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 their public accessibility raises security concerns about parameter extraction by malicious actors. Recent work by Carlini et al. (Crypto'20) and Canales-Martínez et al. (Eurocrypt'24) has drawn parallels between this issue and block cipher key extraction via chosen plaintext attacks. Leveraging differential cryptanalysis, they demonstrated that all the weights and biases of black-box ReLU-based DNNs could be inferred using a polynomial number of queries and computational time. However, their attacks relied on the availability of the exact numeric value of output logits, which allowed the calculation of their derivatives. To overcome this limitation, Chen et al. (Asiacrypt'24) tackled the more realistic *hard-label scenario*, where only the final classification label (e.g., "dog" or "car") is accessible to the attacker. They proposed an extraction method requiring a polynomial number of queries but an exponential execution time. In addition, their approach was applicable only to a restricted set of architectures, could deal only with binary classifiers, and was demonstrated only on tiny neural networks with up to four neurons split among up to two hidden layers.

This paper introduces new techniques that, for the first time, achieve cryptanalytic extraction of DNN parameters in the most challenging hard-label setting, using both a polynomial number of queries *and* polynomial time. We validate our approach by extracting nearly one million parameters from a DNN trained on the CIFAR-10 dataset, comprising 832 neurons in four hidden layers. Our results reveal the surprising fact that all the weights of a ReLU-based DNN can be efficiently determined by analyzing only the geometric shape of its decision boundaries.

Keywords: ReLU-Based Deep Neural Networks · Neural Network Extraction · Hard-label Attack · Polynomial Query and Polynomial Time Attack.

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 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 available for free through an 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 by such an oracle access to its black box implementation has a rich history and had 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 both

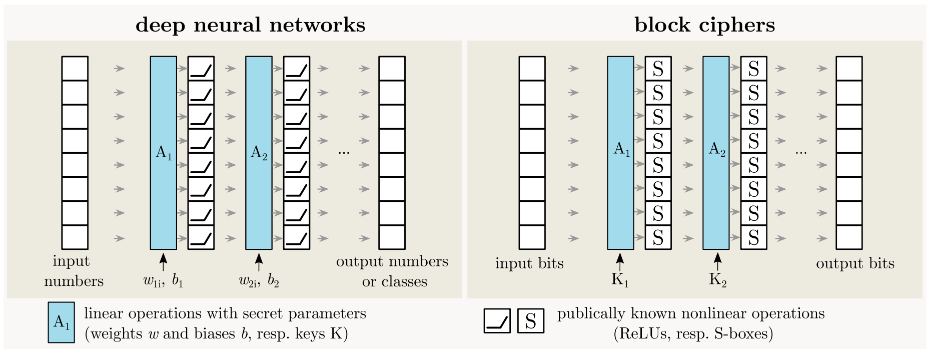

deep neural networks
block ciphers
input numbers
A1
A2
w1i, b1
w2i, b2
output numbers or classes
input bits
S
K1
K2
output bits
linear operations with secret parameters (weights w and biases b, resp. keys K)
publically known nonlinear operations (ReLUs, resp. S-boxes)

Fig. 1: The similarity between DNN's and block ciphers.

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 [13]. 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 only $\lceil \log_2 N \rceil$ 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 [4], 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 [6], over three decades after its initial introduction in 1991 in [2]. Nonetheless, the solution in [6] describes an attack that utilizes a polynomial number of *queries* but requires exponential *time*.[4] This algorithmic limitation forced the authors of [6] 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 harder scenario (S1), consider the way most recent attacks, such as [4], 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

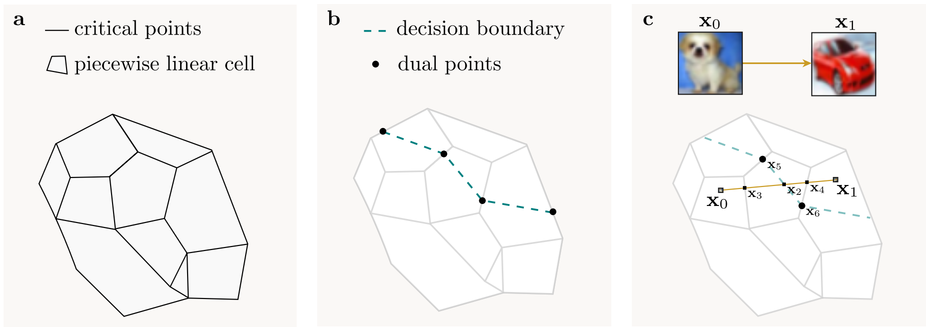


Fig. 2: A schematic description of various attacks. **a.** Input space partition. **b.** Decision boundary. **c.** Path between two inputs.

