Polynomial-Time Cryptanalytic Extraction of Deep Neural Networks in the Hard-Label Setting

Nicholas Carlini[1], Jorge Chávez-Saab[2], Anna Hambitzer[2], Francisco Rodríguez-Henríquez[2], and Adi Shamir ✉[3]

[1] Google DeepMind nicholas@carlini.com
[2] Cryptography Research Center, Technology Innovation Institute
{jorge.saab,anna.hambitzer,francisco.rodriguez}@tii.ae
[3] Weizmann Institute
adi.shamir@weizmann.ac.il

Abstract. Deep neural networks (DNNs) are valuable assets, yet black-box access to them raises security concerns about parameter extraction by malicious adversaries. Recent work by Carlini et al. (Crypto'20) and Canales-Martínez et al. (Eurocrypt'24) drew a parallel between model extraction and block-cipher key recovery via chosen-plaintext attacks. Using tools from differential cryptanalysis, they showed that all weights and biases of black-box ReLU-based DNNs can be recovered with a polynomial number of queries and polynomial-time post-processing. However, their attacks assume access to the exact numerical values of the output logits, enabling the computation of derivatives. To overcome this limitation, Chen et al. (Asiacrypt'24) studied the more realistic hard-label setting in which the attacker observes only the final classification label (e.g., "dog" or "car"). They proposed an extraction method that uses a polynomial number of oracle queries but has exponential running time. Moreover, their algorithm applies only to a restricted family of architectures, handles only binary classifiers, and is demonstrated only on very small networks with at most four neurons across up to two hidden layers. In this paper, we introduce new techniques that, for the first time, achieve cryptanalytic extraction of DNN parameters in the hardest hard-label setting using both a polynomial number of queries and polynomial-time computation for a broad class of ReLU-based networks. We validate our approach on a DNN trained on CIFAR-10 with 832 neurons in four hidden layers and nearly one million parameters in total. We further present an end-to-end black-box implementation of our attack, which we evaluate on a smaller artificial network, and extend our methods to leaky-ReLU activations. Our results show that, for our scenarios, all weights of a ReLU-based DNN can be efficiently recovered by analyzing only the geometric shape of its decision boundaries.

Keywords: ReLU-based deep neural networks · neural network extraction · hard-label attack · polynomial number of queries and polynomial-time attack.

Table of Contents

Polynomial-Time Cryptanalytic Extraction of Deep Neural Networks in the Hard-Label Setting 1
Nicholas Carlini, Jorge Chávez-Saab, Anna Hambitzer, Francisco Rodríguez-Henríquez, and Adi Shamir ✉
1 Introduction 3
1.1 Why Previous Techniques Cannot Be Used 4
1.2 Our Contributions 6
1.3 Organization 7
2 Related Work 8
3 Attack Overview 10
3.1 Basic Definitions and Notation 10
4 Dual Point Finding 16
5 Signature Recovery 18
5.1 Warm-up: Extracting the first layer 18
5.2 Extracting deeper layers 20
6 Sign Recovery 23
6.1 The Basic Technique 23
6.2 Algorithm 25
7 Experiments 28
7.1 Signature Recovery 29
7.2 Sign Recovery 31
8 End-to-end Black-box Attack 33
9 Inferring the Architecture 37
9.1 Complexity and Practical Efficiency 39
9.2 Dealing with persistent neurons 39
10 Extending to Leaky ReLUs 41
11 Conclusions 43
A Appendix: Supporting Figures on Experimental Outcomes 47

1 Introduction

Deep Neural Networks (DNNs) have become ubiquitous in today's technological landscape due to their ability to perform complex tasks such as image classification, speech recognition, natural language processing, and autonomous driving.

The simplest form of a DNN consists of a series of fully connected hidden layers of neurons. Each neuron in the network performs a scalar affine operation; a layer performs a global linear operation (over $\mathbb{R}^{d_i}$ for various round dimensions d_i) followed by the parallel application of local nonlinear operations over $\mathbb{R}$, such as the Rectified Linear Unit (ReLU) activation function. The DNN's parameters are generally obtained by collecting a huge corpus of training examples and iteratively adjusting an initial set of parameters through a lengthy sequence of gradient descent steps aimed at minimizing the DNN's loss on the training examples. This process can take many months and incur costs in the millions of dollars, making a trained DNN a highly valuable asset. However, this asset is often made accessible through an API/oracle interface $\mathcal{O}$ that allows anyone to input data to the DNN classifier and receive the corresponding answer.

The question of whether the DNN parameters can be determined through an oracle access to its black-box implementation has a rich history and has been addressed in numerous papers over the past 30 years (as detailed in Section 2). Recently, cryptographic researchers have noted the close similarities between the structures of DNNs and block ciphers, both of which consist of alternating layers of global linear operations and local nonlinear operations, such as ReLUs in the case of DNNs and S-boxes in the case of block ciphers (see Figure 1). In

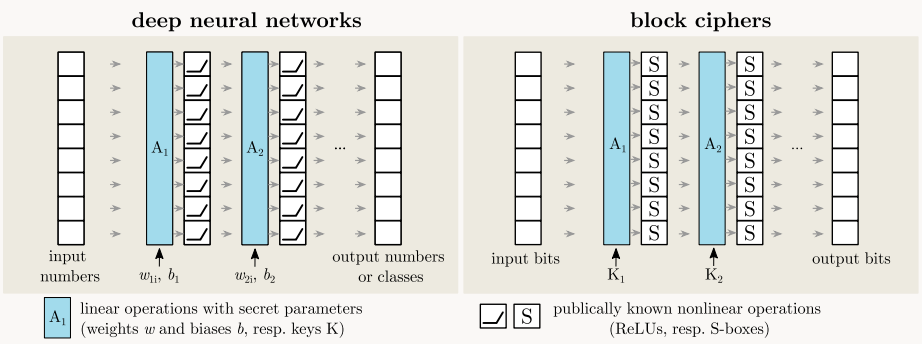


Fig. 1: The similarity between DNNs and block ciphers.

both primitives, the nonlinear operations are publicly known, while the linear operations involve secret elements — the parameters in DNNs, and the round keys in block ciphers. The goal of the attacker is to find these secret elements by applying an adaptive-chosen input attack. Unsurprisingly, the most effective model-extracting attacks currently leverage concepts from the differential cryptanalysis of block ciphers.

A common taxonomy for attack scenarios on DNN classifiers was introduced in [16]. The authors defined five distinct types of attacks based on the information provided by the black-box DNN in response to each input query: (S1) the most likely class label (referred to as the *hard-label* scenario), (S2) the most likely class label along with its probability score, (S3) the top-k labels and their probability scores, (S4) all labels and their probability scores, (S5) the raw output of the DNN (i.e., all the logits before normalizing them into a probability distribution). Scenario (S1) poses the greatest challenge for attackers, as it offers at most $\lceil \log_2 N \rceil$ bits of information per query for classifiers with N possible classes, while scenario (S5) is the least challenging, providing the complete numeric output of the DNN.

Most previous attacks on black-box DNNs have focused on the easiest scenario (S5), culminating in the work of [5], which requires a polynomial number of queries and polynomial time (as a function of the number of neurons). However, the challenge of finding the most effective attack in the hardest scenario (S1) has only been tackled in [8], over three decades after its initial introduction in 1991 in [3]. Nonetheless, the solution in [8] describes an attack that utilizes a polynomial number of *queries* but requires exponential *time*.[4] This algorithmic limitation forced the authors of [8] to report experiments on tiny networks with no more than four neurons split between at most two hidden layers.

1.1 Why Previous Techniques Cannot Be Used

To understand the difficulty posed by the hardest scenario (S1), consider the way most recent attacks, such as [5], operate, as illustrated in Figure 2. Any ReLU-based DNN represents a piecewise-linear mapping from real-valued inputs to real-valued outputs, which partitions the d_0-dimensional input space into a huge number of convex cells, as described in Figure 2a. Within each cell, the mapping

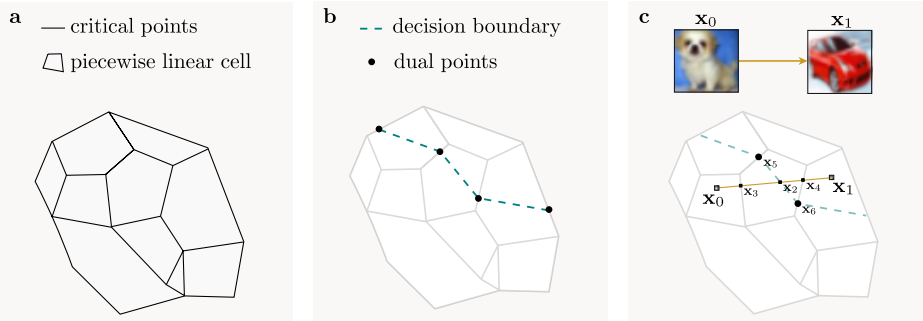


Fig. 2: A schematic description of various attacks. **(a)** Input space partition. **(b)** Decision boundary. **(c)** Path between two inputs.

[4] Moreover, Chen et al. obtain polynomial-time signature recovery only for a restricted subset of parameters and architectures.

behaves as a linear transformation, and the boundaries between adjacent cells are $d_0 - 1$-dimensional hyperplanes. These hyperplanes represent the points where at least one of the ReLU inputs becomes zero, which we refer to as *critical points* following the terminology proposed in [7] and adopted in [5,8]. Such ReLU flips change the behavior of the mapping from one affine mapping into a different one. Note that while the input/output mapping is always continuous, its derivatives become discontinuous at cell boundaries.

In addition to cell boundaries, we can also draw the decision boundaries between pairs of classes, whose points (which are called *transition points*) are the points at which the network's decision changes from one class to a different class.[5] Since the class decision is usually defined by the largest logit, decision boundaries are defined by linear inequalities between piecewise linear functions, and thus they are also piecewise linear $d_0 - 1$-dimensional hyperplanes which partition the d_0-dimensional input space into disjoint (not necessarily connected) regions, where each region represents the locations in which the hard-label assigned by the DNN is one of the classes. Note that decision boundaries can cut a single cell into two subcells which correspond to two different class decisions, even though the linear mapping in both subcells is the same. The connection exploited here is that between cell boundaries and decision boundaries (which are defined by critical points and transition points, respectively) is that within each cell the decision boundary must be a flat hyperplane, which typically changes its orientation as it crosses into an adjacent cell, as depicted in Figure 2b.

Most recent attacks have analyzed the output behavior as the input moves along a straight line between two points in the input space, such as x_0 and x_1 in Figure 2c. This path passes through various critical points such as x_3 and x_4. When dealing with the easiest attack scenario (S5), an attacker can precisely identify the location of such critical points, as the slope of the output changes abruptly. Each identified critical point offers valuable information about the value of some internal neuron during the evaluation of the DNN, by indicating that the input to its ReLU is zero at that point. This concept is akin to side-channel attacks that reveal internal values generated at various intermediate points during the encryption process, significantly reducing the complexity of key recovery. By collecting a sufficient number of such internal values, the attacker can recover the parameters of all the neurons by solving systems of linear equations in polynomial time.

The primary challenge in the hard-label attack scenario (S1) is the lack of access to the numeric values of the outputs, which prevents us from computing derivatives. Consequently, when we pass through critical points (such as points x_3 and x_4 in Figure 2c), we are unaware of this fact, as the DNN outputs "dog" for every input in the vicinity of x_3 and "car" for every input in the vicinity of x_4. The only event we can detect in this scenario is when we pass through the class

[5] Transition points are referred to as *decision boundary points* in [8]. For the sake of simplicity, we will ignore (in this brief description) the rare locations where there is a multi-way competition between three or more equal logits, where several decision boundaries intersect at the same point.

transition point x_2 where the hard-label produced by the network changes from "dog" to "car". However, x_2 is not associated with any internal ReLU flip (in fact, it only reveals that the two top logits *at the end* of the DNN computation are equal), and thus we no longer get the crucial side-channel information about values in the middle of the DNN computation which was revealed by crossing critical points in the easiest attack scenario (S5).

1.2 Our Contributions

The main contribution of this paper is to show that we can effectively replace the analysis of critical points in previous attacks by the analysis of class transition points, and thus recover all the DNN's secret parameters using a polynomial number of oracle queries and polynomial execution time by just analyzing the geometric shape of its decision boundary. In particular, we show that given any initial transition point, we can efficiently move along the decision boundary patch that contains it until the boundary changes its orientation. This can happen only at a point which is simultaneously a transition point and a critical point, and thus we can indirectly sense the location of some critical points even though the attack scenario does not allow us to sense them directly. We call points which are both transition points and critical points *dual points*, and provide two examples of such dual points (x_5 and x_6) in Figure 2c. Even though dual points are only an infinitesimally small subset of the set of all possible critical points, their analysis proves to be a sufficient alternative to the information provided by critical points in previous attacks.

Our new attack follows the same strategy as outlined in [7] and further developed in [5] and [8], but enhances it with novel techniques. The two main technical contributions of this paper are a new polynomial-time algorithm for recovering the critical hyperplanes of all the neurons [6] and a new polynomial-time sign recovery technique [7] in the hard-label scenario. In particular, sign recovery was the main bottleneck in [8]: the only solution the authors found for this problem was to perform an exponential time exhaustive search over all the possible sign combinations of all the neurons in the current layer.

It is important to note that our attack can fail in some situations which may happen in normally trained networks (and also particularly if the network was adversarially generated to resist our attack). For example, if some neuron plays no role in forming the shape of the decision boundary, we will not be able to find its weights. Another rare possibility is that some system of k random-looking linear equations in k unknowns over floating point reals, generated by our attack, has a determinant that is exactly zero (which will make it impossible to recover any additional parameters at later layers).

[6] We will refer to this process as *signature recovery*, which will be discussed in Section 5.

[7] That is, determining which side of the neuron's critical hyperplane provides positive values and which side provides negative values for the subsequent ReLU; this algorithm will be described in Section 6.

We demonstrate the principles behind our attack using a real DNN with 935,370 parameters that was trained on CIFAR-10 classes. This DNN consists of an input layer of length 3072, three fully connected hidden layers with 256 neurons each, another fully connected hidden layer with 64 neurons, and finally 10 neurons that produce the 10 output logits. We present evidence that our method could recover every parameter in this network,[8] for which the best previous attack of [8] would have been infeasible. We also present an end-to-end black-box implementation of the attack, which we validate on a smaller network. Finally, we propose variants of our methods that can be applied to networks with unknown architecture and to networks using leaky ReLUs.

Table 1: **A Comparison of Neural Network Extraction Methods.**

Neural Network			**Parameter Extraction**		
Architecture	Parameters	Hard-Label	Signature	Sign	**Approach**
10-20-20-1	620	✗	*poly*	*exp*	ICML'20 [24]
10-20-20-1	620	✗	*poly*	*exp*	Crypto'20 [7]
784-128-1	≈ 0.1M	✗	*poly*	*exp*	Crypto'20 [7]
784-16×8-1	14,481	✗	*poly*	*poly*	ArXiv'25 [18]
3072-256×8-10	≈ 1.25M	✗	*(from [7])*	*poly*	EC'24 [5]
784-16-16-1	12,849	✗	*poly*	*poly*	NeurIPS'24 [12]
1024-2-2-1	2,508	✓✗	*restr.*	*exp*	AC'24 [8]
3072-256×3-64-10	≈ 0.935M	✓	*poly*	*poly*	*This work*

✗: In these works the attacker has full access to all the output logits in scenario (S5). ✓✗: Restricted hard-label scenario, e.g. only binary classifiers can be handled. ✓: In these works the attacker works in the hard-label scenario (S1). *poly*: In these works, the signature, respectively, sign recovery is done in polynomial time. *exp*: In these works, sign recovery for most networks requires exponential time. *restr.*: The signature recovery polynomiality is only achieved for certain DNN architectures.

