Inverse Abstraction of Neural Networks Using Symbolic Interpolation

Sumanth Dathathri,[1*] Sicun Gao,[2] Richard M. Murray[1]
[1]Computing and Mathematical Sciences, California Institute of Technology
[2]Computer Science and Engineering, University of California, San Diego

Abstract

Neural networks in real-world applications have to satisfy critical properties such as safety and reliability. The analysis of such properties typically requires extracting information through computing *pre-images* of the network transformations, but it is well-known that explicit computation of pre-images is intractable. We introduce new methods for computing compact symbolic abstractions of pre-images by computing their overapproximations and underapproximations through all layers. The abstraction of pre-images enables formal analysis and knowledge extraction without affecting standard learning algorithms. We use inverse abstractions to automatically extract simple control laws and compact representations for pre-images corresponding to unsafe outputs. We illustrate that the extracted abstractions are interpretable and can be used for analyzing complex properties.

1 Introduction

Neural networks have significant potential as a key building block of intelligent systems. However, characterizing the exact behaviors of complex neural networks is an extremely difficult task, which poses a major challenge to their use in safety-critical systems. Recent work on verifying neural networks such as (Zakrzewski 2001; Bunel et al. 2018; Katz et al. 2017) has focused on developing faster algorithms for validating or falsifying formal properties of whole neural networks directly through their encoding as constraint satisfaction problems. These approaches are designed for generating counter-examples to (or for verifying) specific properties for piecewise-linear neural networks. An alternative approach for analysis is to decompose the large networks and perform the analysis in a modular way, which is a standard practice in software program analysis (Hoare 1969). Such decomposition often provides more information than simple monolithic analysis and enables us to verify complex properties in a tractable manner. A crucial task for enabling such modular analysis is that we must be able to represent and manipulate *pre-images* of programs, or computable functions in general. On neural networks, this translates to being able to propagate sets of the network outputs backwards through individual neural layers (as real-valued functions), eventually to the input domain. However, this is in principle harder than direct constraint solving, because of the requirement of representing and manipulating high-dimensional geometric shapes that often do not have polynomial-size representations. Thus, the important question is how to efficiently compute approximate representations (abstractions) of such pre-image sets, so that they are both compact and precise enough for enabling formal analysis, interpretation and knowledge/policy extraction.

In this paper, we develop algorithms for computing *symbolic abstractions* of pre-images of neural networks. We bypass the difficulty of representing the exact pre-images, by maintaining both overapproximations and underapproximations that can be compactly represented as symbolic constraints. We leverage a recent algorithm for computing symbolic *interpolants* (Albarghouthi and McMillan 2013), where an extension of Farkas' lemma is used to *learn* interpolants that have simple structures. The techniques are applicable because the concepts that are learned by neural networks are often simple (Ba and Caruana 2014). We exploit the network structures and propagate pre-images of subsets of the output space through each layer to the input space. We enhance scalability of the algorithms on piecewise-linear neural networks by designing heuristics for the specific symbolic forms of the abstractions.

In experiments, we focus on knowledge/policy analysis and extraction for two control environments: cart-pole and swimmer. We show that for the multilayer perceptron (MLP) network policies trained through standard reinforcement learning algorithms, we can extract knowledge in the form of compact abstractions. For cart-pole, the extracted policy achieves a perfect score. Using the extracted policy we are able to formally verify/falsify certain complex safety properties. For swimmer, we show how high torque outputs are mapped to a compact representation in the input space. We believe these techniques will be important for analyzing learning-enabled components in control applications.

Related Work *Model Extraction:* There has been recent work in extracting verifiable and explainable models from trained networks. In this direction, (Bastani, Pu, and Solar-Lezama 2018) introduce an imitation learning-based ap-

[*]Correspondence: sdathath@caltech.edu. The work is supported by DARPA Assured Autonomy, NSF CNS-1830399 and the VeHICaL project (NSF grant #1545126).

proach to generate data to train a decision tree that allows for easier verification. In (Verma et al. 2018), the authors use a trained neural network to guide a local search over programmatic policies that are human-readable and more verifiable than the complex neural networks themselves. In contrast to both of the above described approaches that focus on reinforcement learning, our work introduces a general tool for the analysis of neural networks by computing abstractions. Additionally, our approach does not rely on learning simpler models or modifying the training process, and extracts symbolic abstractions completely algorithmically for the original neural network. In (Mahendran and Vedaldi 2015), the authors propose a framework for inverting representations. They reconstruct the input image given an encoding of the image. This is different from our work, where given a set of outputs we compute (approximately) the entire pre-image. Interpolants have frequently been used to help improve the scalability of software verification, automated theorem proving, and constraint solving (Bonacina and Johansson 2015; Dathathri et al. 2017; Kroening and Weissenbacher 2011).

Input-Output Abstraction: There has been considerable progress in the direction of computing forward abstractions i.e. ranges of outputs for a given set of inputs and using them to verify simple reachability/robustness properties. In (Wang et al. 2018), symbolic-interval analysis coupled with heuristics for tightening the computed bounds is used to compute approximate output sets corresponding to a set of box-constrained inputs. In (Gehr et al. 2018), in a similar spirit, abstract-interpretation is leveraged to compute approximate output sets for a given set of inputs. Both of these abstraction-based approaches approaches are able to certify robustness/reachability properties significantly faster than solver-based approaches, such as (Katz et al. 2017). However, reasoning about complex properties (for e.g., general specifications in linear-arithmetic over the inputs and outputs of the neural network) is still not possible with these output abstractions. Additionally, these abstractions are also not well suited for policy/knowledge extraction. For instance, for a cart-pole controller, it is not directly possible to answer the question: *What makes the cart go left?* with the above discussed techniques. In contrast, our inverse abstractions can be used for extracting such knowledge from trained networks (see Section 5.2), and the computed compact abstractions can then be used with other solvers for further complex analysis. However, if only a few simple robustness/reachability properties are to be verified, direct verification is preferable instead of our approach that has to represent and manipulating complex symbolic sets.

