Algebraic Attack on Convolutional Neural Networks with Max Pooling

Zirui Chen[1], Shi Tang[2], Zhengchao Gao[2], Yongjia Su[2], Lingyue Qin[1,3,4(✉)], and Xiaoyang Dong[1,3,4(✉)]

[1] Tsinghua University, Beijing, P. R. China
chenzr25@mails.tsinghua.edu.cn, {qinly,xiaoyangdong}@tsinghua.edu.cn
[2] Shandong University, Qingdao, P. R. China
{shi.tang,chao_qwq,yongjia.su}@mail.sdu.edu.cn
[3] Zhongguancun Laboratory, Beijing, P. R. China
[4] State Key Laboratory of Cryptography and Digital Economy Security, Tsinghua University, Beijing, P. R. China

Abstract. Recovering the weights and biases of deep neural networks (DNNs) via black-box input-output queries — known as parameter extraction attacks — has been extensively studied for ReLU-based fully connected neural networks (FCNNs), but remains unexplored for convolutional neural networks (CNNs) with the max pooling function, a core architecture for computer vision and multimedia processing. The key challenge lies in the CNN's max pooling layer, which introduces an additional non-linearity and hides ReLU critical points, rendering existing FCNN extraction methods inapplicable. To address this gap, we propose the first cryptanalytic extraction attack tailored for CNNs with the max pooling function.

First, we establish an algebraic representation of CNNs, formally proving that CNNs are piecewise linear functions — enabling the extension of linearity-based extraction principles. We then identify two novel types of critical points in CNNs: (1) ReLU-Pooling Critical Points (RPCPs), where a ReLU neuron is at its zero-input critical point and its output is selected by max pooling; and (2) Pooling Switching Points (PSPs), where two neurons within a local receptive field yield identical maximum outputs, triggering a switch in the pooling selection.

Leveraging these critical points, we design complementary extraction techniques: a pattern matching method for RPCPs to recover partial signatures and signs (exploiting the property that unselected pooling neurons have negative outputs), and an internal differential extraction attack for PSPs — inspired by cryptographic internal differential analysis — to recover high-accuracy signatures. Given that PSPs are far more abundant than RPCPs and yield a highly efficient extraction method (verified by experiments), and that RPCPs are indispensable for bias recovery, we integrate both methods: the PSP method enables efficient signature extraction, while a single RPCP recovers the sign and bias.

We evaluate our attack on multiple CNN architectures, including modern adaptations of LeNet-5, trained on random data, MNIST, and CIFAR-10. Experimental results demonstrate that our approach achieves high

extraction accuracy with polynomial query complexity and runtime, even for deep CNN layers. This work fills a research gap in CNN security.

Keywords: Convolutional Neural Networks · Model Extraction

1 Introduction

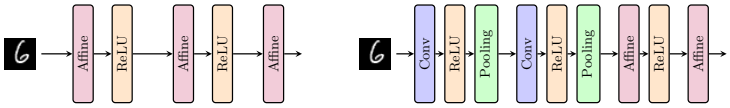
Deep Neural Networks (DNNs) compute a function from inputs to outputs, and the architecture, weights and biases of the network determine the function that is expressed. Treating DNN as a black box and knowing its inputs and outputs, how to extract weights and biases has been a long-standing problem, dating back as far as 30 years ago [4,14]. In 2005, Lowd and Meck proposed an adversarial reverse engineering attack in [36]. Recently, new ideas from research teams in industry and academia have continued to emerge regarding this problem [3,26,38,43,47]. Targeting DNNs with alternating fully-connected linear layers and Rectified Linear Units (ReLU) activation (denoted as *ReLU-based fully connected neural network (FCNN)*), at CRYPTO 2020, Carlini, Jagielski, and Mironov achieved a breakthrough with a *cryptographic differential attack* [9], which requires querying a polynomial number of the raw outputs of FCNN to extract the unsigned weights. However, the sign recovery is a brute-force guessing method, which makes the overall time complexity of Carlini *et al.*'s algorithm exponential [9]. At EUROCRYPT 2024, Canales-Martínez *et al.* [7] introduced the *neuron wiggle technique* to recover the neuron signs in polynomial time. At NIPS 2024, Foerster *et al.* [15] built an end-to-end attack on practical models by combining the methods of Carlini *et al.* [9] and Canales-Martínez et al. [7]. Liu *et al.* [34] introduced more techniques to extract parameters in deep layers. Recently, more model extraction attacks [21,1,41,50] were proposed against different settings of neural networks.

In another direction, still targeting *ReLU-based FCNNs*, where the attackers can only access the final classification labels (*e.g.*, "dog" or "car"), *i.e.*, the *hard-label scenario*, Chen *et al.* [10] first proposed an extraction method requiring a polynomial number of queries but an exponential execution time. At EUROCRYPT 2025, Carlini *et al.* [8] introduced an extraction attack in the hard-label setting with a polynomial number of queries and polynomial time by analyzing the geometric shape of its decision boundaries. Later, Canales-Martínez [6] proposed to recover the output layer, where no ReLU functions exist. In 2025, Ito, Miura, and Todo found that as the depth of the attack-target grows, the attack hypothesis of Carlini *et al.* [8] may become impractical, and they proposed a *cross-layer extraction* method to solve this problem. Until now, all the cryptanalytic extraction attacks focus on FCNNs, where the ReLU-like activation functions are the only non-linear factor in the network. In this paper, we consider the cryptanalytic extraction attacks on the *Convolutional Neural Networks (CNNs)* with the max pooling function for the first time.

1.1 CNNs and the Challenges in the Model Extraction

The concept of convolutional neural networks (CNNs) can be traced back to the concept of receptive fields in the 1960s. In the 1980s, Fukushima *et al.* [17] proposed the concept of the neurocognitive machine. In 1998, LeCun officially proposed convolutional neural networks and designed the well-known LeNet-5 [31] network for handwritten digit recognition. CNNs have been among the most popular and modern neural networks, playing critical roles in the processing of images [29], video [48], time-series signals [49], text data [51], and other domains. Well-known architectures of CNNs include LeNet-5 [31], AlexNet [29], VGG [44], GoogLeNet [46], ResNet [22], DenseNet [23], etc.

While deeper and larger CNN models are continually developed for higher feature extraction performance, shallow architectures remain highly relevant and are actively deployed in scenarios with strict constraints on computational resources, power, or cryptographic overhead, such as edge computing [30,2], low-power wearable devices [28], and privacy-preserving inference [35,18].



(a) FCNN (b) CNN

Fig. 1: FCNN and CNN

In Figure 1, the key difference between the CNN and the ReLU-based FCNN is that the CNN includes a pooling layer (note that the convolutional layer can be transformed into a linear affine layer in Sect. 2.2). In the pooling layer, a pooling function (usually using the max or average function) is used to compress the outputs of all the neurons in a local receptive field, and output a single value. Typically, the max pooling function is widely used in practice due to its unique advantages in feature extraction, anti-interference ability, and prevention of overfitting, which have been verified and applied in many classic CNN papers [31,5,29,44,37]. However, when max pooling function[5] is applied, there will be another non-linear function in the network besides the ReLU function, which brings two problems when applying Carlini *et al.*'s parameter extraction attacks.

– The first problem is that the critical point of the ReLU function will be hidden by max pooling in CNNs. By varying the inputs in the tiny vicinity of the critical point, it is expected that the network's output will change non-linearly as a result of this ReLU transition. However, for CNNs, the outputs of the neurons will be selected by a max pooling function, and only

[5]In fact, different from max pooling, the average pooling does not introduce additional non-linearity to the network, and Carlini *et al.*'s technique [9] can be trivially applied to this case.

the maximum output of the neurons in a local receptive field will be selected as the output. There is a significant probability that the output (whose value is around 0) of the neuron at the critical point will not be selected, and then the network's output will change linearly even though a neuron is at its critical point. This case happens easily when one of the outputs of the other neurons within the same local receptive field is greater than 0.

– The second problem is that the max pooling layer introduces an additional non-linearity, and a similar critical point can be spotted when two outputs of the neurons within a local receptive field are the two maximum values (*e.g.*, $a = b$). By varying the inputs in the tiny vicinity of the critical point, a will exceed b, or b will exceed a. Then, the max pooling layer's selection will switch between a and b. Obviously, this kind of critical point is quite different from Carlini *et al.*'s critical point. How to leverage this new critical point to extract parameters will be a new problem.

1.2 Our Contributions

We introduce an algebraic representation of the CNN, transform all the states and operations into vectors and matrices, and thereby formally make it clear that the CNN is also a piecewise linear function, *i.e.*, within a tiny linear neighborhood of input X, the CNN acts as a linear function, and all operations in the CNN are matrix-vector multiplications. We formally introduce two kinds of critical points, *i.e.*,

- **ReLU-Pooling Critical Point (RPCP):** It is the point where a neuron is at its critical point (input = 0), and subsequent max pooling layers select its output, having a non-linear effect on the CNN's output. Based on RPCPs, Carlini *et al.*'s differential attack is applied to recover the partial signatures (some entries are impossible to recover with a single RPCP, since the ReLU functions suppress negative outputs [9, Section 4.3.2]). We propose a new *pattern matching method* for CNNs to merge multiple partial signatures to get the full one. Furthermore, additional RPCPs are needed to improve the accuracy by applying Carlini *et al.*'s least squares method. Interestingly, the sign recovery is quite easy based on a property of RPCP, *i.e.*, the unselected neurons' outputs by the max pooling function should be negative.
- **Pooling Switching Point (PSP):** It is the point where the outputs of two neurons within a Local Receptive Field of Pooling Layer achieve the same maximum value. To leverage the PSP, we invent a new parameter extraction technique inspired by the cryptographic internal differential attack [12,40], named *internal differential extraction* attack, where the propagation of the difference between different parts of a single PSP is studied through the CNN.

Advantages and limitations of the RPCP and PSP methods:

– According to our experiments, RPCPs are relatively rare in CNNs, since max pooling hides the output of the neuron at the critical point with significant

probability. Therefore, merging partial signatures or improving the accuracy of signatures that require many RPCPs is difficult.

+ The RPCP method can easily recover the sign and bias based on its property.
+ According to our experiments, PSPs are more common in CNNs than RPCPs. Based on the *internal differential extraction* attack, the signature (without bias) can be extracted with higher accuracy, shorter runtime, and fewer model queries than using RPCP methods.
− The PSP method cannot recover the bias.

Unified Model Combining RPCP and PSP. Given the above advantages and limitations, we integrate the RPCP and PSP methods into a unified parameter extraction model, where the PSP method is used to recover the signature (without bias) and sign, while only a single (or a few) RPCPs are needed to recover the bias. Besides, we introduce a method named the *targeted heuristic search strategy for RPCPs*, to handle the case that even one RPCP is hard to find. This method leverages the signature recovered by PSPs to directly detect the points close to an RPCP.

End-to-end Experiments. We perform practical parameter extractions on several CNNs, including the popular LeNet-5 with modern architectures. Taking the full $(m+n)$-deep CNN (m Convolutional Rounds and n FCNN Rounds) as a black box, with the inputs and raw outputs, we present two kinds of experiments:

- We recover all parameters of the m Convolutional Rounds of the first 5 CNNs (including the modern LeNet-5) in Table 1, where $(m+n)$ means only the parameters of the m Convolutional Rounds are recovered.
- We give an experiment recovering the full parameters of an $(m+n)$-deep CNN, which is a reduced version of LeNet-5 with 2 standard Convolutional Rounds, 1 reduced FCNN Round (the modern LeNet-5 has 2 FCNN Rounds), and an output layer (*i.e.*, 3 hidden layers). To our knowledge, this matches the depth of the deepest architecture - 3 hidden layers [9], among existing end-to-end black-box full-model parameter extraction works.
  We also attack the standard modern LeNet-5 (specified in Sect. B.1), extracting the full Convolutional Block and 93.33% weights of FCNN Round 1 (incomplete due to dead neurons and persistent neurons according to [25]).
  We present all results and the comparisons with Carlini *et al.*'s work [9] and Foerster *et al.*'s work [15] in the last four rows of Table 1.

All the source codes and experiments can be found via:

https://github.com/czr-eric/Algebraic-Attack-on-CNN.git

Related and Concurrent Works. At ACISP 2026, Sun *et al.* [45] proposed a cryptanalytic extraction of convolutional neural networks with **average pooling**. However, on the commonly used max pooling function, they claimed that [45, Section 5]:

> "*However, the non-linear nature of max pooling layers, which only retains the maximum value within each region, inherently leads to significant information loss and makes the operation irreversible. As a result, reconstructing the original input or parameters from these layers presents considerable difficulties, making the handling of such layers a persistent challenge for future research.*"

This paper addresses the challenges posed by the max-pooling function and, for the first time, recovers the parameters of a CNN employing max pooling layers.

Recently, Liu *et al.* [33] also proposed a model extraction attack on max-pooling CNNs. They introduced a distinct localization strategy for critical points based on receptive-field analysis, interpreted the max-pooling CNN as a structural generalization of the ReLU setting, and extended the geometric extraction framework accordingly. However, we approach the attack from an algebraic perspective, introducing novel techniques such as the pattern matching method, internal differential [40] extraction, *etc.*

2 Preliminaries

2.1 Basic Definitions and Notations

- **FCNN**: Fully Connected Neural Network (FCNN) in Figure 1.
- **CNN**: Convolutional Neural Network (CNN) in Figure 1.
- **ReLU**: Rectified Linear Unit.
- **FCNN Round**: Including a linear function $f^{(k)}$ called fully-connected layer, and a non-linear function σ (component-wise ReLU function).
- **Convolutional Round**: Including a convolutional layer $f_c^{(k)}$, a non-linear activation layer $\sigma_c^{(k)}$ (ReLU), and a pooling layer $\rho^{(k)}$ (max pooling).
- **$(\mathbf{m}+\mathbf{n})$-Deep Convolutional Neural Network (CNN)**: A CNN with m Convolutional Rounds and n FCNN Rounds, as well as an output layer with a fully-connected linear layer.
- $(\mathsf{c}_{\mathbf{in}}^{(\mathbf{k})}, \mathsf{c}_{\mathbf{out}}^{(\mathbf{k})}, \mathbf{h}_{\mathbf{c}}^{(\mathbf{k})}, \mathbf{w}_{\mathbf{c}}^{(\mathbf{k})})$: Convolutional layer with $\mathsf{c}_{in}^{(k)}$ input and $\mathsf{c}_{out}^{(k)}$ output channels, and the size of kernel matrices is $h_c^{(k)} \times w_c^{(k)}$.
- $\mathbf{X}^{(\mathbf{k})}, \mathbf{Y}^{(\mathbf{k})}, \mathbf{Z}^{(\mathbf{k})}$: Input vectors of the operations $f_c^{(k)}$, $\sigma_c^{(k)}$, $\rho^{(k)}$ in the k-th Convolutional Round, $k \geq 1$. $X^{(k)} \in \mathbb{R}^{d^{(k)}}$, $Y^{(k)}$, $Z^{(k)} \in \mathbb{R}^{d_f^{(k)}}$, since $\sigma_c^{(k)}$ does not change the dimension. $X^{(k)}[i]$ is the i-th $(0 \leq i < d^{(k)})$ entry of $X^{(k)}$.
- $\mathbf{A}^{(\mathbf{k})}$: Matrix whose entry in i-th row and j-th is $A_{i,j}^{(k)}$, i, j start from 0.
- **RPCP, PSP, FCP**: ReLU-Pooling Critical Point, Pooling Switching Point, Critical points in the Fully-connected Block, respectively.
- **LRF-C**, Local Receptive Field of Convolutional Layer: A sub-matrix of $IN^{(k)}$, which is multiplied by *Convolution Kernel Matrix* $C^{(k)}$ to produce each entry in $Y^{(k)}$. The LRF-C will be different for different elements of $Y^{(k)}$. Let $Y^{(k)}[t]$ correspond to the t-th $(0 \leq t \leq d_f^{(k)} - 1)$ LRF-C.

– **LRF-P**, Local Receptive Field of Pooling Layer: A sub-matrix of $Z^{(k)}$, which is processed by a pooling function to produce each entry of $X^{(k+1)}$. The LRF-P will be different for different elements of $X^{(k+1)}$. Let $X^{(k+1)}[t]$ correspond to the t-th $(0 \leq t \leq d^{(k+1)} - 1)$ LRF-P.

Table 1: Experiments on different CNNs. Convolution kernel dimensions are denoted as $(\mathsf{c}_{in}^{(k)}, \mathsf{c}_{out}^{(k)}, h_c^{(k)}, w_c^{(k)})$, with stride $s_c^{(k)} = 1$. Pooling kernels are 2×2 with stride $s_\rho^{(k)} = 2$. The accuracy term $(\varepsilon, 0)$ is defined in Sect. 2.3, and $\max|\theta - \hat{\theta}|$ directly measures the maximum extraction error of model parameters.

Models $(m+n)$	**Trainset**	**Architecture**† $d^{(k)}$–$d_f^{(k)}$–$d^{(k+1)}$	**Kernel**	**Queries**	$(\varepsilon, 0)$	$\max\|\theta - \hat{\theta}\|$
(**1**+1)	Random	C_1: 64–36–9	(1,1,3,3)	$2^{9.98}$	$2^{-43.19}$	$2^{-47.52}$
(**2**+1)	MNIST	C_1: 1024–784–196 C_2: 196–100–25	(1,1,5,5) (1,1,5,5)	$2^{15.06}$	$2^{-30.20}$	$2^{-34.65}$
(**2**+2)*	MNIST	C_1: 1024–4704–1176 C_2: 1176–1600–400	(1,6,5,5) (6,16,5,5)	$2^{27.29}$	$2^{-23.38}$	$2^{-27.31}$
(**3**+1)^{p_2}	MNIST	C_1: 1024–1024–256 C_2: 256–256–64 C_3: 64–64–16	(1,1,5,5) (1,1,5,5) (1,1,5,5)	$2^{16.90}$	$2^{-27.54}$	$2^{-31.15}$
(**2**+2)^{p_1}	CIFAR10	C_1: 3072–3072–768 C_2: 768–1024–256	(3,3,3,3) (3,4,3,3)	$2^{18.37}$	$2^{-29.13}$	$2^{-36.19}$
(**2+1**)‡	MNIST	C_1: 1024–4704–1176 C_2: 1176–1600–400 F: 400–20–10	(1,6,5,5) (6,16,5,5) –	$2^{27.65}$	$2^{-22.06}$	$2^{-27.14}$
(**2+2**)*♣	MNIST	**C_1**: 1024–4704–1176 **C_2**: 1176–1600–400 F: **400–120**–84–10	(1,6,5,5) (6,16,5,5) –	$2^{27.31}$	–	$2^{-26.03}$
(0+**3**) [9]	MNIST	F: 40–20–10–10–1	–	$2^{17.8}$	$2^{-23.4}$	$2^{-27.1}$
(0+**3**) [15] ♠	MNIST	F: 784–16–**16–16**–1	–	$2^{25.4}$	–	–

†: C_i indicates Convolutional Round i; F denotes FCNN Block with $d^{(1)} - \cdots - d^{(n+1)}$.
*: LeNet-5 modern model, specified in Sect. B.1.
p_1, p_2: p_i indicates $pad^{(k)} = i$ defined as "same padding" in Supplementary Material F; otherwise, $pad^{(k)} = 0$.
‡: Reduced version of LeNet-5 with only one FCNN Round.
♣: The red bold layer in Architecture indicates end-to-end extracting of the full Convolutional Block and 93.33% weights of FCNN Round 1 (not full due to existing dead neurons and persistent neurons according to [25]).
♠: Red bold layer in Architecture indicates extracting the single layer, with prior layers' extractions assumed to be correct.

2.2 Algebraic View of CNN

To facilitate the understanding of the algebraic formulations in this section, we provide a concrete toy example of a CNN in Supplementary Material A.

Previous works [9,7,10,8,11] concentrate on the Fully Connected Neural Network (FCNN) in Figure 1, which is composed of a sequence of functions alternating between linear functions $f^{(k)} : \mathbb{R}^{d^{(k)}} \mapsto \mathbb{R}^{d^{(k+1)}}$ ($k \geq 1$, called fully-connected layers), and a non-linear function $\sigma^{(k)}$ (component-wise ReLU function):

$$\mathcal{F}_{\text{fc}} = f^{(n+1)} \circ \sigma^{(n)} \circ f^{(n)} \circ \sigma^{(n-1)} \circ \cdots f^{(2)} \circ \sigma^{(1)} \circ f^{(1)}. \tag{1}$$

In Figure 1, a typical CNN architecture consists of an initial sequence of convolutional, non-linear ReLU, and pooling layers (this paper focuses on the max pooling), followed by a series of fully-connected layers, which are referred to as the *Convolutional Block* and the *Fully-connected Block*, respectively.

Definition 1. ($(m+n)$-Deep Convolutional Neural Network) *A $(m+n)$-deep Convolutional Neural Network (CNN) is a function $\mathcal{F}_\theta$ parameterized by θ that takes inputs from an input space $\mathbb{R}^{d^{(1)}}$ and returns values in an output space $\mathbb{R}^{d^{(m+n+2)}}$. The function $\mathcal{F}_\theta : \mathbb{R}^{d^{(1)}} \to \mathbb{R}^{d^{(m+n+2)}}$ is composed of two sequential blocks: the first block (Convolutional Block, denoted as $\mathcal{F}_{conv}$) consists of m rounds of alternating convolutional layers $f_c^{(k)}$, non-linear activation layers $\sigma_c^{(k)}$, and pooling layers $\rho^{(k)}$ $(1 \leq k \leq m)$; the second block (Fully-connected Block, denoted as $\mathcal{F}_{fc}$) consists of n rounds of alternating linear layers $f^{(k)}$ and activation functions $\sigma^{(k)}$. Then,*

$$\mathcal{F}_\theta = \mathcal{F}_{fc} \circ \mathcal{F}_{conv} = \underbrace{f^{(n+1)} \circ \sigma^{(n)} \circ \cdots \circ \sigma^{(1)} \circ f^{(1)}}_{\textit{Fully-connected Block, } \mathcal{F}_{fc}} \circ \underbrace{\rho^{(m)} \circ \sigma_c^{(m)} \circ f_c^{(m)} \circ \cdots \circ \rho^{(1)} \circ \sigma_c^{(1)} \circ f_c^{(1)}}_{\textit{Convolutional Block, } \mathcal{F}_{conv}}. \tag{2}$$

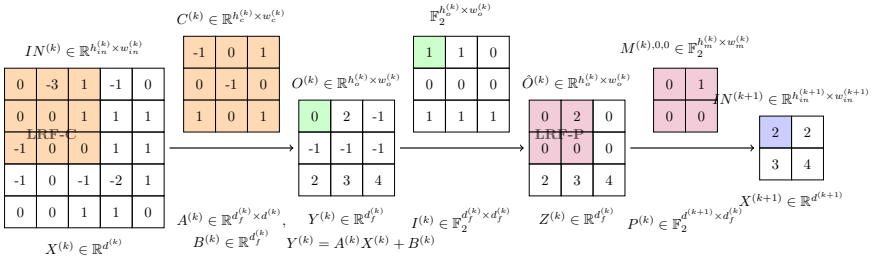


Fig. 2: Symbolic View of CNN

Convolutional Layers. The CNN's inputs are formulated as matrices. The convolution operation of the convolutional layer $f_c^{(k)}$ employs a sliding window mechanism, where a small matrix of learnable parameters (the *Convolution Kernel Matrix*) traverses the input from left to right, top to bottom. At each position,

the sum of element-wise products between the Convolution Kernel Matrix and the overlapping local input patch (*Local Receptive Field of Convolutional Layer*, abbreviated as *LRF-C*), is computed to generate a single output value. A shared bias is then added to the result at each position to constitute the output matrix (*Feature Map*) of the convolutional layer.