In Table 1, we compare our results to prior extraction attacks both in the hardest hard-label setting (S1) and in the easiest logit-output setting (S5).

1.3 Organization

The remainder of this paper is organized as follows. In Section 2, we provide a concise overview of the most relevant works in DNN parameter extraction within the black-box model along with the main precedents attacking the hard-label setting. In Section 3 we give an attack overview, and afterwards present a full description of the main components of our attack: dual point finding (Section 4), signature recovery (Section 5) and sign recovery (Section 6). The experiments conducted to demonstrate the effectiveness of our method are presented

[8] While we do not present a purely black-box attack for this DNN, we utilize white-box information to generate the statistical data required for the attack evaluation. Theoretically (and with infinite precision math) our algorithm is correct and reveals the exact neural network.

in Section 7, while Section 8 presents results from an end-to-end black-box implementation. We also show how to generalize the attack to the case of unknown network architecture in Section 9, and to the case of leaky ReLUs in Section 10. Finally, Section 11 presents our concluding remarks.

2 Related Work

The problem of DNN model extraction was first studied in the 1990s [2,3,4,13]. In 1991, Baum presented an algorithm in [2,3] that could infer the Boolean function computed by the network in polynomial time. This was achieved by utilizing chosen inputs and querying the DNN as an oracle to obtain their labels. Baum demonstrated that his algorithm could provably achieve probably approximately correct (PAC) learning in polynomial time for tiny networks consisting of up to four neurons, and he provided preliminary evidence suggesting that it could be extended to handle larger networks with up to 200 neurons. A few years after this research, Fefferman demonstrated in [11] that complete knowledge of all the (infinitely many) possible outputs from a sigmoid-based network uniquely determines its architecture and the weights of its neurons (up to some unavoidable symmetries). However, his technique did not provide an effective procedure for determining the actual network parameters. Another remarkable result on this topic was established by Blum and Rivest in 1993 [4]. They examined a different scenario in which the attacker was given an adversarially chosen set of known inputs and their corresponding outputs (i.e., the attacker could not choose her own queries). Their main result was that in this scenario, determining whether there exists a corresponding two-layer, three-neuron DNN (with the sign activation function instead of a ReLU) is NP-complete.

Since 2016, the extraction of DNN models from their black-box implementations has been extensively studied in [9, 12, 16, 20, 21, 23, 26]. The main goal of this line of research is to accurately infer all secret weights of the neurons in the DNN, achieving sufficient numerical precision to ensure functional equivalence between the extracted model and the original DNN. Current state-of-the-art attacks in this domain can be found in [5, 7, 8].

In [7], the authors introduced several efficient techniques for recovering neuron weights and their associated biases with remarkable precision, a process they termed *signature recovery*. These methods have polynomial query and running-time complexity. However, this approach can only determine the neuron's signature up to a constant multiplier of unknown sign, thus necessitating an essentially exhaustive search for inferring the neurons' signs, which has exponential time complexity. Consequently, the showcase examples of deep neural networks (DNNs) presented in [7] involved relatively shallow networks. This limitation was addressed in [5], where the authors introduced novel techniques for recovering the missing neuron signs in polynomial time, making it possible to demonstrate attacks on significantly larger and deeper DNNs.

More recently, Foerster et al. [12] reported an end-to-end attack that effectively combines the signature recovery methods from [7] with the sign recovery

techniques from [5]. They claim that their approach provides a more computationally efficient sign recovery process, resulting in significant speedups in the overall execution time of the attack. The authors note that, in this complete attack, most of the time is spent recovering signatures, with considerably less time devoted to extracting the signs.

Based on their experiments, they recommend brute-forcing the signs of particularly challenging neurons rather than spending excessive time trying to extract them. However, the DNNs targeted in [12] are notably shallow, consisting of no more than 100 neurons distributed across two hidden layers and a single logit output, which is not representative of real-world scenarios. For these relatively small networks, the authors demonstrate that it is feasible to exhaustively search the signs of a limited number of difficult-to-extract neurons. Nonetheless, it remains unclear how well this approach will scale to larger multi-output DNNs with thousands of neurons spread across more than a handful of hidden layers.

All the attacks discussed in [5, 7, 12] fundamentally rely on access to sufficiently precise numerical outputs (logits). This information allows attackers to estimate derivatives and gradients with respect to the input, which are essential for conducting their signature and sign recovery procedures.

A more realistic scenario was studied in [8], where the authors explored the so-called hard-label setting. In this context, the DNN outputs only the label of the class with the highest confidence score, while the confidence score itself—an invaluable piece of information for the attacker—remains secret. Extracting DNN models in this hard-label setting presents significantly greater challenges than those based on the assumptions of previous studies. As a historical note, the concept of the hard-label setting was first proposed over three decades ago by Baum in [2, 3], highlighting the longstanding difficulty of this problem, which remained largely open until the recent developments discussed here.

The attack presented by Chen et al. in this hard-label scenario efficiently recovers signatures by exploiting the information provided by what the authors called *decision boundary points*. However, their algorithm has several important technical limitations. In particular, their signature-recovery procedure applies only to a restricted family of architectures and supports only binary hard-label outputs. Moreover, they did not succeed in developing an efficient algorithm for sign recovery, relying instead on a guessing approach that makes the overall extraction algorithm exponential in running time. As a result, signature recovery is not solved in the general case, and sign recovery remains the primary barrier to attacking larger and deeper networks in this setting. In fact, the DNNs successfully targeted in [8] are restricted to those with a maximum of four neurons distributed across two hidden layers.

Very recently, Ito et al. [15] revisited the polynomial-time guarantees of hard-label cryptanalytic model extraction and identified an important limitation of approaches based on collecting dual points. Their analysis shows that, in trained networks, some neurons can be *persistent* (almost always active) or nearly dead, so that observing a state transition—and hence obtaining suitable dual points—may require exponentially many queries as the depth increases. They further

propose a cross-layer extraction technique that exploits interactions with subsequent layers to recover information about such persistent neurons without waiting for a direct state transition. We discuss this issue and its implications for architecture inference in more detail in Section 9.2.

Liu et al. [19] revisit the practical scalability of cryptanalytic extraction in the more informative setting where the attacker has access to numerical network outputs. They identify rank-deficient systems and interference from deeper layers as important bottlenecks in the signature-recovery procedure of Carlini et al., and propose techniques to mitigate these effects, improve numerical precision, and combine polynomial-time sign-recovery methods. Their resulting end-to-end attack extracts substantially deeper ReLU networks than previous end-to-end implementations. This work is complementary to ours: it improves the depth and robustness of extraction when rich numerical outputs are available, whereas our focus is the substantially more restrictive hard-label setting.

Finally, Asselineau et al. [1] extend cryptanalytic model extraction beyond piecewise-linear ReLU networks to a broad class of non-linear activation functions, including GELU, SiLU, SELU, and Sigmoid. Their approach exploits higher-order derivatives and the approximately linear behavior of these activations away from their non-linear transition regions; they further show that, in some cases, the activation function itself can be identified from black-box observations. Their attack assumes access to numerical network outputs and therefore follows the full-logit line of work. In this sense, their results are largely orthogonal to ours, in the sense that they broaden the class of activation functions that can be attacked under a richer oracle, while our work reduces the information required from the oracle.

3 Attack Overview

Recall that in the hard-label scenario, we assume a deep neural network processes inputs $\boldsymbol{x}$ layer-by-layer to produce a final output decision, such as “dog" (cf. Figure 3). Thus, in this scenario, only $\lceil \log_2 N \rceil$ bits of information per query are leaked for classifiers with N possible output classes.

3.1 Basic Definitions and Notation

Here we present some basic definitions and notation used throughout the manuscript which closely follow the terminology first presented in [7] and then adopted in more recent attacks [5, 8].

Definition 1. *An* r-deep neural network f_θ *is a function parameterized by* θ *(we write* $f := f_\theta$*, omitting* θ *when clear) that takes inputs from an input space* $\mathcal{X}$ *and returns values in an output space* $\mathcal{Y}$*. The function* f *is composed as a sequence of functions alternating between affine layers* f_i *(of different dimensions* d_i*) and a nonlinear function (which acts component-wise)* σ*:*

$$f = f_{r+1} \circ \sigma \circ \cdots \circ \sigma \circ f_2 \circ \sigma \circ f_1.$$

As in [5, 7, 8], we study deep neural networks (DNNs) where $\mathcal{X} = \mathbb{R}^{d_0}$, $\mathcal{Y} = \mathbb{R}^{d_{r+1}}$ and $d_0, \ldots, d_{r+1}$ are positive integers. Also, we only consider neural networks using the ReLU activation function defined as $\sigma : x \mapsto \max(x, 0)$.

Definition 2. *The i-th* fully connected layer *of a neural network is a function $f_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$ given by the affine transformation*

$$f_i(x) = A^{(i)}x + b^{(i)},$$

where $A^{(i)} \in \mathbb{R}^{d_i \times d_{i-1}}$ and $b^{(i)} \in \mathbb{R}^{d_i}$ are, respectively, the weight matrix *and the* bias vector *of the i-th* layer*, and d_{i-1}, d_i are positive integers.*

Definition 3. *A* neuron *is a function determined by a row of the corresponding weight matrix followed by an activation function. Particularly, the j-th neuron of layer i is the function η given by*

$$\eta_j^{(i)}(x) = \sigma(A_j^{(i)} \cdot x + b_j^{(i)}),$$

where $A_j^{(i)}$ and $b_j^{(i)}$ denote, respectively, the j-th row of $A^{(i)}$ and the j-th coordinate of $b^{(i)}$. An r-deep neural network has $N = \sum_{k=1}^{r} d_k$ neurons.

Definition 4. *The* architecture *of a fully connected neural network is described by specifying its number of layers along with the dimension d_i (i.e., number of neurons) of each layer $i = 1, \ldots, r+1$. We say that d_0 is the dimension of the inputs to the neural network and d_{r+1} is the number of outputs of the neural network.*

Definition 5. *The* parameters *θ of an r-deep neural network f_θ are the concrete assignments to the weights $A^{(i)}$ and biases $b^{(i)}$, $i \in \{1, 2, \ldots, r+1\}$.*

When working under the hard-label setting, the raw output $f(x)$ is processed before returning the result. We consider the processing employed in [8], which is presented in the definition below.

Definition 6. *Let $f : \mathcal{X} \to \mathcal{Y}$ be an r-deep neural network with $\mathcal{Y} = \mathbb{R}^{d_{r+1}}$. The* hard-label *output of f at $x \in \mathcal{X}$ is*

$$z(x) \;=\; argmax_{i \in \{1,\ldots,d_{r+1}\}} f(x)_i,$$

i.e., the index of the largest coordinate of $f(x)$.[9] *We use $N = d_{r+1}$ as the number of classes (outputs) of the network.*

Definition 7. *Let $\mathcal{V}(\eta; x)$ denote the value that neuron η takes with $x \in \mathcal{X}$ before applying σ. If $\mathcal{V}(\eta; x) > 0$ then η is* active*. Otherwise, the neuron is* inactive*. The state of η on input x (i.e., active, or inactive) is denoted by $\mathcal{S}(\eta; x)$.*

[9] If multiple entries in $f(x)$ have the same maximum value, the hard-label is the smallest coordinate of these equal entries.

Definition 8. *A critical point x satisfies $\mathcal{V}(\eta; x) = 0$ for some neuron η. We call x a critical point for η.*

Definition 9. *A point is said to be on the decision boundary of the network if the two highest logits of the output have the same value.*

Definition 10. *A* dual point *d is a point that lies on the decision boundary and on the critical hyperplane of at least one neuron, i.e., there exists a neuron η such that $\mathcal{V}(\eta; d) = 0$. We often also denote a dual point by $\boldsymbol{x}_{dual}$.*

Definition 11. *Let $x \in \mathcal{X}$. The* linear neighborhood *of x is the set*

$$\{u \in \mathcal{X} \mid \mathcal{S}(\eta; x) = \mathcal{S}(\eta; u) \text{ for all neurons } \eta \text{ in the DNN}\}.$$

Since the state of all neurons remains the same for all inputs in the same linear neighborhood, the DNN behaves as an affine map (within that linear neighborhood). Let $F_{i,j} = \sigma \circ f_j \circ \cdots \circ \sigma \circ f_i$, $1 \leq i \leq j \leq r$. Then, for all x in a linear neighborhood,

$$\begin{aligned} F_{i,j}(x) &= I^{(j)}(A^{(j)} \cdots (I^{(i+1)}(A^{(i+1)}(I^{(i)}(A^{(i)}x + b^{(i)})) + b^{(i+1)}) \cdots + b^{(j)}) \\ &= \Gamma x + \beta, \end{aligned}$$

where $I^{(\ell)}$ are $\{0, 1\}$-valued diagonal matrices with a 0 on the diagonal's k-th entry when neuron k at layer ℓ is inactive and 1 when that neuron is active.

Let $i \in \{1, \ldots, r+1\}$. The DNN can be regarded as a composition of an *input function* F_{i-1}, the i-th layer and an *output function* G_{i+1}:

$$f = \underbrace{f_{r+1} \circ \sigma \circ \cdots \circ \sigma \circ f_{i+1}}_{G_{i+1}} \circ \sigma \circ f_i \circ \underbrace{\sigma \circ f_{i-1} \circ \cdots \circ \sigma \circ f_1}_{F_{i-1}}.$$

When $i = 1$, F_0 is the identity map on the input; when $i = r + 1$, G_{r+2} is the identity map on the output. Otherwise, $F_{i-1} = F_{1,i-1}$. Furthermore, if we restrict inputs x' to be in the linear neighborhood of $x \in \mathcal{X}$, we can write F_{i-1} and G_{i+1}, respectively, as

$$F_{i-1}(x') = F_x^{(i-1)}x' + b_x^{(i-1)} \quad \text{and} \quad G_{i+1}(x') = G_x^{(i+1)}x' + b_x^{(i+1)}.$$

In the context of our attack, i is the index of the target layer and F_{i-1}, G_{i+1} represent, respectively, the recovered and non-recovered layers of the network. This view of the DNN is visually illustrated in Figure 3.

At a high level our attack extracts the parameters θ of a model layer by layer, and the analysis of each layer consists of two steps:

- *Signature recovery* extracts the parameters of each neuron up to an unknown nonzero scalar for each neuron. This determines the location of its critical hyperplane.
- *Sign recovery* determines the side of the critical hyperplane in which the linear function of the neuron produces positive values that pass unchanged through the subsequent ReLU.

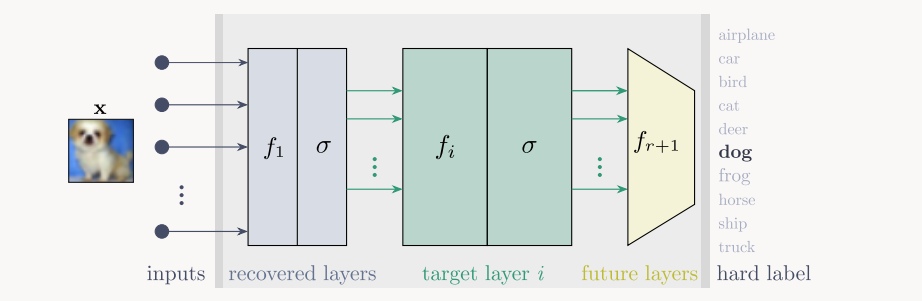


Fig. 3: Representation of the DNN as a composition of the recovered layers $f_1, \ldots, f_{i-1}$, the current target layer f_i and the non-recovered future layers $f_{i+1}, \ldots, f_{r+1}$ (cf. Definition 1).

The main difference from previous attacks is that we no longer have direct access to critical points; instead, we can only observe transition points between two predicted classes. We will concentrate on the subset of transition points which are also critical points. These are the dual points, and we can find them by moving along decision boundaries until their local orientation changes (detailed in Section 4).

