Reverse-Engineering Deep ReLU Networks

David Rolnick [1] **Konrad P. Körding** [1]

Abstract

The output of a neural network depends on its architecture and weights in a highly nonlinear way, and it is often assumed that a network's parameters cannot be recovered from its output. Here, we prove that, in fact, it is frequently possible to reconstruct the architecture, weights, and biases of a deep ReLU network by observing only its output. We leverage the fact that every ReLU network defines a piecewise linear function, where the boundaries between linear regions correspond to inputs for which some neuron in the network switches between inactive and active ReLU states. By dissecting the set of region boundaries into components associated with particular neurons, we show both theoretically and empirically that it is possible to recover the weights of neurons and their arrangement within the network, up to isomorphism.

1. Introduction

A deep neural network computes a function from inputs to outputs, and the architecture and weights of the network determine the function that is expressed. While every parameter influences the overall function, this influence is highly nonlinear, and the effects of different neurons within the network would appear, at least superficially, to be hopelessly entangled. Consequently, it has often been supposed that it is impossible to recover the parameters of a network merely by observing its output on different inputs.

Were it possible to reverse-engineer a neural network from the function it computes, there could be serious implications for security and privacy. In many deployed deep learning systems, the output is freely available but the network used to generate that output is not. The ability to recover a confidential network could even potentially expose the data used to train the network, and could also make it easier to construct adversarial attacks.

The related question of reverse-engineering biological neural networks is of foundational interest in neuroscience. Within the brain, it is often possible to record the activity of a cell but not the presynaptic cells that drive it, and therefore experimental neuroscientists must infer full activity and weights from partial recordings. Our understanding of the brain would be greatly improved if it were possible to identify the internal components of a neural circuit based on recordings of the output of that circuit.

In this work, we show mathematically and empirically that it is, in fact, often possible to recover the architecture, weights, and biases of a deep ReLU network by querying it. Our approach leverages the fact that a ReLU network is piecewise linear and transitions between linear pieces when one of the ReLUs of the network transitions from its inactive to its active state. For neurons in the first layer of the network, such transitions occur along hyperplanes through input space; the equations of these hyperplanes determine the weights and biases of the first layer (up to sign and scaling). For neurons in subsequent layers, transitions occur along "bent hyperplanes" that bend where they intersect bent hyperplanes associated with earlier layers. Measuring the intersections between bent hyperplanes allows us to recover the weights between the corresponding neurons.

Our principal contributions are:

- We prove that the architecture, weights, and biases of a deep ReLU network can be recovered (up to isomorphism) from the arrangement of regions on which the network is linear.
- We describe an algorithm to recover the network by approximating the boundaries between these linear regions, with the only information given about the network being its output on specified queries.
- We demonstrate that our algorithm succeeds in reconstructing both trained and untrained ReLU networks.

Each of these contributions significantly advances the state of the art. No prior work has, to our knowledge, been able to reverse-engineer even the first layer of a network with 2 hidden layers. By contrast, our theoretical results and

[1]University of Pennsylvania, Philadelphia, PA, USA. Correspondence to: David Rolnick <drolnick@seas.upenn.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020.

algorithm hold for recovering any layer of a network of any depth. Furthermore, we show empirically that our algorithm is able to reconstruct the first layer of 2-, 3-, and 4-layer networks, as well as the second layer of 2-layer networks.

2. Related work

Various works within the deep learning literature have considered the problem of inferring a network given its output on inputs drawn (non-adaptively) from a given distribution. It is known that this problem is in general hard (Goel et al., 2017), though positive results have been found for certain specific choices of distribution in the case that the network has only one hidden layer (Ge et al., 2019; Goel & Klivans, 2017). By contrast, we consider the problem of reconstructing a network of arbitrary depth, given the ability to issue queries at specified input points. In this work, we leverage the theory of linear regions within a ReLU network, an area studied e.g. by Telgarsky (2015); Raghu et al. (2017); Hanin & Rolnick (2019a). Most recently, Hanin & Rolnick (2019b) considered the boundaries between linear regions as arrangements of "bent hyperplanes". Milli et al. (2019); Jagielski et al. (2019) show the effectiveness of this strategy for networks with one hidden layer. For inference of other properties of unknown networks, see e.g. Oh et al. (2019).

Neuroscientists have long considered similar problems with biological neural networks, albeit armed with prior knowledge about network structure. For example, it is believed that so-called "complex cells" in the primary visual cortex, which are often seen as translation-invariant edge detectors, obtain their invariance through what is effectively a two-layer neural network (Kording et al., 2004). A first layer is believed to extract edges, while a second layer essentially implements max-pooling. Many biological neurons appear to be well modeled as the ReLU of a linear combination of their inputs (Chance et al., 2002). Heggelund (1981) perform physical experiments akin to our approach of identifying one ReLU at a time, by applying inputs that move individual neurons above their critical threshold one by one. Being able to solve such problems more generically would be useful for a range of neuroscience applications.

3. Theoretical framework

We will, in general, consider fully connected, feed-forward neural networks (multi-layer perceptrons) with ReLU activations. Each such network $\mathcal{N}$ defines a function $\mathcal{N}(\mathbf{x})$ from input space $\mathbb{R}^{n_{\text{in}}}$ to output space $\mathbb{R}^{n_{\text{out}}}$. We denote the layer widths of the network by n_{in} (input layer), $n_1, n_2, \ldots, n_d$, n_{out} (output layer). We denote by $\mathbf{W}^k$ the weight matrix from layer $(k-1)$ to layer k, where layer 0 is the input; and $\mathbf{b}^k$ denotes the bias vector for layer k. Given a neuron z in the network, we use $z(\mathbf{x})$ to denote its preactivation for input $\mathbf{x} \in \mathbb{R}^{n_{\text{in}}}$. Thus, for the jth neuron in layer k, we have

$$z_j^k(\mathbf{x}) = \sum_{i=1}^{n_{k-1}} \mathbf{W}_{ij}^k \, \text{ReLU}(z_i^{k-1}(\mathbf{x})) + \mathbf{b}_j^k$$

