Increasing the Applicability of Verification Tools for Neural Networks

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

David Shriver

December 2022

Acknowledgments

Thank you to Matt Dwyer and Sebastian Elbaum, my advisors, for all of the advice and guidance you've provided over the years. From Nebraska to Virginia, you have helped me become a better researcher. I am extremely thankful to have been able to work with you both for so many years.

Thank you to my committee, Lu Feng, Tom Fletcher, Homa Alemzadeh, and Aaron Dutle, for taking time out of their busy schedules to meet, for reading through this disertation, and for the valuable feedback they have provided.

Thank you to my lab mates for your help and support in both research and life. Whether that be working through difficult research problems, getting lunch to break up a long day, or spending an evening playing board games, this would not have been possible without you all. Special thanks to Meriel, Carl, Will, Felipe, Trey, Soneya, Nusrat, Swaroopa, Mitch, Dong, Chris, and Rory, and all of the other members of the LESS Lab at UVA. Many of you have put the ideas and tools developed in this dissertation to the test and helped identify areas for improvement, and my work has become better because of it.

Thank you to my friends and family for all the support you've given me throughout graduate school and my life in general. Mom, Dad, and Grandma, thank you for the immense amount of love and support you've given throughout my life, and always taking an interest in my work. Thank you to Dan, Kim, and Kathleen for your support and encouragement, even after I moved across the country. Finally, thank you to my hiking partner and best friend, Meriel, for your constant support and inspiration. This work would not have been possible without you all.

Abstract

Machine learning is continually being applied to increasingly complex domains. In some cases, these involve safety-critical systems (e.g., self-driving cars) or influence important financial decisions (e.g., investment recommendations). Unfortunately, mistakes or unexpected behaviors in these domains can have high cost, including large monetary losses or death. As the use of machine learning continues to expand, it is important that we are prepared to ensure these systems behave as intended. Fortunately, many verification and falsification tools have been introduced in the past few years that enable us to prove properties about the behavior of neural networks. Unfortunately, these tools are often limited in the types of networks and properties that they support. Instead of focusing on the development of new verifiers or falsifiers, we instead develop techniques to increase the applicability of existing verification tools for neural network properties by building on the insight that unsupported verification problems can be rewritten into supported ones. We develop rewriting rules for property specifications, neural networks, and environment models, that enable existing off-the-shelf verification tools to be applied to previously unsupported verification problems. In general, we introduce rewritings which transform problems into sets of subproblems that are supported by a given verifier, and for which results can be mapped back to the original verification problem. We evaluate our rewriting rules across a wide selection of benchmarks and show that our techniques significantly increase the applicability of both verifiers and falsifiers.

Contents

List of Figures

List of Tables

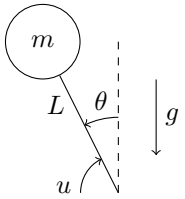
List of Listings

Chapter 1

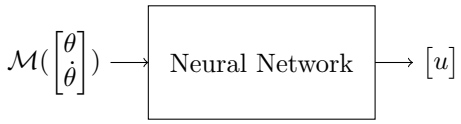
Introduction

Applications of machine learning are ubiquitous throughout our lives. From online shopping recommendations to voice assistants, machine learning models, such as neural networks, are continually applied to new domains. In some cases, these domains involve safety-critical systems (e.g., self-driving cars) or influence important financial decisions (e.g., investment recommendations). Unfortunately, mistakes or unexpected behaviors in these domains can have high cost, including large monetary losses or death. As the use of machine learning continues to expand, it is important that we are prepared to ensure these systems behave as intended. We must develop techniques now for ensuring the safety and correctness of machine learning systems.