[4] Moreover, Chen et al. achieved polynomiality only for stealing part of the parameters of the model, and only for a limited subset of DNN 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 [5] and adopted in [4,6]. Such ReLU flips change the behavior of the mapping from one linear 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 only connection 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 transitions 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

[5] Transition points are referred to as *decision boundary points* in [6]. 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.

the class 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 had become 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 this patch 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 [5] and further developed in [4] and [6], but enhances them 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 [6]: 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 extreme situations which are not likely to happen in normally trained networks (unless 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.

Thanks to our enhanced signature and sign recovery algorithms in the hard-label scenario, we could demonstrate the practical applicability of our attack on a real DNN with $935,370$ parameters that was trained to classify the 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. The best previous hard-label attack of [6] on this DNN would have required 2^{256} time, whereas our new method had succeeded in extracted all these parameters on a standard multi-core server with GPU support.

Table 1: **Comparison of Neural Network Extraction Methods.** Previous works successfully extracted either single-hidden-layer models or networks with very few hidden neurons, often requiring full access to floating-point outputs or resulting in exponential time costs. In contrast, our new attack demonstrates the ability to extract a multilayer neural network with approximately 1,000 hidden neurons under the most challenging hard-label setting.

Neural Network			**Parameter Extraction**		
Architecture	Parameters	Hard-Label	Signature	Sign	**Approach**
10-20-20-1	620	⤫	*poly*	*exp*	ICML'20 [18]
10-20-20-1	620	⤫	*poly*	*exp*	Crypto'20 [5]
784-128-1	≈ 0.1M	⤫	*poly*	*exp*	Crypto'20 [5]
3072-256×8-10	≈ 1.25M	⤫	*(from [5])*	*poly*	EC'24 [4]
100-50-50-1	7,650	⤫	*poly*	*poly*	[10]
1024-2-2-1	2,508	⍻	*restr.*	*exp*	AC'24 [6]
3072-256×3-64-10	≈ 0.9M	✓	*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 has access only to hard-labels in 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). All the practical experiments conducted in this study are described in Section 7, while Section 8 presents our concluding remarks.

2 Related Work

The problem of DNN model extraction was first studied in the 1990s [1,2,3,11]. In 1991, Baum presented an algorithm in [1,2] that could infer the Boolean function describing the model of a neural network algorithm 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 [9] 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 [3]. They examined a different scenario in which the attacker was given an adversarially chosen set of known inputs and their corresponding outputs (i.e., she 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 [7,10,13,15,16,17,20]. 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 [4,5,6].

In [5], the authors introduced several efficient techniques for recovering neuron weights and their associated biases with remarkable precision, a process they termed the *neuron's signature recovery*. These methods operate with a polynomial number of queries and 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 [5] involved relatively shallow networks. This limitation was addressed in [4], 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. [10] reported an end-to-end attack that effectively combines the signature recovery methods from [5] with the sign recovery techniques from [4]. 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 [10] 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 [4, 5, 10] fundamentally rely on the assumption that the DNN outputs confidence scores in the form of logits with 64-bit precision. This information allows attackers to compute derivatives and gradients with respect to the input, which is essential for conducting their signature and sign recovery procedures.

A more realistic scenario was studied in [6], 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 [1, 2], highlighting its lasting significance in this field and the persistent elusiveness of this problem that has remained unsolved until this paper.

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 method achieves polynomial time complexity only for a very limited number of hidden layers and supports only single-bit binary DNN outputs. They did not succeed in developing an efficient algorithm for sign recovery, relying instead on a guessing approach, which incurs exponential time complexity. As a result, the signature recovery cannot be considered solved in the general case and the sign recovery problem remains the primary barrier to attacking larger and deeper networks in this setting. In fact, the DNNs successfully targeted in [6] are restricted to those with a maximum of four neurons distributed across two hidden layers.

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 notations used throughout the manuscript which closely follows the terminology first presented in [5] and then adopted in more recent attacks [4,6].

Definition 1. *An* r-deep neural network f_θ *is a function parameterized by* θ *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 linear 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 [4,5,6], 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 the corresponding weight matrix followed by an activation function. Particularly, the* j*-th neuron of layer* i *is the function* η *given by*

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

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* $i = 1, \cdots, r+1$*. 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 [6], 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 *$z$ on the outputs $f(x)$ is computed as arg $max_i f(x)_i$, the coordinate of the maximum of $f(x)$.*[8]

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 $\mathcal{S}(\eta; x)$.*

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* dual *point d is a point that is both on the decision boundary (i.e., $z(f(d + \varepsilon)) \neq z(f(d - \varepsilon))$ for $\epsilon > 0$) and on some critical hyperplane (i.e., there is some neuron with $\eta(d) = 0$).*

Definition 10. *Let $x \in \mathcal{X}$. The* linear neighbourhood *of x is the set*

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

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

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

where $I^{(\ell)}$ are $0-1$ 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 $i \in \{1, \dots, r+1\}$. 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. Furthermore, if we restrict inputs x' to be in the linear neighbourhood 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:

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

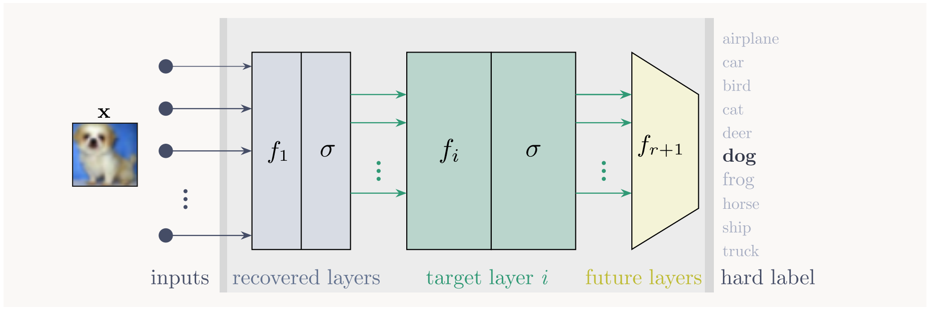


Fig. 3: Representation of the DNN as a composition of the recovered layers $f_1, \ldots, f_{i-1}$, the current target layer f_i and the non-recovered future layers $f_{i+1}, \ldots, f_{r+1}$ (cf. Definition 1).

– *Signature recovery* extracts the parameters of each neuron up to some unknown multiplicative factor. 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.

The main difference from previous attacks is that we no longer have direct access to critical points - we are only aware of transition points between two 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 is depicted 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 inter-

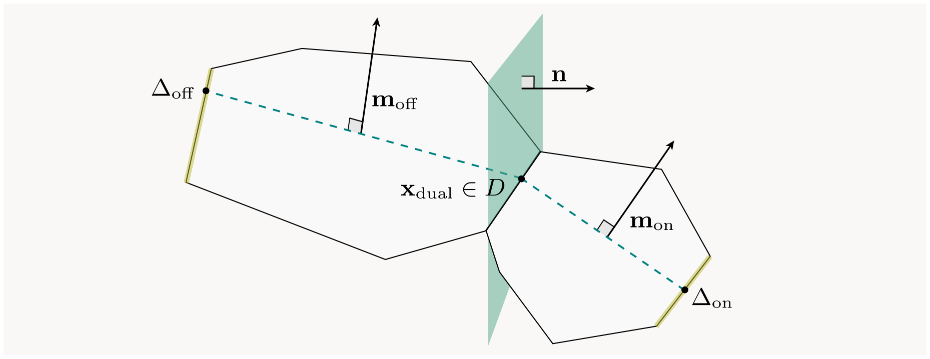


Fig. 4: A green critical hyperplane of some neuron which changes the local orientation of two decision boundary patches on its two sides.

section 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.

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 partial information can be viewed as a set of linear equations, and thus determining 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 [4] 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 absolute value of the 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 wiggling the targeted neuron in a direction which 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 be ended by 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 earlier-layer neurons from our statistics. Fortunately, we already know the signature of all these earlier-layer neurons, so we can test whether the dual subspace that *ends* the decision boundary patch yields a (partial) signature that is already known, and thus discard these cases.

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[9], 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 [5]: **Knowledge of the architecture**. The attacker has knowledge of the number of layers, the number

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

of neurons in each layer, and the number of inputs and outputs in the network. **Full-domain inputs**. The attacker can query arbitrary inputs from $\mathbb{R}^{d_0}$. **Precise computations**. The DNN is specified and evaluated using a sufficiently high precision floating-point arithmetic. **Fully connected network and ReLU activation functions**. The network is comprised of fully connected layers and all activation functions are the ReLU function. Compared to [6], 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

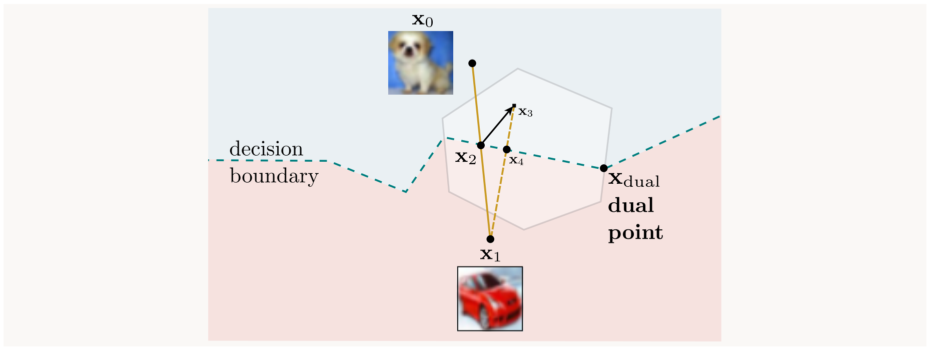


Fig. 5: Visualization of our dual point finding algorithm.

The value of critical points is well understood in the literature [4,5], because it is the location of these points that completely 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 9): inputs that sit both at the decision boundary and also on the critical hyperplane of some neuron. The finding of dual points is one key capability we used throughout our 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. Since we do this with binary search it is impossible to learn the *exact* location of the decision boundary, but with $\log_2 \epsilon$ queries we can recover an input satisfying $|f(x_2)| < \epsilon$. (We will also call this point x_{left}, because later we will find a right point.)

