Cryptanalytic Extraction of Neural Network Models

Nicholas Carlini[1(✉)], Matthew Jagielski[2], and Ilya Mironov[3]

[1] Google, Mountain View, CA, USA
nicholas@carlini.com
[2] Northeastern University, Boston, USA
[3] Facebook, Menlo Park, USA

Abstract. We argue that the machine learning problem of *model extraction* is actually a cryptanalytic problem in disguise, and should be studied as such. Given oracle access to a neural network, we introduce a differential attack that can efficiently steal the parameters of the remote model up to floating point precision. Our attack relies on the fact that ReLU neural networks are piecewise linear functions, and thus queries at the critical points reveal information about the model parameters.

We evaluate our attack on multiple neural network models and extract models that are 2^{20} times more precise and require $100\times$ fewer queries than prior work. For example, we extract a 100,000 parameter neural network trained on the MNIST digit recognition task with $2^{21.5}$ queries in under an hour, such that the extracted model agrees with the oracle on all inputs up to a worst-case error of 2^{-25}, or a model with 4,000 parameters in $2^{18.5}$ queries with worst-case error of $2^{-40.4}$. Code is available at https://github.com/google-research/cryptanalytic-model-extraction.

1 Introduction

The past decade has seen significant advances in machine learning, and deep learning in particular. Tasks viewed as being completely infeasible at the beginning of the decade became almost completely solved by the end. AlphaGo [SHM+16] defeated professional players at Go, a feat in 2014 seen as being at least ten years away [Lev14]. Accuracy on the ImageNet recognition benchmark improved from 73% in 2010 to 98.7% in 2019, a $20\times$ reduction in error rate [XHLL19]. Neural networks can generate photo-realistic high-resolution images that humans find indistinguishable from actual photographs [KLA+19]. Neural network achieve higher accuracy than human doctors in limited settings, such as early cancer detection [EKN+17].

These advances have brought neural networks into production systems. The automatic speech recognition systems on Google's Assistant, Apple's Siri, and Amazon's Alexa are all powered by speech recognition neural networks. Neural

Machine Translation [BCB15] is now the technique of choice for production language translation systems [WSC+16]. Autonomous driving is only feasible because of these improved image recognition neural networks.

High-accuracy neural networks are often held secret for at least two reasons. First, they are seen as a competitive advantage and are treated as trade secrets [Wen90]; for example, none of the earlier systems are open-source. Second, is seen as improving both security and privacy to keep these models secret. With full white-box access it is easy to mount evasion attacks and generate *adversarial examples* [SZS+14,BCM+13] against, for instance, abuse- or spam-detection models. Further, white-box access allows *model inversion* attacks [FJR15]: it is possible to reconstruct identifiable images of specific people given a model trained to recognize specific human faces. Similarly, given a language model trained on text containing sensitive data (e.g., credit card numbers), a white-box attacker can pull this sensitive data out of the trained model [CLE+19].

Fortunately for providers of machine learning models, it is often expensive to reproduce a neural network. There are three reasons for this cost: first, most machine learning requires extensive training data that can be expensive to collect; second, neural networks typically need *hyper-parameter tuning* requiring training many models to identify the optimal final model configuration; and third, even performing a final training run given the collected training data and correctly configured model is expensive.

For all of the above reasons, it becomes clear that (a) adversaries are motivated for various reasons to obtain a copy of existing deployed neural network, and (b) preserving the secrecy of models is highly important. In practice companies ensure the secrecy of these models by either releasing only an API allowing query access, or releasing on-device models, but attempting to tamper-proof and obfuscate the source to make it difficult to extract out of the device.

Understandably, the above weak forms of protection are often seen as insufficient. The area of "secure inference" improves on this by bringing tools from Secure Function Evaluation (SFE), which allows mutually distrustful cooperating parties to evaluate $f(x)$ where f is held by one party and x by the other. The various proposals often apply fully homomorphic encryption [Gen09,GBDL+16], garbled circuits [Yao86,RWT+18], or combinations of the two [MLS+20]. Per the standard SFE guarantee, secure inference "does not hide information [about the function f] that is revealed by the result of the prediction" [MLS+20]. However this line of work often implicitly assumes that total leakage from the predictions is small, and that recovering the function from its output would be difficult.

In total, it is clear that protecting the secrecy of neural network models is seen as important both in practice and in the academic research community. This leads to the question that we study in this paper:

> *Is it possible to* extract *an identical copy of a neural network given oracle (black-box) access to the target model?*

While this question is not new [TZJ+16,MSDH19,JCB+19,RK19], we argue that model extraction should be studied as a cryptanalytic problem. To do this, we focus on model extraction in an idealized environment where a machine

learning model is made available as an oracle $\mathcal{O}$ that can be queried, but with no timing or other side channels. This setting captures that of obfuscated models made public, prediction APIs, and secure inference.

1.1 Model Extraction as a Cryptanalytic Problem

The key insight of this paper is that model extraction is closely related to an extremely well-studied problem in cryptography: the cryptanalysis of block-ciphers. Informally, a symmetric-key encryption algorithm is a keyed function $E_k \colon \mathcal{X} \to \mathcal{Y}$ that maps inputs (plaintexts) $x \in \mathcal{X}$ to outputs (ciphertexts) $y \in \mathcal{Y}$. We expect all practically important ciphers to be resistant, at the very least, to key recovery under the adaptive chosen-plaintext attack, i.e., given some bounded number of (adaptively chosen) plaintext/ciphertext pairs $\{(x_i, y_i)\}$ an encryption algorithm is designed so that the key k cannot be extracted by a computationally-bounded adversary.

Contrast this to machine learning. A neural network model is (informally) a parameterized function $f_\theta \colon \mathcal{X} \to \mathcal{Y}$ that maps input (e.g., images) $x \in \mathcal{X}$ to outputs (e.g., labels) $y \in \mathcal{Y}$. A *model extraction* attack adaptively queries the neural network to obtain a set of input/output pairs $\{(x_i, y_i)\}$ that reveals information about the weights θ. Neural networks are not constructed by design to be resistant to such attacks.

Thus, viewed appropriately, performing a model extraction attack—learning the weights θ given oracle access to the function f_θ—is a similar problem to performing a *chosen-plaintext attack* on a nontraditional "encryption" algorithm.

Given that it took the field of cryptography decades to design encryption algorithms secure against chosen-plaintext attacks, it would be deeply surprising if neural networks, where such attacks are not even considered in their design, were *not* vulnerable. Worse, the *primary* objective of cipher design is robustness against such attacks. Machine learning models, on the other hand, are primarily designed to be *accurate* at some underlying task, making the design of chosen-plaintext secure neural networks an even more challenging problem.

There are three differences separating model extraction from standard cryptanalysis that make model extraction nontrivial and interesting to study.

First, the attack success criterion differs. While a cryptographic break can be successful even without learning key bits—for example by distinguishing the algorithm from a pseudo-random function, only "total breaks" that reveal (some of) the actual model parameters θ are interesting for model extraction.

Second, the earlier analogy to keyed ciphers is imperfect. Neural networks typically take high-dimensional inputs (e.g., images) and return low-dimensional outputs (e.g., a single probability). It is almost more appropriate to make an analogy to cryptanalysis of keyed many-to-one functions, such as MACs. However, the security properties of MACs are quite different from those of machine learning models, for example second preimages are expected rather than shunned in neural networks.

Finally, and the largest difference in practice, is that machine learning models deal in fixed- or floating-point reals rather than finite field arithmetic. As such, there are many components to our attack that would be significantly simplified

given infinitely precise floating-point math, but given the realities of modern machine learning, require far more sophisticated attack techniques.

1.2   Our Results

We introduce a differential attack that is effective at performing *functionally-equivalent* neural network model extraction attacks. Our attack traces the neural network's evaluation on pairs of examples that differ in a few entries and uses this to recover the layers (analogous to the rounds of a block cipher) of a neural network one by one. To evaluate the efficacy of our attack, we formalize the definition of *fidelity* introduced in prior work [JCB+19] and quantify the degree to which a model extraction attack has succeeded:

Definition 1. *Two models f and g are (ϵ, δ)-functionally equivalent on S if*

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

Table 1 reports the results of our differential attack across a wide range of model sizes and architectures, reporting both (ε, δ)-functional equivalence on the set $S = [0, 1]^{d_0}$, the input space of the model, along with a direct measurement of max $|\theta - \hat{\theta}|$, directly measuring the error between the actual model weights θ and the extracted weights $\hat{\theta}$ (as described in Sect. 6.2).

Table 1. Efficacy of our extraction attack which is orders of magnitude more precise than prior work and for deeper neural networks orders of magnitude more query efficient. Models denoted *a-b-c* are *fully connected* neural networks with input dimension *a*, one hidden layer with *b neurons*, and *c* outputs; for formal definitions see Sect. 2. Entries denoted with a † were unable to recover the network after ten attempts.

Architecture	Parameters	Approach	Queries	$(\varepsilon, 10^{-9})$	$(\varepsilon, 0)$	max $|\theta - \hat{\theta}|$
784-32-1	25,120	[JCB+19]	$2^{18.2}$	$2^{3.2}$	$2^{4.5}$	$2^{-1.7}$
		Ours	$2^{19.2}$	$2^{-28.8}$	$2^{-27.4}$	$2^{-30.2}$
784-128-1	100,480	[JCB+19]	$2^{20.2}$	$2^{4.8}$	$2^{5.1}$	$2^{-1.8}$
		Ours	$2^{21.5}$	$2^{-26.4}$	$2^{-24.7}$	$2^{-29.4}$
10-10-10-1	210	[RK19]	2^{22}	$2^{-10.3}$	$2^{-3.4}$	2^{-12}
		Ours	$2^{16.0}$	$2^{-42.7}$	$2^{-37.98}$	2^{-36}
10-20-20-1	420	[RK19]	2^{25}	$\infty^\dagger$	$\infty^\dagger$	$\infty^\dagger$
		Ours	$2^{17.1}$	$2^{-44.6}$	$2^{-38.7}$	2^{-37}
40-20-10-10-1	1,110	Ours	$2^{17.8}$	$2^{-31.7}$	$2^{-23.4}$	$2^{-27.1}$
80-40-20-1	4,020	Ours	$2^{18.5}$	$2^{-45.5}$	$2^{-40.4}$	$2^{-39.7}$

The remainder of this paper is structured as follows. We introduce the notation, threat model, and attacker goals and assumptions used in Sect. 2. In Sect. 4 we introduce an idealized attack that extracts $(0, 0)$-functionally-equivalent neural networks assuming infinite precision arithmetic. Section 5 develops an instantiation of this attack that works in practice with finite-precision arithmetic to yield (ε, δ)-functionally equivalent attacks.

