Model Extraction of Convolutional Neural Networks with Max-Pooling

Haolin Liu[1,2], Adrien Siproudhis[2], Christina Boura[3,4] and Thomas Peyrin[2]

[1] Shanghai Jiao Tong University, Shanghai, China
[2] Nanyang Technological University, Singapore, Singapore
[3] Université Paris Cité, CNRS, IRIF, Paris, France
[4] Institut universitaire de France (IUF), Paris, France

Abstract

Model extraction attacks aim to recover the internal parameters of neural networks through black-box queries. While significant progress has been achieved for fully connected ReLU networks, far less is known about structured architectures such as Convolutional Neural Networks (CNNs), which are widely used in practice. In particular, convolutional layers introduce locality and weight sharing, while max-pooling operations leak only relative activation information, both of which require rethinking and extending existing extraction techniques. In this work, we study the extraction of CNNs combining ReLU activations and max-pooling layers in the soft-label setting. We first demonstrate that max-pooling can be understood as a natural extension of the ReLU non-linear operation, where the attacker only has access to relative information between neurons. The local structure of convolution allows us to overcome this difficulty and reconstruct the underlying convolutional kernel. We also introduce optimizations that take advantage of the specific structure of CNNs: by using receptive-field analysis, we design efficient methods to filter noise and localize critical points. These improvements significantly reduce the computational cost compared to a naive reduction to a large sparse fully connected network. Finally, we validate our methodology experimentally on a compact VGG-style convolutional neural network trained on CIFAR-10. The results demonstrate successful layer-by-layer extraction in practice, accurate localization of critical points, and significant efficiency gains from receptive-field-based localization.

Keywords: model extraction, ReLU-based neural networks, CNNs, max-pooling

1 Introduction

Model-extraction attacks against neural networks aim to recover the internal parameters of these models by interacting with them as a black box. Although neural networks were not designed with security in mind, such attacks can have significant consequences. Training neural networks for specific tasks is often time-consuming and expensive, and these models are frequently deployed as paid services. As a result, the ability to reproduce near-identical copies of them represents a serious threat. Furthermore, if an attacker is able to extract the model of a neural network, additional attack scenarios become possible. Model extraction is rarely the end goal: once an attacker obtains an accurate copy of a target model, it

Emails: hliu033@e.ntu.edu.sg (Haolin Liu), sipr0001@e.ntu.edu.sg (Adrien Siproudhis), christina.boura@irif.fr (Christina Boura), thomas.peyrin@ntu.edu.sg (Thomas Peyrin)

This work is licensed under a "CC BY 4.0" license.
Date of this document: 2026-08-23.

can be used to carry out attacks that are much harder to perform using black-box access alone. These include generating adversarial examples, determining whether a specific data point was used during training, or running the model offline without query limits or input sanitization, thereby bypassing usage policies imposed by the API.

Over the last few years, significant progress has been made on model extraction attacks. The work of Carlini et al. [CJM20] marked an important turning point, as it was the first to recover the internal parameters of ReLU-based fully connected neural networks with up to three hidden layers. By showing that model extraction can be viewed as a cryptanalytic problem in disguise, they argued that it should be approached using tools and techniques from cryptanalysis. Moreover, this work introduced several important techniques, most notably for signature extraction (i.e., recovering the absolute values of the weights), which form the basis of most follow-up attacks. On the other hand, for the sign extraction (i.e., recovering the signs of the weights), the method proposed in [CJM20] relied, except for particular layer architectures, on a brute-force approach, which limits its applicability.

In 2024, Canales Martínez et al. [CMCSH+24] proposed several techniques for the sign-recovery problem, the most applicable of which is the so-called *neuron-wiggle* technique. Subsequently, Foerster et al. [FMSH24] combined the signature extraction approach of [CJM20] with the sign-recovery methods of [CMCSH+24]. By introducing further improvements to the sign-recovery process, they reported end-to-end extraction attacks on neural networks with up to three hidden layers. Interestingly, this work was also the first to identify signature extraction as the main bottleneck of the attack procedure.

More recently, Liu et al. [LSE+26] identified several limitations of the signature-recovery techniques introduced in [CJM20], which prevented their application to neural networks with more than three layers. To address these issues, they proposed several new techniques that enable signature extraction to scale to deeper architectures. In addition, they improved the sign-extraction phase by successfully combining the neuron-wiggle technique with the so-called SOE method from the literature, resulting in end-to-end extraction attacks on fully connected neural networks with up to eight layers.

All the works cited above, notably [CJM20, CMCSH+24, FMSH24] and [LSE+26], assume that the attacker has access to the network's pre-normalization outputs. This setting is commonly referred to as the *soft-label* or S5 setting, and it greatly facilitates the attacker's task, as it allows the computation of accurate derivatives needed to extract the internal parameters. A significantly harder, but also more realistic, setting is the so-called *hard-label* or S1 setting, in which only the most likely class label (for classification tasks) is exposed. Several recent works have successfully addressed model extraction in this setting for shallow networks [CDG+24, CCSH+25, CS25].

However, all these works, whether in the soft-label or hard-label setting, have almost exclusively targeted fully connected network (FCN) architectures, i.e., architectures in which each neuron is connected to all neurons from the previous layer. These architectures are a natural starting point for analysis, as their simple and uniform connectivity makes both the behaviour of the network and the extraction problem easier to study. However, fully connected networks are rarely used in practice for real-world tasks, as they do not exploit the structure of the input data and scale poorly in terms of the number of parameters. Instead, modern neural networks typically rely on more structured architectures that are adapted to the nature of the data. These include models such as *transformers*, and *convolutional neural networks*, which are widely used in image, video, and signal-processing tasks. Among these, Convolutional Neural Networks (CNNs) play a central role, as they are specifically designed to capture local patterns and correlations in high-dimensional inputs. By combining locality, weight sharing, and pooling operations, CNNs achieve excellent performance while keeping the number of parameters manageable, which explains their success and widespread adoption in a broad range of practical applications.

From a methodological point of view, extracting CNNs raises obstacles that differ from

those encountered in fully connected architectures. Although both models are piecewise-linear and can be analyzed through activation patterns and hyperplane arrangements, the structured nature of CNNs introduces additional challenges. In fully connected networks, each neuron depends on all outputs of the previous layer, so the weight matrix is dense. Convolutional layers, in contrast, rely on locality and weight sharing: each neuron depends only on a small part of the input, and the same kernel is reused across spatial positions. As a result, each parameter affects small local regions and is repeated across the gridded input rather than acting on the entire input at once. This structure requires adapting existing extraction techniques.

Pooling operations add further difficulty. Max-pooling, the most utilized pooling technique in practice, does not reveal exact activation values, but only which one is largest within a pooling window. Breaks in linear behaviour may therefore occur either when a neuron crosses zero (as in the ReLU-only case) or when two neurons compete inside the same pooling window. Consequently, less information is directly visible at the output, and the geometry of the critical hyperplanes becomes more complex.

At the same time, the structure of CNNs creates useful regularities. Convolution produces sparse and repeated patterns in the weight matrices, pooling operates independently across channels and windows, and receptive fields grow predictably across layers depending only on kernel sizes, strides, and paddings. These properties provide additional information not available in fully connected networks and, if properly exploited, can help the attacker during extraction.

Our contributions. We investigate black-box model extraction of CNNs that incorporate max-pooling, the most common pooling method for CNNs in practice. Max-pooling introduces non-smoothness that is qualitatively different from pointwise activations: the network becomes piecewise-linear with boundaries defined by comparisons between multiple pre-activations. We show that these boundaries can be exploited to recover internal parameters, and we introduce new techniques specifically designed to exploit CNN locality. We validate our techniques through experiments on a VGG-style compact CNN.

1. **Max-pooling non-linearities.** We generalise core definitions of the literature, such as ReLU critical points, to the setting of max-pooling neural networks. We show that critical points in the max-pooling setting reveal relative information of the form $|\eta_a - \eta_b|$, for two neurons η_a and η_b, rather than absolute comparisons with zero.

2. **Recovering convolutional rows.** We give a simple reconstruction procedure that turns a recovered difference into an actual convolutional row with no sign ambiguity. The method exploits the staircase sparsity pattern of convolution and the spatial shift between neurons in the same pooling window.

3. **Improved extraction pipeline for CNNs.** We introduce a receptive-field analysis to identify the layer and associated neurons of critical points. This allows us to filter out points that belong to deeper layers while reducing extraction computations and memory usage.

Related and concurrent work. A recent work [SLW+26] studies model extraction attacks against CNNs using average pooling. Their analysis is restricted to average-pooling architectures, and therefore does not address the additional complexity introduced by max-pooling operations (the most used pooling technique in practice), in particular the non-linearity and partial information loss they cause. More closely related to our contribution, an independent and concurrent work was recently posted as a preprint [CTG+26]. That work also studies model extraction in the presence of max-pooling layers, and several of the underlying ideas are related. However, the extraction strategies differ in several respects. We introduce a distinct localization strategy for critical points based on receptive-field

analysis, which permits systematic noise detection, increased computational efficiency and lower memory requirements. This aspect is not addressed in a comparable manner in [CTG+26]. Conversely, the authors of [CTG+26] propose an efficient bias-recovery procedure for certain architectures. This contribution is complementary to our techniques and could naturally be incorporated into a unified extraction pipeline. The attack setting also differs; while we chose to attack larger models in the easier layer-by-layer setting, they chose to run the experiments in an end-to-end fashion. Table 1 gives an exact comparison.

Conceptually, the two approaches are framed differently: while [CTG+26] analyses max-pooling through the lens of internal differentials [Pey10], we interpret it as a structural generalization of the fully-connected setting and extend the geometric extraction framework accordingly. In contrast with [CTG+26], whose study interprets CNNs as a distinct class of architectures requiring new tools to attack, we show that some very popular CNN architectures, such as the VGG family, can still be attacked as FCNs, though with reduced efficiency. This shows that CNNs form a broad spectrum of architectures, ranging from FCN-like networks to architectures that differ much more substantially from the fully connected setting. Overall, while these two complementary approaches are mathematically closely related in their treatment of max-pooling, our methodology places CNN extraction within the broader model-extraction literature, allowing past and future techniques to transfer naturally to many CNN architectures.

Table 1: Comparison of the attack settings and experimental CNN architectures considered in this work and in the concurrent work of [CTG+26]. Our experiments follow the VGG design by allowing consecutive convolutional layers before a max-pooling operation (Layer 2). Owing to the layer-by-layer extraction setting, we evaluate deeper networks with more channels. All models use identical convolution strides and max-pooling parameters.

Attack Setting	**Dataset**	**Conv. Size**	**Layer**	**Filters×(Input)×Channels**
Layer-by-Layer **(this work)**	CIFAR-10	3×3	1	$4 \times (32 \times 32) \times 3$
			2	$8 \times (16 \times 16) \times 4$
			3	$8 \times (16 \times 16) \times 8$
			4	$16 \times (8 \times 8) \times 8$
End-to-End ([CTG+26])	CIFAR-10	3×3	1	$3 \times (32 \times 32) \times 3$
			2	$4 \times (16 \times 16) \times 3$
	MNIST	5×5	1	$1 \times (32 \times 32) \times 1$
			2	$1 \times (16 \times 16) \times 1$
			3	$1 \times (8 \times 8) \times 1$
	MNIST	5×5	1	$1 \times (32 \times 32) \times 1$
			2	$6 \times (14 \times 14) \times 1$

Outline. The rest of this article is organized as follows. Section 2 introduces CNN architectures and the different representations we use for our attacks. Section 3 recalls the geometric framework for model extraction in piecewise-linear networks and reviews state-of-the-art techniques developed for fully connected architectures. Section 4 then addresses the max-pooling setting: we reinterpret max-pooling as a generalization of ReLU and show how to recover convolutional parameters. Section 5 presents improvements that significantly accelerate the extraction pipeline through localization techniques based on receptive-field analysis. Finally, Section 6 describes our experimental setup and reports the results validating our approach.

2 Convolutional neural networks

Convolutional Neural Networks (CNNs) are a class of deep learning architectures specifically designed to process data that can be naturally represented on a grid, such as images, videos, or multidimensional signals. Their defining characteristic is the use of convolutional layers, which apply learnable filters locally across the input, permitting the network to capture structured patterns while keeping the number of parameters manageable.

The term CNN covers a wide range of architectural variants. Nevertheless, most modern CNNs follow a common high-level structure. A typical convolutional layer first applies a padding operation to the input. This is followed by a set of filters that extract local patterns by sliding convolutional kernels across the input, sharing parameters across different locations. The outputs of the convolutions are then passed through a pointwise nonlinear function, most commonly the Rectified Linear Unit (ReLU), which sets negative values to zero and introduces nonlinearity. Finally, a pooling operation is typically applied to reduce the spatial dimension of the grid by combining local information.

Over the years, several CNN architectures have had a strong influence on modern deep learning. Early models such as LeNet [LBBH98] showed that convolutions and weight sharing are effective for tasks like digit recognition. Later, AlexNet [KSH12] demonstrated that deep CNNs can achieve very strong performance on large-scale image classification problems, which led to their widespread use. Architectures such as VGG [SZ15] focused on simple and deep designs, while ResNet [HZRS16] introduced residual connections that made it possible to train much deeper networks. Despite these differences, most CNN architectures rely on the same core operations: convolution, nonlinearity, and pooling. These operations are the focus of this work.

In this article, we therefore consider a generic model of CNNs that captures the above common design principles. The attack we present does not rely on specific features of a particular architecture, but rather on structural properties shared by a wide class of CNNs. As a result, our approach applies to any CNN whose architecture can be described using the standard machine-learning notation introduced below.

To simplify the presentation, we restrict our focus to square inputs and square convolutional kernels. This choice reflects common practice and simplifies notation, but it is not fundamental to our approach. The techniques we develop naturally extend to rectangular inputs and kernels. We further focus on CNNs that combine ReLU activations with max-pooling operations, a widely used design choice in practice and one adopted, for example, by the popular VGG family of computer vision models as well as by architectures such as ResNet, which incorporate max-pooling in their early layers.

2.1 CNN architecture

In this section, we describe the generic CNN architecture considered in our work. We begin with the standard grid-based description commonly used in the deep learning literature, and then introduce a matrix–vector representation that is more convenient for our analysis.

2.1.1 Standard (grid-based) description

Unlike fully connected networks, which treat the input as a single flattened vector, CNNs process data as multidimensional arrays or grids. In the standard setting, the input is represented as a three-dimensional array with two spatial dimensions corresponding to the *height* and *width* of the input, each of size d_{in}, and a third dimension corresponding to the number of *channels*, denoted by c_{in}. For example, a color image of size 32×32 pixels with three color components (red, green, and blue) can be represented as an array (or grid) of dimension $d = c_{\text{in}} \times d_{\text{in}} \times d_{\text{in}} = 3 \times 32 \times 32$.

Padding. Each input channel is first optionally padded with values along its borders so that the convolution does not reduce the spatial dimensions of the output and so that the filters can properly cover edge regions. After padding, each of the c_{in} channels is represented as a $d_{\mathcal{P}} \times d_{\mathcal{P}}$ array, where $d_{\mathcal{P}}$ denotes the spatial dimension (height and width) of the padded channel.

Convolution. The convolutional layer applies c_{out} *filters* to the input. Each filter produces one output channel, also referred to as a *feature map*. A single filter consists of c_{in} distinct kernels, one for each input channel. Each of these kernels is a square matrix of size $k_{\text{conv}} \times k_{\text{conv}}$, with its own independent learnable weights. At each spatial location, the filter extracts a $k_{\text{conv}} \times k_{\text{conv}}$ patch from every input channel. Each patch is multiplied by its corresponding channel-specific kernel, and the resulting values are summed across all input channels to produce a single output value, to which a bias b is added. The kernels of the filter are then shifted across the spatial grid with a stride s_{conv}, which determines the distance between two consecutive applications.

Formally, let $X \in \mathbb{R}^{c_{\text{in}} \times d_{\mathcal{P}} \times d_{\mathcal{P}}}$ denote the padded input tensor, where $X_{c,i,j}$ is the value at spatial location (i, j) in channel c. Let $W \in \mathbb{R}^{c_{\text{out}} \times c_{\text{in}} \times k_{\text{conv}} \times k_{\text{conv}}}$ denote the collection of convolutional kernels. The output tensor $Y \in \mathbb{R}^{c_{\text{out}} \times d_{\mathcal{C}} \times d_{\mathcal{C}}}$ is defined by

$$Y_{o,i,j} = \sum_{c=1}^{c_{\text{in}}} \sum_{p=0}^{k_{\text{conv}}-1} \sum_{q=0}^{k_{\text{conv}}-1} W_{o,c,p,q}\, X_{c,\, is_{\text{conv}}+p,\, js_{\text{conv}}+q} + b_o,$$

for all output channels $o \in \{1, \ldots, c_{\text{out}}\}$ and spatial indices (i, j) such that the accessed entries of X are well-defined.

Consequently, the convolutional layer produces c_{out} output channels of spatial dimension $d_{\mathcal{C}} \times d_{\mathcal{C}}$, where $d_{\mathcal{C}} = \lfloor \frac{d_{\mathcal{P}} - k_{\text{conv}}}{s_{\text{conv}}} \rfloor + 1$.