3.1. Isomorphisms of networks

Before showing how to reverse-engineer the parameters of a neural network, we must consider to what extent these parameters can be inferred unambiguously. For every network, there are a number of other networks that define exactly the same function from input space to output space. We say that two networks $\mathcal{N}$ and $\mathcal{N}'$ (possibly with different parameters or architecture) are *isomorphic* if $\mathcal{N}(\mathbf{x}) = \mathcal{N}'(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{R}^{n_{\text{in}}}$. For a fully connected network with ReLU activation, there are two obvious network isomorphisms:

Permutation. The order of neurons in each layer of a network $\mathcal{N}$ does not affect the underlying function. Formally, for a permutation σ and layer k in the network, let $p_{k,\sigma}(\mathcal{N})$ be the network obtained from $\mathcal{N}$ by reindexing layer k according to σ (along with the incoming and outgoing weights, and the biases). Then, $p_{k,\sigma}(\mathcal{N})$ is isomorphic to $\mathcal{N}$.

Scaling. Due to the ReLU's equivariance under multiplication, it is possible to scale the incoming weights and biases of any neuron, while inversely scaling the outgoing weights, leaving the overall function unchanged. Formally, for z the ith neuron in layer k and for any $c > 0$, let $s_{z,c}(\mathcal{N})$ be the network obtained from $\mathcal{N}$ by replacing $\mathbf{W}_{\cdot i}^k$, $\mathbf{b}_i^k$, and $\mathbf{W}^{k+1}$ by $c\mathbf{W}_{\cdot i}^k$, cb_i^k, and $(1/c)\mathbf{W}_{i\cdot}^{k+1}$, respectively. It is simple to prove that $s_{z,c}(\mathcal{N})$ is isomorphic to $\mathcal{N}$ (see Appendix A).

It is worth noting that permutation and scaling are isomorphisms for *every* ReLU network, but some networks may have additional isomorphisms. It is shown in Phuong & Lampert (2019) that there exist ReLU networks of every architecture for which all isomorphisms are given by some combination of permutation and scaling.[1] As we demonstrate in §5.2, there exists a positive-measure set of ReLU networks with additional isomorphisms.

In this work, we show that, where additional isomorphisms do not occur, it is in fact often possible both in theory and in practice to reverse-engineer ReLU networks up to permutation and scaling.

3.2. Linear regions

Let $\mathcal{N}$ be a neural network. For each neuron $z \in \mathcal{N}$, we denote by B_z the set of $\mathbf{x}$ for which $z(\mathbf{x}) = 0$. Thus, the ReLU at z is active for $\mathbf{x}$ on one side of B_z, and inactive for $\mathbf{x}$ on the other side. We call B_z the *boundary* associated with z, and say that $B = \bigcup B_z$ is the *boundary* of the overall

[1] See also Fefferman (1994) for a proof that an analogous set of isomorphisms is comprehensive for networks with tanh activation.

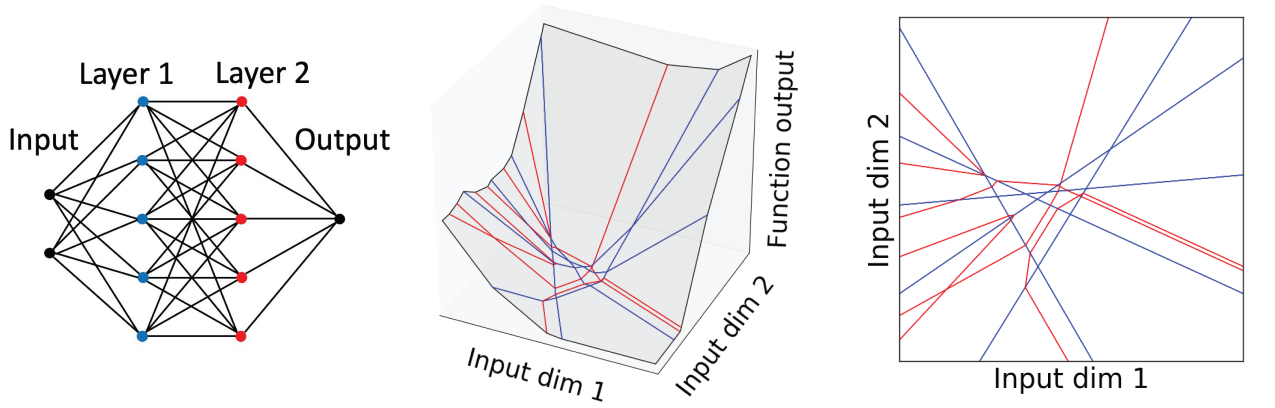

Figure 1. Left: Architecture of a ReLU network $\mathcal{N}(\mathbf{x}) : \mathbb{R}^2 \to \mathbb{R}$ with two hidden layers of width 5. Center: Graph of the piecewise linear function $\mathcal{N}(\mathbf{x})$ as a function of the two input variables. Right: Boundaries between linear regions of $\mathcal{N}$, essentially a "flattened" form of the center image. Boundaries B_z corresponding to neurons z from the first and second layers are shown in blue and red, respectively.

network. We refer to the connected components of $\mathbb{R}^{n_{\text{in}}} \setminus B$ as *linear regions*. Thus, each linear region corresponds to a pattern of active/inactive ReLUs across the network. This means that, within a linear region, the neural network computes a fixed linear function (see Fig. 1).

Throughout this paper, we will make the *Linear Regions Assumption*: Each region represents a maximal connected component of input space on which the piecewise linear function $\mathcal{N}(\mathbf{x})$ is given by a single linear function. While this assumption has tacitly been made in the prior literature, it is noted in Hanin & Rolnick (2019b) that there are cases where it does not hold. For example, if an entire layer of the network is zeroed out for some inputs, then $\mathcal{N}(\mathbf{x})$ may be given by the same linear function across several adjacent regions, despite these regions corresponding to different ReLU activation patterns.

