhibited by the neural distinguisher, with better complexity.

Despite the encouraging results, a number of open questions remain.

- *Feature selection and representation.* While Generic Partial Decryption provides a generic way to derive features, it is still unclear how to select, compress, or weight them optimally. In particular, understanding which partial decryptions carry the most discriminative information, and how to avoid redundant or noisy features, remains an open problem.

- *Scalability to larger ciphers and deeper rounds.* The experiments in this work focus on lightweight block ciphers and a limited number of rounds. It is an open question how well the proposed approach scales to larger designs or to settings where only very small differential biases are available.
- *Interaction between neural and classical cryptanalysis.* The ARADI case study suggests that biases uncovered by neural distinguishers can inform more efficient classical attacks. More generally, it remains to be explored how much similar insight can be leveraged to systematically improve classical attacks, and how to extract and exploit such biases in a principled way.
- *Architectural choices and input size.* We observe that the addition of features can sometimes lower the accuracy of neural distinguishers, possibly due to the resulting increase in input dimensionality. This suggests that current network architectures may not be well suited for handling large or heterogeneous inputs. Addressing this limitation would help further improve automated neural cryptanalysis, in turn improving our understanding of multiple differential properties.
- *Beyond differential properties.* Finally, it would be interesting to investigate whether similar generic feature-engineering techniques can be applied to other forms of cryptanalysis, such as linear or integral distinguishers, or to side-channel-inspired settings.

We believe that addressing these questions will be key to making automated neural cryptanalysis both more robust and more informative, and to further bridging the gap between data-driven and classical cryptanalytic techniques.

References

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, G.s Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. 03 2016.

[ABR20] Tomer Ashur, Tim Beyne, and Vincent Rijmen. Revisiting the wrong-key-randomization hypothesis. *Journal of Cryptology*, 33, 02 2020.

[ADG24] Roberto Avanzi, Orr Dunkelman, and Shibam Ghosh. A note on ARADI and LLAMA. Cryptology ePrint Archive, Paper 2024/1328, 2024.

[AFK+08] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New features of latin dances: Analysis of Salsa, ChaCha, and Rumba. In Kaisa Nyberg, editor, *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, volume 5086 of *Lecture Notes in Computer Science*, pages 470–488. Springer, 2008.

[Apo67] Tom M. Apostol. *Calculus, Vol. 1: One-Variable Calculus, with an Introduction to Linear Algebra*. Wiley, 1967.

[ARS+22a] Seyyed Arash Azimi, Adrián Ranea, Mahmoud Salmasizadeh, Javad Mohajeri, Mohammad Reza Aref, and Vincent Rijmen. A bit-vector differential model for the modular addition by a constant and its applications to differential and impossible-differential cryptanalysis. *Designs, Codes and Cryptography*, 90(8):1797–1855, 2022.

[ARS+22b] Seyyed Arash Azimi, Adrián Ranea, Mahmoud Salmasizadeh, Javad Mohajeri, Mohammad Reza Aref, and Vincent Rijmen. A bit-vector differential model for the modular addition by a constant and its applications to differential and impossible-differential cryptanalysis. *Designs, Codes and Cryptography*, 90(8):1797–1855, 2022.

[Bar93] Andrew Barron. Barron, a.e.: Universal approximation bounds for superpositions of a sigmoidal function. ieee trans. on information theory 39, 930-945. *Information Theory, IEEE Transactions on*, 39:930 – 945, 06 1993.

[BB07] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.

[BBB+10] James Bergstra, Olivier Breuleux, Frederic Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yere Yere. Theano: A cpu and gpu math compiler in python. *PROC. OF THE 9th PYTHON IN SCIENCE CONF*, 01 2010.

[BBCD22] Anubhab Baksi, Jakub Breier, Yi Chen, and Xiaoyang Dong. Machine learning-assisted differential distinguishers for lightweight ciphers. *Classical and Physical Security of Symmetric Key Cryptographic Algorithms*, pages 141–162, 2022.

[BBG+25] Emanuele Bellini, Rocco Brunelli, David Gerault, Anna Hambitzer, and Marco Pedicini. *Generic Partial Decryption as Feature Engineering for Neural Distinguishers*, pages 373–398. 10 2025.

[BBI+21] Subhadeep Banik, Zhenzhen Bao, Takanori Isobe, Hiroyasu Kubo, Fukang Liu, Kazuhiko Minematsu, Kosei Sakamoto, Nao Shibata, and Maki Shigeri. Warp : Revisiting gfn for lightweight 128-bit block cipher. In Orr Dunkelman, Michael J. Jacobson, Jr., and Colin O'Flynn, editors, *Selected Areas in Cryptography*, pages 535–564, Cham, 2021. Springer International Publishing.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[BD62] Arthur E. Bryson and Walter F. Denham. A steepest-ascent method for solving optimum programming problems. *Journal of Applied Mechanics*, 29:247–257, 1962.

[BDCD03] Alex Biryukov, Christophe De Cannière, and Gustaf Dellkrantz. Cryptanalysis of safer++. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 195–211, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[BDD+24] Christina Boura, Nicolas David, Patrick Derbez, Rachelle Heim Boissier, and María Naya-Plasencia. A generic algorithm for efficient key recovery in differential attacks – and its associated tool. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 217–248, Cham, 2024. Springer Nature Switzerland.

