

CopyCNN: Reverse Engineering Based Extraction of Convolutional Neural Networks

Lusky, Yehonatan; Mendelson, Avi; and Nemcovsky, Yaniv

Abstract: *Neural network extraction attacks pose a significant threat to the security and intellectual property of AI systems, enabling adversaries to replicate proprietary models, compromise their confidentiality, or exploit their functionality. In this paper, we present CopyCNN, a novel attack that extracts the exact weights and architecture of black-box, deep, ReLU-based convolutional neural networks (CNNs) using only query access. To the best of our knowledge, CopyCNN is the first method capable of recovering precise weights from black-box CNNs. It effectively reverse engineers key components—including fully connected, convolutional, residual, pooling, and batch normalization layers—enabling the extraction of architectures such as AlexNet, LeNet-5, and variants of VGG and ResNet. We establish the theoretical basis for CopyCNN and demonstrate its effectiveness through the successful extraction of several architectures. Our results highlight the practicality of CopyCNN for convolutional neural network extraction and its implications on AI security and computer vision applications.*

Index Terms: *Convolutional neural networks, model extraction attack, reverse engineering*

1. INTRODUCTION

CONVOLUTIONAL neural networks (CNNs) have become a cornerstone of modern machine learning, driving breakthroughs in computer vision and related tasks such as image classification, object detection, and facial recognition. As these models grow increasingly complex and effective, the intellectual property (IP) embedded in their architectures and parameters becomes highly valuable—particularly for organizations that develop proprietary CNNs to maintain a competitive edge.

The widespread deployment of neural networks in real-world applications [1] has also introduced new security challenges. Among these, neural network extraction has emerged as a significant threat: it enables adversaries to reverse-engineer trained models to uncover their architecture and parameters. A successful extraction allows attackers to replicate a model's functionality without access to the original training data or computational resources. This poses serious risks to model owners, including intellectual property theft [2], the facilitation of adversarial attacks [3], [4], model poisoning [5], and even training data reconstruction [6]. Many of these attacks are far

more difficult, or even infeasible, when the model is accessed purely as a black box [7].

1.1. Related Work

Neural network extraction has been studied across various attack surfaces, including oracle-based methods, side channels, and fault injection. Oracle-based attacks aim to train a substitute model by classifying data according to the target model [8], [9]. These attacks replicate decision boundaries without access to internal parameters, and can be optimized through query synthesis and hyperparameter tuning [10]. In contrast, side-channel techniques passively infer model information through observable signals such as power consumption or memory access patterns [11]–[14], while fault injection methods actively perturb the system (e.g., voltage or EM glitches) to recover internal parameters [15].

A more recent and powerful approach involves reverse engineering, which uses carefully chosen inputs to directly infer architectural components and even the exact weights of the target model. This method, inspired by cryptanalytic chosen-plaintext attacks, achieves high precision with fewer queries and has shown promising results against MLPs [16]–[19]. Our work builds on this line of research allowing to reverse engineering convolutional architectures.

1.2. Our Contribution

We present **CopyCNN**, the first reverse-engineering-based model extraction attack capable of extracting both the architecture and the precise weights of ReLU-based convolutional neural networks (CNNs). While prior work has focused on multilayer perceptrons (MLPs), CopyCNN addresses the structural challenges of CNNs, including convolutional layers, residual connections, pooling, and normalization. We formalize the attack, implement it end-to-end, and demonstrate its effectiveness across multiple architectures, achieving high fidelity with low query complexity.

2. PRELIMINARIES

In this section, we introduce key notations and review essential concepts from previous attacks that are fundamental towards understanding our proposed attack. While we strive to maintain consistency with the notations used in prior work

for comparability, we introduce adjustments where necessary to align more closely with the specifics of our method, ensuring clarity and accuracy.

2.1. Notations

Notation 1: Let X and Y denote the input and output spaces, respectively. Given an input $x \in X$, let $F : X \rightarrow Y$ represent a neural network, where $y = F(x) \in Y$ is the output logit vector. The network is modeled as a composition of k functions, each corresponding to a layer:

$$y = F(x) = F_k \circ F_{k-1} \circ \dots \circ F_1(x) \quad (1)$$

Notation 2: Let $h_x^{(k)}$ denote the output of the k -th layer for input x . It is recursively defined by:

$$h_x^{(k)} = F_k(h_x^{(k-1)}), \quad \text{with } h_x^{(0)} = x \quad (2)$$

Notation 3: We denote common architectural components as follows:

- $f \leftarrow$ Fully connected layer
- $c \leftarrow$ Convolutional layer
- $\sigma \leftarrow$ ReLU activation function
- BN $\leftarrow$ Batch Normalization
- AvgPool $\leftarrow$ Average Pooling
- MaxPool $\leftarrow$ Max Pooling
- MinPool $\leftarrow$ Min Pooling

Notation 4: A fully connected layer is modeled as an affine transformation. Let $A^{(k)} \in \mathbb{R}^{m \times n}$ be the weight matrix and $B^{(k)} \in \mathbb{R}^m$ be the bias vector of the k -th layer. Then, for input $x \in \mathbb{R}^n$, the output is:

$$f_k(x) = A^{(k)}x + B^{(k)} \quad (3)$$

Notation 5: Let $\nu(\eta; x)$ denote the pre-activation value (i.e., before applying the activation function) of neuron η when the network is evaluated on input x .