In the algebraic view as shown in Figure 2, the input matrix of $f_c^{(k)}$ is denoted as $IN^{(k)}$ with dimension $h_{in}^{(k)} \times w_{in}^{(k)}$, so $d^{(k)} = h_{in}^{(k)} \cdot w_{in}^{(k)}$. The convolution kernel matrix is defined by the shape $(\mathsf{c}_{in}^{(k)}, \mathsf{c}_{out}^{(k)}, h_c^{(k)}, w_c^{(k)})$, where $\mathsf{c}_{in}^{(k)}$ and $\mathsf{c}_{out}^{(k)}$ denote the number of input and output channels, and $h_c^{(k)}, w_c^{(k)}$ denote the height and width of the convolution kernel matrix $C^{(k)}$, respectively.

For briefness, we will first consider the simple case of single input and output channel, *i.e.*, $\mathsf{c}_{in}^{(k)} = \mathsf{c}_{out}^{(k)} = 1$, and the case of multiple channels will be analyzed in Supp. B. Given a convolution stride $s_c^{(k)}$, the $h_o^{(k)} \times w_o^{(k)}$ output matrix or the *feature map* of $f_c^{(k)}$ is denoted as $O^{(k)}$, where

$$h_o^{(k)} = \left\lfloor \frac{h_{in}^{(k)} - h_c^{(k)}}{s_c^{(k)}} \right\rfloor + 1, \quad w_o^{(k)} = \left\lfloor \frac{w_{in}^{(k)} - w_c^{(k)}}{s_c^{(k)}} \right\rfloor + 1. \tag{3}$$

For briefness, we let the convolution stride $s_c^{(k)} = 1$. The input of $f_c^{(k)}$ can be a $h_{in}^{(k)} \times w_{in}^{(k)}$ $IN^{(k)}$ or a $d^{(k)}$-dimension vector $X^{(k)}$. Denote $IN^{(k)} = [a_{i,j}^{(k)}]_{h_{in}^{(k)} \times w_{in}^{(k)}}$ and the i-th $(0 \le i \le h_{in}^{(k)} - 1)$ row $IN_i^{(k)} = (a_{i,0}^{(k)}, a_{i,1}^{(k)}, \cdots, a_{i,w_{in}^{(k)}-1}^{(k)})$. Then the vector $X^{(k)} = (IN_0^{(k)}, IN_1^{(k)}, \cdots, IN_{h_{in}^{(k)}-1}^{(k)})^\mathsf{T}$.

Denote the kernel matrix $C^{(k)} = [c_{i,j}^{(k)}]_{h_c^{(k)} \times w_c^{(k)}}$, where $C_i^{(k)} = (c_{i,0}^{(k)}, c_{i,1}^{(k)}, \cdots, c_{i,w_c^{(k)}-1}^{(k)})$. Then convolution operation computes the output matrix $O^{(k)}$ as

$$O_{i,j}^{(k)} = \sum_{p=0}^{h_c^{(k)}-1} \sum_{q=0}^{w_c^{(k)}-1} IN_{i+p,j+q}^{(k)} C_{p,q}^{(k)} + b^{(k)}, \tag{4}$$

where the bias $b^{(k)}$ is the same for all the outputs. In vector form, the output matrix $O^{(k)}$ will be a $(d_f^{(k)} = h_o^{(k)} \cdot w_o^{(k)})$-dimension vector $Y^{(k)} = (O_0^{(k)}, \cdots, O_{h_o^{(k)}-1}^{(k)})^\mathsf{T}$.

Definition 2. (Convolutional Matrix) *Let* $\mathsf{c}_{in}^{(k)} = \mathsf{c}_{out}^{(k)} = s_c^{(k)} = 1$. *In the k-th convolutional layer, given a convolution kernel matrix* $C^{(k)}$, *the function* $f_c^{(k)} : \mathbb{R}^{d^{(k)}} \to \mathbb{R}^{d_f^{(k)}}$, *is defined as* $f_c^{(k)}(X^{(k)}) = A^{(k)}X^{(k)} + B^{(k)}$, *where* $X^{(k)} \in \mathbb{R}^{d^{(k)}}$ *is the input vector.* $A^{(k)} = (A_0^{(k)\mathsf{T}}, A_1^{(k)\mathsf{T}}, \cdots, A_{d_f^{(k)}-1}^{(k)\mathsf{T}})^\mathsf{T} \in \mathbb{R}^{d_f^{(k)} \times d^{(k)}}$, *whose i-th row* $(0 \le i \le d_f^{(k)} - 1)$ *is:*

$$A_i^{(k)} = (\underbrace{0, \cdots, 0}_{w_{in}^{(k)} \cdot \left\lfloor \frac{i}{w_o^{(k)}} \right\rfloor}, \underbrace{\underbrace{0, \cdots, 0}_{i \bmod w_o^{(k)}}, C_0^{(k)}, \underbrace{0, \ldots, 0}_{padding}}_{w_{in}^{(k)}}, \cdots, \underbrace{\underbrace{0, \cdots, 0}_{i \bmod w_o^{(k)}}, C_{h_c^{(k)}-1}^{(k)}, \underbrace{0, \ldots, 0}_{padding}}_{w_{in}^{(k)}}, \underbrace{0, \cdots, 0}_{padding}), \tag{5}$$

and $B^{(k)} = (b^{(k)}, b^{(k)}, \cdots, b^{(k)})^T \in \mathbb{R}^{d_f^{(k)}}$ *(the bias for each neuron is the same).*

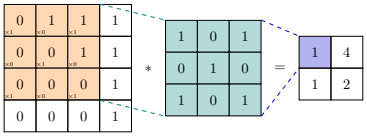


Fig. 3: Example of Convolution Operation (bias $b^{(k)}$=0)

Consider a concrete scenario with $\mathsf{c}_{in}^{(k)} = \mathsf{c}_{out}^{(k)} = s_c^{(k)} = 1$ as shown in Figure 3, where the input matrix $IN^{(k)}$ has dimensions $h_{in}^{(k)} \times w_{in}^{(k)} = 4 \times 4$, i.e., $d^{(k)} = 16$. Given a $h_c^{(k)} \times w_c^{(k)} = 3 \times 3$ kernel matrix $C^{(k)} = [c_{i,j}^{(k)}]_{3\times 3}$, the output matrix $O^{(k)}$ is of dimension $h_o^{(k)} \times w_o^{(k)} = 2 \times 2$, *i.e.*, $d_f^{(k)} = 4$. According to Def. 2,

$$A^{(k)} = \begin{pmatrix} c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 & 0 & 0 & 0 & 0 \\ 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 \\ 0 & 0 & 0 & 0 & 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} \end{pmatrix}.$$

Non-linear Activation Layer. The k-th activation layer $\sigma_c^{(k)}$ of a CNN consists of a set of parallel $d_f^{(k)}$ non-linear activation functions $\mathrm{ReLU}(x_i) = \max\{x_i, 0\}$ for $i = 0, \cdots, d_f^{(k)} - 1$. The input of $\sigma_c^{(k)}$ layer is the $h_o^{(k)} \times w_o^{(k)}$ matrix $O^{(k)}$, and we denote its output as $\hat{O}^{(k)}$ with the same dimension. In vector form, the input is denoted as $Y^{(k)} = (y_0, y_1, \cdots, y_{d_f^{(k)}-1})^{\mathsf{T}}$. Then $\sigma_c^{(k)}$ layer can be expressed as a matrix $I^{(k)}$ multiplied by $Y^{(k)}$, *i.e.*, $Z^{(k)} = (\hat{O}_0^{(k)}, \cdots, \hat{O}_{h_o^{(k)}-1}^{(k)})^{\mathsf{T}} = \sigma_c^{(k)}(Y) = I^{(k)}Y^{(k)}$, where

$$I^{(k)} = \begin{pmatrix} \tau_0^{(k)} & 0 & \cdots & 0 \\ 0 & \tau_1^{(k)} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \tau_{d_f^{(k)}-1}^{(k)} \end{pmatrix}, \ 0 \le i \le d_f^{(k)} - 1, \ \tau_i^{(k)} = \begin{cases} 1, & \text{if} \quad y_i > 0, \\ 0, & \text{if} \quad y_i \le 0. \end{cases} \quad (6)$$

Pooling Layers. In the pooling layer, a small fixed-size window (analogous to a convolution kernel, and referred to as a pooling kernel) slides across a feature map (input matrix $\hat{O}^{(k)}$) with a predefined stride. The region of the input matrix covered by the pooling window is also referred to as the *Local Receptive Field of Pooling Layer* (*LRF-P* for short). For each window position, it applies a pooling function (usually use the max or average function) to compress the

contents in the LRF-P into a single value, producing a dimension reduced output matrix (denoted as $IN^{(k+1)}$). Unlike convolution kernels, pooling kernels are parameter-free (no learnable weights and biases) — their behavior is determined solely by size, stride, and compressing rule. Usually, the max pooling function is more widely used than the average pooling in practice due to its unique advantages in feature extraction, anti-interference ability, and prevention of overfitting, which have been verified and applied in many classic CNN papers [31,5,29,44,37]. Therefore, this paper focuses on the max pooling function. Algebraically, we reformulate the operation of the pooling layer as follows.

In the pooling layer $\rho^{(k)}$, we assume that the max pooling kernel is applied to an LRF-P of size $h_m^{(k)} \times w_m^{(k)}$ with stride $s_\rho^{(k)}$. The input is $h_o^{(k)} \times w_o^{(k)}$ matrix $\hat{O}^{(k)}$ in Figure 2. The output is denoted as a $h_\rho^{(k)} \times w_\rho^{(k)}$ matrix $IN^{(k+1)}$, where:

$$h_\rho^{(k)} = \left\lfloor \frac{h_o^{(k)} - h_m^{(k)}}{s_\rho^{(k)}} \right\rfloor + 1, \quad w_\rho^{(k)} = \left\lfloor \frac{w_o^{(k)} - w_m^{(k)}}{s_\rho^{(k)}} \right\rfloor + 1.$$

In particular, when the pooling kernel size is $h_m^{(k)} \times w_m^{(k)} = 2 \times 2$ and the stride is $s_\rho^{(k)} = 2$, the windows partition the input into disjoint LRF-Ps, and the calculation of the output size reduces to:

$$h_\rho^{(k)} = \left\lfloor \frac{h_o^{(k)}}{2} \right\rfloor, \quad w_\rho^{(k)} = \left\lfloor \frac{w_o^{(k)}}{2} \right\rfloor.$$

The entry of the i-th row and j-th column ($0 \le i \le h_\rho^{(k)} - 1$, $0 \le j \le w_\rho^{(k)} - 1$) of the output matrix $IN^{(k+1)}$ takes the maximum value of the elements in the $(i \cdot w_\rho^{(k)} + j)$-th LRF-P of $\hat{O}^{(k)}$, whose indexes are denoted as $\Omega_{i,j} = \{(i \cdot s_\rho^{(k)} + h,\ j \cdot s_\rho^{(k)} + w),\ 0 \le h \le h_m^{(k)} - 1, 0 \le w \le w_m^{(k)} - 1\}$. Then,

$$IN_{i,j}^{(k+1)} = \max\{\hat{O}_{\Omega_{i,j}}^{(k)}\}. \tag{7}$$

When the input matrix $\hat{O}^{(k)}$ is given, the max pooling operation can be expressed as multiplying a series of Boolean matrices $M^{(k),i,j} \in \mathbb{F}_2^{h_m^{(k)} \times w_m^{(k)}}$ ($0 \le i \le h_\rho^{(k)} - 1$, $0 \le j \le w_\rho^{(k)} - 1$) by different $h_m^{(k)} \times w_m^{(k)}$ LRF-P of $\hat{O}^{(k)}$,

$$IN_{i,j}^{(k+1)} = \sum_{0 \le h \le h_m^{(k)} - 1, 0 \le w \le w_m^{(k)} - 1} M_{\{h,w\}}^{(k),i,j} \cdot \hat{O}_{\{i \cdot s_\rho^{(k)} + h, j \cdot s_\rho^{(k)} + w\}}^{(k)}, \tag{8}$$

In the $(i \cdot w_\rho^{(k)} + j)$-th LRF-P of $\hat{O}^{(k)}$, let $\Lambda_{i,j}$ be the set of all index pairs that achieve the maximum value:

$$\Lambda_{i,j} = \left\{ (h, w) \mid \hat{O}_{\{i \cdot s_\rho^{(k)} + h, j \cdot s_\rho^{(k)} + w\}}^{(k)} = \max\{\hat{O}_{\Omega_{i,j}}^{(k)}\} \right\}. \tag{9}$$

To ensure exactly one neuron is selected from the LRF-P, let $(h^*, w^*) \in \Lambda_{i,j}$ be an arbitrarily chosen index pair. The elements of $M^{(k),i,j}$ are defined as:

$$M_{\{h,w\}}^{(k),i,j} = \begin{cases} 1, & \text{if } (h, w) = (h^*, w^*), \\ 0, & \text{otherwise}, \end{cases} \tag{10}$$

with $0 \le h \le h_m^{(k)} - 1$, $0 \le w \le w_m^{(k)} - 1$. Note that any arbitrary choice of $(h^*, w^*) \in \Lambda_{i,j}$ yields a functionally equivalent matrix $M^{(k),i,j}$ that produces the exact same pooling output $IN_{i,j}^{(k+1)}$.

In vector form, the input and output vector of pooling layer $\rho^{(k)}$ are $Z^{(k)} = (\hat{O}_0^{(k)}, \cdots, \hat{O}_{h_o^{(k)}-1}^{(k)})^\intercal \in \mathbb{R}^{d_f^{(k)}=h_o^{(k)} \cdot w_o^{(k)}}$ and $X^{(k+1)} = (IN_0^{(k+1)}, IN_1^{(k+1)}, \cdots, IN_{h_\rho^{(k)}-1}^{(k+1)})^\intercal \in \mathbb{R}^{d^{(k+1)}}$, $d^{(k+1)} = h_\rho^{(k)} \cdot w_\rho^{(k)}$.

Definition 3. (Boolean Pooling Matrix) *There exists a $d^{(k+1)} \times d_f^{(k)}$ Boolean pooling matrix $P^{(k)}$ so that $\rho^{(k)} : \mathbb{R}^{d_f^{(k)}} \to \mathbb{R}^{d^{(k+1)}}$ satisfying $X^{(k+1)} = \rho^{(k)}(Z^{(k)}) = P^{(k)} \cdot Z^{(k)}$. Assuming the LRF-Ps are disjoint and square (*i.e., *$s_\rho^{(k)} = w_m^{(k)} = h_m^{(k)}$*), then the max pooling operation on the r-th LRF-P can be defined as the r-th row ($0 \le r \le d^{(k+1)} - 1$) of $P^{(k)}$:*

$$P_r^{(k)} = (\underbrace{0, \cdots, 0}_{w_o^{(k)} \cdot i \cdot s_\rho^{(k)}}, \underbrace{\underbrace{0, \cdots, 0}_{j \cdot s_\rho^{(k)}}, M_0^{(k),i,j}, \underbrace{0, \ldots, 0}_{padding}}_{w_o^{(k)}}, \cdots, \underbrace{\underbrace{0, \cdots, 0}_{j \cdot s_\rho^{(k)}}, M_{h_m^{(k)}-1}^{(k),i,j}, \underbrace{0, \ldots, 0}_{padding}}_{w_o^{(k)}}, \underbrace{0, \cdots, 0}_{padding}), \tag{11}$$

where $i = \left\lfloor \frac{r}{w_\rho^{(k)}} \right\rfloor$, $j = r \bmod w_\rho^{(k)}$.

Obviously, similar to the matrix of $I^{(k)}$ for $\sigma_c^{(k)}$ layer, the pooling matrix $P^{(k)}$ varies with the input $X^{(1)}$. For example, the input matrix $\hat{O}^{(k)}$ has dimensions $h_o^{(k)} \times w_o^{(k)} = 4 \times 4$, i.e., $d_f^{(k)} = 16$, the max pooling kernel is of $h_m^{(k)} \times w_m^{(k)} = 2 \times 2$ with stride $s_\rho^{(k)} = 2$. The output matrix $IN^{(k+1)}$ will have dimension $h_\rho^{(k)} \times w_\rho^{(k)} = 2 \times 2$, i.e., $d^{(k+1)} = 4$. The $d^{(k+1)} \times d_f^{(k)} = 4 \times 16$ pooling matrix $P^{(k)} = \begin{pmatrix} P_{0,0}^{(k)} & \mathbf{0}_{2\times 8} \\ \mathbf{0}_{2\times 8} & P_{1,1}^{(k)} \end{pmatrix}$, where $\mathbf{0}_{2\times 8}$ is 2×8 zero matrix, and

$$P_{0,0}^{(k)} = \begin{pmatrix} M_{0,0}^{(k),0,0} & M_{0,1}^{(k),0,0} & 0 & 0 & M_{1,0}^{(k),0,0} & M_{1,1}^{(k),0,0} & 0 & 0 \\ 0 & 0 & M_{0,0}^{(k),0,1} & M_{0,1}^{(k),0,1} & 0 & 0 & M_{1,0}^{(k),0,1} & M_{1,1}^{(k),0,1} \end{pmatrix},$$

$$P_{1,1}^{(k)} = \begin{pmatrix} M_{0,0}^{(k),1,0} & M_{0,1}^{(k),1,0} & 0 & 0 & M_{1,0}^{(k),1,0} & M_{1,1}^{(k),1,0} & 0 & 0 \\ 0 & 0 & M_{0,0}^{(k),1,1} & M_{0,1}^{(k),1,1} & 0 & 0 & M_{1,0}^{(k),1,1} & M_{1,1}^{(k),1,1} \end{pmatrix}.$$

Definition 4. (Fully-connected Layer) *The k-th fully-connected layer within the Fully-connected Block of CNN is a function $f^{(k)} : \mathbb{R}^{d^{(k)}} \to \mathbb{R}^{d^{(k+1)}}$ given by an affine transformation:*

$$f^{(k)}(X) = A_{fc}^{(k)} X_{fc}^{(k)} + B_{fc}^{(k)}, \tag{12}$$

where $X_{fc}^{(k)} \in \mathbb{R}^{d^{(k)}}$ is the input vector. The weight matrix $A_{fc}^{(k)} \in \mathbb{R}^{d^{(k+1)} \times d^{(k)}}$ and the bias vector $B_{fc}^{(k)} \in \mathbb{R}^{d^{(k+1)}}$ consist of floating-point numbers.

The matrix for the non-linear layer σ (ReLU functions) of the Fully-connected Block can be similarly defined by Eq. (6). Given $X^{(1)} = x \in \mathbb{R}^{d^{(1)}}$, the Convolutional Rounds and FCNN Rounds of CNN "collapse" into an affine transformation. Assume that the function of the m Convolutional Rounds is $\mathcal{F}_{\text{conv}}$, and the function of the n FCNN Rounds is $\mathcal{F}_{\text{fc}}$, then

$$\begin{aligned} y = \mathcal{F}_{\text{conv}}(x) &= P^{(m)}(I^{(m)}(A^{(m)} \cdots (P^{(1)}(I^{(1)}(A^{(1)}x + B^{(1)}))) \cdots + B^{(m)})) \\ &= P^{(m)} I^{(m)} A^{(m)} \cdots P^{(2)} I^{(2)} A^{(2)} P^{(1)} I^{(1)} A^{(1)} x + \beta_1 \\ &= \Gamma_1 \cdot x + \beta_1. \end{aligned} \tag{13}$$

According to [7, Section 3.1], given y, $\mathcal{F}_{\text{fc}}$ collapses into

$$\mathcal{F}_{\text{fc}}(y) = \Gamma_2 \cdot y + \beta_2. \tag{14}$$

Consequently, the full CNN "collapses" into

$$\mathcal{F}_\theta(x) = \Gamma_2(\Gamma_1 \cdot x + \beta_1) + \beta_2 = \Gamma_2 \cdot \Gamma_1 \cdot x + \Gamma_2\beta_1 + \beta_2. \tag{15}$$

Definition 5. (Linear Neighborhood) *Given an input $x \in \mathbb{R}^{d^{(1)}}$ with the corresponding output of the CNN computed by Eq. (15), the linear neighborhood of x is defined as the set*

$$\{u \in \mathbb{R}^{d^{(1)}} | \mathcal{F}_\theta(u) = \Gamma_2 \cdot \Gamma_1 \cdot u + \Gamma_2\beta_1 + \beta_2\},$$

i.e., the same affine transformation is used for u and x to compute the output of the CNN.

If we make a change of Δ to the input x, and $x + \Delta$ remains within the linear neighborhood of x, we can observe the corresponding change of the output

$$\mathcal{F}_\theta(x+\Delta) - \mathcal{F}_\theta(x) = \Gamma_2 \cdot \Gamma_1 \cdot (x+\Delta) + \Gamma_2\beta_1 + \beta_2 - (\Gamma_2 \cdot \Gamma_1 \cdot x + \Gamma_2\beta_1 + \beta_2) = \Gamma_2 \cdot \Gamma_1 \cdot \Delta.$$

Definition 6. (Layer Merging) *We focus on the Convolutional Block of the $(m+n)$ network. We assume that we have complete knowledge of the first $k-1$ layers of the Convolutional Block, and we are currently recovering layer k. Let F_x^{k-1} and G_x^{k+1} represent, respectively, the fully recovered and non-recovered parts of the CNN.*

$$\mathcal{F}_\theta = \underbrace{f^{(n+1)} \circ \cdots \circ \sigma^{(1)} \circ f^{(1)} \circ \rho^{(m)} \circ \cdots \circ f_c^{(k+1)}}_{G_x^{k+1}} \circ \rho^{(k)} \circ \sigma_c^{(k)} \circ f_c^{(k)} \circ \underbrace{\rho^{(k-1)} \circ \cdots \circ f_c^{(1)}}_{F_x^{k-1}}.$$

Given $X^{(1)} \in \mathbb{R}^{d^{(1)}}$, G_x^{k+1} and F_x^{k-1} become $G_x^{k+1}(X^{(k+1)}) = G^{k+1}X^{(k+1)} + B^{k+1}$ and $F_x^{k-1}(X^{(1)}) = F^{k-1}X^{(1)} + B^{k-1}$, where the matrices $F^{k-1} \in \mathbb{R}^{d^{(k)} \times d^{(1)}}$, $B^{k-1} \in \mathbb{R}^{d^{(k)}}$, $G^{k+1} \in \mathbb{R}^{d^{(m+n+2)} \times d^{(k+1)}}$, $B^{k+1} \in \mathbb{R}^{d^{(m+n+2)}}$, respectively.

Definition 7. (Model Parameters) *The parameters θ of a $(m+n)$-deep neural network $\mathcal{F}_\theta$ are the concrete assignments to the weights $C^{(k)}$, biases $B^{(k)}$ for $k \in \{1, \cdots, m\}$ in Convolutional Block and the concrete assignments to weights $A_{fc}^{(k)}$ and biases $B_{fc}^{(k)}$ for $k \in \{1, \cdots, n+1\}$ in Fully-connected Block.*

Definition 8. (Signatures [8]) *The signature of a convolutional kernel is equal to $\alpha \cdot C^{(k)}$, where α is an arbitrary rescaling of the corresponding parameter.*

Definition 9. (Neuron State [9]) *Let $\mathcal{V}(\eta; X)$ denote the value that neuron η takes with $X \in \mathbb{R}^{d^{(1)}}$ before applying its corresponding activation function σ. If $\mathcal{V}(\eta; X) > 0$ (respectively, $\mathcal{V}(\eta; X) < 0$), the neuron state of η is activated (respectively, inactivated). If $\mathcal{V}(\eta; X) = 0$, the neuron state is critical.*

2.3 Adversarial Goals and Assumptions

The adversarial goal is to achieve an (ε, ξ)-functionally equivalent parameter extraction, rather than replicating the exact original parameters [9,34].