[BDK01] Eli Biham, Orr Dunkelman, and Nathan Keller. The rectangle attack — rectangling the serpent. In Birgit Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, pages 340–357, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[BFG+24] Emanuele Bellini, Mattia Formenti, David Gérault, Juan Grados, Anna Hambitzer, Yun Ju Huang, Paul Huynh, Mohamed Rachidi, Raghvendra Rohit, and Sharwan K. Tiwari. CLAASPing ARADI: Automated analysis of the ARADI block cipher. Cryptology ePrint Archive, Paper 2024/1324, 2024.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The satisfiability modulo theories library (smt-lib). `http://www.smt-lib.org/`, 2016.

[BG11] Céline Blondeau and Benoît Gérard. Multiple differential cryptanalysis: Theory and practice. In *Fast Software Encryption - 18th International Workshop, FSE 2011*, volume 6733 of *Lecture Notes in Computer Science*, page 35. Springer, 2011.

[BGG+24] Emanuele Bellini, David Gerault, Juan Grados, Yun Huang, Rusydi H. Makarim, Mohamed Rachidi, and Sharwan Tiwari. *CLAASP: A Cryptographic Library for the Automated Analysis of Symmetric Primitives*, pages 387–408. 02 2024.

[BGH+23] E Bellini, D Gerault, A Hambitzer, M Rossi, et al. A cipher-agnostic neural training pipeline with automated finding of good input differences. *IACR TRANSACTION ON SYMMETRIC CRYPTOLOGY*, 2023(3):184–212, 2023.

[BGlL+22] Zhenzhen Bao, Jian Guo, Mei li Liu, Li Ma, and Yi Tu. Enhancing differential-neural cryptanalysis. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2022.

[BGPT21] Adrien Benamira, David Gerault, Thomas Peyrin, and Quan Quan Tan. A deeper look at machine learning-based cryptanalysis. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 805–835, Cham, 2021. Springer International Publishing.

[BHHW04] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. The tractability of global constraints. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 716–720, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[BHL+20] Hamid Boukerrou, Paul Huynh, Virginie Lallemand, Bimal Mandal, and Marine Minier. On the feistel counterpart of the boomerang connectivity table: Introduction and analysis of the fbct. *IACR Transactions on Symmetric Cryptology*, 2020(1):331–362, May 2020.

[BHS79] Arthur Bryson, Y.-C Ho, and George Siouris. Applied optimal control: Optimization, estimation, and control. *Systems, Man and Cybernetics, IEEE Transactions on*, 9:366 – 367, 07 1979.

[Bir11] Alex Biryukov. *Feistel Cipher*, pages 455–455. Springer US, Boston, MA, 2011.

[Bis94] Christopher M. Bishop. Mixture density networks. 1994.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The skinny family of block ciphers and its low-latency variant mantis. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 123–153, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[BK09] Alex Biryukov and Dmitry Khovratovich. Related-key cryptanalysis of the full aes-192 and aes-256. volume 2009, page 317, 12 2009.

[BL07] Yoshua Bengio and Yann Lecun. Scaling learning algorithms towards ai. 01 2007.

[BLP+07] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, and U. Montreal. Greedy layer-wise training of deep networks. volume 19, 01 2007.

[BLP+12] Frederic Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yere Yere. Theano: new features and speed improvements. *CoRR*, 11 2012.

[BLYZ23] Zhenzhen Bao, Jinyu Lu, Yiran Yao, and Liu Zhang. More insight on deep learning-aided cryptanalysis. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 436–467. Springer, 2023.

[BMR26] Rocco Brunelli, Marine Minier, and Loïc Rouquette. Integrating boomerang into TAGADA. Cryptology ePrint Archive, Paper 2026/121, 2026.

[BN10] Alex Biryukov and Ivica Nikolić. Automatic search for related-key differential characteristics in byte-oriented block ciphers: Application to aes, camellia, khazad and others. Cryptology ePrint Archive, Paper 2010/248, 2010. `https://eprint.iacr.org/2010/248`.

[Bot91] Léon Bottou. Stochastic gradient learning in neural networks. 1991.

[BPL10] Y-Lan Boureau, J. Ponce, and Yann Lecun. A theoretical analysis of feature pooling in visual recognition. pages 111–118, 11 2010.

[BR21] Emanuele Bellini and Matteo Rossi. Performance comparison between deep learning-based and conventional cryptographic distinguishers. In *Intelligent Computing: Proceedings of the 2021 Computing Conference, Volume 3*, pages 681–701. Springer, 2021.

[Bri90] J. S. Bridle. Alpha-nets: a recurrent "neural" network architecture with a hidden markov model interpretation. *Speech Commun.*, 9(1):83–92, January 1990.

[BRPL11] Y-Lan Boureau, Nicolas Roux, J. Ponce, and Yann Lecun. Ask the locals: Multi-way local pooling for image recognition. 11 2011.

[BRRT24] Emanuele Bellini, Mohamed Rachidi, Raghvendra Singh Rohit, and Sharwan K. Tiwari. Mind the composition of toffoli gates: Structural algebraic distinguishers of aradi. *IACR Cryptol. ePrint Arch.*, 2024:1559, 2024.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *J. Cryptology*, 4:3–72, 1991.

[BSF94] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Paper 2013/404, 2013.

[C+15] Francois Chollet et al. Keras. `https://keras.io`, 2015. Accessed: 2026-01-17.