1.3 Related Work

Model extraction attacks are classified into two categories [JCB+19]: *task accuracy* extraction and *fidelity* extraction. The first paper to study task accuracy extraction [TZJ+16] introduced techniques to steal *similar* models that approximately solves the same underlying decision task on the natural data distribution, but do not necessarily match the predictions of the oracle precisely. While further work exists in this space [CCG+18,KTP+19], we instead focus on fidelity extraction where the adversary aims to faithfully reproduce the predictions of the oracle model, when it is incorrect with respect to the ground truth. Again, [TZJ+16] studied this problem and developed (what we would now call) functionally equivalent extraction for the case of completely linear models.

This attack was then extended by a theoretical result defining and giving a method for performing functionally-equivalent extraction for neural networks with one layer, assuming oracle access to the gradients [MSDH19]. A concrete implementation of this one layer attack that works in practice, handling floating point imprecision, was subsequently developed through applying finite differences to estimate the gradient [JCB+19]. Parallel work to this also extended on these results, focusing on deeper networks, but required tens to hundreds of millions of queries [RK19]; while the theoretical results extended to deep networks, the implementation in practice only extracts up to the first two layers. Our work builds on all of these four results to develop an approach that is 10^6 times more accurate, requiring 10^3 times fewer queries, and applies to larger models.

Even without query access, it is possible to steal models with just a cache side-channel [BBJP19], although with less fidelity than our attack that we introduce which are $2^{20} \times$ more precise. Other attacks target *hyperparameter* extraction—that is, extracting high-level details about the model: through what method it was trained, if it contains convolutions, or related questions [WG18]. It is further possible to steal hyperparameters with cache side channels [HDK+20].

Recent work has studied the learnability of deep neural networks with random weights in the statistical query (SQ) model [DGKP20], showing that learnability drops off exponentially with the depth of the network. This line of work does not address the *cryptographic* hardness of extraction in the non-SQ model—precisely the question addressed in this work in the empirical setting.

While not directly related to our problem, it is worth noting that we are not the first to treat neural networks as just another type of mathematical function that can be analyzed without any specific knowledge of machine learning. Shamir et al. [SSRD19] explain the existence of adversarial examples [SZS+14,BCM+13], which capture evasion attacks on machine learning classifiers, by considering an abstract model of neural networks.

In a number of places, our attack draws inspiration from the cryptanalysis of keyed block-ciphers, most prominently differential cryptanalysis [BS91]. We neither assume nor require familiarity with this field, but the informed reader may enjoy certain parallels.

2   Preliminaries

This paper studies an abstraction of neural networks as functions $f: \mathcal{X} \to \mathcal{Y}$. Our results are independent of any methods for selecting the function f (e.g., stochastic gradient descent), and are independent of any utility of the function f. As such, machine learning knowledge is neither expected nor necessary.

2.1   Notation and Definitions

Definition 2. *A k-deep neural network $f_\theta(x)$ 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_j and a nonlinear function (acting component-wise) σ:*

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

We exclusively study neural networks over $\mathcal{X} = \mathbb{R}^{d_0}$ and $\mathcal{Y} = \mathbb{R}^{d_k}$. (Until Sect. 5 we assume floating-point numbers can represent $\mathbb{R}$ exactly.)

Definition 3. *The jth* layer *of the neural network f_j is given by the affine transformation $f_j(x) = A^{(j)}x + b^{(j)}$. The* weights *$A^{(j)} \in \mathbb{R}^{d_j \times d_{j-1}}$ is a $d_j \times d_{j-1}$ matrix; the* biases *$b^{(j)} \in \mathbb{R}^{d_j}$ is a d_j-dimensional vector.*

While representing each layer f_j as a full matrix product is the most general definition of a layer, which is called *fully connected*, often layers have more structure. For example, it is common to use (discrete) *convolutions* in neural networks that operate on images. Convolutional layers take the input as a $n \times m$ matrix and convolve it with a kernel, such as a 3×3 matrix. Importantly, however, it is always possible to represent a convolution as a matrix product.

Definition 4. *The* neurons *$\{\eta_i\}_{i=1}^N$ are functions receiving an input and passing it through the* activation function *σ. There are a total of $N = \sum_{j=1}^{k-1} d_j$ neurons.*

In this paper we exclusively study the ReLU [NH10] activation function, given by $\sigma(x) = \max(x, 0)$. Our results are a fundamental consequence of the fact that ReLU neural networks are piecewise linear functions.

Definition 5. *The* architecture *of a neural network captures the structure of f: (a) the number of layers, (b) the dimensions of each layer $\{d_i\}_{i=0}^k$, and (c) any additional constraints imposed on the weights $A^{(i)}$ and biases $b^{(i)}$.*

We use the shorthand *a-b-c* neural network to denote the sizes of each dimension; for example a 10-20-5 neural network has input dimension 10, one layer with 20 neurons, and output dimension 5. This description completely characterizes the structure of f for fully connected networks. In practice, there are only a few architectures that represent most of the deployed deep learning models [ZL16], and developing new architectures is an extremely difficult and active area in research [HZRS16, SIVA17, TL19].

Definition 6. *The* parameters θ *of* f_θ *are the concrete assignments to the weights* $A^{(j)}$ *and biases* $b^{(j)}$, *obtained during the process of* training *the neural network.*

It is beyond the scope of this paper to describe the training process which produces the parameters θ: it suffices to know that the process of training is often computationally expensive and that training is a nondeterministic process, and so training the same model multiple times will give different sets of parameters.

2.2   Adversarial Goals and Resources

There are two parties in a model extraction attack: the oracle $\mathcal{O}$ who returns $f_\theta(x)$, and the adversary who generates queries x to the oracle.

Definition 7. *A* model parameter extraction attack *receives oracle access to a parameterized function* f_θ *(in our case a k-deep neural network) and the architecture of* f, *and returns a set of parameters* $\hat\theta$ *with the goal that* $f_\theta(x)$ *is as similar as possible to* $f_{\hat\theta}(x)$.

Throughout this paper we use the $\hat{\ }$ symbol to indicate an extracted parameter. For example, $\hat\theta$ refers to the extracted weights of a model θ.

There is a spectrum of similarity definitions between the extracted weights and the oracle model that prior work has studied [TZJ+16, JCB+19, KTP+19]; we focus on the setting where the adversarial advantage is defined by (ε, δ)-functionally equivalent extraction as in Definition 1.

Analogous to cryptanalysis of symmetric-key primitives, the degree to which a model extraction attack succeeds is determined by (a) the number of chosen inputs to the model, and (b) the amount of compute required.

Assumptions. We make several assumptions of the oracle $\mathcal{O}$ and the attacker's knowledge. (We believe many of these assumptions are not fundamental and can be relaxed. Removing these assumptions is left to future work.)

- **Architecture knowledge.** We require knowledge of the architecture of the neural network.
- **Full-domain inputs.** We feed arbitrary inputs from $\mathcal{X}$.
- **Complete outputs.** We receive outputs directly from the model f without further processing (e.g., by returning only the most likely class without a score).
- **Precise computations.** f is specified and evaluated using 64-bit floating-point arithmetic.
- **Scalar outputs.** Without loss of generality we require the output dimensionality is 1, i.e., $\mathcal{Y} = \mathbb{R}$.
- **ReLU Activations.** All activation functions (σ's) are ReLU's.[1]

---

[1] This is the only assumption *fundamental* to our work. Switching to any activation that is not piecewise linear would prevent our attack. However, as mentioned, all state-of-the-art models use exclusively (piecewise linear generalizations of) the ReLU activation function [SIVA17, TL19].
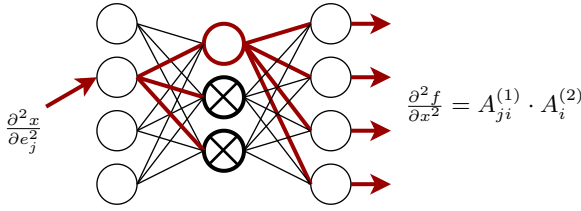
3 Overview of the Differential Attack

Fig. 1. A schematic of our extraction attack on a 1-deep neural network. Let x be an input that causes exactly one neuron to have value zero. The second differential becomes zero at all other neurons—because they remain either fully-inactive or fully-active. Therefore the value of this differential is equal to the product of the weight going into the neuron at its critical point and the weight going out of this neuron.

Given oracle access to the function f_θ, we can estimate ∂f_θ through finite differences along arbitrary directions. For simple linear functions defined by $f(x) = a \cdot x + b$, its directional derivative satisfies $\frac{\partial f}{\partial e_i} \equiv a_i$, where e_i is the basis vector and a_i is the ith entry of the vector a, allowing direct recovery of its weights through querying on these well-chosen inputs.

In the case of deep neural networks, we consider second partial directional derivatives. ReLU neural networks are piecewise linear functions with $\frac{\partial^2 f}{\partial x^2} \equiv 0$ almost everywhere, except when the function has some neuron η_j at the boundary between the negative and positive region (i.e., is at its *critical point*). We show that the value of the partial derivative $\frac{\partial^2 f}{\partial e_i^2}$ evaluated at a point x so that neuron η_j is at such a critical point actually directly reveals the weight $T(A_{i,j}^{(1)})$ for some transform T that is invertible—and therefore the adversary can learn $A_{i,j}^{(1)}$. By repeating this attack along all basis vectors e_i and for all neurons η_j we can recover the complete matrix $A^{(1)}$. Once we have extracted the first layer's weights, we are able to "peel off" that layer and re-mount our attack on the second layer of the neural network, repeating to the final layer. There are three core technical difficulties to our attack:

Recovering the neuron signs. For each neuron η, our attack does not exactly recover $A_i^{(l)}$, the ith row of $A^{(l)}$, but instead a scalar multiple $v = \alpha \cdot A_i^{(l)}$. While losing a constant $\alpha > 0$ keeps the neural network in the same equivalence class, the sign of α is important and we must distinguish between the weight vector $A_i^{(l)}$ and $-A_i^{(l)}$. We construct two approaches that solve this problem, but in the general case we require exponential work (but a linear number of queries).

Controlling inner-layer hidden state. On the first layer, we can directly compute the derivative entry-by-entry, measuring $\frac{\partial^2 f}{\partial e_i^2}$ for each standard basis vector e_i in order to recover $A_{ij}^{(1)}$. Deeper in the network, we can not move along standard basis vector vectors. Worse, for each input x on average half of the neurons are in the negative region and thus their output is identically 0; when this happens it is not possible to learn the weight along edges with value zero. Thus we are required to develop techniques to elicit behavior from every neuron, and techniques to cluster together partial recoveries of each row of $A_i^{(l)}$ to form a complete recovery.

Handling floating-point imprecision. Implementing our attack in practice with finite precision neural networks introduces additional complexity. In order to estimate the second partial derivative, we require querying on inputs that differ by only a small amount, reducing the precision of the extracted first weight matrix to twenty bits, or roughly 10^{-6}. This error of 10^{-6} is not large to begin with, but this error impacts our ability to recover the next layer, compounding multiplicatively the deeper we go in the network. Already in the second layer, the error is magnified to 10^{-4}, which can completely prevent reconstruction for the third layer: our predicted view of the hidden state is sufficiently different from the actual hidden state that our attack fails completely. We resolve this through two means. First, we introduce numerically stable methods assuming that all prior layers have been extracted to high precision. Second, we develop a precision-refinement technique that takes a prefix of the first $j \leq k$ layers of a neural network extracted to n bits of precision and returns the j-deep model extracted to $2n$ bits of precision (up to floating-point tolerance).