Activation and pooling. After convolution, a nonlinear activation (typically ReLU) is applied element-wise:

$$\text{ReLU}(x) = \max(0, x).$$

A max-pooling operation is finally applied independently to each output channel. Each pooling operator takes the maximum value over a $k_{\text{pool}} \times k_{\text{pool}}$ square region and moves by a stride s_{pool}. Therefore after the pooling layer we have c_{out} channels each of dimension $d_{\mathcal{M}} \times d_{\mathcal{M}}$ where $d_{\mathcal{M}} = \lfloor \frac{d_{\mathcal{C}} - k_{\text{pool}}}{s_{\text{pool}}} \rfloor + 1$.

This describes a single layer of the CNN where the convolutions are followed by max-pooling. Figure 1 illustrates the sequence of operations performed in this layer, together with the notation introduced above for the dimensions and other parameters. We denote the parameters of the ℓ-th layer using the superscript (ℓ). For instance, $d_{\mathcal{M}}^{(\ell)} = d_{\text{in}}^{(\ell+1)}$, and in order to make these operations concrete, Figure 2 below shows a toy single-channel example.

2.1.2 Matrix description

For the purpose of the attack, we adopt throughout the rest of the paper a mathematical viewpoint and interpret all the operations described above as matrix-vector computations. In this abstraction, padding and pooling operations are represented by binary matrices, channel dimensions are flattened, and convolutional kernels are modeled as block-circulant matrices. Under this representation, the input to the i-th layer is a vector $v \in \mathbb{R}^{d^{(i)}}$, where $d^{(i)} = c_{\text{in}}^{(i)} \times d_{\text{in}}^{(i)} \times d_{\text{in}}^{(i)}$. To simplify notation, we omit the layer superscript (i) whenever the context is clear.

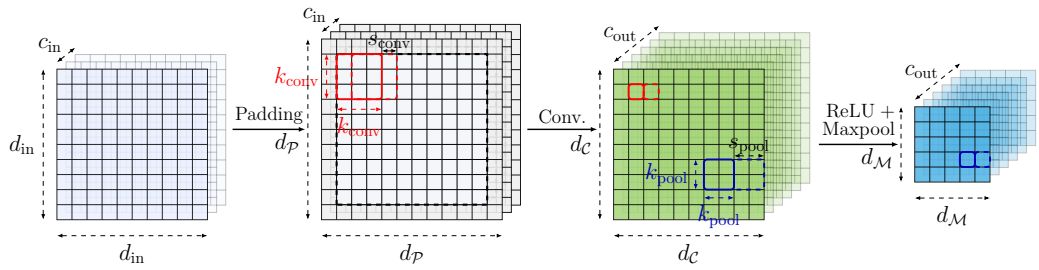


Figure 1: Summary of the CNN layer and the associated notation. An input of size $c_{\text{in}} \times d_{\text{in}} \times d_{\text{in}}$ is first padded to a grid of dimension $c_{\text{in}} \times d_{\mathcal{P}} \times d_{\mathcal{P}}$. A convolution with kernel size $k_{\text{conv}} \times k_{\text{conv}}$ and c_{out} output channels produces feature maps of dimension $d_{\mathcal{C}} \times d_{\mathcal{C}}$. In the depicted example, the convolution uses stride $s_{\text{conv}} = 1$. A ReLU activation is then applied, followed by a max-pooling operation with kernel size $k_{\text{pool}} \times k_{\text{pool}}$. In this example, the pooling stride is $s_{\text{pool}} = 2$, resulting in c_{out} output feature maps of dimension $d_{\mathcal{M}} \times d_{\mathcal{M}}$.

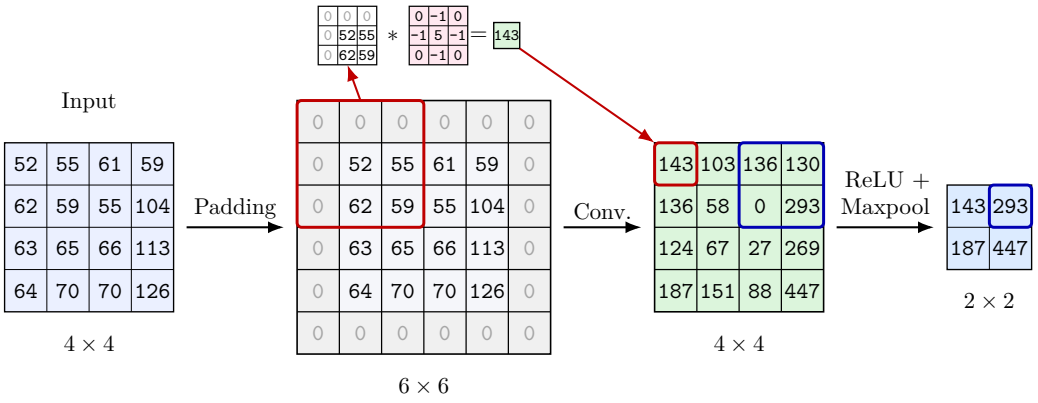


Figure 2: Concrete example of the operations performed by one CNN layer on a single-channel input. A 4×4 image is first zero-padded ($p = 1$) to a 6×6 grid. A 3×3 convolutional kernel (shown above the padded grid) is then applied with stride $s_{\text{conv}} = 1$, producing a 4×4 feature map. The red 3×3 region indicates the receptive field used to compute the top-left output value: the extracted patch is multiplied with the kernel to obtain 143, which is placed in the corresponding output entry. ReLU is applied element-wise, and a 2×2 max-pooling with stride $s_{\text{pool}} = 2$ produces a 2×2 output. The blue region shows one pooling window and the selected maximum.

Definition 1 (Padding)**.** Let $v \in \mathbb{R}^d$ be the flattened input corresponding to c_{in} channels of size $d_{\text{in}} \times d_{\text{in}}$. The padding operation is the fixed affine map $\mathcal{P} : \mathbb{R}^d \to \mathbb{R}^{c_{\text{in}} \cdot d_{\mathcal{P}}^2}$ defined by $\mathcal{P}(v) = Pv$, where P is a sparse binary matrix encoding standard padding schemes.

Common padding schemes include valid, zero, reflection, and replication padding. In practice, the two most common configurations are *valid* padding, where the convolution is applied only where the kernel fully overlaps the input and that reduces the spatial dimensions, and *same* padding, where the input is extended so that the output has the same spatial dimensions as the input. Our attack does not depend on the specific padding scheme.

Convolutional matrix. The convolution operation is the most central operation in CNNs. Its goal is to extract local features by applying learnable *filters* (each composed of channel-wise convolutional *kernels*) that respond to specific input patterns. For each pair (i, j) of output and input channels, let

$$k_{ij} = \begin{pmatrix} a_{1,1} & \cdots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{k,1} & \cdots & a_{k,k} \end{pmatrix} \in \mathbb{R}^{k \times k}$$

denote the convolutional kernel associated to output channel i and input channel j, where for simplicity we write $k = k_{\text{conv}}$. The coefficients $a_{u,v}$ depend on the pair (i, j), although we omit this dependence for readability. Flattening k_{ij} in row-major order results in the weight vector $(a_1, \ldots, a_{k^2})$, where $a_{(u-1)k+v} = a_{u,v}$ for $1 \leq u, v \leq k$.

When applied with stride $s = s_{\text{conv}}$ to the j-th input channel, of spatial dimension $d_{\mathcal{P}} \times d_{\mathcal{P}}$, convolution by k_{ij} defines a linear operator represented by a matrix $\alpha_{ij} \in \mathbb{R}^{d_{\mathcal{C}}^2 \times d_{\mathcal{P}}^2}$, which acts on the flattened vector of that channel:

$$\alpha_{ij} = \left(\begin{array}{cccccccccc} a_1 & \ldots & a_s & a_{s+1} & \ldots & a_k & 0_1 & \ldots & 0_{d_{\mathcal{P}}-k} & a_{k+1} & \ldots \\ 0_1 & \ldots & 0_s & a_1 & \ldots & a_{k-s} & a_{k-s+1} & \ldots & & & \\ & & & & & \vdots & & & & & \\ 0_1 & \ldots & 0_s & 0_{s+1} & \ldots & 0_k & 0_{k+1} & \ldots & 0_{d_{\mathcal{P}}-k} & a_1 & \ldots \\ \hline 0_1 & \ldots & 0_{s \cdot d_{\mathcal{P}}} & a_1 & \ldots & a_s & a_{s+1} & \ldots & & & \\ 0_1 & \ldots & 0_{s \cdot d_{\mathcal{P}}} & 0_1 & \ldots & 0_s & a_1 & \ldots & & & \\ & & & & & \vdots & & & & & \\ \hline 0_1 & \ldots & 0_{2 \cdot s \cdot d_{\mathcal{P}}} & a_1 & \ldots & a_s & a_{s+1} & \ldots & & & \\ 0_1 & \ldots & 0_{2 \cdot s \cdot d_{\mathcal{P}}} & 0_1 & \ldots & 0_s & a_1 & \ldots & & & \\ & & & & & \vdots & & & & & \end{array}\right)$$

Because we adopt a matrix–vector representation of the convolution, blocks of zeros are inserted between the kernel coefficients to encode the shifts of the convolutional window. For example, $0_1 \quad \ldots \quad 0_s \quad a_1$ at the beginning of the second row, indicates that the second row starts with $s = s_{\text{conv}}$ 0's before the first kernel coefficient a_1. This reflects the fact that the convolution does not act on all input pixels simultaneously: as the kernel slides horizontally across a row of the $d_{\mathcal{P}} \times d_{\mathcal{P}}$ input grid, its flattened representation is shifted to the right by s positions at each step. Once the convolution has been applied across an entire row and reaches the right boundary of the input, the kernel is moved back to the left boundary and shifted down by s rows. In the matrix representation, this vertical displacement corresponds to an initial shift of $s \cdot d_{\mathcal{P}}$ zeros. Each horizontal separator in the matrix marks such a transition between successive rows of the input grid. After the second complete horizontal pass, the initial shift becomes $2 \cdot s_{\text{conv}} \cdot d_{\mathcal{P}}$ and so on. We give in Figure 3 a simple single-channel example to bridge the matrix representation and the convolution representation.

We recall that c_{in} and c_{out} denote the number of input and output channels, respectively. A *convolutional operator* $\mathcal{C}$ defines a linear map from $\mathbb{R}^{d_{\mathcal{P}}^2 \cdot c_{\text{in}}}$ to $\mathbb{R}^{d_{\mathcal{C}}^2 \cdot c_{\text{out}}}$, specified by its collection of kernels $\{k_{ij}\}_{1 \leq i \leq c_{\text{out}},\, 1 \leq j \leq c_{\text{in}}}$ and a bias vector $b \in \mathbb{R}^{d_{\mathcal{C}}^2 \cdot c_{\text{out}}}$. For each pair (i, j), the kernel k_{ij} induces a matrix $\alpha_{ij} \in \mathbb{R}^{d_{\mathcal{C}}^2 \times d_{\mathcal{P}}^2}$, as defined above. Arranging these matrices into a block matrix $A \in \mathbb{R}^{(d_{\mathcal{C}}^2 \cdot c_{\text{out}}) \times (d_{\mathcal{P}}^2 \cdot c_{\text{in}})}$ gives the linear map of the convolutional layer

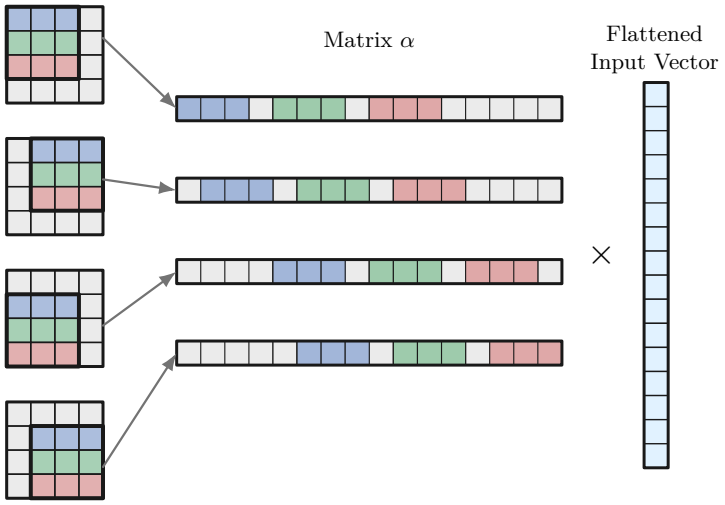


Figure 3: Illustration of the convolution-to-matrix transformation. Each application of a 3×3 convolution kernel on the 4×4 input corresponds to one row of the matrix α. Flattening the input in row-major order turns the convolution into a matrix–vector multiplication $\alpha \cdot x$.

as follows:

$$\mathcal{C}(x) := Ax + b := \begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1c_{\text{in}}} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2c_{\text{in}}} \\ \vdots & \vdots & & \vdots \\ \alpha_{c_{\text{out}}1} & \alpha_{c_{\text{out}}2} & \cdots & \alpha_{c_{\text{out}}c_{\text{in}}} \end{pmatrix} x + b.$$

Definition 2 (Filters and neurons)**.** Let $A \in \mathbb{R}^{(d_{\mathcal{C}}^2 \cdot c_{\text{out}}) \times (d_{\mathcal{P}}^2 \cdot c_{\text{in}})}$ be the matrix representation of a convolutional layer, as defined above. We introduce the following terminology to refer to different components of the weight matrix A:

- For each output channel $i \in \{1, \ldots, c_{\text{out}}\}$, we define the *$i$-th filter*

$$\alpha_i := \alpha_{i1} \,\|\, \alpha_{i2} \,\|\, \cdots \,\|\, \alpha_{ic_{\text{in}}} \in \mathbb{R}^{d_{\mathcal{C}}^2 \times (d_{\mathcal{P}}^2 \cdot c_{\text{in}})},$$

  obtained by concatenating the convolutional matrices associated with all input channels.

- For each output channel i and spatial position $k \in \{1, \ldots, d_{\mathcal{C}}^2\}$, we denote by $\eta_{i,k} : \mathbb{R}^{d_{\mathcal{P}}^2 \cdot c_{\text{in}}} \to \mathbb{R}$ the function $x \mapsto (\alpha_i)_k \cdot x + b_{(i-1)d_{\mathcal{C}}^2 + k}$, where $(\alpha_i)_k$ denotes the k-th row of α_i. We refer to $\eta_{i,k}$ as *convolutional neuron*. Equivalently, using the global row indexing of A, we write $\eta_{(i-1)d_{\mathcal{C}}^2 + k}$ for the same neuron, namely the function $x \mapsto A_{(i-1)d_{\mathcal{C}}^2 + k} \cdot x + b_{(i-1)d_{\mathcal{C}}^2 + k}$.

Definition 3 (Convolutional layer)**.** Using the notation introduced above, we define a convolutional layer $\ell_{\mathcal{C}}$ from $\mathbb{R}^d$ to $\mathbb{R}^{d_{\mathcal{C}}^2 \cdot c_{\text{out}}}$ as the composition

$$\ell_{\mathcal{C}}(x) = \mathcal{C} \circ \mathcal{P}(x).$$

ReLU, max-pooling and pooling windows. We recall that k_{pool} and s_{pool} denote the pooling window size and pooling stride, respectively. Let $x \in \mathbb{R}^{d_{\mathcal{C}}^2 \cdot c_{\text{out}}}$ be the flattened output of a convolutional layer (after ReLU).

Definition 4 (Pooling windows)**.** For each output channel, let $d_{\mathcal{M}} = \left\lfloor \frac{d_{\mathcal{C}} - k_{\text{pool}}}{s_{\text{pool}}} \right\rfloor + 1$ denote the spatial dimension after max-pooling. For each pooling-window index $m \in \{0, \ldots, d_{\mathcal{M}}^2 - 1\}$, we define

$$i_m = \left\lfloor \frac{m}{d_{\mathcal{M}}} \right\rfloor s_{\text{pool}} d_{\mathcal{C}} + (m \bmod d_{\mathcal{M}}) s_{\text{pool}} \text{ and } S_m = \left\{ i_m + a d_{\mathcal{C}} + b \;\middle|\; 0 \le a, b < k_{\text{pool}} \right\}.$$

We refer to S_m as the m-th pooling window.

Example 1. To illustrate Definition 4, consider a feature map of spatial dimension $d_{\mathcal{C}} = 4$, with pooling parameters $k_{\text{pool}} = 2$ and $s_{\text{pool}} = 2$. Then $d_{\mathcal{M}} = \left\lfloor \frac{4-2}{2} \right\rfloor + 1 = 2$, so there are $d_{\mathcal{M}}^2 = 4$ pooling windows. With row-major flattening, the indices of the 4×4 grid are shown in Figure 4.

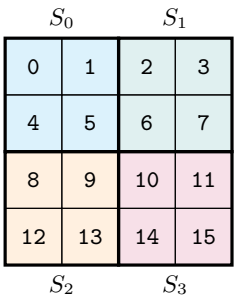


Figure 4: Pooling windows for $d_{\mathcal{C}} = 4$, $k_{\text{pool}} = 2$ and $s_{\text{pool}} = 2$. The entries are indexed in row-major order. The four pooling windows are $S_0 = \{0, 1, 4, 5\}$, $S_1 = \{2, 3, 6, 7\}$, $S_2 = \{8, 9, 12, 13\}$ and $S_3 = \{10, 11, 14, 15\}$.

For instance, for $m = 2$, we have $i_2 = \left\lfloor \frac{2}{2} \right\rfloor \cdot 2 \cdot 4 + (2 \bmod 2) \cdot 2 = 8$, and therefore $S_2 = \{8, 9, 12, 13\}$.