Notation 6: A neuron η is said to be at a *critical point* for input x if its pre-activation value is zero, i.e., $\nu(\eta; x) = 0$.

Notation 7: The set of all inputs for which a neuron η is at a critical point is denoted $W(\eta)$:

$$W(\eta) = \{x \in X \mid \nu(\eta; x) = 0\} \quad (4)$$

Notation 8: Let e_j denote the j -th standard basis vector in $\mathbb{R}^n$ — a vector with 1 in the j -th coordinate and 0 elsewhere.

2.2. Network's Piecewise Linearity and Critical Points

The success of our extraction method relies on the piecewise linear nature of ReLU-based neural networks. While fully connected and convolutional layers are inherently linear, the ReLU activation function introduces non-linearities that partition the input space into distinct linear regions. Within each region, the network behaves as a linear function.

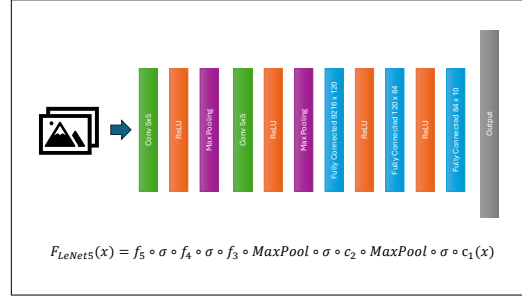


Fig. 1. Architecture of a modernized LeNet-5 neural network variant incorporating ReLU activation function

Formally, the ReLU activation at layer k can be modeled by a diagonal matrix $I^{(k)}$, where each entry is 1 for active neurons and 0 otherwise:

$$I^{(k)} = \begin{cases} 1 & \text{if } \nu(\eta; x) > 0, \\ 0 & \text{otherwise} \end{cases}$$

This yields the linear form: $F_k(x) = I^{(k)} \cdot (A^{(k)}x + B^{(k)})$.

The boundaries between linear regions correspond to discontinuities in the gradient of the network's output. The use of these gradient discontinuities, also known as critical points to reverse-engineer neural network parameters is rooted in classical cryptanalytic techniques, where structured input-output differences reveal internal system components [20].

To identify critical points, we use the 2-linearity algorithm, which locates them for arbitrary neurons without revealing their neuron or layer of origin [18].

3. OUR METHOD

Modern CNN architectures are composed of a diverse set of layer types, including linear layers (fully connected, convolutional, and residual blocks), pooling layers (min, max, and average pooling), normalization layers (e.g., batch normalization), regularization layers (dropout) and activation functions such as ReLU (Rectified Linear Unit). Prominent architectures within this scope include AlexNet, LeNet-5, ResNet, and VGG.

In this section, we establish the theoretical framework underpinning **CopyCNN**, detailing how it enables the weight extraction from CNN architectures. We systematically address the unique challenges posed by each of the possible layer types and present a tailored strategy to overcome them.

Our method extracts the weights layer by layer, following the sequential structure of the target model. Each layer-specific strategy serves as a building block of the overall attack and is applied in the sequential order, according to the architecture of the targeted model. To achieve this,

we classify the layers into two classes and outline the approach for each:

- For **Linear layers**:
 - Develop a method to extract the layer's weights.
- For **Non-linear layers**:
 - Explain how the overall piecewise linearity characteristic of the neural network is preserved despite the non-linear nature of the layer. This property is essential to maintain the existing extraction methods for fully connected layers [18].
 - Propose strategies to overcome the challenges posed by the layer while maintaining the extraction methods of the linear layers.

3.1. Convolutional Layers

Convolutional layers can be considered as a specialized form of fully connected layers. The convolution operation is a linear transformation, allowing it to be expressed as a matrix multiplication, similar to fully connected layers [21]. This reformulation enables the direct application of extraction techniques developed for fully connected layers [18], without requiring changes to the core algorithm.

However, matrices representing convolutional operations are inherently sparse and repetitive, as the same convolutional kernels are applied across overlapping input windows. We exploit these characteristics and propose an enhancement to the existing method.

The convolution weights can be represented as a 4D tensor:

$$C \in \mathbb{R}^{C_{out} \times C_{in} \times K_h \times K_w}$$

where C_{out} and C_{in} correspond to the output and input channels, respectively K_h and K_w represent the height and width of the convolution kernel.

Given a critical point x^* to a neuron η , where $x^* \in W(\eta)$ the neuron lies in one of the output channels, we'll denote this channel as c_o . Suppose we are extracting the weights of k -th convolutional layer, having already extracted all preceding layers. The value corresponding to neuron η is the result of applying the convolution kernel to the previous layer's output, h_{k-1} at an offset (x, y) over the input. The relationship can be expressed as:

$$\sum_{c_i=1}^{C_{in}} \sum_{i=1}^{K_h} \sum_{j=1}^{K_w} C_{c_o, c_i, i, j}^k \cdot h_{k-1, c_i, (x+i), (y+j)} + B_{c_o}^{(k)} = 0$$

For each convolutional layer, there are C_{out} sets of convolution kernels to extract, with each critical point enabling the extraction of one convolution

kernel set. Each set contains C_{in} kernels. This extraction process is analogous to the row-wise weight extraction of fully connected layers. We begin the extraction by sampling $N = C_{in} \cdot K_h \cdot K_w + 1$ directions around the critical point.