[CBSY22] Yi Chen, Zhenzhen Bao, Yantian Shen, and Hongbo Yu. A deep learning aided key recovery framework for large-state block ciphers. *Cryptology ePrint Archive*, 2022.

[CDF+23] Thibaut Cuvelier, Frederic Didier, Vincent Furnon, Steven Gay, Sarah Mohajeri, and Laurent Perron. OR-Tools' Vehicle Routing Solver: a Generic Constraint-Programming Solver with Heuristic Search for Routing Problems. In *24e congrès annuel de la société française de recherche opérationnelle et d'aide à la décision*, Rennes, France, February 2023. ROADEF.

[CHP+18] Carlos Cid, Tao Huang, Thomas Peyrin, Yu Sasaki, and Ling Song. *Boomerang Connectivity Table: A New Cryptanalysis Tool*, pages 683–714. 03 2018.

[CLN+17] Anne Canteaut, Eran Lambooij, Samuel Neves, Shahram Rasoolzadeh, Yu Sasaki, and Marc Stevens. Refined probability of differential characteristics including dependency between multiple rounds. *IACR Transactions on Symmetric Cryptology*, 2017(2):203–227, Jun. 2017.

[Col04] Ronan Collobert. Large scale machine learning. 01 2004.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[Cop94] D. Coppersmith. The Data Encryption Standard (DES) and its strength against attacks. *IBM J. Res. Dev.*, 38(3):243–250, May 1994.

[CR15] Anne Canteaut and Joëlle Roué. On the behaviors of affine equivalent sboxes regarding differential and linear attacks. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 45–74, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[Cra46] Harald Cramér. *Mathematical Methods of Statistics*. Princeton University Press, Princeton, 1946.

[CS15] Christopher Clark and Amos Storkey. Training deep convolutional neural networks to play go. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1766–1774, Lille, France, 07–09 Jul 2015. PMLR.

[CSKX15] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2722–2730, 2015.

[CSL13] Meng Cai, Yongzhe Shi, and Jia Liu. Deep maxout neural networks for speech recognition. pages 291–296, 12 2013.

[CSY23] Yi Chen, Yantian Shen, and Hongbo Yu. Neural-aided statistical attack for cryptanalysis. *The Computer Journal*, 66:2480–2498, 2023.

[CSYY21] Yi Chen, Yantian Shen, Hongbo Yu, and Sitong Yuan. A new neural distinguisher considering features derived from multiple ciphertext pairs. Cryptology ePrint Archive, Paper 2021/310, 2021.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[DB11] Olivier Delalleau and Yoshua Bengio. *Shallow vs. Deep Sum-Product Networks*, volume 24, pages 666–674. 01 2011.

[DBB+00] Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. Incorporating second-order functional knowledge for better option pricing. In Todd K. Leen, Thomas G. Dietterich, and Volker Tresp, editors, *NIPS*, pages 472–478. MIT Press, 2000.

[DCC23] Haoran Deng, Xianghui Cao, and Yu Cheng. Attention in differential cryptanalysis on lightweight block cipher SPECK. In *2023 20th Annual International Conference on Privacy, Security and Trust (PST)*, pages 1–9. IEEE, 2023.

[DCDK09] Christophe De Cannière, Orr Dunkelman, and Miroslav Knežević. Katan and ktantan — a family of small and efficient hardware-oriented block ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 272–288, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[DDG+24] François Delobel, Patrick Derbez, Arthur Gontier, Loïc Rouquette, and Christine Solnon. *A CP-Based Automatic Tool for Instantiating Truncated Differential Characteristics*, pages 247–268. 03 2024.

[DDV20a] Stéphanie Delaune, Patrick Derbez, and Mathieu Vavrille. Catching the fastest boomerangs: Application to skinny. *IACR Transactions on Symmetric Cryptology*, 2020(4):104–129, Dec. 2020.

[DDV20b] Stéphanie Delaune, Patrick Derbez, and Mathieu Vavrille. Catching the fastest boomerangs: Application to skinny. *IACR Transactions on Symmetric Cryptology*, 2020(4):104–129, Dec. 2020.

[DEM15] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. Heuristic tool for linear cryptanalysis with applications to caesar candidates. 12 2015.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 07 2011.

[DKS14] Orr Dunkelman, Nathan Keller, and Adi Shamir. A practical-time related-key attack on the kasumi cryptosystem used in gsm and 3g telephony. *Journal of Cryptology*, 27(4):824–849, oct 2014.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[DR06] Joan Daemen and Vincent Rijmen. Understanding two-round differentials in aes. pages 78–94, 09 2006.

[Dre62] Stuart Dreyfus. The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45, 1962.

[Dre73] S. Dreyfus. The computational solution of optimal control problems with time lag. *IEEE Transactions on Automatic Control*, 18(4):383–385, 1973.

[DSPK15] Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, and Koray Kavukcuoglu. Natural neural networks. 07 2015.

[ERP22] Amirhossein Ebrahimi, Francesco Regazzoni, and Paolo Palmieri. Reducing the cost of machine learning differential attacks using bit selection

and a partial ML-distinguisher. In *International Symposium on Foundations and Practice of Security*, pages 123–141. Springer, 2022.

[FJP13] Pierre-Alain Fouque, Jérémy Jean, and Thomas Peyrin. Structural evaluation of aes and chosen-key distinguisher of 9-round aes-128. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 183–203, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Fuk75] Kunihiko Fukushima. Cognitron: A self-organizing multilayered neural network. *Biol. Cybern.*, 20(3–4):121–136, September 1975.