Definition 5 (ReLU max-pooling operator)**.** For a given input x, we define a binary selection matrix $I_x \in \mathbb{R}^{(d_{\mathcal{M}}^2 \cdot c_{\text{out}}) \times (d_{\mathcal{C}}^2 \cdot c_{\text{out}})}$, whose entries satisfy

$$(I_x)_{m,i} = 1 \quad \Longleftrightarrow \quad i \in S_m,\ x_i > 0,\ \text{and } x_i = \max_{j \in S_m} x_j,$$

with ties broken by selecting the largest index. Each row of I_x contains at most one non-zero entry.

The ReLU max-pooling operator is then defined as

$$\mathcal{M} : \mathbb{R}^{d_{\mathcal{C}}^2 \cdot c_{\text{out}}} \longrightarrow \mathbb{R}^{d_{\mathcal{M}}^2 \cdot c_{\text{out}}}, \qquad \mathcal{M}(x) = I_x x.$$

In summary, the pooling operates on groups of k_{pool}^2 elements, referred to as pooling windows and denoted by S_m. An element x_i is selected by the pooling operation if and only if it is the maximal non-zero element (with ties broken by selecting the largest index) within its pooling window, that is, $x_i = \max\{x_j \text{ s.t. } x_j > 0\}_{j \in S_m}$ for all $j \in S_m$. Following the definition, in the case where no elements are strictly positive, no element is pulled.

Definition 6 (Max-pooling convolutional network)**.**
Let $r \geq 1$ and let $[d_0, \ldots, d_{r+1}]$ be positive integers. An *r-deep max-pooling convolutional neural network* of architecture $[d_0, \ldots, d_{r+1}]$ is a function $f : \mathbb{R}^{d_0} \to \mathbb{R}^{d_{r+1}}$ of the form $f = \ell^{(r+1)} \circ \sigma^{(r)} \circ \ell^{(r)} \circ \cdots \circ \sigma^{(1)} \circ \ell^{(1)}$, where for each $i \in \{1, \ldots, r+1\}$:

- $\ell^{(i)} : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$ is a linear layer, either
  - a convolutional layer $\ell_{\mathcal{C}}^{(i)}$, or
  - a fully connected layer $\ell_{\mathrm{FC}}^{(i)}(x) = A^{(i)}x + b^{(i)}$, with $A^{(i)} \in \mathbb{R}^{d_i \times d_{i-1}}$ and $b^{(i)} \in \mathbb{R}^{d_i}$;
- $\sigma^{(i)}$ is a non-linear function, either the ReLU activation function applied component-wise, $\mathrm{ReLU}(x) = \max(0, x)$, or the max-pooling operator $\mathcal{M}$. Max-pooling layers can only follow convolutional layers.

The integer r is called the *depth* of the network, and d_i is the *width* of the i-th layer.

We summarize in Appendix A the notation used throughout the paper. Figure 1 provides a visual representation of the most relevant quantities. Finally, when the context is ambiguous, we use a superscript (i) to indicate the layer under consideration, and we denote by $\hat{x}$ the extracted version of a quantity x obtained during the attack.

3 Model extraction of neural networks

We consider two parties in this model extraction attack: an oracle $\mathcal{O}$ and an adversary. The adversary generates queries x and sends them to the oracle, which then responds with the correct output $f(x)$.

3.1 Adversarial resources and goal

Adversarial resources. We make the following assumptions regarding the target neural network and the attacker's capabilities. Except for the architecture of the target neural network that is now different, these assumptions are identical to those commonly made in prior soft-label extraction works [CMCSH+24, CJM20, FMSH24, LSE+26]:

- **Architecture**. The target neural network f is a CNN with max-pooling layers.
- **Known architecture**. The attacker knows in detail the architecture of f. In particular, this includes the parameters introduced in Section 2.1.1 (e.g., $k_{\mathrm{conv}}, s_{\mathrm{pool}}, d_{\mathrm{out}}, \ldots$), as well as the type and ordering of layers (convolutional, pooling, fully connected, etc.).
- **Unrestricted input access**. The attacker can query the network on any input $x \in \mathbb{R}^{d_0}$.
- **Raw output access**. The oracle returns the complete raw output $f(x)$, with no post-processing.
- **Precise computations**. The oracle computes $f(x)$ using 64-bit arithmetic.

Adversarial goal. The objective of the extraction is not to recover the exact internal parameters of the target network, but rather to construct a model $\hat{f}$ that is functionally equivalent to f on the considered input space [CJM20]. Informally, this means that $\hat{f}$ reproduces the input–output behaviour of f up to a negligible error. We do not require the recovered weights to coincide with the original ones. Indeed, neural networks are not uniquely parameterized: different assignments of weights may implement exactly

the same function. In particular, permutations of neurons within a layer or rescalings that are absorbed by subsequent layers lead to different parameter sets representing the same model. Therefore, recovering the parameters up to these inherent symmetries is sufficient to achieve a successful extraction. This notion differs from model distillation or high-fidelity approximation attacks, where the goal is to learn a function that approximates f empirically. Here, we aim to recover the internal structure of the network up to functional equivalence.

3.2 Definitions for the extraction of piecewise-linear networks

We recall here the geometric framework underlying model extraction for piecewise-linear networks. The following notions are now standard in the literature [CJM20, CMCSH+24, LSE+26] and are required for our analysis.

A central notion in the analysis of [CJM20] and all follow-up works is that of *critical points* of neurons. They permit the extraction of the corresponding neuron's weights through carefully chosen queries. The definition cannot be applied directly when max-pooling intervenes. We need to generalise the definition, and this requires the introduction of activation patterns first. The activation pattern for a given input x is the set of all (active) neurons that contribute to the output $f(x)$.

Definition 7 (activation pattern)**.** For an input x and a layer i, the *activation pattern up to layer i* is the set

$$P_f^{(i)}(x) = \left\{ \eta_k^{(j)} \;\middle|\; (I_x^{(j)})_{m,k} = 1 \text{ for some } m,\ 1 \le j \le i \right\}.$$

In other words, $P_f^{(i)}(x)$ consists of all neurons from the first i layers whose outputs are selected by the corresponding activation matrices. Neurons belonging to $P_f^{(i)}(x)$ are said to be *active* at x. For the last non-linear layer r, we simply write $P_f(x) := P_f^{(r)}(x)$.

Figure 5 illustrates activation patterns and critical points in a fully connected ReLU network. For an input x, the coloured neurons are the active neurons selected by the activation matrices and therefore form the activation pattern $P_f(x)$. Crossing a critical hyperplane may change this activation pattern. This motivates the following definition of a critical point.

Definition 8 (critical point)**.** An input $x^* \in \mathbb{R}^{d_0}$ is called a *critical point of neuron η_k* if crossing x^* changes the activation status of η_k. Formally,

$$\eta_k \in P_f(x^* + \epsilon\delta) \,\Delta\, P_f(x^* - \epsilon\delta)$$

for a direction $\delta \in \mathbb{R}^{d_0}$ and all sufficiently small $\epsilon > 0$, where Δ denotes the symmetric difference: $A \Delta B = (A \setminus B) \cup (B \setminus A)$. For a point x, we write $\Delta_x = P_f(x+\epsilon\delta) \,\Delta\, P_f(x-\epsilon\delta)$. Thus we say that x is *a critical point* if and only if $|\Delta_x| > 0$.

Such points lie on a hyperplane separating two linear regions of the network. Crossing this boundary changes the activation pattern and therefore the behaviour of the network locally.

Definition 9 (polytope)**.** The *polytope* of x at layer i, is the largest connected open subset of

$$\{x' \in \mathbb{R}^{d_0} \mid P_f(x) = P_f(x')\}$$

containing x. On each such region, the activation pattern is constant.

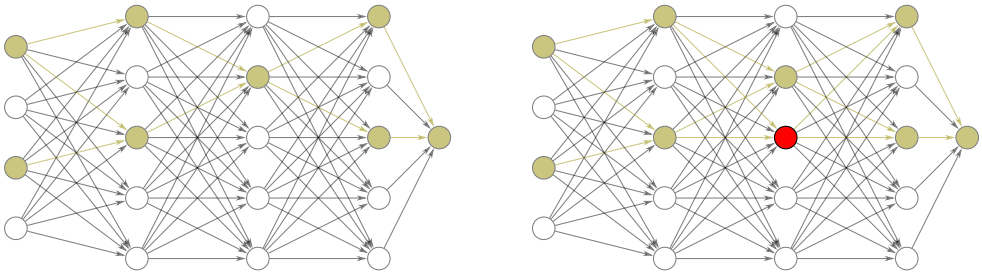


Figure 5: Activation patterns and critical points. The fully-connected network is represented as a graph, with the input layer on the left and the output layer on the right. For ReLU layers, the matrices $I_x^{(i)}$ are diagonal, and a non-zero diagonal entry $(I_x^{(i)})_{j,j}$ indicates that the corresponding neuron is active. Active neurons are shown in colour, while inactive neurons are shown in white. The coloured neurons therefore represent the activation pattern $P_f(x)$. The two pictures illustrate how the activation pattern changes when moving from a critical point x^* to the neighbouring points $x^* + \varepsilon\delta$ and $x^* - \varepsilon\delta$. In this example, a single neuron η, highlighted in red on the third layer, switches between active and inactive. This is the typical behaviour of a ReLU critical point: the pre-activation of η changes sign across the critical hyperplane, so that its output is blocked by the ReLU on one side and is positive on the other.

Local affine behaviour. Because ReLU and max-pooling are piecewise-linear, the network behaves affinely inside each polytope. More precisely, for every x and $1 \leq i \leq r+1$, there exist matrices Γ_x and vectors γ_x such that

$$f(x') = \Gamma_x x' + \gamma_x$$

for all x' in the polytope of x at layer i. In particular, if x is not critical, there exists $\epsilon > 0$ such that this affine representation holds in a ϵ-neighbourhood of x.

The extraction strategy relies on detecting changes in the local affine map Γ_x, which occur precisely when crossing critical hyperplanes. Such transitions reveal information about the parameters of the corresponding neurons. Figure 6 illustrates this phenomenon: a change in local affine behaviour induced by a critical point.

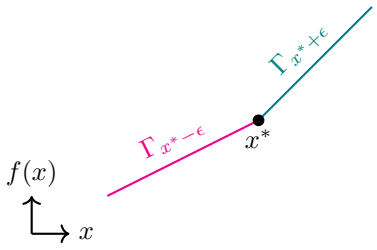


Figure 6: Polytopes and breaks in linearity. On the $-\epsilon$ side of the critical point x^*, the network is locally described by the affine map $x \mapsto \Gamma_{x^*-\epsilon}x + \gamma_{x^*-\epsilon}$, whereas on the $+\epsilon$ side it is described by $x \mapsto \Gamma_{x^*+\epsilon}x + \gamma_{x^*+\epsilon}$, a different function corresponding to a different activation pattern. Since the network is continuous, both affine representations coincide at x^*. Equivalently, x^* can be viewed as a solution of $\Gamma_{x^*-\epsilon}x + \gamma_{x^*-\epsilon} = \Gamma_{x^*+\epsilon}x + \gamma_{x^*+\epsilon}$. In higher dimensions, the set of solutions of this equation forms a critical hyperplane separating two adjacent linear regions of the network.

3.3 Extraction without max-pooling

Before addressing explicitly max-pooling in the next section, we briefly recall the extraction procedure for FCNs as developed in [CJM20, CMCSH+24, LSE+26]. Since a convolutional network without max-pooling can be represented in matrix form as a fully connected network with sparse connections, these techniques form the basis of the extraction of CNNs.

The extraction proceeds layer by layer. Assuming that the first $i-1$ layers have been recovered, the goal is to extract the parameters of layer i. The remainder of this section gives a high-level description of this procedure. An algorithmic description is provided in Appendix B for additional clarity. We refer the reader to the original works for a more detailed presentation.

Random search for critical points. Inside each polytope the network behaves affinely, hence the gradient of f is locally constant. Therefore, critical points can be identified when a change in the derivative occurs, indicating that a critical hyperplane has been crossed. In practice, such points are found by performing binary search along random directions in the input space.

Partial signature recovery. Let x^* be a critical point of a single neuron η_k in layer i. Using a second-order finite-difference operator as in [CJM20], and assuming the previous layers are known, we can query the network in a small neighbourhood of x^* and construct a linear system that isolates the contribution of that neuron. This allows us to recover its weight vector $A_k^{(i)}$ up to an unknown scaling factor $c_k^{(i)}$ (and up to a sign). Because neurons are not labelled, the extraction recovers the rows of $A^{(i)}$ up to permutation. The unknown scaling factors are naturally absorbed by the extraction of the following layers, giving a functionally equivalent reconstruction.

Intersections and partial signatures. If too few neurons in the preceding layer are active at a critical point, the resulting linear system is rank-deficient and outputs a space of solutions rather than a unique row of $A^{(i)}$. To resolve this, we intersect the solution spaces obtained from multiple critical points. Intersections of dimension one correspond to a neuron recovered up to scaling.

Merging signatures into components. Because inactive neurons in layer $i-1$ may block some coordinates, the recovered row may be only partial, with certain entries missing. Full signatures are reconstructed by merging compatible partial signatures obtained from different critical points of the same neuron. Each partial signature corresponds to a scaled version of $A_k^{(i)}$ with some missing entries. Two partial signatures are merged whenever their overlapping non-zero coordinates are proportional, which identifies them as originating from the same neuron.

Discarding deeper components. Components that are too small or show unusual behaviour (e.g., merging with points from deeper layers) are discarded. After this filtering step, the remaining components correspond to neurons of the target layer.

Recovering the bias. Once a scaled signature $c_k^{(i)} A_k^{(i)}$ has been recovered, the corresponding scaled bias $c_k^{(i)} b_k^{(i)}$ can be obtained by evaluating the neuron at any associated critical point x^*.

Sign recovery. After recovering each neuron's signature up to a scaling factor $c_k^{(i)}$, the remaining task is to determine the sign of each constant. Canales-Martínez et al. [CMCSH+24] propose two complementary sign-extraction techniques: the *system-of-equations* (SOE) approach and the *neuron wiggle*. Liu et al. [LSE+26] combine both methods to yield a more robust one. We do not further detail these methods here, as sign recovery will be revisited in the max-pooling setting, where it becomes significantly simpler.

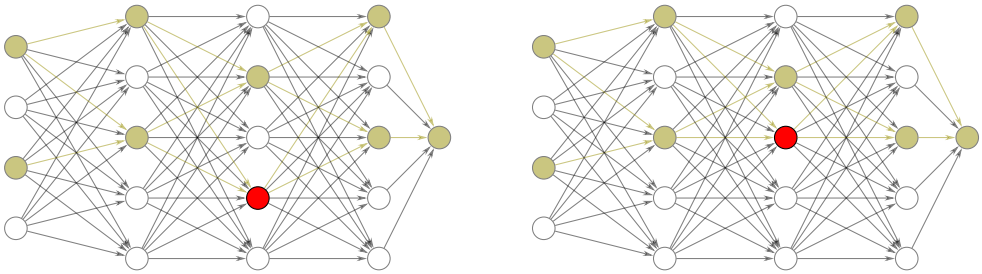


Figure 7: Illustration of $|\Delta_x| = 2$ in max-pooling networks, shown on a fully connected network for simplicity. The left and right pictures correspond to the neighbouring points $x^* - \epsilon\delta$ and $x^* + \epsilon\delta$, respectively. Crossing x^* changes which neuron is selected by the max-pooling operation: one neuron on the third layer becomes active while another becomes inactive. The symmetric difference between the two activation patterns therefore contains two neurons, yielding $|\Delta_x| = 2$. In contrast, without max-pooling, ReLU only causes a single neuron to change activation status when crossing x^*, giving $|\Delta_x| = 1$ (Figure 5).

Last layer recovery. The oracle returns the complete raw outputs of f, and the last output layer is a linear layer. Therefore, its extraction is straightforward. No additional query is needed for extraction, as we can reuse previously queried critical points to build the linear system and recover the last linear layer's weights.

4 Max-pooling in CNNs

In this section, we extend the extraction methodology from FCNs to CNNs with max-pooling. The main challenge is that critical points in the max-pooling setting reveal relative information between competing neurons, rather than isolating a single neuron. We first interpret max-pooling as a generalisation of ReLU, which allows ReLU and non-ReLU critical points to be handled in a unified framework. We then show how this relative information can be transformed into recoverable convolutional kernels, and discuss bias and sign recovery.

4.1 Max-pooling as a generalisation of ReLU

Max-pooling can be viewed as a generalization of the ReLU activation function. Indeed, while ReLU acts independently on each coordinate through $v \mapsto \max(0, v)$, max-pooling acts on an entire pooling window: $(v_1, \ldots, v_{k^2_{\text{pool}}}) \mapsto \max(0, v_1, \ldots, v_{k^2_{\text{pool}}})$. In the former case only one neuron changes activation: $|\Delta_x| = |P_f(x + \epsilon\delta)\Delta P_f(x - \epsilon\delta)| = 1$. With max-pooling, neurons can exchange dominance and $|\Delta_x|$ can be equal to 2, deeply affecting weight extraction. Figure 7 illustrates this case. For mathematical simplicity, we artificially view the max-pooling operator as the usual max function applied not to $\{v_1, \ldots, v_{k^2_{\text{pool}}}\}$ but to $\{0, v_1, \ldots, v_{k^2_{\text{pool}}}\}$ instead. This allows us to remove the special status of 0 in the original definition, and therefore the need to treat ReLU as a separate case systematically. Our analysis shows that all recovery strategies applying to cases where the max function selects between two v_i's directly apply to cases where max selects between one v_i and 0: max-pooling is a full-blown generalisation of ReLU.