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 corresponds to the one of x_0, find the transition point x_4 using a binary search between x_3 and x_1. If the class $z(f(x_3)) = z(f(x_1))$, find the transition point x_4 using a binary search between x_3 and x_0 instead.

Step 4: move along the decision boundary until $\boldsymbol{x}_{dual}$. 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 appear to *bend*, because we are no longer in the same linear region. The location of the bend is exactly the dual point which is both on the decision boundary, and also 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 the inputs 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 ϵ, the point $x + (\epsilon - \text{proj}_m \epsilon)$ 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 first and second coordinates of the normal vector. By repeating this procedure for each of the remaining directions $e_2, e_3, \ldots, e_{d_0}$, 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).

Definition 11. *The* signature *of a neuron η is equal to $\alpha \cdot A_\eta^{(i)}$, where α is an arbitrary rescaling of the corresponding parameter.*

It is easy to verify that positive constants can be pushed through the network arbitrarily (specifically: if all the weights and the bias of one neuron are multiplied by a constant $a > 0$, and the corresponding inputs to every neuron on the next layer are multiplied by $\frac{1}{a}$, then the model will behave identically). However, negative constants can not be pushed through the model, since this causes neurons to flip from active to inactive which changes the behavior of the model. Therefore, our signature recovery attack mirrors the methodology of prior work [5]: we search for a set of dual points, and then use the locations of these dual points to determine the signatures on the first layer.

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—and, as a consequence, the signatures of all the 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 [5], computing the normal vector to the critical hyperplane induced by the neuron η was possible directly via finite differences, because they had the power to query the model f at arbitrary points $x \in \mathbb{R}^{d_0}$. But now 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 12. *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.[10] 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 dual space contains the $d_0 - 1$ dimensional critical hyperplane, which again by definition, does not contain the normal vector to the 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.

Algorithm 1 RecoverFirstLayerWeights($x_{\text{dual},0}$, $x_{\text{dual},1}$)

```
Input: $x_i$, dual points of the model
Output: The parameters of the model $\alpha \cdot A_\eta^{(1)}$, or $\perp$ if the dual points are inconsistent.
1: $D_i = \text{ComputeDualSpace}(x_i)$
2: $D = \text{span}(\text{basis}(D_0) \cup \text{basis}(D_1))$
3: if $D = \mathbb{R}^{d_0}$ then
4:     return $\perp$
5: else
6:     Find $w$ such that $\text{Span}(\{w\})^\perp = D$
7:     return $w$
```

Algorithm 2 CollectFirstLayerDualPoints(K)

```
Input: $K$, number of dual points to search for
Output: Set of clusters of consistent dual points
1: $S \leftarrow \emptyset$
2: for $i \leftarrow 1$ to $K$ do
3:     $\boldsymbol{x}_{\text{dual}} \leftarrow \text{FindRandomDualPoint}()$
4:     $S \leftarrow S \cup \{\boldsymbol{x}_{\text{dual}}\}$
5: clusters $\leftarrow \emptyset$
6: for each $\boldsymbol{x}_{\text{dual}} \in S$ do
7:     matched $\leftarrow$ false
8:     for each $c \in$ clusters do
9:         $r \leftarrow \text{RandomElement}(c)$
10:        if $\text{IsConsistent}(r, \boldsymbol{x}_{\text{dual}})$ then
11:            $c \leftarrow c \cup \{\boldsymbol{x}_{\text{dual}}\}$
12:            matched $\leftarrow$ true
13:            break
14:    if not matched then
15:        clusters $\leftarrow$ clusters $\cup \{\{\boldsymbol{x}_{\text{dual}}\}\}$
16: return $\{c \in \text{clusters} : |c| > 1\}$
```

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

[10] 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.

the first layer from dual points that correspond to neurons on later layers. We detail this algorithm in Algorithm 2, and describe it here.

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.

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 $D_j^{\perp} = \text{span}(\{A_j^{(1)}, r_j\})$. Thus, because we assume that no two critical hyperplanes are exactly parallel, we will have that $\text{span}(\{A_i^{(1)}, r_i\}) \cap \text{span}(\{A_j^{(1)}, r_j\}) = \emptyset$, 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 13. *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, we can easily filter out and remove any dual points that belong to layers 1 through to j, 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 subsection 5.1.

Definition 14. *Let $f_{x_0;i}$ be a linear 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 15. *The hidden state at layer j is the output of the function $f_{1..j}$, 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 $\hat{n}_i = f_{\boldsymbol{x}_{\text{dual}\,i};i}(n_i)$.[11]

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 16. *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 $d_i/2$ [4].