Next, we "slice" the input window that is covered by the convolution kernel and corresponds to neuron η 's value. We flatten the results into vectors, resulting in the following vectors:

$$\{v_j\} = \left\{ \left(\mathbf{1}(h_{x^*}^{(k-1)} > 0) \cdot A^{(k-1)} \cdot \dots \right. \right. \\ \left. \left. \dots \cdot \mathbf{1}(h_{x^*}^{(1)} > 0) \cdot A^{(1)} \cdot \delta_j \right)_{0:c_{in}, x:x+K_h, y:y+K_w} \right\}_{j=1}^{N+1}$$

Where δ_j represents the perturbation applied in the j -th direction and $\mathbf{1}$ represents the indicator function. To retrieve the weights, we proceed with finite differences attack [18] to solve a weight vector w such that $y_j = v_j \cdot w$ for all j , up to a global sign.

Since the neuron corresponding to a given critical point is not known a priori, we perform this calculation for every possible convolution window within the layer's input and then correlate the results using the weight correlation method described in Section 3-F. Notably, once the second gradient for each direction is computed, further queries to the original model are unnecessary. As so, the remaining calculations, including the window slicing and weight correlation, can be performed offline for each critical point (i.e., without additional queries to the target model).

For simplicity, convolutional layers can be treated as matrix multiplications, however in practice their extraction is made using this tailored method.

3.2. Residual Blocks

Residual blocks are widely used in modern neural networks, particularly in CNN architectures such as ResNet and its variants. Their ability to improve training stability and performance has made them a standard component in many state-of-the-art models for tasks such as image classification, object detection, and natural language processing [22].

Residual connections maintain the linearity of the neural network. In their simplest form, they add the output of a preceding layer to a subsequent layer. Other variations operate another piecewise linear function and add its output to a given layer output. Typically, a ReLU activation function is added after adding both results. The two computational paths in a residual block are commonly referred to as the *main path* (which undergoes a series of transformations) and the *shortcut path* or *skip connection* (which provides a direct bypass, often with fewer transformations).

For example, consider a residual block spanning from the k 'th layer to the $(k + L)$ 'th layer of a network. We denote this residual block as $F_{k...k+L}$ and define the total number of layers in the shortcut path as Z . The residual block can be mathematically expressed as follows:

$$\begin{aligned} F_{k...k+L}(x) &= \sigma \circ \left(F^{(Main)}(x) + F^{(Shortcut)}(x) \right) \\ &= \sigma \circ \left(F_L^{(Main)} \circ \dots \circ F_2^{(Main)} \circ F_1^{(Main)}(x) \right. \\ &\quad \left. + F_Z^{(Shortcut)} \circ \dots \circ F_2^{(Shortcut)} \circ F_1^{(Shortcut)}(x) \right) \\ &= \sigma \circ \left(f_L^{(Main)} \circ F_{L-1...1}^{(Main)}(x) \right. \\ &\quad \left. + f_Z^{(Shortcut)} \circ F_{Z-1...1}^{(Shortcut)}(x) \right) \end{aligned}$$

where $F^{(Main)}$ denotes the main computational path within the residual block and $F^{(Shortcut)}$ represents the shortcut path.

Unlike standard layers, where ReLU is applied to each individual layer's output, residual blocks apply ReLU only after summing the outputs of $f_L^{(Main)}$ and $f_Z^{(Shortcut)}$. This structural difference suggests a modified approach for extracting weights in the presence of residual blocks. Additionally, since a residual block consists of two separate computational paths, an important consideration is the order in which layers should be extracted.

1) *Layer Extraction Order*: Except for the final layers in the main and shortcut paths, the extraction of layers in each path can be performed independently. Within each path, layers must be extracted iteratively in the same sequential order as in a neural network without residual blocks. Once all layers in both paths (except for the final layers) have been extracted, we proceed to extract the last layers of the residual block, which require special treatment.

2) *Last Layers' Extraction*: Since a ReLU layer is applied after summing the outputs of $f_L^{(Main)}$ and $f_Z^{(Shortcut)}$, extracting their weights requires a modified approach. Specifically, given a critical point to neuron in that layer, weights from the main path and shortcut path must be considered simultaneously when applying the finite differences attack. To achieve this, we perform the following steps:

- 1) **Random Direction Sampling**: Sample a number of random directions equal to the total number of weights in both operators plus one. We denote it as (N) .
- 2) **Input Segmentation**: If one or more of the operators is a convolutional layer, select the input window according to the kernel applied.
- 3) **Equation Construction**: Construct equations for the finite differences attack

method based on the concatenated inputs for both operators.

Given the layer types and the two distinct computational paths that an input may traverse, we construct two sets of vectors: $\{v_j^{(Main)}\}$ for the main path and $\{v_j^{(Shortcut)}\}$ for the shortcut path. We then apply the finite differences attack to the concatenated vector set $\{v_j\}$, where:

$$\{v_j\} = \{v_j^{(Main)} | v_j^{(Shortcut)}\}$$

This process enables the parallel extraction of the weights from the final layers of the residual block: $f_L^{(Main)}$ and $f_Z^{(Shortcut)}$. Residual blocks introduce additional challenges in the weight extraction process. These challenges, along with solutions, are further discussed in Section 3-G.