4   Idealized Differential Extraction Attack

We now introduce our $(0,0)$-functionally-equivalent model extraction attack that assumes infinite precision arithmetic and recovers completely functionally equivalent models. Recall our attack assumptions (Sect. 2.2); using these, we present our attack beginning with two "reduced-round" attacks on 0-deep (Sect. 4.1) and 1-deep (Sect. 4.2) neural networks, and then proceeding to k-deep extraction for contractive (Sect. 4.3) and expansive (Sect. 4.4) neural networks. Section 5 refines this idealized attack to work with finite precision.

4.1   Zero-Deep Neural Network Extraction

Zero-deep neural networks are linear functions $f(x) \equiv A^{(1)} \cdot x + b^{(1)}$. Querying d_0 linearly independent suffices to extract f by solving the resulting linear system.

However let us view this problem differently, to illuminate our attack strategy for deeper networks. Consider the parallel evaluations $f(x)$ and $f(x + \delta)$, with

$$f(x + \delta) - f(x) = A^{(1)} \cdot (x + \delta) - A^{(1)} \cdot x = A^{(1)} \cdot \delta.$$

If $\delta = e_i$, the ith standard basis vector of $\mathbb{R}^{d_0}$ (e.g., $e_2 = \begin{bmatrix} 0\ 1\ 0\ 0 \dots 0 \end{bmatrix}$), then

$$f(x + \delta) - f(x) = A^{(1)} \cdot \delta = A_i^{(1)}.$$

This allows us to directly read off the weights of $A^{(1)}$. Put differently, we perform finite differences to estimate the gradient of f, given by $\nabla_x f(x) \equiv A^{(1)}$.

4.2   One-Deep Neural Network Extraction

Many of the important problems that complicate deep neural network extraction begin to arise at 1-deep neural networks. Because the function is no longer completely linear, we require multiple phases to recover the network completely. To do so, we will proceed layer-by-layer, extracting the first layer, and then use the 0-deep neural network attack to recover the second layer.
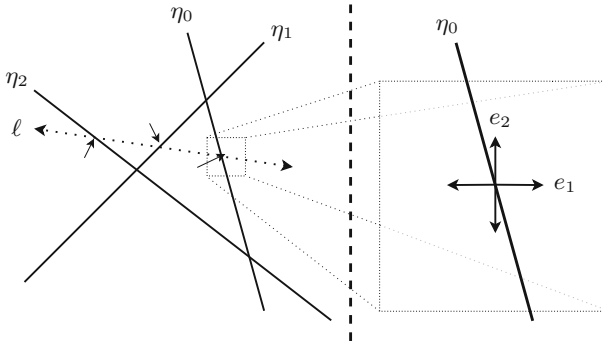

Fig. 2. (left) Geometry of a 1-deep neural network. The three solid line corresponds to "critical hyperplanes" of neurons. We identify one *witness* to each neuron with binary search on the dotted line ℓ. **(right)** For each discovered critical point, we compute the second partial derivative along axis e_1 and e_2 to compute the angle of the hyperplane.

For the remainder of this paper it will be useful to have two distinct mental models of the problem at hand. First is the *symbolic* view shown previously in Fig. 1. This view directly studies the flow of information through the neural networks, represented as an alternating sequence of linear layers and non-linear transformations. This view helps understanding the algebraic steps of our attack.

The second is the *geometric* view. Because neural networks operate over the real vector space, they can be visualized by plotting two dimensional slices of the landscape [MSDH19]. Figure 2 (left) contains an example of such a figure. Each solid black line corresponds to a change in gradient induced in the space by a neuron changing sign from positive to negative (or vice versa)—ignoring for now the remaining lines. The problem of neural network extraction corresponds to recovering the locations and angles of these neuron-induced *hyperplanes*: in general with input dimension d_0, the planes have dimension $d_0 - 1$.

Definition 8. *The function that computes the first j layers (up to and including f_j but not including σ) of f is denoted as $f_{1..j}$. In particular, $f = f_{1..k}$.*

Definition 9. *The* hidden state *at layer j is the output of the function $f_{1..j}$, before applying the nonlinear transformation σ.*

Layer f_j is a linear transformation of the $(j-1)$st hidden state after σ.

Definition 10. $\mathcal{V}(\eta; x)$ *denotes the input to neuron η (before applying σ) when evaluated at x. $\mathcal{L}(\eta)$ denotes the layer of neuron η. The first layer starts at 1.*

Definition 11. *A neuron η is at a* critical point *when $\mathcal{V}(\eta; x) = 0$. We refer to this input x as a* witness *to the fact that η is at a critical point, denoted by $x \in \mathcal{W}(\eta)$. If $\mathcal{V}(\eta; x) > 0$ then η is* active, *and otherwise* inactive.

In Fig. 2 the locations of these critical points correspond exactly to the solid black lines drawn through the plane. Observe that because we restrict ourselves to ReLU neural networks, the function f is piecewise linear and infinitely differentiable almost everywhere. The gradient $\nabla_x f(x)$ is well defined at all points x except when there exists a neuron that is at its critical point.

Extracting the rows of $A^{(1)}$ up to sign. Functionally, the attack as presented in this subsection has appeared previously in the literature [MSDH19, JCB+19]. By framing it differently, our attack will be extensible to deeper networks.

Assume we were given a witness $x^* \in \mathcal{W}(\eta_j)$ that caused neuron η_j to be at its critical point (i.e., its value is identically zero). Because we are using the ReLU activation function, this is the point at which that neuron is currently "inactive" (i.e., is not contributing to the output of the classifier) but would become "active" (i.e., contributing to the output) if it becomes slightly positive. Further assume that *only* this neuron η_j is at its critical point, and that for all others neurons $\eta \neq \eta_j$ we have $|\mathcal{V}(\eta, x_j)| > \delta$ for a constant $\delta > 0$.

Consider two parallel executions of the neural network on pairs of examples. Begin by defining e_i as the standard basis vectors of $\mathcal{X} = \mathbb{R}^N$. By querying on the two pairs of inputs $(x^*, x^* + \epsilon e_i)$ and $(x^*, x^* - \epsilon e_i)$ we can estimate

$$\alpha^i_+ = \left.\frac{\partial f(x)}{\partial e_i}\right|_{x=x^*+\epsilon e_1} \quad \text{and} \quad \alpha^i_- = \left.\frac{\partial f(x)}{\partial e_i}\right|_{x=x^*-\epsilon e_1}$$

through finite differences.

Consider the quantity $|\alpha_+ - \alpha_-|$. Because x^* induces a critical point of η_j, exactly one of $\{\alpha_+, \alpha_-\}$ will have the neuron η_j in its active regime and the other will have η_j in its inactive regime. If no two columns of $A^{(1)}$ are collinear, then as long as $\epsilon < \frac{\delta}{\sum_{i,j}|A^{(1)}_{i,j}|}$, we are guaranteed that all other neurons in the neural network will remain in the same state as before—either active or inactive. Therefore, if we compute the difference $|\alpha^i_+ - \alpha^i_-|$, the gradient information flowing into and out of all other neurons will cancel and we will be left with just the gradient information flowing along the edge from the input coordinate i to neuron η_j to the output. Concretely, we can write the 1-deep neural network as

$$f(x) = A^{(2)}\text{ReLU}(A^{(1)}x + b^{(1)}) + b^{(2)}.$$

and so either $\alpha^i_+ - \alpha^i_- = A^{(1)}_{j,i} \cdot A^{(2)}$ or $\alpha^i_- - \alpha^i_+ = A^{(1)}_{j,i} \cdot A^{(2)}$. However, if we repeat the above procedure on a new basis vector e_k then either $\alpha^k_+ - \alpha^k_- = A^{(1)}_{j,k} \cdot A^{(2)}$ or $\alpha^k_- - \alpha^k_+ = A^{(1)}_{j,k} \cdot A^{(2)}$ will hold. Crucially, whichever of the two relations that holds for along coordinate i will be the same relation that holds on coordinate k. Therefore we can divide out $A^{(2)}$ to obtain the ratio of pairs of weights

$$\frac{\alpha^k_+ - \alpha^k_-}{\alpha^i_+ - \alpha^i_-} = \frac{A^{(1)}_{j,k}}{A^{(1)}_{j,i}}.$$

This allows us to compute every row of $A^{(1)}$ up to a single scalar c_j. Further, we can compute $b^{(1)}_j = -\hat{A}^{(1)}_j \cdot x^*$ (again, up to a scaling factor) because we know that x^* induces a critical point on neuron η_j and so its value is zero.

Observe that the magnitude of c_j is unimportant. We can always push a constant $c > 0$ through to the weight matrix $A^{(2)}$ and have an a functionally equivalent result. However, the *sign* of c_j does matter.

Extracting row signs. Consider a single witness x_i for an arbitrary neuron η_i. Let $h = f_1(x)$, so that at least one element of h is identically zero. If we assume that $A^{(1)}$ is contractive (Sect. 4.4 studies non-contractive networks) then we can find a preimage x to any vector h. In particular, let e_i be the unit vector in the space $\mathbb{R}^{d_1}$. Then we can compute a preimage x_+ so that $\hat{f}_1(x_+) = h + e_i$, and a preimage x_- so that $\hat{f}_1(x_-) = h - e_i$.

Because x_i is a witness to neuron η_i being at its critical point, we will have that either $f(x_+) = f(x_i)$ or $f(x_-) = f(x_i)$. Exactly one of these equalities is true because $\sigma(h - e_i) = \sigma(h)$, but $\sigma(h + e_i) \neq \sigma(h)$ when $h_i = 0$. Therefore if the second equality holds true, then we know that our extracted guess of the ith row has the correct sign. However, if the first equality holds true, then our extracted guess of the ith row has the incorrect sign, and so we invert it (along with the bias $b^{(1)}_i$). We repeat this procedure with a critical point for every neuron η_i to completely recover the signs for the full first layer.

Finding witnesses to critical points. It only remains to show how to find witnesses $x^* \in \mathcal{W}(\eta)$ for each neuron η on the first layer. We choose a random line in input space (the dashed line in Fig. 2, left), and search along it for nonlinearities in the partial derivative. Any nonlinearity must have resulted from a ReLU changing signs, and locating the specific location where the ReLU changes signs will give us a critical point. We do this by binary search.

To begin, we take a random initial point $x_0, v \in \mathbb{R}^{d_0}$ together with a large range T. We perform a binary search for nonlinearities in $f(x_0 + tv)$ for $t \in [-T, T]$. That is, for a given interval $[t_0, t_1]$, we know a critical point exists in the interval if $\frac{\partial f(x+tv)}{\partial v}\big|_{t=t_0} \neq \frac{\partial f(x+tv)}{\partial v}\big|_{t=t_1}$. If these quantities are equal, we do not search the interval, otherwise we continue with the binary search.