Definition 17. *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. [12]

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

```
Input: $\boldsymbol{x}_{\text{dual}\,i}$ a dual point of the model, $S_i$ the corresponding dual space
Output: True if the two dual points are on the same neuron's critical hyperplane,
    otherwise False.
1: Let $\tilde{D}_j = f(\boldsymbol{x}_{\text{dual}\,i}; i)(D_j)$
2: Let $D = \text{span}(\text{basis}(\tilde{D}_0) \cup \text{basis}(\tilde{D}_1))$.
3: Let $X$ = the number of neurons active in either $\boldsymbol{x}_{\text{dual}0}$ or $\boldsymbol{x}_{\text{dual}1}$.
4: if Rank($D$) = $X$ then
5:     return No
6: else
7:     return Yes                                  ▷ Specifically, $Rank(D) < X$
```

[11] 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.

[12] Dead neurons are those whose values before the ReLU almost never change sign. Typically, they are always-off neurons as always-on neurons are uncommon.

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.

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 $O((\#\text{ duals})^2)$ work algorithm 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, with low probability the IsConsistent algorithm will spuriously claim two neurons are consistent when in fact they are not with low (e.g., 10^{-6}) probability due to numerical instabilities. To resolve this issue, we instead search for *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 subsection 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 [4]. 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: $\boldsymbol{x}_{\text{dual}j}$ a dual point of the model, $D_j$ the corresponding dual space
Output: The parameters of the model $A_\eta^{(i)}$.
1: assert all dual points $\boldsymbol{x}_{\text{dual}j}$ are (pairwise) consistent.
2: Let $\tilde{D}_j = f(d_j; i)(D_j)$
3: Let $D = \text{span}(\text{basis}(\tilde{D}_0) \cup \text{basis}(\tilde{D}_1) \cup \text{basis}(\tilde{D}_2) \cup \cdots \cup \text{basis}(\tilde{D}_n))$
4: if Rank$(D) < d_0 - 1$ then
5:     return $\perp$                    ▷ Insufficient data to reconstruct parameters.
6: else
7:     Find $w$ such that Span$(\{w\})^\perp = D$
8:     return $w$
```

6 Sign Recovery

Recall that a neuron's signature is related to its actual weights by some arbitrary scaling factor c. Finding the correct *sign* of c is crucial in order to correctly recover the model. Similarly to [4], 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 (subsection 6.1). Next, we present the technique in algorithmic form (subsection 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 [4] 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 [4] to directly measure these infinitesimal changes in the output logits of the neural network.

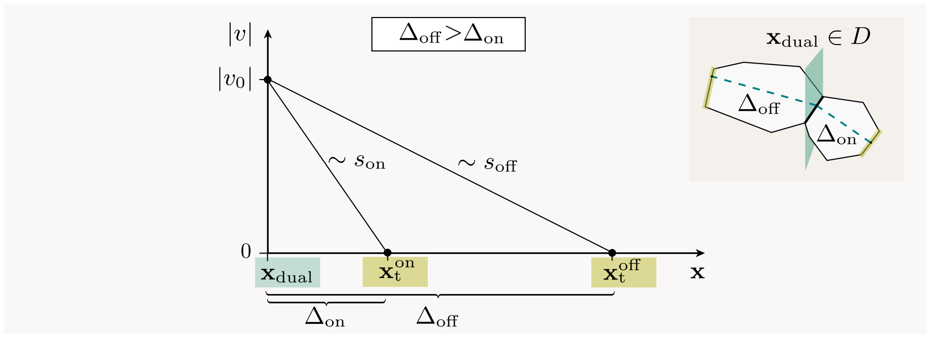
One key intuition for sign recovery without direct access to output logits is that the average speed with which *all* future neurons[13] change their values depends on the target layer output norms. Accordingly, the speed with which the average future neurons change their value on the target neuron's on-side, s_{on}, is larger than that on its off-side, s_{off}, see also Figure 6. The core assumption is that this speed difference correlates with the target layer's output norms, leading to the approximate magnitude estimates given in Equation 1 and Equation 2.

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

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

Any neuron toggling point introduces bends in the decision boundary, which allows us to measure Δ_{on} and Δ_{off}. If a future neuron is approaching its toggling point $\boldsymbol{x}_{\text{t}}$, a higher speed implies it will reach this point *earlier* than if moving at a slower speed (see Figure 6). The ratio between the two very rough theoretical estimates of speed given 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

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



$\Delta_{\text{off}} > \Delta_{\text{on}}$
$|v|$
$|v_0|$
$\sim s_{\text{on}}$
$\sim s_{\text{off}}$
0
$\mathbf{x}_{\text{dual}}$
$\mathbf{x}_{\text{t}}^{\text{on}}$
$\mathbf{x}_{\text{t}}^{\text{off}}$
$\mathbf{x}$
Δ_{on}
Δ_{off}
$\mathbf{x}_{\text{dual}} \in D$
Δ_{off}
Δ_{on}

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 $\Delta_{\text{off}} > \Delta_{\text{on}}$.

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.

Note that in the case of perfect control only the target neuron will change by a value ϵ on the on-side while all other neurons in the target layer will change by 0, resulting in $||v_{\text{on}}||/||v_{\text{off}}|| = \infty$. In this extreme case of perfect control it is obvious that $\Delta_{\text{off}} > \Delta_{\text{on}}$ since there will never be a future toggle on the off-side.[14]

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 CIFAR10 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[15] across several dual points. At each dual point, the speed of each future neuron is calculated as $s_{\text{on, off}} = |v(\boldsymbol{x}_{\text{dual}}) - v(\boldsymbol{x}_{\text{t}})|/\Delta_{\text{on, off}}$. 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 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

[14] If we move too far away from $\boldsymbol{x}_{\text{dual}}$, a past layer toggle will eventually occur. We discuss in detail later how these cases are handled.

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

Table 2: Whitebox verification of the intuition behind the sign attack across target neurons in Our CIFAR10 Network.

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

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 plane 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 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. 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.

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 plane will change directions whenever any neuron toggles. Therefore, we perform our walk over the decision plane (moving not in the direction of $\pm\hat{\boldsymbol{n}}$ but its projection onto the decision plane), and infer that there has been a neuron toggle whenever we notice a change in the direction of the plane.

It is important to stress that the toggle should only be counted if it comes from a future layer, since toggles in pasts 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.

A unique case of future-neuron influence occurs in the penultimate layer, where

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 $-1$
 1: $\texttt{votes}_+ \leftarrow 0$
 2: $\texttt{votes}_- \leftarrow 0$
 3: for $\texttt{experiment} = 1, \dots, \texttt{N}_{exp}$ do
 4:     $\boldsymbol{x}_{\text{dual}} \leftarrow$ RandomDualPoint($i$)
 5:     $F \leftarrow$ LocalMatrix($\mathbf{W}, \mathbf{B}, \boldsymbol{x}_{\text{dual}}$)
 6:     $\hat{\boldsymbol{n}} \leftarrow F[:, i] \;/\; \|F[:, i]\|$
 7:     $\Delta_+ \leftarrow$ DistanceToToggle($\boldsymbol{x}_{\text{dual}}, \hat{\boldsymbol{n}}$)
 8:     $\Delta_- \leftarrow$ DistanceToToggle($\boldsymbol{x}_{\text{dual}}, -\hat{\boldsymbol{n}}$)
 9:     if $\Delta_+ < \Delta_-$ :  $\texttt{votes}_+ = \texttt{votes}_+ + 1$
10:     else :  $\texttt{votes}_- = \texttt{votes}_- + 1$
11: if $\texttt{votes}_+ > \texttt{votes}_-$ : return $+1$
12: else: return $-1$
```