[Fuk80] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[GBY10] Xavier Glorot, Antoine Bordes, and Yere Yere. Deep sparse rectifier neural networks. volume 15, 01 2010.

[GHHP24] David Gerault, Anna Hambitzer, Moritz Huppert, and Stjepan Picek. Sok: 5 years of neural differential cryptanalysis. *Cryptology ePrint Archive*, 2024.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[GLN22] Aron Gohr, Gregor Leander, and Patrick Neumann. An assessment of differential-neural distinguishers. Cryptology ePrint Archive, Paper 2022/1521, 2022.

[GMD+13] Ian Goodfellow, Mehdi Mirza, Xia Da, Aaron Courville, and Yere Yere. An empirical investigation of catastrophic forgeting in gradient-based neural networks. 12 2013.

[GMS16] David Gerault, Marine Minier, and Christine Solnon. Constraint programming models for chosen key differential cryptanalysis. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 584–601, Cham, 2016. Springer International Publishing.

[GMW24] Patricia Greene, Mark Motley, and Bryan Weeks. Aradi and llama: Low-latency cryptography for memory encryption. *IACR Cryptol. ePrint Arch.*, 2024:1240, 2024.

[Goh19] Aron Gohr. Improving attacks on round-reduced Speck32/64 using deep learning. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part II 39*, pages 150–179. Springer, 2019.

[Gom58] Ralph E. Gomory. Essentials of an algorithm for integer solutions to linear programs. In Robert L. Graves and Philip Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 269–302. McGraw-Hill, New York, USA, 1958.

[Goo10] Ian J. Goodfellow. Multidimensional, downsampled convolution for autoencoders. Technical report, Université de Montréal, 2010. Technical Report, Université de Montréal.

[Gra13] Alex Graves. Generating sequences with recurrent neural networks. *ArXiv*, abs/1308.0850, 2013.

[GWFM+13] Ian Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yere Yere. Maxout networks. *30th International Conference on Machine Learning, ICML 2013*, 1302, 02 2013.

[GY10] Xavier Glorot and Yere Yere. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010.

[Hås86] Johan Håstad. Almost optimal lower bounds for small depth circuits. In *Symposium on the Theory of Computing*, 1986.

[HBS21] Hosein Hadipour, Nasour Bagheri, and Ling Song. Improved rectangle attacks on skinny and craft. *IACR Transactions on Symmetric Cryptology*, 2021(2):140–198, Jun. 2021.

[HE21] Hosein Hadipour and Maria Eichlseder. Autoguess: A tool for finding guess-and-determine attacks and key bridges. 11 2021.

[HG91] Johan Håstad and Mikael Goldmann. On the power of small-depth threshold circuits. *Computational Complexity*, 1(2):113–129, 1991.

[Hin91] Geoffrey E. Hinton, editor. *Connectionist Symbol Processing*. MIT Press, Cambridge, MA, USA, 1991. Online edition.

[HKM95] Carlo Harpes, Gerhard G. Kramer, and James L. Massey. A generalization of linear cryptanalysis and the applicability of matsui's piling-up lemma. In Louis C. Guillou and Jean-Jacques Quisquater, editors, *Advances in Cryptology — EUROCRYPT '95*, pages 24–38, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[HLL+00] Seokhie Hong, Sangjin Lee, Jongin Lim, Jaechul Sung, Dong Hyeon Cheon, and Inho Cho. Provable security against differential and linear cryptanalysis for the spn structure. In *Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings*, volume 1978 of *Lecture Notes in Computer Science*, pages 273–283. Springer, 2000.

[HMK81] H. Haken and G. Mayer-Kress. Chapman-kolmogorov equation and path integrals for discrete chaos in presence of noise. *Zeitschrift für Physik B Condensed Matter*, 43(2):185–187, June 1981.

[HMP+93] András Hajnal, Wolfgang Maass, Pavel Pudlak, Mario Szegedy, and Gyorgy Turan. Threshold circuits of bounded depth. *J. Comput. Syst. Sci.*, 46:129–154, 04 1993.

[HMR86] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart. *Distributed representations*, page 77–109. MIT Press, Cambridge, MA, USA, 1986.

[HNE22] Hosein Hadipour, Marcel Nageler, and Maria Eichlseder. Throwing boomerangs into feistel structures: Application to clefia, warp, lblock, lblock-s and twine. *IACR Transactions on Symmetric Cryptology*, 2022(3):271–302, Sep. 2022.

[Hoc91] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. 04 1991.

[HOT06] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 07 2006.

[HRCF21] ZeZhou Hou, JiongJiong Ren, ShaoZhen Chen, and AnMin Fu. Improve Neural Distinguishers of SIMON and SPECK. *Sec. and Commun. Netw.*, 2021, jan 2021.

[HRWR02] Geoffrey Hinton, David Rumelhart, Ronald Williams, and Yoesoep Rachmad. *Learning Representations by Back-Propagating Errors*. 09 2002.

[HSW89] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert L. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[HSW90] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert L. White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3:551–560, 1990.

[HZRS15a] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2015.

[HZRS15b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *IEEE International Conference on Computer Vision (ICCV 2015)*, 1502, 02 2015.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[Jac87] Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1:295–307, 1987.

[JHD12] Yangqing Jia, Chang Huang, and T. Darrell. Beyond spatial pyramids: Receptive field learning for pooled image features. pages 3370–3377, 06 2012.