Extracting the second layer. Once we have fully recovered the first layer weights, we can "peel off" the weight matrix $A^{(1)}$ and bias $b^{(1)}$ and we are left with extracting the final linear layer, which reduces to 0-deep extraction.

4.3 k-Deep Contractive Neural Networks

Extending the above attack to deep neural networks has several complications that prior work was unable to resolve efficiently; we address them one at a time.

Critical Points Can Occur Due to ReLUs on Different Layers. Because 1-deep networks have only one layer, all ReLUs occur on that layer. Therefore all critical points found during search will correspond to a neuron on that layer. For k-deep networks this is not true, and if we want to begin by extracting the first layer we will have to remove non-first layer critical points. (And, in general, to extract layer j, we will have to remove non-layer-j critical points.)

The Weight Recovery Procedure Requires Complete Control of the Input. In order to be able to directly read off the weights, we query the network on basis vectors e_i. Achieving this is not always possible for deep networks, and we must account for the fact that we may only be able to query on non-orthogonal directions.

Recovering Row Signs Requires Computing the Preimage of Arbitrary Hidden States. Our row-sign procedure requires that we be able to invert $A^{(1)}$, which in general implies we need to develop a method to compute a preimage of $f_{1..j}$.

4.3.1   Extracting Layer-1 Weights with Unknown Critical Point Layers

Suppose we had a function $\mathcal{C}_0(f) = \{x_i\}_{i=1}^{M}$ that returns at least one critical points for every neuron in the first layer (implying $M \geq d_1$), but never returns critical points for any deeper layer. We claim that the exact differential attack from above still correctly recovers the first layer of a deep neural network.

We make the following observation. Let $x^* \notin \bigcup_{\eta_i} \mathcal{W}(\eta_i)$ be an input that is a witness to no critical point, i.e., $|\mathcal{V}(\eta_i; x^*)| > \epsilon > 0$. Define f_{local} as the function so that for a sufficiently small region we have that $f_{\text{local}} \equiv f$, that is,

$$f_{\text{local}}(x) = \big(A^{(k+1)} \cdots (I^{(2)}(A^{(2)}(I^{(1)}(A^{(1)}x + b^{(1)})) + b^{(2)})) + \dots\big) + b^{(k+1)}$$
$$= A^{(k+1)} I^{(k)} A^{(k)} \cdots I^{(2)} A^{(2)} I^{(1)} A^{(1)} x + \beta$$
$$= \Gamma x + \beta$$

Here, $I^{(j)}$ are 0–1 diagonal matrices with a 0 on the diagonal when the neuron is inactive and 1 on the diagonal when the neuron is active:

$$I_{n,n}^{(j)} = \begin{cases} 1 & \text{if } \mathcal{V}(\eta_n; x) > 0 \\ 0 & \text{otherwise} \end{cases}$$

where η_n is the nth neuron on the first layer. Importantly, observe that each $I^{(j)}$ is a constant as long as x is sufficiently close to x^*. While β is unknown, as long as we make only gradient queries $\partial f_{\text{local}}$ its value is unimportant. This observation so far follows from the definition of piecewise linearity.

Consider now some input that is a witness to exactly one critical point on neuron η^*. Formally, $x^* \in \mathcal{W}(\eta^*)$, but $x^* \notin \bigcup_{\eta_j \neq \eta^*} \mathcal{W}(\eta_j; x^*)$. Then

$$f_{\text{local}}(x) = A^{(k+1)} I^{(k)} A^{(k)} \cdots I^{(2)} A^{(2)} I^{(1)}(x) A^{(1)} x + \beta(x)$$

where again $I^{(j)}$ are 0–1 matrices, but except that now, $I^{(1)}$ (and only $I^{(1)}$) is a function of x returning a 0–1 diagonal matrix that has one of two values, depending on the value of $\mathcal{V}(\eta^*; x) > 0$. Therefore we can no longer collapse the matrix product into one matrix Γ but instead can only obtain

$$f_{\text{local}}(x) = \Gamma I^{(1)}(x) A^{(1)} x + \beta(x).$$

But this is exactly the case we have already solved for 1-deep neural network weight recovery: it is equivalent to the statement $f_{\text{local}}(x) = \Gamma \sigma(A^{(1)} x + b^{(1)}) + \beta_2$, and so by dividing out Γ exactly as before we can recover the ratios of $A^{(1)}_{i,j}$.

Finding first-layer critical points. Assume we are given a set of inputs $S = \{x_i\}$ so that each x_i is a witness to neuron η_{x_i}, with η_{x_i} unknown. By the coupon collector's argument (assuming uniformity), for $|S| \gg N \log N$, where N is the total number of neurons, we will have at least *two* witnesses to every neuron η.

Without loss of generality let $x_0, x_1 \in \mathcal{W}(\eta)$ be witnesses to the same neuron η on the first layer, i.e, that $\mathcal{V}(\eta; x_0) = \mathcal{V}(\eta; x_1) = 0$. Then, performing the weight recovery procedure beginning from each of these witnesses (through finite differences) will yield the correct weight vector $A^{(1)}_j$ up to a scalar.

Typically elements of S will *not* be witnesses to neurons on the first layer. Without loss of generality let x_2 and x_3 be witnesses to any neuron on a deeper layer. We claim that we will be able to detect these error cases: the outputs of the extraction algorithm will appear to be random and uncorrelated. Informally speaking, because we are running an attack designed to extract first-layer neurons on a neuron actually on a later layer, it is exceedingly unlikely that the attack would, by chance, give consistent results when run on x_2 and x_3 (or any arbitrary pair of neurons).

Formally, let $h_2 = f_1(x_2)$ and $h_3 = f_1(x_3)$. With high probability, $\text{sign}(h_2) \neq \text{sign}(h_3)$. Therefore, when executing the extraction procedure on x_2 we compute over the function $\Gamma_1 I^{(1)}(x_2) A^{(1)} x + \beta_1$, whereas extracting on x_3 computes over $\Gamma_2 I^{(1)}(x_3) A^{(1)} x + \beta_2$. Because $\Gamma_1 \neq \Gamma_2$, this will give inconsistent results.

Therefore our first layer weight recovery procedure is as follows. For all inputs $x_i \in S$ run the weight recovery procedure to recover the unit-length normal vector to each critical hyperplane. We should expect to see a large number of vectors only once (because they were the result of running the extraction of a layer 2 or greater neuron), and a small number of vectors that appear duplicated (because they were the result of successful extraction on the first layer). Given the first layer, we can reduce the neural network from a k-deep neural network to a $(k-1)$-deep neural network and repeat the attack. We must resolve two difficulties, however, discussed in the following two subsections.

4.3.2   Extracting Hidden Layer Weights with Unknown Critical Points

When extracting the first layer weight matrix, we were able to compute $\frac{\partial^2 f}{\partial e_1 \partial e_j}$ for each input basis vectors e_i, allowing us to "read off" the ratios of the weights on the first layer directly from the partial derivatives. However, for deeper layers, it is nontrivial to exactly control the hidden layers and change just one coordinate in order to perform finite differences.[2] Let j denote the current layer we are extracting. Begin by sampling $d_j + 1$ directions $\delta_i \sim \mathcal{N}(0, \epsilon I_{d_0}) \in \mathcal{X}$ and let

$$\{y_i\} = \left\{ \frac{\partial^2 f(x)}{\partial \delta_1 \partial \delta_i} \bigg|_{x=x^*} \right\}_{i=1}^{d_j+1}.$$

From here we can construct a system of equations: let $h_i = \sigma(f_{1..j-1}(x + \delta_i))$ and solve for the vector w such that $h_i \cdot w = y_i$.

As before, we run the weight recovery procedure assuming that each witness corresponds to a critical point on the correct layer. Witnesses that correspond to neurons on incorrect layers will give uncorrelated errors that can be discarded.

Unifying partial solutions. The above algorithm overlooks one important problem. For a given critical point x^*, the hidden vector obtained from $f_{1..j}(x^*)$ is likely to have several (on average, half) neurons that are negative, and therefore $\sigma(f_{1..j}(x^*))$ and any $\sigma(f_{1..j}(x^* + \delta_i))$ will have neurons that are identically zero. This makes it impossible to recover the complete weight vector from just one application of least squares—it is only possible to compute the weights for those entries that are non-zero. One solution would be to search for a witness x^* such that component-wise $f_{1..j}(x^*) \geq 0$; however doing this is not possible in general, and so we do not consider this option further.

Instead, we combine together multiple attempts at extracting the weights through a *unification* procedure. If x_1 and x_2 are witnesses to critical points for the same neuron, and the partial vector $f_{1..j}(x_1)$ has entries $t_1 \subset \{1, \ldots, d_j\}$ and the partial vector $f_{1..j}(x_2)$ has entries $t_2 \subset \{1, \ldots, d_j\}$ defined, then it is possible to recover the ratios for all entries $t_1 \cup t_2$ by unifying together the two partial solutions as long as $t_1 \cap t_2$ is non-empty as follows.

Let r_i denote the extracted weight vector on witness x_1 with entries at locations $t_1 \subset \{1, \ldots, d_j\}$ (respectively, r_2 at x_2 with locations at t_2). Because the two vectors correspond to the solution for the same row of the weight matrix $A_i^{(j)}$, the vectors r_1 and r_2 must be consistent on $t_1 \cap t_2$. Therefore, we will have that $r_1[t_1 \cap t_2] = c \cdot r_2[t_1 \cap t_2]$ for a scalar $c \neq 0$. As long as $t_1 \cap t_2 \neq \emptyset$ we can compute the appropriate constant c and then recover the weight vector $r_{1,2}$ with entries at positions $t_1 \cup t_2$.

Observe that this procedure also allows us to *check* whether or not x_1 and x_2 are witnesses to the same neuron n reaching its critical point. If $|t_1 \cap t_2| \geq \gamma$, then as long as there do not exist two rows of $A^{(j)}$ that have $\gamma + 1$ entries that are scalar multiples of each other, there will be a unique solution that merges the

---

[2] For the expansive networks we will discuss in Sect. 4.4 it is actually impossible; therefore this section introduces the most general method.

two partial solutions together. If the unification procedure above fails—because there does not exist a single scalar c so that $c \cdot r_1[t_1 \cap t_2] = r_2[t_1 \cap t_2]$—then x_1 and x_2 are not witnesses to the same neuron being at a critical point.

4.3.3 Recovering Row Signs in Deep Networks

The 1-layer contractive sign recovery procedure can still apply to "sufficiently contractive" neural network where at layer j there exists an $\epsilon > 0$ so that for all $h \in \mathbb{R}^{d_j}$ with $\|h\| < \epsilon$ there exists a preimage x with $f_{1..j}(x) = h$. If a neural network is sufficiently contractive it is easy to see that the prior described attack will work (because we have assumed the necessary success criteria).