Consider the simple pendulum control system shown in Figure 1.1. This system consists of an environment that models a pendulum with a mass on a length of rod and a neural network that controls the torque applied to the pendulum. The goal is to keep the pendulum in a vertical position, with the mass directly above the pivot point. The environment, depicted in Figure 1.1a, models the current state of the environment and how the state changes over time. For example, the environment could track the angle and velocity of the pendulum and define how these values update with an input torque value. The neural network, depicted in Figure 1.1b, takes in a set of inputs derived from the state of the environment and outputs a torque. To ensure the correctness of this system, we must specify properties describing the expected behavior, such as those in Figure 1.1c. These behavioral specifications can range from simple properties over only the output of the network, to properties describing relationships between network inputs and outputs, and to properties describing how the state changes over time. Given an environment model, network, and property, we can check whether the property is true for the network operating within the specified environment model, providing



(a) An environment consisting of a pendulum with a controlled torque, T.



(b) An neural network to control torque, to maintain a pendulum angle of 0 degrees.

$\phi_{3.1}$: *The neural network should never output a value less than* -1 *or greater than* 1.

$\phi_{3.2}$: *If the pendulum in the bottom left quadrant, then the predicted torque should be negative.*

$\phi_{3.3}$: *If the pendulum is in the upper left quadrant and moving clockwise, then it should never fall below the pivot point.*

(c) Three example properties for this system.

Figure 1.1: A system with a network to control the angle of a pendulum.

assurance that the network behaves as expected.

In the past few years, an abundance of techniques have been introduced for verifying properties of neural networks [2, 11, 14, 16–18, 20, 32–34, 36, 40, 50, 55, 59, 102, 104, 111, 112, 114–117, 125, 129, 133, 135, 136, 141–143, 151]. In general, these techniques take in a neural network and a logical formula (i.e., a property) specifying a relationship between the inputs and outputs of that network, and seek to prove that the property is true for all inputs and outputs of the network. Unfortunately these existing verification tools for neural networks tend to make several restrictive assumptions about the network, property, and environment that significantly reduce the applicability of these techniques to real-world problems.

First, many verifiers have limited support for neural network operations and architectures, often due to the significant engineering cost. For example, the publicly available Neurify [135] tool only supports convolution, matrix multiplication, vector addition, and ReLU operations, and adding support for new operations to the tool requires defining a new input format and an encoding of the new operation as constraints that are consumable by the tool. Commonly used networks, such as VGG [113] and ResNet [48], do not satisfy this requirement, due to their use of unsupported operations such as max pooling, average pooling, and batch normalization. Additionally, many verifiers, such as nnenum [10] and JuliaReach [14] are restricted to network architectures in which operations are applied sequentially to the inputs to get the final output. However, modern neural networks such as ResNet and DenseNet [54], include operations which rely not only on the output of a single previous operation, but the output of several previous operations, prohibiting the application of verifiers that expect a sequential architecture.

Second, many verifiers have no mechanism for specifying explicit environment models. Many tools, such as Planet [34], Marabou [69], and VeriNet [49] are only applicable to correctness problems in which the environment model can be ignored. Generally, these are problems where behavioral properties are specified over the inputs and outputs of a network at a single instance of time. We refer to such tools as open-loop verifiers because there is no feedback loop from the network output to the environment. This contrasts with closed-loop verifers, which require an explicit environment model to support verification of properties that take into account feedback from actions controlled by the network. Due to their lack of environment support, open-loop verifiers are limited to verification of properties over the inputs and outputs of a network, such as properties $\phi_{3.1}$ and $\phi_{3.2}$ in Figure 1.1c. However, they cannot verify properties like $\phi_{3.3}$, which specifies how the environment can change over time in response to the network.

Finally, many tools have limited support for complex property specifications, often focusing on a specific property type due to the cost of designing and implementing an expressive property specification language. For example, adversarial attacks such as projected gradient descent are extremely effective at quickly finding violations to a type of property referred to as local robustness. However, off-the-shelf, adversarial attacks are limited to only this single property type, reducing their applicability and usefulness. Additionally, almost all verification tools only support properties over a single neural network. This means that verifiers such as NNV [129] and Neurify [135] cannot check property specifications over multiple networks, such as verifying whether two networks are equivalent or whether a system with multiple network controllers, e.g., a steering controller and a throttle controller for an autonomous vehicle, satisfies some behavioral property.