3.3. Main result

Hanin & Rolnick (2019b) show that for all but a measure-zero set of networks, B_z is an $(n_{\text{in}} - 1)$-dimensional piecewise linear surface in $\mathbb{R}^{n_{\text{in}}}$, which they call a *bent hyperplane*. We say that B_z *bends* at a point if B_z is nonlinear at that point. As observed in Hanin & Rolnick (2019b), B_z can bend only at points where it intersects boundaries $B_{z'}$ for z' in an earlier layer of the network (see Fig. 1, in which input dimension is 2 and the B_z are simply bent lines). The following theorem (proven in Appendix B) shows that the converse also holds.

Theorem 1 (Boundaries imply network structure)**.** *Let $\mathcal{N}$ be a fully connected ReLU network satisfying the Linear Regions Assumption. Then, the following statement holds except for $\mathcal{N}$ in a measure-zero set of networks: For every neuron z, the boundary B_z bends exactly where it intersects a boundary $B_{z'}$ such that z' is in an earlier layer than z.*

From this theorem, it follows that for any two intersecting boundaries B_z and $B_{z'}$, one of the following must hold: B_z bends at their intersection (in which case z occurs in a deeper layer of the network), $B_{z'}$ bends (in which case z' occurs in a deeper layer), or neither bends (in which case z and z' occur in the same layer). It is not possible for both B_z and $B_{z'}$ to bend at their intersection – unless that intersection is also contained in another boundary, which is vanishingly unlikely in general. Therefore, the architecture of the network can be determined by identifying the boundaries B_z and where they bend in relation to one another.

In fact, a much stronger statement holds; namely, the weights and biases of the network can also be determined from the boundaries:

Theorem 2 (Boundaries imply network weights)**.** *Let $\mathcal{N}$ be a fully connected ReLU network satisfying the Linear Regions Assumption. Suppose that for every neuron z in $\mathcal{N}$, the boundary B_z is a connected set, and suppose that for any two neurons z and z' with a weight between them, the boundaries B_z and $B_{z'}$ intersect. Then, given the set of boundary points between linear regions, it is possible to recover both the architecture of $\mathcal{N}$ and the weights and biases of every hidden layer, up to permutation and scaling, except for $\mathcal{N}$ in a measure-zero set of networks.*

As an intuition for why Theorem 2 holds, observe first that for neurons z in the first layer of the network, the boundaries B_z are simply hyperplanes (see Fig. 1). The equations of these hyperplanes reveal the weights from the input to the first layer (up to permutation, scaling, and sign). For each subsequent layer, the weight between neurons z and z' can be determined by calculating how $B_{z'}$ bends when it crosses B_z, as this dictates how much the input to z changes when neuron z' is zeroed out by its ReLU.

From Theorem 2, we see that in order to reconstruct a net-

work, it is sufficient to identify the boundaries between linear regions. We show this can be achieved simply by querying the network:

Theorem 3 (Reverse-engineering deep ReLU networks)**.** *Suppose that $\mathcal{N}$ is a deep ReLU network satisfying the assumptions of Theorem 2. Then, by observing the output of $\mathcal{N}$ on specified queries, it is possible to approximate the boundaries of the network and therefore to recover the full architecture, weights, and biases.*

Theorems 2 and 3 follow from the explicit algorithm we now present. We prove the algorithm's validity in App. C.

4. Algorithm

We now describe an algorithm to reverse-engineer a network $\mathcal{N}$ by approximating the boundaries between linear regions. Our algorithm assumes that the output of the network $\mathcal{N}(\mathbf{x})$ can be queried for different inputs $\mathbf{x}$, but does not assume any *a priori* knowledge of the linear regions or boundaries.

4.1. The first layer

We begin by identifying the first layer of the network $\mathcal{N}$, for which we must infer the number of neurons, the weight matrix $\mathbf{W}^1$, and the bias vector $\mathbf{b}^1$. As noted above, for each $z = z_i^1$ in the first layer, the boundary B_z is a hyperplane with equation $\mathbf{W}^1_{i\cdot}\mathbf{x} + \mathbf{b}^1_i = 0$. For each neuron z in a later layer of the network, the boundary B_z will, in general, bend and not be a (full) hyperplane (Theorem 1). We may therefore find the number of neurons in layer 1 by counting the hyperplanes contained in the network's boundary B, and we can infer weights and biases by determining the equations of these hyperplanes.

Boundary points along a line. Our algorithm is based upon the identification of points on the boundary B. One of our core algorithmic primitives is a subroutine `PointsOnLine` that takes as input a line segment $\ell \subset \mathbb{R}^n_{\text{in}}$ and approximates the set $\ell \cap B$ of boundary points along ℓ. The algorithm proceeds by leveraging the fact that boundary points subdivide ℓ into regions within which $\mathcal{N}(\mathbf{x})$ is linear. We maintain a list of points in order along ℓ (initialized to the endpoints and midpoint of ℓ) and iteratively perform the following operation: For each three consecutive points on our list, $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$, we determine if the vectors $(\mathcal{N}(\mathbf{x}_2) - \mathcal{N}(\mathbf{x}_1))/||\mathbf{x}_2 - \mathbf{x}_1||_2$ and $(\mathcal{N}(\mathbf{x}_3) - \mathcal{N}(\mathbf{x}_2))/||\mathbf{x}_3 - \mathbf{x}_2||_2$ are equal (to within computation error) – if so, we remove the point $\mathbf{x}_2$ from our list, otherwise we add the points $(\mathbf{x}_1 + 2\mathbf{x}_2)/3$ and $(\mathbf{x}_3 + 2\mathbf{x}_2)/3$ to our list.[2] The points in the list converge by binary search to the set of discontinuities of the gradient $\nabla\mathcal{N}(\mathbf{x})$, which are our desired boundary points. Note that `PointsOnLine` is where we make use of our ability to query the network.