Since dual points are defined by the intersection of two locally linear $d_0 - 1$-dimensional hyperplanes (defined by the critical and transition conditions, respectively), they form a locally linear $d_0 - 2$-dimensional subspace D which is associated with some neuron. Unfortunately, we can directly access only the transition hyperplane and not the critical hyperplane. We overcome this problem by computing the subspace D by intersecting two adjacent decision boundary patches with different orientations (instead of intersecting a decision hyperplane and a critical hyperplane). In other words, once we discover some dual point $\boldsymbol{x}_{\text{dual}} \in D$, we can explore the vicinity of $\boldsymbol{x}_{\text{dual}}$ in order to find the local orientations of the two decision boundary patches on its two sides, intersect them, and thus discover the whole subspace D from a single point $\boldsymbol{x}_{\text{dual}}$ in it. Since D is a $d_0 - 2$-dimensional subspace of the $d_0 - 1$-dimensional critical hyperplane of some neuron, knowledge of this D makes it possible to find most (but not all) of the parameters of this neuron by just moving along the DNN's decision boundary and exploring its geometric shape.

A 3D depiction of this situation appears in Figure 4, where the vertical green 2D plane is the (unknown) critical subspace of some internal neuron, and the two 2D almost horizontal patches of the decision boundary on its two sides change their local orientation on the two sides of the critical plane. The intersection of the two 2D decision boundary patches yields a 1D line of dual points, which are all contained in the unknown critical plane. The only thing we do not observe about this critical plane is how it rotates around the known line of critical points within it.

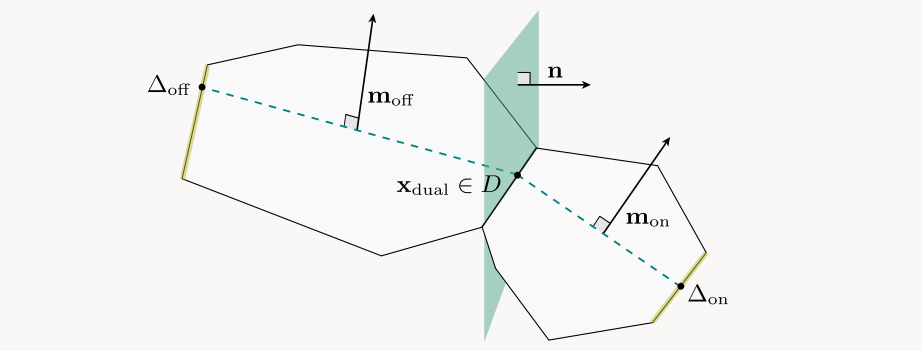


Fig. 4: Geometry around a dual point $x_{\text{dual}} \in D$. The neuron's critical hyperplane, shown in green and with normal vector n, intersects two adjacent decision-boundary patches with normals m_{on} and m_{off}. Their intersection defines the dual subspace D, while Δ_{on} and Δ_{off} denote the distances traveled along the two patches before the next change in boundary orientation.

To get a complete description of the neuron, we just have to find another instance of criticality of the same neuron in a different location of the input space. Our analysis is likely to yield a differently oriented subspace of dual points within it, and by combining the information they provide, we are likely to find the missing parameters of that neuron. Note that each piece of partial information can be viewed as a set of linear equations, and thus, to determine whether two subspaces of dual points were produced by the same neuron, we simply have to check if the two systems of equations are consistent. If they are, their common solution is likely to provide a full description of this neuron, including its missing orientation around each one of the two subsets of dual points we have found within it.

The one remaining mystery about this neuron is which one of its two sides corresponds to positive inputs to its ReLU. This is referred to as the problem of finding *the sign of the neuron*, and without having this crucial information about all the neurons in the first $i-1$ layers of the DNN we will not be able to peel off the effect of these layers and proceed to the analysis of the i-th layer neurons, as depicted in Figure 3.

The attack presented in [5] solved this problem in the easy scenario (S5) by noticing that the output of the network tended to change faster (i.e., had a higher value for the larger absolute slope), when the input x moved from the side of negative ReLU inputs (where the output of this neuron was stuck at zero) to the side of positive ReLU inputs (where it was allowed to change and thus contributed to the overall change in the output). This effect could be maximized by perturbing the input in a direction that changes the targeted neuron, which is the direction that is perpendicular to its critical hyperplane. While each individual test of slopes may give a wrong answer, we are likely to see a statistically significant difference between the speed of change of the outputs

on the two sides of this neuron when we explore sufficiently many critical points belonging to the same neuron during a large number of traversals of the input space. Unfortunately, in the hard-label scenario (S1) considered in this paper we no longer have access to the numeric values of the output, and thus cannot compute the rate of change of these values.

The way we solve the sign problem in the hard-label scenario is by exploiting a different type of measurement which is also strongly correlated to the speed of change of neuronal values, and can thus act as a proxy for the immeasurable output slopes. Consider Figure 4 once again, where we are given some dual point $\boldsymbol{x}_{\text{dual}}$. After finding the two $d_0 - 1$-dimensional decision boundary patches on its two sides, we can find the two dashed lines which start at $\boldsymbol{x}_{\text{dual}}$, lie on the two decision boundary patches, and are perpendicular to D. Each one of these line segments ends when *another neuron* flips sides, and thus changes again the orientation of the decision boundary. We can now measure the two distances Δ_{on}, Δ_{off} required to traverse the two sides of the critical hyperplane until this change occurs. If this other neuron that flipped is at one of the later layers (compared to layer i where the targeted neuron resides), then the speed at which its inputs change is affected by whether the targeted neuron is contributing to the change or not. This implies that typically (but not always), we expect decision boundary patches to be narrower on the positive side of the ReLU than on the negative side of the ReLU (i.e., that $\Delta_{\text{on}} < \Delta_{\text{off}}$). This is a measurable property even when the attacker has only access to the hard labels assigned to inputs. In Section 7 we show experimentally that this statistical difference becomes very prominent once we test a sufficiently large number of dual points $\boldsymbol{x}_{\text{dual}}$ which all lie on the critical hyperplane of the same targeted neuron.

One important observation is that any particular patch of the decision boundary can terminate when flipping neurons which are located within the DNN either before or after the targeted neuron. Only neurons in later layers can possibly be affected by whether the targeted neuron's ReLU was on its positive or negative side. Therefore, we should discard any patches that are ended by earlier-layer neurons from our statistics. Fortunately, we already know the signature of all these earlier-layer neurons, so we can test whether any known neuron is critical at the point where the patch ends.

Adversarial Goals and Assumptions. According to our security model, the attacker can adaptively select queries x as inputs to oracle $\mathcal{O}$, which returns the label of x produced by the DNN f_θ. The attacker's objective is to obtain the extracted parameters $\hat{\theta}$ which are the same as the original parameters θ, up to roundoff errors produced by using finite precision real numbers,[10] and up to unavoidable symmetries (such as permuting the order of internal neurons).

In addition, we make the following typical assumptions regarding the capabilities of the attacker mirroring prior work on this topic [7]:

[10] Note that increasing the number of floating point bits by a factor of k exponentially reduces these roundoff errors, while increasing the time complexity of the arithmetic operations in our attack by only a polynomial factor of k^2.

– **Knowledge of the architecture**. The attacker has knowledge of the number of layers, the number of neurons in each layer, and the number of inputs and outputs in the network. We make this assumption to simplify the description of the attack, but later show how to lift it.
– **Full-domain inputs**. The attacker can query arbitrary inputs from $\mathbb{R}^{d_0}$.
– **Precise computations**. The DNN is specified and evaluated using sufficiently high precision floating-point arithmetic.
– **Fully connected network and ReLU activation functions**. The network comprises fully connected layers and all activation functions are the ReLU function (later we generalize this to the leaky ReLU function as well).

Compared to [8], we do not assume that the classifier is binary, and can accommodate any number of classes (with multiple decision boundaries between them).

4 Dual Point Finding

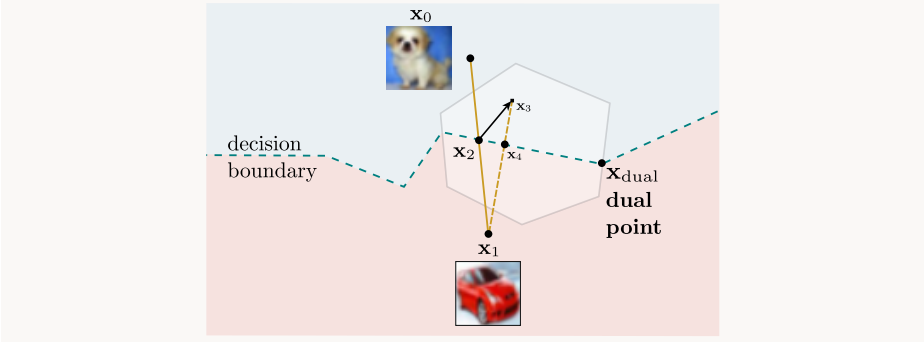


Fig. 5: Visualization of our dual point finding algorithm.

The value of critical points is well understood in the literature [5,7], because the location of these points fully determine the parameters of a neural network. Unfortunately, without the ability to directly access the function $f(\cdot)$ that returns the logits of the model, it is not possible to identify general critical points. Instead, we use dual points (cf. Definition 10): inputs that sit both at the decision boundary and also on the critical hyperplane of some neuron. Finding dual points is a key capability used throughout the attack. To identify dual points, we implement a simple algorithm, visualized in Figure 5:

Step 1: find a point x_2 on the decision boundary. Initially, we sample two random inputs x_0 and x_1 that are labeled differently by the neural network, i.e., $z((f(x_0)) \neq z((f(x_1))$. We then perform binary search starting from these two

points to find the exact location of the decision boundary where $z(f(x_2))$ changes from one class to another. This gives us a point on the decision boundary, or more precisely, a point x_{left} which is slightly off from the boundary by a small distance ϵ depending on the number of rounds in the binary search.

Step 2: make a random excursion to x_3. We move away from x_2 in a random direction to reach a new point x_3.

Step 3: find another point x_4 on the decision boundary. Evaluate the class $z(f(x_3))$. If the class equals the label of x_0, find the transition point x_4 using a binary search between x_3 and x_1. If the class corresponds to the one of x_1, find the transition point x_4 using a binary search between x_3 and x_0 instead.

Step 4: move along the decision boundary to a dual point $\boldsymbol{x}_{dual}$. By following the direction $dx = x_4 - x_2$ we can move to $x_\alpha = x_2 + \alpha \cdot dx$. For a small α the value x_α remains on the decision boundary. But after some distance, eventually x_α will deviate away from the decision boundary. Or, more accurately, the decision boundary will deviate away from x_α. Why is this? Recall that ReLU neural networks divide the input space into piecewise linear regions. Within any region, the decision boundary will behave completely linearly, but once we cross from one linear region to another the decision boundary will *bend* and follow a different linear behavior. The location of the bend is exactly the dual point, which is both on the decision boundary and on some neuron's critical hyperplane.

Step 5: re-locate the decision boundary. After crossing the critical hyperplane, we now project this point back onto the decision boundary. We refer to this point as x_{right}. At this point, we have collected three points: ($x_{\text{left}}, \boldsymbol{x}_{\text{dual}}$, and x_{right}).

Find the normal vector m to the decision boundary. One final piece of information is necessary that we will use in several places: the normal vector to the decision boundary at either side x_{left} and x_{right}. Formally, the normal vector to the decision boundary m is a vector such that, for any input x on the decision boundary and any random vector dx perpendicular to m, the point $x + dx$ also lies on the boundary. By interpreting the normal vector this way, we can determine its value through a series of queries.

To begin, we choose (arbitrarily and without loss of generality), two unit-length and orthogonal vectors e_0 and e_1. We then choose a small constant α, and step in the direction $x' = x + \alpha e_0$. Then, via binary search, we find the smallest value of β such that $x^* = x' + \beta e_1$ lies on the decision boundary. Once that we have the value of β, we compute $\frac{\beta}{\alpha}$, which gives the ratio between the e_0 and e_1 coordinates of the normal vector (up to sign and constant). By repeating this procedure for each of the remaining vectors $e_2, e_3, \ldots, e_{d_0}$ of an orthonormal basis, we can completely reconstruct (up to scale) the normal vector m to the decision boundary.

5 Signature Recovery

The first step in our attack recovers the parameters of each layer of the model up to a real-value scalar per neuron. Formally, the i-th fully connected layer of the neural network is $f_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$, which is parameterized by the weight matrix $A^{(i)} \in \mathbb{R}^{d_i \times d_{i-1}}$ and the bias vector $b^{(i)} \in \mathbb{R}^{d_i}$ (cf. Definition 2). In what follows we will ignore biases for ease of presentation, noting that they can be treated as an additional column in the weight matrix that acts on a virtual neuron of constant value 1.

Definition 12. *The* signature *of neuron j in layer i is equal to $\alpha \cdot A^{(i)}_j$, where α is an arbitrary scaling factor and $A^{(i)}_j$ is the j-th row of the layer's weight matrix.*

It is easy to verify that positive rescalings can be done without affecting the output (specifically: if all the weights and the bias of one neuron are multiplied by a constant $a > 0$, and the corresponding outgoing weights, the matching column of the next layer's weight matrix, are multiplied by $\frac{1}{a}$, then the model will behave identically). However, rescaling by a negative constant does change the output, since this causes neurons to flip from active to inactive and flips the behavior of the ReLUs. Therefore, our signature recovery attack mirrors the methodology of prior work [7]: this section covers the recovery of the weights up to a scaling factor, and the next section will focus on recovering only the sign of this factor.

5.1 Warm-up: Extracting the first layer

The previous section developed an algorithm that allows us to efficiently identify a vast set of dual points. We now show how to use this information to recover the normal vectors (up to sign and magnitude) to the neurons' critical hyperplanes — and, as a consequence, the signatures, starting with neurons in the first layer of the neural network.

To begin, assume that we are given a dual (and thus critical) point $\boldsymbol{x}_{\text{dual}}$ for some neuron η. In prior work [7], computing the normal vector to the critical hyperplane induced by the neuron η was possible directly via small shifts, because they had the power to query the model f at arbitrary points $x \in \mathbb{R}^{d_0}$ and observe the change in output. In the hard-label scenario, the only useful points our attack can sense are constrained to a piecewise linear $d_0 - 1$-dimensional subspace—the set of points on the decision boundary of the neural network, and thus we must implement a slightly different attack, formalized in Algorithm 1.

Step 1: Compute the $(d_0 - 2)$-dimensional dual space in the vicinity of $\boldsymbol{x}_{dual}$.

Definition 13. *The* dual space *D^η of a neuron η is the locally linear $(d_0 - 2)$-dimensional subspace which is the intersection between the $d_0 - 1$-dimensional critical hyperplane for η and the $(d_0 - 1)$-dimensional decision boundary in the vicinity of $\boldsymbol{x}_{dual}$.*

This definition at first may not appear very useful: we do not yet know the $(d_0 - 1)$-dimensional critical hyperplane, and so how do we compute the intersection between this hyperplane and the decision boundary? But now let us make the following observation: the dual space for a neuron η is also equal to the intersection of the two decision boundary hyperplanes around the neuron, specifically, the ones at the points x_{left} and x_{right}. As a result, it is possible to compute the dual space (which has a strong dependence on the parameters of the model) by making use of label-only oracle queries to the model.

Step 2: Combine two dual spaces to recover the lost dimension.[11] Suppose for the moment (and we will show how to do this next) that we were given two different dual points, and their corresponding dual spaces D_0^η and D_1^η for the same neuron. Consider the quantity $A_j^{(1)}$: the j-th row of the first layer weight matrix; put differently, this is the normal vector to the critical hyperplane. Notice that each of these dual spaces satisfies $A_j^{(1)} \notin D_0$ and $A_j^{(1)} \notin D_1$ (because, by definition, the $d_0 - 1$-dimensional critical hyperplane contains the dual space, and therefore cannot contain the normal vector to that hyperplane).