[JJNH91] Robert Jacobs, Michael Jordan, Steven Nowlan, and Geoffrey Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3:79–87, 03 1991.

[JKRL09] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann Lecun. What is the best multi-stage architecture for object recognition? volume 12, 09 2009.

[Kar14] William Karush. Minima of functions of several variables with inequalities as side conditions. 2014.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[Kel60] Henry J. Kelley. Gradient theory of optimal flight paths. *ARS Journal*, 30:947–954, 1960.

[KGV83] Scott Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science (New York, N.Y.)*, 220:671–80, 06 1983.

[Kir15] Aleksandar Kircanski. Analysis of boomerang differential trails via a sat-based constraint solver ursa. pages 331–349, 01 2015.

[KKS00] John Kelsey, Tadayoshi Kohno, and Bruce Schneier. Amplified boomerang attacks against reduced-round mars and serpent. In *Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings*, volume 1978 of *Lecture Notes in Computer Science*, pages 75–93. Springer, 2000.

[KL51] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[KMT01] Liam Keliher, Henk Meijer, and Stafford E. Tavares. New method for upper bounding the maximum average linear hull probability for spns. In

Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding, volume 2045 of *Lecture Notes in Computer Science*, pages 420–436. Springer, 2001.

[Knu95] Lars R. Knudsen. Truncated and higher order differentials. In Bart Preneel, editor, *Fast Software Encryption*, pages 196–211, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[Köl20] Stefan Kölbl. Cryptosmt: An easy to use tool for cryptanalysis of symmetric primitives. `https://github.com/kste/cryptosmt`, 2020.

[KS07] Liam Keliher and J. Sui. Exact maximum expected differential and linear probability for two-round advanced encryption standard. *Information Security, IET*, 1:53–57, 07 2007.

[KSMM12] Eita Kobayashi, Tomoyasu Suzaki, Kazuhiko Minematsu, and Sumio Morioka. Twine: A lightweight block cipher for multiple platforms. volume 7707, 03 2012.

[KT51] Harold W. Kuhn and Albert W. Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, Berkeley, Calif., 1951. University of California Press.

[L.47] CAUCHY A. L. Methode generale pour la resolution des systemes d'equations simultanees. *Comptes Rendus de l'Academie des Science*, 25:536–538, 1847.

[LBH+98] Yann Lecun, Yoshua Bengio, Patrick Haffner, Yoesoep Rachmad, and Leon Bottou. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278 – 2324, 12 1998.

[LCLH22] Dongdong Lin, Shaozhen Chen, Manman Li, and Zezhou Hou. The construction and application of (related-key) conditional differential neural distinguishers on KATAN. In *International Conference on Cryptology and Network Security*, pages 203–224. Springer, 2022.

[LD10] Ailsa H. Land and Alison G. Doig. *An Automatic Method for Solving Discrete Programming Problems*, pages 105–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[LDLS21] Luc Libralesso, François Delobel, Pascal Lafourcade, and Christine Solnon. Automatic generation of declarative models for differential cryptanalysis, 2021.

[Lec85] Y. Lecun. Une procédure d'apprentissage pour réseau à seuil asymétrique. *Proceedings of Cognitiva 85, Paris*, pages 599–604, 1985.

[LeC89] Yann LeCun. Generalization and network design strategies. 1989.

[Lei76] Gottfried Wilhelm Leibniz. Memoir using the chain rule, 1676. Historical manuscript, cited in The Mathematics Enthusiast (TMME), Vol. 7, Issues 2&3, pp. 321–332, 2010.

[Leu12] Gaëtan Leurent. Analysis of differential attacks in arx constructions. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, pages 226–243, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Leu13] Gaëtan Leurent. Construction of differential characteristics in arx designs application to skein. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 241–258, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[L'H96] Guillaume François Antoine L'Hôpital. *Analyse des infiniment petits, pour l'intelligence des lignes courbes*. L'Imprimerie Royale, Paris, 1696.

[LH88] Kevin J. Lang and Geoffrey E. Hinton. A time-delay neural network architecture for speech recognition, 1988.

[Lin76] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

[LKF10] Yann Lecun, Koray Kavukcuoglu, and Clement Farabet. Convolutional networks and applications in vision. pages 253–256, 05 2010.

[LLPS93] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

[LLS+24] Jinyu Lu, Guoqiang Liu, Bing Sun, Chao Li, and Li Liu. Improved (related-key) differential-based neural distinguishers for SIMON and SIMECK block ciphers. *The Computer Journal*, 67(2):537–547, 2024.

[LM02] Helger Lipmaa and Shiho Moriai. Efficient algorithms for computing differential properties of addition. pages 35–45, 06 2002.

[LMM91] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In Donald W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, pages 17–38, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[LMR22] Virginie Lallemand, Marine Minier, and Loïc Rouquette. Automatic search of rectangle attacks on feistel ciphers: Application to warp. *IACR Transactions on Symmetric Cryptology*, 2022(2):113–140, Jun. 2022.

[LSF87] Yann Lecun and Francoise Soulie Fogelman. Modeles connexionnistes de l'apprentissage. *Intellectica, special issue apprentissage et machine*, 2, 01 1987.

[LTZ22] Lijun Lyu, Yi Tu, and Yingjie Zhang. Improving the deep-learning-based differential distinguisher and applications to Simeck. In *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 465–470. IEEE, 2022.