2 Preliminaries

2.1 Neural Networks and Constraints

Consider a neural network f with n layers. That is, $f(x) = h.g_n.g_{n-1}.\ldots.g_1(x)$ where g_r is the transfer function representing the map from the input to the output space for layer r and $h : \mathbb{R}^k \to Y$ is a map from the logits to the k class-labels (e.g. $\arg\max$). For example, if a network has $n-1$ layers with ReLU activation and a final linear layer:

$$\begin{aligned} g_r(z) &= \max(W_r x + b_r, 0) && \forall r \in 1, 2, \ldots, n-1 \\ g_r(z) &= W_r x + b_r && r = n. \end{aligned}$$

For simplicity, we only discuss neural networks that map to a discrete set of outputs but the approach is valid even when the network produces a continuous set of outputs. By $y_i^f(x)$, we refer to the output from the i^{th} layer of the neural network i.e $y_i^f(x) = g_i.g_{i-1}.\ldots.g_1(x)$. Often, in classification tasks, the outputs from the layer g_n are fed through a softmax layer to normalize the scores. Here, in our analysis, we do not consider the softmax layer as it preserves the ordering amongst scores corresponding to the different classes. For a vector $z \in \mathbb{R}^m$, $\arg\max_i\{z_i\} = \arg\max_i\{\texttt{softmax}(z)_i\}$.

Piecewise-linear neural networks (without the softmax layer) can be expressed as constraints in the theory of quantifier free linear rational arithmetic (QFLRA). This includes neural networks with activation functions that are piecewise-linear (e.g. ReLU, Leaky ReLU, MaxOut, MaxPool). We follow the encoding described in (Ehlers 2017). For example, a ReLU node $y = \max(0, x)$ is written as:

$$(y = 0 \land x \leq 0) \lor (y = x \land x \geq 0).$$

The entire network is similarly encoded into QFLRA.

As a slight abuse of notation we interchangeably use $g(z)$ to represent the map of z through the function g, and the first-order logic constraint that enforces the same. For example, consider the function $g(x) = \max(0, w^T x)$. We interchangeably use $y = g(x)$ to also represent the constraint $(y = w^T x \land w^T x \geq 0) \lor (y = 0 \land w^T x < 0)$. For a satisfying assignment (y, x) for this constraint, we have $y = g(x)$. For a formula φ that has a vector of free variables s, we write $x \models \varphi$ if φ interprets to `True` when we set $s = x$.

Definition 1 (Pre-images) *Consider a neural network f. Let X be the domain and Y be the codomain. The preimage of a set $S \subset Y$ for the neural network f is the set $\{x \in X | f(x) \in S\}$.*

For example, consider a neural network based cart-pole controller with the action space $\{\texttt{left}, \texttt{right}\}$. The pre-image corresponding to $S = \{\texttt{right}\}$ is the set of observations that cause the controller to output `right` as the action. In the remainder of this work, we refer to this pre-image of set S for the neural network f as $\text{Pre}_f(S)$. The exact pre-image of the network for a given set S, in the worst case, can have exponentially many linear regions. To overcome this we consider abstractions that provably (over) underapproximate the exact pre-image $\text{Pre}_f(S)$. The restriction on the structure of S for our work is that it has to be expressible in QFLRA, which includes all constraints that have half-spaces as atoms combined with Boolean operators.

2.2 Symbolic Interpolation

Symbolic interpolation is a well-studied concept in propositional and first-order logic (Craig 1957). Given two quantifier-free first-order formulas A and B, such that $A \land B$ is unsatisfiable, a Craig Interpolant I is a formula satisfying:

- $A \implies I$;
- $B \wedge I \implies \bot$;
- I only contain variables that are shared by A and B.

Intuitively, the interpolant I provides an overapproximation of A that is still precise enough to exhibit its conflict with B, and does not contain redundant information that involves any variable that is not shared by both A and B.

Definition 2 (Overapproximation) *We call $\alpha(x)$ an overapproximator of $A(x)$ if $\forall x.A(x) \implies \alpha(x)$.*

Definition 3 (Underapproximation) *Conversely, we call $\beta(x)$ the underapproximator of $A(x)$ if $\forall x.\beta(x) \implies A(x)$.*

When $A \wedge B$ is not satisfiable, Craig's interpolation theorem guarantees the existence of an interpolant I such that I overapproximates A and $\neg I$ overapproximates B. These interpolants have found application in compositional approaches to program-verification and SMT solving. In our work, we build on the algorithm from (Albarghouthi and McMillan 2013) for computing interpolants, as opposed to other approaches based on lazy SMT that produces complex interpolants. The intuition behind this choice is that simpler interpolants are more likely to provide general explanations corresponding to the neural network's parameters and the task for which the network was trained, rather than complex ones that may overfit the specific instantiation. Further, simpler interpolants provide the added advantage of being easier to reason over using automated reasoning engines (e.g. `z3`).