Definition 10. ((ε, ξ)-Functional Equivalence [9]) *Two neural networks $\mathcal{F}_\theta$ and $\mathcal{F}_{\hat{\theta}}$ are (ε, ξ)-functional equivalent on the input dataset S if*

$$Pr_{x \in S}(|\mathcal{F}_\theta(x) - \mathcal{F}_{\hat{\theta}}(x)| \leq \varepsilon) \geq 1 - \xi.$$

$(\varepsilon, 0)$-Functional Equivalence [9]. To evaluate the precision of our attack, we adopt the concept of (ε, ξ)-Functional Equivalence proposed by Carlini et al. [9] and subsequently used by Chen et al. [10,11].

In our analysis, we specifically focus on the $(\varepsilon, 0)$-functional equivalence given by Carlini *et al.* [9] which represents an upper bound on the maximum error of the output. We adopt the error bounds propagation method introduced by Carlini *et al.* [9, Section 6.2] to compute the $(\varepsilon, 0)$-functional equivalence.

Assumptions. We make the following assumptions about the Oracle and the capabilities of the attacker: **Architecture knowledge:** We require knowledge of the model architecture of the neural network, including dimensions of kernel matrix, max pooling kernel, and all the strides $s_c^{(k)}$, $s_\rho^{(k)}$. **Full-domain inputs:** The attacker can query arbitrary inputs from $\mathbb{R}^{d^{(1)}}$. **Precise Computation:** The CNN is specified and evaluated using sufficiently high precision floating-point arithmetic. **ReLU Activations:** All activation functions $\sigma^{(k)}$ are the ReLU function. **Raw Output Accessible[6]:** Given input X, the raw outputs of the CNN is accessible to the adversary. The same assumption is also used by Carlini *et al.*'s attack [9].

[6]When the CNN applies a Softmax function to the raw outputs (logits) to generate probability scores, the neuron fusion technique proposed by Chen *et al.* [11] at ASIACRYPT 2025 allows us to use the scores to compute the raw output of a CNN*, which adds an additional and known linear transformation after the last layer of the original CNN. Then our raw-output based attack still works.

3 Input Space Partition and Critical Points Recognition

The input space $\mathbb{R}^{d^{(1)}}$ is partitioned into many linear regions. Within each distinct linear region, the activation states of all neurons and the selection indices of the pooling layers remain constant, and the neural network computes a fixed linear function over the region. However, when linearly varying inputs cross the boundaries between these regions, the output exhibits non-linear changes. According to the definitions in Section 2.2, it is evident that both convolutional and fully-connected layers represent linear operations. The non-linearity is jointly introduced by the activation layers and the pooling layers:

- **ReLU Activation.** This determines whether a neuron's input is propagated to the subsequent layer, where a non-linear transition between a continuous zero state (inactivated) and a linear non-zero state (activated) occurs. For a fixed input, the activation matrix is deterministic.
  Carlini *et al.*'s seminal attack [9] and all the follow-up results [7,8,10,11] are based on the non-linearity introduced by ReLU at specific inputs, which are called *critical points*. The adversary studies the behavior of the FCNN in the vicinity of critical points, which are defined as inputs in whose tiny vicinity exactly one ReLU input in one of the layers changes sign. It is expected that the FCNN's output will change non-linearly as a result of this ReLU transition, which makes this event noticeable when we observe the network's outputs by varying the inputs in those tiny vicinities.
  It is important to note that for the critical point to be effective, the neuron must be selected by the subsequent pooling layers; otherwise, its activation state has no impact on the final output.
- **Max Pooling.** This operation selects the maximum value within each local receptive field. Since different neurons have different gradients in their linear outputs, a shift in the pooling selection results in a non-linear change in the final output. Similar to the activation layer, the pooling matrix is fixed when the input is determined.

3.1 Does Carlini *et al.*'s Critical Point Work for CNN?

At EUROCRYPT 2024, Canales-Martinez *et al.* [7] stated that the attack based on Carlini *et al.*'s critical points also works for CNNs without modification:

> "*Our attack can be applied without modifications to convolutional networks since they can be described as a special form of a fully connected network.*"

In the FCNN, a critical point can be identified by changing the inputs in its tiny vicinity and testing whether the network's output changes non-linearly. However, unlike FCNN, the non-linearity of CNN is introduced through ReLU or max pooling, which may lead to two problems when applying Carlini *et al.*'s critical point methods.

Problem I: Take Figure 4 as an example. The critical point $\bar{X}^{(1)}$ causes the 0-th entry of the input vector $Y^{(k)}$ of the $\sigma_c^{(k)}$ layer to be zero. However, the 2×2 max pooling layer does not select the output of $Y^{(k)}[0]$. It selects the maximum value among the neurons $Y^{(k)}[0,1,3,4]$, which outputs the value 2 in Figure 4.

Therefore, changing the inputs within a tiny vicinity of the critical point $\bar{X}^{(1)}$ will cause the network's output to change linearly, making the critical point $\bar{X}^{(1)}$ undetectable. Only when the max pooling layer selects the output of neuron α, the network's output will be affected non-linearly by changing the inputs around $\bar{X}^{(1)}$. As shown in Figure 5, the values in neuron $Y^{(k)}[0,1,3,4]$ are 0, -2, -1, -1, and neuron α is at the critical point. The max pooling layer is expected to select the output of neuron α when changing the inputs near $\bar{X}^{(1)}$, and to cause the network's output to change non-linearly, which makes the critical point $\bar{X}^{(1)}$ detectable. Formally, the critical point $\bar{X}^{(1)}$ makes $Y_t^{(k)} = 0$ $(0 \le t \le d_f^{(k)} - 1)$ of

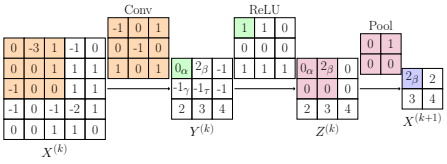


Fig. 4: Undetectable Critical Point

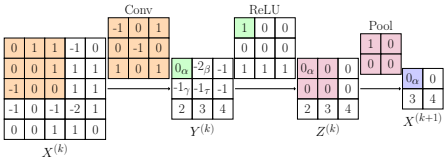


Fig. 5: Detectable Critical Point

$\sigma_c^{(k)}$. According to Eq. (6), $\tau_t = 0$. Let $\bar{X}_+^{(1)} = \bar{X}^{(1)} + \epsilon$ and $\bar{X}_-^{(1)} = \bar{X}^{(1)} - \epsilon$ with small enough ϵ, and assuming the corresponding $Y_{t+}^{(k)} > 0$ and $Y_{t-}^{(k)} < 0$. Hence, the corresponding $\tau_{t+} = 1$, $\tau_{t-} = 0$. Considering the matrix $P^{(k)} \cdot I^{(k)}$, in order to apply Carlini *et al.*'s attack, the entries $\tau_{t+} = 1$, $\tau_{t-} = 0$ should result in different matrices $P^{(k)} \cdot I^{(k)}$, else the two inputs $\bar{X}_+^{(1)}$ and $\bar{X}_-^{(1)}$ will have a linear impact on the output of the CNN. Note that the entries of $P^{(k)} \cdot I^{(k)}$ related to τ_t are the t-th column of $P^{(k)} \cdot I^{(k)}$. Only when $(P_{0,t}^{(k)}, P_{1,t}^{(k)}, \cdots, P_{d^{(k+1)}-1,t}^{(k)})^\intercal \neq \mathbf{0}$, there exists i such that $P_{i,t}^{(k)} = 1$. Then, $(P^{(k)} \cdot I^{(k)})_{i,t} = \tau_t$, which makes the $P^{(k)} \cdot I^{(k)}$ different for $\tau_{t+} = 1$ and $\tau_{t-} = 0$. Moreover, in the following layers' operations, τ_t should also appear in the operations matrix to affect the CNN's output, *i.e.*, the matrix $G^{k+1} \cdot P^{(k)} \cdot I^{(k)}$ should also be affected by τ_t, otherwise, the critical point is undetectable as well. We give a definition on this kind of detectable critical point.

Definition 11. (ReLU-Pooling Critical Point, RPCP) *A ReLU-Pooling Critical Point (RPCP) is an input $X^{(1)} \in \mathbb{R}^{d^{(1)}}$ that makes the input to the ReLU function in $\sigma_c^{(k)}$ layer 0, and by varying the inputs in the tiny vicinity of $X^{(1)}$, the output of CNN changes non-linearly.*

Property 1. If $X^{(1)}$ is a RPCP corresponding to the t-th neuron in layer k, then $A_t^{(k)}X^{(k)}+B_t^{(k)}=0$, and the inputs of other neurons in the same Local Receptive Field of Pooling Layer are smaller than 0.

Problem II: Besides RPCPs, the network's output may also change non-linearly because the pooling layer selection changes. As shown in Figure 6, in layer k, there is no 0 in $Y^{(k)}$, *i.e.*, no RPCP. However, in $Z^{(k)}$, the input $X^{(1)}$ makes $Z_{(k)}[0]=Z_{(k)}[1]\geq Z_{(k)}[3]=Z_{(k)}[4]$, hence the max pooling kernel matrix $M^{(k),0,0}=\begin{bmatrix}1&0\\0&0\end{bmatrix}$. Let $\bar{X}_+^{(1)}=\bar{X}^{(1)}+\epsilon$ and $\bar{X}_-^{(1)}=\bar{X}^{(1)}-\epsilon$ with small enough ϵ, and assuming the corresponding $(Z_+^{(k)}[0]<2,\ Z_+^{(k)}[1]>2)$ and $(Z_-^{(k)}[0]>2,\ Z_-^{(k)}[1]<2)$. Then, the selection matrix $M^{(k),0,0}$ for $\bar{X}_+^{(1)}$ and $\bar{X}_-^{(1)}$ will be different. Thus, the output of CNN will be non-linearly impacted.

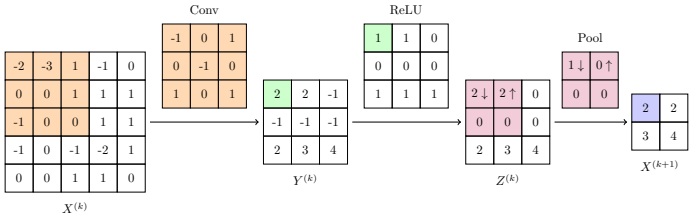


Fig. 6: Pooling Switch Point (PSP)

Definition 12. (Pooling Switching Point, PSP) *A Pooling Switching Point (PSP) is an input $X^{(1)}\in\mathbb{R}^{d^{(1)}}$ such that two inputs within a local receptive field achieve the same maximum value. By varying the inputs in the tiny vicinity of $X^{(1)}$, the output of CNN changes non-linearly.*

Property 2. If $X^{(1)}$ is a PSP corresponding to the i-th and j-th neurons in layer k, then $A_i^{(k)}X^{(k)}+b^{(k)}=A_j^{(k)}X^{(k)}+b^{(k)}$, and the inputs of other neurons in the same LRF-P are smaller than it.

Thus, non-linearity in CNNs is jointly induced by the changes in neuron activation states via ReLU and the shifts in neuron selection via pooling operations. Meanwhile, note that in the Fully-connected Block, critical points of each neuron lead to the existence of a corresponding ReLU state change.

Definition 13. (Fully-connected Critical Point, FCP [9]) *A Fully-connected Critical Point (FCP) is an input $X^{(1)}\in\mathbb{R}^{d^{(1)}}$ that makes the input of ReLU function $\sigma^{(k)}$ of a neuron in the Fully-connected Block equal to 0.*

We collect RPCPs, PSPs and FCPs from all layers by continuously varying inputs along any straight lines in $\mathbb{R}^{d^{(1)}}$, and searching for the breakpoints where non-linear changes happen.

Remarks. In the Fully-connected Block, the FCPs will definitely affect the output. However, the Convolutional Block is in fact a series of partially connected layers. Therefore, the RPCPs and PSPs should satisfy the properties 1 and 2 respectively, and they should also be selected by subsequent Convolutional Rounds (which must be activated and selected through subsequent activation and pooling operations until the model's output) to affect the final output of CNN.

4 Model Extraction with ReLU-Pooling Critical Point

4.1 Signature Extraction: Algebraic View of Carlini *et al.*'s Differential Attack

The signature extraction is mainly based on Carlini *et al.*'s differential attack [9]. We formalize it in an algebraic view. Consider a specific input $X \in \mathbb{R}^{d^{(1)}}$ identified as a ReLU-Pooling Critical Point (RPCP)[7] corresponding to the i-th neuron in the k-th convolutional layer. X is on the boundary between its two linear neighborhoods where only the activation state of neuron i changes among all neurons in the network by slightly altering X. A small perturbation $\vec{\delta}$ is introduced such that the points $X + \vec{\delta}$ and $X - \vec{\delta}$ fall into these two respective linear neighborhoods. Consequently, the corresponding activation matrices, denoted as $I_+^{(k)}$ (for $X + \vec{\delta}$) and $I_-^{(k)}$ (for $X - \vec{\delta}$), differ only at the i-th diagonal entry. Specifically, the i-th entry is 1 (assuming activation under $X + \vec{\delta}$) and 0 (assuming inactivation under $X - \vec{\delta}$).

Assume that the $r = (p \cdot w_\rho^{(k)} + q)$-th LRF-P of $\hat{O}^{(k)}$ covers the critical neuron i, where $(0 \le p \le h_\rho^{(k)} - 1,\ 0 \le q \le w_\rho^{(k)} - 1,\ 0 \le r < d^{(k+1)})$. According to Eqs. (9) and (10), on the active side of X, the selection Boolean matrix $M^{(k),p,q}$ corresponding to the activated state $X + \vec{\delta}$ only selects the positive activation state of neuron i, *i.e.*, the corresponding selected position is 1 and all others are 0 in $M^{(k),p,q}$. However, on the inactive side of X, the $M^{(k),p,q}$ of the inactive state $X - \vec{\delta}$ can be any Boolean matrix with only one element being 1 and the rest being 0, because $X - \vec{\delta}$ makes all neurons in $(p \cdot w_\rho^{(k)} + q)$-th LRF-P be 0 by the ReLU functions due to Prop. 1 of RPCP. Hence, we choose the same selection matrix $M^{(k),p,q}$ for the inactive side as for the active side. Consequently, the pooling matrix $P^{(k)}$ remains identical in the both sides of X.

Then, the output differences can be expressed as:

$$\mathcal{F}_\theta(X + \vec{\delta}) - \mathcal{F}_\theta(X) = G^{k+1} P^{(k)} I_+^{(k)} A^{(k)} F^{k-1} \vec{\delta}, \tag{16}$$

$$\mathcal{F}_\theta(X) - \mathcal{F}_\theta(X - \vec{\delta}) = G^{k+1} P^{(k)} I_-^{(k)} A^{(k)} F^{k-1} \vec{\delta}. \tag{17}$$

Define the second-order differential function as $\mathcal{H}(X; \vec{\delta})$ by the subtraction of Eqs. (16) and (17), *i.e.*, $\mathcal{H}(X; \vec{\delta}) = \mathcal{F}_\theta(X + \vec{\delta}) + \mathcal{F}_\theta(X - \vec{\delta}) - 2\mathcal{F}_\theta(X)$, therefore

$$\mathcal{H}(X; \vec{\delta}) = G^{k+1} P^{(k)} (I_+^{(k)} - I_-^{(k)}) A^{(k)} F^{k-1} \vec{\delta}, \tag{18}$$

[7] Step 3 in Sect. 6 discusses how to distinguish between RPCP, PSP, and FCP.

where the term $I_+^{(k)} - I_-^{(k)}$ yields a diagonal matrix with a single non-zero entry (value 1) at the entry (i, i), effectively acting as a selection operator. The product $P^{(k)}(I_+^{(k)} - I_-^{(k)})$ extracts the specific column of the pooling matrix $P^{(k)}$ associated with the critical neuron i. Since X is an RPCP, the output of the neuron i is the unique maximum within its LRF-P, implying that the corresponding entry of $P^{(k)}$ is 1 (*i.e.*, it is selected), while the contributions from other neurons in the window are zero.

Assume that the LRF-Ps are disjoint and the network has a single output (targeting one of the components of the output vector). The matrix G^{k+1} reduces to a single row vector, denoted as $g = (g_0, g_1, \cdots, g_{d^{(k+1)}-1})$, then

$$G^{k+1}P^{(k)}(I_+^{(k)} - I_-^{(k)})A^{(k)} = (g_r A_{i,0}^{(k)}, g_r A_{i,1}^{(k)}, \cdots, g_r A_{i,d^{(k)}-1}^{(k)}). \tag{19}$$

By sampling n different directions $\vec{\delta}^{(j)}$ $(0 \leq j \leq n-1)$ and querying the model to get $\mathcal{H}(X; \vec{\delta}^{(j)})$, we construct a system of equations:

$$\begin{pmatrix} (F^{k-1}\vec{\delta}^{(0)})^\mathsf{T} \\ (F^{k-1}\vec{\delta}^{(1)})^\mathsf{T} \\ \vdots \\ (F^{k-1}\vec{\delta}^{(n-1)})^\mathsf{T} \end{pmatrix} \begin{pmatrix} g_r A_{i,0}^{(k)} \\ g_r A_{i,1}^{(k)} \\ \vdots \\ g_r A_{i,d^{(k)}-1}^{(k)} \end{pmatrix} = \begin{pmatrix} \mathcal{H}(X; \vec{\delta}^{(0)}) \\ \mathcal{H}(X; \vec{\delta}^{(1)}) \\ \vdots \\ \mathcal{H}(X; \vec{\delta}^{(n-1)}) \end{pmatrix} = \vec{\mathcal{H}}. \tag{20}$$

Denote $S = (\vec{\delta}^{(0)}, \vec{\delta}^{(1)}, \cdots, \vec{\delta}^{(n-1)})^\mathsf{T}$, the system above can be rewritten as:

$$S{F^{k-1}}^\mathsf{T} {g_r A_i^{(k)}}^\mathsf{T} = \vec{\mathcal{H}}. \tag{21}$$

4.2 Pattern Matching Method: Recover Full Signature with Multiple RPCPs

To ensure the linear system above possesses a unique solution, the coefficient matrix of Eq. (21) should maintain full column rank, i.e., $\text{rank}(S(F^{k-1})^\mathsf{T}) = d^{(k)}$. As discovered by Carlini *et al.* [9, Section 4.3.2] in the attack on FCNN, the rank of the matrix is usually not full since the ReLU functions suppress negative output, and therefore some weights are impossible to recover (denoted as NaN values) with only one critical point. They have to find at least two critical points for the same i-th neuron to derive several partial signatures and merge them to extract the full signature [9,34].

Similarly, given one RPCP, the $\text{rank}(S(F^{k-1})^\mathsf{T})$ of Eq. (21) in CNNs is also affected by ReLU functions (i.e., the rank of $I^{(k-1)}$ may not be full), leading to $rank(S(F^{(k-1)})^\mathsf{T}) < d^{(k)}$. Note that according to Definitions 2 and 3, the convolutional matrix $A^{(k)}$ and the pooling matrix $P^{(k)}$ are full row rank. Therefore, some entries of ${g_r A_i^{(k)}}^\mathsf{T}$ are impossible to recover with only one RPCP. Similar to Carlini *et al.*'s method [9], we also need multiple RPCPs to recover several partial signatures and merge them, since $rank(S(F^{k-1})^\mathsf{T}) < d^{(k)}$ in most cases.

The difference is that our RPCPs do not necessarily correspond to the same neuron, since according to Eq. (5), each row of $A^{(k)}$ contains the full information about the kernel matrix $C^{(k)}$, whose entries are to be recovered. According to Eq. (5), different RPCPs for different neurons in layer k recover different rows of $A^{(k)}$. *The problem is to determine which rows of* $A^{(k)}$ *the RPCPs recover, and then merge them according to Eq.* (5) *to get the full signature.* In theory, given a row $\hat{A}_i^{(k)}$ recovered by an RPCP, the row number i could also be determined by the non-zero and zero positions in Eq. (5). However, in practice, both the zero-values and kernel weights (nonzero) in $A_i^{(k)}$ have possibility to be recovered as NaN. Consequently, it cannot be directly recognized which row the RPCP recovers, *i.e.*, one cannot directly determine which neuron is associated with the RPCP. Therefore, it is not trivial to align the partially recovered $C^{(k)}$ and merge them. We propose the *Pattern Matching Method* to solve this problem.

Pattern Matching Method. According to Eq. (5), for the i-th row of $A^{(k)}$, define the corresponding i-th convolution kernel pattern vector $\vec{V}_i \in \mathbb{F}_2^{d^{(k)}}$ as

$$\vec{V}_i[j] = \begin{cases} 1, & \text{if } A_i^{(k)}[j] \neq 0, \\ 0, & \text{else.} \end{cases} \tag{22}$$

Define its bitwise inverse vector as $\bar{\vec{V}}_i \in \mathbb{F}_2^{d^{(k)}}$ with $\bar{\vec{V}}_i[j] = \vec{V}_i[j] \oplus 1$.

Suppose we have an RPCP corresponding to the i-th ReLU function of layer k, denote the computed vector from Eq. (21) as $\hat{A}_i^{(k)} \approx g_r A_i^{(k)}$, where there exists NaN values. According to Eq. (23), the recovered $\hat{A}_i^{(k)}$ could be

$$\hat{A}_i^{(k)} = (\underbrace{\mathbf{0}}_{w_{in}^{(k)} \cdot \bar{i}}, \underbrace{\underbrace{\mathbf{0}}_{\underline{i}}, \hat{C}_{0,0}^{(k)}, \mathsf{NaN}, \cdots, \hat{C}_{0,w_c^{(k)}-1}^{(k)}, \underbrace{\mathbf{0}}_{pad}}_{w_{in}^{(k)}}, \cdots, \underbrace{\underbrace{\mathbf{0}}_{\underline{i}}, \mathsf{NaN}, \mathsf{NaN}, \cdots, \hat{C}_{h_c^{(k)}-1,w_c^{(k)}-1}^{(k)}, \underbrace{\mathbf{0}}_{pad}}_{w_{in}^{(k)}}, \underbrace{\mathbf{0}}_{pad}), \tag{23}$$

where $\bar{i} = \left\lfloor \frac{i}{w_o^{(k)}} \right\rfloor$, $\underline{i} = i \bmod w_o^{(k)}$. Please note that any entry in $\hat{A}_i^{(k)}$ may contain a NaN value, including those entries whose actual value should be 0. Then, we define two binary mask vectors for $\hat{A}_i^{(k)}$ as $\vec{\alpha}, \vec{\alpha}' \in \mathbb{F}_2^{d^{(k)}}$, where

$$\vec{\alpha}[j] = \begin{cases} 1, & \text{if } A_i^{(k)}[j] \neq 0 \text{ or } A_i^{(k)}[j] = \mathsf{NaN} \\ 0, & \text{else,} \end{cases}, \quad \vec{\alpha}'[j] = \begin{cases} 1, & \text{if } A_i^{(k)}[j] = 0 \text{ or } A_i^{(k)}[j] = \mathsf{NaN} \\ 0, & \text{else.} \end{cases}$$

Given an RPCP, $\vec{\alpha}, \vec{\alpha}' \in \mathbb{F}_2^{d^{(k)}}$ are computed. Then, we use the following *matching principle* to detect which ReLU function the RPCP corresponds to. Assuming the RPCP corresponds to the i-th ReLU function, we consider the i-th convolution kernel pattern vector $\vec{V}_i$ and the extracted vector $\hat{A}_i^{(k)}$.