Algorithm 1 The first layer

```
Initialize P_1 = P_2 = S_1 = {}
for t = 1, ..., L do
    Sample line segment ℓ
    P_1 ← P_1 ∪ PointsOnLine(ℓ)
end for
for p ∈ P_1 do
    H = InferHyperplane(p)
    if TestHyperplane(H) then
        S_1 ← S_1 ∪ GetParams(H)
    else
        P_2 ← P_2 ∪ {p}
    end if
end for
return parameters S_1,
       unused sample points P_2
```

Sampling boundary points. In order to identify the boundaries B_z for z in layer 1, we begin by identifying a set of boundary points with at least one on each B_z. A randomly chosen line segment through input space will intersect some of the B_z – indeed, if it is long enough, it will intersect any fixed hyperplane with probability 1. We sample line segments ℓ in $\mathbb{R}^n_{\text{in}}$ and run `PointsOnLine` on each. Many sampling distributions are possible, but in our implementation we choose to sample segments of fixed (long) length, tangent at their midpoints to a sphere of fixed (large) radius.[3] This ensures that each of our sample lines remains far from the origin, where boundaries are in closer proximity and therefore more easily confused with one another (this will become useful in the next step). Let P_1 be the overall set of boundary points identified on our sample line segments.

Inferring hyperplanes. We now fit a hyperplane to each of the boundary points we have identified. For each $\mathbf{p} \in P_1$, there is a neuron z such that $\mathbf{p} \in B_z$. The boundary B_z is piecewise linear, with nonlinearities only along other boundaries, and with probability 1, $\mathbf{p}$ does not lie on a boundary besides B_z. Therefore, within a small enough neighborhood of $\mathbf{p}$, B_z is given by a hyperplane, which we call the *local hyperplane* at $\mathbf{p}$. If z is in layer 1, then B_z equals the local hyperplane. The subroutine `InferHyperplane` takes as input a point $\mathbf{p}$ on a boundary B_z and approximates the local hyperplane within which $\mathbf{p}$ lies. This algorithm proceeds by sampling many small line segments around $\mathbf{p}$, running `PointsOnLine` to find their points of intersection with B_z, and performing linear regression to find the equation of the hyperplane containing these points.

[2] These weighted averages speed up the search algorithm by biasing it towards the center of the segment, which is where we expect the most intersections given our choice of segments.

[3] The algorithm is not sensitive to the exact settings of these hyperparameters, so long as they are relatively large.

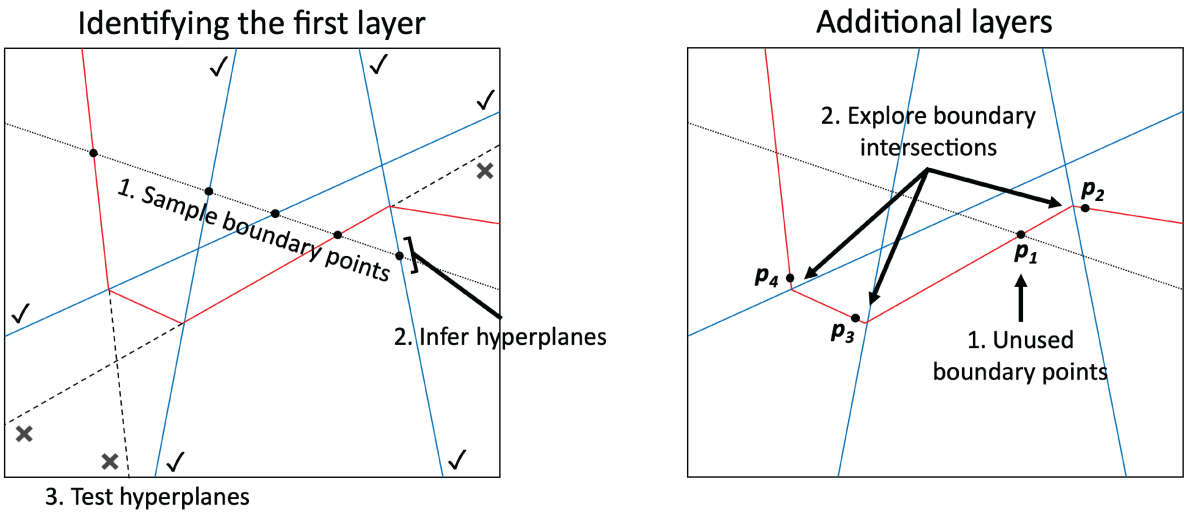

Figure 2. Schematic of our algorithms for identifying the first layer of $\mathcal{N}$ and the additional layers. We query the network to estimate the gradient $\nabla\mathcal{N}(\mathbf{x})$ of the network with respect to its inputs. Discontinuities in $\nabla\mathcal{N}(\mathbf{x})$ correspond to points on the boundary between linear regions. We begin by identifying all boundary points on a line through input space. For each such point $\mathbf{x}$, we use regression to infer the local behavior of the boundary B_z containing $\mathbf{x}$, and test whether B_z is a full hyperplane (which means z is in layer 1), or a bent hyperplane (which means z is in a later layer). For z in layer 1, we infer the weight vector from the equation of B_z. For z in a later layer, we infer the weights from the extent to which B_z bends when it intersects each $B_{z'}$ for z' in an earlier layer.

Testing hyperplanes. Not all of the hyperplanes we have identified are actually boundaries for neurons in layer 1, so we need to test which hyperplanes are contained in B in their entirety, and which are the local hyperplanes of boundaries that bend. The subroutine `TestHyperplane` takes as input a point $\mathbf{p}$ and a hyperplane H containing that point, and determines whether the entire hyperplane H is contained in the boundary B of the network. This algorithm proceeds by sampling points within H that lie far from $\mathbf{p}$ and applying `PointsOnLine` to a short line segment around each such point to check whether these points all lie on B. Applying `TestHyperplane` to those hyperplanes inferred in the preceding step allows us to determine those B_z for which z is in layer 1.