3 Inverse Abstraction of Neural Networks

We seek to compute approximations for the pre-image that closely approximate the pre-image, but are provable (super) subsets of the pre-image. The fact that these are provable (over) underapproximations (unlike the approximated models in (Bastani, Pu, and Solar-Lezama 2018; Verma et al. 2018)) allows us to prove properties that hold for the neural network itself. For example, suppose we wish to prove a property that for every input from some set W, the corresponding output from the neural network does not belong to the set S. By computing an overapproximation O for $\mathrm{Pre}_f(S)$ and showing that $O \wedge W$ is not satisfiable we have verified the property. Similarly, if the property was that every output belongs to some set S, then by computing the underapproximation U for $\mathrm{Pre}_f(S)$ and showing that $W \implies U$ is valid, we have verified the property.

Here, we give a brief overview of the algorithm for computing the overapproximation of the pre-image for set S. Consider the neural network described earlier. Let $p_n^{(f,S)}$ be the set of inputs to layer g_n of the neural network that lead to the output being in set S. Similarly, let $p_{n-1}^{(f,S)}$ be the set of inputs to layer g_{n-1} of the neural network that result in outputs in S. Note that $p_n^{(f,k)}$ is the set of assignments to s that satisfy:

$$h(g_n(s)) \models S.$$

For the other layers ($r < n$), we can iteratively define $p_r^{(f,S)}$ as assignments to s that satisfy:

$$p_r^{(f,S)} = \{s \mid g_r(s) \models p_{r+1}^{(f,S)}\} \tag{1}$$

The core idea is to begin by computing an approximate representation of $p_n^{(f,k)}$, and using this to then compute the approximations for $p_{n-1}^{(f,k)}$. And, then by iterating through the layers of the network we can compute an approximation for $\mathrm{Pre}_f(S)$. Computing these approximations involves proving the interpolation condition (See Section 3.1). We could compute the approximation across the entire network, but proving the interpolation condition for the entire network is computationally expensive. This is because the worst-case complexity of proving properties for piecewise-linear networks scales exponentially in the number of nodes under consideration (Katz et al. 2017). The layer-wise approach breaks down the problem, where we compute approximations for each layer. This requires proving properties across one layer at a time – can be further simplified to computing approximations over sets of nodes instead of entire layers.

3.1 Computing Overapproximations

Here we outline how to compute a useful overapproximation of $p_r^{(f,S)}$, assuming we have the overapproximation of $p_{r+1}^{(f,S)}$. Denote the over-approximation of $p_{r+1}^{(f,S)}$ as $O_{r+1}^{(f,S)}$ with $O_{n+1}^{(f,S)} = S$. First, consider a set of randomly chosen points $\bar{X}$, either by sampling from the input domain or from the training data. Let $X_S \subseteq \bar{X}$ be the set of points such that for every $x \in X_S$, $f(x) \notin S$. Introduce the auxiliary free variable vector $\bar{p}_r$ and construct the formula:

$$\phi_{r-1} := \bigvee_{\bar{x} \in X_S} \left(\bar{p}_r = y_{r-1}^f(\bar{x})\right). \tag{2}$$

This formula allows $\bar{p}_r$ to assume the value of the output at layer g_{r-1} corresponding to inputs from X_S. Recall that $y_{r-1}^f(x) = g_{r-1}.g_{r-2}\ldots g_1(x)$ i.e. $y_{r-1}^f(x)$ is the vector of activation values corresponding to x from layer $r-1$ of the network. Now, consider the formula:

$$\xi_r := g_r(\bar{p}_r) \models O_{r+1}^{(f,S)}. \tag{3}$$

Note that the set of valid assignments for $\bar{p}_r$ represents an overapproximation of the set $p_r^{(f,S)}$. This is because:

$$\left(g_r(\bar{p}_r) \models p_{r+1}^{(f,S)}\right) \implies \left(g_r(\bar{p}_r) \models O_{r+1}^{(f,S)}\right),$$

which follows from $O_{r+1}^{(f,S)}$ overapproximating $p_{r+1}^{(f,S)}$ and as a consequence of equation (1).

Lemma 4 *If $O_{r+1}^{(f,S)} \wedge \phi_r$ is unsatisfiable, then the formula $\xi_r \wedge \phi_{r-1}$ is unsatisfiable for each $r \in \{1, 2, \ldots, n\}$.*

Proof Assume $O_{r+1}^{(f,S)} \wedge \phi_r$ is unsatisfiable. Suppose $\exists \bar{p}_r$ satisfying ϕ_{r-1} and ξ_r. By definition of ϕ_r, we have $g_r(\bar{p}_r) \models \phi_r$ and by eq. (3), $g_r(\bar{p}_r) \models O_{r+1}^{(f,S)}$. This results in a contradiction since $O_{r+1}^{(f,S)} \wedge \phi_r$ is unsatisfiable.

Lemma 5 *If $O_{r+1}^{(f,S)} \wedge \phi_r$ is unsatisfiable, $\exists I_r$ that satisfies the following:*

$$p_r^{(f,S)} \implies \xi_r \implies I_r, \tag{4}$$

$$I_r \implies \neg\phi_{r-1}. \tag{5}$$

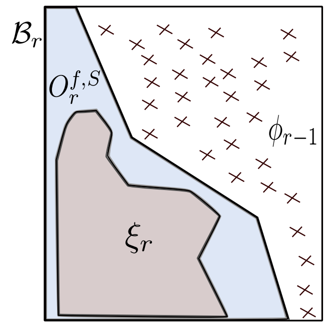

Figure 1: Illustration of the approach. ξ_r is the set of inputs to layer $g_r(.)$ that map into the overapproximating abstraction for the subsequent layer $g_{r+1}(.)$. ϕ_{r-1} is a set of inputs to layer $g_r(.)$ sampled so that the neural network maps them to outside the set S, where S is the set whose pre-image is being abstracted.