But also notice that, unless we are exceptionally unlucky, $D_0^\eta \neq D_1^\eta$. Therefore, by simply finding the subspace that contains both of these two subspaces, we can recover a $d_0 - 1$-dimensional space parameterized by its normal vector n. And therefore, $A_j^{(1)} = n \cdot \alpha$ for some real-valued (and possibly negative) constant α.

Filtering first-layer dual points. The final missing piece to our puzzle is to give an algorithm that filters out the dual points that correspond to neurons on the first layer from dual points that correspond to neurons on later layers. We detail this algorithm in Algorithm 2, and describe it here.[12]

Fortunately, this is straightforward and does not require any new work. Suppose that we have two dual spaces D_i and D_j and want to know if they correspond to the same neuron on the first layer. Let us simply compute

$$E \;=\; \text{span}(\text{basis}(D_i) \cup \text{basis}(D_j))$$

as the subspace that contains these two $(d_0 - 2)$-dimensional spaces. We already know that E is $(d_0 - 1)$-dimensional when both dual points correspond to the same neuron on the first layer. Let us now consider what happens if D_i and D_j correspond to different neurons on the first layer, and after that, any neurons on later layers.

[11] A simple 3D example of this process is to find a 2D plane from some 1D lines it contains. A single line does not fully define the plane, since it can rotate around it, but two different lines in it uniquely define the plane. Note that if the two lines are arbitrary lines in 3D space, they are extremely unlikely to belong to any common 2D plane, so we can cluster consistent lines that belong to the same neuronal plane.

[12] The IsConsistent procedure used in Step 10 is discussed below and formalized in Algorithm 3.

Algorithm 1 RecoverFirstLayerWeights(x_{dual_0}, x_{dual_1})

Input: x_{dual_0}, x_{dual_1}, dual points of the model
Output: The parameters of the model $\alpha \cdot A^{(1)}_\eta$, where η is one of the neurons in the first layer, or $\perp$ if the dual points are inconsistent.

$D_0 \leftarrow$CompDualSpace(x_{dual_0})
$D_1 \leftarrow$CompDualSpace(x_{dual_1})
$D \leftarrow D_0 \cup D_1$
if $D = \mathbb{R}^{d_0}$ **then**
    return $\perp$
else
    Find w such that $\text{Span}(\{w\})^\perp = D$
    return w

Algorithm 2 CollectFirstLayerDualPoints(K)

Input: K, number of dual points to search for
Output: Set of clusters of consistent dual points

$S \leftarrow \emptyset$
for $i \leftarrow 1$ to K **do**
    $\boldsymbol{x}_{\text{dual}} \leftarrow$ FindRandomDualPoint()
    $S \leftarrow S \cup \{\boldsymbol{x}_{\text{dual}}\}$
clusters $\leftarrow \emptyset$
for each $\boldsymbol{x}_{\text{dual}} \in S$ **do**
    matched $\leftarrow$ **false**
    for each $c \in$ clusters **do**
        $r \leftarrow$ RandomElement(c)
        if IsConsistent($r, \boldsymbol{x}_{\text{dual}}$) **then**
            $c \leftarrow c \cup \{\boldsymbol{x}_{\text{dual}}\}$
            matched $\leftarrow$ **true**
            break
    if not matched **then**
        clusters $\leftarrow$ clusters $\cup \{\{\boldsymbol{x}_{\text{dual}}\}\}$
return $\{c \in \text{clusters} : |c| > 1\}$

If D_i and D_j both correspond to different neurons on the first layer, then we will have that $D_i^\perp = \text{span}(\{A_i^{(1)}, r_i\})$ where r_i acts as an arbitrary, essentially random vector (determined by later layers); similarly $D_j^\perp = \text{span}(\{A_j^{(1)}, r_j\})$. Because we assume that no two critical hyperplanes are exactly parallel, we expect $A_i^{(1)}, r_i, A_j^{(1)}, r_j$ to all be linearly independent, and therefore the smallest subspace that contains both D_i and D_j is $\mathbb{R}^{d_0}$. This allows us to distinguish dual points that share a neuron from those that do not.

5.2 Extracting deeper layers

Having shown how to recover the first layer, we now present our general algorithm. Without loss of generality, we assume that all previous layers of the network are extracted, and our goal is to extract the i-th layer.

Definition 14. *The function that computes the first i layers (up to and including i) of f is called the input function and is denoted as $f_{1..i}$. In particular, $f = f_{1..r+1}$.*

Because we have extracted the parameters of the model up to layer $i-1$, we can easily filter out and remove any dual points that belong to layers 1 through $i-1$ by taking every dual point $\boldsymbol{x}_{\text{dual}}$ and computing $f_{1..j}(\boldsymbol{x}_{\text{dual}})$ for every $j < i$ and rejecting any dual point where there exists some layer j and neuron η such that $f_{1..j}(\boldsymbol{x}_{\text{dual}})$ is at a critical point.

Identifying layer-i dual points To start, we consider all dual points that have not already been found to be part of some prior layer. Now, we repeat an algorithm very similar to the algorithm from Section 5.1.

Definition 15. *Let $f_{x_0;i}$ be an affine transformation that satisfies the conditions $f_{x_0;i}(x_0) = f_i(x_0)$ and $\nabla f_{x_0;i}(x)\big|_{x=x_0} = \nabla f_i(x)\big|_{x=x_0}$.*

That is, $f_{x_0;i}$ is just the linear transformation that the function f_i defines within the linear region around the point x_0.

Definition 16. *The hidden state at layer j is the output of the function $f_{1..j}$, right before applying the nonlinear transformation σ.*

For each dual point $\{\boldsymbol{x}_{\text{dual}i}\}$ and corresponding normal vector $\{n_i\}$, we compute the hidden state $\hat{\boldsymbol{x}}_{\text{dual}i} = f_{1..i}(\boldsymbol{x}_{\text{dual}i})$ and normal vector $\hat{n}_i = f_{\boldsymbol{x}_{\text{dual}i};i}(n_i)$.[13]

By performing this linear transformation, we can now pretend that we are working from within the space of the i-th hidden layer. By doing this, it is now possible to check if two dual points are *consistent*.

Definition 17. *Two dual points $\boldsymbol{x}_{dual0}$ and $\boldsymbol{x}_{dual1}$ are* consistent *if they occur because of the same neuron η on the same layer i of the model.*

To check consistency, we leverage the fact that the union of the dual spaces for dual points that share the same neuron will not have full rank, whereas the dual spaces for dual points from different neurons will have full rank.

This allows us to repeat our prior algorithm identically; with one key challenge. Because we are working with ReLU networks, all inputs that are negative will be set to zero. Therefore, the linear transformation defined by $f_{x_0;i}(x_0)$ will not be *onto*—in fact, it will have a null space roughly equal to $d_i/2$ [5].

Definition 18. *A* partial dual space $D_{partial}$ *is a subspace of the dual space D.*

What this means practically speaking is that when we compute the union of two dual spaces, even if they correspond to completely different neurons, they will (with high probability) share some coordinates which are identically zero. Therefore, the correct measure to check if two dual points are consistent is not if they have completely full rank, but rather whether they achieve the highest rank possible, given the number of shared coordinates where both dual points have dead neurons. [14]

Recovering layer-i signatures Given a method to identify whether or not two layer-i dual points correspond to the same neuron, we can now finally recover the signatures for this layer of the model.

[13] Notice the reason we are using this function is because n_i is not an input to the neural network, but we just want to appropriately transform the normal vector.

[14] We define "dead neurons" as those whose ReLU activations exhibit minimal sign change (i.e., they are effectively always off or always on). Following [15], the implications of the latter case (almost always on/persistent neurons) are analyzed in detail in Section 9.2.

Algorithm 3 IsConsistent($\boldsymbol{x}_{\text{dual}0}, \boldsymbol{x}_{\text{dual}1}, D_0, D_1$)

Input: $\boldsymbol{x}_{\text{dual}j}$ a dual point of the model, D_j the corresponding dual space for $j = 0, 1$.
Output: True if the two dual points are on the same neuron's critical hyperplane, otherwise False.

$\tilde{D}_j \leftarrow f(\boldsymbol{x}_{\text{dual}j}; i)(D_j)$
$D \leftarrow \tilde{D}_0 \cup \tilde{D}_1$.
$X \leftarrow$ the number of neurons active in either $\boldsymbol{x}_{\text{dual}0}$ or $\boldsymbol{x}_{\text{dual}1}$.
if Rank$(D) == X$ **then**
    return False
else
    return True ▷ Specifically, $Rank(D) < X$

Step 1: cluster dual points. Given all of the dual points we have found thus far, we first cluster them together using the IsConsistent function from Algorithm 3. It is straightforward to implement an $O(M^2)$-time algorithm for M dual points, to achieve this. We begin by constructing a graph; each dual point corresponds to a node, and we connect two dual points with an edge if they are consistent.

In an ideal world, this graph would be disconnected and have exactly d_i cliques of size greater than one, and all remaining nodes isolated from each other (corresponding to neurons on deeper layers). In practice, however, due to numerical instabilities the IsConsistent algorithm may, with small probability (e.g., 10^{-6}), spuriously claim that two neurons are consistent when in fact they are not. To resolve this issue, we instead search for a *maximal* clique, because we find that spurious errors are typically independent; this way, single edges that connect to cliques are automatically discarded.

Step 2: unify dual spaces to recover signatures. At this point in our algorithm we have one set of dual points per neuron on the i-th layer of the model. All that remains now is to recover the parameters for each of these neurons.

Fortunately, our layer-1 attack in Section 5.1 can be re-used almost exactly. Only now, instead of collecting a pair of $(d_0 - 2)$-dimensional dual spaces and unifying them together to obtain a $(d_0 - 1)$-dimensional critical hyperplane, we now unify a larger set $\Omega(\log(d_i))$ dual points together.

The reason why we need a larger number of dual points per neuron is that a small constant fraction of the neurons in any given layer will be dead [5]. And so, in order to observe at least one non-dead input in each of the d_i positions, we should expect to need to see $\Omega(\log(d_i))$ different examples assuming a perfectly uniform distribution of active and inactive neurons. If we collect our set of dual points and find that they do not have sufficient diversity, then we simply return back to the first step of our attack and collect more dual points. A complete description of the algorithm for this attack is given in Algorithm 4.

Algorithm 4 RecoverDeeperWeights($\{\boldsymbol{x}_{\text{dual}\,j}\}_{j=1}^{n}, \{D_j\}_{j=1}^{n}$)

Input: A list of dual points $(\{\boldsymbol{x}_{\text{dual}\,j}\}_{j=1}^{n}$ for a target neuron η in layer i, and a list of corresponding dual spaces $\{D_j\}_{j=1}^{n}$).

Output: The parameters of the model $\alpha \cdot A_{\eta}^{(i)}$ for some scalar α.

assert all dual points $\boldsymbol{x}_{\text{dual}\,j}$ are (pairwise) consistent.
$\tilde{D}_j \leftarrow F_{\boldsymbol{x}_{\text{dual}\,j}}^{(i-1)}(D_j)$
$D \leftarrow \tilde{D}_1 \cup \tilde{D}_2 \cup \tilde{D}_3 \cup \cdots \cup \tilde{D}_n$
if $\text{Rank}(D) < d_0 - 1$ **then**
    return $\perp$ ▷ Insufficient data to reconstruct parameters.
else
    Find w such that $\text{Span}(\{w\})^{\perp} = D$
    return w

6 Sign Recovery

Recall that a neuron's signature is related to its actual weights by some arbitrary scaling factor α. Finding the correct *sign* of α is crucial in order to correctly recover the model. Similarly to [5], we use a heuristic argument and develop a statistical test that is likely to identify the correct sign. We begin by providing the intuition behind our sign recovery technique and experimentally validating its key assumption (Section 6.1). Next, we present the technique in algorithmic form (Section 6.2).

6.1 The Basic Technique

To correctly determine the sign of a target neuron, we require a property that differs measurably between the neuron's on- and off-sides.

In [5] the authors demonstrated that a small perturbation (*neuron wiggle*) in a carefully chosen direction is expected to change the target neuron's output by ϵ, while affecting all other neurons in the same layer by about $\pm\frac{\epsilon}{\sqrt{d}}$, where d is the target layer's width. Consequently, the output norms of the target layer, $||v_{\text{on}}||$ and $||v_{\text{off}}||$, differ between the on- and off-sides of the target neuron. Assuming that the neurons in the remaining layers behave randomly, the changes in the floating point values of the output logits also differ in magnitude between the on- and off-side. This approach allowed the authors of [5] to directly measure these infinitesimal changes in the output logits of the neural network.

One key intuition for sign recovery without direct access to output logits is that the average speed with which *all* future neurons[15] change their values depends on the norm of the change at the target layer. Accordingly, the speed with which the average future neuron changes its value is larger on the target neuron's on side than on its off side (see Figure 6).

[15] A *future neuron* means a neuron that is located in any of the layers $f_{i+1}, \ldots, f_r$ that *follow* the current target layer i.

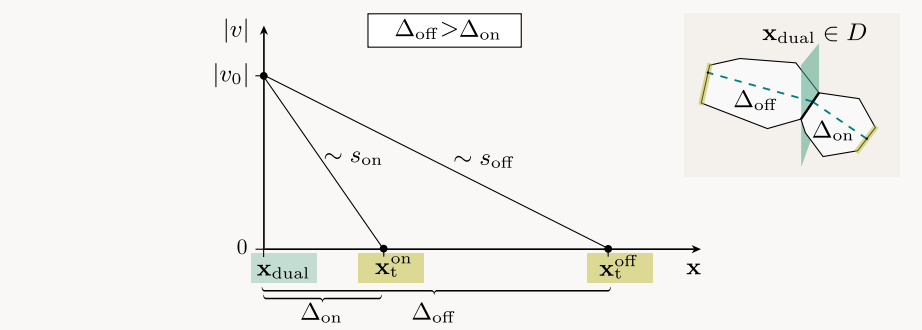


Fig. 6: Assume a target neuron at a dual point $\boldsymbol{x}_{\text{dual}}$, and a future neuron at initial value v_0. A higher average speed on the on-side s_{on} compared to the off-side s_{off} will result in a measurable difference $\Delta_{\text{off}} > \Delta_{\text{on}}$.

More precisely, for a fixed walking direction dx we define the speed of change of a given neuron η in a hidden layer $i = 1, \ldots, r$ as

$$s_\eta := \frac{|\mathcal{V}(\eta; x + dx) - \mathcal{V}(\eta; x)|}{\|dx\|}.$$

The core assumption is that the speed of a future layer neuron correlates with the rate of change in the target layer's output norm, leading to the approximate magnitude estimates given in Equation (1) (when the target neuron is on) and Equation (2) (when the target neuron is off).

$$s_{\text{on}} \propto ||v_{\text{on}}|| \approx \| \left(\pm\tfrac{\epsilon}{\sqrt{d_i}}, \pm\tfrac{\epsilon}{\sqrt{d_i}}, \ldots, \epsilon, \ldots, \pm\tfrac{\epsilon}{\sqrt{d_i}}, \pm\tfrac{\epsilon}{\sqrt{d_i}} \right) \| \tag{1}$$

$$s_{\text{off}} \propto ||v_{\text{off}}|| \approx \| \left(\pm\tfrac{\epsilon}{\sqrt{d_i}}, \pm\tfrac{\epsilon}{\sqrt{d_i}}, \ldots, 0, \ldots, \pm\tfrac{\epsilon}{\sqrt{d_i}}, \pm\tfrac{\epsilon}{\sqrt{d_i}} \right) \| \tag{2}$$

Walking in the $\pm dx$ direction on either side of the target neuron's critical hyperplane, we can measure the distances Δ_{on} and Δ_{off} until some other neuron toggles and the decision hyperplane bends. If a future neuron is approaching its toggling point $\boldsymbol{x}_{\text{t}}$, the higher speed on the side where the target neuron is on implies it will reach this point *earlier* than if moving at a slower speed on the off side (see Figure 6). The ratio between the two speed estimates above is about $\sqrt{2} \approx 1.4$, while actual experiments yield an average speed ratio of about 1.2. This significant ratio of distances $\Delta_{\text{off}}/\Delta_{\text{on}}$ (which does not depend on the dimension d and is expected to remain roughly the same at different layers) is easily measurable. By running a small number of experiments and taking their majority vote (i.e., assigning a minus sign to the side of its critical hyperplane which tends to produce longer distances) we are extremely likely to find the correct sign of the neuron.