3.3. Batch Normalization

Batch normalization is a technique used to stabilize and accelerate the training of deep neural networks by normalizing the inputs to each layer across mini-batches. It reduces internal covariate shift, which helps the network converge faster and become more resilient to different initializations. The data is normalized, shifted and scaled using learnable parameters μ , σ_{var} , γ and β , allowing the network to maintain representational capacity. During inference these parameters remain constant for each feature.

Given an input vector x , the batch normalization transformation produces an output $\hat{x}$, where each element $\hat{x}$ is given by:

$$\begin{aligned} \hat{x}_i &= BN_i(x) = \frac{x_i - \mu_i}{\sqrt{\sigma_{var_i}^2 + \epsilon}} \cdot \gamma_i + \beta_i \\ &= \frac{\gamma_i}{\sqrt{\sigma_{var_i}^2 + \epsilon}} \cdot x_i + \left(\frac{-\mu_i \cdot \gamma_i}{\sqrt{\sigma_{var_i}^2 + \epsilon}} + \beta_i \right) \end{aligned}$$

This transformation can be considered as fully connected layer, where the weight matrix A_{BN} is a diagonal matrix with $\frac{\gamma_i}{\sqrt{\sigma_{var_i}^2 + \epsilon}}$ at the i -th diagonal element, and the bias vector B_{BN} with $\frac{-\mu_i \cdot \gamma_i}{\sqrt{\sigma_{var_i}^2 + \epsilon}} + \beta_i$ at the i -th index.

In practice, batch normalization is typically applied after a linear layer and before the ReLU activation function. This allows us to treat the batch normalization and the linear layer as a single, functionally equivalent linear layer f' :

$$\begin{aligned} f' &= A_{BN}(A \cdot x + B) + B_{BN} = \\ &A_{BN}A \cdot x + A_{BN}B + B_{BN} = A' \cdot x + B' \end{aligned}$$

The equivalent weight matrix A' is given by $A_{BN}A$, which is the weight matrix A scaled by the diagonal elements of A_{BN} . Similarly, the equivalent bias vector B' can be computed as $B' = A_{BN}B + B_{BN}$.

This equivalence implies that batch normalization can be seamlessly incorporated

into the preceding linear layer during the extraction process. Consequently, the presence of a batch normalization layer has no meaningful impact on the extracted network. In fact, batch normalization layers can effectively be ignored during extraction, as the weights retrieved through the finite differences attack are already scaled.

3.4. Pooling Layers

Pooling layers are fundamental components in modern convolutional neural networks (CNNs), primarily used to reduce spatial dimensions. This downsampling lowers computational cost, reduces model complexity, and improves robustness against overfitting. Two common variants are max pooling and average pooling which we will discuss.

1) *Average Pooling*: Average pooling computes the mean over an $s \times s$ window in the input feature map. It can be expressed as a convolution with uniform weights of $\frac{1}{s^2}$ and zero bias. As a result, it is inherently linear and preserves the network's piecewise linearity.

In practice, average pooling is typically applied after ReLU and before the next linear layer. To accommodate this during extraction, we apply the same average pooling operation to the output of the preceding ReLU layer. The resulting feature maps are then used to construct the vectors $\{v_j\}$ for weight recovery procedure.

2) *Min/Max Pooling*: Min and max pooling introduce localized non-linearities, as the active neuron within a pooling window may change depending on the input. While these operations preserve the overall piecewise linearity of the network, they alter linear region boundaries.

Mathematically, this operation can be represented as a matrix multiplication. The 3D input tensor h'_k is flattened and multiplied by a matrix M defined as:

$$M = \begin{cases} 1 & \text{if element is the min/max} \\ & \text{within the window,} \\ 0 & \text{otherwise} \end{cases}$$

As long as the min/max indices remain fixed, M is constant, and the operation behaves linearly.

We note that during the critical point search, transitions in the active neuron within a pooling window may be misidentified as critical points. These false positives are illustrated in Fig. 2 and are later filtered during the weight correlation and unification process.

3.5. Dropout

Dropout is a common regularization technique in convolutional neural networks, where a fraction of neurons is randomly deactivated during training to prevent overfitting [23]. During inference, dropout is disabled and has no effect on the

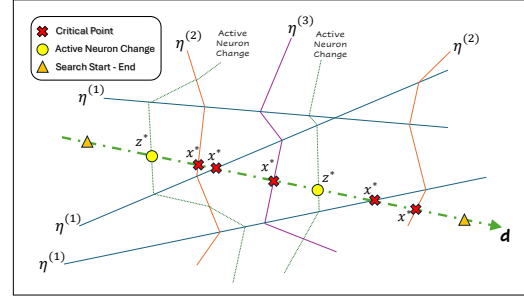


Fig. 2. This figure illustrates a critical point search process in a neural network containing a Max/Min pooling layer. Points labeled x^* are true critical points identified during the search, while z^* marks boundaries where the active neuron changes - falsely identified as critical points. The notation $\eta^{(i)}$ denotes neurons in the i -th layer.

network's structure or learned weights. As a result, CopyCNN does not need to account for dropout layers during extraction.