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 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 investigate a larger number of N_{dual} dual points.

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 ensure our observed probability p_{exp} is not simply a result of random variation around $p_0 = 0.5$, we assess 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 [12] provides a bound on the error probability α, stating:

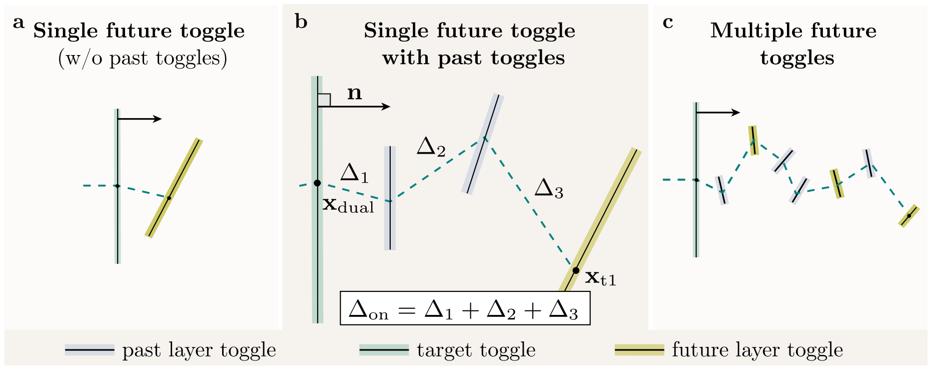


Fig. 7: Visualization of the distance measurement using the DistanceToToggle algorithm. **a.** In its simplest variant the distance measurement only accepts distance measurements from dual points at which the first toggle is a future layer toggle. **b.** A slightly more elaborate variant of the distance 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.

$\alpha = P\left(p_{\text{exp}} - p_0 > \delta_{\text{p}}\right) \leq \exp\left(-2\delta_{\text{p}}^2 N_{\text{exp}}\right)$. 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 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, 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

In our attack on Our CIFAR10 Network we first evaluate the signature recovery (subsection 7.1), and then the sign recovery (subsection 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 sequentially 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 [5], we perform all arithmetic in 64 bit floating point precision and assume that the prior layers were perfectly extracted. Our software implementation is available at

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

Our CIFAR10 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 [19]. Each class consists of images with RGB channels of size 32×32 pixels, resulting in 3,072 pixels 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[16] as [4, 14].