The classical ReLU-based signature extraction strategy developed in [CJM20], and recalled in Section 3.3, does not directly generalize to the max-pooling setting. Indeed, in networks containing only ReLU activations, the method of [CJM20] relies on the ability to isolate individual neurons, that is, on critical points x^* satisfying $|\Delta_{x^*}| = 1$. At such a

critical point, the network reveals information of the form $|\eta|$, which permits to determine the absolute value of the corresponding weight vector.

The max-pooling setting suggests a different interpretation of this quantity. The information $|\eta|$ revealed by a ReLU critical point can be viewed as the special case $|\eta - 0|$. In this sense, ReLU is itself a max operator, namely $\mathrm{ReLU}(x) = \max(0, x)$, where the comparison is performed against the zero function. Its relative-information nature is therefore hidden by this particularly restrictive form. In the more general max-pooling setting, the attacker only observes comparisons between activations and obtains information of the form $|\eta_a - \eta_b|$, where η_b may or may not be the zero function. This leads to an important difficulty. In general, knowing all pairwise differences $\{e_a - e_b\}_{a,b}$ between elements of a set E is insufficient to reconstruct the elements themselves. Indeed, adding a constant c to every element of E leaves all pairwise differences unchanged: $(e_a + c) - (e_b + c) = e_a - e_b$. Consequently, the relative information leaked by max-pooling is not, by itself, sufficient to recover the underlying convolutional kernel. To achieve recovery, we must exploit structural properties of CNNs.

4.2 Parameter Recovery in CNNs

The central property we exploit is convolutional locality. Since each neuron depends only on a small input patch, a general critical point in the max-pooling setting reveals the difference between two nearby neurons with almost identical receptive fields. This implies that the recovered vector is the difference between two shifted row patterns corresponding to these neurons (e.g., the a-th row of the weight matrix A for neuron η_a). Because these neurons belong to the same convolutional filter, their row patterns contain the same weights. We can therefore determine the shift between them and reconstruct an actual convolutional row (up to a scalar factor) of the weight matrix, thereby removing the ambiguity coming from knowing only differences. In the special case where the critical point corresponds to a neuron and the zero neuron, the recovered vector is directly a convolutional row, and the extraction reduces to the original FCN setting.

Property 1 (Locality of convolution)**.** Let η be a convolutional neuron and consider its restriction to a fixed single input channel. Then the distance between the first and last non-zero entries (i.e., width of the support) of η, located at positions i and j respectively, satisfies

$$j - i \;=\; (k_{\mathrm{conv}} - 1)\, d_{\mathcal{P}} + (k_{\mathrm{conv}} - 1).$$

Geometrically, this property captures the locality of convolution: each neuron only depends on a small contiguous $k_{\mathrm{conv}} \times k_{\mathrm{conv}}$ patch of the input grid. When this patch is flattened row by row into a vector, each row of the kernel contributes a contiguous block of k_{conv} non-zero entries. Successive rows are shifted by $d_{\mathcal{P}}$ positions in the flattened representation, resulting in a characteristic staircase-shaped sparse support. The above formula is obtained by measuring the distance between the first and last non-zero entries of this staircase-shaped support for the first input channel. The proof is straightforward and can be found in Appendix C. See Figure 3 for illustration.

Signature recovery. Because pooling is performed independently in each channel, η_a and η_b belong to the same convolutional filter α. Thus we can write η_a and η_b as $x \mapsto \alpha_a x + \beta_a$ and $x \mapsto \alpha_b x + \beta_b$, where α_a denotes the a-th row of α (or the zero vector), rather than the corresponding row of the matrix A, β_a denotes the corresponding bias (or 0), and similarly for η_b. Suppose that we have applied the signature extraction procedure from [CJM20] at x, a critical point. We thus recover $|\alpha_a - \alpha_b|$ up to a scaling factor.

Step 1: Identifying the shift. If η_a and η_b are matrix rows of the same filter matrix, they are identical up to a horizontal shift (see Figure 3). The entries of the weight vector α_a are

therefore equal to those of α_b shifted to the right by some value s_{ab}. Let $\eta^* = |\eta_a - \eta_b|$ and denote its signature vector by w. By Property 1, the distance between the first and last non-zero entries of the weight vector of a single neuron is $(k_{\text{conv}}-1)d_{\mathcal{P}}+(k_{\text{conv}}-1)$. Since w contains the union of two shifted supports, the distance between its first and last non-zero entries satisfies $|i-j| = (k_{\text{conv}}-1)d_{\mathcal{P}}+(k_{\text{conv}}-1)+s_{ab}$, where i and j denote the first and last non-zero indices of w smaller than $d_{\mathcal{C}}^2$. Hence $s_{ab} = |i-j| - \Big((k_{\text{conv}}-1)d_{\mathcal{P}}+(k_{\text{conv}}-1)\Big)$. If η_b is the zero function, then $|\alpha_a - \alpha_b| = |\alpha_a|$ and thus $s_{ab} = 0$.

Step 2: Recovering the neuron. Once the shift s_{ab} is known, we can reconstruct α_a recursively from our extracted vector w up to a scaling factor. Denoting by $\alpha_{a,e}$ and w_e the e-th coordinates of respectively α_a and w, we have

$$w_e = \begin{cases} \alpha_{a,e} - \alpha_{a,e-s_{ab}} & \text{if } e > i + s_{ab}, \\ \alpha_{a,e} & \text{if } e \leq i + s_{ab}. \end{cases}$$

This relation can be inverted to recover α_a:

$$\alpha_{a,e} = \begin{cases} w_e + \alpha_{a,e-s_{ab}} & \text{if } e > i + s_{ab}, \\ w_e & \text{if } e \leq i + s_{ab}. \end{cases}$$

Thus, starting from the leftmost non-zero coordinate, we reconstruct the entire staircase of the filter α, and therefore recover the corresponding convolutional kernel up to a scaling factor. Note how, if $s_{ab} = 0$, the signature recovery step performs $w = \alpha_a$, replicating the original signature recovery from [CJM20] as a special case. In fully connected networks, functionally equivalent extraction permutes the rows of each recovered layer, which in turn permutes the columns of the next layer and prevents direct computation of shifts. In convolutional networks, however, the staircase structure fixes the relative order of neurons within each filter. As a result, once a filter is recovered, the ordering of its neurons is known, and the corresponding columns in the next layer are correctly aligned. The extraction may still permute the different filters, since convolution processes channels independently. This does not affect the recovery of individual kernels. We give recovery results and further experiments in Section 6.1.

Bias recovery with ReLU critical points. If x^* corresponds to a competition between two neurons η_a and η_b, then $(\alpha_a - \alpha_b)x^* + (\beta_a - \beta_b) = 0$. After the signature recovery step, we recover $|\alpha_a - \alpha_b|x^*$. If $\beta_a = \beta_b$, information about the bias is hidden and no bias can be recovered. The only case in which $\beta_a \neq \beta_b$ occurs is when either η_a or η_b is the zero function pooled by the max operator, in which case one of the biases is equal to 0. In this situation, the bias can be recovered up to a sign, exactly as in the extraction of FCNs. We therefore introduce the following definition:

Definition 10 (ReLU critical point)**.** A critical point $x^* \in \mathbb{R}^{d_0}$ is a *ReLU critical point* if there exists a neuron η_a on the n-th layer belonging to a pooling window S such that

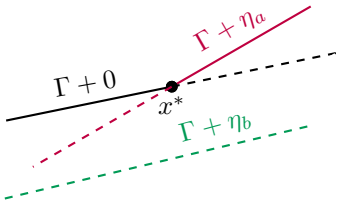
$$\eta_a(F^{(n)}(x^*)) = 0 \quad \text{and} \quad \eta_b(F^{(n)}(x^*)) < 0 \text{ for all } b \in S \setminus \{a\}.$$

Equivalently, the activation pattern changes by exactly one neuron:

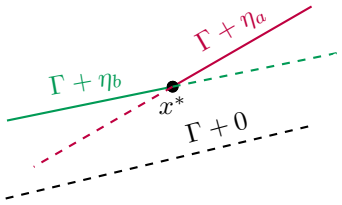
$$|\Delta_x| = |P_f(x + \epsilon\delta)\Delta P_f(x - \epsilon\delta)| = 1,$$

Critical points that are not ReLU critical points, i.e., critical points satisfying $|\Delta_x| = 2$, are called *non-ReLU critical points*.

Here, for $1 \leq i \leq r+1$, we define $F^{(i)} : \mathbb{R}^{d_0} \to \mathbb{R}^{d_i}$ by $F^{(i)} := \sigma^{(i)} \circ \ell^{(i)} \circ \sigma^{(i-1)} \circ \cdots \circ \sigma^{(1)} \circ \ell^{(1)}$ with $F^{(0)}$ the identity and $F^{(r+1)} = f$. ReLU critical points are easy to identify through the observed width of the support of η^* which corresponds to that of a single neuron. Figure 8 illustrates the specificity of ReLU critical points and their frequency among all critical points is studied later in Section 6.1.



(a) $\eta_a(x) \geq 0 > \eta_b(x)$ for $x \geq x^*$
$|\Delta_{x^*}| = 1$.
$|\beta - 0|$: bias can be recovered.



(b) $\eta_a(x) \geq \eta_b(x) > 0$ for $x \geq x^*$
$|\Delta_{x^*}| = 2$
$|\beta - \beta|$: bias cannot be recovered.

Figure 8: Bias recovery through ReLU critical points. Solid lines represent the network's outputs and dashed lines the equations given by their label. The description of the network as $\Gamma + \eta$'s is an oversimplification for illustrative purposes. **Left (ReLU critical points):** the 0 function and η_a exchange dominance. They do not have the same bias, which makes it possible for the attacker to learn β. **Right (non-ReLU critical points):** two neurons η_a and η_b with the same bias exchange dominance inside a pooling window, preventing the attacker from learning the bias.

Sign Recovery. Sign recovery is very easy in layers following max-pooling, even without access to the bias. Indeed, we have the requirement that, for competing neurons η_a and η_b, possibly 0, their outputs are greater than those of all other neurons in the same pooling window. Since all neurons pooled together share the same bias, it suffices to consider the inner products of the recovered signature vectors with the input. Thus, we must have $\eta_a \cdot x^* = \eta_b \cdot x^* > \eta_o \cdot x^*$ for all η_o in the same pooling window. Getting the sign wrong immediately leads to $\eta_a \cdot x^* = \eta_b \cdot x^* < \eta_o \cdot x^*$ and there is therefore no ambiguity. This sign-recovery procedure is identical to that of [CTG+26]. Figure 9 illustrates sign recovery. In the case where a convolutional layer follows another convolutional layer, the attacker reverts to the techniques of [CMCSH+24] and [LSE+26].

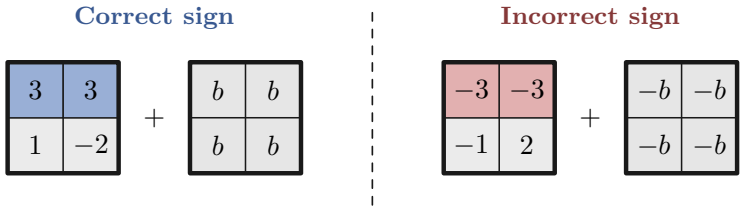


Figure 9: Sign recovery from critical points from computed outputs. Neurons we use for recovery, the ones highlighted, are those selected by max-pooling. Other entries are computed by shifting the extracted row. **Left:** neurons pulled have a higher output than those in the same pooling window. **Right:** flipping the sign makes these two neurons the smallest values in the window, no matter the actual bias value, contradicting max-pooling selection. This shows that sign extraction can be done even without access to the bias.

We provide an explicit algorithmic presentation of these methods in Algorithm 10 in Appendix D. The algorithm summarizes how the recovered kernels, biases, and signs are updated during the extraction of each convolutional layer.

Example 2. We illustrate the recovery procedure using a critical point obtained from the fourth filter of the first convolutional layer of a CNN with a 3×3 convolutional kernel and stride 1. The full architecture and the corresponding extraction results are given in

Section 6. For clarity, we display only the coefficients corresponding to the first input channel; the same procedure applies independently to the other channels. This critical point is induced by two neighbouring convolutional neurons located at positions $(11, 5)$ and $(11, 6)$ in the output feature map. These neurons belong to the pooling window centred at position $(6, 3)$ (using row and column coordinates in the pooled feature map). The corresponding convolutional kernel (true weights) is

$$\begin{pmatrix} 0.83208459 & 0.96697978 & 0.55399803 \\ 0.34862656 & 0.53197297 & 0.37392191 \\ 0.07707245 & -0.01195289 & -0.25148758 \end{pmatrix}$$

Solving the linear system described above gives the recovered signature (only non-zero coefficients are shown). The recovered support occupies rows 10 to 12 and columns 4 to 7. Taking the padding into account, this support corresponds exactly to the union of the receptive fields of two neighbouring neurons in the same row:

$$\begin{pmatrix} 0.05302319 & 0.00859335 & -0.02631167 & -0.03530202 \\ 0.02221535 & 0.01168424 & -0.01006625 & -0.02381989 \\ 0.00490425 & -0.00566750 & -0.01525858 & 0.01602470 \end{pmatrix}$$

Since the two neurons are horizontally adjacent, their receptive fields overlap on $k_{\text{conv}} - 1$ columns. Consequently, the recovered difference is the difference between two shifted copies of the same kernel. For a kernel row $(a_{r,1}, a_{r,2}, a_{r,3})$, the corresponding row of the recovered signature is $(a_{r,1},\ a_{r,2} - a_{r,1},\ a_{r,3} - a_{r,2},\ -a_{r,3})$. The extraction algorithm exploits this overlap structure to reconstruct the original kernel up to a global scaling factor. Applying it to the recovered signature above gives the following kernel:

$$\begin{pmatrix} 0.0530232 & 0.06161655 & 0.03530487 \\ 0.02221536 & 0.0338996 & 0.02383335 \\ 0.00490425 & -0.00076326 & -0.01602184 \end{pmatrix}$$

To verify the recovery, we compute the entrywise ratio between the recovered kernel and the true kernel:

$$\begin{pmatrix} 0.06372332 & 0.06372062 & 0.06372744 \\ 0.06372251 & 0.06372429 & 0.06373885 \\ 0.06363169 & 0.06385568 & 0.06370829 \end{pmatrix}$$

Since the ratios are nearly constant, the recovery succeeds and the recovered kernel weights are scalar multiples of the true ones, as expected.

5 Extraction pipeline improvements in CNNs

As discussed above, CNNs without max-pooling are functionally equivalent to large fully connected networks. The authors of [LSE+26] show that, for wide FCNs, the runtime and memory requirements of the extraction procedure grow rapidly with the network width and quickly become prohibitive. These computational limitations must therefore be addressed with the specificities of CNNs.

We exploit structural properties of CNNs to improve the extraction. We present a method, localization analysis, for identifying both the layer and the neuron (up to the filter) to which some critical points belong, even for points deeper than the target layer. Algorithm 1 describes how localization is incorporated within the attack. The advantages of our method are as follows:

1. **Guaranteed noise removal**: the attacker can determine whether a critical point may belong to a deeper layer. Unlike previous approaches, this noise can no longer go undetected and can therefore be removed entirely, allowing extraction to proceed without any noise.

2. **Memory improvement**: by identifying the neuron to which a critical point belongs, we can take advantage of the sparse structure of CNNs to reduce the amount of memory required to process and store information associated with that critical point.

3. **Speed improvement**: first, only non-localized points need to be reconsidered at subsequent layers. Second, once the neuron associated with a critical point has been identified, weight recovery can be performed using only its active weights, resulting in a much smaller linear system and thus a significant speed-up.

Algorithm 1 Critical point management with localization

```
Collect N critical points
for each layer ℓ^(i) do
    Let p^(i) the probability that a neuron in the previous layer is active
    for all localized points on ℓ^(i) do
        ● Extract partial weights by solving a system of size at most 2k_conv^(i)2 c_in^(i)
        Complete the recovery (Section 3.3)
    end for
    for all ● non-localized points do
        if no derivatives are stored then
            Query, on average, p^(i) d_in^(i)2 c_in^(i) local derivatives
        end if
        Extract partial weights by solving, on average, a system of size p^(i) d_in^(i)2 c_in^(i)
        Perform localization analysis (Section 5.2)
        if ● the point belongs to ℓ^(i) then
            Complete the recovery (Section 3.3)
        else if the point is localized on ℓ^(j) with j > i then
            ● Clear memory to keep at most 2k_conv^(j)2 c_in^(j) derivatives
        end if
    end for
end for
```

● Speedup due to smaller weight system. ● Speedup due to fewer points processed. ● Noise removal. The attacker can choose to process more points, relying on non-guaranteed existing methods to remove noise later on. ● Memory improvement due to the sparse structure of CNNs. By local derivative we mean the result of $f(x^* + \epsilon) + f(x^* - \epsilon) - 2f(x^*)$ for some small vector ϵ.

To motivate the description of this method later on, we first explain here in detail why being able to associate a critical point with a potentially deeper neuron gives the claimed improvements. CNNs have a very sparse structure: only a few neurons are connected to the same neuron on the next layer. This means that if we know which neuron a critical point belongs to, we can know the small fraction of all neurons on its preceding layer that can contribute to its output. Equivalently, we know which entries in the weight vector extracted from our critical point should be exactly zero. All improvements rely on knowing, possibly ahead of time, the possible sparse structure of the recovered vector.