While the efficiency of verification techniques continues to improve, their application is still limited by the networks, environments, and properties that they support. In order to verify complex behaviors of real-world neural networks we introduce a rewriting approach for verification problems which transforms networks, environment models, and property specifications from forms unsupported by a given verifier to supported forms. For example, network rewriting, or refactoring, can enable verifiers to support VGG by rewriting max pooling, average pooling, and batch normalization to equivalent supported forms using only convolution and relu operations. Similarly, property rewriting, or reduction, can enable verifiers to support more complex property types by embedding some of the property semantics into the network and defining new properties over the modified network. We can, for example, rewrite properties into a conjunction of local robustness properties, enabling adversarial attacks such as projected gradient descent to be applied, as well as many other verification tools. The environment rewritings we introduce can aid other rewritings, transforming

complex environment models into forms that are easier for other rewritings to work with. For example, one rewriting introduced in Chapter 7 rewrites functions in an environment model to be neural networks, enabling subsequent rewritings of the problem.

While our rewritings enable verification of more problems than ever before, they wouldn't be very useful if they didn't provide insights about the original problem. A rewriting that simply removed any unsupported operation in a neural network may allow a verifier to run, but any results would not be applicable to the original network. In designing rewritings, we take care to preserve the semantics of the original problem whenever we can. In many cases we can rewrite problems such that any result to the rewritten problem also applies to the original problem. In some cases, however, rewritings only preserve one type of result. For example, for some rewritings if we can prove the property is true for the rewritten problem, then we know the original property must also be true for that problem, but showing the rewritten property is not true doesn't necessarily mean the original property is not true. Occassionally it can even be useful to define rewritings that do not guarantee that the semantics of the original property are preserved, such as when environment models cannot be explicitly defined, as is the case in Chapter 6.

This work makes several novel contributions towards increasing the applicability of verification tools for neural networks.

- First, we define a theoretical framework for verification of neural networks, which enables the definition of network, environment, and property rewritings.
- Second, under this framework, we introduce instantiations of these rewritings to increase the applicability of existing verification tools. We have developed instantiations for network [108, 110], environment [127], and property [108, 109] rewritings for open-loop verification, as well as network, environment, and property rewritings for closed-loop verification.
- Third, we have empirically evaluated our rewritings across multiple benchmarks to demonstrate the effectiveness of rewriting at increasing the applicability of verification tools.
- Finally, we implement these instantiations and make them available in publicly accessible tools: DNNV[1], DNNF[2], and R4V. This immediately increases the applicability of tools already supported by DNNV and DNNF and enables other verifiers to make use of these rewritings to increase their applicability and usefulness going forward.

[1] `https://github.com/dlshriver/dnnv`
[2] `https://github.com/dlshriver/dnnf`

The remainder of this dissertation is organized as follows. Chapter 2 presents the necessary background information on neural networks, verification, and relevant software engineering concepts required in this thesis. In Chapter 3 we introduce rewriting, providing an overview and definitions of terms that make up the theoretical foundations of our approach. The following chapters then introduce instantiations of rewritings to increase the applicability of verifiers. First, Chapter 4 introduces instantiations of semantics-preserving property reduction. Second, Chapter 5 introduces an instantiation of both non-semantics-preserving and semantics-preserving network refactoring. Third, Chapter 6 introduces an instantiation of environment environment rewriting. Finally, Chapter 7 introduces instantiations of network, environment, and property rewritings for closed-loop verification problems. After presenting our instantiations of rewriting, we conclude and discuss possible future research directions in Chapter 8.

Chapter 2

Background

This chapter provides background on neural networks and verification that will be useful throughout this work.