1. Non-zero Consistency Principle: Obviously, when $\vec{V}_i[j] = 1$, $\hat{A}_i^{(k)}[j]$ will be either non-zero or NaN, *i.e.*, $\vec{V}_i[j] = \vec{\alpha}[j]$ and

$$\vec{V}_i \cdot \vec{\alpha}^{\mathsf{T}} = w_c^{(k)} \cdot h_c^{(k)}. \tag{24}$$

2. Zero Consistency Principle: Similarly, when $\vec{V}_i[j] = 1$, $\vec{\alpha}'[j] = 1$ we have

$$\vec{V}_i \cdot \vec{\alpha}'^{\mathsf{T}} = d^{(k)} - w_c^{(k)} \cdot h_c^{(k)}. \tag{25}$$

We iteratively compute Eqs. (24) and (25) for each $0 \leq i \leq d_f^{(k)} - 1$ and check if these two equations hold. If Eqs. (24) and (25) hold, we deduce that the RPCP corresponds to the i-th ReLU function, and then extract partial values of the $\hat{C}^{(k)}$ from Eq. (23). For multiple RPCPs, we recover the partial $\hat{C}^{(k)}$ and merge them to build a full $\hat{C}^{(k)}$.

Remarks. By employing the pattern matching method, we identify the specific neuron and determine the constituent elements of the kernel matrix from a single RPCP. This makes the reconstruction of kernel weights simple. The identification mechanism is both intuitive and robust: results derived from noisy points (non-RPCPs) fail to yield a kernel pattern vector $\vec{V}_i$ that conforms to the matching principle, causing the pattern matching algorithm to inherently reject them.

Bias Recovery. In layer k, after the neuron i at the RPCP is identified, the bias (note that the bias for each neuron is the same according to Definition 2) is computed, since the input of the neuron i is zero:

$$\hat{b}^{(k)} = -\hat{A}_i^{(k)}(F^{k-1}x + B^{k-1}) \ . \tag{26}$$

4.3 Sign Recovery

After recovering $\hat{A}^{(k)}$ and $\hat{B}^{(k)}$ without sign, we compute the input vector of the neurons as $\hat{A}^{(k)}(F^{k-1}x + B^{k-1}) + \hat{B}^{(k)}$, where x is the RPCP of the i-th neuron of layer k. Assume that neuron i belongs to the t-th LRF-P Ω_t. According to Prop. 1, all the inputs of the neurons within Ω_t should be ≤ 0. Then,

- if all these inputs are ≤ 0, the sign of the extracted kernel matrix is correct.
- if there exists a positive input, the extracted kernel matrix has an inverted sign, and we multiply the matrices $\hat{A}^{(k)}$ and $\hat{B}^{(k)}$ by "-1".

5 Parameter Extraction with Pooling Switching Point

The attack based on the RPCP given in Sect. 4 relies highly on the successful acquisition of RPCPs. When extending the attack to deeper layers, the proportion of valid RPCPs among all discovered critical points diminishes greatly and the efficiency of the attack decreases significantly (detailed experiments confirm this phenomenon in Sect. 6). Therefore, we propose a more efficient parameter extraction attack based on the Pooling Switching Point (PSP). To leverage the PSP, a novel *internal differential attack* on CNNs is applied, instead of Carlini *et al.*'s differential attack [9].

5.1 Internal Differential Extraction Attack on CNN

The Internal differential attack [40] was proposed by Peyrin at CRYPTO 2010 against Grøstl, which was subsequently applied to the cryptanalysis of Keccak by Dinur, Dunkelman, and Shamir [12]. This section proposes the first internal differential attack on CNNs, named the *internal differential extraction* attack.

According to Prop. 2 of the Pooling Switching Point (PSP): Given a PSP X in layer k, suppose the i-th and j-th neurons' inputs within the same t-th LRF-P Ω_t $(0 \leq t < d^{(k+1)})$ have the same largest value, which satisfies:

$$Y^{(k)}[i] = A_i^{(k)}(F^{k-1}X + B^{k-1}) + b^{(k)} = Y^{(k)}[j] = A_j^{(k)}(F^{k-1}X + B^{k-1}) + b^{(k)} > 0. \tag{27}$$

Then, X is on the boundary between two linear neighborhoods, across which the selection of the t-th LRF-P switches between neuron i and neuron j. It is important to note that $A_i^{(k)}$ is a sparse vector according to Eq. (5), which only contains non-zero entries at the indices where the convolution kernel is applied. This sparsity allows us to explicitly extract the positional indices of the convolution kernel's receptive field (LRF-C). Denote the vector form of kernel matrix $C^{(k)}$ as $\vec{C}^{(k)} = (C_0^{(k)}, C_1^{(k)}, \cdots, C_{h_c^{(k)}-1}^{(k)})^\mathsf{T}$. Thus the length of the vector $\vec{C}^{(k)}$ is $l_c^{(k)} = h_c^{(k)} w_c^{(k)}$. Assume that the stride $s_c^{(k)} = 1$, and let $\mathcal{K}_i^{(k)} \subset \{0, 1, \cdots, d^{(k)} - 1\}$ denote the set of indices belonging to the LRF-C of the i-th neuron $(0 \leq i \leq d_f^{(k)} - 1)$, then

$$\mathcal{K}_i^{(k)} := \left\{ w_{in}^{(k)} \cdot \left\lfloor \frac{i}{w_o^{(k)}} \right\rfloor + i \bmod w_o^{(k)} + w_{in}^{(k)} \cdot \left\lfloor \frac{t}{w_c^{(k)}} \right\rfloor + t \bmod w_c^{(k)},\ 0 \leq t < l_c \right\}. \tag{28}$$

The set $\mathcal{K}_i^{(k)}$ picks out the nonzero entries of $A_i^{(k)}$ according to Eq. (5). Then,

$$A_i^{(k)} X^{(k)} = \vec{C}^{(k)} \cdot X^{(k)}[\mathcal{K}_i^{(k)}]. \tag{29}$$

Similar equations can be deduced for neuron j. Then, subtract the two equations, according to Eq. (27), we get

$$\vec{C}^{(k)} \cdot (X^{(k)}[\mathcal{K}_i^{(k)}] - X^{(k)}[\mathcal{K}_j^{(k)}]) = 0. \tag{30}$$

This is quite similar to the internal-differential attack [40], where the difference between two distinct parts of the unique input $X^{(k)}$ is studied through the cryptographic/CNN operations. The input difference is $\nabla X^{(k)} = X^{(k)}[\mathcal{K}_i^{(k)}] - X^{(k)}[\mathcal{K}_j^{(k)}]$, and the output difference after the convolutional layer is 0, which will be identified as a PSP. Given a PSP, Eq. (30) is used to construct the system of equations to compute $\vec{C}^{(k)}$. With the deduced parameters of $k-1$ layers, $X^{(k)} = F^{k-1}X + B^{k-1}$ is deduced. The only unknowns are the positions of the pooling selection switch, *i.e.*, i and j.

5.2 Extracting i and j of the Pooling Selection Switch

Similar to Carlini *et al.*'s differential attack [9], we introduce a small perturbation $\vec{\delta}$ such that the points $X+\vec{\delta}$ and $X-\vec{\delta}$ remain within the linear neighborhoods of the PSP X. Exactly one of the two points will make the pooling select the i-th neuron's output. Assume that this point is $X+\vec{\delta}$, and denote the corresponding pooling matrix as $P_+^{(k)}$. The other point will make the output of the j-th neuron selected, and $P_-^{(k)}$ is defined similarly. Then, we have

$$\mathcal{F}_\theta(X+\vec{\delta})-\mathcal{F}_\theta(X)=G^{k+1}P_+^{(k)}I^{(k)}A^{(k)}F^{k-1}\vec{\delta}, \tag{31}$$

$$\mathcal{F}_\theta(X)-\mathcal{F}_\theta(X-\vec{\delta})=G^{k+1}P_-^{(k)}I^{(k)}A^{(k)}F^{k-1}\vec{\delta}. \tag{32}$$

Note that because neurons i and j share the same maximum value at X, selecting either neuron yields the correct pooling output. Consequently, $P_+^{(k)}$ and $P_-^{(k)}$ are functionally equivalent at X. Hence, for $\mathcal{F}_\theta(X)$, Eq. (31) uses $P_+^{(k)}$ to maintain the same matrix as $\mathcal{F}_\theta(X+\vec{\delta})$, and similarly for Eq. (32). Assuming the t-th ($0\le t\le d^{(k+1)}-1$) LRF-P includes neurons i and j ($i<j$) whose input values are $Y^{(k)}[i]$ and $Y^{(k)}[j]$, the matrices $P_+^{(k)}$ and $P_-^{(k)}$ will be different only in their t-th row. Specifically, in $P_+^{(k)}$, this row is $(0\cdots0,1_i,0\cdots0,0_j,0\cdots0)$, while in $P_-^{(k)}$, the same row becomes $(0\cdots0,0_i,0\cdots0,\ 1_j,0\cdots0)$. Still, the second-order difference function, denoted as $\mathcal{H}(x;\delta)=\mathcal{F}_\theta(X+\vec{\delta})+\mathcal{F}_\theta(X-\vec{\delta})-2\mathcal{F}_\theta(X)$:

$$\mathcal{H}(X;\vec{\delta})=G^{k+1}(P_+^{(k)}-P_-^{(k)})I^{(k)}A^{(k)}F^{k-1}\vec{\delta}. \tag{33}$$

Similar to the RPCP method in Sect. 4, when the matrix G^{k+1} is reduced to a single row vector $g=(g_0,g_1,\cdots,g_{d^{(k)}-1})$, then

$$G^{k+1}(P_+^{(k)}-P_-^{(k)})I^{(k)}A^{(k)}=g_t(A_i^{(k)}-A_j^{(k)}), \tag{34}$$

where t-th row of $P_+^{(k)}-P_-^{(k)}$ is $(0\cdots0,1_i,0\cdots0,-1_j,0\cdots0)$, and the other rows are zero vectors. By sampling n distinct directions $\vec{\delta}^{(\ell)}$ ($0\le\ell\le n-1$), we can construct a system of equations and recover $g_t(A_i^{(k)}-A_j^{(k)})$,

$$g_t(A_i^{(k)}-A_j^{(k)})\cdot F^{k-1}\vec{\delta}^{(\ell)}=\mathcal{H}(X;\vec{\delta}^{(\ell)}). \tag{35}$$

According to Eq. (5), $A_i^{(k)}$ and $A_j^{(k)}$ correspond to the i-th and j-th row vectors of the matrix $A^{(k)}$, where their non-zero values are offset from each other. Assuming $i<j$, we deduce i and j from the positions of the first and last nonzero entries of the recovered vector $g_t(A_i^{(k)}-A_j^{(k)})$, respectively, according to Eq. (5). Then $\mathcal{K}_i$ and $\mathcal{K}_j$ are deduced from Eq. (28).

Handling vector containing NaN. Sometimes the calculated row vector $\Delta_{ij}\hat{A}^{(k)}\approx g_t(A_i^{(k)}-A_j^{(k)})$ from Eq. (35) is incomplete for reasons similar to those in Sect. 4.2. It is necessary to check whether the deduced $\hat{i}$ and $\hat{j}$ match the practical $g_t(A_i^{(k)}-A_j^{(k)})$, following the matching principles:

1. Union Non-zero Consistency: At indices where either $A_i^{(k)}$ or $A_j^{(k)}$ is non-zero, the corresponding element in the difference vector $\Delta_{ij}\hat{A}^{(k)}$ should be either non-zero or NaN.
2. Common Zero Consistency: At indices where both $A_i^{(k)}$ and $A_j^{(k)}$ are zero, the corresponding element in $\Delta_{ij}\hat{A}^{(k)}$ should be either 0 or NaN.

Following the above two principles and Eq. (5), we determine $\hat{i}$ by finding the first $w_c^{(k)}$ consecutive non-zero or NaN entries that contain at least one non-zero value in $\Delta_{ij}\hat{A}^{(k)}$ (denote this partial vector as $\vec{\beta}$), then

- If the first element $\vec{\beta}[0]$ is not NaN, return the position index of $\vec{\beta}[0]$ as $\hat{i}$.
- If $\vec{\beta}[0]$ is NaN, let $\hat{i}$ choose two possible values, *i.e.*,
  - $\hat{i}$ is the index of $\vec{\beta}[0]$ in Δ_{ij} by assuming the true value $\vec{\beta}[0]$ is not 0.
  - Increment the index of $\vec{\beta}[0]$ in Δ_{ij} by 1, and use it as the value of $\hat{i}$, by assuming the true value $\vec{\beta}[0]$ is 0.

Similarly, we determine $\hat{j}$ by the last $w_c^{(k)}$ consecutive non-zero or NaN values in $\Delta_{ij}\hat{A}^{(k)}$. Only the correct $(\hat{i},\hat{j})$ will strictly conform to the matching principles; otherwise we discard the PSP and choose a new one. After recovering the pair of $(\hat{i},\hat{j})$, we construct Eq. (30). We need slightly more than $h_c^{(k)}w_c^{(k)}$ PSPs to build a full-rank system of Eq. (30) to recover $\vec{C}^{(k)}$ as $\hat{\vec{C}}^{(k)}$.

5.3 Sign Recovery

Reconstruct the convolutional matrix $\hat{A}^{(k)}$ from the recovered kernel weights $\hat{\vec{C}}^{(k)}$. And once we extract $\hat{i}$ and $\hat{j}$, the involved LRF-P Ω_t $(0 \le t < d^{(k+1)})$ corresponding to the PSP could be deduced. According to Prop. 2, the outputs of the i-th and j-th neurons should be the largest value in Ω_t. Then, with $\hat{A}^{(k)}$,

- if $\hat{A}_i^{(k)}(F^{k-1}X+B^{k-1}) = \hat{A}_j^{(k)}(F^{k-1}X+B^{k-1}) > \hat{A}_\ell^{(k)}(F^{k-1}X+B^{k-1})$, $\ell \in \Omega_t$, the sign of the extracted kernel matrix is correct.
- if $\hat{A}_i^{(k)}(F^{k-1}X+B^{k-1}) = \hat{A}_j^{(k)}(F^{k-1}X+B^{k-1}) < \hat{A}_\ell^{(k)}(F^{k-1}X+B^{k-1})$, $\ell \in \Omega_t$, the extracted kernel matrix has an inverted sign, and we multiply the vector $\hat{A}_i^{(k)}$ by "-1".

On the bias recovery. Note that the bias term $b^{(k)}$ is eliminated in Eq. (27) on both sides, making it impossible to derive the bias information from PSP points alone. Consequently, we use the method described in Eq. (26) in Sect. 4 to recover the bias with a single RPCP.

6 Attacks in Practice: Unified Model Combining PSP and RPCP

6.1 Experiment on RPCP and PSP Methods

We first experimentally compare the RPCP and PSP methods on a $(2+1)$-deep CNN with a single input and output channel, trained on the MNIST dataset,

– In the first round, the input matrix is convolved with a kernel of size $(1, 1, 5, 5)$, stride $s_c^{(1)} = 1$ and padding $pad^{(1)} = 0$, followed by ReLU functions, and then downsampled by a pooling kernel of size 2×2, with stride $s_\rho^{(1)} = 2$.
– In the second round, the operations are identical to those in the first round.

We record the number of model queries, attack runtime, and error upper bound, as summarized in Table 2 and discussed as follows.

Table 2: Comparison of RPCP and PSP Methods

Layer/ Models	**Architecture** $d^{(k)}-d_f^{(k)}-d^{(k+1)}$	**Method**	**Time** seconds	**Queries**	$(\varepsilon, 0)$	$\max\lvert\boldsymbol{\theta}-\hat{\boldsymbol{\theta}}\rvert$	**Number**
1st/(2+1)	1024–784–196	RPCP	$2^{5.39}$	$2^{15.10}$	$2^{-12.78}$	$2^{-16.38}$	1400
		PSP	$2^{4.16}$	$2^{12.62}$	$2^{-32.09}$	$2^{-37.10}$	6487
2nd/(2+1)	196–100–25	RPCP	$2^{6.07}$	$2^{15.95}$	$2^{-16.04}$	$2^{-22.07}$	15
		PSP	$2^{4.60}$	$2^{14.49}$	$2^{-31.74}$	$2^{-35.21}$	1548

Number: Counting from 10,000 collected critical points.

– **Distribution density of different types of critical points.** Among the identified critical points, the number of PSPs significantly exceeds that of RPCPs within the same layer. Experimental results for the second layer show that out of 10,000 collected critical points, only 15 correspond to RPCPs, whereas PSP identifies 1548 in Table 2. Consequently, the PSP method achieves model extraction with shorter runtime and fewer model queries.
– **High precision extraction.** Recall the linear systems of equations used to solve for the convolution kernel in RPCP Eq. (21) for $A_i^{(k)}$ and PSP Eq. (30) for $\vec{C}^{(k)}$. For RPCPs, we must solve for $|A_i^{(k)}|{=}d^{(k)}$ variables, whereas for PSPs, the system reduces to only $|\vec{C}^{(k)}| = l_c$ variables, and $d^{(k)} \gg l_c$ according to Eq. (5). Therefore, the system for RPCP possesses higher degrees of freedom. Note that although the non-zero values in $A_i^{(k)}$ in Eq. (5) are the elements of $\vec{C}^{(k)}$, the locations of the zero entries in $A_i^{(k)}$ are unknown before solving the linear system in Eq. (21).
According to statistical learning theory, the expected prediction error of a least-squares estimator increases linearly with the number of input parameters [20, Section 2.5]. Consequently, estimating the $d^{(k)}$ variables tends to introduce a larger variance compared to solving the low-dimensional PSP system. Meanwhile, the method of least squares tends to distribute values across all coefficients [13], which often 'overfits' the noise by assigning spurious non-zero values to the zero-valued entries (note that many entries of $A_i^{(k)}$ are zeros), further exacerbating the extraction error. Therefore, the solution to the PSP system in Eq. (30) achieves higher precision.
In our practical experiment shown in Table 2, the PSP method significantly enhances the precision of parameter extraction. It achieves a much lower upper bound on the maximum output error (the term $(\varepsilon, 0)$) and a much lower

maximum parameter error (the term $\max|\theta - \hat{\theta}|$) than the RPCP method with fewer queries and less time.
- **Bias recovery.** A critical limitation remains: Since PSPs do not reveal any information regarding the bias, we need to collect at least one RPCP to recover one bias, following the procedure in Eq. (26) in Sect. 4.

6.2 Unified Model Combining PSP and RPCP

Suppose we have precisely extracted the CNN up to layer k, *i.e.*, the recovered $\hat{f}^{(1..k-1)}$ is functionally equivalent to the $f^{(1..k-1)}$ of the victim model. As detailed in Sects. 4 and 5, the procedures for computing the row vector $\hat{A}_i^{(k)}$ from RPCP and extracting position i and j of the pooling selection switch from PSP are similar (both are based on the computation of the second-order differential function). Meanwhile, we rely on the solution itself to distinguish between RPCPs and PSPs. The full attack is to leverage the PSP method in Sect. 5 for high-precision weights solving, and the RPCP method in Sect. 4 for bias and sign recovery. The detailed procedure is described below.

Step 1. Identify Critical Points and Filter Feasible Candidates. Following Carlini *et al.*'s method [9], we identify a collection of critical points by performing a binary search sweep along continuous linear paths, where the output exhibits a non-linear transition. Then we filter out those belonging to previous layers. For each candidate point, we reconstruct the transformation matrix F^{k-1}. Let $N_{positive}^{(k-1)}$ denote the number of positive values in the output of the $(k-1)$-th layer. We verify whether the point satisfies the condition: $N_{positive}^{(k-1)} \leq \text{rank}(F^{(k-1)})$. According to [34, Section 3.1], points that fail to meet the constraint will result in underdetermined linear systems in Eq. (20) or Eq. (35), and therefore will be discarded.

Step 2. Calculate the Signature. For a critical point X (RPCP or PSP or FCP), we sample tiny perturbations $\vec{\delta}$ in different directions, compute the second-order differential function $\mathcal{H}(X;\vec{\delta}) = \mathcal{F}_\theta(X+\vec{\delta}) + \mathcal{F}_\theta(X-\vec{\delta}) - 2\mathcal{F}_\theta(X)$, which is used in both the RPCP method (e.g., Eq. (18)) and the PSP method (e.g., Eq. (33)) to establish linear systems, such as Eq. (20) for RPCP or Eq. (35) for PSP. Let this linear system be $(F^{k-1}\vec{\delta}^{(i)})^\intercal \cdot \vec{\mathbf{sol}} = \mathcal{H}(X;\vec{\delta}^{(i)})$, and we solve it by the least squares method.

Step 3. Distinguish Critical Points and Extract Position Information. If the critical point in Step 2 is an RPCP, the solution derived will be the row vector $A_i^{(k)}$. Its non-zero entries are strictly confined to the LRF-C of the i-th output neuron (with a length of l_c). If the critical point is a PSP, the solution is the difference vector $A_i^{(k)} - A_j^{(k)}$, when two neurons share the same maximum value within an LRF-P. Here, the non-zero entries correspond to the union of the two respective LRF-Cs.

We extract the vector $\vec{sol}$ with all non-zero (excluding NaN) values from the calculated row vector $\vec{\mathbf{sol}}$. Let $l_{\vec{sol}}$ represents the length of $\vec{sol}$. Since NaNs appear in the solutions, the length of $l_{\vec{sol}}$ does not strictly correspond to the above scenarios. We use the relations between $l_{\vec{sol}}$ and l_c to distinguish if the critical point is an RPCP, a PSP, or an FCP:

- $l_{\vec{sol}} \leq l_c$: It may correspond to an RPCP or a PSP: we first attempt to match the convolution kernel pattern vector $\vec{V}_i$ defined in Eq. (22) following the method given in Sect. 4.2. If no valid result is returned, we treat it as a PSP to extract positional information.
- $l_c < l_{\vec{sol}} \leq 2l_c$: Since in $A_i^{(k)} - A_j^{(k)}$, non-zero values of the i-th and j-th row vectors are offset from each other, the number of non-zero values will not exceed twice l_c. The critical point can be assumed as a PSP and we retrieve the positional information.
- $l_{\vec{sol}} > 2l_c$: The critical point is an FCP.

If a PSP is misclassified as an RPCP, it will be automatically filtered out during the matching phase in Step 5. This is because the RPCP method attempts to directly recover the convolution kernel weights, yielding an invalid solution that fails the matching criteria.

Step 4. Calculate Precise Solution. We employ the PSPs to compute a high-precision solution. After gathering adequate PSPs and their corresponding position information (more than the number of kernel parameter), we establish the linear system of equations in Eq. (30) and solve it using the singular value decomposition (SVD) method to extract the convolution kernel weights.

Step 5. Recover the Biases and Signs. We employ RPCPs to recover the biases and signs. Suppose that a solution derived in Step 2 is recognized as RPCP's in Step 3. Then, it contains a partial signature of the row vector $A_i^{(k)}$ due to the existence of NaN discussed in Sect. 4.2. With the full signature recovered in Step 4, we match it with the partial signature (ignoring the positions of NaN values) derived from the RPCP to recognize the row index i of $A_i^{(k)}$. Then, we recover the bias according to Eq. (26) and the sign by Sect. 4.3.

When RPCPs are not found in Step 1, we propose a *targeted heuristic search strategy for RPCP* with the signature recovered by the PSP method in Step 4.

Targeted heuristic search strategy for RPCPs. Our objective is to identify an input $X^{(1)}$ that makes a specific neuron i at layer k reach its critical state *i.e.*, $\hat{A}_i^{(k)} X^{(k)} + \hat{b}_i^{(k)} = 0$. Given that the input preprocessing usually normalizes the space of $X^{(1)}$ (e.g., via standardization [22,44] or range scaling [42]), the bias term $\hat{b}^{(k)}$ is not required to compensate for large input offsets [24], *i.e.*, the absolute value of the bias $\hat{b}^{(k)}$ is typically small. The strategy is as follows:

1. Recover the sign by the PSP method in Sect. 5.3. Reconstruct the kernel matrix $\hat{A}^{(k)}$ with the correct signs.