Proof Lemma 4 and Craig's Interpolation theorem guarantee the existence of an I_r satisfying the two conditions.

Lemma 5 guarantees the existence of an overapproximator of $p_r^{(f,S)}$ that is disjoint from the set of sampled points that map to the complement of S. This ensures that the overapproximations do not get arbitrarily slack as we propagate the interpolants through the layers and remain tight. Further, since we use the algorithm from (Albarghouthi and McMillan 2013) that is designed to compute simple interpolants, they are likely to generalize to other data points. A formal description is provided in Algorithm 1. Note that I_r is only a function of the $\bar{p}_r$ (the only shared free variables between ϕ_{r-1} and ξ_r). Figure 1 outlines the setup for a single layer.

3.2 Bounding the Problem

The interpolants computed that serve as overapproximations are in the disjunctive normal form (DNF) with the atoms being half-spaces. The convergence of the algorithm for computing the interpolant can be made faster by restricting the domain in which ϕ_{r-1} and ξ_r need to be separated by the interpolant. Given a lower bound (l_r) and an upper bound (u_r) on the outputs y_r^f, we can construct an additional constraint $\mathcal{B}_r$ defined as:

$$\mathcal{B}_r := (\bar{p}_r \leq u_r) \wedge (\bar{p}_r \geq l_r).$$

We then compute the interpolant such that $(\mathcal{B}_r \wedge \xi_r) \implies I_r$ and $I_r \implies \neg\phi_{r-1}$. To compute u_r and l_r we use the relaxation proposed in (Ehlers 2017). In most tasks, the inputs come from a bounded domain. For example in image processing, pixels have values ranging between 0 and 255. These bounds can then be propagated through the network as in (Ehlers 2017) using a convex relaxation of the network. Every ReLU node $y = \max(0, x)$ that behaves non-linearly is approximated with its convex hull:

$$y \geq 0, y \geq 0, y \leq u_x \frac{x - l_x}{u_x - l_x} \tag{6}$$

Using this relaxation for the non-linear constraints results in a linear program (LP). By optimizing over the resulting LP, the bounds for the nodes can be computed in a sequential manner, starting with the input layer. These bounds can further be tightened by splitting the input domain as in (Bunel et al. 2018), but for our work we do not split the input domain. As an added benefit, bounding the problem makes checking the interpolation condition significantly faster.

Algorithm 1 Computing compact abstractions

```
 1: procedure ABSTRACTION ROUTINE
 2:         ▷ Returns Simple Overapproximator of $\mathrm{Pre}_f(S)$
 3:     Compute $\forall \bar{x} \in X_S, y_1^f(\bar{x}) = g_1(\bar{x})$
 4:     Compute $l_1$, $u_1$ and $\mathcal{B}_1$
 5:     for r=2 ... n do
 6:         Compute $\forall \bar{x} \in X_S, y_r^f(\bar{x}) = g_r(\bar{x})$
 7:         Compute $l_r$, $u_r$ and construct $\mathcal{B}_r$
 8:     for r=n ... 1 do
 9:         Construct $\phi_{r-1}$ (See equation (2))
10:         Construct $\xi_r \wedge \mathcal{B}_r$ (See equation (3))
11:         Compute $I_r$ satisfying equations (4) and (5)
12:         Set $O_r^{(f,S)} = I_r$
    return $O_1^{(f,S)}$
```

Theorem 6 *(Soundness and Completeness) Algorithm 1 always terminates to return an overapproximator $O_1^{(f,S)}$ of $\mathrm{Pre}_f(S)$. Further, $O_1^{(f,S)}$ is disjoint from X_S.*

Proof At $r = n$, by construction we have $O_{r+1}^{(f,S)} \wedge \phi_r$ is not satisfiable. Lemmas 4 and 5 guarantee the existence of an overapproximator (interpolant) satisfying equations (4) and (5). The algorithm from (Albarghouthi and McMillan 2013) is both sound and complete, and hence is guaranteed to find an interpolant if one exists. For $r = n - 1$, we again have that $O_{r+1}^{(f,S)} \wedge \phi_r$ is not satisfiable since $O_{r+1}^{(f,S)} \implies \neg\phi_r$ (equation (5)). Repeating the arguments above, for every $r \leq n$, we compute $O_{r+1}^{(f,S)}$ (I_r) satisfying equations (4) and (5). Hence, Algorithm 1 terminates to return an overapproximator for $\mathrm{Pre}_f(S)$ (equation (4)) that is disjoint from X_S (equation (5)).