In the case of 1-deep networks, it suffices for $d_1 \leq d_0$ and $A^{(1)}$ to be onto as described. In general it is necessary that $d_k \leq d_{k-1} \leq \cdots \leq d_1 \leq d_0$ but it is not sufficient, even if every layer $A^{(i)}$ were an onto map. Because there is a ReLU activation after every hidden layer, it is not possible to send negative values *into* the second layer f_j when computing the preimage.

Therefore, in order to find a preimage of $h_i \in \mathbb{R}^{d_i}$ we must be more careful in how we mount our attack: instead of just searching for $h_{i-1} \in \mathbb{R}^{d_{i-1}}$ so that $f_{i-1}(h_{i-1}) = h_i$ we must additionally require that component-wise $h_{i-1} \geq 0$. This ensures that we will be able to recursively compute $h_{i-2} \to h_{i-1}$ and by induction compute $x \in \mathcal{X}$ such that $f_{1..j}(x) = h_j$.

It is simple to test if a network is sufficiently contractive without any queries: try the above method to find a preimage x; if this fails, abort and attempt the following (more expensive) attack procedure. Otherwise it is contractive.
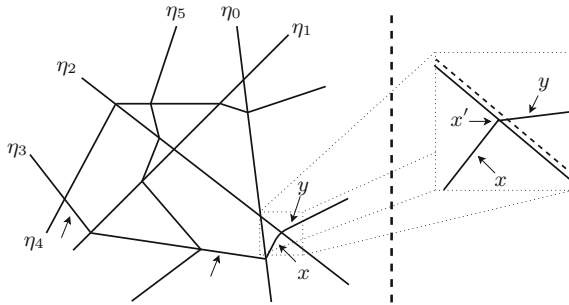
4.4 k-Deep Expansive Neural Networks

Fig. 3. (left) Geometry of a k-deep neural network, following [RK19]. Critical hyperplanes induced from neuron η_0, η_1, η_2 are on the first layer and are linear. Critical hyperplanes induced from neurons η_3, η_4 are on the second layer and are "bent" by neurons on the first layer. The critical hyperplane induced from neuron η_5 is a neuron on the third layer and is bent by neurons on the prior two layers. **(right)** Diagram of the hyperplane following procedure. Given an initial witness to a critical point x, follow the hyperplane to the double-critical point x'. To find where it goes next, perform binary search along the dashed line and find the witness y.

While most small neural networks are contractive, in practice almost all interesting neural networks are expansive: the number of neurons on some intermediate layer is larger than the number of inputs to that layer. Almost all of the prior methods still apply in this setting, with one exception: the column sign recovery procedure. Thus, we are required to develop a new strategy.

Recovering signs of the last layer. Observe that sign information is not lost for the final layer: because there is no ReLU activation and we can directly solve for the weights with least squares, we do not lose sign information.

Recovering signs on the second-to-last layer. Suppose we had extracted completely the function $\hat{f}_{1..k-1}$ (the third to last layer), and further had extracted the weights $\hat{A}^{(k)}$ and biases $\hat{b}^{(k)}$ up to sign of the rows. There are three unknown quantities remaining: a sign vector $s \in \{-1, 1\}^{d_k}$, $\hat{A}^{(k+1)}$ and $\hat{b}^{(k+1)}$. Suppose we were given $S \subset \mathcal{X}$ so that $|S| > d_k$. Then it would be possible to solve for all three unknown simultaneously through brute force.

Definition 12. *Let $v \odot M = M'$ denote multiplying rows of matrix $M \in \mathbb{R}^{a \times b}$ by the corresponding coordinate from $v \in \mathbb{R}^a$. Thus, $M'_{ij} = M_{ij} \cdot v_i$.*

Let $h_i = \sigma(f_{1..k-1}(x_i))$. Enumerate all 2^{d_k} assignments of s and compute $g_i = \sigma((s \odot \hat{A}^{(k)})h_i + (s \odot \hat{b}^{(k)}))$. We know that if we guessed the sign vector s correctly, then there would exist a solution to the system of equations $v \cdot g_i + b = f(x_i)$. This is the zero-deep extraction problem and solving it efficiently requires just a single call to least squares. This allows us to—through brute forcing the sign bits—completely recover both the signs of the second-to-last layer as well as the values (and signs) of the final layer.

Unfortunately, this procedure does not scale to recover the signs of layer $k-1$ and earlier. It relies on the existence of an efficient testing procedure (namely, least squares) to solve the final layer. If we attempted this brute-force strategy at layer $k-3$ in order to test if our sign assignment was correct, we would need to run the complete layer $k-2$ extraction procedure, thus incurring an exponential number of queries to the oracle.

However, we can use this idea in order to still recover signs even at earlier layers in the network with only a linear number of queries (but still exponential work in the width of the hidden layers).

Recovering signs of arbitrary hidden layers. Assume that we are given a collection of examples $\{x_i\} \subset \mathcal{W}(\eta)$ for some neuron η that is on the layer after we extracted so far: $\mathcal{L}(\eta) = j + 1$. Then we would know that there should exist a single unknown vector v and bias b such that $f_j(x_i) \cdot v + b = 0$ for all x_i.

This gives us an efficient procedure to test whether or not a given sign assignment on layer j is correct. As before, we enumerate all possible sign assignments and then check if we can recover such a vector v. If so, the assignment is correct; if not, it is wrong. It only remains left to show how to implement this procedure to obtain such a collection of inputs $\{x_i\}$.

4.4.1   The Polytope Boundary Projection Algorithm

Definition 13. *The* layer j polytope *containing x is the set of points $\{x + \delta\}$ so that $sign(\mathcal{V}(\eta; x)) = sign(\mathcal{V}(\eta; x + \delta))$ for all $\mathcal{L}(\eta) \leq j$.*

Observe that the layer j polytope around x is an open, convex set, as long as x is not a witness to a critical point. In Fig. 3, each enclosed region is a layer-k polytope and the triangle formed by $\eta_0, \eta_1,$ and η_2 is a layer-$(k - 1)$ polytope.

Given an input x and direction Δ, we can compute the distance α so that the value $x' = x + \alpha\Delta$ is at the boundary of the polytope defined by layers 1 to k. That is, starting from x traveling along direction Δ we stop the first time a neuron on layer j or earlier reaches a critical point. Formally, we define

$$\text{Proj}_{1..j}(x, \Delta) = \min_{\alpha \geq 0} \{\alpha : \exists \eta \text{ s.t. } \mathcal{L}(\eta) \leq j \wedge \mathcal{V}(\eta; x + \alpha\Delta) = 0\}$$

We only ever compute $\text{Proj}_{1..j}$ when we have extracted the neural network up to layer j. Thus we perform the computation with respect to the extracted function $\hat{f}$ and neuron-value function $\hat{\mathcal{V}}$, and so computing this function requires no queries to the oracle. In practice we solve for α via binary search.

4.4.2   Identifying a Single Next-Layer Witness

Given the correctly extracted network $\hat{f}_{1..j-1}$ and the weights (up to sign) of layer $j - 1$, our sign extraction procedure requires *some* witness to a critical point on layer j. We begin by performing our standard binary search sweep to find a collection $S \subset \mathcal{X}$, each of which is a witness to some neuron on an unknown layer. It is simple to filter out critical points on layers $j - 1$ or earlier by checking if any of $\hat{\mathcal{V}}(\eta; x) = 0$ for $\mathcal{L}(\eta) \leq j - 1$. Even though we have not solved for the sign of layer j, it is still possible to compute whether or not they are at a critical point because critical points of $\hat{A}^{(j)}$ are critical points of $-\hat{A}^{(j)}$. This removes any witnesses to critical points on layer j or lower.

Now we must filter out any critical points on layers strictly later than j. Let $x^* \in \mathcal{W}(\eta^*)$ denote a potential witness that is on layer j or later (having already filtered out critical points on layers $j - 1$ or earlier). Through finite differences, estimate $g = \pm\nabla_x f(x)$ evaluated at $x = x^*$. Choose any random vector r perpendicular to g, and therefore parallel to the critical hyperplane. Let $\alpha = \text{Proj}_{1..j}(x^*, r)$. If it turns out that x^* is a witness to a critical point on layer j then for all $\epsilon < \alpha$ we must have that $x^* + \epsilon r \in \mathcal{W}(\eta^*)$. Importantly, we also have the converse: with high probability for $\delta > \alpha$ we have that $x^* + \delta r \notin \mathcal{W}(\eta^*)$. However, observe that if x^* is *not* a witness to a neuron on layer j then one of these two conditions will be false. We have already ruled out witnesses on *earlier* neuron, so if x^* is a witness to a *later* neuron on layer $j' > j$ then it is unlikely that the layer-j' polytope is the same shape as the layer-j polytope, and therefore we will discover this fact. In the case that the two polytopes are actually identical, we can mount the following attack and if it fails we know that our initial input was on the wrong layer.

4.4.3   Recovering Multiple Witnesses for the Same Neuron

The above procedure yields a single witness $x^* \in \mathcal{W}(\eta^*)$ so that $\mathcal{L}(\eta^*) = j + 1$. We expand this to a collection of witnesses W where all $x \in W$ have $x \in \mathcal{W}(\eta^*)$, requiring the set to be *diverse*:

Definition 14. *A collection of inputs S is* fully diverse *at layer j if for all η with $\mathcal{L}(\eta) = j$ and for $s \in \{-1, 1\}$ there exists $x \in S$ such that $s \cdot \mathcal{V}(\eta; x) \geq 0$.*

Informally, being diverse at layer j means that if we consider the projection onto the space of layer j (by computing $f_{1..j}(x)$ for $x \in S$), for every neuron η there will be at least one input $x_+ \in S$ that the neuron is positive, and at least one input $x_- \in S$ so that the neuron is negative.

Our procedure is as follows. Let n be normal to the hyperplane x^* is on. Choose some r with $r \cdot n = 0$ and let $\alpha = \mathrm{Proj}_{1..j}(x^*, r)$ to define $x' = x^* + \alpha r$ as a point on the layer-j polytope boundary. In particular, this implies that we still have that $x' \in \mathcal{W}(\eta^*)$ (because r is perpendicular to n) but also $x' \in \mathcal{W}(\eta_u)$ for some neuron $\mathcal{L}(\eta_u) < j$ (by construction of α). Call this input x' the double-critical point (because it is a witness to two critical points simultaneously).

From this point x', we would like to obtain a new point y so that we still have $y \in \mathcal{W}(\eta^*)$, but that also y is on the other side of the neuron η_u, i.e., $\mathrm{sign}(\mathcal{V}(\eta_u; x^*)) \neq \mathrm{sign}(\mathcal{V}(\eta_u; y))$. Figure 3 (right) gives a diagram of this process. In order to follow x^* along its path, we first need to find it a critical point on the new hyperplane, having just been bent by the neuron η_u. We achieve this by performing a critical-point search starting ϵ-far away from, and parallel to, the hyperplane from neuron η_u (the dashed line in Fig. 3). This returns a point y from where we can continue the hyperplane following procedure.