[LW66] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Oper. Res.*, 14(4):699–719, August 1966.

[LWH90] Kevin J. Lang, Alexander H. Waibel, and Geoffrey E. Hinton. A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3:23–43, 1990.

[LXG+14] Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. 38, 09 2014.

[Maa06] Wolfgang Maass. Bounds for the computational power and learning complexity of analog neural nets. *SIAM Journal on Computing*, 26:708–732, 07 2006.

[Maa13] Andrew L. Maas. Rectifier nonlinearities improve neural network acoustic models. 2013.

[Mat94] Mitsuru Matsui. On correlation between the order of s-boxes and the strength of des. In *Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994, Proceedings*, volume 950 of *Lecture Notes in Computer Science*, pages 366–375. Springer, 1994.

[Mat96] Mitsuru Matsui. New structure of block ciphers with provable security against differential and linear cryptanalysis. In Dieter Gollmann, editor, *Fast Software Encryption*, pages 205–218, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[MI15] Hossein Mobahi and John III. A theoretical analysis of optimization by gaussian continuation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 29, 02 2015.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[MM88] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of ECAI*, pages 651–656, 1988.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943. Reprint available online.

[MP69] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, 1969. Scanned version available online.

[MP13] Nicky Mouha and Bart Preneel. A proof that the arx cipher salsa20 is secure against differential cryptanalysis. *IACR Cryptol. ePrint Arch.*, 2013:328, 2013.

[MPCB14] Guido Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. *NIPS 2014*, 02 2014.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. *IACR Cryptol. ePrint Arch.*, 2016:921, 2016.

[MRH04] James Mcclelland, David Rumelhart, and G. Hinton. The appeal of parallel distributed processing. *Parallel Distributed Processing*, 1, 01 2004.

[MSBV93] Patrick McGeer, Jagesh Sanghavi, Robert Brayton, and Alberto Sangiovanni Vincentelli. Espresso-signature: a new exact minimizer for logic functions. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, page 618–624, New York, NY, USA, 1993. Association for Computing Machinery.

[MSS94] W. Maass, G. Schnitger, and E. D. Sontag. *A Comparison of the Computational Power of Sigmoid and Boolean Threshold Circuits*, pages 127–151. Springer US, Boston, MA, 1994.

[MSS99] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, May 1999.

[Mur11] Sean Murphy. The return of the cryptographic boomerang. *Information Theory, IEEE Transactions on*, 57:2517 – 2521, 05 2011.

[MvOV96] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[MW06] Jorge Mor and Zhijun Wu. Global continuation for distance geometry problems. *SIAM Journal on Optimization*, 7:814–836, 07 2006.

[MWGP12] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In Chuan-Kun Wu, Moti Yung, and Dongdai Lin, editors, *Information Security and Cryptology*, pages 57–76, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Nes83] Yurii Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Proceedings of the USSR Academy of Sciences*, 269:543–547, 1983.

[Nes14] Yurii Nesterov. Introductory lectures on convex optimization - a basic course. In *Applied Optimization*, 2014.

[NK93] Kaisa Nyberg and Lars Ramkilde Knudsen. Provable security against differential cryptanalysis. In Ernest F. Brickell, editor, *Advances in Cryptology — CRYPTO' 92*, pages 566–574, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[NW06] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, 2 edition, 2006.

[Par85] D. B. Parker. Learning-logic. Technical Report TR-47, Center for Comp. Research in Economics and Management Sci., MIT, 1985.

[PFL16] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. Version 4.0.5.

[PGCP99] Martin Pelikan, D.E. Goldberg, and Erick Cantu-Paz. Boa: The bayesian optimization algorithm. *BOA: The Bayesian Optimization Algorithm*, pages 525–532, 01 1999.

[PHG17] Stjepan Picek, Annelie Heuser, and Sylvain Guilley. Template attack versus Bayes classifier. *Journal of Cryptographic Engineering*, 7(4):343–351, September 2017.

[PJ92] Boris Polyak and Anatoli Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30:838–855, 07 1992.

[Pol64] Boris Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 12 1964.

[PR91] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[PSC+02] Sangwoo Park, Soo Hak Sung, Seongtaek Chee, E-Joong Yoon, and Jongin Lim. On the security of rijndael-like structures against differential and linear cryptanalysis. In Yuliang Zheng, editor, *Advances in Cryptology — ASIACRYPT 2002*, pages 176–191, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[PSH+18] Stjepan Picek, Ioannis Petros Samiotis, Annelie Heuser, Jaehun Kim, Shivam Bhasin, and Axel Legay. On the Performance of Convolutional Neural Networks for Side-channel Analysis. In *LNCS*, volume 11348 of *LNCS*, pages 157–176, Kanpur, India, December 2018. Springer.

[PSLL03] Sangwoo Park, Soo Hak Sung, Sangjin Lee, and Jongin Lim. Improving the upper bound on the maximum differential and the maximum linear hull probability for spn structures and aes. In Thomas Johansson, editor, *Fast Software Encryption*, pages 247–260, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[R94] Jean-Charles RÉGIN. A filtering algorithm for constraints of difference in csps. In *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*, AAAI'94, page 362–367. AAAI Press, 1994.

[Rao21] C. Rao. *Information and the Accuracy Attainable in the Estimation of Statistical Parameters*, pages 1–13. 07 2021.