An even better method is possible in the case where the local matrix $F_x^{(i-1)}$ corresponding to the previous layers is invertible (either because the network is

rapidly shrinking or because we are at the first hidden layer). We call this the case of *perfect control*, since we can always find a walking direction dx which produces a change only in the target neuron while leaving all other neurons in the layer fixed. This makes it particularly easy to recover the sign, since there will be no change at all on the side where the target neuron is suppressed by the ReLU and so $||v_{\text{on}}||/||v_{\text{off}}|| = \infty$.

Experimental Verification of the Basic Technique To experimentally verify our technique in terms of the level of control ($||v_{\text{on}}||/||v_{\text{off}}||$) and the average future neuron speed (s_{on} and s_{off}), we record these values in a white-box setting for the network described in Our CIFAR-10 Network. Details of this network will be provided in the experimental section; here, we note that it contains four hidden layers, with 256 neurons in hidden layers 1 to 3, and 64 neurons in hidden layer 4. For a large number of target neurons in every hidden layer, we examine the speed of all future neurons and the level of control during our sign recovery attack[16] across several dual points. At each dual point, the speed of each future neuron is calculated as $s_{\text{on, off}} = |\mathcal{V}(\eta; \boldsymbol{x}_{\text{dual}}) - \mathcal{V}(\eta; \boldsymbol{x}_{\text{t}})|/\Delta_{\text{on, off}}$. This represents the change in the neuron's value before its ReLU, divided by the distance $\Delta_{\text{on, off}}$ to the first future neuron toggling point on the on- or off-side.

Table 2: White-box verification of the intuition behind the sign attack across target neurons in Our CIFAR-10 Network, with n_{ID} denoting the ID of a worst target neuron.

		Average target neuron		Worst target neuron		
Layer	Neurons	$\|\|v_{\text{on}}\|\|/\|\|v_{\text{off}}\|\|$	$s_{\text{on}} > s_{\text{off}}$	n_{ID}	$\|\|v_{\text{on}}\|\|/\|\|v_{\text{off}}\|\|$	$s_{\text{on}} > s_{\text{off}}$
1	256	∞	100%	-	-	-
2	256	(1.20 ± 0.04)	$(96 \pm 5)\%$	170	1.07	64%
3	256	(1.21 ± 0.04)	$(82 \pm 7)\%$	226	1.19	59%
4	64	(3.2 ± 0.8)	100%	-	-	-

Table 2 shows that the intuition on the future neuron speed being larger on the on- than the off-side of the neurons is true for all the $(3 \times 256 + 64 = 832)$ target neurons in the network. The average ratio is about 1.2, and even for the worst neurons the ratio is above 1.07 (which will simply require a larger number of corresponding dual points to get a reliable result).

6.2 Algorithm

Let $\boldsymbol{x}_{\text{dual}}$ be a dual point for the target neuron. Assuming momentarily that we do know the correct sign of the weights, we let $\hat{\boldsymbol{n}}$ be the unit normal vector of the ReLU hyperplane pointing towards the direction where the neuron is positive. Walking in the direction of $\hat{\boldsymbol{n}}$ will produce a maximal rate of change in the

[16] The construction details of the neuron wiggle for the attack are discussed in Section 6.2.

target neuron, which in turn may toggle neurons in future layers more often. On the other hand, walking in the direction of $-\hat{\boldsymbol{n}}$ will produce no change in the target neuron since it is being suppressed by the ReLU, and may be less likely to toggle neurons in future layers. Of course, this general rule may not hold depending on the unpredictable effect that the displacements have on non-target neurons. However, when conducting many experiments that explore dual points throughout different regions of the input space, we expect that walks on regions where the target neuron is known to be on, will on average trigger future neurons faster than those where the target neuron is off. Based on the above argument, we devise Algorithm 5: we assume that we know the correct sign of the neuron, and measure the distance that we need to walk on either side of the ReLU before a future-layer neuron is toggled. The measurement of the distances is implemented by Algorithm 6, which follows the description in the previous subsection. These distances are compared, and the experiment is repeated at many different dual points. If our guess for the sign was correct, we expect that a majority of experiments will walk a shorter distance on the on-side than on the off-side. Otherwise, we conclude that the real sign is opposite to our guess.

Algorithm 5 RecoverSign$(i, \mathbf{W}, \mathbf{B})$

Input: i the index of the target neuron and $\mathbf{W}, \mathbf{B}$ the weights and biases of the network up to and including the target layer (with the target layer known only up to sign per neuron).
Output: $+1$ if the sign of the target neuron is correct, otherwise -1

$\texttt{votes}_+ \leftarrow 0$
$\texttt{votes}_- \leftarrow 0$
for $\texttt{experiment} = 1, \ldots, \texttt{N}_{exp}$ **do**
  $\boldsymbol{x}_{\text{dual}} \leftarrow$ RandomDualPoint(i)
  $F \leftarrow$ LocalMatrix$(\mathbf{W}, \mathbf{B}, \boldsymbol{x}_{\text{dual}})$
  $\hat{\boldsymbol{n}} \leftarrow F[:, i] \;/\; \|F[:, i]\|$
  $\Delta_+ \leftarrow$ DistanceToToggle$(\boldsymbol{x}_{\text{dual}}, \hat{\boldsymbol{n}})$
  $\Delta_- \leftarrow$ DistanceToToggle$(\boldsymbol{x}_{\text{dual}}, -\hat{\boldsymbol{n}})$
  if $\Delta_+ < \Delta_-$ **:** $\texttt{votes}_+ = \texttt{votes}_+ + 1$
  else : $\texttt{votes}_- = \texttt{votes}_- + 1$
if $\texttt{votes}_+ > \texttt{votes}_-$ **: return** $+1$
else: return -1

The precise way in which we perform our walk is visualized in Figure 7 (and detailed in Algorithm 6). In a black-box approach we cannot observe the values of future-layer neurons, but we know that the decision hyperplane will change directions whenever any neuron toggles.
Therefore, we perform our walk over the decision hyperplane (moving not in the direction of $\pm\hat{\boldsymbol{n}}$ but its projection onto the decision hyperplane), and infer that there has been a neuron toggle whenever we notice a change in the direction of the hyperplane.
It is important to stress that the toggle should only be counted if it comes from

Algorithm 6 DistanceToToggle($\boldsymbol{x}_{\text{dual}}, \hat{\boldsymbol{n}}$)

Input: A dual point $\boldsymbol{x}_{\text{dual}}$ for the target neuron, and $\hat{\boldsymbol{n}}$ the normal unit vector of the neuron's ReLU hyperplane. Also uses parameters `nToggles` with default value 1, and δ (a small number).

Output: Walks in the direction of $\hat{\boldsymbol{n}}$ projected to the decision hyperplane until `nToggles` neurons from future layers are toggled, and returns the total distance walked along the decision hyperplane in the input space.

$\Delta \leftarrow 0$
$\boldsymbol{x}_0 \leftarrow \boldsymbol{x}_{\text{dual}}$
$\mathbf{dx} \leftarrow \hat{\boldsymbol{n}}$
`futureLayerToggles` $\leftarrow 0$
while `futureLayerToggles` $<$ `nToggles` **do**
    $\hat{\boldsymbol{m}} \leftarrow$ DecisionPlaneUnitVector($\mathbf{x}_0 + \delta \cdot \mathbf{dx}$)
    $\mathbf{dx} \leftarrow \hat{\boldsymbol{n}} - \langle \hat{\boldsymbol{n}}, \hat{\boldsymbol{m}} \rangle \hat{\boldsymbol{m}}$ ▷ Project $\hat{\boldsymbol{n}}$ onto the decision hyperplane
    Walk from $\boldsymbol{x}_0$ in the direction $\mathbf{dx}$ and let $\boldsymbol{x}_1$ be the point at which the
    ... direction of the decision boundary first changes.
    if no past- or current-layer neuron being toggled between $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ **then**
        `futureLayerToggles` $\leftarrow$ `futureLayerToggles` $+ 1$
    else
        either discard the dual point or handle the past- or current-layer toggle
    $\Delta \leftarrow \Delta + ||\boldsymbol{x}_1 - \boldsymbol{x}_0||$
    $\mathbf{x}_0 \leftarrow \mathbf{x}_1$
return Δ

a future layer, since toggles in past layers are unaffected by whether the target neuron is on or off. We can easily check if the toggle is from a past layer, since the weights and biases of all past-layer neurons are already known.

The last hidden layer requires a separate treatment, since no further ReLU layers (and thus no toggles) exist; only the output layer follows. Here, the output logits, though inaccessible in the hard-label scenario, change with the target neuron's output.

The current decision boundary eventually intersects with another, allowing us to evaluate this crossing distance, similar to assessing distances to future toggles.

Once we have detected a non-future layer toggle we have two options: Either we decide to discard the dual point altogether (Figure 7a), or, we try to follow the decision hyperplane *through* the unwanted toggle and continue our search for a future toggle (Figure 7b).

In our experiments we choose this middle variant which aims to handle non-future layer toggles, and measures the distance Δ at each input point $\boldsymbol{x}_{\text{dual}}$ by summing the entire walked distance along the decision hyperplane until one future toggle is encountered.

The handling of non-future layer toggles does not always succeed, and we still discard the dual point if we keep on encountering the same past layer neurons. Therefore, to successfully perform N_{exp} experiments we usually need to inspect $N_{\text{dual}} > N_{\text{exp}}$ dual points.

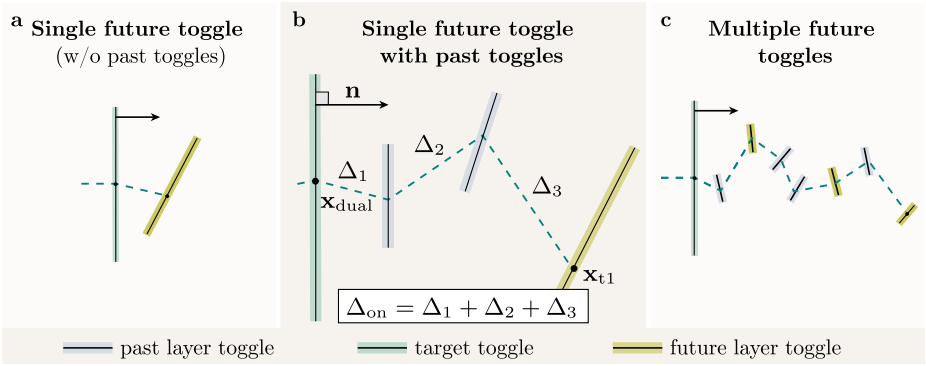


Fig. 7: Visualization of the distance measurement using the DISTANCETOTOGGLE algorithm. **a.** In its simplest variant the distance measurement only accepts distance measurements from dual points at which the first toggle is a future layer toggle. **b.** A slightly more elaborate variant of the distance measurement handles past layer toggles by recomputing the decision hyperplane normal vector and continuing to move until the first future layer toggle is encountered. **c.** In its most elaborate variant the distance measurement can gain statistics at a single dual point, by moving through multiple (instead of only a single) future toggle.

Confidence Level In our experiment, after conducting N_{exp} trials (or, in other words, successfully analyzing N_{exp} dual points), we calculate the observed probability for one sign decision (either + or -) as $p_{\text{exp}} = \frac{\max(\text{votes}_+, \text{votes}_-)}{N_{\text{exp}}}$. While the average future neuron speed s_{on} is typically higher on the on-side, this is not consistent across every single dual point (refer to Table 2). Thus, we might observe a vote for the correct side with a probability such as $p_{\text{exp}} = 0.55$. To assess whether our observed probability p_{exp} is significantly different from $p_0 = 0.5$, we compute the probability of error, that is the significance level α. The significance level is connected to the confidence level CL by $\text{CL} = 100 \times (1 - \alpha)\%$. Hoeffding's inequality [14] provides a bound on the error probability α, stating: $\alpha = P\left(p_{\text{exp}} - p_0 > \delta_{\text{p}}\right) \leq \exp\left(-2\delta_{\text{p}}^2 N_{\text{exp}}\right)$, where $\delta_p = p_{\text{exp}} - p_0$. This implies that the number of trials N_{exp} required to achieve a specified error probability α is $N_{\text{exp}} \geq \ln(1/\alpha)/(2\delta_{\text{p}}^2)$. As a practical example, at least $N_{\text{exp}} \gtrapprox 1000$ trials are required to achieve a confidence level of $\text{CL} = 99\%$ for $p_{\text{exp}} = 0.55$. If we don't find p_{exp} to be close to $p_0 = 0.5$, a much smaller minimum number of trials can suffice, such as $N_{\text{exp}} \gtrapprox 100$ (for $p_{\text{exp}} = 0.65$) or $N_{\text{exp}} \gtrapprox 10$ (for $p_{\text{exp}} = 1.0$).

7 Experiments

We validate our method on Our CIFAR-10 Network by first evaluating the signature recovery (Section 7.1), and then the sign recovery (Section 7.2). In practice, the recovery process begins with signature recovery for hidden layer 1, followed by sign recovery. Once this layer is fully recovered, the process is repeated se-

quentially for each subsequent layer until the entire network is reconstructed. For all experiments in this paper, we validate the correctness of the algorithms with an implementation. To avoid the numerical instability difficulties raised in [7], we perform all arithmetic in 64-bit floating point precision and assume that the prior layers were perfectly extracted. We stress that the experiments presented in this section do not aim to replicate a full implementation of the attack, but only to present evidence that the method is viable even for a network of this scale. In the following section, we present an implementation that is significantly closer to a fully black-box attack and test it on a smaller network. Our software implementation is available at

https://github.com/Jchavezsaab/hard-label-dnn-extraction

Our CIFAR-10 Network CIFAR-10 is a widely used benchmark dataset in the field of visual deep learning. It is a balanced subset of the larger 80 Million Tiny Images dataset, containing ten classes, including categories like airplanes, cats, and frogs [25]. Each class consists of images with RGB channels of size 32×32 pixels, resulting in 3,072 input values per image. The dataset comprises a total of 50,000 training images and 10,000 test images.

Our model for CIFAR-10 employs 3,072 input neurons (one per pixel) and consists of four densely connected hidden layers. The first three hidden layers contain 256 neurons each, while the fourth layer contains 64 neurons, all using ReLU activations. The network's output layer has 10 neurons with softmax activation. We follow the same training procedure[17] as [5, 17].

Our four-hidden-layer model achieves a test accuracy of 52% on CIFAR-10, which aligns with the expected performance for densely connected neural networks utilizing pure ReLU activation functions, typically around 0.53 [17]. It is important to note that higher test accuracies are attainable with more sophisticated neural network architectures. For example, Google Research's Vision Transformer (ViT-H/14) currently achieves a state-of-the-art test accuracy of 99.5% on CIFAR-10 [10, Table 2].

7.1 Signature Recovery

We now have each of the major algorithms necessary to recover the signatures of the model.