The geometric view hurts us here: because the diagram is a two-dimensional projection, it appears that from the critical point y there are only two directions we can travel in: *away* from x' or *towards* x'. Traveling away is preferable—traveling towards x' will not help us construct a fully diverse set of inputs.

However, a d_0-dimensional input space has in general (d_0-1) dimensions that remain on the neuron η^*. We defer to Sect. 5.5.1 an efficient method for selecting the continuation direction. For now, observe that choosing a random direction will eventually succeed at constructing a fully-diverse set, but is extremely inefficient: there exist better strategies than choosing the next direction.

4.4.4   Brute Force Recovery

Given this collection S, we can now—through brute force work—recover the correct sign assignment as follows. As described above, compute a fully diverse set of inputs $\{x_i\}$ and define $h_i = f_{1..j}(x_i)$. Then, for all possible 2^{d_j} assignments of signs $s \in \{-1, 1\}^{d_j}$, compute the guessed weight matrix $\hat{A}_s^{(j)} = s \odot \hat{A}^{(j)}$.

If we guess the correct vector s, then we will be able to compute $\hat{h}_i = \sigma(\hat{A}_v^{(j)} h_i + \hat{b}_v^{(j)}) = \sigma(\hat{A}_v^{(j)} f_{1..j-1}(x_i) + \hat{b}_v^{(j)})$ for each $x_i \in S$. Finally, we know that there will exist a vector $w \neq \mathbf{0}$ and bias $\hat{b}$ such that for all h_i we have $\hat{h}_i w + b = 0$. As before, if our guess of s is wrong, then with overwhelming

probability there will not exist a valid linear transformation w, b. Thus we can recover sign with a linear number of queries and exponential work.

5   Instantiating the Differential Attack in Practice

The above idealized attack would efficiently extract neural network models but suffers from two problems. First, many of the algorithms are not numerically stable and introduce small errors in the extracted weights. Because errors in layer i compound and cause further errors at layers $j > i$, it is necessary to keep errors to a minimum. Second, the attack requires more chosen-inputs than is necessary; we develop new algorithms that require fewer queries or re-use previously-queried samples.

Reading this section. Each sub-section that follows is independent from the surrounding sub-sections and modifies algorithms introduced in Sect. 4. For brevity, we assume complete knowledge of the original algorithm and share the same notation. Readers may find it helpful to review the original algorithm before proceeding to each subsection.

5.1   Improving Precision of Extracted Layers

Given a precisely extracted neural network up to layer j so that $\hat{f}_{1..j-1}$ is functionally equivalent to $f_{1..j-1}$, but so that weights $\hat{A}^{(j)}$ and biases $\hat{b}^{(j)}$ are imprecisely extracted due to imprecision in the extraction attack, we will now show how to extend this to a *refined* model $\tilde{f}_{1..j}$ that is functionally equivalent to $f_{1..j}$. In an idealized environment with infinite precision floating point arithmetic this step is completely unnecessary; however empirically this step brings the relative error in the extracted layer's weights from 2^{-15} to 2^{-35} or better.

To begin, select a neuron η with $\mathcal{L}(\eta) = j$. By querying the already-extracted model $\hat{f}_{1..j}$, analytically compute witnesses $\{x_i\}_{i=1}^{d_j}$ so that each $x_i \in \hat{\mathcal{W}}(\eta)$. This requires no queries to the model as we have already extracted this partial model.

If the $\hat{A}^{(j)}$ and $\hat{b}^{(j)}$ were exactly correct then $\mathcal{W}(\eta; \cdot) \equiv \hat{\mathcal{W}}(\eta; \cdot)$ and so each computed critical point x_i would be exactly a critical point of the true model f and so $\mathcal{V}(\eta; x_i) \equiv 0$. However, if there is any imprecision in the computation, then in general we will have that $0 < |\mathcal{V}(\eta; x_i)| < \epsilon$ for some small $\epsilon > 0$.

Fortunately, given this x_i it is easy to compute x_i' so that $\mathcal{V}(\eta; x_i') = 0$. To do this, we sample a random $\Delta \in \mathcal{R}^{d_0}$ and apply our binary search procedure on the range $[x_i + \Delta, x_i - \Delta]$. Here we should select Δ so that $\|\Delta\|$ is sufficiently small that the only critical points it crosses is the one induced by neuron η, but sufficiently large that it does reliably find the true critical point of η.

Repeating this procedure for each witness x_i gives a set of witnesses $\{x_i'\}_{i=1}^{d_j}$ to the same neuron η. We compute $h_i = \hat{f}_{1..j-1}(x_i')$ as the hidden vector that layer j will receive as input. By assumption h_i is precise already and so $\hat{f}_{1..j-1} \approx f_{1..j-1}$. Because x_i' is a witness to neuron η having value zero, we know that that $A_n^{(j)} \cdot h_i = 0$ where n corresponds to the row of neuron η in $A^{(j)}$.

Ideally we would solve this resulting system with least squares. However, in practice, occasionally the conversion from $x \to x'$ fails because x' is no longer a witness to the same neuron η'. This happens when there is some other neuron (i.e., η') that is closer to x than the true neuron η. Because least squares is not robust to outliers this procedure can fail to improve the solution.

We take two steps to ensure this does not happen. First, observe that if Δ is smaller, the likelihood of capturing incorrect neurons η' decreases faster than the likelihood of capturing the correct neuron η. Thus, we set Δ to be small enough that roughly half of the attempts at finding a witness x' fails. Second, we apply a (more) robust method of determining the vector that satisfies the solution of equations [JOB+18]. However, even these two techniques taken together occasionally fail to find valid solutions to improve the quality. When this happens, we reject this proposed improvement and keep the original value.

Our attack could be improved with a solution to the following robust statistics problem: Given a (known) set $S \subset \mathbb{R}^N$ such that for some (unknown) weight vector w we have $\Pr_{x \in S}[|w \cdot x + 1| \leq \epsilon] > \delta$ for sufficiently small ϵ, sufficiently large $\delta > 0.5$, and $\delta|S| > N$, efficiently recover the vector w to high precision.

5.2 Efficient Finite Differences

Most of the methods in this paper are built on computing second partial derivatives of the neural network f, and therefore developing a robust method for estimating the gradient is necessary. Throughout Sect. 4 we compute the partial derivative of f along direction α evaluated at x with step size ε as

$$\frac{\partial_\varepsilon}{\partial_\varepsilon \alpha} f(x) \stackrel{\text{def}}{=} \frac{f(x + \varepsilon \cdot \alpha) - f(x)}{\varepsilon}.$$

To compute the second partial derivative earlier, we computed α_+^i and α_-^i by first taking a step towards $x^* + \epsilon_0 e_1$ for a different step size ϵ_0 and then computed the first partial derivative at this location. However, with floating point imprecision it is not desirable to have two step sizes (ϵ_0 controlling the distance away from x^* to step, and ϵ controlling the step size when computing the partial derivative). Worse, we must have that $\epsilon \ll \epsilon_0$ because if $\frac{\partial f}{\partial e_1}\epsilon_0 > \frac{\partial f}{\partial e_i}\epsilon$ then when computing the partial derivative along e_i we may cross the hyperplane and estimate the first partial derivative incorrectly. Therefore, instead we compute

$$\alpha_+^i = \left.\frac{\partial f(x)}{\partial e_i}\right|_{x=x^*+\epsilon e_i} \quad \text{and} \quad \alpha_-^i = \left.\frac{\partial f(x)}{\partial - e_i}\right|_{x=x^*-\epsilon e_i}$$

where we both step along e_i and also take the partial derivative along the same e_i (and similarly for $-e_i$). This removes the requirement for an additional hyperparameter and allows the step size ϵ to be orders of magnitude larger, but introduces a new error: we now lose the relative signs of the entries in the row when performing extraction and can only recover $\left|A_{i,j}^{(1)}/A_{i,k}^{(1)}\right|$.

Extracting column signs. We next recover the value $\text{sign}(A_{i,j}^{(1)}) \cdot \text{sign}(A_{i,k}^{(1)})$. Fortunately, the same differencing process allows us to learn this information, using the following observation: if $A_{i,j}^{(1)}$ and $A_{i,k}^{(1)}$ have the same sign, then moving in the $e_j + e_k$ direction will cause their contributions to add. If they have different signs, their contributions will cancel each other. That is, if

$$\left| \alpha_+^{j+k} - \alpha_-^{j+k} \right| = \left| \alpha_+^j - \alpha_-^j \right| + \left| \alpha_+^k - \alpha_-^k \right|,$$

we have that

$$\left| (A_{i,j}^{(1)} + A_{i,k}^{(1)}) \cdot A^{(2)} \right| = \left| A_{i,j}^{(1)} \cdot A^{(2)} \right| + \left| A_{i,k}^{(1)} \cdot A^{(2)} \right|,$$

and therefore that

$$\left| \frac{A_{i,j}^{(1)}}{A_{i,k}^{(1)}} \right| = \frac{A_{i,j}^{(1)}}{A_{i,k}^{(1)}}.$$

We can repeat this process to test whether each $A_{i,j}^{(1)}$ has the same sign as (for example) $A_{i,1}^{(1)}$. However, we still do not know whether any single $A_{i,j}^{(1)}$ is positive or negative—we still must recover the row signs as done previously.
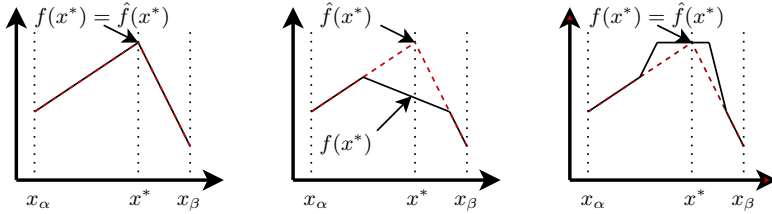
5.3   Finding Witnesses to Critical Points

Fig. 4. Efficient and accurate witness discovery. **(left)** If x_α and x_β differ in only one ReLU (as shown left), we can precisely identify the location x^* at which the ReLU reaches its critical point. **(middle)** If instead more than one ReLU differs (as shown right), we can detect that this has happened: the predicted of $\hat{f}(\cdot)$ evaluated at x^* as inferred from intersecting the dotted lines does not actually equal the true value of $f(x^*)$. **(right)** This procedure is not *sound* and still may potentially incorrectly identify critical points; in practice we find these are rare.