3.6. Weights Correlation and Unification

After extracting candidate weights from critical points, we must identify those that correspond to the current layer. This is complicated by false positives—especially in the presence of min/max pooling layers which may yield weights from unrelated neurons or layers.

To address this, we extend prior correlation and unification techniques [17], [18], adapting them for the challenges posed by CNNs. We define a correlation function f_{criteria} that identifies weight vectors as scaled versions of the same true weight (i.e., $w_1 = c \cdot w_2$) and use it to construct a graph where nodes represent extracted weights and edges indicate correlation.

Clusters (cliques) in this graph represent sets of consistent results. We unify each cluster by scaling to a common baseline and aggregating results (e.g., via element-wise median) to reconstruct full rows or kernels. Spurious weights typically fail to form large, consistent clusters and are naturally filtered out in this process. This combined correlation and unification step ensures robust recovery of the target layer's weights up to a global sign.

3.7. Global Sign Recovery

Recovering the global signs of elements in deeper layers poses challenges, especially in the presence of convolutions and residual blocks. This task involves determining the global sign of each element in a linear layer, i.e., each row in a fully connected layer or output channel in a convolutional layer. Residual blocks introduce further complexity. If both paths in the residual block start with the same type of layer and characteristics (e.g., fully connected or convolutional with the same kernel size), clustering may yield elements from both paths.

This makes it difficult to assign the correct global sign to each element and determine its origin path.

1) *Brute-Force Approach*: A straightforward solution is to brute-force all possible sign combinations and path assignments (in case a residual block is present). For each combination, if subsequent layer is extraction succeeds, it indicates the correct signs and assignments were found.

Formally, consider a residual block starting at the z -th layer, whose weights have been extracted up to their global signs. Let $\{x_j^*\}$ be critical points corresponding to a neuron in one of the $z + 1$ layers, and let $h_{x_j^*}^{(z)}$ denote the output from the z -th layers. There exists a vector $s \in \{-1, 0, 1\}^N$, a weight vector w , and a bias b belonging to one of the elements in the $z + 1$ layer such that:

$$s \cdot h_{x_j^*}^{(z)} \cdot w + b = 0, \quad \text{for all } j.$$

$s_i = 0$ indicates that the i -th element does not belong to the layer we are targeting, while $s_i \in \{-1, 1\}$ indicates the elements' global sign.

This brute-force method avoids additional queries, but is computationally expensive. With N elements, the number of combinations is 3^N , which becomes infeasible for large layers. For non-residual layers, where all elements originate in a single source path, the search space reduces to 2^N .

2) *Mixed-Integer Linear Programming (MILP)*: To overcome brute-force complexity, we use mixed-integer linear programming (MILP), which efficiently handles both discrete (sign) and continuous (weight) variables [24]. Prior work [18] showed that critical points for a neuron in the $z + 1$ layer can be retrieved using a linear number of queries. Given such a set $\{x_j^*\}$, we solve for s and w with the bias fixed to -1 :

$$s \cdot h_{x_j^*}^{(z)} \cdot w = 1, \quad \text{for all } j.$$

While MILP has worst-case exponential complexity, modern solvers perform well in practice, making it a practical and efficient solution for global sign recovery and element selection.

4. ARCHITECTURE EXTRACTION

So far, we have described how to extract a network's weights when its architecture is known. In practice, most models are deployed as black boxes, making prior knowledge of the type and order of layers unrealistic.

CopyCNN addresses this challenge without requiring additional queries. As long as the network consists of layer supported by our method, the architecture can be reconstructed by adjusting the weight extraction process. The successful extraction of a layer's weights inherently validates the guessed type and

structure of that layer, providing a guiding signal for reconstructing the architecture sequentially

4.1. Key Challenges and Brute-Force Strategy

During the extraction process, there are two steps that require the knowledge of the type, number and shape of the extracted layer:

- 1) **Weight Computation**: For convolutional layers, the kernel size must be known in order to apply the correct window to the layer's input and compute the weights. Without this information, the weight extraction process cannot proceed correctly.
- 2) **Clustering and Unification**: While clustering the weights graph to retrieve the extracted weights, it is essential to know the number of rows (for fully connected layers) or the number of output channels (for convolutional layers). This determines the number of clusters taken to the weights reconstruction.

Since this information is unknown, we adopt a brute-force approach. The number of parameters per element, denoted $n_{\text{parameters}}$, can be iteratively guessed. Begin with sufficiently small $n_{\text{parameters}}$ and perform the following:

- 1) Assume it is a convolutional layer and perform the weight extraction procedure given $n_{\text{parameters}}$.
- 2) Inspect the clustering results. If large, coherent clusters form, the guess is likely to be correct (Section 4-C). If not, increment $n_{\text{parameters}}$ and retry.
- 3) If $n_{\text{parameters}}$ reaches the input size, the layer is assumed to be fully connected.

As the second-gradient computations are independent of $n_{\text{parameters}}$, previously computed values can be reused, making the search computationally cheaper and query-efficient.