Noise removal. The FCN techniques for noise removal from [LSE+26] do not provide if-and-only-if conditions. Similarly, other works tackling CNNs, namely [CTG+26] and [SLW+26], only provide a one-way condition: if the shape of the recovered vector is not correct (i.e., does not match any of the possible sparse configurations), then the vector must be on a deeper layer. However, we show the converse is false. Some recovered vectors with a correct shape may still be on a deeper layer. We give a detailed example in Appendix E. Our localization strategy can identify which sparse pattern associated with the target layer can also be reproduced by critical points in deeper layers. The attacker

is then offered a choice. She can either eliminate the risk of noise entirely by discarding points that match these unsafe patterns, even when they belong to the target layer, or she can deliberately keep them, label them as unsafe, and remove part of the resulting noise later using this label. The former strategy requires searching for more points and thus comes with an increased query complexity, while the latter comes with having to handle noise and the computational cost of processing additional points.

Memory improvement. Suppose we localize a critical point x^* on the j-th layer. Out of all the $d_{\text{in}}^{(j)2} c_{\text{in}}^{(j)}$ neurons on the previous layer, we know which neurons cannot contribute to η, and therefore the position of at least $d_{\text{j}}^{(j)2} c_{\text{j}}^{(j)} - 2k_{\text{conv}}^{(j)2} c_{\text{in}}^{(j)}$ corresponding zero entries in the recovered vector. The factor 2 comes from the recovered vector which can have a number of non-zero entries at most the sum of that of the two competing neurons, and each of these competing neurons has at most $k_{\text{conv}}^{(j)2} c_{\text{in}}^{(j)}$ non-zero entries (the size of the kernel multiplied by the number of channels). We can therefore shrink our weight recovery system to a much smaller system of dimension at most $2k_{\text{conv}}^{(j)2} c_{\text{in}}^{(j)}$. This in turn means that we only need to keep in memory across layers $2k_{\text{conv}}^{(j)2} c_{\text{in}}^{(j)}$ values to solve the system when we reach the neuron's layer. If the extraction is done from layer i without localization, we would instead store $p \cdot d_{\text{in}}^{(i)2} c_{\text{in}}^{(i)}$ values where p is the proportion of active neurons on layer i for $f(x^*)$. Let's see how large this gap might be in practice. For VGG-16, a widely used CNN, all convolution kernels are of size 3×3 while the two first layers are of size 224×224 replicated across 64 channels. This means that we would only use $(2 \cdot 3^2)/(p \cdot 224 \cdot 224)$ of the usual memory, so $0.00035/p$ of the memory. On the smaller networks we attack, those relative results are naturally not as good as the size of the input is not 224×224 but at most 32×32 while we keep 3×3 kernels. Nonetheless, on a 32×32, our experiments align with theory and we save on average 93.80% of the memory despite the size of the input. More details are given in Section 6.

Computational improvement. Solving a smaller linear system for weight recovery also reduces the computational cost of extraction. As expected, the resulting speedups closely mirror the memory savings. In our experiments, a memory reduction of 93.80% translates into a runtime reduction of 93.36%; see Section 6. Localization provides a second computational benefit. Once a critical point has been identified as belonging to a deeper layer, it does no longer needs to be considered when attacking intermediate layers. For example, in our running setting, when attacking convolutional layer 2, we already know that 88 out of 704 collected critical points belong to convolutional layers 3 or 4. These points can therefore be skipped when attacking layer 2, reducing the number of critical points that must be processed.

Overall, localized critical points are substantially cheaper to store and reprocess than their non-localized counterparts. Moreover, localization allows the attacker to reliably discard critical points originating from deeper layers, thereby eliminating a significant source of noise. We now turn to the localization procedure itself and examine what information can be inferred from the patterns of non-zero entries in the extracted vectors.

5.1 Receptive fields: RFs, CRFs and ERFs

A main goal of our localization technique is to determine whether a critical point belongs to the target layer or if it is associated with a deeper layer. The most direct way to do so would be to compute all possible patterns that could be extracted from deeper layers and to keep only the points whose signatures do not match any of them. However, this computation is exponential in the number of neurons: it would require considering all possible activation patterns, since before extracting deeper layers we do not know the linear dependencies they may induce. To avoid this exhaustive procedure, we introduce

the notion of the core receptive field, which is easy to precompute while still allowing us to reliably identify many critical points.

Our strategy exploits the locality inherent to convolutional networks. Neurons in deeper layers depend on increasingly large regions of the input, as their activations are influenced by progressively more neurons from preceding layers. To formalize this notion of "being influenced", we introduce the concepts of *receptive field (RF)*, *effective receptive field (ERF)*, and *core receptive field (CRF)*. The sizes of the receptive field and the core receptive field will serve respectively as upper and lower bounds on the size of the effective receptive field.

Before proceeding, we clarify our layer counting convention. In this section, we count max-pooling layers as separate layers. For instance, the network $\ell^{(2)} \circ \mathcal{M} \circ \ell^{(1)}$ is considered to have three layers, where the second layer is the max-pooling operator $\mathcal{M}$. By adding conditions on the parameters of the neural network, we will be able to identify the exact neuron associated to a critical point before any intersections or merges are done. These conditions are in line with the locality CNNs aim at emulating: they are thus naturally met by many CNN structures such as the VGG neural networks or LeNet-5. AlexNet satisfies these conditions for all but the second layer. Experimental validation is provided in Section 6.2. We also denote by ℓ the last extracted layer and by L the layer containing the neurons $\{\eta_a, \eta_b\}$ associated with a critical point x. In the following, for simplicity, when discussing the size or location of a receptive field (RF) or core receptive field (CRF), we consider only its component on a single output channel of layer ℓ, since the corresponding components are identical across all output channels. We use a one-dimensional representation of input signals and convolutional neurons. This representation should not be confused with the flattened representation used elsewhere in the paper, it is mainly a convenient way to describe the propagation of receptive fields. For higher-dimensional signals (e.g., 2D images), the same derivations apply independently along each dimension. The full multi-dimensional receptive field is then obtained by combining the results from each dimension.

Definition 11 (Receptive Field (RF))**.** Let η be a convolutional neuron in layer L, and let $\ell < L$ be an earlier layer (with $\ell = 0$ denoting the input layer). The *receptive field* of η with respect to layer ℓ is the set of neurons in layer ℓ whose outputs **can** influence the output of η if they are active. We denote this set by $\mathrm{RF}_\ell(\eta)$. By $\mathrm{RF}_\ell(\eta_a, \eta_b)$ we mean $\mathrm{RF}_\ell(\eta_a) \cup \mathrm{RF}_\ell(\eta_b)$. Naturally, $\mathrm{RF}(0) = \emptyset$.

RF is generally a slightly larger set than

$$\left\{ \eta_k^{(\ell)} \;\middle|\; \exists x \in \mathbb{R}^{d_0} \text{ such that } \exists j \text{ such that } (I_x^{(\ell)})_{j,k} = 1 \right\}.$$

By "can influence", we mean structural influence: we ignore the activation constraints imposed by ReLU and max-pooling, and we allow that, within any pooling window, each neuron may potentially be selected. Consequently, $RF_\ell(\eta)$ depends only on the architectural parameters between layers ℓ and L (kernel sizes, strides, and paddings), and not on the specific values of the weights nor on a particular input x. This convention ensures that receptive fields are computable *a priori*, purely from the network architecture. Thus, $RF_\ell(\eta)$ represents the maximal subset of neurons in layer ℓ whose outputs may, through the successive linear and pooling operations, contribute to the value of the neuron η in layer L.

Computation. For each layer $i \in \{0, \ldots, L\}$ (convolutional or max-pooling), the spatial configuration is determined by three architectural parameters: the kernel size $k^{(i)}$, the stride $s^{(i)}$, and the padding $p^{(i)}$ applied to each side of the input. From this configuration, we follow the standard derivation of receptive field formulas and refer the reader to [ANS19]

for a detailed analysis. We use the notation $|\cdot|$ not to denote the number of neurons in a set, but the length of the interval it spans, defined as the difference between the largest and smallest indices plus one. For example, if $\mathrm{RF} = \{\eta_2, \eta_3, \eta_7\}$, then $|\mathrm{RF}| = (7-2)+1 = 6$, and not 3.

- **1-dimensional RF size:** For any convolutional neuron η on layer L, the size of the receptive field with respect to a layer $\ell < L$ is

$$|\mathrm{RF}_\ell(\eta)| = \sum_{i=\ell+1}^{L} \left((k^{(i)} - 1) \prod_{j=\ell+1}^{i-1} s^{(j)} \right) + 1.$$

  We generalise this result to pairs of neurons,

$$|\mathrm{RF}_\ell(\eta_a, \eta_b)| = |\mathrm{RF}_\ell(\eta_a)| + S_{\ell,L}|i-j| = |\mathrm{RF}_\ell(\eta_b)| + S_{\ell,L}|i-j|.$$

- **1-dimensional RF location:** Let $u_\ell(\eta)$ and $v_\ell(\eta)$ denote the leftmost and rightmost entry of $RF_\ell(\eta)$ in layer ℓ, respectively. Let $u_L(\eta) = v_L(\eta)$ denote the index of the convolutional neuron η on layer L. All indices are zero-indexed. Then,

$$u_\ell(\eta) = u_L(\eta) \prod_{i=\ell+1}^{L} s^{(i)} - \sum_{i=\ell+1}^{L} p^{(i)} \prod_{j=\ell+1}^{i-1} s^{(j)} = u_L(\eta) S_{\ell,L} - P_{\ell,L}$$
$$v_\ell(\eta) = u_\ell(\eta) + |\mathrm{RF}_\ell(\eta)|$$

  where $S_{\ell,L}$ and $P_{\ell,L}$ denote the effective stride and padding between layers ℓ and L, defined as $S_{\ell,L} = \prod_{i=\ell+1}^{L} s^{(i)}$ and $P_{\ell,L} = \sum_{i=\ell+1}^{L} p^{(i)} \prod_{j=\ell+1}^{i-1} s^{(j)}$. The definition is the same for $u_\ell(\eta_a, \eta_b)$ as $\mathrm{RF}_\ell(\eta_a, \eta_b)$ has been defined above.

Thus, all convolutional neurons in layer L have receptive fields of identical size, while their spatial locations differ according to their index $u_L(\eta)$. We now turn our attention to critical points.

Definition 12 (Effective Receptive Field (ERF))**.** Let $\hat{v}$ be the recovered signature associated with a critical point x^* when extracting layer ℓ. The *effective receptive field* of x with respect to ℓ is defined as

$$\mathrm{ERF}_\ell(x) = \left\{ \eta_k^{(\ell)} \;\middle|\; \hat{v}_k \neq 0 \right\}.$$

In other words, $\mathrm{ERF}_\ell(x)$ consists of the neurons in layer ℓ that contribute to the local affine behaviour of the network around x^*.

RF_ℓ represents (a bit more) than the union of all possible effective receptive fields $\mathrm{ERF}_\ell(x)$ over all critical points x. In this sense, it is the largest set that $\mathrm{ERF}_\ell(x)$ can theoretically attain. A natural question then arises: can we define, in a similar structural manner, the intersection of all possible $\mathrm{ERF}_\ell(x)$? In other words, does there exist a pre-computable minimal subset of layer ℓ whose neurons are guaranteed to influence the output of η, independently of the specific weights or inputs? Without any assumption on the weights, such an intersection need not exist. Indeed, it is possible that, for a given input, all neurons in a pooling window are inactive, in which case the effective receptive field may be empty. However looking across all channels, it is extremely unlikely that $c_{\text{in}} k_{\text{pool}}^2$ neurons are all inactive for an input. It is therefore a reasonable assumption that each max-pooling window selects at least one non-zero element (across all channels).

Definition 13 (Core Receptive Field (CRF))**.**

Let η be a convolutional neuron in layer L. The *core receptive field* of η with respect to layer ℓ is the set of neurons in layer ℓ whose outputs **must** influence the output of η if max-pooling selects at least one neuron per window. We denote this set by $\mathrm{CRF}_\ell(\eta)$. Similarly we define, $\mathrm{CRF}_\ell(\eta_a, \eta_b) = \mathrm{CRF}_\ell(\eta_a) \cup \mathrm{CRF}_\ell(\eta_b)$

Bounding the size of ERFs. We now derive upper and lower bounds on the size of effective receptive fields using the corresponding receptive fields (RFs) and core receptive fields (CRFs). The maximum size of the effective field is achieved when η_a and η_b occupy extremal positions within the same pooling window. In this case, their receptive fields are separated by $(k^{(L+1)}-1)S_{\ell,L}$, leading to the upper bound $|\mathrm{RF}_\ell(\eta_a)| + (k^{(L+1)}-1)S_{\ell,L}$. The lower bound is again given by the corresponding core receptive field:

$$|\mathrm{CRF}_\ell(\eta_a)| \leq |\mathrm{ERF}_\ell(x^*)| \leq |\mathrm{RF}_\ell(\eta_a)| + (k^{(L+1)}-1)S_{\ell,L}.$$

An important special case is for ReLU critical points where $S_{\ell,L} = 0$, leading to $|\mathrm{CRF}_\ell(\eta_a)| \leq |\mathrm{ERF}_\ell(x^*)| \leq |\mathrm{RF}_\ell(\eta_a)|$. This bound applies for all points if layer $L+1$ is a convolutional or fully connected layer as every critical point is a ReLU critical point in those layers. Computing CRFs in general is difficult, and we return to this problem later under additional assumptions on the target network. Nevertheless, by definition, the effective receptive field of a critical point must contain the corresponding core receptive field and be contained in the corresponding receptive field. Thus, for a critical point x^* associated with neurons η_a and η_b, we have:

$$\mathrm{CRF}_\ell(\eta_a, \eta_b) \subseteq \mathrm{ERF}_\ell(x^*) \subseteq \mathrm{RF}_\ell(\eta_a, \eta_b).$$

A simple example in Fig. 10 provides an intuitive illustration of the relationship among RF, CRF, and ERF.

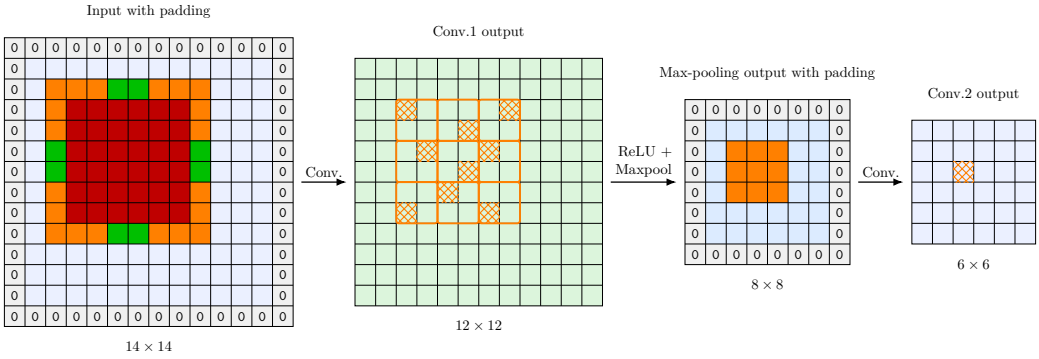


Figure 10: Illustration of the relationship between the receptive field (RF), core receptive field (CRF), and effective receptive field (ERF). Convolutional layers use padding = 1, stride = 1, and kernel size = 3, while max-pooling layers use padding = 0, stride = 2, and kernel size = 2. Consider a ReLU critical point x^* associated with a Conv.2 neuron (orange crosses in the Conv.2 output). Relative to the max-pooling output, the RF and CRF of this neuron, as well as the ERF of the critical point, coincide and correspond to the 3×3 convolutional window (orange region in the max-pooling output grid), since Conv.2 immediately follows the max-pooling layer. Relative to the input, however, these regions differ because max-pooling selects only the largest value in each pooling window. The Conv.1 neurons selected by max-pooling are marked by orange crosses in the Conv.1 grid. The ERF of the critical point is the union of the convolutional windows associated with these selected neurons (the red region together with the four orange corner regions in the input grid). The CRF of the Conv.2 neuron is the red region, consisting of all input coordinates that necessarily influence the neuron regardless of the max-pooling outcome. The RF consists of the CRF together with all coordinates that may influence the neuron (the orange and green regions). For the particular input x^*, the orange regions influence the neuron, whereas the green regions do not.

5.2 Identifying the layer of a critical point

We begin with an idealized setting. If convolutions were performed on a torus rather than on a padded grid, then by symmetry, the quantity $|\mathrm{CRF}_\ell(\eta^{(L)})|$ would be identical for all neurons $\eta^{(L)}$ within the same layer. Under this assumption, we simplify the notation by writing $|\mathrm{CRF}_{\ell,L}|$ instead of $|\mathrm{CRF}_\ell(\eta^{(L)})|$. We use the same convention for receptive fields. The edge effects introduced by padding will be addressed separately later in this section.

Suppose that, for every layer L, every critical point $x^* \in \mathbb{R}^{d_0}$ belonging to layer L, and the next convolutional layer $n > L$, the following inequalities hold:

$$|\mathrm{CRF}_{\ell,L}| \leq |\mathrm{ERF}_\ell(x^*)| < |\mathrm{CRF}_{\ell,n}|.$$

In this situation, identifying the layer to which x^* belongs would be straightforward. After extracting the signature of x^* from layer ℓ, we would compute the size of its ERF and compare it with the precomputed CRF sizes. Since the sequence $|\mathrm{CRF}_{\ell,1}|$, $|\mathrm{CRF}_{\ell,2}|$, ... is strictly increasing, the predecessor of $|\mathrm{ERF}_\ell(x^*)|$ in this sequence uniquely identifies the layer of x^*. Our goal is therefore to establish conditions under which this situation occurs.