From hyperplanes to parameters. Finally, we identify the first layer of $\mathcal{N}$ from the equations of hyperplanes contained in B. The number of neurons in layer 1 is given simply by the number of distinct B_z that are hyperplanes. As we have observed, for $z = z_i^1$ in layer 1, the hyperplane B_z is given by $\mathbf{W}^1_{\cdot i}\mathbf{x} + \mathbf{b}^1_i = 0$. We can thus determine $\mathbf{W}^1_{\cdot i}$ and $\mathbf{b}^1_i$ up to multiplication by a constant. However, we have already observed that scaling $\mathbf{W}^1_{\cdot i}$ and $\mathbf{b}^1_i$ by a positive constant (while inversely scaling $\mathbf{W}^2_{i\cdot}$) is a network isomorphism (§3.1). Therefore, we need only determine the true sign of the multiplicative constant, corresponding to determining which side of the hyperplane is zeroed out by the ReLU. This determination of sign will be performed in §4.2.

4.2. Additional layers

We now assume that the weights $\mathbf{W}^1, \ldots, \mathbf{W}^{k-1}$ and biases $\mathbf{b}^1, \ldots, \mathbf{b}^{k-1}$ have already been determined within the network $\mathcal{N}$, with the exception of the sign choice for weights and biases at each neuron in layer $k-1$. We now show how it is possible to determine the weights $\mathbf{W}^k$ and biases $\mathbf{b}^k$, along with the correct signs for $\mathbf{W}^{k-1}$ and $\mathbf{b}^{k-1}$.

Closest boundary along a line. In this part of our algorithm, we will need the ability to move along a boundary to its intersection with another boundary. For this purpose, the subroutine `ClosestBoundary` will be useful. It takes as input a point $\mathbf{p}$, a vector $\mathbf{v}$ and the network parameters as determined up to layer $k-1$, and outputs the smallest $c > 0$ such that $\mathbf{q} = \mathbf{p} + c\mathbf{v}$ lies on B_z for some z in layer at most $k-1$. In order to compute c, we consider the region R within which $\mathbf{p}$ lies, which is associated with a certain pattern of active and inactive ReLUs. For each boundary B_z, we calculate the hyperplane equation which would define B_z were it to intersect R, due to the fixed activation pattern within R, and we calculate the distance from $\mathbf{p}$ to this hyperplane. While not every boundary B_z intersects R, the closest boundary does, allowing us to find the desired c.

Unused boundary points. In order to identify the boundaries B_z for z in layer k, we wish to identify a set of boundary points with at least one on each such boundary. However, in previous steps of our algorithm, a set P_{k-1} of boundary points was created, of which some were used in ascertaining the parameters of earlier layers. We now consider the subset

$P_k \subset P_{k-1}$ of points that were not found to belong to B_z, for z in layers 1 through $k-1$. These points have already had their local hyperplanes determined.

Algorithm 2 Additional layers

Input P_k and $S_1, \ldots, S_{k-1}$
Initialize $S_k = \{\}$
for $\mathbf{p}_1 \in P_{k-1}$ on boundary B_z **do**
  Initialize $A_z = \{\mathbf{p}_1\}$, $L_z = \mathcal{H}_z = \{\}$
  while $L_z \not\supseteq$ Layer $k-1$ **do**
    Pick $\mathbf{p}_i \in A$ and $\mathbf{v}$
    $\mathbf{p}', B_{z'} = \texttt{ClosestBoundary}(\mathbf{p}_i, \mathbf{v})$
    if $\mathbf{p}'$ on boundary **then**
      $A_z \leftarrow A_z \cup \{\mathbf{p}' + \epsilon\}$
      $L_z \leftarrow L_z \cup \{z'\}$
      $\mathcal{H}_z \leftarrow \mathcal{H}_z \cup \{\texttt{InferHyperplane}(\mathbf{p}_i)\}$
    else
      $P_k \leftarrow P_k \cup \{\mathbf{p}_1\}$; **break**
    end if
  end while
  if $L_z \supseteq$ Layer $k-1$ **then**
    $S_k \leftarrow \texttt{GetParams}(T_z)$
  end if
end for
return parameters S_k, unused sample points P_{k+1}

Exploring boundary intersections. Consider a point $\mathbf{p}_1 \in P_k$ such that $\mathbf{p}_1 \in B_z$. Note that B_z will, in general, have nonlinearities where it intersects $B_{z'}$ such that z' lies in an earlier layer than z. We explore these intersections, and in particular attempt to find a point of $B_z \cap B_{z'}$ for every z' in layer $k-1$. Given the local hyperplane H at $\mathbf{p}_1$, we pick a direction $\mathbf{v}$ along H and apply `ClosestBoundary` to calculate the closest point of intersection $\mathbf{p}'$ with $B_{z'}$ for all z' already identified in the network. (We discuss below how to pick $\mathbf{v}$.) Note that if z is in layer k, then $\mathbf{p}'$ must be on B_z as well as $B_{z'}$, while if z is in a later layer of the network, then there may exist unidentified neurons in layers below z and therefore B_z may bend before meeting $B_{z'}$. We check if $\mathbf{p}'$ lies on B_z by applying `PointsOnLine`, and if so apply `InferHyperplane` to calculate the local hyperplane of B_z on the other side of $B_{z'}$ from $\mathbf{p}_1$. We select a representative point $\mathbf{p}_2$ on this local hyperplane. We repeat the process of exploration from the points $\mathbf{p}_1, \mathbf{p}_2, \ldots$ until one of the following occurs: (i) a point of $B_z \cap B_{z'}$ has been identified for every z' in layer $k-1$ (this may be impossible; see §5.2), (ii) z is determined to be in a layer deeper than k (as a result of $\mathbf{p}'$ not lying on B_z), or (iii) a maximum number of iterations has been reached.