Simplifications. The total runtime for the signature recoveries of each target layer can be expressed as

$$t_{\text{signatures}} = t_{\text{dual}} + t_{\text{cluster}} + t_{\text{unify}}, \tag{3}$$

[17] For training, we use standard preprocessing and optimization techniques. The pixel values are rescaled from the original range of 0 to 255 to a range of 0 to 1. The model is trained using stochastic gradient descent (SGD) with a momentum of 0.9 and sparse categorical cross-entropy as the loss function. A batch size of 64 is used during training.

where t_{dual} is the time taken to find dual points, t_{cluster} is the time taken to cluster the dual points (cf. step 1 in Section 5.2), i.e. identify which target neuron in the target layer they belong to, and t_{unify} the time taken to unify the corresponding dual spaces (cf. step 2 in Section 5.2).

Our proof-of-concept implementation disregards the runtime of the n^2 pairwise clustering t_{cluster} needed to identify dual points that are mutually consistent. Although this method is known to work in principle (we validated it post-hoc) and clearly runs in polynomial time, applying it naively (i.e., without many potential optimizations) to a real attack would involve approximately 2^{20} pairwise tests, demanding weeks of computation. Further, efficiency is improved by computing the gradient symbolically at decision boundary points, instead of relying on binary search, thereby enhancing performance by a constant factor, and achieving a shorter t_{dual}.

Consistency Verification. Recall that, after generating dual points, the first step in our attack is to cluster dual points according to whether or not they are *consistent*. To implement this in a numerically stable manner, we collect a set of inputs that form a basis for each of the two partial dual spaces, and then compute the singular value decomposition on this set of points. By inspecting the smallest singular value, we can see if the dual spaces are consistent, because consistent dual spaces will not span $\mathbb{R}^{d_0}$ whereas inconsistent ones will. As we can see in Figure 10, running the algorithm on dual points that come from the same neuron result in a significantly different distribution of singular values than when the algorithm is run on dual points from different neurons.

In a very small number of cases, we find that pairs from different neurons can yield small singular values. To address this rare occurrence, we apply a secondary filter where we consider triples of dual points (a, b, c) that are believed to be consistent with each other; if the triple is inconsistent, we discard all three. Empirically we observe that this completely eliminates all false positives.

Identifying a diverse set of dual points. In order for our algorithm to recover neurons at deeper layers, we must collect a *diverse* set of inputs that cause every input to the neuron to be active at least once. Unfortunately, we find that some neurons in the model we have trained are *almost dead*, and rarely activate. As a result, it is necessary to identify a very large number of dual points in order to reconstruct these last few parameters.

Figure 11 summarizes this analysis. The number of dual points required for extraction increases with each layer because it becomes progressively harder to activate specific neurons in the deeper layers.

Measuring Extraction Fidelity. As mentioned earlier, throughout this paper we describe an algorithm that succeeds as long as arithmetic is performed to an arbitrary level of precision. However, in our implementation we make use of 64-bit floating point arithmetic. In this section we validate that our attack correctly

recovers the weights of the model up to a high degree of precision, assuming that all prior layers have been perfectly recovered. As we can see in Figure 12, all neurons are extracted with extremely high fidelity—usually 18 or more bits of precision.

This proof-of-concept assessment of the signature-recovery algorithm (neglecting t_{cluster}, and with an improved t_{∇}) has been successfully implemented, and runs in approximately 16 hours on a 256-core machine with additional GPU support.

7.2 Sign Recovery

We will first describe assumptions and possible further optimizations of our current implementation of the sign recovery attack, and then conclude with the obtained results.

Description of the Implementation The total runtime for the sign recovery of each target neuron can be expressed as:

$$t_{\text{sign}} = (t_{\text{dual}} + t_m + t_{\text{vote}}) \times N_{\text{dual}}, \tag{4}$$

where t_{dual} is the time needed to identify a single dual point for the target neuron, t_m is the time needed to determine the decision hyperplane normal vectors ($\boldsymbol{m}_{\text{on}}$, $\boldsymbol{m}_{\text{off}}$), t_{vote} is the actual time needed to execute the attack on the target neuron and obtain a single vote for sign recovery, and N_{dual} is the total number of dual points required to attain the desired confidence level (cf. Section 6.2) for the attacked neuron.

In our current implementation, we assume the dual points and decision hyperplane vectors are predetermined, as these components are common to both sign recovery and signature recovery. Notably, here we use as inputs critical points for the target neuron derived from a set of random uniformly distributed points over the interval $[-5, +5]^{d_0}$. Additionally, the measurement of the distances $\Delta_{\text{on}}, \Delta_{\text{off}}$ until a future-layer neuron toggles is done by looking at the parameters of future layers in a white-box manner, rather than performing black-box queries to observe where the decision hyperplane bends. This trick is used for simplicity, and is eliminated in the implementation described in the next section.

Another potential optimization involves adaptive control of the number of dual points investigated. Currently, we use a fixed minimum number of input points ($N_{\text{exp}}^{\text{min}} = 100$ for hidden layers 1 and 4, and $N_{\text{exp}}^{\text{min}} = 1,000$ for hidden layers 2 and 3 as detailed in Section 6.2). This fixed minimum is often higher than necessary for many of the targeted neurons.

Results We use the confidence level CL introduced in Section 6.2 to monitor the sign recovery. Figure 13 shows examples for the evolution of CL with the number of investigated dual points N_{dual} in hidden layers 1, ..., 4. In hidden layer 1 the likelihood of encountering current layer toggles is very low, allowing for a

successful investigation of every dual point. Further, we have perfect control over the target layer, i.e. $p_{\text{exp}} = 100\%$. Consequently, after analyzing approximately $N_{\text{exp}} = N_{\text{dual}} \approx 10$ points, we achieve a confidence level of CL $\approx 100\%$ for all neurons.
Similarly, in hidden layer 4, we expect strong control over the target layer due to the drop in the number of neurons. However, this layer is characterized by a high likelihood of encountering current or past layer toggles, which results in variability in the number of dual points $N_{\text{dual}} \approx 876$ required to attain the final confidence level (see Table 3 for more details).
In hidden layers 2 and 3, the behavior of the confidence level with respect to the number of dual points shows greater variability. While the easiest neurons achieve confidence levels of around 100% after approximately $N_{\text{dual}} \approx 250$, average neurons require about $N_{\text{dual}} \approx 750$ dual points to reach similarly high confidence levels (cf. Figure 13). Notably, there are six and seven particularly challenging neurons in layers 2 and 3, respectively, that maintain $CL \leq 75\%$ even after extensive analysis. For these neurons we re-run the sign recovery on a fresh set of input points and can indeed achieve higher confidence levels.

Table 3: Summary of sign recovery results on our CIFAR-10 network.

			Confidence level			Times per neuron	
Layer	**Recovered**	N_{dual}	min(CL)	mean(CL)	max(CL)	$\bar{t}_{\text{vote}}$	$\sum t_{\text{vote}}$
1	**256**/256	(100±0)	100%	100%	100%	(0.71 ± 0.13) s	(220 ± 127) s
2	**256**/256*	(1142 ± 268)	86.27%	99.39%	100.00%	(0.29 ± 0.07) s	(1670±1131) s
3	**256**/256*	(1275±193)	88.06%	99.48%	100.00%	(2.16±0.43) s	(3468±485) s
4	**64**/64	(876±377)	98.17%	99.63%	99.69%	(1.48±0.39) s	(2311±661) s

N_{dual}: The average number of the dual points across all neurons. $\bar{t}_{\text{vote}}$: The average of the runtime per dual point. $\sum t_{\text{vote}}$: The average of the total runtime per neuron.
* In layer 2 and 3, six and seven neurons, respectively, finalized the first run with CL $< 75\%$ and were re-run a second time to confirm the sign votes and achieve higher confidence levels.

The sign recovery process for all neurons across all layers yields correct results. Table 3 summarizes the sign recovery results. This proof-of-concept assessment of the sign-recovery algorithm indicates that given the indicated number of input points and their decision hyperplane normal vectors, i.e. $t_{\text{sign}} \approx t_{\text{vote}} \times N_{\text{dual}}$, we can run our neuron sign recovery attack on 64 neurons in parallel on our 256-core server and recover hidden layers $1, \ldots, 4$ in around 8.5 hours runtime.[18]

[18] This estimate is based on Table 3: In detail we can recover the signs of hidden layer 1 in four batches with a total runtime of about 4×220 s$\approx$ 15 minutes, hidden layer 2 in about $(4+1) \times 1,670$ s$\approx$2.5 hours (note that we added one extra-run for the initially low-confidence neurons.), hidden layer 3 in about 5 hours, and hidden layer 4 in 40 minutes.

8 End-to-end Black-box Attack

Our previous experiments relied on white-box information for efficiency purposes, but still show that it is plausible for a powerful adversary to carry out the attack on a large network. We now present a fully black-box implementation which we test on a medium-sized network to demonstrate our methods' effectiveness in a scenario closer to the real world.

Our target network consists of 32 input neurons, 3 hidden layers of size 32 followed by another of size 10, and a 10-logit output layer with a hard-label output. The model used was not trained, but had the weights of each neuron set to a random unit vector and the bias chosen so that each neuron is positive about half the time to avoid issues with neurons that do not change signs regularly.

The previous experiments had used white-box information only in three key routines: classifying and clustering the dual points by the neuron that is toggled, computing the normal vector m to the decision hyperplane, and walking along a decision hyperplane until it bends. Additionally, once we verified that a layer had been recovered with reasonable precision, we substituted its parameters by the exact white-box values to avoid error buildup across layers. We briefly describe how each of these shortcuts were removed.

Tagging dual points. We had previously used white-box information to immediately tag dual points with the neuron that they are associated with. In practice, the attack is able to tag a dual point as soon as it has progressed to the correct layer and detected enough consistent dual points for that neuron, and dual points that remain untagged can be assumed to be from future layers and saved for later. Note that sign recovery requires dual points that belong to future layers in order to work properly, but we do not need to know the specific layer to which they belong and can simply infer that a dual point belongs to a future layer when it doesn't produce a critical point on any of the already recovered signatures.

Computing the normal vector. The decision hyperplane's normal vector had previously been derived through a gradient computation of the white-box parameters, but now we do it using the process described at the end of Section 4 purely through black-box queries. As a reminder, the method works by first walking in one direction away from the hyperplane and then walking in a second direction until we get back to the hyperplane, which gives us a vector in the hyperplane. A practical difficulty arises when computing the normal vector in close vicinity to a dual point, since the starting and the ending point can end up on opposite ends of the ReLU hyperplane and thus lie on differently oriented decision hyperplanes, yielding a vector outside either hyperplane as shown in Figure 8. In a black-box scenario we have no immediate way of knowing when this has happened, so we safeguard against this in two ways. When the signature of the neuron is already known (for example, during sign recovery), we can choose the walking directions so that we are always moving away from the ReLU hyperplane and never cross it (changing the sign of the randomly-picked directions as

needed). When the signature is not known, we start the routine a small distance ε away from the dual point, and discard cases where the final vector along the decision hyperplane has norm greater than ε.

Fig. 8: (Left) Walking in two different directions until we get back to the decision hyperplane gives a vector on the plane, shown in red. (Right) If the walk crossed the ReLU plane, the resulting vector is not in either decision plane.



Walking to a decision hyperplane bend. Walking along a decision hyperplane until it bends is a routine needed both in signature recovery (to find dual points) and sign recovery (to walk from one dual point to another and measure the patch distance). Once the normal vector for a decision hyperplane at a point x has been found, we are able to compute a walking direction dx along the hyperplane and conduct a binary search to find the point where the hyperplane bends. In order to detect the bend, we use the normal vector to compute two points $x_\pm = x \pm \varepsilon m$ on opposite sides of the decision boundary, then move them both in parallel in the direction of dx as shown in Figure 9. Once the decision hyperplane bends, one of the two paths will cross the boundary shortly after, which we can readily detect through queries using a binary search with a given precision δ as shown in Algorithm 7. The constants δ, ε must be carefully tuned to be small enough so as not to significantly shift this crossing point away from the real dual point, but large enough that errors in the computation of dx do not cause the path to collide with the decision hyperplane when no bend has occurred. We expect this tuning to be mostly dependent on the arithmetic precision, and for our case of 64-bit floating point precision we found $\varepsilon = 10^{-7}$, $\delta = 10^{-14}$ to work well.

Managing error buildup across layers. The signatures produced by the clustering are accurate to 10^{-7} relative error, because dual points are found by a bisection process that locates the critical hyperplane. This process is not perfectly precise, and the recovered signature inherits this error. When the next layer is attacked behind such imperfect signatures rather than the exact ones, the error is amplified from layer to layer and after two or three layers the consistency tests begin to fail.

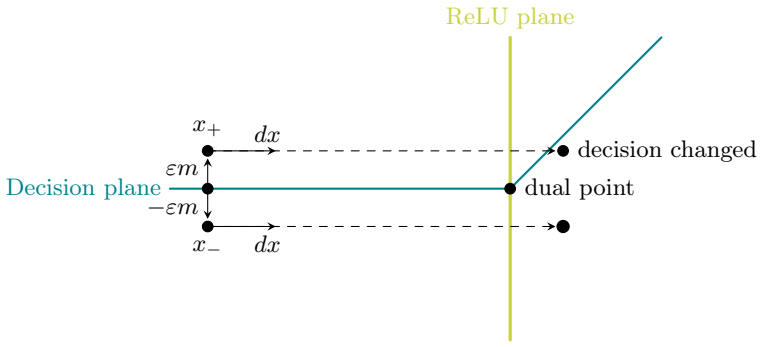
We remove this problem with a refinement step following the techniques from [7] adapted to the hard-label setting, which is run on every layer after its signatures have been recovered and before its signs are recovered (it does not need the signs).

This step works as follows. Once the signature of a neuron is known approximately, we now know roughly *where* its critical hyperplane H is, and can therefore generate fresh critical points directly on it. But we can place these critical points far apart from each other, and that lets us make use of the full floating-point precision. For a random starting point, we first project it onto H using the approximate signature and the (already refined) layers below it; this requires no queries. We then walk inside H until the hard label changes and bisect, which gives a point x_B lying on a decision boundary B and, up to the roughly 10^{-7} error of the approximate signature, on H. In the plane spanned by the normal of H and the walking direction, B consists of two straight lines which meet exactly on H (this is precisely the situation of a dual point, cf. Section 4). We therefore bisect the position of B at three locations on each side of H (at distances $\varepsilon, 2\varepsilon, 3\varepsilon$ from x_B along the approximate normal, $\varepsilon = 10^{-3}$), fit a line through each triple, and take the intersection of the two lines. If either triple is not collinear, the two lines do not bend, or they intersect outside the window, some other hyperplane passes through the window and the point is discarded.

The intersection point lies on the true H to within floating point tolerance. After collecting 64 such points per neuron (about $2 \cdot 10^4$ queries per neuron), the refined signature and bias are obtained as the nullspace of the $64 \times (d_{i-1} + 1)$ matrix of these points.

In our experiments this reduces the maximum relative error of every layer from roughly $\cdot 10^{-7}$ before refinement to $1.0 \cdot 10^{-14}$, $1.7 \cdot 10^{-14}$ and $4.0 \cdot 10^{-14}$ on each of the first three layers after it (see Table 4), i.e. to approximately the precision of 64-bit arithmetic, so that each layer can be attacked behind the layers recovered before it with no observable accumulation of errors.

Fig. 9: Depiction of the routine to walk to a decision-plane bend.



Algorithm 7 WalkToDecisionBend$(x, m, \varepsilon, \delta)$

Input: x a point at the decision boundary, m the decision hyperplane normal vector of unit norm, and small constants ε, δ.
Output: A point x' at which the decision hyperplane bends.

$x_\pm \leftarrow x \pm \varepsilon m$
$dx \leftarrow$ random vector[19]
$dx \leftarrow dx - (dx \cdot m)m$
while $z(x_+ + dx) \neq z(x_- + dx)$ **do**
    $dx \leftarrow 2dx$ ▷ double until boundary is crossed
while $|dx| > \delta$ **do** ▷ binary search for the crossing point
    if $z(x_+ + dx/2) \neq z(x_- + dx/2)$ **then**
        $x_\pm \leftarrow x_\pm + dx/2$
    $dx \leftarrow dx/2$
return $(x_+ + x_-)/2$

Results. We ran the complete attack on the network described above, first finding a collection of 6,880 dual points (together with the respective normal vectors of the decision boundary on both sides of each of them) at a cost of $2.02 \cdot 10^7$ queries.