The ideal condition above does not hold for every CNN architecture. We therefore introduce the following structural condition on the target network.

Condition 1.

Let ℓ be a convolutional layer. The following conditions hold for every subsequent convolutional layer $i \geq \ell$.

1. *If layer i is followed by a max-pooling layer $i+1$ and a convolutional layer $n = i+2$, then*
$$|\mathrm{RF}_{\ell,i}| + (k^{(i+1)} - 1)S_{\ell,i} < |\mathrm{CRF}_{\ell,n}|$$

2. *f layer i is followed directly by a convolutional layer $n = i+1$, then*
$$|\mathrm{RF}_{\ell,i}| < |\mathrm{CRF}_{\ell,n}|$$

If $\ell^{(i+1)}$ or $\ell^{(i+2)}$ is a fully-connected layer then, trivially their CRF is their whole input region and thus naturally meet the bounds we are looking for. In a practical layer-by-layer attack, where all layers preceding the target layer $\ell+1$ have already been extracted, this condition implies that, with respect to the extracted layer ℓ, the upper bound of the ERF at any convolutional layer from the target layer $\ell+1$ onward is strictly smaller than the lower bound of the ERF at the following convolutional layer.

Proposition 1 (Layer identification)**.** *Let the network satisfy Condition 1, and assume that convolutions are performed on a torus. Then, for every convolutional layer L, every critical point x^* belonging to layer L, and the next convolutional layer n after L,*

$$|\mathrm{CRF}_{\ell,L}| \leq |\mathrm{ERF}_\ell(x^*)| < |\mathrm{CRF}_{\ell,n}|$$

Proof. Let x^* be a critical point associated with neurons η_1 and η_2 in layer L, where η_1 and η_2 may coincide. By definition of the core, effective, and receptive fields, $\mathrm{CRF}_\ell(\eta_1, \eta_2) \subseteq \mathrm{ERF}_\ell(x^*) \subseteq RF_\ell(\eta_1, \eta_2)$. Taking cardinalities gives $|\mathrm{CRF}_{\ell,L}| \leq |\mathrm{ERF}_\ell(x^*)| < |\mathrm{RF}_{\ell,L}|$. By Condition 1, the receptive field of layer L is strictly smaller than the core receptive field of the next convolutional layer n, that is, $|\mathrm{RF}_{\ell,L}| < |\mathrm{CRF}_{\ell,n}|$. Combining the two inequalities gives $|\mathrm{CRF}_{\ell,L}| \leq |\mathrm{ERF}_\ell(x^*)| < |\mathrm{CRF}_{\ell,n}|$, which proves the claim. □

This proposition shows that, under Condition 1, the layer of any critical point can be determined entirely from the size of its effective receptive field. In practice, however, we do not have a direct and convenient method to compute $|\mathrm{CRF}_{\ell,L}|$ in order to verify whether a given network satisfies Condition 1. We therefore introduce another structural condition.

Condition 2. *For every convolutional layer L that is followed by a max-pooling layer at $L+1$, the architectural parameters satisfy $k^{(L)} \geq s^{(L)}(k^{(L+1)} - 1)$.*

This condition ensures that for the $k^{(L+1)}$ convolutional neurons on convolutional layer L within the same max-pooling window corresponding to a neuron η on the max-pooling layer $L+1$, their receptive fields with respect to layer $L-1$ overlap. This overlapping region consistently influences the output of η, no matter which convolutional neuron on layer L in the max-pooling window is selected. An illustration of this condition is shown in Fig. 11.

Proposition 2. *Let the network satisfy Condition 2, and let L be the target layer. Then, for every $\ell < L$,*

$$|\mathrm{CRF}_{\ell,L}| = |\mathrm{RF}_{\ell,L}| - \sum_{i=\ell+1}^{L-1} a_i,$$

where

$$a_i = \begin{cases} 2\big(k^{(\ell+1)} - 1\big) \prod_{j=\ell+1}^{i} s^{(j)}, & \text{if layer } i \text{ is convolutional and } i+1 \text{ is max-pooling,} \\ 0, & \text{otherwise.} \end{cases}$$

Proof. If two convolutional layers i and $i+1$ are consecutive, then no max-pooling occurs between them, and therefore $|\mathrm{CRF}_{\ell,i}| = |\mathrm{RF}_{\ell,i}|$. Now suppose that i is followed by a max-pooling layer. In this situation, only those neurons in layer $i-1$ that contribute to all neurons of layer i within the same pooling window are guaranteed to influence the output. Consequently, the core receptive field shrinks compared to the full receptive field because the network satisfies Condition 2. Iterating this argument over all convolutional–max-pooling transitions between layers ℓ and $L-1$ gives the stated formula. □

Dealing with critical points of neurons on the edge. As defined above, the sizes of CRF and RF are computed only from the architectural parameters and do not account for situations where their leftmost or rightmost coordinates extend beyond the spatial boundaries of the input. Because of padding between layers, the RF and CRF regions may extend outside the valid input domain, and the ERF may not fully contain the CRF. Therefore, comparing the size of an ERF with the precomputed sizes of RF and CRF is meaningful only when the CRF is fully contained within the ERF. If the CRF region is connected, this full inclusion can be verified simply by checking whether any neuron in the ERF lies on the boundary of the input domain. However, if CRF is not connected, to prevent the distance between its connected parts from misleading the check, we require that the ERF maintains a margin from the border that is larger than the (constant) largest distance between the CRF components.

How large can the separation between two connected components of a CRF be? As shown in Fig. 11, consider a convolutional layer i followed by a max-pooling layer $i+1$. The gap between two CRF components on layer $i-1$ can be derived from the gap G_{i+1}

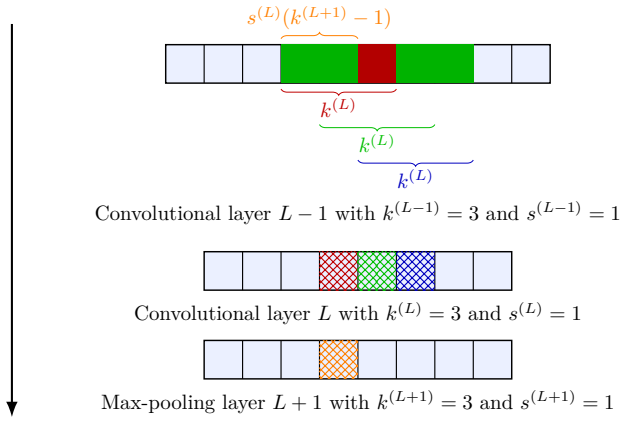


Figure 11: Illustration of the condition for the existence of a CRF in CNNs. The condition states that the $k^{(L+1)}$ convolutional neurons on layer L (marked by three colored crosses), which lie within the same max-pooling window corresponding to a neuron η (marked by the orange cross) on layer $L+1$, must have overlapping receptive fields with respect to layer $L-1$. The overlapped region (marked in red) forms the CRF, whereas the non-overlapping regions (marked in green) are discarded in the computation of Proposition 2.

between the corresponding CRF components on layer $i+1$, together with the structural parameters on layers i and $i+1$, as follows:

$$G_{i-1} = \max(0, s^{(i)}(s^{(i+1)} + G_{i+1} \cdot s^{(i+1)} + k^{(i+1)} - 1) - k^{(i)}).$$

Next, consider if i is followed by another convolutional layer. The gap between two CRF components on layer $i-1$ can be derived from the gap G_i between the corresponding CRF components on layer i, together with the structural parameters of layers i and $i+1$, as follows:

$$G_{i-1} = \max(0, (1 + G_i) \cdot s^{(i)} - k^{(i)})$$

By applying these two recursive formulas backwards, starting from a deeper layer L (where $G_L = 0$) and proceeding towards the target layer ℓ, we can derive the gap between the CRF components at each preceding layer.

Progressing through the network. For a fixed layer ℓ, the deeper a neuron in a layer $L > \ell$ lies, the larger its effective receptive field becomes. Beyond a certain depth, determined by the architecture, this region grows so large that it reaches the boundaries of the grid and the torus assumption diverts from reality. Neurons in even deeper layers exhibit the same behaviour. As a consequence, our identification strategy has a limited range that depends on the architecture: we can reliably classify only critical points originating from the next few layers.

The workflow is therefore as follows. Starting from the first layer, we collect a large number of critical points and compute their ERFs and insert them in the chain of CRFs. This allows us to classify many points belonging to the next few layers. For points from layers close to the limit of our range, classification is possible only when the corresponding neurons lie near the centre of the grid. For points coming from deeper layers, we can only identify them as originating from a sufficiently deep layer, without determining the exact one. We then recover the first layer using only those critical points that have been identified as belonging to it. After this recovery step, we reassess all points previously marked as too deep. Since one layer has now been extracted, our effective range increases, allowing

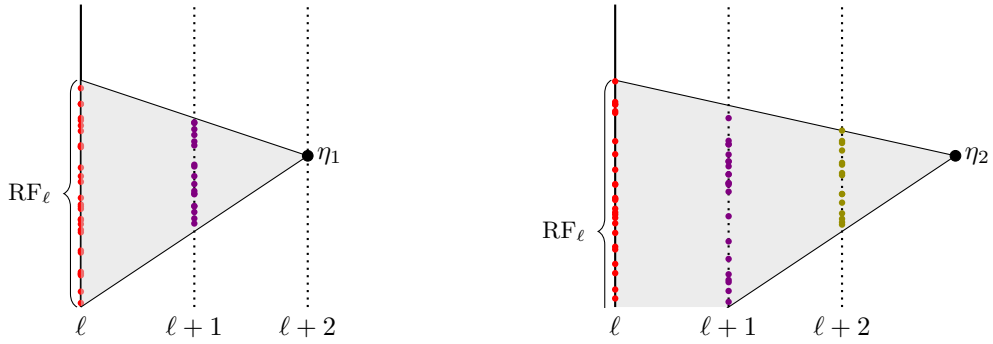


Figure 12: Evolution of the ERFs, shown as coloured points, as we progress through the network. Each vertical line represents a single channel. Although $|\mathrm{RF}_\ell(\eta_2)|$ is much larger than $|\mathrm{RF}_\ell(\eta_1)|$, this difference is not visible when observing $\mathrm{ERF}_\ell(x_1)$ and $\mathrm{ERF}_\ell(x_2)$ (both shown in red), where x_1 is a critical point of η_1 and x_2 of η_2. Because these ERFs touch the border, they appear truncated and are therefore discarded. After progressing to layer $\ell+1$, we reevaluate them. From $|\mathrm{ERF}_{\ell+1}(x_1)|$ (shown in purple), we infer that η_1 lies on layer $\ell+2$, while η_2 belongs to an even deeper layer.

us to classify additional points. We repeat this process iteratively until the extraction is complete. This evolution of the effective range is illustrated in Fig. 12.

5.2.1 Identifying the neuron of a critical point

To reduce memory requirements, we present a method for identifying, up to the filter, the neurons associated with a critical point originating from a deeper layer, under an additional condition on the target network.

Condition 3. *On any convolutional layer $i > \ell$, the sum of the center distance between the receptive fields of two nearby convolutional neurons and half of a neuron's CRF size must exceed half of the neuron's RF size. Formally, we require for any convolutional layer $i > \ell$,*

$$|\mathrm{CRF}_{\ell,i}|/2 + S_{\ell,i} > |\mathrm{RF}_{\ell,i}|/2.$$

This condition ensures that the CRF of one neuron does not lie entirely within the RF of a neighboring neuron, preventing ambiguity in identifying the associated convolutional neurons. Indeed, once we know the critical point is on layer i, we compute $|\mathrm{CRF}_{\ell,i}|$ and identify which CRF regions fall within $\mathrm{ERF}_\ell(x)$ by iterating over all possible CRF centers. In this way, we can precisely identify all CRF regions associated with x and determine their corresponding convolutional neurons, up to the filter.

6 Experiments

In this section, we evaluate our critical point localization and weight recovery techniques. We first analyze the distribution of ReLU critical points for bias recovery, and then validate the effectiveness of our reconstruction and localization procedures on a realistic convolutional architecture. Our code is available to ensure reproducibility at

https://github.com/PsyduckLiu/Model-Extraction-of-CNNs-with-Max-Pooling

All our experiments are conducted on an Intel Core i7-14700F CPU using a compact VGG-style CNN [SZ15]. The model is trained on the CIFAR-10 dataset, which consists of 32×32 RGB images of real-world objects from ten classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck) [KH09]. The network follows the standard VGG design paradigm, but at a smaller scale. It contains two types of convolutional layers: those directly followed by a max-pooling layer and those followed by another convolutional layer. A detailed description of the architecture and its parameters is given in Table 2.

Table 2: Architecture of the compact VGG-style CNN used in our experiments. For convolutional layers, p denotes the padding width applied to each side of the input, and s the convolutional stride. For max-pooling layers, s denotes the pooling stride.

Block	Layer	Type	Parameters	Output size	# Params
B1	1	Conv	4 filters, 3×3, $s = 1$, $p = 1$	$4 \times 32 \times 32$	112
	2	MaxPool	2×2, $s = 2$	$4 \times 16 \times 16$	0
B2	3	Conv	8 filters, 3×3, $s = 1$, $p = 1$	$8 \times 16 \times 16$	296
	4	Conv	8 filters, 3×3, $s = 1$, $p = 1$	$8 \times 16 \times 16$	584
	5	MaxPool	2×2, $s = 2$	$8 \times 8 \times 8$	0
B3	6	Conv	16 filters, 3×3, $s = 1$, $p = 1$	$16 \times 8 \times 8$	1 168
	7	MaxPool	2×2, $s = 2$	$16 \times 4 \times 4$	0
FC	8	FC + ReLU	$256 \to 16$	16	4 112
	9	FC + Softmax	$16 \to 10$	10	170

6.1 Experimental evaluation of the weight recovery

Proportion of ReLU points for bias recovery. We applied the critical point search algorithm to the trained model and collected 1,000 critical points. The distribution of ReLU points across layers is reported in Table 3. As expected, layers without max-pooling show only ReLU critical points. Results show that bias recovery is possible for all layers.

Table 3: Distribution of ReLU critical points for 1,000 collected critical points across layers and types.

Critical point	Conv.1	Conv.2	Conv.3	Conv.4	FC.1	Total
ReLU	133	276	88	35	5	537
Total	383	276	237	99	5	1000

Conv. = convolutional layer, FC. = fully connected layer.

Layer-by-layer model extraction results. We now present the layer-by-layer extraction results on the target CNN model. The results are reported in Table 4. For signature recovery, each convolutional layer contains multiple filters. The weight recovery ratio of a filter is defined as the fraction of its parameters that are recovered, i.e., (recovered weights + recovered bias)/(all weights + bias). We compute the weight recovery ratio separately for each recovered filter, and then report the worst, average, and best ratios over all recovered filters in that layer. For sign recovery, we report the number of filters whose signs are successfully recovered. Conv.2 being followed by another convolutional layer, sign recovery for this layer relies on the neuron wiggle technique [CMCSH+24]. Although signs can be recovered from each critical point, some sign-recovery failures may still occur when certain weights remain unrecovered. Consider a pooling window containing neurons η_i associated with inputs x_i. If a kernel coefficient a_j is missing, then replacing $a \cdot x + b$ with $a \cdot x - a_j x_j + b$ may change not only its sign but also its relative magnitude compared

Table 4: Layer-by-layer extraction results, where each layer is extracted under the assumption that all preceding layers have already been recovered. For Conv.2, which is followed by another convolutional layer, sign recovery is performed using Neuron Wiggle [CMCSH+24].

Layer	Filters	Always-off filters	CPs used	Signature recovery				Sign recovery
				Filters	Worst	Avg.	Best	
Conv.1	4	0	44	4/4	100.00%	100.00%	100.00%	4/4
Conv.2	8	1	1 000	7/8	100.00%	100.00%	100.00%	7/7
Conv.3	8	0	1 000	8/8	79.45%	90.75%	100%	7/8
			2 000	8/8	93.15%	96.75%	100.00%	8/8
			5 000	8/8	97.26%	99.14%	100.00%	8/8
			10 000	8/8	98.63%	99.49%	100.00%	8/8
Conv.4	16	4	1 000	11/16	79.45%	90.29%	98.63%	4/11
			2 000	11/16	90.41%	96.26%	100.00%	7/11
			5 000	11/16	98.63%	99.38%	100.00%	11/11
			10 000	12/16	86.30%	98.52%‡	100.00%	11/12

‡ For Conv.4 with 10,000 critical points, a newly recovered filter that is almost always off lowers both the worst and average recovery ratios. However, its sign cannot be determined due to unrecovered weights.

with other neurons in the same pooling window. In this case, the direct sign-recovery procedure described above may produce an incorrect result. Both issues could potentially be addressed by introducing practical assumptions on the range of admissible biases and weight values for the target network.