4.2. Handling Pooling Layers

Pooling layers (average, max, or min) and their window sizes are also unknown. We extend the brute-force process to test common configurations—e.g., 2×2 or 3×3 —with each pooling type. The small configuration space keeps this process tractable.

4.3. Inferring Output Size via Clustering

The number of elements in every layer - rows for fully connected layers or output channels for convolutional layers, can be inferred using clustering. As unrelated weight vectors rarely form large clusters, the number of significant clusters provides a strong signal about the true output size. Thus, clustering not only aids in weight recovery but also reveals structural information about the targeted layer.

5. EVALUATION

Our attack aims to extract the model’s weights as accurately as possible while minimizing the number of queries to the target network. This is critical because each query may incur costs—monetary, computational, or stealth-related (e.g., avoiding detection during model extraction). These constraints guided us in the design and implementation of CopyCNN.

5.1. Evaluation Methods

To ensure comparability with prior work, we adopted standard evaluation metrics from recent research [17], [18]: **functional equivalence** and **weight equivalence**. These metrics provide a comprehensive assessment of the attack’s success.

1) *Functional Equivalence*: Functional equivalence measures how closely an extracted model g replicates the behavior of the target model f . Formally, f and g are (ϵ, δ) -functionally equivalent over the input space S if:

$$\Pr_{x \in S} [|f(x) - g(x)| \leq \epsilon] \geq 1 - \delta.$$

2) *Weight Equivalence*: Weight equivalence measures the direct similarity between the extracted weights θ and the target model weights $\hat{\theta}$. This is quantified as $\max |\theta - \hat{\theta}|$. Since CopyCNN extracts an unordered and scaled version of the weights, we first align the extracted weights by reordering (e.g., rows, columns, or output channels) and scaling. After alignment, weight equivalence is computed to assess extraction accuracy.

5.2. Finite Precision

In theory, CopyCNN should enable the extraction of the exact model’s weights. However, practical limitations arise due to the finite precision of computational systems. Modern machines typically use floating-point arithmetic which introduces rounding errors and limited precision when performing operations on very small or very large numbers [25]. These numerical errors accumulate across layers, leading to slightly reduced precision in the extracted weights of deeper layers as depicted in Fig. 3.

The choice of data type used for mathematical calculations also plays a significant role. Higher-precision data types yield better extraction results by minimizing numerical errors. In our implementation, we used PyTorch’s `float64` data type to maximize precision. While lower-precision options are available, they lead to reduced accuracy and may necessitate additional queries to achieve sufficient precision to enable the extraction for deeper layers.

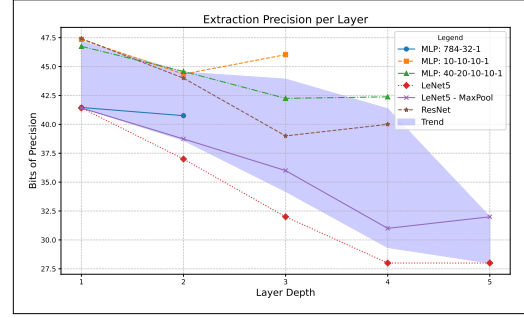


Fig. 3. This figure illustrates the precision (in bits) achieved in each extracted layer across the architectures under consideration. The graph reveals a trend where deeper layers generally exhibit reduced precision compared to earlier layers.

5.3. Results

The results of our experiments are presented in Table 1. For MLP architectures, although CopyCNN extraction methods are similar to the prior method [18], for many of the extractions we achieve results that are significantly superior in terms of the extraction precision, functional equivalence and total queries to the target model. We relate this to various optimizations that we have made through CopyCNN implementation.

The most notable achievement of CopyCNN is its ability to extract weights from architectures that were previously non extractable. In our experiments, we successfully extracted two variants of the LeNet-5 architecture: one with intermediate max-pooling layers and one without. This demonstrates CopyCNN’s capability to handle networks with convolutional layers, batch normalization, and pooling layers. Furthermore, we tested CopyCNN on a ResNet block, the residual block foundational in most ResNet variants - confirming its ability to extract networks with residual blocks.

Although these results offer strong evidence for the correctness and applicability of CopyCNN in CNN weight extraction, they are demonstrated on relatively simple architectures. Scaling the attack to deeper or more complex networks presents several implementation challenges. In practice, a successful extraction often requires careful fine-tuning and adjustment of multiple sensitive thresholds, making full automation difficult for large models. Additionally, as illustrated in Fig. 2, the phenomenon of critical hyperplane bending, where critical hyperplanes from deeper layers are distorted by those of preceding layers, making it harder to find certain critical points, especially in larger models.

6. DISCUSSION

6.1. Limitations

Our work has several limitations. First, our evaluation focuses on convolutional neural

Table 1
COMPARATIVE EVALUATION OF COPYCNN AND PRIOR METHODS UNDER A COMMON SETTING WHERE THE TARGET MODEL ARCHITECTURE IS KNOWN, LOGITS ARE OBSERVABLE, AND ARBITRARY INPUTS CAN BE QUERIED.