Our four-hidden-layer model achieves a test accuracy of 0.52 on CIFAR-10, which aligns with the expected performance for densely connected neural networks utilizing pure ReLU activation functions, typically around 0.53 [14]. 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 0.995 on CIFAR-10 [8, Table 2].

7.1 Signature Recovery Attack

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

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

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

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 a (1 million)^2 time complexity, requiring 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}.

[16] 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.

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 to 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 8, 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 different neurons can incorrectly have very low 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 9 summarizes this analysis. The number of queries 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 10, 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 Attack

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 $[-5, +5]^{d_0}$.
The sign recovery process for a single neuron utilizes four CPU cores. On a 256-core computer, this enables the simultaneous recovery of 64 neurons. A potential optimization is parallelizing the recovery at the dual point level, as multiple dual points can, in principle, be analyzed independently in parallel. Each vote evaluation is independent, allowing for simultaneous processing.
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 11 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. 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 11). 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 CIFAR10 network.

			Confidence level			**Times per neuron**	
Layer	**Recovered**	$N_{\textbf{dual}}$	min(CL)	mean(CL)	max(CL)	$\bar{t}_{\text{vote}}$	$\sum t_{\text{vote}}$
1	**256**/256	(100±0) s	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 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, respectively seven neurons 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. $t_{\text{sign}} \approx t_{\text{vote}}$, 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.[17]

8 Conclusions

In this paper we have solved the long standing open problem of how difficult it is to extract all the secret parameters of a ReLU-based DNN by interacting with its black box implementation. While previous papers have shown that this problem can be solved in polynomial time in the easiest attack scenario (in which the attacker is given the precise numeric values of all the DNN's logits), in this paper we develop the first polynomial time attack even in the hardest attack scenario (in which only the hard-labels of chosen inputs are provided). In our proof-of-concept implementation we demonstrated the general correctness of our attack by successfully extracting all the approximately one million parameters of a realistic CIFAR10 network with about a thousand neurons. However, a fully optimized end-to-end blackbox implementation which can be used by third parties remains future work.

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.

[17] 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 $4 \times 220\,\text{s} \approx 15$ minutes, hidden layer 2 in about $(4+1) \times 1,670\,\text{s} \approx 2.5$ 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.

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

References

1. 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
2. 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
3. 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)
4. Canales-Martinez, 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)
5. 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)
6. 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)
7. 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)
8. 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 16x16 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)
9. Fefferman, C.: Reconstructing a neural net from its output. Revista Matemática Iberoamericana **10**(3), 507–555 (1994), http://eudml.org/doc/39464
10. Foerster, H., Mullins, R.D., Shumailov, I., Hayes, J.: Beyond slow signs in high-fidelity model extraction. In: Globersons, 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)
11. Hancock, T.R., Golea, M., Marchand, M.: Learning nonoverlapping perceptron networks from examples and membership queries. Mach. Learn. **16**(3), 161–183 (1994). https://doi.org/10.1007/BF00993305, https://doi.org/10.1007/BF00993305
12. Hoeffding, W.: Probability inequalities for sums of bounded random variables. The collected works of Wassily Hoeffding pp. 409–426 (1994)

13. 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)
14. Lin, Z., Memisevic, R., Konda, K.: How far can we go without convolution: Improving fully-connected networks. arXiv preprint arXiv:1511.02580 (2015)
15. 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)
16. Milli, S., Schmidt, L., Dragan, A.D., Hardt, M.: Model reconstruction from model explanations. In: danah boyd, Morgenstern, J.H. (eds.) Proceedings of the Conference on Fairness, Accountability, and Transparency, FAT* 2019, Atlanta, GA, USA, January 29-31, 2019. pp. 1–9. ACM (2019)
17. 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)
18. 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)
19. 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)
20. 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 Appendices

A.1 Algorithms

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 plane. 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 plane until `nToggles` neurons from future layers are toggled, and returns the total distance walked along the decision hyperplane in the input space.

```
 1: $\Delta \leftarrow 0$
 2: $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}_{\text{dual}}$
 3: $\mathbf{dx} \leftarrow \hat{\boldsymbol{n}}$
 4: futureLayerToggles$\leftarrow 0$
 5: while futureLayerToggles $<$ nToggles do
 6:     $\hat{\boldsymbol{m}} \leftarrow$ DecisionPlaneUnitVector($\mathbf{x}_0 + \delta \cdot \mathbf{dx}$)
 7:     $\mathbf{dx} \leftarrow \hat{\boldsymbol{n}} - \langle \hat{\boldsymbol{n}}, \hat{\boldsymbol{m}} \rangle \hat{\boldsymbol{m}}$          ▷ Project $\hat{\boldsymbol{n}}$ onto the decision plane
 8:     Walk from $\boldsymbol{x}_0$ in the direction $\mathbf{dx}$ and let $\boldsymbol{x}_1$ be the point at which the
 9:     ... direction of the decision boundary first changes.
10:     if no past- or current-layer neuron being toggled between $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ then
11:         futureLayerToggles $\leftarrow$ futureLayerToggles $+ 1$
12:     else
13:         either discard the dual point or handle the past- or current-layer toggle
14:     $\Delta \leftarrow \Delta + ||\boldsymbol{x}_1 - \boldsymbol{x}_d||$
15:     $\mathbf{x}_0 \leftarrow \mathbf{x}_1$
    return $\Delta$
```