3.3 2D example

To illustrate the ideas developed above, we introduce a small example. Consider a simple-neural network $f : \mathbb{R}^2 \to \mathbb{R}^4$, with 2 hidden layers with 10 ReLU nodes in each layer. The network is trained to predict which quadrant a point x belongs to and achieves an accuracy of 99.65% on hold-out data. Here, $\arg\max(f(x))$ represents the quadrant x belongs to. Along each dimension, the input is restricted to the domain $[-1, 1]$. Here, we are interested in computing a compact representation of the set the network classifies as the third quadrant. To compute an overapproximation, we sample a set of 150 points that are classified as being outside the third quadrant by the neural network. Here, $O_n^{f,S} = \bigwedge_{j=1,2,4} (y_3 > y_j)$ where $y = f(x)$ is the output of the neural network. To compute the underapproximation, we sample 50 points that are classified as being in the third

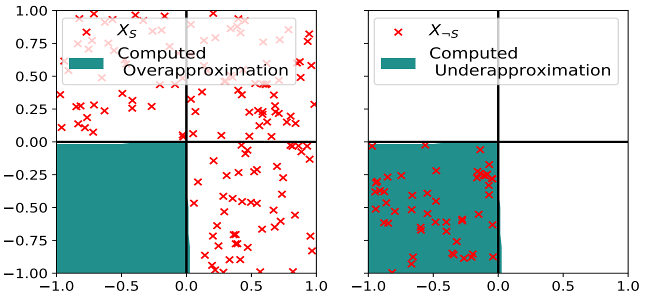

Figure 2: Left: An overapproximation of the region classified by the network as the third quadrant. Right: An underapproximation of the region classified by the network as the third quadrant. The overapproximation and the underapproximation are close to each other. It can be observed that in both the under and the overapproximations, parts of the 3rd and the 4th quadrant are misclassified – this indicates faulty behavior for the neural network f.

quadrant. Here, $O_n^{f,S} = \bigvee_{j=2,3,4} (y_3 < y_j)$ and Figure 2 depicts the over and under-approximations computed. The overapproximation computed is a union of 5 polytopes while the underapproximation computed consists of 2 polytopes.

4 Algorithms

The core of the computational effort in Algorithm 1 comes from Line 11 where the interpolant I_r is computed. Computing the interpolant for two constraints A and B has the following key steps:

- Sampling polytopes satisfying A and B and separating them with Farkas' lemma,
- Checking the conditions $A \implies I$ and $I \implies \neg B$, and generating counter examples (if any),
- Merging and splitting of the polytopes into sets, which are then separated by Farkas' lemma.

Counterexample guided abstraction refinement (CEGAR)(Clarke et al. 2003) – checking the interpolation condition and generating counter-examples to refine the abstraction – is integral to computing the interpolants. The sampling at the first stage of CEGAR can done with training samples, and subsequently an oracle can be used to find counter-examples to the condition $A \implies I$ and $I \implies \neg B$. Checking the conditions $A \implies I$ and $I \implies \neg B$ using an external oracle turns out to be the most expensive part of the algorithm.

Note that in our work A encodes the behavior of a layer of the neural network (with the nonlinearity), and conventional SMT solvers are inefficient at verifying properties for non-linear neural networks. For checking $A \implies I$, we use the framework (PLNN-v) from (Bunel et al. 2018) designed for verifying piecewise-linear neural networks. PLNN-v utilizes an encoding where the network and the property to be verified are encoded as a single network ($\bar{f}$), and an optimization problem is solved to determine if there exists an input to generate an output whose value is greater than 0. If there is none, the property is `unsat` over the input domain for original neural network f. $A \implies I$ is verified by checking that $A \wedge \neg I$ is unsatisfiable. Recall that for each r, $O_r^{f,S}$ is in DNF with linear atoms. For a layer $g_r(.)$, A has the form:

$$(s = g_r(\bar{p}_r)) \wedge \left(\bigvee_i T_i s \leq t_i\right),$$

and I has the form $\left(\bigvee_i Q_m \bar{p}_r \leq q_m\right)$. Then, $A \wedge \neg I$ can equivalently be written as:

$$(s = g_r(\bar{p}_r))) \wedge (-\max(\max_i(-\max_j(-Q_{i,j}\bar{p}_r + q_{i,j}), -\max_m(-\max_n(-T_{m,n}s + t_r)) \geq 0) \tag{7}$$

This can then be encoded into a neural network with $g_r(.)$ as the first layer followed by a sequence of MaxPool layers to encode the above constraint. For checking $I \implies \neg B$ we use the SMT-solver z3 (De Moura and Bjørner 2008).

Splitting Heuristic The algorithm for computing interpolants from (Albarghouthi and McMillan 2013) relies on sampling and separating sets of polytopes S_A and S_B satisfying formulas A and B respectively. The objective is to separate the sets of polytopes by sequentially applying a set of merging and splitting heuristics with hyperplanes generated using Farkas' lemma. If the two sets of polytopes cannot be separated by a single hyperplane, the sets are broken down into smaller subsets using the `unsat-core` returned by z3. However, generating the `unsat-core` is computationally expensive and further, we do not use $z3$ for checking $A \implies I$. Alternatively, we develop an intuitive heurstic where we first consider one polytope each satisfying $A_i \in S_A$ and $B_i \in S_B$. Then, we compute a separating hyperplane $\langle w, x\rangle + b = 0$ such that $\langle w, x\rangle + b < 0 \implies A_i$ and $\langle w, x\rangle + b > 0 \implies B_i$. Subsequently, we check if any of the other polytopes in our set are already separated by this hyperplane. If there exists a polytope p satisfying A that is not separated by the hyperplane, we combine p with the rest of the separated polytopes from A and compute a new hyperplane $\langle \hat{w}, x\rangle + b = 0$. If we cannot compute such a hyperplane, we split p from the rest. This is outlined in Algorithm 2, and the polytopes returned by Algorithm 2 are split from the initial set. In Algorithm 2, x is the set of shared free variables for formulas A and B.