How to explore. An important step in our algorithm is exploring points of B_z that lie on other boundaries. Given a set of points $A_z = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m\}$ on B_z, there are several methods for picking a point $\mathbf{p}_i$ and direction $\mathbf{v}$ along the local hyperplane at $\mathbf{p}_i$ to apply `ClosestBoundary`. One approach is to pick a random point $\mathbf{p}_i$ from those already identified and a random direction $\mathbf{v}$; this has the advantage of simplicity. However, it is somewhat faster to consider for which z' the intersection $B_z \cap B_{z'}$ has not yet been identified and attempt specifically to find points on these intersections. One approach for this is to pick a missing z' and identify for which $\mathbf{p}_i$ the boundary $B_{z'}$ lies on the boundary of the region containing $\mathbf{p}_i$ and solve a linear program to find $\mathbf{v}$. Another approach is to pick a missing z' and a point $\mathbf{p}_i$, calculate the hyperplane H which would describe $B_{z'}$ under the activation pattern of $\mathbf{p}_i$, and choose $\mathbf{v}$ along the local hyperplane to $\mathbf{p}_i$ such that the distance to H is minimized. This is the approach which we take in our implementation, though more sophisticated approaches may exist and present an interesting avenue for further work.

From boundaries to parameters. We now identify layer k of $\mathcal{N}$, along with the sign of the parameters of layer $k-1$, by measuring the extent to which boundaries bend at their intersection. We are also able to identify the correct signs at layer $k-1$ by solving an overconstrained system of constraints capturing the influence of neurons in layer $k-1$ on different regions of input space. Theorem 4 formalizes the inductive step that allows us to go from what we know at layer $k-1$ (weights and biases, up to scaling and sign) to the equivalent set of information for layer k.

5. Discussion

5.1. Correctness and sample complexity

The following theorem (proven in Appendix C) establishes the validity of our algorithm above.

Theorem 4 (Algorithm proof of correctness)**.** *The above algorithm successfully recovers, up to permutation and scaling, the structure and weights of any deep ReLU network for which the assumptions of Theorem 2 hold. No prior knowledge of the weights, structure, or linear region boundaries is assumed, merely the ability to query the network.*

Specifically, the following holds true for fully connected ReLU networks $\mathcal{N}$ satisfying the Linear Region Assumption (§3.2), excluding a set of networks with measure zero: Suppose that the weights and biases of $\mathcal{N}$ are known up through layer $k-1$, with the exception that for each neuron in layer $k-1$, the sign of the incoming weights and the bias is unknown. Suppose also that for each z in layer k, there exists an ordered set of points $A_z = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m\}$ such that: (i) Each point lies on the boundary of B_z, and in (the interior of) a distinct region with respect to the earlier-layer boundaries already known; (ii) each point (except for $\mathbf{p}_1$) has a precursor in an adjacent region; (iii) for each such pair of points, the local hyperplanes of B_z are known, as is

the boundary $B_{z'}$ dividing them (z' in an earlier layer); (iv) the set of such z' includes all of layer $k-1$.

Then, it is possible to recover the weights and biases for layer k, with the exception that for each neuron, the sign of the incoming weights and the bias is unknown. It is also possible to recover the sign for every neuron in layer $k-1$.

Note that even when assumption (iv) of the Theorem is violated, the algorithm recovers the weights corresponding to whichever boundaries are successfully crossed, as we verify empirically in §6.

We expect the number of queries necessary to obtain weights and biases (up to sign) for the first layer should grow as $O(n_{\text{in}}(\sum_i n_i)\log n_1)$, which for constant-width networks is only slightly above the number of parameters being inferred. Namely, if the biases in the network are bounded above, then each sufficiently long line has constant probability of hitting a given hyperplane, suggesting $\log n_1$ lines are required according to a coupon collector-style argument. Hanin & Rolnick (2019a) prove that under natural assumptions, the number of boundary points intersecting a given line through input space grows linearly in the total number of neurons in the network. Finally, each boundary point on a line requires $O(n_{\text{in}})$ queries in order to fit a hyperplane.

For deeper layers, the number of queries depends on the approach taken to explore the intersections between boundaries. It is possible, depending on the arrangement of intersections, for the number of queries to grow linearly in the number of parameters, as each weight can be inferred by examining a single intersection between boundaries.

5.2. Assumptions

It is possible that for some neurons z and z' in consecutive layers, there is no point of intersection between the boundaries B_z and $B_{z'}$ (or that this intersection is very small), making it impossible to infer the weight between z and z' by our algorithm. Some such cases represent an ambiguity in the underlying network – an additional isomorphism to those described in §3.1. Namely, $B_z \cap B_{z'}$ is empty if one of the following cases holds: (1) whenever z is active, z' is inactive; (2) whenever z is active, z' is active; (3) whenever z is inactive, z' is inactive; or (4) whenever z is inactive, z' is active. In case 1, observe that a slight perturbation to the weight w between z and z' has no effect upon the network's output; thus w is not uniquely determined. Cases 2-4 present a more complicated picture; depending on the network, there may or may not be additional isomorphisms.

For simplicity in our algorithm, we have not considered the relatively rare cases where boundaries B_z are disconnected or bounded. If B_z is disconnected, then it may not be possible to find a connected path along it that intersects all boundaries arising from the preceding layer. In this case, it is simple to infer that two independently identified pieces of the boundary belong to the same neuron to infer the full weight vector. Next, if B_z is bounded for some z, then it is a closed $(d-1)$-dimensional surface within d-dimensional input space[4]. While our algorithm requires no modification in this case, bounded B_z may be more difficult to find by intersection with randomly chosen lines, and a more principled sampling method may be helpful.

5.3. Simpler approaches

It may be wondered whether simpler approaches would suffice to solve the problem. In particular, the method we describe uses different algorithms for the first layer and for subsequent layers. One might wonder why, once the first $k-1$ layers have been identified, it is not possibly simply to apply our first-layer algorithm to the n_{k-1}-dimensional "input space" arising from activations at layer $k-1$. Unfortunately, this is not possible in general, as this would require the ability to evaluate layer k for arbitrary settings of layer $k-1$. ReLU networks are hard to invert, and therefore it is unclear how one could manufacture an input for a specified layer $k-1$ activation, even while knowing the parameters for the first $k-1$ layers.

It is also not possible simply to use gradient descent to train a new network to imitate the output of the desired network. Even if the architecture of the network is known in advance, such an approach fails completely to recover the original weights.