A.2 Supporting Figures on Experimental Outcomes

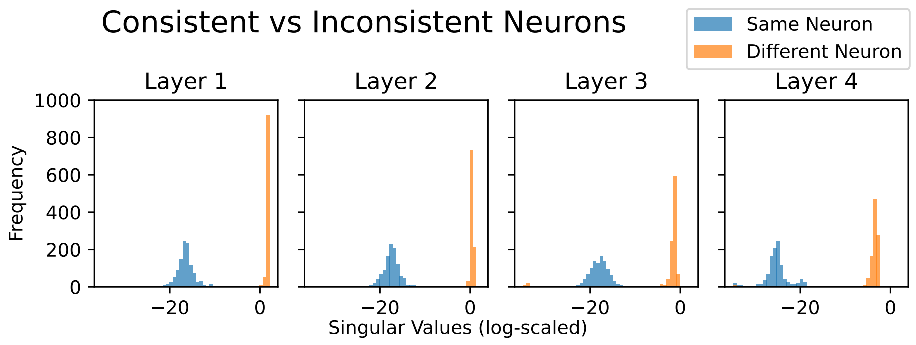


Fig. 8: Our method that computes the consistency between two dual points nearly perfectly separates the distribution of consistent and inconsistent neurons.

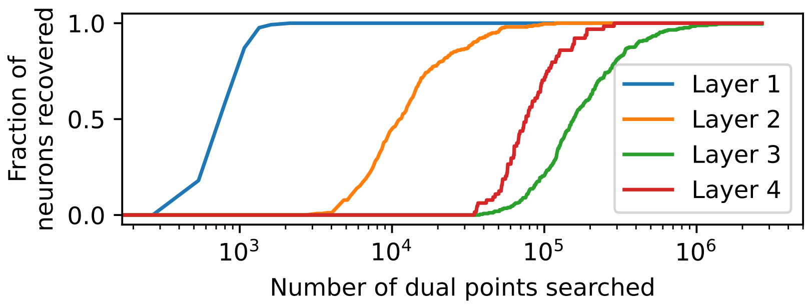


Fig. 9: Fraction of inputs on each layer recovered as a function of the number of dual points explored. Early layers are easier to recover than later layers, requiring just a few thousand dual points, but deeper layers can require millions of dual points.

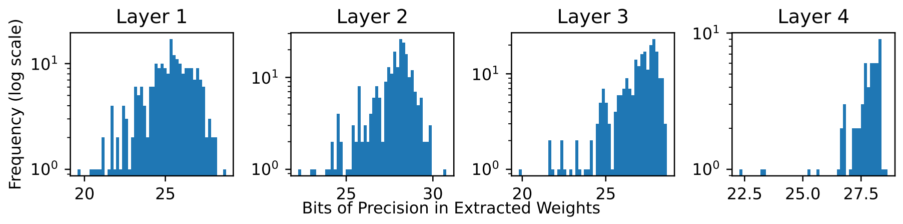


Fig. 10

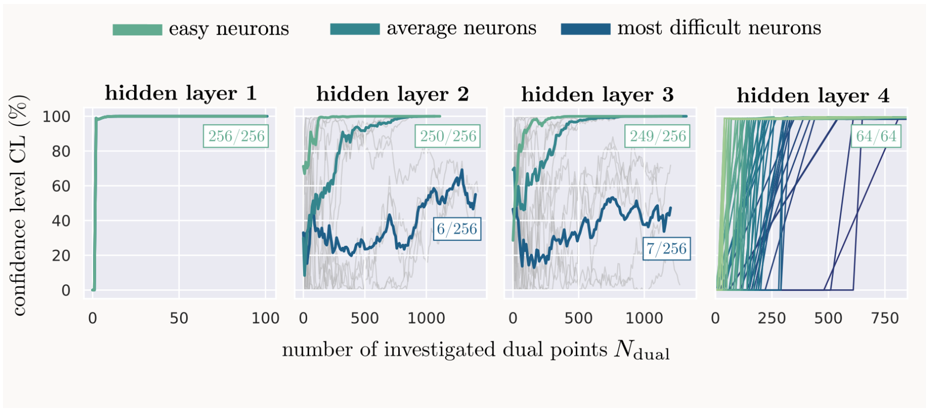


Fig. 11: The evolution of the sign recovery confidence level CL with the investigated number of dual points N_{dual} for easy neurons, average neurons, and the most difficult neurons in each layer.