2. With the assumption that the absolute value of the bias $\hat{b}^{(k)}$ is quite small, we employ a gradient-based optimizer to search the input space. The optimization process is governed by three primary constraints:
   - There exists a neuron i such that the term $\hat{A}_i^{(k)} X^{(k)}$ approaches 0.
   - Within the same LRF-P Ω_t, the values $\hat{A}_j^{(k)} X^{(k)},\ j \in \Omega_t,\ j \neq i$ must be substantially smaller than 0.
   - As many neurons as possible in other LRF-Ps have positive values.
3. Upon reaching a candidate point $X^{(1)}$, execute a randomized binary search within the local neighborhood of $X^{(1)}$ and query the CNN.

This increases the probability of finding the RPCPs compared to randomly testing the input as in Step 1.

7 End-to-End Practical Experiments

Taking the full CNN $\mathcal{F}_\theta$ as a black box, with the inputs and the raw outputs, we implemented the algebraic attacks on a variety of CNNs. Most experiments were conducted on a server equipped with an NVIDIA RTX 5080 GPU (16 GB memory), although the attack can also be executed on CPUs.

LeNet-5 with modern architectures. Our target models include modern implementations of LeNet-5 featuring multiple input and output channels. Unlike the original architecture, modern variants typically incorporate ReLU activation and max pooling. We introduce this LeNet-5 style architecture and the algebraic attack on *Multiple Channels* in detail in Supplementary Material B.

Extracting the parameters of the Convolutional Block of CNNs. Given a $(m+n)$-deep CNN's inputs and outputs, we first perform the experiment to recover the parameters of the m-round *Convolutional Block*, which directly proves the efficiency and correctness of our attack. We conduct the experiments on various CNNs with different architectures (including the modern implementation of LeNet-5), with details given in Supplementary Material C.2, whose parameters were trained on random data, MNIST or CIFAR 10, respectively. We summarize the results of the experiments in the top 5 experiments in Table 1 in Sect. 1.

End-to-end Attack on Full and Reduced LeNet-5. Combining Carlini *et al.*'s differential attack [9] and Canales-Martínez *et al.*'s Neuron Wiggle sign-recovery method [7], we perform a full parameter extraction on both the *Convolutional Block* and *Fully-connected Block*, as well as the final output layer. We conduct the experiment on a reduced version of the modern $(2+2)$-deep LeNet-5. It is a $(2+1)$-deep CNN with the same 2-round *Convolutional Block* as the full LeNet-5, and a 1-round *Fully-connected Block* with the number of neurons reduced from 84 to 20, and the output layer.

We also attack the full modern LeNet-5, extending the black-box extraction to the Convolutional Block and FCNN Round 1. Due to the existence of the persistent and dead neurons [25], we recover 93.3% of the weights in FCNN Round 1. However, a functionally equivalent attack on the FCNN Block could be implemented with the CrossLayer method in [25].

The details of the reduced and full LeNet-5 are given in Supplementary Material C.2. The experiment results and the comparison with Carlini *et al.*'s experiment [9] and Foerster *et al.*'s experiment [15] are given in the last four rows of Table 1 in Sect. 1. A more detailed results table, including runtime, is provided in Table 3 in Supplementary Material C.1.

Noise-sensitivity Analysis. The PSP method used in the Convolutional Block and the least squares method (introduced by Carlini *et al.* [9]) used in the FCNN Block ensure high precision extraction and noise robustness. Take the (2+1)-reduced version of LeNet-5 as an example, as shown in Table 4 of Supplementary Material D. Under the end-to-end black-box setting, the upper bound on the maximum output error increases only marginally across layers: $2^{-27.86}$ (1st layer), $2^{-24.91}$ (2nd layer), $2^{-23.13}$ (3rd layer), and $2^{-22.21}$ (last layer). This accumulated precision loss is only $(27.86 - 22.21) = 5.65$ bits across four layers, confirming our method's numerical stability. We report the per-layer relative errors and analyze the error propagation in Supplementary Material D.

Persistent and Dead Neurons. According to Ito *et al.* [25], if the input distribution during the random binary search exclusively falls into the activated (or inactivated) region of a neuron, it is classified as a persistent (or dead) neuron. In CNNs, due to the weight-sharing mechanism, neurons in layer k are structurally symmetric, *i.e.*, if a specific neuron in layer k is persistent (or dead), the other neurons will likely exhibit the same behavior, since all the inputs of the neurons of layer k are generated by the same kernel matrix $C^{(k)}$ and bias. Therefore, it is difficult to find RPCPs in this case. For dead neurons, simply discard them since their outputs remain 0. While for persistent neurons, our PSP method does not require the neuron's output to cross zero and can be used to recover the kernel weights (including signs), as introduced in Sect. 5. However, RPCPs remain indispensable for bias recovery. In Sect. 6.2, we introduced a targeted heuristic search strategy to improve the search for RPCPs using the kernel weights recovered via PSPs. If this strategy also fails, we follow the cross-layer extraction by Ito *et al.* [25] to recover the bias from deeper layers.

8 Discussions and Conclusions

With the development of CNNs, many other structural hyperparameters have emerged, such as zero-padding [29,44,46,22], various convolutional strides $s_c > 1$ [29,46,22], various pooling strides s_ρ, and the dropout technique [29,44,46]. We discuss their effects on our attacks in Supp. F. Furthermore, we explore a more

restricted scenario where the kernel size is unknown to the attacker, and give a potential attack in detail in Supp. E.

In this paper, we tackle the challenging problem of cryptanalytic parameter extraction from Convolutional Neural Networks (CNNs) with max pooling for the first time. Unlike existing works focusing only on ReLU-based fully connected neural networks (FCNNs), we extend cryptanalytic extraction attacks to CNNs by overcoming max pooling-induced non-linearity challenges. We propose an algebraic representation of CNNs, identify two novel types of critical points (RPCPs and PSPs), and design corresponding extraction techniques. A unified model integrating RPCPs and PSPs is developed to leverage their strengths, with a heuristic strategy to address RPCP rarity. End-to-end black-box experiments on multiple CNNs validate our method's effectiveness in recovering full CNN parameters. Future work will extend the method to hard-label scenarios and complex CNN architectures.

Acknowledgement. We thank the anonymous reviewers from CRYPTO 2026 for their valuable comments. This work is supported by the National Key R&D Program of China (2024YFA1013000), the National Natural Science Foundation of China (U25B2014, 62272257, 62302250), the Fundamental and Interdis ciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM114), and the Zhongguancun Laboratory.

References

1. Roderick Asselineau, Patrick Derbez, Pierre-Alain Fouque, and Brice Minaud. Cryptanalytic extraction of deep neural networks with non-linear activations. *Cryptology ePrint Archive*, 2026.
2. Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. Mlperf tiny benchmark. *arXiv preprint arXiv:2106.07597*, 2021.
3. Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. {CSI}{NN}: Reverse engineering of neural network architectures through electromagnetic side channel. In *USENIX Security 2019*, pages 515–532, 2019.
4. Avrim Blum and Ronald Rivest. Training a 3-node neural network is NP-complete. *Advances in neural information processing systems*, 1, 1988.
5. Y-Lan Boureau, Jean Ponce, and Yann LeCun. A theoretical analysis of feature pooling in visual recognition. In *ICML-10*, pages 111–118, 2010.
6. Isaac A. Canales-Martínez and David Santos. Extracting some layers of deep neural networks in the hard-label setting. In *LATINCRYPT 2025*, volume 16129, pages 399–421. Springer, 2025.
7. Isaac Andrés Canales-Martinez, Jorge Chávez-Saab, Anna Hambitzer, Francisco Rodríguez-Henríquez, Nitin Satpute, and Adi Shamir. Polynomial time cryptanalytic extraction of neural network models. In *EUROCRYPT 2024, Proceedings, Part III*, volume 14653, pages 3–33. Springer, 2024.
8. Nicholas Carlini, Jorge Chávez-Saab, Anna Hambitzer, Francisco Rodríguez-Henríquez, and Adi Shamir. Polynomial time cryptanalytic extraction of deep neural networks in the hard-label setting. In *EUROCRYPT 2025, Proceedings, Part I*, volume 15601, pages 364–396. Springer, 2025.

9. Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. Cryptanalytic extraction of neural network models. In *CRYPTO 2020, Proceedings, Part III*, volume 12172, pages 189–218. Springer, 2020.
10. Yi Chen, Xiaoyang Dong, Jian Guo, Yantian Shen, Anyu Wang, and Xiaoyun Wang. Hard-label cryptanalytic extraction of neural network models. In *ASIACRYPT 2024, Proceedings, Part VIII*, volume 15491, pages 207–236.
11. Yi Chen, Xiaoyang Dong, Ruijie Ma, Yantian Shen, Anyu Wang, Hongbo Yu, and Xiaoyun Wang. Delving into cryptanalytic extraction of prelu neural networks. In *ASIACRYPT 2025*, pages 576–607. Springer, 2025.
12. Itai Dinur, Orr Dunkelman, and Adi Shamir. Collision attacks on up to 5 rounds of SHA-3 using generalized internal differentials. In Shiho Moriai, editor, *FSE 2013*, volume 8424, pages 219–240. Springer, 2013.
13. D.L. Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52(4):1289–1306, 2006.
14. Charles Fefferman et al. Reconstructing a neural net from its output. *Revista Matemática Iberoamericana*, 10(3):507–556, 1994.
15. Hanna Foerster, Robert Mullins, Ilia Shumailov, and Jamie Hayes. Beyond slow signs in high-fidelity model extraction. In *NeurIPS 2024*.
16. Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *ICLR 2019*. OpenReview.net, 2019.
17. Kunihiko Fukushima, Sei Miyake, and Takayuki Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE transactions on systems, man, and cybernetics*, (5):826–834, 1983.
18. Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *ICML 2016*, volume 48, pages 201–210. PMLR, 2016.
19. Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 249–256.
20. Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer Series in Statistics. Springer, 2009.
21. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV 2015*, pages 1026–1034.
22. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR 2016*, pages 770–778, 2016.
23. Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *CVPR 2017*, pages 4700–4708, 2017.
24. Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 448–456. JMLR.org, 2015.
25. Akira Ito, Takayuki Miura, and Yosuke Todo. Is the hard-label cryptanalytic model extraction really polynomial? *arXiv preprint arXiv:2510.06692*, 2025.
26. Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High accuracy and high fidelity extraction of neural networks. In *USENIX Security 2020*, pages 1345–1362, 2020.
27. Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACM Multimedia 2014*, pages 675–678.

28. Sangkyu Kim, Sangil Chon, Jin-Kook Kim, Joomin Kim, Yeongjoon Gil, and Sunghoon Jung. Lightweight convolutional neural network for real-time arrhythmia classification on low-power wearable electrocardiograph. In *EMBC 2022*, pages 1915–1918. IEEE.
29. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS 2012*, pages 1106–1114, 2012.
30. Liangzhen Lai, Naveen Suda, and Vikas Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.
31. Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 2002.
32. Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *ICLR 2017, Conference Track Proceedings*. OpenReview.net, 2017.
33. Haolin Liu, Adrien Siproudhis, Christina Boura, and Thomas Peyrin. Model extraction of convolutional neural networks with max-pooling. *Cryptology ePrint Archive*, 2026.
34. Haolin Liu, Adrien Siproudhis, Samuel Experton, Peter Lorenz, Christina Boura, and Thomas Peyrin. Navigating the deep: End-to-end extraction on deep neural networks. *To appear in EUROCRYPT 2026*.
35. Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *CCS 2017*, pages 619–631. ACM.
36. Daniel Lowd and Christopher Meek. Adversarial learning. In *KDD 2005*, pages 641–647. ACM, 2005.
37. Kyle Matoba, Nikolaos Dimitriadis, and François Fleuret. Benefits of max pooling in neural networks: Theoretical and experimental evidence. *Transactions on Machine Learning Research*, 2023.
38. Daryna Oliynyk, Rudolf Mayer, and Andreas Rauber. I know what you trained last summer: A survey on stealing machine learning models and defences. *ACM Computing Surveys*, 55(14s):1–41, 2023.
39. Sang-Soo Park, Jung-Hyun Hong, and Ki-Seok Chung. Modified convolution neural network for highly effective parallel processing. In *IRI 2017, San Diego, CA, USA, August 4-6, 2017*, pages 325–331. IEEE Computer Society, 2017.
40. Thomas Peyrin. Improved differential attacks for ECHO and grøstl. In Tal Rabin, editor, *CRYPTO 2010, Proceedings*, volume 6223, pages 370–392. Springer, 2010.
41. Xiaokang Qi, Hao Lei, Longxiang Wei, Xiaohan Sun, and Meiqin Wang. Cryptanalytic extraction of neural networks with various activation functions. *Cryptology ePrint Archive*, 2026.
42. Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *ICLR 2016*.
43. David Rolnick and Konrad Kording. Reverse-engineering deep relu networks. In *International conference on machine learning*, pages 8178–8187. PMLR, 2020.
44. Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR 2015, San Diego, CA, USA*.
45. Xiaohan Sun, Hao Lei, Longxiang Wei, Xiaokang Qi, Kai Hu, Meiqin Wang, and Wei Wang. Cryptanalytic extraction of convolutional neural networks. Cryptology ePrint Archive, Paper 2026/139, 2026.
46. Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR 2015*, pages 1–9.

47. Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction {APIs}. In *USENIX Security 2016*, pages 601–618, 2016.
48. Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. In *ICCV 2015*, pages 4489–4497. IEEE.
49. 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. *arXiv preprint arXiv:1609.03499*, 2016.
50. Longxiang Wei, Hao Lei, Xiaokang Qi, Xiaohan Sun, Lei Gao, Kai Hu, Wei Wang, and Meiqin Wang. Cryptanalytic extraction of recurrent neural network models. *Cryptology ePrint Archive*, 2026.
51. Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *NIPS 2015*, volume 28, pages 649–657.

Supplementary Material

A A Toy Example of the Algebraic View of CNN

To facilitate a clearer understanding of the algebraic notations introduced in Section 2.2, this appendix provides a concrete, step-by-step toy example. We explicitly demonstrate the complete computational pipeline — including Convolution, ReLU activation, and Max Pooling — for a single-input and single-output channel scenario (*i.e.*, $\mathsf{c}_{in}^{(k)} = \mathsf{c}_{out}^{(k)} = 1$).

Figure 7 visualizes the spatial mapping mechanism of this process. In the following subsections, we formulate this exact process algebraically by completely expanding the equivalent transformation matrices without any truncation.

We also provide illustrative examples of RPCPs and PSPs in Supp. A.5 to help understand the two novel types of critical points in CNNs introduced in Sect. 3, along with a warm-up extraction example for the first layer in Supp. A.6 to demonstrate the RPCP and PSP methods detailed in Sects. 4 and 5.

A.1 Setup and Vector Flattening

Consider an input matrix $IN^{(k)} \in \mathbb{R}^{h_{in}^{(k)} \times w_{in}^{(k)}} = \mathbb{R}^{5\times 5}$. We apply a convolutional layer $f_c^{(k)}$ with a kernel $C^{(k)} \in \mathbb{R}^{h_c^{(k)} \times w_c^{(k)}} = \mathbb{R}^{2\times 2}$, a stride of $s_c^{(k)} = 1$, yielding an intermediate output matrix $O^{(k)}$. According to the definitions in Section 2.2, the spatial dimensions of $O^{(k)}$ are calculated as follows:

$$h_o^{(k)} = \left\lfloor \frac{h_{in}^{(k)} - h_c^{(k)}}{s_c^{(k)}} \right\rfloor + 1 = \left\lfloor \frac{5-2}{1} \right\rfloor + 1 = 4, \tag{36}$$

$$w_o^{(k)} = \left\lfloor \frac{w_{in}^{(k)} - w_c^{(k)}}{s_c^{(k)}} \right\rfloor + 1 = \left\lfloor \frac{5-2}{1} \right\rfloor + 1 = 4. \tag{37}$$

Consequently, the resulting intermediate output matrix is $O^{(k)} \in \mathbb{R}^{4\times 4}$. Subsequently, a ReLU activation layer $\sigma_c^{(k)}$ is applied to $O^{(k)}$, generating an activated output matrix $\hat{O}^{(k)} \in \mathbb{R}^{4\times 4}$.

Following the activation, a max pooling layer $\rho^{(k)}$ with a window size of $h_m^{(k)} \times w_m^{(k)} = 2\times 2$ and a stride of $s_\rho^{(k)} = 2$ processes $\hat{O}^{(k)}$. The spatial dimensions of the final output matrix $IN^{(k+1)}$ for the subsequent layer are determined by:

$$h_\rho^{(k)} = \left\lfloor \frac{h_o^{(k)} - h_m^{(k)}}{s_\rho^{(k)}} \right\rfloor + 1 = \left\lfloor \frac{4-2}{2} \right\rfloor + 1 = 2, \tag{38}$$

$$w_\rho^{(k)} = \left\lfloor \frac{w_o^{(k)} - w_m^{(k)}}{s_\rho^{(k)}} \right\rfloor + 1 = \left\lfloor \frac{4-2}{2} \right\rfloor + 1 = 2. \tag{39}$$

This yields the matrix $IN^{(k+1)} \in \mathbb{R}^{2\times 2}$.

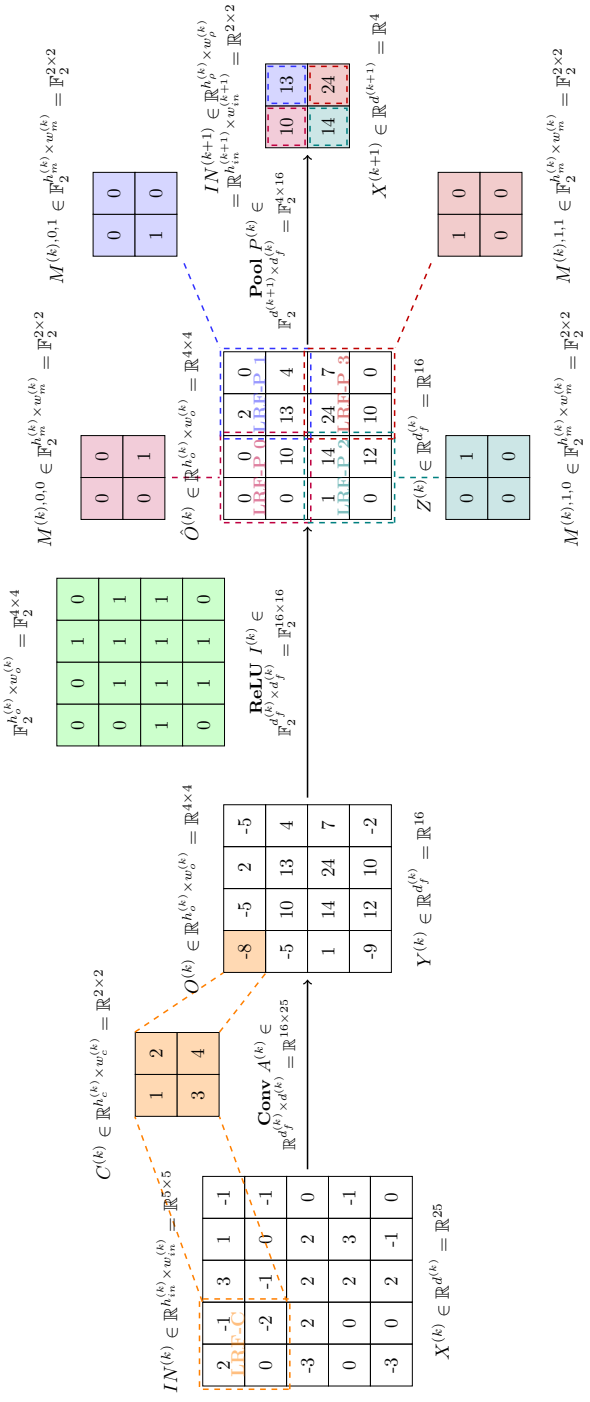


Fig. 7: Overall Visual Illustration of One Convolutional Round.

To represent these operations algebraically, we flatten the spatial matrices into vectors using row-major ordering. For the input matrix $IN^{(k)}$, its i-th row $(0 \le i \le h_{in}^{(k)} - 1)$ is denoted as $IN_i^{(k)} = (a_{i,0}^{(k)}, a_{i,1}^{(k)}, \ldots, a_{i,w_{in}^{(k)}-1}^{(k)})$. Based on the specific numerical values from Figure 7, we instantiate the first and the last row vectors as follows:

$$\begin{aligned} IN_0^{(k)} &= (2, -1, 3, 1, -1), \\ &\vdots \\ IN_4^{(k)} &= (-3, 0, 2, -1, 0). \end{aligned} \tag{40}$$

The flattened input vector $X^{(k)} \in \mathbb{R}^{d^{(k)}} = \mathbb{R}^{h_{in}^{(k)} \cdot w_{in}^{(k)}} = \mathbb{R}^{25}$ is constructed by concatenating these row vectors as $X^{(k)} = (IN_0^{(k)}, IN_1^{(k)}, \ldots, IN_4^{(k)})^\mathsf{T}$, which explicitly evaluates to:

$$X^{(k)} = [2, -1, 3, 1, -1, 0, -2, -1, 0, -1, -3, 2, 2, 2, 0, 0, 0, 2, 3, -1, -3, 0, 2, -1, 0]^\mathsf{T}. \tag{41}$$

A.2 The Convolutional Matrix ($A^{(k)}$)

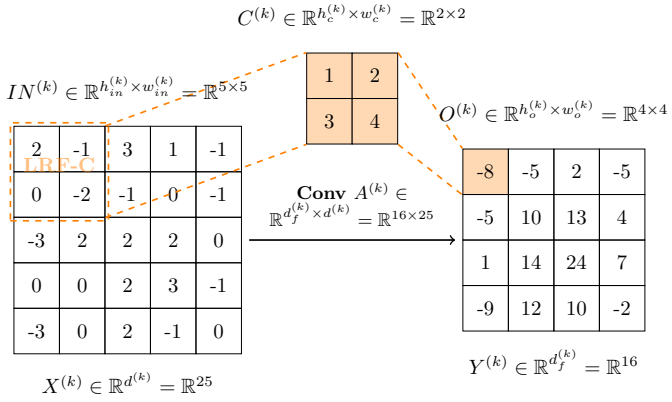


Fig. 8: Local view of the convolution operation $f_c^{(k)}$

Let the 2×2 convolution kernel matrix be $C^{(k)} = \left[\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right]$. For simplicity in this example, we assume the bias is $b^{(k)} = 0$, yielding an all-zero bias vector $B^{(k)} = \mathbf{0}$. Therefore, the convolution operation translates into a linear matrix multiplication $Y^{(k)} = A^{(k)} X^{(k)}$, where $Y^{(k)} \in \mathbb{R}^{d_f^{(k)}} = \mathbb{R}^{h_o^{(k)} \cdot w_o^{(k)}} = \mathbb{R}^{16}$ is the flattened output vector and $A^{(k)} \in \mathbb{R}^{d_f^{(k)} \times d^{(k)}} = \mathbb{R}^{16\times 25}$ is the convolutional matrix.