Throughout the paper we require the ability to find witnesses to critical points. Section 4.2 uses simple binary search to achieve this which is (a) imprecise in practice, and (b) query inefficient. We improve on the witness-finding search procedure developed by [JCB+19]. Again we interpolate between two examples u, v and let $x_\alpha = (1-\alpha)u + \alpha v$. Previously, we repeatedly performed binary search as long as the partial derivatives were not equal $\partial f(x_\alpha) \neq \partial f(x_\beta)$, requiring p queries to obtain p bits of precision of the value x^*. However, observe that if x_α

and x_β differ in the sign of exactly one neuron i, then we can directly compute the location x^* at which $\mathcal{V}(\eta_i; x^*) = 0$ but so that for all other η_j we have

$$\text{sign}\big(\mathcal{V}(\eta_j; x_\alpha)\big) = \text{sign}\big(\mathcal{V}(\eta_j; x^*)\big) = \text{sign}\big(\mathcal{V}(\eta_j; x_\beta)\big)$$

This approach is illustrated in Fig. 4 and relies on the fact that f is a piecewise linear function with two components. By measuring, $f(x_\alpha)$ and $\partial f(x_\alpha)$ (resp., $f(x_\beta)$ and $\partial f(x_\beta)$), we find the slope and intercept of both the left and right lines in Fig. 4 (left). This allows us to solve for their expected intersection $(x^*, \hat{f}(x^*))$. Typically, if there are more than two linear segments, as in the middle of the figure, we will find that the true function value $f(x^*)$ will not agree with the expected function value $\hat{f}(x^*)$ we obtained by computing the intersection; we can then perform binary search again and repeat the procedure.

However, we lose some soundness from this procedure. As we see in Fig. 4 (right), situations may arise where many ReLU units change sign between x_α and x_β, but $\hat{f}(x^*) = f(x^*)$. In this case, we would erroneously return x^* as a critical point, and miss all of the other critical points in the range. Fortunately, this error case is pathological and does not occur in practice.

5.3.1 Further Reducing Query Complexity of Witness Discovery

Suppose that we had already extracted the first j layers of the neural network and would like to perform the above critical-point finding algorithm to identify all critical points between x_α and x_β. Notice that we do not need to collect any more critical points from the first j layers, but running binary search will recover them nonetheless. To bypass this, we can analytically compute S as the set of all witnesses to critical points on the extracted neural network $\hat{f}_{1..j}$ between x_α and x_β. As long as the extracted network $\hat{f}$ is correct so far, we are guaranteed that all points in S are also witnesses to critical points of the true f.

Instead of querying on the range (x_α, x_β) we perform the $|S| + 1$ different searches. Order the elements of S as $\{s_i\}_{i=1}^{|S|}$ so that $s_i < s_j \implies |x_\alpha - s_i| < |x_\alpha - s_j|$. Abusing notation, let $s_1 = x_\alpha$ and $s_{|S|} = x_\beta$. Then, perform binary search on each disjoint range (S_i, S_{i+1}) for $i = 1$ to $|S| - 1$ and return the union.

5.4 Unification of Witnesses with Noisy Gradients

Recall that to extract $\hat{A}^{(l)}$ we extract candidates candidates $\{r_i\}$ and search for pairs r_i, r_j that agree on multiple coordinates. This allows us to merge r_i and r_j to recover (eventually) full rows of $\hat{A}^{(l)}$. With floating point error, the unification algorithm in Sect. 4.3 fails for several reasons.

Our core algorithm computes the normal to a hyperplane, returning pairwise ratios $\hat{A}_{i,j}^{(1)}/\hat{A}_{i,k}^{(1)}$; throughout Sect. 4 we set $\hat{A}_{i,1}^{(1)} = 1$ without loss of generality.

Unfortunately in practice there is loss of generality, due to the disparate impact of numerical instability. Consider the case where $A_{i,1}^{(l)} < 10^{-\alpha}$ for $\alpha \gg 0$, but $A_{i,k}^{(l)} \geq 1$ for all other k. Then there will be substantially more (relative)

floating point imprecision in the weight $A_{i,1}^{(l)}$ than in the other weights. Before normalizing there is no cause for concern since the absolute error is no larger than for any other. However, the described algorithm now normalizes every *other* coordinate $A_{i,k}^{(l)}$ by dividing it by $A_{i,1}^{(l)}$—polluting the precision of these values.

Therefore we adjust our solution. At layer l, we are given a collection of vectors $R = \{r_i\}_{i=1}^n$ so that each r_i corresponds to the extraction of some (unknown) neuron η_i. First, we need an algorithm to cluster the items into sets $\{S_j\}_{j=1}^{d_l}$ so that $S_j \subset R$ and so that every vector in S_j corresponds to one neuron on layer l. We then need to unify each set S_j to obtain the final row of $\hat{A}_j^{(l)}$.

Creating the Subsets S with Graph Clustering. Let $r_m^{(a)} \in S_n$ denote the ath coordinate of the extracted row r_m from cluster n. Begin by constructing a graph $G = (V, E)$ where each vector r_i corresponds to a vertex. Let $\delta_{ij}^{(k)} = |r_i^{(k)} - r_j^{(k)}|$ denote the difference between row r_i and row r_j along axis k; then connect an edge from r_i to r_j when the approximate $\|\cdot\|_0$ norm is sufficiently large $\sum_k \mathbb{1}\left[\delta_{ij}^{(k)} < \epsilon\right] > \log d_0$. We compute the connected components of G and partition each set S_j as one connected component. Observe that if $\epsilon = 0$ then this procedure is exactly what was described earlier, pairing vectors whose entries agree perfectly; in practice we find a value of $\varepsilon = 10^{-5}$ suffices.

Unifying Each Cluster to Obtain the Row Weights. We construct the three dimensional $M_{i,a,b} = r_a^{(i)}/r_b^{(i)}$. Given M, the a good guess for the scalar c_{ab} so that $r_a^{(i)} = r_b^{(i)} \cdot C_{ab}$ along as many coordinates i as possible is the assignment $C_{ab} = \text{median}_i M_{i,a,b}$, where the estimated error is $e_{ab} = \text{stdev}_i M_{i,a,b}$.

If all r_a were complete and had no imprecision then C_{ab} would have no error and so $C_{ab} = C_{ax} \cdot C_{xb}$. However because it does have error, we can iteratively improve the guessed C matrix by observing that if the error $e_{ax} + e_{xb} < e_{ab}$ then the guessed assignment $C_{ax} \cdot C_{xb}$ is a better guess than C_{ab}. Thus we replace $C_{ab} \leftarrow C_{ax} \cdot C_{xb}$ and update $e_{ab} \leftarrow e_{ax} + e_{xb}$. We iterate this process until there is no further improvement. Then, finally, we choose the optimal dimension $a = \arg\min_a \sum_b e_{ab}$ and return the vector C_a. Observe that this procedure closely follows constructing the union of two partial entries r_i and r_j except that we perform it along the best axis possible for each coordinate.

5.5   Following Neuron Critical Points

Section 4.4.3 developed techniques to construct a set of witnesses to the same neuron being at its critical point. We now numerically-stabilize this procedure.

As before we begin with an input $x^* \in \mathcal{W}(\eta^*)$ and compute the normal vector n to the critical plane at x^*, and then choose r satisfying $r \cdot n = 0$. The computation of n will necessarily have some floating point error, so r will too.

This means when we compute $\alpha = \text{Proj}_{1..j}(x^*, r)$ and let $x' = x^* + r\alpha$ the resulting x' will be almost exactly a witness to some neuron η_u with $\mathcal{L}(\eta_u) < j$, (because this computation was performed analytically on a precisely extracted
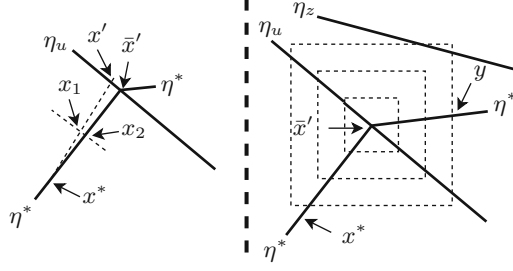
Fig. 5. Numerically stable critical-point following algorithm. **(left)** From a point x' compute a parallel direction along η^*, step part way to x_1 and refine it to x_2, and then finish stepping to x'. **(right)** From x' grow increasingly large squares until there are more than four intersection points; return y as the point on η^* on the largest square.

model), but x' has likely drifted off of the original critical plane induced by η^* (Fig. 5).

To address this, after computing α we initially take a smaller step and let $x_1 = x^* + r\sqrt{\alpha}$. We then refine the location of this point to a point x_2 by performing binary search on the region $x_1 - \epsilon n$ to $x_1 + \epsilon n$ for a small step ϵ. If there was no error in computing n then $x_1 = x_2$ because both are already witnesses to η^*. If not, any error has been corrected. Given x^* and x_2 we now can now compute $\alpha_2 = \text{Proj}_{1..j}(x^*, x_2 - x^*)$ and let $\bar{x}' = x^* + (x_2 - x^*)\alpha_2$ which will actually be a witness to both neurons simultaneously.

Next we give a stable method to compute y that is a witness to η^* and on the other side of η_u. The previous procedure required a search parallel to η_u and infinitesimally displaced, but this is not numerically stable without accurately yet knowing the normal to the hyperplane given by η_u.

Instead we perform the following procedure. Choose two orthogonal vectors of equal length β, γ and and perform binary search on the line segments that trace out the perimeter of a square with coordinates $\bar{x}' \pm \beta \pm \gamma$.

When $\|\beta\|$ is small, the number of critical points crossed will be exactly four: two because of η_u and two because of η^*. As long as the number of critical points remains four, we double the length of β and γ.

Eventually we will discover more than four critical points, when the perimeter of the square intersects another neuron η_z. At this point we stop increasing the size of the box and can compute the continuation direction of η^* by discarding the points that fall on η_u. We can then choose y as the point on η^* that intersected with the largest square binary search.

5.5.1 Determining Optimal Continuation Directions
The hyperplane following procedure will *eventually* recover a fully diverse set of inputs W but it may take a large number of queries to do so. We can reduce the number of queries by several orders of magnitude by carefully choosing the continuation direction r instead of randomly choosing any value so that $r \cdot n = 0$.

Given the initial coordinate x and after computing the normal n to the hyperplane, we have $d_0 - 1$ dimensions that we can choose between to travel next. Instead of choosing a random $r \cdot n = 0$ we instead choose r such that we make progress towards obtaining a fully diverse set W.

Define W_i as the set of witnesses that have been found so far. We say that this set is diverse on neuron η if there exists an $x_+, x_- \in W_i$ such that $\mathcal{V}(\eta; x_+) \geq 0$ and $\mathcal{V}(\eta; x_-) < 0$. Choose an arbitrary neuron η_t such that W_i is not diverse on η_t. (If there are multiple such options, we should prefer the neuron that would be *easiest* to reach, but this is secondary.)

Our goal will be to choose a direction r such that (1) as before, $r \cdot n = 0$, however (2) $W_i \cup \{x + \alpha r\}$ is closer to being fully diverse. Here, "closer" means that $d(W) = \min_{x \in W} |\mathcal{V}(\eta_t; x)|$ is smaller. Because the set is not yet diverse on η_t, all values are either positive or negative, and it is our objective to switch the sign, and therefore become closer to zero. Therefore our procedure sets

$$r = \arg \min_{r:\ r \cdot n = 0} d(W_i \cup \{x + \alpha r\})$$

performing the minimization through random search over 1,000 directions.

6   Evaluation

We implement the described extraction algorithm in JAX [BFH+18], a Python library that mirrors the NumPy interface for performing efficient numerical computation through just in time compilation.