5.4. Other architectures

While we have expressed our algorithm in terms of multi-layer perceptrons with ReLU activation, it also extends to various other architectures of neural network. Other piecewise linear activation functions admit similar algorithms. For a network with convolutional layers, it is possible to use the same approach to infer the weights between neurons, with two caveats: (i) As we have stated it, the algorithm does not account for weight-sharing – the number of "neurons" in each layer is thus dependent on the input size, and is very large for reasonably sized images. (ii) Pooling layers *do* affect the partition into activation regions, and indeed introduce new discontinuities into the gradient; our algorithm therefore does not apply.

For skip connections as in ResNets (He et al., 2016), our algorithm should hold with slight modifications, which we outline here. As in a multi-layer perceptron, each boundary B_z is a bent hyperplane that bends when it intersects $B_{z'}$ for z' at an earlier layer than z. However, potential weights

[4] For 2D input, such B_z must be topological circles, but for higher dimensions, it is conceivable for them to be more complicated surfaces, such as toroidal polyhedra.

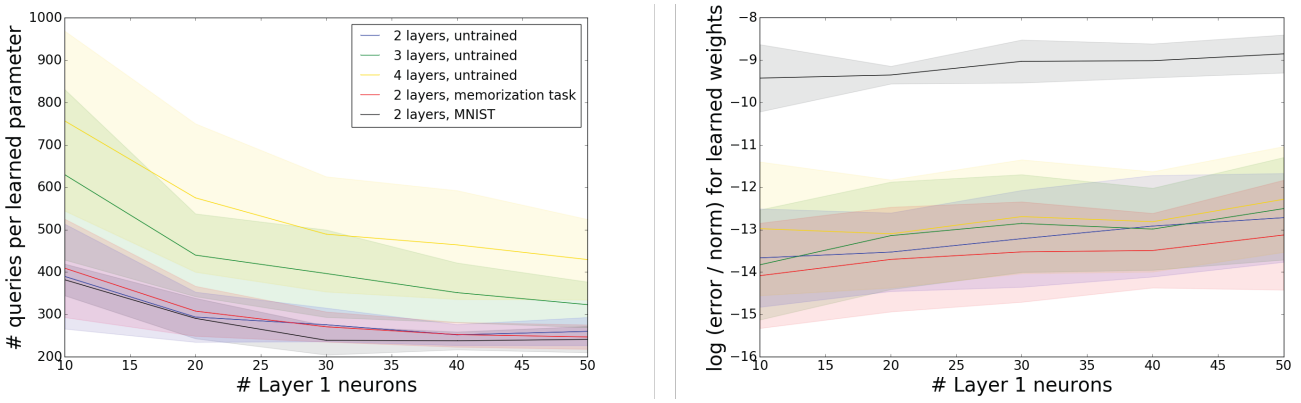

Figure 3. Results of our first-layer algorithm, applied to networks with two or more hidden layers as the width of the first layer varies. All other layers have fixed width 10. Untrained networks have input and output dimension 10, those trained on the memorization task have input dimension 10 and output dimension 2, and those trained on MNIST have input dimension 784 and output dimension 10. Left: The number of queries issued by our algorithm per parameter identified in the network $\mathcal{N}$; the algorithm is terminated once the desired number of neurons have been identified. Right: Log normalized error $\log(||\hat{\mathbf{W}}^1 - \mathbf{W}^1||_2/||\hat{\mathbf{W}}^1||_2)$ for $\hat{\mathbf{W}}^1$ the approximated weights. Weight vectors were scaled to unit norm to account for isomorphism (see §3.1). Curves are averaged over 5 runs in the case of MNIST and 40 runs otherwise, with standard deviations between runs shown as shaded regions.

must in this case be considered between neurons in any two different layers. Deriving the weights for such skip connections is somewhat more complex than for multi-layer perceptrons, as the "bend" is influenced not merely by the skip connection but by the weights along all other paths between the two neurons through the network. Thus, it is necessary to move backward through the network – for a neuron in layer k, one must first derive the weights of connections arising from the preceding layer $k-1$, then from $k-2$, and so on.

6. Experiments

We demonstrate the success of our algorithm on both untrained and trained networks. In keeping with literature on ReLU network initialization (He et al., 2015; Hanin & Rolnick, 2018), networks were initialized using i.i.d. normal weights with variance $2/\text{fan-in}$ and i.i.d. normal biases with unit variance. Networks were trained on either the MNIST dataset ($n_{\text{in}} = 784, n_{\text{out}} = 10$) or a memorization task of 1000 "datapoints" ($n_{\text{in}} = 10, n_{\text{out}} = 2$) with coordinates drawn i.i.d. from a unit Gaussian and given arbitrary binary labels. Training was performed using the Adam optimizer (Kingma & Ba, 2014) and a cross-entropy loss applied to the softmax of the final layer, over 20 epochs for MNIST and 1000 epochs for the memorization task. The trained networks (when sufficiently large) were able to attain near-perfect accuracy.

We observe that both the first-layer algorithm and additional-layer algorithm identified weights and biases to within extremely high accuracy (see Figs. 3 and 4). In order to compare estimated and true parameters, we scaled weights at each neuron to unit norm and accounted for possible permutations of the neurons within a layer. (As described in §3.1, these transformations do not change the function computed by the network and represent unavoidable isomorphisms in recovering the network.) Figures show the log normalized error $\log(||\hat{\mathbf{W}}^k - \mathbf{W}^k||_2/||\hat{\mathbf{W}}^k||_2)$ between true weights $\mathbf{W}^k$ and approximated weights $\hat{\mathbf{W}}^k$ and likewise the log normalized error $\log(||\hat{\mathbf{b}}^k - \mathbf{b}^k||_2/||\hat{\mathbf{b}}^k||_2)$ between true biases $\mathbf{b}^k$ and approximated weights $\hat{\mathbf{b}}^k$.

In keeping with our analysis in §5.1, the number of queries needed to recover the first layer of a network scales linearly with the number of neurons in that layer (Fig. 3). In the second layer, a small fraction of weights are sometimes not identified (see §5.2 for a discussion of reasons why such behavior is inevitable); even in such cases, the algorithm is able to correctly predict the remaining parameters (Fig. 4).