Merging Heuristic In (Albarghouthi and McMillan 2013), the heuristic used for merging is to group together polytopes based on the syntactic similarity. However, for our problem all the sampled polytopes corresponding to the generated counter-examples during CEGAR are syntactically similar. Instead, we merge the polytopes into the set that has the closest matching pattern of activation states (e.g. constant or linear for ReLU nodes), in terms of Hamming distance. This results in data-points/counter-examples that have similar non-linear activation patterns being grouped together.

Algorithm 2 Splitting Heuristic

```
procedure SPLIT ROUTINE
Require: $S_A = \{A_1, \ldots, A_c\}$ (Polytopes satisfying $A$), $S_B = \{B_1, \ldots, B_d\}$ (Polytopes satisfying $B$)
    Set $A_{\text{count}} = 1$,$B_{\text{count}} = 1$
    Compute $(w, b)$:$\langle w, x\rangle + b = 0$ separates $A_1$, $B_1$
    $A_{\text{sat-set}} = \emptyset$, $B_{\text{sat-set}} = \emptyset$, Unsat-Set $= \emptyset$
    while $A_{\text{count}} \leq c \vee B_{\text{count}} \leq d$ do
        $A_{\text{old-count}} = A_{\text{count}}, B_{\text{old-count}} = B_{\text{count}}$
        for i = $A_{\text{old-count}}, \ldots, c$ do
            $A_{\text{count}} = A_{\text{count}} + 1$
            if $A_i \notin A_{\text{sat-set}}$ then
                if $\langle w, x\rangle + b < 0 \implies A$ then
                    $A_{sat-set} = A_{sat-set} \cup \{A_i\}$
                else
                    $\bar{S}_A = A_{sat-set} \cup \{A_i\}$
                    try    Find $(w, b)$ to sep. $\bar{S}_A$, $B_{\text{sat-set}}$
                    catch Unsat-Set = Unsat-Set$\cup \{A_i\}$
                             break
        for i = $B_{\text{old-count}}, \ldots, d$ do
            $B_{\text{count}} = B_{\text{count}} + 1$
            if $B_i \notin A_{\text{sat-set}}$ then
                if $\langle w, x\rangle + b > 0 \implies B$ then
                    $B_{sat-set} = B_{sat-set} \cup \{B_i\}$
                else
                    $\bar{S}_B = B_{sat-set} \cup \{B_i\}$
                    try    Find $(w, b)$ to sep. $\bar{S}_B$, $A_{\text{sat-set}}$
                    catch Unsat-Set = Unsat-Set$\cup \{B_i\}$
                             break
return Unsat-Set
```

5 Experiments

In this section, we implement and test our approach with neural networks trained on multiple tasks.

5.1 2D toy-example

We use the simple 2D-example introduced in Section 3.3 to study the scalability of the approach. On the same task, we train networks of varying sizes and measure the run-times for computing underapproximations of the third quadrant, as classified by the neural network. Figure 3 depicts the run-times for different network sizes.

Robustness We use the computed underapproximations to verify the robustness of the classifier. Robustness has been extensively studied for classifiers, particularly in image-processing (Szegedy et al. 2013; Carlini and Wagner 2017). For a given input x and the corresponding output-label k, we say the classifier f is ϵ robust if

$$\forall \bar{x} : \|\bar{x} - x\|_\infty \leq \epsilon, f(\bar{x}) = k.$$

To measure the robustness of the networks trained on this task, we compute both an underapproximation $U^{f,S}$ and an overapproximation $O^{f,S}$ of the pre-image corresponding to third quadrant for a network with 4-hidden layers and 16 nodes per hidden layer. For a set of 50 points from the third quadrant and $\varepsilon = 0.5$ (recall the problem domain is $[-1, 1]$ along each dimension), we check with `z3` for each point if there exists a counter-example satisfying:

$$\|\bar{x} - x\|_\infty \leq \epsilon, \bar{x} \notin U^{f,S}, \text{ and } \|\bar{x} - x\|_\infty \leq \epsilon, \bar{x} \notin O^{f,S}.$$

If a counter-example is found for both conditions, the point is not robust, and if a counter-example is found for the underapproximation but not the overapproximation, the point's robustness is unknown. If no counter-example is found for both conditions, the point is robust. We are able to validate/invalidate the robustness of 49 points and for one point, the result is unknown, and verifying these set of properties using the computed abstractions takes 1.4s. This shows that the approximations are quite accurate for this task. The computations were performed on a 2.40GHz Quadcore machine with 16 GB of RAM.

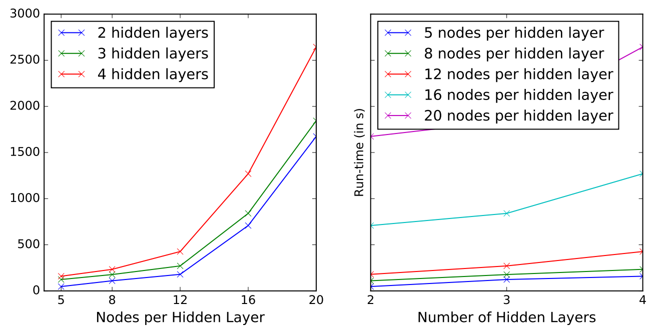

Figure 3: Runtimes for computing abstractions. Left: Varying number of nodes in every hidden layer. Right: Varying depth of the trained neural network. The run-time scales exponentially with increasing number of nodes, this is because the worst case complexity of checking the interpolation condition scales exponentially in the size of the hidden layer. The run-time scales almost linearly with increasing depth.