All subsequent signature recovery steps work on this set without any further queries. Of these dual points, about a quarter ended up belonging to the first hidden layer, a third to the second layer, and the rest to the third. This means that each neuron is represented by between 40 and 90 dual points, far more than the 3 to 12 dual points that are needed to recover it.

The clustering algorithm used here is the full pairwise algorithm of Section 5, with two practical additions to improve runtime of the algorithm. First, a dual point that has not found a consistent partner after 1000 comparisons is assumed to belong to a deeper layer and abandoned. Second, a cluster is only accepted once it contains at least three mutually consistent dual points in order to ensure degenerate solutions are discarded. With these, each layer is clustered and solved in well under ten seconds on a 32 core machine, with each of the 96 neurons recovered nearly perfectly (see Table 4). The signatures obtained directly from the dual points are accurate to about 10^{-7}; the refinement step described above brings every layer to about 10^{-14} at a cost of roughly $2 \cdot 10^4$ queries per neuron. These refined extracted neurons are then used for running the attack on future layers.

For sign recovery we generate, for each target neuron, fresh dual points that lie in the region in which all neurons of the previously recovered layers are active, so that we have perfect control over the target neuron in the sense of Section 6 on every layer. A dual point for which the walk of Algorithm 7 runs into a toggle of an already recovered neuron is discarded. Because we now have perfect control,

[19] For sign recovery, dx is initialized as the normal vector to the target neuron's ReLU hyperplane rather than a random vector.

a vote is essentially never wrong, and we can recover all 96 signs after just 20 votes each as shown in Table 5.

Finally we recover the output layer by directly solving the system of equations determined by using the 1,810 points on the decision boundaries with $9.5 \cdot 10^4$ queries.

The entire attack used a total of $3.97 \cdot 10^7$ queries and 74 seconds of wall-clock time on a 32-core machine. The extracted network agrees with the original to a relative error of at most $4 \cdot 10^{-14}$ in every layer (after matching neurons up to order and scale). It assigns exactly the same label as the original network on 100,000 random inputs as well as to $3 \times 3{,}644$ inputs placed at distances of 10^{-3}, 10^{-6} and 10^{-9} on either side of the original's decision boundary.

Table 4: End-to-end black-box signature recovery on the 32-32-32-32-10 network. The attack is fully black-box and end-to-end: each layer is recovered behind the (refined) layers recovered before it, so the errors of the earlier layers are taken into account. All three layers are recovered from the same set of 6,880 dual points, which took $2.02 \cdot 10^7$ queries (7 seconds) to find; the clustering and solving steps need no further queries since the normal vectors are stored together with the dual points.

		Clustering		**Refinement**	
Layer	**Recovered**	max. rel. error	time	queries	max. rel. error
1	**32**/32	$8.5 \cdot 10^{-8}$	7 s	$6.4 \cdot 10^5$	$1.0 \cdot 10^{-14}$
2	**32**/32	$4.9 \cdot 10^{-8}$	8 s	$7.4 \cdot 10^5$	$1.7 \cdot 10^{-14}$
3	**32**/32	$5.6 \cdot 10^{-8}$	4 s	$8.9 \cdot 10^5$	$4.0 \cdot 10^{-14}$

Table 5: End-to-end black-box sign recovery results on the random untrained 32-32-32-32-10 network, run on the refined signatures of Table 4 (i.e. behind the recovered, not the true, previous layers). Experiments are stopped after 20 non-discarded dual points; all but one neuron were decided unanimously.

		Experiments per neuron		**Confidence level**				
Layer	**Recovered**	total	non-discarded	min(CL)	mean(CL)	max(CL)	queries per neuron	time per neuron
1	**32**/32	21 ± 1	20 ± 0	99.97%	99.99%	99.99%	$(5.7 \pm 0.2) \cdot 10^4$	(1.0 ± 0.04)s
2	**32**/32	25 ± 4	20 ± 0	99.99%	99.99%	99.99%	$(8.1 \pm 1.4) \cdot 10^4$	(1.5 ± 0.3)s
3	**32**/32	119 ± 67	20 ± 0	99.99%	99.99%	99.99%	$(4.0 \pm 2.3) \cdot 10^5$	(9.7 ± 5.8)s

9 Inferring the Architecture

So far we have assumed knowledge of the network architecture for simplicity but we can in fact lift this assumption: it is possible to discover both the depth (the number of layers) and the width (the number of neurons on each layer) with a black-box attack.

Recovering the depth is trivial: one would simply implement the attack exactly as described, one layer at a time, and stop once the recovered layers can account for every single dual point that has been recorded.

It remains to show how to recover the width of the current layer, so that we know when it is time to move on to the next layer. As long as the network is densely connected and ReLU-based, our attack can identify that a neuron belongs to the current target layer when at least two dual points with consistent dual spaces are found, and it just samples dual points at random without needing to know how many neurons are in the layer beforehand. Assuming that dual points are distributed uniformly among the different neurons, we can continue collecting dual points and clustering them until we are sufficiently sure that no further neurons exist in the layer. Note however that uniform distribution among neurons may not be a reasonable assumption for trained models, which have been observed to produce neurons that almost never toggle. This problem has been extensively studied by Ito et al. [15], and we will delve into it later in the section.

We first assume that dual points are evenly distributed among the neurons. By the coupon collector argument, if the layer contains d neurons, then $\mathcal{O}(d \log d)$ dual points suffice to observe every neuron with high probability. The same asymptotic complexity holds if every neuron must be observed at least twice.
When d is unknown, we use an adaptive stopping rule. Suppose that d' distinct neurons have been discovered so far, and sample M additional dual points without discovering a new neuron. If the true number of neurons is $d > d'$, then assuming uniform distribution the probability of this event is

$$\left(\frac{d'}{d}\right)^M \leq \left(\frac{d'}{d'+1}\right)^M \leq \exp\left(-\frac{M}{d'+1}\right).$$

Therefore, choosing

$$M \geq (d'+1)\ln(1/\delta)$$

makes the probability of stopping while at least one neuron remains undiscovered at most δ for this test.
To obtain an overall failure probability that is bounded, we may use

$$\delta_{d'} = \frac{6\delta}{\pi^2(d'+1)^2}$$

at the stage with d' discovered neurons. Since $\sum_{d'=0}^{\infty} \delta_{d'} = \delta$, a union bound shows that the complete procedure fails with probability at most δ. The resulting number of samples remains polynomial in d and $\log(1/\delta)$; even pessimistically summing the stopping budgets over all intermediate values of d' gives

$$\mathcal{O}\big(d^2\big(\log d + \log(1/\delta)\big)\big). \tag{5}$$

Thus, the width of the layer can still be inferred in polynomial time under the uniform-dual-point assumption.[20]

9.1 Complexity and Practical Efficiency

Putting the above bounds in context, the dominant generic post-processing step of our attack is the clustering of collected dual points: for Q dual points, the naive pairwise procedure requires $\mathcal{O}(Q^2)$ consistency tests, each involving polynomial-time linear algebra in the relevant layer dimension. For architecture inference, under the uniform-dual-point assumption, the coupon-collector argument requires $\mathcal{O}(d \log d)$ dual-point samples when the layer width d is known, while the conservative adaptive stopping rule above uses $\mathcal{O}\big(d^2\big(\log d + \log(1/\delta)\big)\big)$ dual-point samples when d is unknown (cf. Equation (5)).

We stress that one dual-point sample should not be identified with one hard-label oracle query. Finding and characterizing a dual point requires multiple oracle queries, including binary searches along the decision boundary and the reconstruction of the corresponding decision-boundary normal vectors. Hence, the total number of hard-label queries contains additional polynomial factors depending among other parameters, on the input dimension d_0 and the numerical precision required by these procedures. Thus, under the uniform dual-point assumption used above, the relevant sampling and post-processing procedures have polynomial complexity in the network dimensions and required numerical precision. However, the above dual-point bounds should not be interpreted directly as bounds on the total number of hard-label oracle queries.

This asymptotic picture is complemented by the end-to-end experiment of Section 8, where the fully black-box extraction required 3.97×10^7 hard-label queries and 74 seconds of wall-clock time on a 32-core machine.

An important caveat is that, in trained networks, some neurons may be persistent or nearly dead, making suitable dual points extremely unlikely and potentially requiring exponentially many queries as the depth grows. We discuss this issue and the corresponding cross-layer techniques in the following subsection.

9.2 Dealing with persistent neurons

In a recent paper, Ito et al. [15] highlight a practical issue in inferring trained DNNs: some neurons become "stuck," effectively remaining always on or always off for the vast majority of inputs. These neurons are challenging since we are unable to find dual points for them, making it impossible to recover their signatures. While always-off neurons can be ignored to achieve an equivalent model

[20] Given that attempting to peel away a layer with incomplete neurons will result in incoherent predictions for the next layer, we note that an attacker would unavoidably notice the missing neuron when no consistent dual points are found in the next layer and could go back to fix the mistake at that point. Therefore, although we have provided a robust method to make mistakes arbitrarily unlikely, such level of care might not be required in practice.

with fewer neurons, always-on neurons are problematic due to the fact that not taking them into account will induce an error in future layers as described in detail in [15]. Moreover, they introduce a further challenge when we do not assume knowledge of the architecture, since the coupon-collector strategy will never discover them and so far we have no way of discovering how many always-on neurons exist.

The authors of [15] gave an effective solution to extract the parameters of these neurons despite not being able to obtain dual points for them, but still require knowledge of the architecture to know how many of them exist. We briefly summarize their method, and then describe a new extension to it that lifts the requirement of architecture knowledge.

We assume without loss of generality that at layer l the last k neurons are always on, and we write

$$A^{(l)} = \begin{pmatrix} A'^{(l)} \\ A^{*(l)} \end{pmatrix},$$

where $A'^{(l)}$ and $A^{*(l)}$ consist of the first $d_l - k$ and the last k rows, respectively. Since always-off neurons can always be ignored, we assume the network contains none of them. Likewise, we split

$$A^{(l+1)} = \begin{pmatrix} A'^{(l+1)} & A^{*(l+1)} \end{pmatrix}$$

into the first $d_l - k$ and the last k columns. The hidden vector at layer $l+1$ is then

$$\begin{aligned} y^{(l+1)} &= A'^{(l+1)} y'^{(l)} + A^{*(l+1)} \left(A^{*(l)} y^{(l-1)} + b^{*(l)} \right) + b^{(l+1)} \\ &= \begin{pmatrix} A'^{(l+1)} & A^{*(l+1)} A^{*(l)} \end{pmatrix} \begin{pmatrix} y'^{(l)} \\ y^{(l-1)} \end{pmatrix} + b^{(l+1)} + A^{*(l+1)} b^{*(l)}, \end{aligned}$$

so we can picture an equivalent model where layer $l+1$ has weights connecting it both to layers l and $l-1$, and a modified bias $b^{(l+1)} + A^{*(l+1)} b^{*(l)}$. All weights can be recovered via the usual method, including the *cross-layer weights* $C^{(l+1)} := A^{*(l+1)} A^{*(l)}$, by finding the gradients with respect to the *extended hidden state* $\begin{pmatrix} y'^{(l)} & y^{(l-1)} \end{pmatrix}^T$.

Recall that $A^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$. By construction, $A^{*(l)}$ consists of the last k rows of $A^{(l)}$, so $A^{*(l)} \in \mathbb{R}^{k \times d_{l-1}}$, while $A^{*(l+1)}$ consists of the last k columns of $A^{(l+1)}$, so $A^{*(l+1)} \in \mathbb{R}^{d_{l+1} \times k}$. Consequently

$$C^{(l+1)} := A^{*(l+1)} A^{*(l)} \in \mathbb{R}^{d_{l+1} \times d_{l-1}}$$

has rank at most k since it is the product of a $d_{l+1} \times k$ matrix and a $k \times d_{l-1}$ matrix.

Since the authors of [15] assume the architecture of the network to be known, they can infer the number k of always-on neurons and finish by factoring $C^{(l+1)}$ into two matrices of size $k \times d_{l-1}$ and $d_{l+1} \times k$. There are many possible choices for this factorization, all of which represent functionally equivalent incoming and

outgoing weights related to the always-on neurons, which are recovered without ever needing them to toggle.

We now present an extension of this method, which can infer the number of always-on neurons without assuming knowledge of the architecture. The key insight is that since $C^{(l+1)}$ is the product of matrices with dimensions $d_{l+1} \times k$ and $k \times d_{l-1}$, its rank is bounded by k. In fact, in any non-adversarial example the rank is likely to be exactly k, provided that $k < d_{l-1}, d_{l+1}$, which is likely to be true in reasonable architectures since always-on neurons are relatively rare.

Therefore, we can still recover the architecture of the network by first using the coupon-collector strategy to recover all the neurons that are not always on, and then assuming the presence of cross-layer weights and computing the rank of this matrix to determine the number of always-on neurons.

10 Extending to Leaky ReLUs

Throughout this paper we have focused on the setting of ReLU activated neurons with the activation function $\max(x, 0)$. In this section we show how to implement our attack for the so-called *leaky ReLU* activation function defined by $\max(x, c \cdot x)$ where c is a small constant like 0.1. This leaky ReLU is so-named because instead of being a constant function zero when $x < 0$, the function "leaks" some small amount of signal to the negative region.

Leaky ReLUs have been shown to increase model accuracy because they help mitigate the problem of dead neurons. In a ReLU network, if a neuron is (unluckily) initialized so that no input causes it to enter the active regime, the gradient through this neuron may always remain at 0 and thus remain "dead" forever. In a leaky ReLU network, no neuron is ever completely inactive and so dead neurons are impossible.

It turns out that every algorithm we have introduced above applies identically to the case of leaky ReLU networks nearly without modification. We briefly summarize the primary modifications that are necessary in the remainder of this section. We note that our work was developed concurrently and independently of that of Qi *et al.* [22], who also study parameter extraction from leaky-ReLU networks.

Signature Recovery Our normal signature recovery attack works by observing that the difference between the activations of a neuron in the positive regime minus the negative regime directly allows us to read off the exact weights of the neuron. Leaky ReLUs change this only trivially: instead of the difference directly giving the value of the weights, it is off by a factor of $(1-c)$. But because our attack *already* is invariant to the magnitude of the weights, introducing a multiplicative error is indistinguishable from the perspective of our attack: the recovery algorithm does not need to change *at all*!

In fact, the only other necessary change to our algorithm is made easier by the presence of leaky ReLUs. In Algorithm 4, in order to recover neurons at deeper layers, it was possible that we might need to unify together many partial

dual spaces. In some cases, for deep models, when we are unlucky, it requires searching over hundreds of neurons to find enough inputs so that each input is active at least once. Leaky ReLUs significantly simplify the extraction process because neurons are never inactive, and therefore only exactly two dual points are ever required for completely recovering any neuron on any layer.

Sign Recovery Similarly to signature recovery, the algorithm for recovering signs can be applied without any meaningful modification to the case of leaky ReLUs. Indeed, the core insight of the sign recovery is that future layers will change slightly slower when one of the neurons is off, but the statement remains true when the neuron is attenuated by a small constant c rather than being off.

The use of leaky ReLUs actually opens the door for even better recovery methods. Since there is no loss in dimension due to inactive neurons, we will always have full degree of control in networks where the layer size is non-increasing. We propose an efficient signature recovery method based on an observation that Canales-Martínez and Santos made for the case of regular ReLUs [6].