As illustrated by the orange dashed lines in Figure 8, the first element of the feature map $O_{0,0}^{(k)}$ (which corresponds to $Y_0^{(k)}$) is computed by the sum of the element-wise multiplications between $C^{(k)}$ and the top-left LRF-C. Formally, this is calculated as:

$$\begin{aligned} O_{0,0}^{(k)} &= \sum_{p=0}^{h_c^{(k)}-1} \sum_{q=0}^{w_c^{(k)}-1} IN_{0+p,0+q}^{(k)} C_{p,q}^{(k)} + b^{(k)} \\ &= IN_{0,0}^{(k)} C_{0,0}^{(k)} + IN_{0,1}^{(k)} C_{0,1}^{(k)} + IN_{1,0}^{(k)} C_{1,0}^{(k)} + IN_{1,1}^{(k)} C_{1,1}^{(k)} + b^{(k)} \\ &= (2)(1) + (-1)(2) + (0)(3) + (-2)(4) + 0 = -8. \end{aligned} \tag{42}$$

According to Eq. (5), the first row of the convolutional matrix ($i = 0$) is systematically constructed by substituting the kernel rows into the padded structure. Given $w_{in}^{(k)} = 5$ and $w_o^{(k)} = 4$, the substitution and expansion explicitly demonstrate how the kernel parameters are positioned within $A_0^{(k)}$:

$$\begin{aligned} A_0^{(k)} &= (\underbrace{0, \cdots, 0}_{w_{in}^{(k)} \cdot \lfloor 0/w_o^{(k)} \rfloor}, \underbrace{\underbrace{0, \cdots, 0}_{0 \bmod w_o^{(k)}}, C_0^{(k)}, \underbrace{0, \ldots, 0}_{padding}}_{w_{in}^{(k)}}, \underbrace{\underbrace{0, \cdots, 0}_{0 \bmod w_o^{(k)}}, C_1^{(k)}, \underbrace{0, \ldots, 0}_{padding}, \underbrace{0, \cdots, 0}_{padding}}_{w_{in}^{(k)}}) \\ &= (\underbrace{}_{0}, \underbrace{\underbrace{}_{0}, \mathbf{1}, \mathbf{2}, \underbrace{0, 0, 0}_{3}}_{5}, \underbrace{\underbrace{}_{0}, \mathbf{3}, \mathbf{4}, \underbrace{0, 0, 0}_{3}}_{5}, \underbrace{0, \cdots, 0}_{15}) \\ &= (1, 2, 0, 0, 0, 3, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0). \end{aligned} \tag{43}$$

By representing the exhaustive sliding window operations over the 2D input matrix, the entire equivalent matrix $A^{(k)}$ is expanded below. We highlight the non-zero elements corresponding to the convolution kernel in each row in bold:

$$A^{(k)} = \begin{bmatrix} \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & \mathbf{3} & \mathbf{4} \end{bmatrix} \tag{44}$$

Performing the multiplication yields the pre-activation vector $Y^{(k)}$:

$$Y^{(k)} = A^{(k)} X^{(k)} = [-8, -5, 2, -5, -5, 10, 13, 4, 1, 14, 24, 7, -9, 12, 10, -2]^{\mathsf{T}}. \tag{45}$$

A.3 The ReLU Activation Matrix ($I^{(k)}$)

Fig. 9 shows the operation of ReLU activation. Algebraically, it is represented as $Z^{(k)} = I^{(k)}Y^{(k)}$, where $I^{(k)} \in \mathbb{F}_2^{d_f^{(k)} \times d_f^{(k)}} = \mathbb{F}_2^{16\times 16}$ is a diagonal matrix. According to Eq. (6), the i-th diagonal entry of $I^{(k)}$ is set to 1 if the corresponding element $Y_i^{(k)}$ is strictly positive, and 0 otherwise. Based on the numerical values

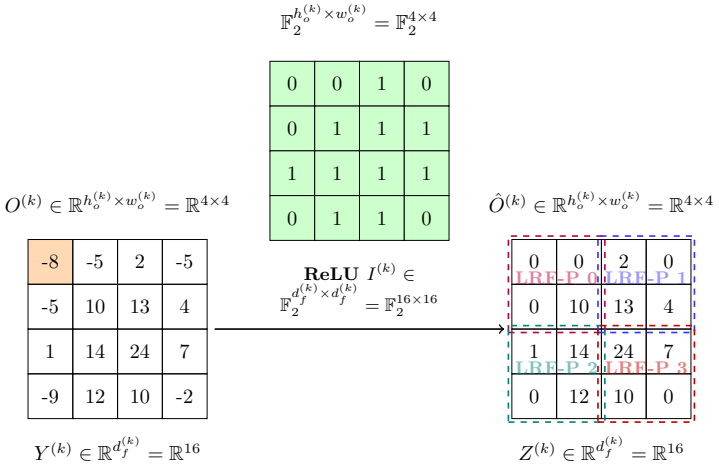


Fig. 9: Local view of the ReLU activation $\sigma_c^{(k)}$

of $Y^{(k)}$ calculated previously, the fully expanded activation matrix is constructed as follows. We highlight the entire main diagonal in bold:

$$I^{(k)} = \begin{bmatrix}
\mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0}
\end{bmatrix} \tag{46}$$

Applying this masking matrix to $Y^{(k)}$ zeros out its non-positive components, yielding the activated vector $Z^{(k)}$:

$$Z^{(k)} = I^{(k)}Y^{(k)} = [0, 0, 2, 0, 0, 10, 13, 4, 1, 14, 24, 7, 0, 12, 10, 0]^{\mathsf{T}}. \tag{47}$$

A.4 The Max Pooling Matrix ($P^{(k)}$)

In the pooling layer $\rho^{(k)}$, the 2×2 max pooling operation with stride 2 partitions the 4×4 intermediate feature map $\hat{O}^{(k)}$ into 4 disjoint LRF-Ps. For each LRF-P, the pooling function selects the maximum value. Fig. 10 illustrates the operation

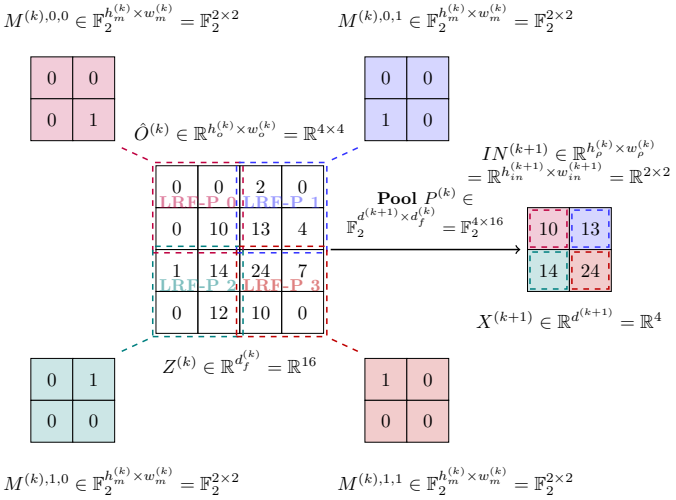


Fig. 10: Local view of the Max Pooling $\rho^{(k)}$

of max pooling. According to Eq. (7), the first element of the pooled output is the maximum value within the $(i \cdot w_\rho^{(k)} + j) = (0 \cdot w_\rho^{(k)} + 0) = 0$-th LRF-P denoted as $\Omega_{0,0}$:

$$IN_{0,0}^{(k+1)} = \max\{\hat{O}_{\Omega_{0,0}}^{(k)}\} = \max(\hat{O}_{0,0}^{(k)}, \hat{O}_{0,1}^{(k)}, \hat{O}_{1,0}^{(k)}, \hat{O}_{1,1}^{(k)}) = \max(0, 0, 0, 10) = 10. \tag{48}$$

Since the maximum value "10" in $\Omega_{0,0}$ is located at local index $(h^* = 1, w^* = 1)$, the Boolean matrix $M^{(k),0,0}$ assigns 1 exclusively to this position, making its row vectors $M_0^{(k),0,0} = (0, 0)$ and $M_1^{(k),0,0} = (0, 1)$. According to Eq. (11), the first row $(r = 0)$ of the pooling matrix is constructed by mapping these Boolean rows of $M^{(k),0,0}$ into the padded structure. Given $w_o^{(k)} = 4$ and $s_\rho^{(k)} = 2$, the

substitution explicitly demonstrates how the selected bits are positioned within $P_0^{(k)}$:

$$\begin{aligned} P_0^{(k)} &= (\underbrace{0,\cdots,0}_{w_o^{(k)}\cdot 0\cdot s_\rho^{(k)}}, \underbrace{\underbrace{0,\cdots,0}_{0\cdot s_\rho^{(k)}}, M_0^{(k),0,0}, \underbrace{0,\ldots,0}_{padding}}_{w_o^{(k)}}, \underbrace{\underbrace{0,\cdots,0}_{0\cdot s_\rho^{(k)}}, M_1^{(k),0,0}, \underbrace{0,\ldots,0}_{padding}}_{w_o^{(k)}}, \underbrace{0,\cdots,0}_{padding}) \\ &= (\underbrace{}_{0}, \underbrace{}_{0}, \underbrace{\mathbf{0},\mathbf{0}}_{4}, \underbrace{0,0}_{2}, \underbrace{}_{0}, \underbrace{\mathbf{0},\mathbf{1}}_{4}, \underbrace{0,0}_{2}, \underbrace{0,\cdots,0}_{8}) \\ &= (0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0). \end{aligned} \tag{49}$$

By extending this construction to all four LRF-Ps, the entire equivalent matrix $P^{(k)}$ extracts exactly one element per row. We highlight these selected positions in bold:

$$P^{(k)} = \begin{bmatrix} 0\,0\,0\,0\,0\,\mathbf{1}\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,\mathbf{1}\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,\mathbf{1}\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,\mathbf{1}\,0\,0\,0\,0\,0 \end{bmatrix} \tag{50}$$

The final output vector for the entire layer is therefore:

$$X^{(k+1)} = P^{(k)} Z^{(k)} = [10, 13, 14, 24]^\mathsf{T}. \tag{51}$$

A.5 Examples of RPCPs and PSPs

RPCPs. Let $k = 1$ in the Figure 7 and modify inputs of the LRF-C 0 of $IN^{(1)}$ to derive Figure 11, which leads to an example of an RPCP. Given the targeted input $X^{(1)}$, $O_{1,1}^{(1)}$ evaluates to exactly 0 ($Y^{(1)}[5] = 0$ in vector form). Meanwhile, the values of the other competing neurons within the LRF-P 0 are strictly negative ($Y_0^{(1)} = -5$, $Y_1^{(1)} = -20$, and $Y_4^{(1)} = -10$). Consequently, the ReLU functions suppress their outputs to 0.

Consider a small perturbation $\vec{\delta}$ such that the inputs $X + \vec{\delta}$ and $X - \vec{\delta}$ fall into two adjacent linear neighborhoods. Assume that $Y^{(1)}[5] > 0$ at $X + \vec{\delta}$ (the active side) and $Y^{(1)}[5] < 0$ at $X - \vec{\delta}$ (the inactive side). Then, the max pooling will select neuron 5 on the active side ($X + \vec{\delta}$). On the inactive side ($X - \vec{\delta}$), the max pooling can still select neuron 5, since all neurons in the LRF-P 0 output 0. Consequently, the corresponding diagonal entry (the 5-th entry) in the activation matrix $I^{(1)}$ can be either 1 or 0.

PSPs. Figure 12 demonstrates a PSP instance, where a "tie" occurs in the LRF-P 0. Specifically, the pre-activation values $Y^{(1)}[4]$ and $Y^{(1)}[5]$ (corresponding to $O_{1,0}^{(1)}$ and $O_{1,1}^{(1)}$) both evaluate to the exact same maximum value of 19.

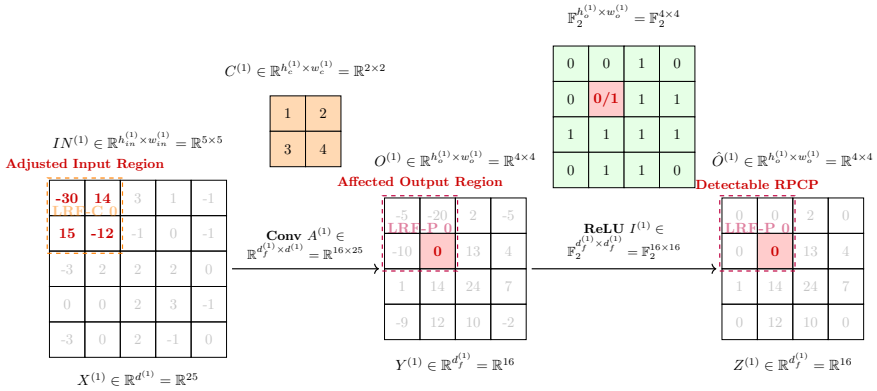


Fig. 11: An example of an RPCP. For the targeted input $X^{(1)}$, $Y^{(1)}[5] = 0$ and $Y^{(1)}[0],\ Y^{(1)}[1],\ Y^{(1)}[4] < 0$. Consequently, the corresponding diagonal entry in the activation matrix $I^{(1)}$ can be either 1 or 0.

This equality forces the local pooling selection matrix $M^{(1),0,0}$ to switch between selecting the indices $i = 4$ and $j = 5$ when subjected to tiny input perturbations, such as $X^{(1)} + \vec{\delta}$ and $X^{(1)} - \vec{\delta}$. At exact PSP $X^{(1)}$, the max pooling can select either neuron 4 or neuron 5 since the output is always 19. Consequently, in $M^{(1),0,0}$, we can have either $M^{(1),1,0} = 1$ or $M^{(1),1,1} = 1$, with all other entries evaluating to 0.

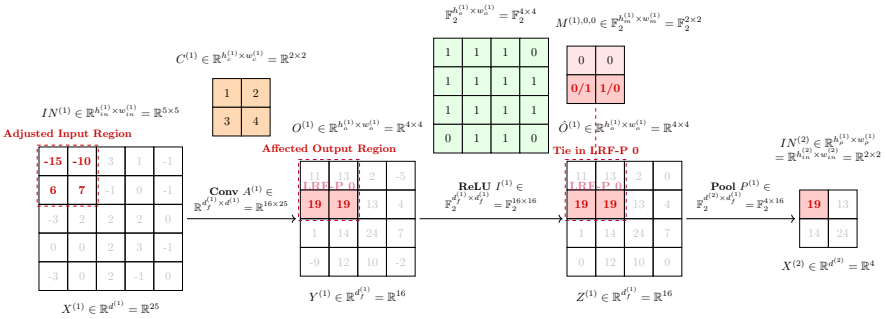


Fig. 12: An example of a PSP. The input $X^{(1)}$ induces a tie ($Y^{(1)}[4] = Y^{(1)}[5] = 19$) within the LRF-P 0. This makes the pooling selection matrix $M^{(1),0,0}$ to select between either index $i = 4$ or $j = 5$.

A.6 Warm-up: Extracting the First Layer

The RPCP method. As illustrated in Figure 11, the neuron 5 is in the critical state. By querying the model with perturbed inputs $X^{(1)} + \vec{\delta}$ and $X^{(1)} - \vec{\delta}$, the neuron flips between positive value and negative value, with the corresponding activation matrix having a 1 at the 5-th diagonal entry ($I_+^{(1)}$) on the active side and 0 ($I_-^{(1)}$) on the inactive side. As formulated in Eq. (18), by calculating the second-order differential function $\mathcal{H}(X;\vec{\delta})$ of $\mathcal{H}(X;\vec{\delta}) = \mathcal{F}_\theta(X+\vec{\delta}) + \mathcal{F}_\theta(X-\vec{\delta}) - 2\mathcal{F}_\theta(X)$, we have:

$$\mathcal{H}(X^{(1)};\vec{\delta}) = G^2 P^{(1)}(I_+^{(1)} - I_-^{(1)})A^{(1)}\vec{\delta} \tag{52}$$

Note that

$$I_+^{(1)} - I_-^{(1)} = \begin{bmatrix}
\mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0}
\end{bmatrix}, \tag{53}$$

and

$$P^{(1)} = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0
\end{bmatrix}. \tag{54}$$

The sparse structure of $(I_+^{(1)} - I_-^{(1)})$ mathematically acts as a column selector. When multiplied by the pooling matrix $P^{(1)}$, it nullifies all columns except the 5-th one, yielding:

$$P^{(1)}(I_+^{(1)} - I_-^{(1)}) = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}. \tag{55}$$

Subsequently, multiplying this resulting matrix by the weight matrix $A^{(1)}$ transforms it into a row selector. Specifically, it perfectly isolates the 5-th row of $A^{(1)}$ into the top row of the resulting matrix, while leaving all other rows as zeros:

$$P^{(1)}(I_+^{(1)} - I_-^{(1)})A^{(1)} = \begin{bmatrix}
A_{5,0}^{(1)} & A_{5,1}^{(1)} & \cdots & A_{5,24}^{(1)} \\
0 & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0
\end{bmatrix}. \tag{56}$$

Finally, let $G^2 = [g_0, g_1, g_2, g_3]$ represent the aggregated linear transformation vector from the subsequent layers. Multiplying G^2 by the isolated matrix projects the 5-th row of $A^{(1)}$ into a single vector scaled by g_0:

$$G^2 P^{(1)}(I_+^{(1)} - I_-^{(1)})A^{(1)} = [g_0, g_1, g_2, g_3] \begin{bmatrix} A_{5,0}^{(1)} & A_{5,1}^{(1)} & \cdots & A_{5,24}^{(1)} \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \end{bmatrix} = (g_0 A_{5,0}^{(1)}, g_0 A_{5,1}^{(1)}, \cdots, g_0 A_{5,24}^{(1)}).$$

i.e.,

$$G^2 P^{(1)}(I_+^{(1)} - I_-^{(1)})A^{(1)} = g_0 A_5^{(1)} \tag{57}$$

By sampling a set of linearly independent perturbations $S = (\vec{\delta}^{(0)}, \vec{\delta}^{(1)}, \ldots, \vec{\delta}^{(n-1)})^\intercal$, we construct a linear system according to Eq. (52). Solving this system directly yields a sparse vector proportional to $A_5^{(1)}$. Based on the equivalent matrix expansion demonstrated in Eq. (44), and substituting the true convolutional kernel parameters $C = \left[\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right]$, the explicit structure of $A_5^{(1)}$ is revealed as a 25-dimensional vector:

$$A_5^{(1)} = (0, 0, 0, 0, 0,\ \ 0, \mathbf{1}, \mathbf{2}, 0, 0,\ \ 0, \mathbf{3}, \mathbf{4}, 0, 0,\ \ 0, \ldots, 0) \tag{58}$$

This explicit algebraic structure immediately exposes the true parameter ratios $(1 : 2 : 3 : 4)$ of the convolution kernel $C^{(1)}$ according to Eqs. (43) and (44).

The PSP method. As illustrated in Fig. 12, a PSP $X^{(1)}$ occurs when two competing neurons within the LRF-P 0 reach a same maximum value.

By querying the model with perturbed inputs $X^{(1)} + \vec{\delta}$ and $X^{(1)} - \vec{\delta}$, the max pooling selection switches between neurons 4 and 5. Consequently, the first row vector of the pooling matrix $P^{(1)}$ has a 1 at the 4-th entry ($P_+^{(1)}$) for one perturbation direction (*e.g.,* $X^{(1)} + \vec{\delta}$) and a 1 at the 5-th entry ($P_-^{(1)}$) for the opposite direction ($X^{(1)} - \vec{\delta}$). As formulated in Eq. (33), by calculating the second-order differential function $\mathcal{H}(X; \vec{\delta})$ defined as $\mathcal{H}(X; \vec{\delta}) = \mathcal{F}_\theta(X + \vec{\delta}) + \mathcal{F}_\theta(X - \vec{\delta}) - 2\mathcal{F}_\theta(X)$, we have:

$$\mathcal{H}(X^{(1)}; \vec{\delta}) = G^2(P_+^{(1)} - P_-^{(1)})I^{(1)}A^{(1)}\vec{\delta}. \tag{59}$$

Assuming the max pooling selections in the remaining LRF-Ps are unchanged upon the tiny perturbations, their corresponding rows in $P_+^{(1)}$ and $P_-^{(1)}$ are identical. Thus, the difference matrix evaluates strictly to zeros except for the first row:

$$P_+^{(1)} - P_-^{(1)} = \begin{bmatrix} 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{-1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{60}$$

Because neurons 4 and 5 are both strictly positive (achieving the maximum value of 19), the diagonal entries for these indices in the activation matrix $I^{(1)}$ are both 1. Thus, left-multiplying the weight matrix $A^{(1)}$ by $(P_+^{(1)} - P_-^{(1)})I^{(1)}$

acts as a precise row selector that extracts the difference between the 4-th and 5-th rows of $A^{(1)}$:

$$(P_+^{(1)} - P_-^{(1)})I^{(1)}A^{(1)} = \begin{bmatrix} A_4^{(1)} - A_5^{(1)} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}. \tag{61}$$

Subsequently, left-multiplying by the aggregated linear transformation vector $G^2 = [g_0, g_1, g_2, g_3]$ isolates this row difference, scaled by the scalar g_0:

$$G^2(P_+^{(1)} - P_-^{(1)})I^{(1)}A^{(1)} = g_0(A_4^{(1)} - A_5^{(1)}). \tag{62}$$

By solving the linear system constructed from multiple perturbed queries, we recover a vector proportional to the difference $A_4^{(1)} - A_5^{(1)}$. Substituting the specific kernel parameters $(1, 2, 3, 4)$ into our expanded matrix (Eq. (44)), we can explicitly visualize the alignment of these two rows:

$$A_4^{(1)} = (0, 0, 0, 0, 0,\ \mathbf{1}, \mathbf{2}, 0, 0, 0,\ \mathbf{3}, \mathbf{4}, 0, 0, 0,\ 0, \ldots, 0), \tag{63}$$

$$A_5^{(1)} = (0, 0, 0, 0, 0,\ 0, \mathbf{1}, \mathbf{2}, 0, 0,\ 0, \mathbf{3}, \mathbf{4}, 0, 0,\ 0, \ldots, 0). \tag{64}$$

Consequently, their difference perfectly captures the shifted structural pattern of the convolution window:

$$\begin{aligned} A_4^{(1)} - A_5^{(1)} &= (0, 0, 0, 0, 0,\ \mathbf{1},\ \mathbf{2}-\mathbf{1},\ -\mathbf{2},\ 0, 0,\ \mathbf{3},\ \mathbf{4}-\mathbf{3},\ -\mathbf{4},\ 0, 0,\ 0, \ldots, 0) \\ &= (0, 0, 0, 0, 0,\ \mathbf{1},\ \mathbf{1},\ -\mathbf{2},\ 0, 0,\ \mathbf{3},\ \mathbf{1},\ -\mathbf{4},\ 0, 0,\ 0, \ldots, 0). \end{aligned} \tag{65}$$

The non-zero elements in this difference vector in Eq. (62) uniquely identify the spatial indices $i = 4$ and $j = 5$. Once identified, we apply the internal differential property of the PSP defined in Eq. (30). This relationship elegantly maps the kernel weights to the input difference:

$$C_{0,0}(X_5^{(1)} - X_6^{(1)}) + C_{0,1}(X_6^{(1)} - X_7^{(1)}) + C_{1,0}(X_{10}^{(1)} - X_{11}^{(1)}) + C_{1,1}(X_{11}^{(1)} - X_{12}^{(1)}) = 0. \tag{66}$$

To demonstrate its exact correctness, we substitute the specific spatial values from our adjusted input vector $X^{(1)}$ (where $X_5^{(1)} = 6, X_6^{(1)} = 7, X_7^{(1)} = -1, X_{10}^{(1)} = -3, X_{11}^{(1)} = 2, X_{12}^{(1)} = 2$):

$$\begin{aligned} C_{0,0}(6-7) + C_{0,1}(7-(-1)) + C_{1,0}(-3-2) + C_{1,1}(2-2) = 0, \\ -1 \cdot C_{0,0} + 8 \cdot C_{0,1} - 5 \cdot C_{1,0} + 0 \cdot C_{1,1} = 0. \end{aligned} \tag{67}$$

As expected, the true kernel parameters $(1, 2, 3, 4)$ perfectly satisfy this derived equation: $-1(1) + 8(2) - 5(3) + 0(4) = 0$. By collecting slightly more than $h_c^{(1)} \times w_c^{(1)} = 4$ such independent PSP instances, the adversary establishes a full-rank linear system. This directly and explicitly yields the exact kernel weights $C^{(1)}$.