Attack	Queries	$(\epsilon, 10^{-9})$	$(\epsilon, 0)$	$\max \theta - \hat{\theta} $
MLP: 784-32-1				Params: 25,120
CopyCNN	2^{20}	$2^{-37.2}$	2^{-32}	$2^{-40.7}$
CJM [18]	$2^{19.2}$	$2^{-28.8}$	$2^{-27.4}$	$2^{-30.2}$
JCB [17]	$2^{18.2}$	$2^{3.2}$	$2^{4.5}$	$2^{-1.7}$
MLP: 10-10-10-1				Params: 420
CopyCNN	$2^{14.7}$	$2^{-44.7}$	$2^{-40.5}$	$2^{-44.3}$
CJM [18]	2^{16}	$2^{-42.7}$	$2^{-37.98}$	2^{-36}
RK [16]	2^{22}	$2^{-10.3}$	$2^{-3.4}$	2^{-12}
MLP: 40-20-10-10-1				Params: 1,110
CopyCNN	$2^{17.1}$	$2^{-41.5}$	$2^{-36.2}$	$2^{-42.2}$
CJM [18]	$2^{17.8}$	$2^{-31.7}$	$2^{-23.4}$	$2^{-27.1}$
LeNet5				Params: 1,117,770
CopyCNN	2^{34}	2^{-27}	2^{-23}	2^{-28}
LeNet5 w/ MaxPool				Params: 59,850
CopyCNN	$2^{31.7}$	2^{-29}	2^{-26}	2^{-31}
ResNet Block				Params: 3,096
CopyCNN	$2^{29.2}$	2^{-38}	2^{-35}	2^{-39}

networks that are relatively small compared to contemporary deep vision architectures, also, it does not address the modern and popular large-scale models. While the extraction of large convolutional networks is possible in principle, in practice we observed that finite numerical precision and accumulated reconstruction errors make it difficult to fully recover deep architectures such as VGG-style networks. This points to an important direction for future work - improving extraction robustness through better numerical methods and higher-precision computation. In this work, we therefore focused on smaller networks in order to validate the correctness and feasibility of our proposed method.

Second, our attack assumes access to pre-softmax outputs. This setting provides richer information than the hard-label regime, in which only the final predicted class is observable. Recent work has shown that multilayer perceptrons can be extracted even in the hard-label setting [19]. Extending such techniques to convolutional architectures remains an open problem and a promising direction for future research.

Finally, our attack requires a large number of adaptive queries and assumes that the adversary is not subject to strict query limits, rate limiting, or monitoring. In realistic deployment settings where such controls are typically enforced, the practicality of the attack may be reduced or may require significantly more time and resources.

6.2. Defenses

Various mitigations exist to defend neural networks against model extraction attacks such as CopyCNN. While there is no silver bullet, a layered defense can substantially increase the

cost and complexity of extraction. In deployed systems, neural networks should be treated as sensitive assets and protected accordingly.

1) General Best Practices:

- **Anomalous Query Detection:** Model extraction typically requires a large number of carefully structured queries. Monitoring query volume and distribution can reveal suspicious behavior. Indicators include unusually high query rates, repeated access from a single client, and inputs sampled from a typical or adversarial distributions. Practical mitigations include rate limiting, authentication with quotas, and behavioral fingerprinting [10].
- **API Output Hardening:** Public APIs should minimize the information revealed by model outputs. Returning only predicted class labels instead of logits or confidence scores significantly reduces the information available to the attacker and hinders substitute model training and gradient estimation.

2) CopyCNN-Specific Defenses:

- 1) **Input Randomization:** Adding controlled noise or random perturbations to inputs can reduce the attacker's ability to perform precise probing. However, such techniques must be carefully tuned to avoid degrading model correctness and may offer limited protection on their own.
- 2) **Network Swapping / Architectural Randomization:** Employing a moving-target defense, where multiple functionally equivalent models (e.g., with different architectures or weights) are randomly rotated at runtime, could increase the attacker costs significantly and may even

render extraction infeasible under realistic budgets.

- 3) **Label-Only Outputs:** Restricting outputs to hard labels (top-1 predictions) directly limits CopyCNN's ability to estimate gradients and recover internal parameters.

7. CONCLUSION

We have shown that precise, layer-wise extraction of convolutional neural networks is feasible in practice, and that both model architectures and parameters can be recovered with high fidelity under certain access assumptions. Beyond the specific techniques presented, we hope the methodology introduced in this work will serve as a useful foundation for future research on the extraction of larger, more complex, and different architectures of neural networks.