6.1   Computing $(\varepsilon, 10^{-9})$-Functional Equivalence

Computing $(\varepsilon, 10^{-9})$-functional equivalence is simple. Let $\bar{S} \subset S$ be a finite set consisting of $|\bar{S}| > 10^9$ different inputs drawn $x \in S$. Sort $\bar{S}$ by $|f(x) - \hat{f}(x)|$ and choose the lowest ε so that

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

In practice we set $|\bar{S}| = 10^9$ and compute the max so that evaluating the function is possible under an hour per neural network.

6.2   Computing $(\varepsilon, 0)$-Functional Equivalence

Directly computing $(\varepsilon, 0)$-functional equivalence is infeasible, and is NP-hard (even to approximate) by reduction to Subset Sum [JCB+19]. We nevertheless propose two methods that efficiently give upper bounds that perform well.

Error bounds propagation. The most direct method to compute $(\varepsilon, 0)$-functional equivalence of the extracted neural network $\hat{f}$ is to compare the weights $A^{(i)}$ to the weights $\hat{A}^{(i)}$ and analytically derive an upper bound on the error when performing inference. Observe that (1) permuting the order of the neurons in the network does not change the output, and (2) any row can be multiplied by a positive scalar $c > 0$ if the corresponding column in the next layer is divided by c. Thus, before we can compare $\hat{A}^{(i)}$ to $A^{(i)}$ we must "align" them. We identify the permutation mapping the rows of $\hat{A}^{(l)}$ to the rows of $A^{(l)}$ through a greedy matching algorithm, and then compute a single scalar per row $s \in \mathbb{R}^{d_i}_+$. To ensure that multiplying by a scalar does not change the output of the network, we multiply the columns of the next layer $\hat{A}^{(l+1)}$ by $1/s$ (with the inverse taken pairwise). The process to align the bias vectors $b^{(l)}$ is identical, and the process is repeated for each further layer.

This gives an aligned $\tilde{A}^{(i)}$ and $\tilde{b}^{(i)}$ from which we can analytically derive upper bounds on the error. Let $\Delta_i = \tilde{A}^{(i)} - A^{(i)}$, and let δ_i be the largest singular value of Δ_i. If the ℓ_2-norm of the maximum error going into layer i is given by e_i then we can bound the maximum error going out of layer i as

$$e_{i+1} \leq \delta_i \cdot e_i + \|\tilde{b}^{(i)} - b^{(i)}\|_2.$$

By propagating bounds layer-by-layer we can obtain an upper bound on the maximum error of the output of the model.

This method is able to prove an upper bound on $(\epsilon, 0)$ functional equivalence for some networks, when the pairing algorithm succeeds. However, we find that there are some networks that are $(2^{-45}, 10^{-9})$ functionally equivalent but where the weight alignment procedure fails. Therefore, we suspect that there are more equivalence classes of functions than scalar multiples of permuted neurons, and so develop further methods for tightly computing $(\varepsilon, 0)$ functional equivalence.

Error overapproximation through MILP. The above analysis approach is loose. Our second approach gives exact bounds with an additive error at most 10^{-10}.

Neural networks are piecewise linear functions, and so can be cast as a mixed integer linear programming (MILP) problem [KBD+17]. We directly express Definition 1 as a MILP, following the process of [KBD+17] by encoding linear layers directly, and encoding ReLU layers by assigning a binary integer variable to each ReLU. Due to the exponential nature of the problem, this approach is limited to small networks.

State-of-the-art MILP solvers offer a maximum (relative, additive) error tolerance of 10^{-10}; for our networks the SVD upper bound is often 10^{-10} or better, so the MILP solver gives a *worse* bound, despite theoretically being tight.

7   Results

We extract a wide range of neural network architectures; key results are given in Table 1 (Sect. 1). We compute (ε, δ)-functional equivalence at $\delta = 10^{-9}$ and $\delta = 0$ on the domain $S = \{x \colon \|x\|_2 < d_0 \ \wedge \ x \in \mathcal{X}\}$, sufficient to explore both sides of every neuron.

8   Concluding Remarks

We introduce a cryptanalytic method for extracting the weights of a neural network by drawing analogies to cryptanalysis of keyed ciphers. Our differential attack requires multiple orders of magnitude fewer queries per parameter than prior work and extracts models that are multiple orders of magnitude more accurate than prior work. In this work, we do not consider defenses—promising approaches include detecting when an attack is occuring, adding noise at some stage of the model's computation, or only returning the label corresponding to the output, any of these easily break our presented attack.

The practicality of this attack has implications for many areas of machine learning and cryptographic research. The field of secure inference relies on the assumption that observing the output of a neural network does not reveal the weights. This assumption is false, and therefore the field of secure inference will need to develop new techniques to protect the secrecy of trained models.

We believe that by casting neural network extraction as a cryptanalytic problem, even more advanced cryptanalytic techniques will be able to greatly improve on our results, reducing the computational complexity, reducing the query complexity and reducing the number of assumptions necessary.

References

[BBJP19] Batina, L., Bhasin, S., Jap, D., Picek, S.: CSI NN: reverse engineering of neural network architectures through electromagnetic side channel. In: 28th USENIX Security Symposium (2019)

[BCB15] Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. In: 3rd International Conference on Learning Representations (ICLR) (2015)

[BCM+13] Biggio, B., et al.: Evasion attacks against machine learning at test time. In: Blockeel, H., Kersting, K., Nijssen, S., Železný, F. (eds.) ECML PKDD 2013. LNCS (LNAI), vol. 8190, pp. 387–402. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40994-3_25

[BFH+18] Bradbury, J., et al.: JAX: composable transformations of Python+NumPy programs (2018)

[BS91] Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. J. Cryptol. **4**(1), 3–72 (1991). https://doi.org/10.1007/BF00630563

[CCG+18] Chandrasekaran, V., Chaudhuri, K., Giacomelli, I., Jha, S., Yan, S.: Exploring connections between active learning and model extraction. arXiv preprint arXiv:1811.02054 (2018)

[CLE+19] Carlini, N., Liu, C., Erlingsson, Ú., Kos, J., Song, D.: The secret sharer: evaluating and testing unintended memorization in neural networks. In: USENIX Security Symposium, pp. 267–284 (2019)

[DGKP20] Das, A., Gollapudi, S., Kumar, R., Panigrahy, R.: On the learnability of random deep networks. In: ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, pp. 398–410 (2020)

[EKN+17] Esteva, A., et al.: Dermatologist-level classification of skin cancer with deep neural networks. Nature **542**(7639), 115–118 (2017)

[FJR15] Fredrikson, M., Jha, S., Ristenpart, T.: Model inversion attacks that exploit confidence information and basic countermeasures. In: ACM CCS, pp. 1322–1333 (2015)

[GBDL+16] Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: CryptoNets: applying neural networks to encrypted data with high throughput and accuracy. In: International Conference on Machine Learning, pp. 201–210 (2016)

[Gen09] Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009)

[HDK+20] Hong, S., Davinroy, M., Kaya, Y., Dachman-Soled, D., Dumitraş, T.: How to 0wn the NAS in your spare time. In: International Conference on Learning Representations (2020)

[HZRS16] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)

[JCB+19] Jagielski, M., Carlini, N., Berthelot, D., Kurakin, A., Papernot, N.: High-fidelity extraction of neural network models. arXiv:1909.01838 (2019)

[JOB+18] Jagielski, M., Oprea, A., Biggio, B., Liu, C., Nita-Rotaru, C., Li, B.: Manipulating machine learning: poisoning attacks and countermeasures for regression learning. In: 2018 IEEE Symposium on Security and Privacy (S&P), pp. 19–35. IEEE (2018)

[KBD+17] Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5

[KLA+19] Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., Aila, T.: Analyzing and improving the image quality of StyleGAN. CoRR, abs/1912.04958 (2019)

[KTP+19] Krishna, K., Tomar, G.S., Parikh, A.P., Papernot, N., Iyyer, M.: Thieves on sesame street! Model extraction of BERT-based APIs. arXiv preprint arXiv:1910.12366 (2019)

[Lev14] Levinovitz, A.: The mystery of Go, the ancient game that computers still can't win. Wired, May 2014

[MLS+20] Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., Popa, R.A.: DELPHI: a cryptographic inference service for neural networks. In: 29th USENIX Security Symposium (2020)

[MSDH19] Milli, S., Schmidt, L., Dragan, A.D., Hardt, M.: Model reconstruction from model explanations. In: Proceedings of the Conference on Fairness, Accountability, and Transparency, FAT* 2019, pp. 1–9 (2019)

[NH10] Nair, V., Hinton, G.E.: Rectified linear units improve restricted Boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning (ICML), pp. 807–814 (2010)

[RK19] Rolnick, D., Kording, K.P.: Identifying weights and architectures of unknown ReLU networks. arXiv preprint arXiv:1910.00744 (2019)

[RWT+18] Riazi, M.S., Weinert, C., Tkachenko, O., Songhori, E.M., Schneider, T., Koushanfar, F.: Chameleon: a hybrid secure computation framework for machine learning applications. In: ACM ASIACCS, pp. 707–721 (2018)

[SHM+16] Silver, D., et al.: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484 (2016)

[SIVA17] Szegedy, C., Ioffe, S., Vanhoucke, V., Alemi, A.A.: Inception-v4, Inception-ResNet and the impact of residual connections on learning. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI 2017, pp. 4278–4284. AAAI Press (2017)

[SSRD19] Shamir, A., Safran, I., Ronen, E., Dunkelman, O.: A simple explanation for the existence of adversarial examples with small Hamming distance. CoRR, abs/1901.10861 (2019)

[SZS+14] Szegedy, C., et al.: Intriguing properties of neural networks. In: 2nd International Conference on Learning Representations (ICLR 2014). arXiv:1312.6199 (2014)

[TL19] Tan, M., Le, Q.V.: EfficientNet: rethinking model scaling for convolutional neural networks. arXiv preprint arXiv:1905.11946 (2019)

[TZJ+16] Tramèr, F., Zhang, F., Juels, A., Reiter, M.K., Ristenpart, T.: Stealing machine learning models via prediction APIs. In: USENIX Security Symposium, pp. 601–618 (2016)

[Wen90] Wenskay, D.L.: Intellectual property protection for neural networks. Neural Netw. **3**(2), 229–236 (1990)

[WG18] Wang, B., Gong, N.Z.: Stealing hyperparameters in machine learning. In: 2018 IEEE Symposium on Security and Privacy (S&P), pp. 36–52. IEEE (2018)

[WSC+16] Wu, Y., et al.: Google's neural machine translation system: bridging the gap between human and machine translation. arXiv preprint arXiv:1609.08144 (2016)

[XHLL19] Xie, Q., Hovy, E., Luong, M.-T., Le, Q.V.: Self-training with noisy student improves ImageNet classification. arXiv preprint arXiv:1911.04252 (2019)

[Yao86] Yao, A.C.-C.: How to generate and exchange secrets. In: FOCS 1986, pp. 162–167. IEEE (1986)

[ZL16] Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578 (2016)