B Attack on LeNet-5 Style Architectures

This section focuses on the practical convolutional neural network adopting the LeNet-5 Style Architecture [31], which was first proposed by LeCun in 1998. We demonstrate the complete process of applying our proposed algebraic attack to extract the model parameters.

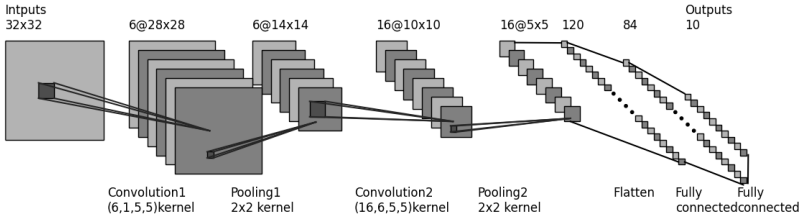


Fig. 13: LeNet-5 Architecture

B.1 LeNet-5 Style Architecture

As shown in Figure 13, the classic LeNet-5 [31] consists of two Convolutional Rounds and two FCNN Rounds, employing a sparse channel connection in its second convolutional layer, Tanh activation and Average Pooling. However, the sparse structure becomes a computational bottleneck for parallel GPU processing [32,39] and restricts complex feature extraction [16]. Activation functions (Sigmoid/Tanh) pose a higher risk of vanishing and exploding gradients [19] and slow convergence [29]. Consequently, modern implementations of LeNet-5 and other CNNs with similar architecture typically incorporate the following modifications [29,27,44,46]:

- **Convolutional Connectivity:** The original asymmetric sparse connections are replaced with full channel connections. In modern frameworks, the second layer kernel is standardized to a size of $(16, 6, 5, 5)$, with standard convolution stride $s_c^{(k)} = 1$ and pooling stride $s_\rho^{(k)} = 2$, $(k = 1, 2)$.
- **Activation Function:** The Tanh function is replaced by ReLU.
- **Pooling Strategy:** Average pooling is substituted with max pooling to capture the most prominent features.

In this section, our practical attack targets this modern architecture with raw output accessible. By definition, the victim model is a $(2 + 2)$-Deep CNN.

B.2 Multiple Channels

LeNet-5 adopts multiple input and output channels in the Convolutional Block. Suppose in layer k, the convolution layer has $\mathsf{c}_{in}^{(k)}$ input channels and $\mathsf{c}_{out}^{(k)}$ output

channels, *i.e.*, there are $\mathsf{c}_{in}^{(k)}$ input matrices, denoted as $IN^{(k),0}, \cdots, IN^{(k),\mathsf{c}_{in}^{(k)}-1}$, and $\mathsf{c}_{out}^{(k)}$ output matrices, denoted as $O^{(k),0}, \cdots, O^{(k),\mathsf{c}_{out}^{(k)}-1}$. An example of convolutional layer with $\mathsf{c}_{in}^{(k)} = 3$ input channels, $\mathsf{c}_{out}^{(k)} = 2$ output channels is shown in Figure 14. Formally, denote the kernel matrices as $C^{(k),m,n} =$

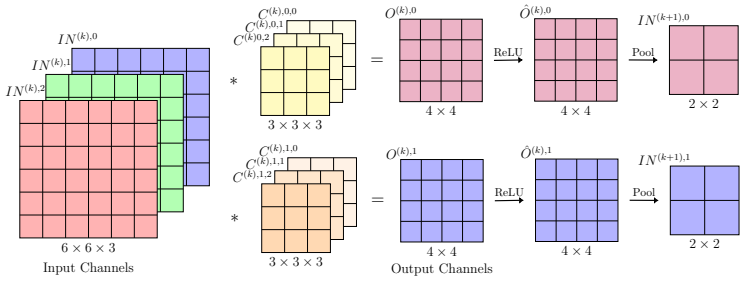


Fig. 14: Multiple Input and Output Channels: $\mathsf{c}_{in}^{(k)} = 3,\ \mathsf{c}_{out}^{(k)} = 2$.

$[C_{i,j}^{(k),m,n}]_{h_c^{(k)} \times w_c^{(k)}}$, where $0 \leq m \leq \mathsf{c}_{out}^{(k)} - 1$ and $0 \leq n \leq \mathsf{c}_{in}^{(k)} - 1$. The entries $O_{i,j}^{(k),m}$ $(0 \leq m \leq \mathsf{c}_{out}^{(k)} - 1,\ 0 \leq i \leq h_o^{(k)} - 1,\ 0 \leq j \leq w_o^{(k)} - 1)$ of the output matrices generated by the convolution operation are defined as:

$$O_{i,j}^{(k),m} = \sum_{n=0}^{\mathsf{c}_{in}^{(k)}-1} \sum_{p=0}^{h_c^{(k)}-1} \sum_{q=0}^{w_c^{(k)}-1} IN_{i+p,j+q}^{(k),n} C_{p,q}^{(k),m,n} + b^{(k),m}, \tag{68}$$

where $b^{(k),m}$ are the biases for the m-th output channel, and the stride $s_c^{(k)} = 1$.

First Layer of LeNet-5. As shown in Figure 15, the input and output channels of the first layer LeNet-5 are $\mathsf{c}_{in}^{(k)} = 1,\ \mathsf{c}_{out}^{(k)} = 6$. The sizes of the output matrices $O^{(1),m}$ $(0 \leq m \leq 5)$ are $h_o^{(k)} \times w_o^{(k)} = 5 \times 5$. In vector form, the operation of the first convolutional layer can be modeled as applying six distinct convolution kernel matrices to the input vector $X^{(1)}$. The 6 output matrices are then flattened and sequentially concatenated to form a single vector $Y^{(1)}$ as shown on the right side of Figure 15. Consequently, the convolutional matrix $A^{(1)}$ for this layer can be constructed by vertically stacking (or concatenating) the six corresponding convolutional sub-matrices, as shown in Figure 15. More details are given in the following 2nd layer of LeNet-5.

Second Layer of LeNet-5. Similarly, from an algebraic perspective, the operation of the second convolutional layer involves 6 input channels and 16 output channels, *i.e.*, $\mathsf{c}_{in}^{(2)} = 6, \mathsf{c}_{out}^{(2)} = 16$. The 6 input matrices $IN^{(2),0-5}$ are transformed into a single vector $X^{(2)} = (X^{(2),0}, X^{(2),1}, ..., X^{(2),n}, ..., X^{(2),5})^{\mathsf{T}}$, where $X^{(2),n} = (IN_0^{(2),n}, .., IN_{h_{in}^{(2)}-1}^{(2),n})^{\mathsf{T}}$. According to Eq. (68), $X^{(2)}$ is convolved

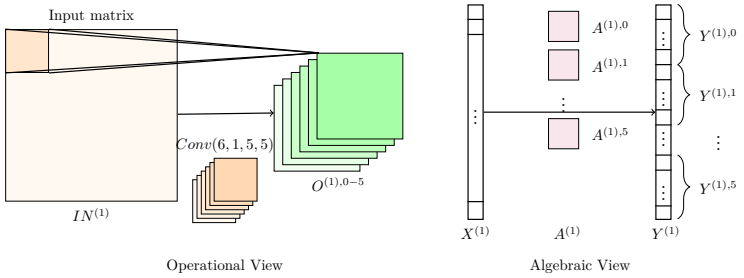


Fig. 15: Operational View and Algebraic View, $(\mathsf{c}_{out}^{(k)}, \mathsf{c}_{in}^{(k)}, h_c^{(k)}, w_c^{(k)}) = (6, 1, 5, 5)$.

with $A^{(2),m}$ (the kernel corresponding to the m-th output channel), and the 6 input channels are accumulated element-wise to generate the target 16 feature matrices $O^{(2),m}$ $(0 \leq m \leq 15)$. Flattening the feature matrix $O^{(2),m}$, $Y^{(2),m} = (O_0^{(2),m}, ..., O_{h_o^{(2)}-1}^{(2),m})^\mathsf{T}$. Consequently, the single $(d_f^{(2)} = \mathsf{c}_{out}^{(2)} \cdot h_o^{(2)} \cdot w_o^{(2)})$-dimensional output vector $Y^{(2)}$, which is composed of multiple output channels, is denoted as $Y^{(2)} = (Y^{(2),0}, Y^{(2),1}, \cdots, Y^{(2),15})^\mathsf{T}$, as shown in Figure 16.

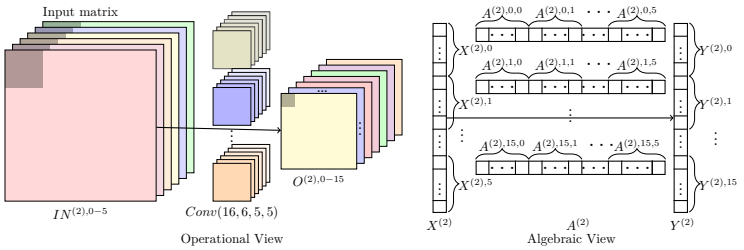


Fig. 16: Operational View and Algebraic View of multiple input and output channels convolutional operations. $(\mathsf{c}_{out}^{(k)}, \mathsf{c}_{in}^{(k)}, h_c^{(k)}, w_c^{(k)}) = (16, 6, 5, 5)$.

For the i-th output channel, this operation is equivalent to horizontally concatenating 6 individual convolutional matrices into a matrix $A^{(2),i} = (A^{(2),i,0}, A^{(2),i,1}, \cdots, A^{(2),i,5})$ $(0 \leq i \leq 15)$, where $A^{(2),i,j}$ $(0 \leq j \leq 5)$ is derived by Def. 2. For the full output of the second convolutional layer, 16 such matrices $A^{(2),i}$ $(0 \leq i \leq 15)$ are vertically stacked to form the final convolutional matrix $A^{(2)}$.

Based on our analysis, we can derive the algebraic expression of the multi-channel convolutional layer: Suppose that for the i-th output channel and the j-th input channel, the convolutional matrix of the corresponding single convolution kernel is $A^{(k),i,j}$ derived by Def. 2, then the full convolutional matrix of

the entire k-th convolutional layer is

$$A^{(k)} = \begin{pmatrix} A^{(k),0,0} & A^{(k),0,1} & \cdots & A^{(k),0,\mathrm{c}_{in}^{(k)}-1} \\ A^{(k),1,0} & A^{(k),1,1} & \cdots & A^{(k),1,\mathrm{c}_{in}^{(k)}-1} \\ \vdots & \vdots & \ddots & \vdots \\ A^{(k),\mathrm{c}_{out}^{(k)}-1,0} & A^{(k),\mathrm{c}_{out}^{(k)}-1,1} & \cdots & A^{(k),\mathrm{c}_{out}^{(k)}-1,\mathrm{c}_{in}^{(k)}-1} \end{pmatrix}. \tag{69}$$

Therefore, the multiple channels can also be transformed into the regular operations *i.e.*, the input vector is multiplied with a matrix to derive the output vector. Hence, Def. 6 also holds in this setting, and our algebraic attacks based on the RPCP or PSP method proposed in Sect. 4 and 5 also work.

C More Detailed Information in Experiments

In this section, we demonstrate the full details of our experimental results in Sect. C.1 and describe the detailed structure information of the models we experimented on in Sect. C.2.

C.1 Experimental Results

We include a more concise version of our experimental results in Section 1 (*i.e.*, Table 1). For the complete experimental result metrics – including runtime, model queries, and error evaluation $(\varepsilon, 0)$ and $\max|\theta - \hat{\theta}|$ – we provide these full details in Table 3.

C.2 Detailed Model Structure Information in Experiments

(1+1)-Deep CNN:

1. Convolutional Round 1:
   - **Convolutional layer** $f_c^{(1)}$: 8×8 Input matrix $IN^{(1)}$, convolved with $(1, 1, 3, 3)$ convolution kernel, with stride $s_c^{(1)} = 1$, padding $pad^{(1)} = 0$, produces 6×6 output matrix $O^{(1)}$, followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(1)}$: 2×2 max pooling kernel, with stride $s_\rho^{(1)} = 2$, produces 3×3 output matrix $IN^{(2)}$.
2. FCNN Round 1:
   - Flatten the matrix $IN^{(2)}$ and get the 9-dimensional vector $X^{(1)}$.
   - **Fully-connected Layer** $f^{(1)}$: $d^{(1)} = 9$ input neurons and $d^{(2)} = 5$ output neurons, followed by ReLU activation.
3. Last Layer $f^{(2)}$: $d^{(2)} = 5$ input neurons and $d^{(3)} = 10$ output neurons.

Table 3: Experiments on different CNNs.

Models $(m+n)$	**Trainset**	**Architecture** $d^{(k)}$–$d_f^{(k)}$–$d^{(k+1)}$	**Kernel**†	**Param**	**Runtime** Seconds	**Queries**	$(\varepsilon, 0)$‡	$\max\lvert\theta-\hat{\theta}\rvert$‡
(**1**+1)	Random	Conv R1: 64–36–9	(1,1,3,3)	10	$2^{-0.26}$	$2^{9.98}$	$2^{-43.19}$	$2^{-47.52}$
(**2**+1)	MNIST	Conv R1: 1024–784–196 Conv R2: 196–100–25	(1,1,5,5) (1,1,5,5)	52	$2^{5.97}$	$2^{15.06}$	$2^{-30.20}$	$2^{-34.65}$
(**2**+2)*	MNIST	Conv R1: 1024–4704–1176 Conv R2: 1176–1600–400	(1,6,5,5) (6,16,5,5)	2572	$2^{17.21}$	$2^{27.29}$	$2^{-23.38}$	$2^{-27.31}$
(**3**+1)^{p2}	MNIST	Conv R1: 1024–1024–256 Conv R2: 256–256–64 Conv R3: 64–64–16	(1,1,5,5) (1,1,5,5) (1,1,5,5)	78	$2^{8.76}$	$2^{16.90}$	$2^{-27.54}$	$2^{-31.15}$
(**2**+2)^{p1}	CIFAR10	Conv R1: 3072–3072–768 Conv R2: 768–1024–256	(3,3,3,3) (3,4,3,3)	196	$2^{10.96}$	$2^{18.37}$	$2^{-29.13}$	$2^{-36.19}$
(**2**+1)♣	MNIST	Conv R1: 1024–4704–1176 Conv R2: 1176–1600–400 FCNN: 400–20–10	(1,6,5,5) (6,16,5,5) –	10772 (2572+8230)	$2^{17.38}$	$2^{27.65}$	$2^{-22.06}$	$2^{-27.14}$
(**2**+**2**)*▲	MNIST	**C1**: 1024–4704–1176 **C2**: 1176–1600–400 FCNN: **400–120**–84–10	(1,6,5,5) (6,16,5,5) –	61706 (2572+59134)	$2^{18.46}$	$2^{27.31}$	–	$2^{-26.03}$
(0+**3**) [9]	MNIST	FCNN: 40–20–10–10–1	–	1110	–	$2^{17.8}$	$2^{-23.4}$	$2^{-27.1}$
(0+**3**) [15] ♠	MNIST	FCNN: 784–16–**16–16**–1	–	272	$2^{16.0}$	$2^{25.4}$	–	–

*: LeNet-5 modern model, specified in in Sect. B.1.

p_1, p_2: p_i indicates $pad^{(k)} = i$ defined as "same padding" in Supplementary Material F; otherwise, $pad^{(k)} = 0$.

†: Convolutional kernel size of $(c_{in}^{(k)}, c_{out}^{(k)}, h_c^{(k)}, w_c^{(k)})$, with stride $s_c^{(k)} = 1$.

‡: The accuracy term $(\varepsilon, 0)$ is defined in Sect. 2.3, $\max\lvert\theta-\hat{\theta}\rvert$ directly measures the maximum extraction error of model parameters.

♣: Reduced version of LeNet-5 with only one FCNN Round.

▲: Red bold layer in Architecture indicates end-to-end extracting the full Convolutional Block and 93.33% weights of FCNN Round 1 (not full due to existing dead neurons and persistent neurons according to [25]).

♠ : Red bold layer in Architecture indicates extracting the single layer, with prior layers' extractions assumed to be correct.

(2+1)-Deep CNN:

1. Convolutional Round 1:
   - **Convolutional layer** $f_c^{(1)}$: 32×32 Input matrix $IN^{(1)}$, convolved with $(1, 1, 5, 5)$ convolution kernel, with stride $s_c^{(1)} = 1$, padding $pad^{(1)} = 0$, produces 28×28 output matrix $O^{(1)}$, followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(1)}$: 2×2 max pooling kernel, with stride $s_\rho^{(1)} = 2$, produces 14×14 output matrix $IN^{(2)}$.
2. Convolutional Round 2:
   - **Convolutional layer** $f_c^{(2)}$: 14×14 Input matrix $IN^{(2)}$, convolved with $(1, 1, 5, 5)$ convolution kernel, with stride $s_c^{(2)} = 1$, padding $pad^{(2)} = 0$, produces 10×10 output matrix $O^{(2)}$, followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(2)}$: 2×2 max pooling kernel, with stride $s_\rho^{(2)} = 2$, produces 5×5 output matrix $IN^{(3)}$.
3. FCNN Round 1:
   - Flatten the matrix $IN^{(3)}$ and get the 25-dimensional vector $X^{(1)}$.
   - **Fully-connected Layer** $f^{(1)}$: $d^{(1)} = 25$ input neurons and $d^{(2)} = 18$ output neurons, followed by ReLU activation.

4. Last Layer $f^{(2)}$: $d^{(2)} = 18$ input neurons and $d^{(3)} = 10$ output neurons.

(2+2)-Deep LeNet-5:

1. Convolutional Round 1:
   - **Convolutional layer** $f_c^{(1)}$: 32×32 Input matrix $IN^{(1)}$, convolved with $(6, 1, 5, 5)$ convolution kernel, with stride $s_c^{(1)} = 1$, padding $pad^{(1)} = 0$, produces 6 output matrices $O^{(1)}$ of size 28×28 , followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(1)}$: 2×2 max pooling kernel, with stride $s_\rho^{(1)} = 2$, produces 6 output matrices $IN^{(2)}$ of size 14×14 .
2. Convolutional Round 2:
   - **Convolutional layer** $f_c^{(2)}$: 6 Input matrix $IN^{(2)}$ of size 14×14, convolved with $(6, 16, 5, 5)$ convolution kernel, with stride $s_c^{(2)} = 1$, padding $pad^{(2)} = 0$, produces 16 output matrix $O^{(2)}$ of size 10×10, followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(2)}$: 2×2 max pooling kernel, with stride $s_\rho^{(2)} = 2$, produces 16 output matrix $IN^{(3)}$ of size 5×5.
3. FCNN Round 1:
   - Flatten the matrix $IN^{(3)}$ and get the 400-dimensional vector $X^{(1)}$.
   - **Fully-connected Layer** $f^{(1)}$: $d^{(1)} = 400$ input neurons and $d^{(2)} = 120$ output neurons, followed by ReLU activation.
4. FCNN Round 2:
   - **Fully-connected Layer** $f^{(2)}$: $d^{(2)} = 120$ input neurons and $d^{(3)} = 84$ output neurons, followed by ReLU activation.
5. Last Layer $f^{(3)}$: $d^{(3)} = 84$ input neurons and $d^{(3)} = 10$ output neurons.

(3+1)-Deep CNN:

1. Convolutional Round 1:
   - **Convolutional layer** $f_c^{(1)}$: 32×32 Input matrix $IN^{(1)}$, convolved with $(1, 1, 5, 5)$ convolution kernel, with stride $s_c^{(1)} = 1$, padding $pad^{(1)} = 2$, produces 32×32 output matrix $O^{(1)}$, followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(1)}$: 2×2 max pooling kernel, with stride $s_\rho^{(1)} = 2$, produces 16×16 output matrix $IN^{(2)}$.
2. Convolutional Round 2:
   - **Convolutional layer** $f_c^{(2)}$: 16×16 Input matrix $IN^{(2)}$, convolved with $(1, 1, 5, 5)$ convolution kernel, with stride $s_c^{(2)} = 1$, padding $pad^{(2)} = 2$, produces 16×16 output matrix $O^{(2)}$, followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(2)}$: 2×2 max pooling kernel, with stride $s_\rho^{(2)} = 2$, produces 8×8 output matrix $IN^{(3)}$.
3. Convolutional Round 3:
   - **Convolutional layer** $f_c^{(3)}$: 8×8 Input matrix $IN^{(3)}$, convolved with $(1, 1, 5, 5)$ convolution kernel, with stride $s_c^{(3)} = 1$, padding $pad^{(3)} = 2$, produces 8×8 output matrix $O^{(3)}$, followed by ReLU activation.

   - **Max Pooling layer** $\rho^{(3)}$: 2×2 max pooling kernel, with stride $s_\rho^{(3)} = 2$, produces 4×4 output matrix $IN^{(4)}$.
4. FCNN Round 1:
   - Flatten the matrix $IN^{(4)}$ and get the 16-dimensional vector $X^{(1)}$.
   - **Fully-connected Layer** $f^{(1)}$: $d^{(1)} = 16$ input neurons and $d^{(2)} = 25$ output neurons, followed by ReLU activation.
5. Last Layer $f^{(2)}$: $d^{(2)} = 25$ input neurons and $d^{(3)} = 10$ output neurons.

(2+2)-Deep CNN trained on CIFAR 10:

1. Convolutional Round 1:
   - **Convolutional layer** $f_c^{(1)}$: 3 Input matrix $IN^{(1)}$ of size 32×32, convolved with $(1, 1, 3, 3)$ convolution kernel, with stride $s_c^{(1)} = 1$, padding $pad^{(1)} = 1$, produces 3 output matrix $O^{(1)}$ of size 32×32 , followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(1)}$: 2×2 max pooling kernel, with stride $s_\rho^{(1)} = 2$, produces 3 output matrix $IN^{(2)}$ of size 16×16 .
2. Convolutional Round 2:
   - **Convolutional layer** $f_c^{(2)}$: 3 Input matrix $IN^{(2)}$ of size 16×16, convolved with $(3, 4, 3, 3)$ convolution kernel, with stride $s_c^{(2)} = 1$, padding $pad^{(2)} = 1$, produces 4 output matrix $O^{(2)}$ of size 16×16, followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(2)}$: 2×2 max pooling kernel, with stride $s_\rho^{(2)} = 2$, produces 4 output matrix $IN^{(3)}$ of size 8×8.
3. FCNN Round 1:
   - Flatten the matrix $IN^{(3)}$ and get the 256-dimensional vector $X^{(1)}$.
   - **Fully-connected Layer** $f^{(1)}$: $d^{(1)} = 256$ input neurons and $d^{(2)} = 256$ output neurons, followed by ReLU activation.
4. FCNN Round 2:
   - **Fully-connected Layer** $f^{(2)}$: $d^{(2)} = 256$ input neurons and $d^{(3)} = 120$ output neurons, followed by ReLU activation.
5. Last Layer $f^{(3)}$: $d^{(3)} = 120$ input neurons and $d^{(3)} = 10$ output neurons.

(2+1)-Deep LeNet-5:

1. Convolutional Round 1:
   - **Convolutional layer** $f_c^{(1)}$: 32×32 Input matrix $IN^{(1)}$, convolved with $(6, 1, 5, 5)$ convolution kernel, with stride $s_c^{(1)} = 1$, padding $pad^{(1)} = 0$, produces 6 output matrices $O^{(1)}$ of size 28×28 , followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(1)}$: 2×2 max pooling kernel, with stride $s_\rho^{(1)} = 2$, produces 6 output matrices $IN^{(2)}$ of size 14×14 .
2. Convolutional Round 2:
   - **Convolutional layer** $f_c^{(2)}$: 6 Input matrices $IN^{(2)}$ of size 14×14, convolved with $(6, 16, 5, 5)$ convolution kernel, with stride $s_c^{(2)} = 1$, padding $pad^{(2)} = 0$, produces 16 output matrices $O^{(2)}$ of size 10×10, followed by ReLU activation.