5.2 Cart-pole Control

We consider the classical control problem introduced in (Barto, Sutton, and Anderson 1983). The inputs to the network are observations from a four dimensional state space comprising of the position of the cart (x), the velocity of the cart($\dot{x}$), the angle of the pole (θ) and the angular velocity of the pole ($\dot{\theta}$). We train a neural network with 2-hidden layers for the problem with Deep-Q learning using the environment in (OpenAI-CartPole-v0 2018). The neural-network achieves a reward of perfect score of 200.0, averaged over 100 episodes. The output from the network maps to the discrete actions $\{\texttt{left}, \texttt{right}\}$. For both the output actions, we compute the overapproximations of the pre-image and computing each abstraction takes under 5 minutes. Note that since there are just two output classes, the negation of the overapproximation of one output action results in an underapproximation of the other output action. Both the overapproximations consists of a union of two half-spaces, which implies that the underapproximations are just one single polytope.

On replacing the neural network controller with a controller based on the overapproximation corresponding to

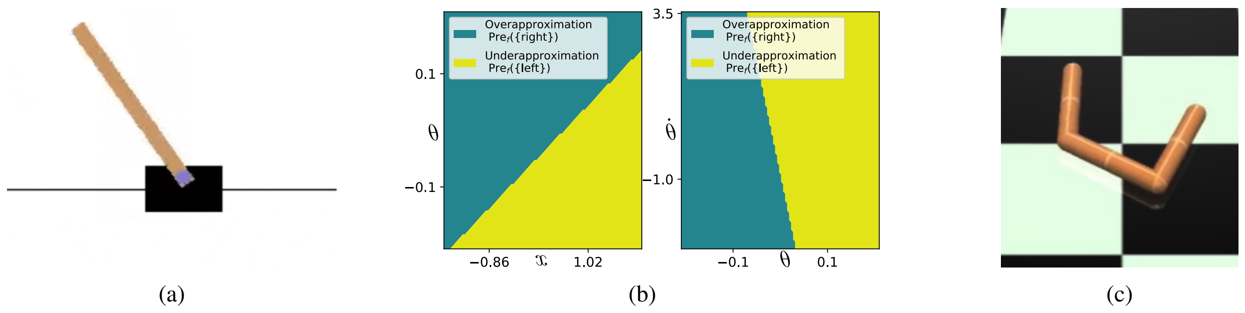

Figure 4: (a) Cart-pole: the neural-network controller is abstracted into simple control laws, (b) Computed abstractions for the neural-network cart-pole controller. Left:Varying x, θ with $(\dot{x}, \dot{\theta}) = 0$. Right: Varying $\theta, \dot{\theta}$ with $(x, \dot{x}) = 0$, (c) Swimmer-robot: the set of inputs corresponding to high-torque outputs are abstracted into a simple representation.

{`left`}, the new controller still achieves a perfect score of 200.0. This shows that the abstraction closely matches the exact pre-images for the neural-network. Further, these simple abstractions give insight into the internal strategy learned by the neural network. The computed overapproximation for the pre-image of the output set {`left`} (Pre_f(`left`)) is:

$$(-0.335x - 0.06\dot{x} + 0.918\theta + 0.202\dot{\theta} \leq -0.665)$$
$$\vee(-0.110x + 0.156\dot{x} + 0.950\theta + 0.245\dot{\theta} \leq -0.015).$$

We see that a negative x (the cart is to the left of the workspace), causes the cart to apply a force to the right. A negative θ causes the controller to make the cart move left. (See Figure 5). This matches the expected intuitive behavior.

Further, we can use the approximations to verify properties about the neural-network. For e.g., consider the property that $\forall x_t, \dot{x_t}, \theta_t, \dot{\theta}_t$ such that $\|x_t\| \leq 0.1, \|\dot{x_t}\| \leq 0.1, \|\theta_t\| \leq 0.1, \|\dot{\theta}_t\| \leq 0.1$, the condition $\|\dot{\theta}_{t+1}\| \leq 0.5$ holds. First, we can show that for all points that satisfy the overapproximation of Pre_f(`left`) and the action `left`, the property holds with the cart-pole dynamics. Next, we can repeat a similar procedure with Pre_f(`right`), to fully verify the neural network controller. We verify this property with `dReal`(Gao, Kong, and Clarke 2013), as it can reason over the sin and cos functions that occur in the dynamics. This computation takes 0.124 s. However, on checking for the condition (with the underapproximations) $\|\dot{\theta}_{t+1}\| \leq 0.3$, the solver finds a counterexample with $(x_t, \dot{x}_t, \theta_t, \dot{\theta}_t) = (-0.09, -0.09, 0.0, 0.097)$ as the initial condition with $\dot{\theta}_{t+1} = -0.31$ in 0.037 s.

5.3 Swimmer

For this task, we construct a compact abstraction that allows for run-time monitoring. The setting for the problem is to determine if a noisy observation by a monitor could possibly lead to unsafe behavior. The controller we consider is a neural network with 2 hidden layers trained with proximal policy optimization (Schulman et al. 2017) on the `Swimmer` environment (OpenAI-Swimmer-v2 2018). The task is to control a 3-link robot in a viscous fluid to make it swim forward as fast as possible. The network (f) maps from an 8-dimensional state space ($x \in \mathbb{R}^8$) to a 2-dimensional space (τ_1, τ_2) corresponding to the joint actuation torques.