Overall, the results show that our method enables effective layer-by-layer extraction on the targer CNN. The first convolutional layer, Conv.1, is fully recovered using only 44 critical points, and Conv.2 is also fully recovered except for one always-off filter which does not contribute to the output. Weights connected to this always-off filter are considered recovered, since they do not affect the model output. For deeper layers, increasing the number of critical points consistently improves both signature recovery and sign recovery. Even with only 1,000 critical points in total, the average weight recovery ratio is at least 90.29% for every extracted layer. When the budget is increased to 10,000 critical points, the average weight recovery ratio reaches at least 98.52% for every layer. The number of filters with recovered signs also increases with the critical-point budget: for example, Conv.3 improves from 7/8 to 8/8, while Conv.4 improves from 4/11 to 11/12 over the same range.

The main limitation is that ReLU critical points are not always easy to find for every filter despite the distribution reported in Table 3, especially in deeper layers. Insufficient ReLU critical points can prevent bias recovery, so the corresponding neurons may not be fully recovered even if their weights are mostly extracted. This suggests that improving the search for ReLU critical points, as explored in [CTG+26], is an important direction for a more efficient end-to-end extraction.

6.2 Experimental evaluation of the localization strategy

Localization results. The localization results are summarized in Table 5. For each layer, we collect a fixed budget of 1,000 critical points. As a reminder, localization requires no additional queries and can be performed with very little computational effort. In our experiments, exactly all critical points satisfying our localization conditions are the ones assigned to the correct layer with respect to the ground truth; the ground truth is used only for evaluation and is not available to the attacker. As predicted by the theory, two phenomena are observed. First, critical points can often be correctly identified beyond

the next layer. Second, the number of correctly localized points increases as we progress through the network. Partial localization, in which a critical point is assigned to the correct layer but not to a specific neuron, is not included in the table. Appendix F discusses the causes of partial localization and how useful information can still be extracted from it.

Table 5: Critical-point localization results for the target CNN model using 1,000 critical points.

Layer	CPs on layer	CP classification			Localized CPs on		Unsafe target-layer CPs
		Prev. CPs	Reused CPs	Fully processed CPs	Target layer	Deeper layers	
Conv.1	383	0	0	1 000	296	249 161 on Conv.2, 87 on Conv.3, 1 on Conv.4	87
Conv.2	276	383	161	368	162	88 87 on Conv.3, 1 on Conv.4	114
Conv.3	237	659	87	253	134	13 13 on Conv.4	103
Conv.4	99	896	13	91	21	–	78

Conv. denotes a convolutional layer. Prev. CPs are critical points originating from layers before the target layer. Reused CPs are critical points that were already localized to the target layer during the recovery of previous layers. Fully processed CPs are the remaining critical points that require full signature recovery. Unsafe target-layer CPs are true target-layer critical points discarded because their ERFs intersect the boundary, where localization conditions are not applicable.

Efficiency gain from localization. The efficiency gain obtained from localization must be evaluated in two stages. First, we measure how many critical points can be localized. Second, we quantify the computational savings achieved for those points. The localization rates are reported in Table 5, while the savings obtained per localized critical point are given in Table 6. For Conv.2, localization allows us to completely skip processing 88/617 critical points and to solve a reduced system on 161/617 points. For Conv.3, these numbers become 1/341 and 87/341, respectively, while for Conv.4 they are 0/104 and 13/104. The effectiveness of localization decreases in deeper layers, mainly because the spatial dimensions become smaller. As a result, pooling windows more frequently intersect the boundary of the input grid, increasing the impact of the edge effects discussed in Section 5.2. This effect is reflected in Table 5 by the number of unsafe target-layer CPs discarded by the localization test: in our experiments, the corresponding ratios are 87/383, 114/276, 103/237, and 78/99 for Conv.1 to Conv.4, respectively. The frequency of this failure mode is model-dependent; in networks with larger feature maps, ERFs are less likely to touch the boundary, which can improve localization efficiency. In practice, however, CNNs often operate on substantially larger feature maps. For example, in VGG-16 the spatial dimensions range from 224×224 in the first layers to 14×14 in the deepest convolutional layers, whereas in our experimental network Conv.2 and Conv.3 operate on 16×16 feature maps and Conv.4 on only 8×8 feature maps.

Memory and runtime savings. We now evaluate the benefits of reusing previously localized critical points during extraction. Memory usage is measured by the number of query results that must be stored for the signature recovery of a single critical point. The results show that previously localized critical points require much less memory and computation than full-system processing. Despite the reduced localization rate observed in deeper layers, the savings obtained for each localized point remain significant. For Conv.4,

the deepest convolutional layer in our experiments, each previously localized critical point reduces the number of stored query results by at least 75.40% and the running time by at least 73.73% compared with solving the full system. These results indicate that localization provides a significant benefit by reducing both the memory usage and the running time for signature recovery. As explained in Section 5, memory and runtime savings also benefit from larger feature maps.

Table 6: Efficiency improvements due to localization. Crucially, we very conservatively overdetermine the system by twenty variables for stability. More aggressive solving would push the saved percentage higher. Our weight recovery is not optimised, but runs exactly the same for non-localized and localized, hence not affecting the saved percentage. Last layer is less effective because k_{conv} over d_{in} grows closer to one. As explained above, on industrial-sized networks, this ratio is much closer to zero and thus the saved percentage would increase.

Layer	**Average Memory per CP (float64 query)**			**Average Time per CP (s)**		
	Non-localized	**Localized**	**Saved (%)**	**Non-localized**	**Localized**	**Saved (%)**
Conv.1	$2^{14.59}$	–	–	$2^{5.38}$	–	–
Conv.2	$2^{12.81}$	$2^{8.79}$	93.80	$2^{3.65}$	$2^{-0.26}$	93.36
Conv.3	$2^{12.70}$	$2^{9.75}$	87.09	$2^{3.67}$	$2^{0.78}$	86.54
Conv.4	$2^{11.83}$	$2^{9.80}$	75.40	$2^{2.88}$	$2^{0.95}$	73.73

Conv. denotes a convolutional layer. For Conv.1, there are no previously localized target critical points, and hence no reduced-system recovery or corresponding saving is reported.

6.3 ReLU critical points in the attack: from FCNs to CNNs

We described max-pooling as a generalisation of ReLU, showing that on ReLU critical points the attack is almost the same as that performed on FCNs, only differing on the sign recovery part. A natural question is therefore whether an attacker could rely exclusively on ReLU critical points. Answering this question helps clarify the relationship between CNN extraction and FCN extraction, and indicates whether future improvements or defences developed for FCNs are likely to transfer to CNNs. It also provides insight into the effectiveness of defences that specifically aim at preventing the extraction of non-ReLU critical points. To investigate this question, we consider an attacker who discards all non-ReLU critical points. The attacker still exploits all other aspects of the attack, including localization, weight sharing in the sparse matrix representation of convolutions, and the easier sign recovery. This setting therefore corresponds as closely as possible to the FCN extraction scenario. We run the extraction procedure with 1,000 critical points and report the results in Tables 7 and 8.

Table 7 shows that restricting the attack to ReLU critical points significantly reduces the number of recoverable filters in deeper layers. This reduction closely follows the proportion of available ReLU critical points reported in Table 3. For example, on Conv.3, our method recovers all 8 filters, whereas the FCN-style variant recovers only 5 out of 8 filters (62%), compared with a ReLU-point proportion of 59%. A similar phenomenon is observed on Conv.4, where the FCN-style variant recovers 55% of the filters recovered by our method, matching exactly the proportion of ReLU critical points (55%). Conv.1 is an exception because all collected ReLU critical points already suffice for complete recovery, while Conv.2 contains ReLU critical points.

The quality of the recovered filters also decreases when only ReLU critical points are used, although the degradation remains relatively small. On Conv.3, the worst, average,

and best recovery ratios decrease from 79.45%, 90.75%, and 100.00% to 68.49%, 88.49%, and 95.89%, respectively. Similar behaviour is observed on Conv.4, where the average and best recovery ratios decrease from 90.29% and 98.63% to 89.95% and 97.26%. Sign recovery is affected as well, with three additional sign-recovery failures on each convolutional layer. A more important degradation is observed for localization (Table 8). However, this affects efficiency rather than the ability of a stronger attacker to perform an FCN-style extraction.

Overall, these results suggest that the effectiveness of FCN-style extraction on CNNs is largely determined by the proportion of ReLU critical points. This proportion is itself strongly influenced by the network architecture. For example, the authors of [CTG+26] report ReLU-point proportions of only 22% and 0.96% in their first two convolutional layers when using 5×5 kernels. More generally, larger pooling regions tend to reduce the proportion of ReLU critical points, making FCN-style extraction less effective. Conversely, architectures that chain convolutional layers with little or no max-pooling resemble FCNs more closely and are therefore more amenable to FCN-style extraction.

Table 7: Comparison of localization results with the FCN view under a budget of 1,000 critical points. For both Conv.1 and Conv.2 (where all the critical points are ReLU critical points), both strategies recover all the weights and signs. Though the recovery per filter is not much worse, the number of filters recovered is weak for the FCN method.

Method	**Layer**	**Signature recovery**				**Sign recovery**
		Filters	**Worst**	**Avg.**	**Best**	
CNNs	Conv.3	8/8	79.45%	90.75%	100.00%	7/8
	Conv.4	11/16	79.45%	90.29%	98.63%	4/11
FCNs	Conv.3	5/8	68.49%	88.49%	95.89%	4/5
	Conv.4	6/16	79.45%	89.95%	97.26%	1/6

Table 8: Comparison of localization results for each convolutional layer between the CNN and FCN-style extraction methods using a total budget of 1,000 critical points. The complete extraction of Conv.1 requires only 44 critical points, after which the attack proceeds to the next layer. Consequently, the numbers reported here are lower than those in Table 5.

Method	**Localized CPs**	**Conv.1**	**Conv.2**	**Conv.3**	**Conv.4**
FCNs/CNNs	Target Layer	4/10	156/156	15/47	5/8
	Deeper Layers	9/11	34/83	3/12	–/–

7 Conclusion

In this work, we studied the extraction of convolutional neural networks with ReLU and max-pooling layers. We explained how max-pooling generalises ReLU and showed how the limited relative information it reveals can be turned into full convolutional rows by using the typical locality of CNNs. We also proposed structural improvements that make the extraction much more efficient than simply reducing the network to a large fully connected model.

Scaling to large architectures is still computationally challenging, mainly because convolutional layers generate many critical points. However, our results show that the specific structure of CNNs can be used to simplify extraction. We believe that better search and filtering methods could make full end-to-end extraction of practical CNNs feasible in the future.

Acknowledgements. The authors with NTU affiliation are supported by the Singapore NRF-NRFI08-2022-0013 grant. Christina Boura is partially supported by the France 2030 program under grant agreement No. ANR-22-PECY-0010 Cryptanalyse. Haolin Liu is also supported by the National Natural Science Foundation of China (No. 62372294) and the China Scholarship Council (No. 202406230017).

References

[ANS19] André Araujo, Wade Norris, and Jack Sim. Computing receptive fields of convolutional neural networks. *Distill*, 2019. https://distill.pub/2019/computing-receptive-fields. doi:10.23915/distill.00021.

[CCSH+25] 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 Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part I*, volume 15601 of *LNCS*, pages 364–396. Springer, Cham, May 2025. doi:10.1007/978-3-031-91107-1_13.

[CDG+24] Yi Chen, Xiaoyang Dong, Jian Guo, Yantian Shen, Anyu Wang, and Xiaoyun Wang. Hard-label cryptanalytic extraction of neural network models. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part VIII*, volume 15491 of *LNCS*, pages 207–236. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0944-4_7.

[CJM20] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. Cryptanalytic extraction of neural network models. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 189–218. Springer, Cham, August 2020. doi:10.1007/978-3-030-56877-1_7.

[CMCSH+24] Isaac A. Canales-Martínez, Jorge Chávez-Saab, Anna Hambitzer, Francisco Rodríguez-Henríquez, Nitin Satpute, and Adi Shamir. Polynomial time cryptanalytic extraction of neural network models. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part III*, volume 14653 of *LNCS*, pages 3–33. Springer, Cham, May 2024. doi:10.1007/978-3-031-58734-4_1.

[CS25] Isaac A. Canales-Martínez and David Santos. Extracting some layers of deep neural networks in the hard-label setting. In Daniel Escudero and Ivan Damgård, editors, *LATINCRYPT 2025*, LNCS, pages 399–421. Springer, 2025. doi:10.1007/978-3-032-06754-8_15.

[CTG+26] Zirui Chen, Shi Tang, Zhengchao Gao, Yongjia Su, Lingyue Qin, and Xiaoyang Dong. Algebraic attack on convolutional neural networks with max pooling. In Nadia Heninger and Mike Rosulek, editors, *CRYPTO 2026, Part VII*, volume 16806 of *LNCS*, pages 3–35. Springer, 2026. doi:10.1007/978-3-032-35415-0_1.

[FMSH24] Hanna Foerster, Robert Mullins, Ilia Shumailov, and Jamie Hayes. Beyond slow signs in high-fidelity model extraction. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *NeurIPS 2024*, 2024. URL: http://papers.nips.cc/paper_files/paper/2024/hash/22ae669a35bb9e70eb93ab77c1eff5b4-Abstract-Conference.html.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016. doi:10.1109/CVPR.2016.90.

[KH09] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical Report 0, University of Toronto, Toronto, Ontario, 2009. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012.*, pages 1106–1114, 2012. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998. doi:10.1109/5.726791.

[LSE+26] Haolin Liu, Adrien Siproudhis, Samuel Experton, Peter Lorenz, Christina Boura, and Thomas Peyrin. Navigating the deep: End-to-end extraction on deep neural networks. In Joan Daemen and Emmanuel Thomé, editors, *EUROCRYPT 2026, Part VI*, LNCS, pages 482–512. Springer, 2026. doi:10.1007/978-3-032-25333-0_17.

[Pey10] Thomas Peyrin. Improved differential attacks for ECHO and Grøstl. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 370–392. Springer, Berlin, Heidelberg, August 2010. doi:10.1007/978-3-642-14623-7_20.

[SLW+26] 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. URL: https://eprint.iacr.org/2026/139.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL: http://arxiv.org/abs/1409.1556.

A Summary of notations

We summarize in Table 9 the notations used throughout the paper for a max-pooling convolutional network $\ell_{\mathcal{C}}$.

Table 9: Summary of notations

Symbol	Description	Dimension
d_{in}	spatial dimension of each input channel	$\mathbb{N}$
c_{in}	number of input channels	$\mathbb{N}$
d_{out}	spatial dimension of each output channel	$\mathbb{N}$
c_{out}	number of output channels	$\mathbb{N}$
Padding layer $\mathcal{P} : \mathbb{R}^{d_{\text{in}}^2 c_{\text{in}}} \to \mathbb{R}^{d_{\mathcal{P}}^2 c_{\text{in}}}$		
$d_{\mathcal{P}}$	spatial dimension after padding	$\mathbb{N}$
P	binary matrix implementing the padding operation	$\mathbb{R}^{c_{\text{in}} d_{\mathcal{P}}^2 \times c_{\text{in}} d_{\text{in}}^2}$
Convolution layer: $\mathcal{C} : \mathbb{R}^{d_{\mathcal{P}}^2 c_{\text{in}}} \to \mathbb{R}^{d_{\mathcal{C}}^2 c_{\text{out}}}$		
$d_{\mathcal{C}}$	spatial dimension after convolutional	$\mathbb{N}$
k_{conv}	convolutional kernel size	$\mathbb{N}$
s_{conv}	convolutional stride	$\mathbb{N}$
α_{ij}	i-th kernel matrix acting on the j-th channel	$\mathbb{R}^{d_{\mathcal{C}}^2 \times d_{\mathcal{P}}^2}$
α_i	filter made up of all the c_{in} kernels for output channel i	$\mathbb{R}^{d_{\mathcal{C}}^2 \times d_{\mathcal{P}}^2 c_{\text{in}}}$
A	global weight matrix	$\mathbb{R}^{d_{\mathcal{C}}^2 c_{\text{out}} \times d_{\mathcal{P}}^2 c_{\text{in}}}$
a	entry of A (a weight of the network)	$\mathbb{R}$
b	bias vector	$\mathbb{R}^{d_{\mathcal{C}}^2 c_{\text{out}}}$
b_i	i-th entry of b	$\mathbb{R}$
η_i	convolutional neuron, (function $x \mapsto A_i x + b_i$)	$\mathbb{R}^{d_{\mathcal{P}}^2 \times c_{\text{in}}} \to \mathbb{R}$
ReLU max-pooling layer $\mathcal{M} : \mathbb{R}^{d_{\mathcal{C}}^2 c_{\text{out}}} \to \mathbb{R}^{d_{\mathcal{M}}^2 c_{\text{out}}}$		
$d_{\mathcal{M}}$	spatial dimension after pooling (equals d_{out})	$\mathbb{N}$
k_{pool}	pooling window size	$\mathbb{N}$
s_{pool}	pooling stride	$\mathbb{N}$
I_x	binary selection matrix (local mask at x)	$\mathbb{R}^{d_{\mathcal{M}}^2 c_{\text{out}} \times d_{\mathcal{C}}^2 c_{\text{out}}}$
S_m	m-th pooling window (set of indices pooled together)	$\subset \mathbb{N}$

B Algorithmic description of FCN layer extraction without max-pooling

This section provides an algorithmic description of the layer-by-layer extraction procedure for fully connected networks without max-pooling. The algorithm follows the high-level steps described in Section 3.3: it first searches for critical points, then recovers partial signatures from second-order finite differences, merges compatible partial signatures into components, filters out components corresponding to deeper layers, and finally recovers the biases and signs of the target-layer neurons. Here, for $1 \leq i \leq r+1$, we define $F^{(i)} : \mathbb{R}^{d_0} \to \mathbb{R}^{d_i}$ by

$$F^{(i)} := \sigma^{(i)} \circ \ell^{(i)} \circ \sigma^{(i-1)} \circ \cdots \circ \sigma^{(1)} \circ \ell^{(1)}$$

with $F^{(0)}$ the identity and $F^{(r+1)} = f$. A more detailed presentation is given in previous works [CMCSH+24, CJM20, FMSH24, LSE+26] and exact implementation details are available in our commented code, as well as in the code of previous works.