[RB08] Matthew Robshaw and Olivier Billet, editors. *The Salsa20 Family of Stream Ciphers*, pages 84–97. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[RHW86a] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. 1986.

[RHW86b] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[Riv91] Ronald L. Rivest. Cryptography and machine learning. In *International Conference on the Theory and Application of Cryptology and Information Security*, 1991.

[RKK19] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond, 2019.

[RLA17] Adrián Ranea, Yunwen Liu, and Tomer Ashur. An easy to use tool for rotational-xor cryptanalysis of arx block ciphers. *Proceedings of the Romanian Academy, Series A*, 18(special issue):307–316, 2017.

[RM51] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400 – 407, 1951.

[RMtPRG86] David E. Rumelhart, James L. McClelland, and the PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, Cambridge, MA, 1986. 2-volume set.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 11 1958.

[Ros60] J. B. Rosen. The gradient projection method for nonlinear programming. part i. linear constraints. *Journal of the Society for Industrial and Applied Mathematics*, 8(1):181–217, 1960.

[Rou22] Loïc Rouquette. *Improving scalability and reusability of differential cryptanalysis models using constraint programming*. Theses, INSA Lyon, November 2022.

[RR22] Adrián Ranea and Vincent Rijmen. Characteristic automated search of cryptographic algorithms for distinguishing attacks (cascada). *IET Information Security*, 16:n/a–n/a, 08 2022.

[Rud91] Walter Rudin. *Functional Analysis*. McGraw–Hill, New York, 2 edition, 1991.

[RYK+14] Adriana Romero, Yere Yere, Samira Ebrahimi Kahou, Nicolas Ballas, Antoine Chassang, Yoesoep Rachmad, and Carlo Gatta. Fitnets: Hints for thin deep nets. *arXiv*, 12 2014.

[Sam59] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[Sch99] Mike Schuster. *On Supervised Learning from Sequential Data with Applications for Speech Recognition*. PhD thesis, University of California, San Diego, 1999.

[SGS15] Rupesh Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. 05 2015.

[Sha49] C. Shannon. Communication theory of secrecy systems. *Bell Systems Techn. Journal*, 28:656–715, 1949.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, L. Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. 12 2017.

[Sku23] Ruslan Skuratovskii. Differential analysis of non-markov ciphers. *WSEAS TRANSACTIONS ON CIRCUITS AND SYSTEMS*, 22:10–15, 02 2023.

[SLJ+14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, D. Erhan, Vincent Vanhoucke, and Andrew

Rabinovich. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2014.

[SM87] Akihiro Shimizu and Shoji Miyaguchi. Fast data encipherment algorithm feal. In *Proceedings of the 6th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'87, page 267–278, Berlin, Heidelberg, 1987. Springer-Verlag.

[SM23] Ayan Sajwan and Girish Mishra. Comparative analysis of resnet and densenet for differential cryptanalysis of speck 32/64 lightweight block cipher. *Cryptology ePrint Archive*, 2023.

[SMDH13] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 244–257, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[SQH19] Ling Song, Xianrui Qin, and Lei Hu. Boomerang connectivity table revisited. application to skinny and aes. *IACR Transactions on Symmetric Cryptology*, 2019(1):118–141, Mar. 2019.

[SS97] João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '96, page 220–227, USA, 1997. IEEE Computer Society.

[SSL+22] Tao Sun, Dongsu Shen, Saiqin Long, Qingyong Deng, and Shiguo Wang. Neural distinguishers on TinyJAMBU-128 and GIFT-64. In *International Conference on Neural Information Processing*, pages 419–431. Springer, 2022.

[SSP03] Patrice Simard, David Steinkraus, and John Platt. Best practices for convolutional neural networks applied to visual document analysis. pages 958–962, 01 2003.

[ST17] Yu Sasaki and Yosuke Todo. New impossible differential search tool from design and cryptanalysis aspects - revealing structural properties of several ciphers. *IACR Cryptol. ePrint Arch.*, 2016:1181, 2017.

[Sta10a] Paul Stankovski. Automated algebraic cryptanalysis. *Proceedings of the ECRYPT Workshop on Tools for Cryptanalysis 2010*, pages 11–11, 2010. Tools for Cryptanalysis 2010 ; Conference date: 22-06-2010 Through 23-06-2010.

[Sta10b] Paul Stankovski. Greedy distinguishers and nonrandomness detectors. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010*, pages 210–226, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Sti06] Douglas R Stinson. Cryptography: Theory and practice. 2006.

[SVLD91] Patrice Simard, Bernard Victorri, Yann LeCun, and John Denker. Tangent prop - a formalism for specifying selected invariances in an adaptive network. In J. Moody, S. Hanson, and R.P. Lippmann, editors, *Advances in Neural Information Processing Systems*, volume 4. Morgan-Kaufmann, 1991.

[SWW21] Ling Sun, Wei Wang, and Meiqin Wang. Accelerating the search of differential and linear characteristics with the SAT method. *IACR Trans. Symmetric Cryptol.*, 2021(1):269–315, 2021.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[SZM21] Heng-Chuan Su, Xuan-Yong Zhu, and Duan Ming. Polytopic attack on round-reduced Simon32/64 using deep learning. In *Information Security and Cryptology: 16th International Conference, Inscrypt 2020, Guangzhou, China, December 11–14, 2020, Revised Selected Papers*, pages 3–20. Springer, 2021.