7. Conclusion

In this work, we have proven that it is possible to recover the architecture, weights, and biases of deep ReLU networks from the boundaries between linear regions defined by the network. In many cases, we show that it is possible to reconstruct these boundaries and thus the network itself merely by querying the network's output on certain inputs. Networks can, we observe, often be reconstructed unambiguously up to permutation of neurons within each layer and scaling of weights and biases at individual neurons. In some cases, there are additional isomorphisms, in which case our algorithm is able to reconstruct a significant fraction of the parameters of the network.

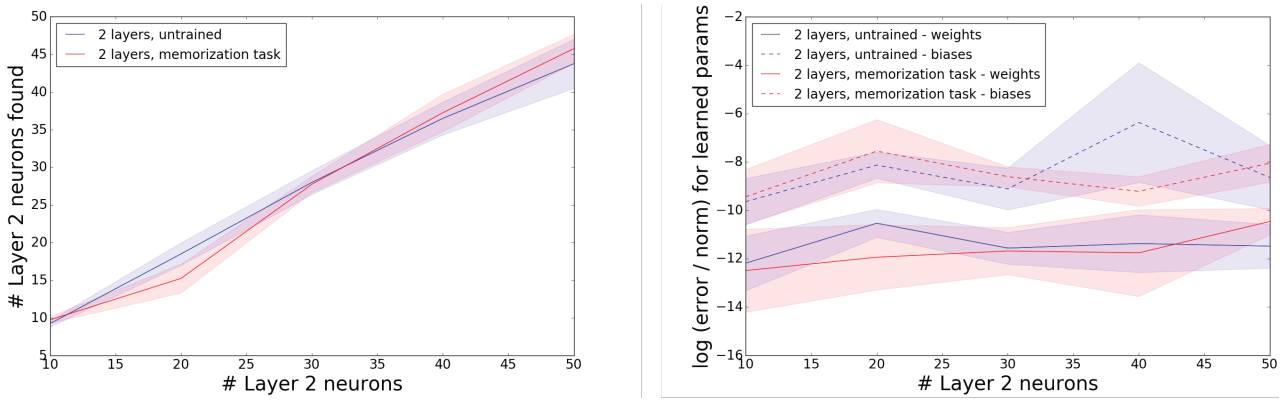

Figure 4. Results of our algorithm for additional layers, applied to networks with two layers, the first layer of width 10, as the width of the second layer varies. Left: Number of estimated layer-2 neurons. Right: Log normalized error between estimated and corresponding true neurons for approximated layer-2 weights and biases. Curves are averaged over 4 runs, with standard deviations shown as shaded regions.

Our approach works for a wide variety of networks, though not all. It is limited to ReLU or other piecewise linear activation functions, though we believe it possible that a continuous version of this method could potentially be developed in future work for use with sigmoidal activation. If used with convolutional layers, our method does not account for the symmetries of the network and therefore scales with the size of the input as well as the number of features, resulting in high computation. Finally, the method is not robust to defenses such as adding noise to the outputs of the network, and therefore can be thwarted by a network designer that seeks to hide their architecture and weights.

We believe that the methods we have introduced here will lead to considerable advances in reverse-engineering neural networks, both in the context of deep learning and, more speculatively, in neuroscience. While the implementation we have demonstrated here is effective in small instances, we anticipate future work that optimizes these methods for efficient use with different architectures and at scale.

Acknowledgments

We are grateful to Boris Hanin and the anonymous reviewers for their helpful comments. D.R. was supported by the US National Science Foundation, grants Nos. 1803547 and 1910864.

References

Chance, F. S., Abbott, L. F., and Reyes, A. D. Gain modulation from background synaptic input. *Neuron*, 35(4): 773–782, 2002.

Fefferman, C. Reconstructing a neural net from its output. *Revista Matemática Iberoamericana*, 10(3):507–555, 1994.

Ge, R., Kuditipudi, R., Li, Z., and Wang, X. Learning two-layer neural networks with symmetric inputs. In *International Conference on Learning Representations (ICLR)*, 2019.

Goel, S. and Klivans, A. Learning neural networks with two nonlinear layers in polynomial time. *Preprint arXiv:1709.06010*, 2017.

Goel, S., Kanade, V., Klivans, A., and Thaler, J. Reliably learning the ReLU in polynomial time. In *Conference on Learning Theory (COLT)*, 2017.

Hanin, B. and Rolnick, D. How to start training: The effect of initialization and architecture. In *Neural Information Processing Systems (NeurIPS)*, 2018.

Hanin, B. and Rolnick, D. Complexity of linear regions in deep networks. In *International Conference on Machine Learning (ICML)*, 2019a.

Hanin, B. and Rolnick, D. Deep ReLU networks have surprisingly few activation patterns. In *Neural Information Processing Systems (NeurIPS)*, 2019b.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *IEEE International Conference on Computer Vision (ICCV)*, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Heggelund, P. Receptive field organization of simple cells in cat striate cortex. *Experimental Brain Research*, 42(1): 89–98, 1981.

Jagielski, M., Carlini, N., Berthelot, D., Kurakin, A., and Papernot, N. High-fidelity extraction of neural network models. *Preprint arXiv:1909.01838*, 2019.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *Preprint arXiv:1412.6980*, 2014.

Kording, K. P., Kayser, C., Einhauser, W., and Konig, P. How are complex cell properties adapted to the statistics of natural stimuli? *Journal of neurophysiology*, 91(1): 206–212, 2004.

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

Oh, S. J., Schiele, B., and Fritz, M. Towards reverse-engineering black-box neural networks. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pp. 121–144. Springer, 2019.

Phuong, M. and Lampert, C. H. Functional vs. parametric equivalence of ReLU networks. In *International Conference on Learning Representations (ICLR)*, 2019.

Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., and Sohl-Dickstein, J. On the expressive power of deep neural networks. In *International Conference on Machine Learning (ICML)*, 2017.

Telgarsky, M. Representation benefits of deep feedforward networks. *Preprint arXiv:1509.08101*, 2015.