For this task, suppose that in the the domain $x \in [-2, 2]^8$, the observations made by a run-time monitor are noisy such that $\|x - x_{\text{true}}\|_\infty \leq 0.1$, where x is the observed state and x_{true} is the true state. The controller has access to x_{true}, while the monitor only has the noisy reading x. We want to construct a monitor that during the operation of the robot flags an input x as unsafe if x is a noisy observation and it is possible that the true state x_{true} can cause a large torque. Formally, x is unsafe if it satisfies:

$$x \in [-2, 2]^8 \wedge \exists \bar{x}. \|\bar{x} - x\|_\infty \leq 0.1$$
$$\wedge f(\bar{x}) = (\tau_1, \tau_2) \wedge |\tau_1| + |\tau_2| \geq 1.0.$$

Since the monitoring is run-time, the flagging has to be near instantaneous and we would like to avoid reasoning over the entire network. To allow for this, we compute an overapproximating abstraction $\varphi(\bar{x})$ for the set of inputs to the network in the domain $[-2.1, 2.1]^8$ such that the network outputs $|\tau_1| + |\tau_2| \geq 1.0$. The monitor can be set up using $\varphi(\bar{x})$ as follows:

$$x \in [-2, 2]^8 \wedge \exists \bar{x}. \|\bar{x} - x\|_\infty \leq 0.1 \wedge \varphi(\bar{x}).$$

Algorithm 1 computes a $\varphi(\bar{x})$ that has a simple structure such that, for 50 inputs (sampled from observations seen during training) such that $x \in [-2, 2]^8$, the average time per input for checking the condition above with `z3` is 0.14 seconds. This time can further be reduced by parallelizing the check across polytopes.

6 Conclusion

We have developed an approach to algorithmically abstract neural network pre-images into compact representations that allow for interpretation and verification. The approach introduced here opens several possible directions for future contributions. An interesting direction to explore is if the current approach can be coupled with current verification algorithms for neural networks to improve verification itself. Another avenue to explore is if the abstraction procedure introduced in our work can be coupled with training to learn neural networks that satisfy certain desired properties. Alternatively, given an abstraction for a neural network, an interesting open question is if we can tune the abstraction to satisfy certain desired specifications without compromising significantly on performance.

References

Albarghouthi, A., and McMillan, K. L. 2013. Beautiful interpolants. In Sharygina, N., and Veith, H., eds., *Computer Aided Verification*, 313–329. Berlin, Heidelberg: Springer Berlin Heidelberg.

Ba, J., and Caruana, R. 2014. Do deep nets really need to be deep? In Ghahramani, Z.; Welling, M.; Cortes, C.; Lawrence, N. D.; and Weinberger, K. Q., eds., *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc. 2654–2662.

Barto, A. G.; Sutton, R. S.; and Anderson, C. W. 1983. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13(5):834–846.

Bastani, O.; Pu, Y.; and Solar-Lezama, A. 2018. Verifiable reinforcement learning via policy extraction. In *Neural Information Processing Systems, NIPS 2018*.

Bonacina, M. P., and Johansson, M. 2015. On interpolation in automated theorem proving. *Journal of Automated Reasoning* 54(1):69–97.

Bunel, R.; Turkaslan, I.; Torr, P. H. S.; Kohli, P.; and Kumar, M. P. 2018. In *Neural Information Processing Systems, NIPS 2018*.

Carlini, N., and Wagner, D. A. 2017. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, 39–57.

Clarke, E.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50(5):752–794.

Craig, W. 1957. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Sym. Logic* 3:269–285.

Dathathri, S.; Arechiga, N.; Gao, S.; and Murray, R. M. 2017. Learning-based abstractions for nonlinear constraint solving. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 592–599.

De Moura, L., and Bjørner, N. 2008. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 337–340. Berlin, Heidelberg: Springer-Verlag.

Ehlers, R. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, 269–286.

Gao, S.; Kong, S.; and Clarke, E. M. 2013. dReal: An SMT solver for nonlinear theories over the reals. In Bonacina, M. P., ed., *Automated Deduction – CADE-24*, 208–214. Berlin, Heidelberg: Springer Berlin Heidelberg.

Gehr, T.; Mirman, M.; Drachsler-Cohen, D.; Tsankov, P.; Chaudhuri, S.; and Vechev, M. 2018. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, 3–18.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. In *Communications of the ACM*.

Katz, G.; Barrett, C. W.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Kroening, D., and Weissenbacher, G. 2011. Interpolation-based software verification with wolverine. In Gopalakrishnan, G., and Qadeer, S., eds., *Computer Aided Verification*, 573–578. Berlin, Heidelberg: Springer Berlin Heidelberg.

Mahendran, A., and Vedaldi, A. 2015. Understanding deep image representations by inverting them. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, 5188–5196.

OpenAI-CartPole-v0. 2018. CartPole-v0. https://gym.openai.com/envs/CartPole-v0/. [Online; accessed 2-Sep-2018].

OpenAI-Swimmer-v2. 2018. Swimmer-v2. https://gym.openai.com/envs/Swimmer-v2/. [Online; accessed 2-Sep-2018].

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *CoRR* abs/1707.06347.

Szegedy, C.; Zaremba, W.; Sutskever, I.; Bruna, J.; Erhan, D.; Goodfellow, I. J.; and Fergus, R. 2013. Intriguing properties of neural networks. *CoRR* abs/1312.6199.

Verma, A.; Murali, V.; Singh, R.; Kohli, P.; and Chaudhuri, S. 2018. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning, ICML 2018*.

Wang, S.; Pei, K.; Whitehouse, J.; Yang, J.; and Jana, S. 2018. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems 32*.

Zakrzewski, R. R. 2001. Verification of a trained neural network accuracy. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, volume 3, 1657–1662 vol.3.