[TH12] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5—rmsprop: Divide the gradient by a running average of its recent magnitude. Neural Networks for Machine Learning, Coursera, 2012. Lecture notes.

[TH21] Wenqiang Tian and Bin Hu. Deep learning assisted differential cryptanalysis for the lightweight cipher Simon. *KSII Transactions on Internet & Information Systems*, 15, 2021.

[Tib18] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 12 2018.

[Tur50] A. M. Turing. Computing machinery and intelligence. *Mind*, 49(236):433–460, 1950. Version available online.

[UML14] Benigno Uria, Iain Murray, and Hugo Larochelle. A deep and tractable density estimator. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 467–475, Bejing, China, 22–24 Jun 2014. PMLR.

[Vau98] Serge Vaudenay. Provable security for block ciphers by decorrelation. In Michel Morvan, Christoph Meinel, and Daniel Krob, editors, *STACS 98*, pages 249–275, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[VdoDZ+16] Aaron Van den oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. 09 2016.

[VLoASL24] Vesselinux, Cryptology Laboratory of Algorithmics, and University of Luxembourg Security (LACS). Vesselinux/yaarx: Yet Another Toolkit for Analysis of ARX Cryptographic Algorithms. `https://github.com/vesselinux/yaarx`, 2024.

[Wag99] David Wagner. The boomerang attack. In Lars Knudsen, editor, *Fast Software Encryption*, pages 156–170, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Wer70] Paul Werbos. *Applications of advances in nonlinear sensitivity analysis*, volume 38, pages 762–770. 01 1970.

[WH60] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *IRE Wescon Convention Record*, volume 4, pages 96–104, 1960.

[WHH+89] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K.J. Lang. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(3):328–339, 1989.

[WM03] D.R. Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural networks : the official journal of the International Neural Network Society*, 16 10:1429–51, 2003.

[WP19] Haoyang Wang and Thomas Peyrin. Boomerang switch in multiple rounds. application to aes variants and deoxys. *IACR Transactions on Symmetric Cryptology*, 2019(1):142–169, Mar. 2019.

[WSC+16] Yonghui Wu, Mike Schuster, Z. Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason R. Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Gregory S. Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *ArXiv*, abs/1609.08144, 2016.

[WTZ+22] Huijiao Wang, Jiapeng Tian, Xin Zhang, Yongzhuang Wei, and Hua Jiang. Multiple differential distinguisher of SIMECK32/64 based on deep learning. *Security & Communication Networks*, 2022.

[WW21] Gao Wang and Gaoli Wang. Improved differential-ml distinguisher: machine learning based generic extension for differential analysis. In *International Conference on Information and Communications Security*, pages 21–38. Springer, 2021.

[WWS23] Dachao Wang, Baocang Wang, and Siwei Sun. Sat-aided automatic search of boomerang distinguishers for arx ciphers. *IACR Transactions on Symmetric Cryptology*, 2023(1):152–191, Mar. 2023.

[YCL+14] Yere Yere, Jeff Clune, Hod Lipson, Yoesoep Rachmad, and Jason Yosinski. How transferable are features in deep neural networks? *Advances in Neural Information Processing Systems (NIPS)*, 27, 11 2014.

[YK15] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *CoRR*, abs/1511.07122, 2015.

[YK21] Tarun Yadav and Manoj Kumar. Differential-ML distinguisher: Machine learning based generic extension for differential cryptanalysis. In *International Conference on Cryptology and Information Security in Latin America*, pages 191–212. Springer, 2021.

[YLC+09] Yere Yere, Jérôme Louradour, Ronan Collobert, Yoesoep Rachmad, and Jason Weston. Curriculum learning. *Journal of the American Podiatry Association*, 60:6, 06 2009.

[YWD11] Dong Yu, Shizhen Wang, and li Deng. Sequential labeling using deep-structured conditional random fields. *Selected Topics in Signal Processing, IEEE Journal of*, 4:965 – 973, 01 2011.

[YZS+15] Gangqiang Yang, Bo Zhu, Valentin Suder, Mark D. Aagaard, and Guang Gong. The simeck family of lightweight block ciphers. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 307–329, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[ZC88] {Y. T.} Zhou and R. Chellappa. Computation of optical flow using a neural network. pages 71–78, 1988.

[ZK16] neng-fa Zhou and Hakan Kjellerstrand. The picat-sat compiler. volume 9585, pages 48–62, 01 2016.

[ZLWL23] Liu Zhang, Jinyu Lu, Zilong Wang, and Chao Li. Improved differential-neural cryptanalysis for round-reduced Simeck32/64. *Frontiers of Computer Science*, 17(6):176817, 2023.

[ZS14] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *ArXiv*, abs/1410.4615, 2014.

[ZWC23] Liu Zhang, Zilong Wang, and Yindong Chen. Improving the accuracy of differential-neural distinguisher for des, chaskey, and present. *IEICE TRANSACTIONS on Information and Systems*, 106:1240–1243, 2023.

[ZWw22] Liu Zhang, Zilong Wang, and Baocang wang. Improving differential-neural cryptanalysis. Cryptology ePrint Archive, Paper 2022/183, 2022.

[ZZY+21] Runlian Zhang, Mi Zhang, Jiaxu Yan, Yixing Li, Xiaonian Wu, and Lingchen Li. Differential cryptanalysis of TweGIFT-128 based on neural network. In *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*, pages 529–534. IEEE, 2021.