REFERENCES

- [1] W. Samek, G. Montavon, S. Lapuschkin, C. J. Anders, and K.-R. Müller, "Explaining deep neural networks and beyond: A review of methods and applications," *Proceedings of the IEEE*, vol. 109, no. 3, pp. 247–278, 2021.
- [2] J. Zhang, Z. Gu, J. Jang, H. Wu, M. P. Stoecklin, H. Huang, and I. Molloy, "Protecting intellectual property of deep neural networks with watermarking," in *Proceedings of the 2018 on Asia conference on computer and communications security*, 2018, pp. 159–172.
- [3] J. Ebrahimi, A. Rao, D. Lowd, and D. Dou, "Hotflip: White-box adversarial examples for text classification," *arXiv preprint arXiv:1712.06751*, 2017.
- [4] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song, "Generating adversarial examples with adversarial networks," *arXiv preprint arXiv:1801.02610*, 2018.
- [5] M. Fang, X. Cao, J. Jia, and N. Gong, "Local model poisoning attacks to {Byzantine-Robust} federated learning," in *29th USENIX security symposium (USENIX Security 20)*, 2020, pp. 1605–1622.
- [6] N. Haim, G. Vardi, G. Yehudai, O. Shamir, and M. Irani, "Reconstructing training data from trained neural networks," *Advances in Neural Information Processing Systems*, vol. 35, pp. 22911–22924, 2022.
- [7] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 506–519.
- [8] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction {APIs}," in *25th USENIX security symposium (USENIX Security 16)*, 2016, pp. 601–618.
- [9] T. Orekondy, B. Schiele, and M. Fritz, "Knockoff nets: Stealing functionality of black-box models," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 4954–4963.
- [10] M. Juuti, S. Szyller, S. Marchal, and N. Asokan, "Prada: protecting against dnn model stealing attacks," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 512–527.
- [11] L. Batina, S. Bhasin, D. Jap, and S. Picek, "Csi neural network: Using side-channels to recover your artificial neural network information," *arXiv preprint arXiv:1810.09076*, 2018.
- [12] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2003–2020.
- [13] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes attack: Steal {DNN} models with lossless inference accuracy," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [14] B. Coqueret, M. Carbone, O. Sentieys, and G. Zaid, "A hard-label cryptanalytic extraction of non-fully connected deep neural networks using side-channel attacks," *arXiv preprint arXiv:2411.10174*, 2024.
- [15] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Practical fault attack on deep neural networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2204–2206.
- [16] D. Rolnick and K. Kording, "Reverse-engineering deep relu networks," in *International conference on machine learning*. PMLR, 2020, pp. 8178–8187.
- [17] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot, "High accuracy and high fidelity extraction of neural networks," in *29th USENIX security symposium (USENIX Security 20)*, 2020, pp. 1345–1362.
- [18] N. Carlini, M. Jagielski, and I. Mironov, "Cryptanalytic extraction of neural network models," in *Annual international cryptology conference*. Springer, 2020, pp. 189–218.
- [19] N. Carlini, J. Chávez-Saab, A. Hambitzer, F. Rodríguez-Henríquez, and A. Shamir, "Polynomial time cryptanalytic extraction of deep neural networks in the hard-label setting," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2025, pp. 364–396.
- [20] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993.
- [21] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2017, pp. 19–24.
- [22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [23] G. Ghiasi, T.-Y. Lin, and Q. V. Le, "Dropblock: A regularization method for convolutional networks," *Advances in neural information processing systems*, vol. 31, 2018.
- [24] A. H. Land and A. G. Doig, *An automatic method for solving discrete programming problems*. Springer, 2010.
- [25] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM computing surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

Yehonatan Lusky is a Security Architect at NVIDIA and holds an M.Sc. in Computer Science from the Technion - Israel Institute of Technology. His research interests focus on system security, AI security, and the intersection between the two.

Avi Mendelson is a professor at the CS department at Technion, head of the ATLAS (AI Trust, Learning, Architecture, and Security) Lab, and an IEEE Fellow. He has experience in both industrial and academic settings, spanning security and safety of LLM-based systems, computer architecture, hardware security, and machine learning architectures. Among his industrial jobs, he worked for 11 years as a senior researcher and principal engineer at Intel, and served as a manager of the Academic-Relation of Microsoft R&D Israel. His current academic work focuses on Hardware Security and the security of LLM systems in general and agentic systems in particular.

Professor Avi Mendelson was a member of the ACM-Europe Council (2009-2014) and the IEEE Computer Society Board of Governors (2017-2019 as a member and second VP). He served as a member and chair of the Eckert-Mauchly Award committee. He also served as an associate editor for various journals, including IEEE CAL, IEEE Transactions on Computers, and IEEE Transactions on Emerging Topics in Computing

Yaniv Nemcovsky is an Associate Research Scientist in Enterprise AI at the Infosys Topaz Columbia University Enterprise AI Center, working with Professor Junfeng Yang. His research develops foundations for trustworthy AI by establishing robust performance evaluation and adaptive routing as core mechanisms for system reliability and safety across CV, NLP, RAG, and multimodal architectures. His work addresses physically realizable adversarial attacks on vision-based autonomous systems, jailbreak mechanisms and bias concealment in large language models, and success-based semantic representations for per-query performance prediction and model selection. His research on physical adversarial attacks received the Most Promising Paper Award at the ECCV 2022 AROW Workshop as one of the first credible demonstrations of real-world adversarial threats. Prior to joining Columbia, he was a Postdoctoral Researcher at Ben-Gurion University of the Negev with Professor Chaim Baskin, where he led a cooperative academia–industry research group on physical adversarial threats, and a Research Intern at IBM Research, where he studied industrial-scale multimodal retrieval systems. He received his PhD in Computer Science from the Technion in 2025, advised by Professors Alex Bronstein and Avi Mendelson, and during his doctoral studies served as an Adjunct Senior Lecturer at the Technion, where he designed and instructed a graduate-level course on adversarial ML. He also holds an MSc in Computer Science and a double BSc in Computer Science and Physics from the Technion. He has published in top-tier venues and co-authored successful national grant proposals, serving as assistant investigator on the resulting projects and directing collaborative teams of researchers and engineers.