C Proof of Property 1

Proof. When the $k_{\text{conv}} \times k_{\text{conv}}$ kernel is flattened row by row, moving one row down in the grid corresponds to skipping $d_{\mathcal{P}}$ entries in the flattened representation. Starting from the

Algorithm 2 FCN layer extraction: random search for critical points

Require: Black-box oracle $\mathcal{O}_f$ for the target network f; target layer index i; number of target neurons d_i; number of critical points N.
Ensure: Recovered layer parameters $\widehat{A}^{(i)}, \widehat{b}^{(i)}$.

Initialize $\widehat{A}^{(i)}$ and $\widehat{b}^{(i)}$ as length-d_i arrays filled with $\perp$.
Initialize the set of candidate critical points $\mathcal{X} \leftarrow \emptyset$.
while $|\mathcal{X}| < N$ **do**
    Sample a random line in the input space $\mathbb{R}^{d_0}$.
    Locate an interval on this line where the local affine derivative of f changes.
    Use binary search on this interval to locate a critical point x^*, and add it to $\mathcal{X}$.
end while
for each critical point $x^* \in \mathcal{X}$ **do** ▷ Discard deeper critical points
    Test whether the recovered hyperplane bends on an unrecovered layer.
    if the test indicates that x^* belongs to a deeper layer **then**
        Mark x^* as a deeper critical point.
    end if
end for
return $\mathcal{X}$.

Algorithm 3 FCN layer extraction: partial signature recovery

Continuation of Algorithm 2.

if the target layer is the final linear layer **then**
    Proceed directly to Algorithm 9.
end if
Initialize the set of recovered objects $\mathcal{P} \leftarrow \emptyset$.
for each critical point $x^* \in \mathcal{X}$ **do**
    Compute the local linear map $\widehat{\Gamma}_{x^*}^{(i-1)}$ of the extracted network up to layer $i-1$.
    Sample several random directions Δ.
    for each direction Δ **do**
        Query $\mathcal{O}_f$ near x^* and compute the second-order finite difference

$$\partial_\Delta^2 f(x^*) := \frac{1}{\varepsilon_1}\left(f(x^* + \varepsilon_2\Delta) + f(x^* - \varepsilon_2\Delta) - 2f(x^*)\right).$$

    end for
    Construct the linear system $\left(\widehat{\Gamma}_{x^*}^{(i-1)}\Delta\right) \cdot y = \partial_\Delta^2 f(x^*)$.
    Solve the system to obtain either a partial signature y_{x^*} or a solution space $\mathcal{S}_{x^*}$.
    Add the result to $\mathcal{P}$.
end for
return $\mathcal{P}$.

Algorithm 4 FCN layer extraction: intersections and partial signatures

Continuation of Algorithm 2.

Initialize the set of partial signatures $\mathcal{P}' \leftarrow \emptyset$.
for each result in $\mathcal{P}$ **do**
  if the result is already a one-dimensional partial signature **then**
    Add it to $\mathcal{P}'$.
  else
    Intersect its solution space with solution spaces obtained from other critical points.
    if an intersection has dimension one **then**
      Convert the intersection into a partial signature and add it to $\mathcal{P}'$.
    end if
  end if
end for
return $\mathcal{P}'$.

Algorithm 5 FCN Layer extraction: merging signatures into components

Continuation of Algorithm 2.

Initialize the set of components $\mathcal{C} \leftarrow \emptyset$.
for each partial signature $y \in \mathcal{P}'$ **do**
  if there exists a component $C \in \mathcal{C}$ whose recovered coordinates overlap proportionally with y **then**
    Rescale y on the common non-zero coordinates and merge it into C.
  else
    Create a new component containing y and add it to $\mathcal{C}$.
  end if
end for
return $\mathcal{C}$.

Algorithm 6 FCN layer extraction: discarding deeper components

Continuation of Algorithm 2.

for each component $C \in \mathcal{C}$ **do**
  Compute the component size $|C|$.
  Compute the ratio of merged critical points in C that are marked as deeper.
  Compute the number of unrecovered entries in the merged signature of C.
  if C is too small, or its deeper-point ratio is too large, or it has many unrecovered entries together with deeper-point evidence **then**
    Discard C.
  else
    Insert the signature of C into $\widehat{A}^{(i)}$. ▷ Seen as a candidate neuron in layer i.
  end if
end for

Algorithm 7 FCN layer extraction: recovering the bias

Continuation of Algorithm 2.

for each remaining component C **do**
  Let k be the index assigned to C in $\widehat{A}^{(i)}$.
  Choose a critical point x^* merged into C.
  Set $\widehat{b}^{(i)}[k] \leftarrow -\widehat{A}^{(i)}[k]\widehat{F}^{(i-1)}(x^*)$.
end for

Algorithm 8 FCN layer extraction: sign recovery

Continuation of Algorithm 2.

for each recovered component C **do**
  Let k be the index assigned to C in $\widehat{A}^{(i)}$ and $\widehat{b}^{(i)}$.
  Use neuron wiggle around the critical points associated with C to recover its sign s.
  Update the recovered parameters by setting $\widehat{A}^{(i)}[k] \leftarrow s \cdot \widehat{A}^{(i)}[k]$, $\widehat{b}^{(i)}[k] \leftarrow s \cdot \widehat{b}^{(i)}[k]$.
end for
Assemble the recovered entries into matrix-vector form.
return The recovered matrix $\widehat{A}^{(i)}$ and vector $\widehat{b}^{(i)}$.

Algorithm 9 FCN layer extraction: last layer recovery

Continuation of Algorithm 2.

Reuse previously queried inputs x and oracle outputs $f(x)$.
Construct the linear system $\widehat{A}^{(r+1)}\widehat{F}^{(r)}(x) + \widehat{b}^{(r+1)} = f(x)$ over sufficiently many queried points. ▷ Here the target layer is the last layer, i.e., $i = r + 1$.
Solve this system to recover $\widehat{A}^{(r+1)}$ and $\widehat{b}^{(r+1)}$.
return $\widehat{A}^{(r+1)}$, $\widehat{b}^{(r+1)}$.

first non-zero entry at index i, reaching the last row of the kernel requires $k_{\text{conv}} - 1$ such row transitions, which accounts for $(k_{\text{conv}} - 1)\, d_{\mathcal{P}}$ indices. Within the last row, moving from the first to the last column of the kernel adds $k_{\text{conv}} - 1$ additional indices. Therefore, the distance between the first and last non-zero entries is

$$j - i = (k_{\text{conv}} - 1)\, d_{\mathcal{P}} + (k_{\text{conv}} - 1),$$

which concludes the proof. □

D Algorithmic description of CNN layer extraction

For completeness, Algorithm 10 summarizes the extraction procedure for a convolutional layer. The merge operation follows Section 3.3: a newly recovered kernel is merged with a compatible recovered filter if one exists; otherwise, it is inserted into the next unrecovered entry. Rather than unifying ReLU and non-ReLU points' sign and signature recovery, we separate them to make it slightly faster for ReLU points.

E An example of shared recovered-neuron patterns across convolutional layers

We consider a specially designed CNN architecture to illustrate that two critical points originating from different convolutional layers can share the same pattern of recovered neurons. The corresponding architecture and its parameters are summarized in Table 10.

Algorithm 10 Layer extraction in CNNs

Require: Target layer index n; convolutional kernel size k_{conv}; padded spatial dimension $d_{\mathcal{P}}$; number of filters c_{out}
Ensure: Recovered layer parameters $\widehat{A}, \widehat{b}$

Initialize $\widehat{K}$, $\widehat{b}$, and $\widehat{s}$ as length-c_{out} arrays filled with $\perp$
Set $w_{\text{conv}} \leftarrow (k_{\text{conv}} - 1)d_{\mathcal{P}} + (k_{\text{conv}} - 1)$
Search for critical points and recover their signature using the method of Section 3.3
for each critical point x^* with recovered signature w **do**
    Let i and j be the first and last non-zero indices of w
    Set $s_{ab} \leftarrow |i - j| - w_{\text{conv}}$
    if $s_{ab} = 0$ **then** ▷ ReLU critical point of a neuron η_a
        Set the recovered filter row $\widehat{\alpha}_a \leftarrow w$.
        Extract the kernel of η_a from the non-zero entries of $\widehat{\alpha}_a$.
        Merge the kernel into $\widehat{K}$ and obtain its index r.
        if $\widehat{b}[r] = \perp$ **then** ▷ Recover the bias
            Set $\widehat{b}[r] \leftarrow -\widehat{\alpha}_a \widehat{F}^{(n-1)}(x^*)$.
        end if
        if $\widehat{s}[r] = \perp$ **then** ▷ Recover the sign
            Let S be the max-pooling window containing η_a
            if $\eta_b(\widehat{F}^{(n)}(x^*)) < 0$ for all $b \in S \setminus \{a\}$ **then**
                Set $\widehat{s}[r] \leftarrow +1$
            else
                Set $\widehat{s}[r] \leftarrow -1$
            end if
            Update the recovered parameters $\widehat{K}[r] \leftarrow \widehat{s}[r] \cdot \widehat{K}[r]$, $\widehat{b}[r] \leftarrow \widehat{s}[r] \cdot \widehat{b}[r]$.
        end if
    else ▷ Non-ReLU critical point of two neurons η_a and η_b
        Initialize $\widehat{\alpha}_a$ with zeros
        for $e \leftarrow i$ **to** j **do** ▷ Recover the signature
            if $e \leq i + s_{ab}$ **then**
                Set $\widehat{\alpha}_{a,e} \leftarrow w_e$
            else
                Set $\widehat{\alpha}_{a,e} \leftarrow w_e + \widehat{\alpha}_{a,e-s_{ab}}$
            end if
        end for
        Extract the kernel of η_a from the non-zero entries of $\widehat{\alpha}_a$
        Merge the kernel into $\widehat{K}$ and obtain its filter index r
        if $\widehat{s}[r] = \perp$ **then** ▷ Recover the sign
            Let S be the max-pooling window containing η_a and η_b
            if $\eta_a(\widehat{F}^{(n)}(x^*)) > \eta_k(\widehat{F}^{(n)}(x^*))$ for all $k \in S \setminus \{a, b\}$ **then**
                Set $\widehat{s}[r] \leftarrow +1$
            else
                Set $\widehat{s}[r] \leftarrow -1$
            end if
            Update the recovered parameters $\widehat{K}[r] \leftarrow \widehat{s}[r] \cdot \widehat{K}[r]$, $\widehat{b}[r] \leftarrow \widehat{s}[r] \cdot \widehat{b}[r]$.
        end if
    end if
    if all filters in the target layer are recovered **then**
        break
    end if
end for
Assemble the recovered kernel array $\widehat{K}$ into the convolutional matrix $\widehat{A}$ and the recovered bias array b into vector form.
return $\widehat{A}$, $\widehat{b}$.

Table 10: Architecture of a specifically designed CNN used in our experiments. For convolutional layers, p denotes the padding width applied to each side of the input, and s the convolutional stride. For max-pooling layers, s denotes the pooling stride.

Layer	Type	Parameters	Output size
1	Conv	4 filters, 3×3, $s = 1$, $p = 1$	$4 \times 32 \times 32$
2	MaxPool	2×2, $s = 2$	$4 \times 16 \times 16$
3	Conv	8 filters, 3×3, $s = 1$, $p = 1$	$8 \times 16 \times 16$
4	MaxPool	2×2, $s = 1$	$8 \times 15 \times 15$
5	Conv	16 filters, 2×2, $s = 1$, $p = 1$	$16 \times 16 \times 16$
6	MaxPool	2×2, $s = 2$	$16 \times 8 \times 8$
7	FC + ReLU	$256 \to 16$	16
8	FC + Softmax	$16 \to 10$	10

Next, we consider two critical points from different convolutional layers that nevertheless produce the same pattern of recovered neurons on the input grid. The first is a non-ReLU critical point x_1^* from Conv.2, while the second is a ReLU critical point x_2^* from Conv.3. For each point, Figs. 13 and 14 illustrate the neurons in the preceding layers that contribute to the critical point, together with the resulting recovered-neuron pattern on the input grid. Since these patterns are identical, simply matching recovered-neuron patterns is generally insufficient to eliminate noise originating from deeper layers.

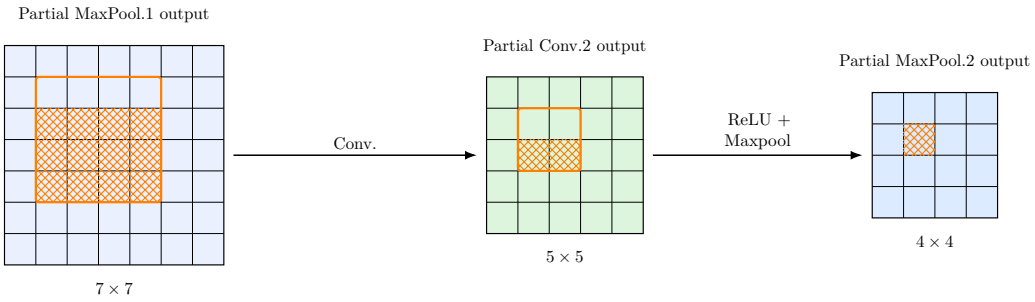


Figure 13: Derivation of the recovered-neuron pattern for a non-ReLU critical point x_1^* in Conv.2 with respect to the MaxPool.1 output. This critical point corresponds to two Conv.2 neurons (marked by orange crosses in the Conv.2 output grid) belonging to the same max-pooling window, whose output neuron is also marked by an orange cross in the MaxPool.2 output grid. Consequently, the recovered-neuron pattern on the MaxPool.1 output grid is the union of the convolutional windows of these two neurons and is highlighted by orange crosses.

If we collect 200 critical points for this CNN model, the three convolutional layers contain 58, 76, and 65 critical points, respectively. However, during model extraction on the second convolutional layer, three critical points originating from the third convolutional layer are misclassified as belonging to the second convolutional layer. This is caused by misleading recovered-neuron patterns in their signature vectors. This observation confirms that noise still exists in general CNN extraction.

F From partial localization to full localization

As shown in Table 11, under a total budget of 200 critical points, 4 of the 22 critical points assigned to convolutional layer 3 are localized to the correct layer but cannot be matched

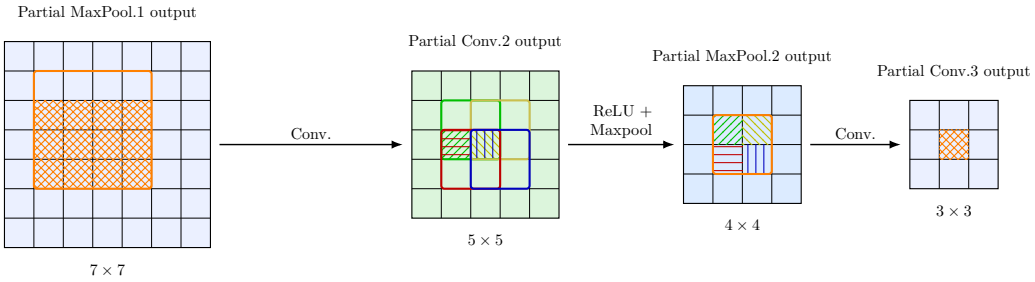


Figure 14: Derivation of the recovered-neuron pattern for a ReLU critical point x_2^* on Conv.3, with respect to the MaxPool.1 output. This critical point corresponds to a Conv.3 neuron (marked by orange crosses in the Conv.3 output grid). Its convolutional window covers four valid MaxPool.2 outputs, marked by green, yellow, red, and blue lines in the MaxPool.2 output grid, respectively. Within each of these max-pooling windows, the Conv.2 neuron with the largest output is marked using the same color and line style in the Conv.2 output grid. Therefore, the recovered-neuron pattern on the MaxPool.1 output grid includes all inputs within the convolutional windows of these four neurons with the largest outputs. This pattern is highlighted with orange crosses on the input grid.

to a specific neuron. This occurs when certain neurons in convolutional layer 2 remain inactive across all output channels within the relevant convolutional window. Because our toy model contains only 8 filters (compared to 64–512 in VGG-16), this situation appears more frequently. As a result, the recovered ERF is incomplete and does not fully cover the corresponding core receptive field (CRF), preventing precise neuron identification. However, since the output of the previous layer has already been extracted, the attacker can explicitly incorporate nearby inactive neurons as potential contributors to the ERF. After this correction step, all 22 critical points in convolutional layer 3 can be successfully localized. This experiment illustrates the discussion of Section 5.2: whenever the ERF is sufficiently informative to distinguish between candidate layers and neurons, correct localization follows.

Table 11: Critical point localization results for 200 critical points

Metric	**Conv.1**	**Conv.2**	**Conv.3**	**Conv.4**	**FC.1**
Distribution of 200 critical points	73	56	54	16	1
Critical points with distinguishable ERFs	57	31	22	4	-
Localized critical points	57	31	$18 \rightarrow 22^{\ddagger}$	4	-

‡ The increase is caused by observing which neurons are inactive on the previous layer, yielding full localization.