In the linear neighborhood of a point x in a decision boundary, we represent the behavior of the network as

$$f(x+\delta) - f(x) = G_x \cdot I_x \cdot A \cdot F_x \cdot \delta,$$

where A is the weight matrix of the target layer, F_x and G_x are the local matrices corresponding to the past and future layers, respectively, and I_x is a diagonal matrix of 1's and 0's (in the case of regular ReLUs) or 1's and c's (in the case of leaky ReLUs). Let a and b be the two classes whose output is tied at the dual point x, and let

$$c_x \coloneqq (G_x \cdot I_x)[a,:] - (G_x \cdot I_x)[b,:]$$

be the difference of the corresponding rows of $G_x \cdot I_x$. Since the activation I_x is included in c_x, computing c_x itself is sufficient to recover the signs of the target layer: neurons that are negative will have a corresponding entry in c_x that is 0 (in the case of normal ReLUs), or that is smaller than the others by a factor of c (in the case of leaky ReLUs).

Assuming that δ is any displacement along the decision boundary, this means that

$$c_x \cdot A \cdot F_x \cdot \delta = 0.$$

Canales-Martínez and Santos [6] note that plugging $c_x = m \cdot (AF_x)^{-1}$, where m is the decision plane normal vector and $(AF_x)^{-1}$ is a left-inverse, verifies the above equation, and thus claim that c_x can be computed as $m \cdot (AF_x)^{-1}$. More precisely, both c_x and $m \cdot (AF_x)^{-1}$ are orthogonal to the space $AF_x\Delta$, where Δ is the space of all vectors along the decision boundary. Since $AF_x\Delta$ has dimension at least $d_i - 1$, it follows that c_x and $m \cdot (AF_x)^{-1}$ are multiples of each other, which still allows us to read off the neuron signs.

Note that in the case of leaky ReLUs we still require a statistical test, since points where a neuron is negative will have a smaller c_x coefficient *on average*, but any single point may violate this trend. Our sign recovery thus proceeds as follows:

1. Find multiple points $x^{(i)}$ at the decision boundary
2. Find the corresponding decision hyperplane normal vectors $m^{(i)}$
3. Compute $c^{(i)} = m^{(i)} \cdot (AF_{x^{(i)}})^{-1}$
4. For each neuron j, split the experiments into two subsets where the target neuron takes opposite signs. Check which of the subsets has a larger average value for $|c_j^{(i)}|$, and guess that the neuron is positive on this subset.

We stress that steps 1-3 are independent of the target neuron, so the computations only need to be carried out once to attack every neuron in the layer. Moreover, steps 1 and 2 are even independent of the target layer, so finding a decision-hyperplane point and computing the normal vector m (which represents the heaviest part of the computation) only needs to be done once to produce a sample that is reused for every neuron in every layer. We also stress that this method does not require sampling dual points, only points at any decision boundary which are easier to generate. Because of these advantages, we are able to produce a fully black-box sign recovery that is not just simpler to implement but also drastically more efficient.

Results We experimentally confirm the analysis in the above section. We train a similar model to the one used in the above section (32 neurons wide, repeated for three hidden layers) but with a leaky ReLU activation function with a leakiness value of 0.1 (a standard value used in the literature—but implementations frequently differ significantly, PyTorch defaults to 0.01 and TensorFlow defaults to 0.3).

We were able to perform signature recovery on this model without modifications to the algorithm, whereas for sign recovery we implemented the aforementioned method which proves to be significantly more efficient. We conduct the experiment with an incrementing list of decision-boundary points generated randomly, and stop as soon as the described process predicts the correct sign for every neuron in the network using this list of decision-boundary points. On average, only 145 decision-boundary points are required to recover all 96 hidden-layer signs, and the entire process (implemented in a fully black-box setting) takes under a minute on a standard personal computer.

11 Conclusions

In this paper we resolve the long-standing question of whether polynomial-time parameter extraction is possible in the hard-label setting for the class of fully connected ReLU networks considered here, under our stated assumptions. Previous polynomial-time cryptanalytic extraction attacks relied on access to numerical logits; our attack uses only hard labels of adaptively chosen inputs.

Beyond the asymptotic analysis of the individual attack components, we evaluate their concrete query and computational costs. Our large-scale proof-of-concept experiments recover approximately 935,000 parameters of a trained CIFAR-10 network, demonstrating the scalability of the core attack components

to realistic model sizes. Separately, we implemented the complete attack in a fully black-box, end-to-end setting on a smaller artificial 32–32–32–32–10 network. This extraction required 3.97×10^7 hard-label queries and 74 seconds of wall-clock computation on a 32-core machine, recovering all 96 hidden neurons and the output layer to approximately 10^{-14} relative parameter error. These experiments complement the asymptotic analysis by quantifying the concrete cost of the attack and also highlight the remaining gap between large-scale proof-of-concept extraction and fully end-to-end extraction of trained networks.

We further described extensions to unknown architectures and leaky-ReLU activations. As discussed in Section 9.1, persistent or nearly dead neurons constitute an important caveat to the direct dual-point sampling analysis. We addressed this issue with the use of cross-layer techniques that do not require such neurons to toggle directly, thereby avoiding the potentially exponential cost of obtaining dual points for them by direct sampling.

Acknowledgments. We would like to express our sincere gratitude to Isaac Canales-Martínez for his valuable contributions to this paper. His experimental tests provided critical insights that shaped our findings. We also appreciate his proofreading of the manuscript, which enhanced its clarity and coherence. Additionally, his review of the code written by others helped to maintain the integrity and accuracy of our work.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Asselineau, R., Derbez, P., Fouque, P.A., Minaud, B.: Cryptanalytic extraction of deep neural networks with non-linear activations. In: Advances in Cryptology – CRYPTO 2026. Springer (2026), https://eprint.iacr.org/2026/253, to appear
2. Baum, E.B.: A polynomial time algorithm that learns two hidden unit nets. Neural Comput. **2**(4), 510–522 (1990), https://doi.org/10.1162/neco.1990.2.4.510
3. Baum, E.B.: Neural net algorithms that learn in polynomial time from examples and queries. IEEE Trans. Neural Networks **2**(1), 5–19 (1991), https://doi.org/10.1109/72.80287
4. Blum, A., Rivest, R.L.: Training a 3-node neural network is NP-Complete. In: Hanson, S.J., Remmele, W., Rivest, R.L. (eds.) Machine Learning: From Theory to Applications - Cooperative Research at Siemens and MIT. Lecture Notes in Computer Science, vol. 661, pp. 9–28. Springer (1993)
5. Canales-Martínez, I.A., Chávez-Saab, J., Hambitzer, A., Rodríguez-Henríquez, F., Satpute, N., Shamir, A.: Polynomial time cryptanalytic extraction of neural network models. In: Joye, M., Leander, G. (eds.) Advances in Cryptology - EUROCRYPT 2024, Proceedings, Part III. Lecture Notes in Computer Science, vol. 14653, pp. 3–33. Springer (2024)
6. Canales-Martínez, I.A., Santos, D.: Extracting some layers of deep neural networks in the hard-label setting. Cryptology ePrint Archive, Paper 2025/1118 (2025), https://eprint.iacr.org/2025/1118

7. Carlini, N., Jagielski, M., Mironov, I.: Cryptanalytic extraction of neural network models. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology - CRYPTO 2020, Proceedings, Part III. Lecture Notes in Computer Science, vol. 12172, pp. 189–218. Springer (2020)
8. Chen, Y., Dong, X., Guo, J., Shen, Y., Wang, A., Wang, X.: Hard-label cryptanalytic extraction of neural network models. In: Chung, K., Sasaki, Y. (eds.) Advances in Cryptology - ASIACRYPT 2024, Proceedings, Part VIII. Lecture Notes in Computer Science, vol. 15491, pp. 207–236. Springer (2024)
9. Daniely, A., Granot, E.: An exact poly-time membership-queries algorithm for extracting a three-layer ReLU network. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net (2023)
10. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., Houlsby, N.: An image is worth 16 x 16 words: Transformers for image recognition at scale. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net (2021)
11. Fefferman, C.: Reconstructing a neural net from its output. Revista Matemática Iberoamericana **10**(3), 507–555 (1994), http://eudml.org/doc/39464
12. Foerster, H., Mullins, R.D., Shumailov, I., Hayes, J.: Beyond slow signs in high-fidelity model extraction. In: Globerson, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J.M., Zhang, C. (eds.) Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024 (2024)
13. Hancock, T.R., Golea, M., Marchand, M.: Learning nonoverlapping perceptron networks from examples and membership queries. Mach. Learn. **16**(3), 161–183 (1994)
14. Hoeffding, W.: Probability inequalities for sums of bounded random variables. The collected works of Wassily Hoeffding pp. 409–426 (1994)
15. Ito, A., Miura, T., Todo, Y.: Is the hard-label cryptanalytic model extraction really polynomial? In: Advances in Cryptology – CRYPTO 2026 (2026), to appear. IACR Cryptology ePrint Archive, Report 2025/1868, https://eprint.iacr.org/2025/1868.pdf
16. Jagielski, M., Carlini, N., Berthelot, D., Kurakin, A., Papernot, N.: High accuracy and high fidelity extraction of neural networks. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1345–1362 (2020)
17. Lin, Z., Memisevic, R., Konda, K.: How far can we go without convolution: Improving fully-connected networks. arXiv preprint arXiv:1511.02580 (2015)
18. Liu, H., Siproudhis, A., Experton, S., Lorenz, P., Boura, C., Peyrin, T.: Navigating the deep: Signature extraction on deep neural networks (2025), https://arxiv.org/abs/2506.17047
19. Liu, H., Siproudhis, A., Experton, S., Lorenz, P., Boura, C., Peyrin, T.: Navigating the deep: End-to-end extraction on deep neural networks. In: Daemen, J., Thomé, E. (eds.) Advances in Cryptology – EUROCRYPT 2026. Lecture Notes in Computer Science, vol. 16546, pp. 482–512. Springer (2026)
20. Martinelli, F., Simsek, B., Gerstner, W., Brea, J.: Expand-and-cluster: Parameter recovery of neural networks. In: Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024. OpenReview.net (2024)
21. Milli, S., Schmidt, L., Dragan, A.D., Hardt, M.: Model reconstruction from model explanations. In: danah boyd, Morgenstern, J.H. (eds.) Proceedings of the Con-

ference on Fairness, Accountability, and Transparency, FAT* 2019, Atlanta, GA, USA, January 29-31, 2019. pp. 1–9. ACM (2019)
22. Qi, X., Lei, H., Wei, L., Sun, X., Wang, M.: Cryptanalytic extraction of neural networks with various activation functions. IACR Transactions on Symmetric Cryptology **2026**(1), 468–505 (Mar 2026). https://doi.org/10.46586/tosc.v2026.i1.468-505, https://tosc.iacr.org/index.php/ToSC/article/view/12793
23. Reith, R.N., Schneider, T., Tkachenko, O.: Efficiently stealing your machine learning models. In: Cavallaro, L., Kinder, J., Domingo-Ferrer, J. (eds.) Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society, WPES@CCS 2019, London, UK, November 11, 2019. pp. 198–210. ACM (2019)
24. Rolnick, D., Körding, K.P.: Reverse-engineering deep ReLU networks. In: Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event. Proceedings of Machine Learning Research, vol. 119, pp. 8178–8187. PMLR (2020)
25. Torralba, A., Fergus, R., Freeman, W.T.: 80 million tiny images: A large data set for nonparametric object and scene recognition. IEEE transactions on pattern analysis and machine intelligence **30**(11), 1958–1970 (2008)
26. Tramèr, F., Zhang, F., Juels, A., Reiter, M.K., Ristenpart, T.: Stealing machine learning models via prediction {APIs}. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 601–618 (2016)

A Appendix: Supporting Figures on Experimental Outcomes

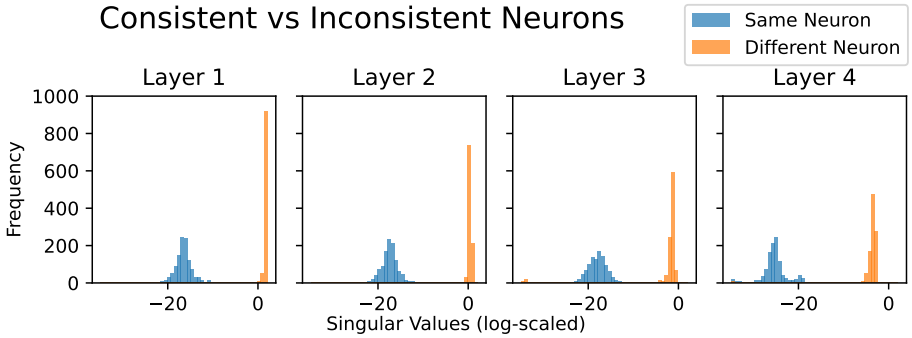


Fig. 10: Our method that computes the consistency between two dual points nearly perfectly separates the distributions of consistent and inconsistent dual-point pairs.

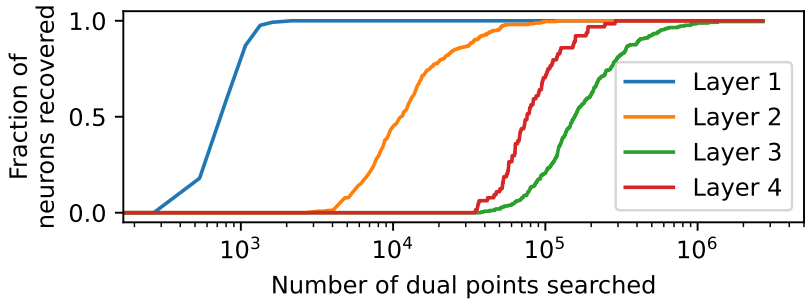


Fig. 11: Fraction of neurons recovered on each layer as a function of the number of dual points explored. Early layers are easier to recover than later layers, requiring just a few thousand dual points, but deeper layers can require millions of dual points.

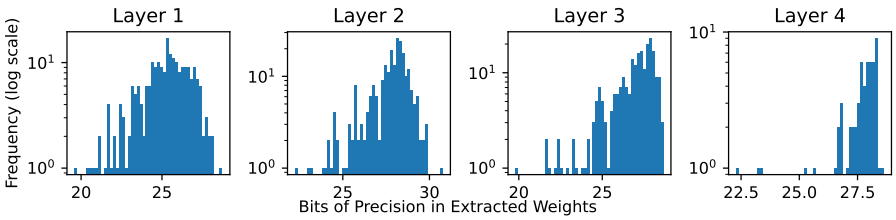


Fig. 12: Bit precision of extracted neuron weights, assuming perfect recovery of all preceding layers.

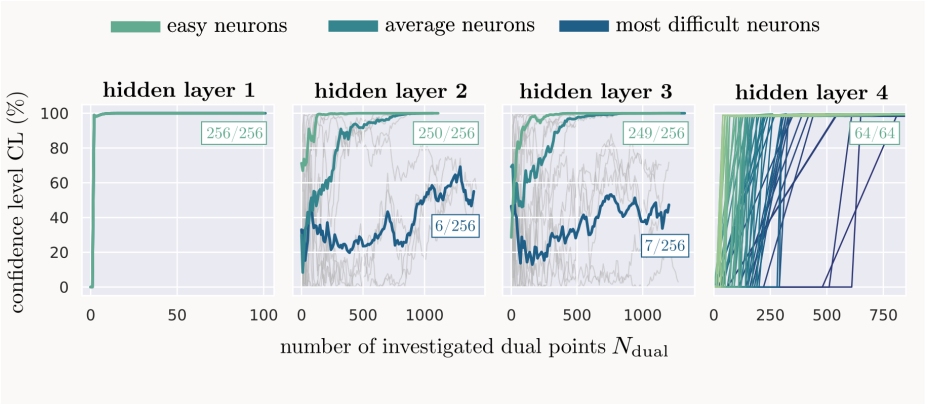


Fig. 13: The evolution of the sign recovery confidence level CL with the investigated number of dual points N_{dual} for easy neurons, average neurons, and the most difficult neurons in each layer.