- **Max Pooling layer** $\rho^{(2)}$: 2×2 max pooling kernel, with stride $s_\rho^{(2)} = 2$, produces 16 output matrix $IN^{(3)}$ of size 5×5.

3. FCNN Round 1:
   - Flatten the matrix $IN^{(3)}$ and get the 400-dimensional vector $X^{(1)}$.
   - **Fully-connected Layer** $f^{(1)}$: $d^{(1)} = 400$ input neurons and $d^{(2)} = 20$ output neurons, followed by ReLU activation.
4. Last Layer $f^{(2)}$: $d^{(2)} = 20$ input neurons and $d^{(3)} = 10$ output neurons.

Modern (2+2)-Deep LeNet-5:

1. Convolutional Round 1:
   - **Convolutional layer** $f_c^{(1)}$: 32×32 Input matrix $IN^{(1)}$, convolved with $(6, 1, 5, 5)$ convolution kernel, with stride $s_c^{(1)} = 1$, padding $pad^{(1)} = 0$, produces 6 output matrices $O^{(1)}$ of size 28×28 , followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(1)}$: 2×2 max pooling kernel, with stride $s_\rho^{(1)} = 2$, produces 6 output matrices $IN^{(2)}$ of size 14×14 .
2. Convolutional Round 2:
   - **Convolutional layer** $f_c^{(2)}$: 6 Input matrices $IN^{(2)}$ of size 14×14, convolved with $(6, 16, 5, 5)$ convolution kernel, with stride $s_c^{(2)} = 1$, padding $pad^{(2)} = 0$, produces 16 output matrices $O^{(2)}$ of size 10×10, followed by ReLU activation.
   - **Max Pooling layer** $\rho^{(2)}$: 2×2 max pooling kernel, with stride $s_\rho^{(2)} = 2$, produces 16 output matrix $IN^{(3)}$ of size 5×5.
3. FCNN Round 1:
   - Flatten the matrix $IN^{(3)}$ and get the 400-dimensional vector $X^{(1)}$.
   - **Fully-connected Layer** $f^{(1)}$: $d^{(1)} = 400$ input neurons and $d^{(2)} = 120$ output neurons, followed by ReLU activation.
4. FCNN Round 2:
   - **Fully-connected Layer** $f^{(2)}$: $d^{(2)} = 120$ input neurons and $d^{(3)} = 84$ output neurons, followed by ReLU activation.
5. Last Layer $f^{(3)}$: $d^{(3)} = 84$ input neurons and $d^{(4)} = 10$ output neurons.

D Noise Robustness Analysis on the Reduced LeNet-5

In this section, we take the black-box end-to-end attack on a reduced version of LeNet-5 as an example. The target CNN consists of two Convolutional blocks, one Fully-connected Block, and a final fully-connected layer, yielding exactly four layers of parameters to extract. While the overall end-to-end performance is summarized in Table 1 (Section 1.2) and detailed model structures are given in Supplementary Material C.2, this section presents a detailed layer-by-layer extraction record.

Table 4: Layer-wise Recovery Errors of the (2+1) reduced LeNet-5 Model

Layer	**Architecture*** $d^{(k)}-d_f^{(k)}-d^{(k+1)}$	**Relative Error**	**Max Rel Error**	$\max\lvert\theta-\hat{\theta}\rvert$	$(\varepsilon,0)$
Conv R1	1024–4704–1176	$2^{-31.78}$	$2^{-28.97}$	$2^{-33.45}$	$2^{-27.86}$
Conv R2	1176–1600–400	$2^{-27.55}$	$2^{-23.94}$	$2^{-30.78}$	$2^{-24.91}$
FCNN R1	400–20	$2^{-23.21}$	$2^{-19.61}$	$2^{-27.26}$	$2^{-23.13}$
FCNN R2	20–10	$2^{-26.26}$	$2^{-22.61}$	$2^{-27.26}$	$2^{-22.21}$

* : The architectures of the layers in FCNN Block (FCNN Round 1 and Round 2) are denoted in the form of $d^{(k)}-d^{(k+1)}$.

Table 4 reports the layer-wise error propagation evaluated across four metrics: Relative Error, element-wise Maximum Relative Error, element-wise Maximum Absolute Error ($\max|\theta-\hat{\theta}|$ [9]), and the $(\varepsilon,0)$-functional equivalence [9].

Relative Error is defined as the maximum L2-based relative error between the parameters and the aligned recovered parameters,

$$\text{Relative Error} = \max\left(\frac{\left\|\hat{A}^{(k)}-A^{(k)}\right\|_2}{\left\|A^{(k)}\right\|_2}, \frac{\left\|\hat{b}^{(k)}-b^{(k)}\right\|_2}{\left\|b^{(k)}\right\|_2}\right). \tag{70}$$

Max Rel Error denotes the element-wise Maximum Relative Error,

$$\text{Max Rel Error} = \max\left(\max\left|\frac{\hat{A}_{i,j}^{(k)}-A_{i,j}^{(k)}}{A_{i,j}^{(k)}}\right|, \max\left|\frac{\hat{b}_i^{(k)}-b_i^{(k)}}{b_i^{(k)}}\right|\right). \tag{71}$$

The $\max|\theta-\hat{\theta}|$ denotes the global maximum absolute element-wise deviation between the original and recovered parameters across the entire recovered k-layers network. Formally,

$$\max|\theta-\hat{\theta}| = \max_{1\le t\le k}\left(\max\left|\hat{A}_{i,j}^{(t)}-A_{i,j}^{(t)}\right|, \max\left|\hat{B}_i^{(t)}-B_i^{(t)}\right|\right). \tag{72}$$

As described in Section 2.3, $(\varepsilon,0)$-functional equivalence denotes the maximum output error bound derived via forward error propagation. After aligning the recovered and original parameters, let e_k denote the input error to layer k and δ_i denote the largest singular value of $\hat{A}^{(k)}-A^{(k)}$, the error propagates according to [9]:

$$e_{k+1} \le \delta_k \cdot e_k + \left\|\hat{B}^{(k)}-B^{(k)}\right\|_2. \tag{73}$$

All four error metrics demonstrate that our proposed attack achieves extremely high recovery precision with well-controlled error accumulation.

E Analysis on Unknown Kernel Size

This section explores a more restricted scenario where the convolutional kernel size is unknown to the attacker. We argue that our proposed attack remains effective, albeit requiring additional offline analysis and computational attempts. It is worth noting that this process – inferring the bounds of the kernel size and enumerating potential configurations – relies solely on the extracted solution vectors. Therefore, it is conducted entirely offline and does not require any further queries to the victim model.

Analyzing the Bound of the Kernel Size. First, an attacker can simply guess the kernel size based on common practices. Modern CNNs tend to use small, standardized kernel sizes to achieve better feature extraction (e.g., 3×3, 5×5). Consequently, the search space is highly restricted. An attacker can simply enumerate the possible kernel sizes and strides based on empirical values.

Furthermore, our attack inherently extracts additional structural information, significantly reducing the reliance on blind guessing. The solution vectors ($A_i^{(k)}$ of the linear system in Eq. (21) and $A_i^{(k)} - A_j^{(k)}$ of the system in (35)) expose the structural information of the convolution kernel. From these vectors, we can infer the bounds of the kernel size by identifying the length of contiguous non-zero segments.

Specifically, in practice, the row vector $A_i^{(k)}$ of a convolutional matrix for an RPCP is partially recovered because some neurons in the $(k-1)$-th layer are suppressed by ReLU. Recall from Eq. (23) that the recovered $\hat{A}_i^{(k)}$ could be

$$\hat{A}_i^{(k)} = \Big(\underbrace{\mathbf{0}}_{w_{in}^{(k)} \cdot \bar{i}}, \underbrace{\underbrace{\mathbf{0}}_{\underline{i}}, \hat{C}_{0,0}^{(k)}, \mathsf{NaN}, \cdots, \hat{C}_{0,w_c^{(k)}-1}^{(k)}, \underbrace{\mathbf{0}}_{pad}}_{w_{in}^{(k)}}, \cdots, \underbrace{\underbrace{\mathbf{0}}_{\underline{i}}, \mathsf{NaN}, \mathsf{NaN}, \cdots, \hat{C}_{h_c^{(k)}-1,w_c^{(k)}-1}^{(k)}, \underbrace{\mathbf{0}}_{pad}}_{w_{in}^{(k)}}, \underbrace{\mathbf{0}}_{pad} \Big),$$

where $\bar{i} = \left\lfloor \frac{i}{w_o^{(k)}} \right\rfloor$, $\underline{i} = i \bmod w_o^{(k)}$. Note that, the length of the contiguous non-zero values in each row vectors $A_i^{(k)}$ reflects the kernel width. However, any entry in the recovered $\hat{A}_i^{(k)}$ may contain a NaN value, including those entries whose actual value should be 0. Assuming there are n contiguous non-zero segments in $\hat{A}_i^{(k)}$, with lengths denoted by l_j for $0 \le j \le n-1$, the actual kernel width $w_c^{(k)}$ is less than or equal to the length of any such segment, *i.e.*,

$$\omega_c^{(k)} \le \min \{l_j,\ 0 \le j \le n-1\}. \tag{74}$$

The kernel height $h_c^{(k)}$ is bounded by the total number of these non-zero segments, *i.e.*,

$$h_c^{(k)} \le n. \tag{75}$$

For a PSP, the non-zero entries in the calculated row vector $\Delta_{ij}\hat{A}^{(k)} \approx g_t(A_i^{(k)} - A_j^{(k)})$, solved from the linear system in Eq. (35), represents the union

of those in $A_i^{(k)}$ and $A_j^{(k)}$ because their non-zero values are are offset from each other. Consequently, Eqs. (74) and (75) still hold, *i.e.*, the length of each contiguous non-zero segment is greater than or equal to the kernel width ($w_c^{(k)} \leq \min\{l_j,\ 0 \leq j \leq n-1\}$), and similarly, the kernel height is bounded by the number of these segments ($h_c^{(k)} \leq n$).

Enumerating the Kernel Size. Recall from the *Pattern Matching Method* in Sect. 4.2 that we determine the correct neuron index i by verifying whether the convolution kernel pattern vector $\vec{V}_i$ and the recovered $\hat{A}_i^{(k)}$ strictly follow the *matching principle* (requiring "Non-zero Consistency" and "Zero Consistency"). When the kernel size is unknown, we can enumerate candidate values for the kernel width $w_c^{(k)}$ and height $h_c^{(k)}$ within the previously inferred bounds. For each guessed kernel size, we attempt to find a matching neuron i. The guessed sizes $(w_c^{(k)}, h_c^{(k)})$ that successfully yield a valid neuron i form a candidate set of kernel sizes, denoted as $\mathcal{K}$.

Similarly, for a PSP, recall from Sect. 5.2 that we extract the competing neurons' indices i and j by checking whether the difference vector $A_i^{(k)} - A_j^{(k)}$ and the recovered $\Delta_{ij}\hat{A}^{(k)}$ satisfy the *matching principle* (requiring "Union Non-zero Consistency" and "Common Zero Consistency"). By enumerating the candidate dimensions $w_c^{(k)}$ and $h_c^{(k)}$, we evaluate potential pairs (i, j). The guessed kernel sizes that successfully yield a valid pair (i, j) form the candidate size set $\mathcal{K}$ for this specific PSP.

With multiple critical points (including both RPCPs and PSPs), we can uniquely determine the true kernel size. By taking the intersection of the candidate size sets ($\bigcap \mathcal{K}$) derived above, we rapidly filter out incorrect guesses, ultimately isolating the exact structural dimensions of the convolution kernel.

F Analysis of Other Structural Hyperparameters

In this section, we discuss some variants of CNN. Specifically, we focus on modifications to those structural hyperparameters, including padding, stride (in both convolutional and pooling layers), and dropout mechanisms. Naturally, this raises a question: *Will our attack still succeed with the modifications to the hyperparameters?*

F.1 Zero-padding in the Convolutional Layer

To fully capture the features near the boundaries, zeros are usually padded around the input matrix $IN^{(k)}$. For example, $pad^{(k)} = 1$ denotes adding one row/column of zeros to all four sides of $IN^{(k)}$. Consequently, the output matrix $O^{(k)}$ after convolution with $pad^{(k)}$ has a shape of $h_o^{(k)} \times w_o^{(k)}$, where

$$h_o^{(k)} = \left\lfloor \frac{h_{in}^{(k)} + 2 \cdot pad^{(k)} - h_c^{(k)}}{s_c^{(k)}} \right\rfloor + 1,\ w_o^{(k)} = \left\lfloor \frac{w_{in}^{(k)} + 2 \cdot pad^{(k)} - w_c^{(k)}}{s_c^{(k)}} \right\rfloor + 1 \tag{76}$$

Two common types of padding are widely used in practice:

1. **Valid padding.** No zero padding is applied ($pad^{(k)} = 0$). The convolution kernel is restricted to visit only positions where it is fully contained within the input boundaries.
2. **Same padding.** Zeros are padded around the input matrix such that the size of the output feature map is identical to that of the input.

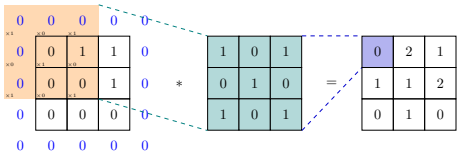


Fig. 17: Example of Convolution Operation with zero padding

Figure 17 illustrates a scenario using same padding, with $\mathsf{c}_{in}^{(k)} = \mathsf{c}_{out}^{(k)} = s_c^{(k)} = 1$, where the input matrix $IN^{(k)}$ has dimensions $h_{in}^{(k)} \times w_{in}^{(k)} = 3 \times 3$, *i.e.*, $d^{(k)} = 9$. Given a $h_c^{(k)} \times w_c^{(k)} = 3 \times 3$ kernel matrix $C^{(k)}$, with the padding $pad^{(k)} = 1$, the output matrix $O^{(k)}$ is of dimension $h_o^{(k)} \times w_o^{(k)} = 3 \times 3$, *i.e.*, $d_f^{(k)} = 9$. In the algebraic view, the padding operation can be modeled as padding zeros to the input vector $X^{(k)}$ and get the zero-padded input vector $X_{pad}^{(k)} \in \mathbb{R}^{d_{pad}^{(k)}}$, $d_{pad}^{(k)} = (h_{in}^{(k)} + 2 \cdot pad^{(k)})(w_{in}^{(k)} + 2 \cdot pad^{(k)})$, *i.e.*, $d_{pad}^{(k)} = 25$. According the Def. 2, the convolutional matrix corresponding to $X_{pad}^{(k)}$ is:

$$A_{pad}^{(k)} = \begin{pmatrix} c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} \end{pmatrix}.$$

However, for the padded input vector $X_{pad}^{(k)} \in \mathbb{R}^{d_{pad}^{(k)}}$, we can only change the $d^{(k)}$ dimensions ($d^{(k)} < d_{pad}^{(k)}$), as the padded entries must remain fixed at zero. This constraint results in a loss of full control over the entire input space $\mathbb{R}^{d_{pad}^{(k)}}$. Instead, we continue to use the raw input vector $X^{(k)} \in \mathbb{R}^{d^{(k)}}$, and the convolutional matrix $A^{(k)}$ is obtained by extracting the columns of $A_{pad}^{(k)}$ that correspond to the elements of the raw input vector $X^{(k)}$ before padding. Let $Col^{(k)} \subset \{0, 1, \cdots, d_{pad}^{(k)}\}$ denote the set of corresponding column indices, then

$$Col^{(k)} := \{(w_{in}^{(k)} + 2pad^{(k)}) \cdot i + j,\ pad^{(k)} \leq i \leq pad^{(k)} + h_{in}^{(k)} - 1,\ pad^{(k)} \leq j \leq pad^{(k)} + w_{in}^{(k)} - 1\}. \tag{77}$$

In this example, the column indices $Col^{(k)} = \{6, \ldots, 8, 11, \ldots, 13, 16, \ldots, 18\}$,

$$A^{(k)} = \begin{pmatrix} c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 & 0 & 0 & 0 \\ c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 & 0 & 0 \\ 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} & 0 & 0 & 0 \\ c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 & c_{2,1}^{(k)} & c_{2,2}^{(k)} & 0 \\ c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} & c_{2,0}^{(k)} & c_{2,1}^{(k)} & c_{2,2}^{(k)} \\ 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} & 0 & c_{2,0}^{(k)} & c_{2,1}^{(k)} \\ 0 & 0 & 0 & c_{0,1}^{(k)} & c_{0,2}^{(k)} & 0 & c_{1,1}^{(k)} & c_{1,2}^{(k)} & 0 \\ 0 & 0 & 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & c_{0,2}^{(k)} & c_{1,0}^{(k)} & c_{1,1}^{(k)} & c_{1,2}^{(k)} \\ 0 & 0 & 0 & 0 & c_{0,0}^{(k)} & c_{0,1}^{(k)} & 0 & c_{1,0}^{(k)} & c_{1,1}^{(k)} \end{pmatrix}.$$

In our attack, the RPCP and PSP methods are designed for critical points corresponding to vector $A_i^{(k)}$ and $A_i^{(k)} - A_j^{(k)}$, respectively. Both $A_i^{(k)}$ and $A_j^{(k)}$ need to contain the full information about the kernel matrix (e.g., $A_4^{(k)}$ in this case). Other row vectors with truncated subset of the kernel weights (e.g., $A_0^{(k)} \sim A_3^{(k)}$ and $A_5^{(k)} \sim A_8^{(k)}$ in this case) will interfere with the merging of partial weights in Sect. 4.2 and the computation of $\vec{C}^{(k)}$ in Sect. 5.1.

However, the pattern matching method for RPCPs in Sect. 4.2 is designed to identify the row vector with full kernel parameters, this will filter out noisy RPCPs. For PSPs, after extracting i and j from $\Delta_{ij}\hat{A}^{(k)} = A_i^{(k)} - A_j^{(k)}$ in Sect. 5.2, we can deduce $\mathcal{K}_i$ and $\mathcal{K}_j$ which denote the set of indices belonging to the LRF-C of the i-th and j-th neuron, respectively, according to Eq. 28. Denote the first element in $\mathcal{K}_i$ as $\mathcal{K}_i[0]$, and the last element in $\mathcal{K}_j$ as $\mathcal{K}_j[l_c^{(k)} - 1]$, where $l_c^{(k)} = h_c^{(k)} w_c^{(k)}$). Appropriate i and j follow the principle: $\frac{\Delta_{ij}\hat{A}^{(k)}[\mathcal{K}_i[0]]}{\Delta_{ij}\hat{A}^{(k)}[\mathcal{K}_j[l_c^{(k)}-1]]} = \frac{\vec{C}^{(k)}[0]}{\vec{C}^{(k)}[l_c^{(k)}-1]}$. This enables us to cluster the $\frac{\Delta_{ij}\hat{A}^{(k)}[\mathcal{K}_i[0]]}{\Delta_{ij}\hat{A}^{(k)}[\mathcal{K}_j[l_c^{(k)}-1]]}$ from each PSP and discard noisy PSPs (if $\Delta_{ij}\hat{A}^{(k)}[\mathcal{K}_i[0]] = \mathsf{NaN}$ or $\Delta_{ij}\hat{A}^{(k)}[\mathcal{K}_j[l_c^{(k)} - 1]] = \mathsf{NaN}$, also discard them).

A small fraction of the differences $A_i^{(k)} - A_j^{(k)}$ from PSPs may yield row vectors structurally identical to the $A_i^{(k)}$ in RPCP method which satisfy the matching principle in Sect. 4.2 and are therefore identified as RPCPs. We filter these out via Step 5 in Sect. 6.2. Therefore we can effectively discard these false-positives cases, excluding them from the set of valid information used for reconstruction.

F.2 Stride in Convolutional Layer

If the convolution stride $s_c > 1$, the convolution kernel will skip certain positions when sliding over the input matrix, which can be regarded as downsampling the output of the full convolution ($s_c = 1$). However, this does not undermine our method. While it results in a lower density of available critical points, the

algebraic integrity of the extracted row vector (specifically, $A_i^{(k)}$ derived from RPCPs in Sect. 4.1 and $A_i^{(k)} - A_j^{(k)}$ derived from PSPs in Sect. 5.2) remains complete and valid.

F.3 Stride in Pooling Layer

Different values of the pooling stride $s_\rho^{(k)}$ lead to two distinct scenarios:

1. **Disjoint Pooling** ($s_\rho^{(k)} \geq h_m^{(k)}, w_m^{(k)}$). Some elements of the output $\hat{O}^{(k)}$ after convolution and activation may not be covered by the pooling window.
2. **Overlapping Pooling** ($s_\rho^{(k)} < h_m^{(k)}, w_m^{(k)}$). Pooling windows intersect, meaning a single neuron may belongs to several local receptive fields.

Under Disjoint Pooling, when $s_\rho^{(k)} > h_m^{(k)}, w_m^{(k)}$, the pooling kernel skips certain neurons. Note that neuron i in the convolutional layer selects the i-th column of the pooling matrix $P^{(k)}$. If neuron i shows the maximum value in the t-th LRF-P, then we have $P_{t,i}^{(k)} = 1$, otherwise $P_{t,i}^{(k)} = 0$. Therefore, these skipped neurons correspond to constant all-zero column vectors. However, this does not interfere with the identification and calculation of RPCPs and PSPs, as they will not be detected.

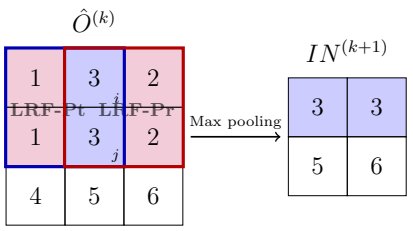


Fig. 18: Overlapping Pooling case 1

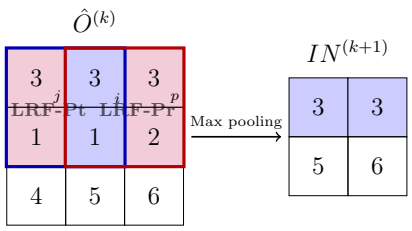


Fig. 19: Overlapping Pooling case 2

Under Overlapping Pooling, for RPCPs, a neuron in the critical state (*i.e.*, input=0) may show the maximum value across multiple overlapping pooling windows, resulting in multiple ones in its column vector. Any pooling window that selects this critical neuron, is sufficient to compute the row vector $A_i^{(k)}$. The simultaneous selection by multiple windows does not induce calculation failure. In the PSP method, suppose in the t-th LRF-P, neuron i and j achieve the same maximum value, as shown in the top-left of Fig. 18 and Fig. 19 (outlined in blue).

1. As illustrated in figure 18, if both neurons i and j also show the maximum value in another r-th LRF-P (outlined in red), the output difference still presents the gradient information $A_i^{(k)} - A_j^{(k)}$.

2. As illustrated in figure 19 , if neuron i (j) and another neuron p show the maximum value in r-th LRF-P (outlined in red), with high probability we will get a combination of $A_i^{(k)} - A_j^{(k)}$ and $A_i^{(k)} - A_p^{(k)}$ ($A_j^{(k)} - A_p^{(k)}$). However, the extraction in Sect. 5.2 returns no valid result and therefore filters out this noise.

F.4 Dropout

The dropout technique in CNN was first introduced by Krizhevsky *et al.* [29] in 2012, working by setting the output of each hidden neuron to zero with a predefined probability during the training period. The neurons which are "drop out" in this way do not contribute to the forward and backward procedure. However, this technique does not delete neurons from the network topology. The model function f is still static and deterministic. From a practical attack perspective, this mechanism makes no difference to our attack.