2.1 Neural Networks

A neural network $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$ is a machine learned model trained to approximate some partial target function $f : \mathcal{X} \to \mathcal{Y}$. For example, f may classify some n-dimensional input (e.g., an image) as one of m possible classes (e.g., a digit in the range 0 to 9). f is partial in the sense that it is trained to generalize to a target data distribution, $\mathcal{X} \subset \mathbb{R}^n$. For inputs outside this distribution, its behavior should be considered undefined.

Training involves applying $\mathcal{N}$ to samples, $(\boldsymbol{x}, \boldsymbol{y})$, from a training set, $X \subset \mathcal{X}$, and repeatedly updating the parameters of $\mathcal{N}$ based on the value of some loss function $error(\mathcal{N}(\boldsymbol{x}), \boldsymbol{y})$. Training is performed over a training set rather than the entire data distribution, since this distribution is generally unmanageably large (e.g., the set of all possible road images from a vehicle-mounted, forward-facing camera). The training set is assumed to be a representative sample of the expected input distribution.

The literature on machine learning has developed a broad range of rich operation types and explored the benefits of different combinations of operations in realizing accurate approximations of different target functions [43]. For example, convolution (Conv) and pooling (e.g., MaxPool, AveragePool) operations were designed for processing inputs known to have a grid-like structure, such as images. Other operations, such as batch normalization (BatchNorm) attempt to eliminate known issues to improve accuracy or speed up training [57]. And some operations, such as ReLU,

Sigmoid, Tanh, and LeakyReLU, provide the non-linearity that neural networks require to learn.

One particularly important class of operations for this work are *continuous piecewise-linear* operations, which are functions where the domain of definition can be partitioned into convex polytopes each of which corresponds to a linear function. Many common non-linear network operations, such as absolute value (Abs), ReLU, LeakyReLU, Min, and MaxPool are continuous piecewise-linear. Prior work has shown that continuous piecewise-linear functions can be represented by the linear combination of other continuous piecewise-linear functions, such as Abs and Max [65, 138, 139]. We make use of this relationship between piecewise-linear functions when defining some semantics preserving network refactorings in Chapter 5.

Additionally, one class of neural network architecture is a *sequential* model, in which operations are applied sequentially to a given input vector to get an output vector. Sequential architectures are often described as having *layers*, which generally consist of several successive operations, such as an affine transformation followed by a non-linear element-wise operation (e.g., ReLU, Sigmoid, Tanh). Two common layers are fully-connected layers and convolutional layers. A fully-connected layer consists of a matrix multiplication and vector addition, followed by a non-linear activation function such as a ReLU operation. A convolutional layer similarly consists of affine operations and activation function, but replaces the matrix multiplication with a 2-dimensional convolution. While many verification tools assume that neural networks have a sequential structure, many non-sequential architectures have been introduced in the machine learning literature, such as ResNet [48], DenseNet [54], and Inception [121, 122]. This means that many widely used networks are not currently supported by existing tools.

2.2 Verification of Neural Networks

Given a neural network, $\mathcal{N}$, an environment model, $\mathcal{E}$, and property specification, ϕ, over the environment and inputs and outputs of the network, verification attempts to prove or falsify whether ϕ is true for the given $\mathcal{N}$ and $\mathcal{E}$. Formally, a verifier checks whether $\mathcal{E}, \mathcal{N} \models \phi$ is satisfiable. Here we will present an overview of existing property types prior to this work, as well as a brief overview of existing verification approaches.

2.2.1 Property Specificatons

A survey on verification of neural networks [80] identifies several representations for input and output constraints in property specifications. Two of these are particularly useful for certain instantiations of

rewritings introduced in this work. A *hyperrectangle* is an n-dimensional rectangle where constraints are formulated as $(x_i \geq lb_i) \wedge (x_i \leq ub_i)$, where $lb_i, ub_i \in \mathbb{R}$ and $0 \leq i < n$ define the lower and upper bounds on the value of each dimension of x, respectively. A special case of hyperrectangles is the *unit hypercube*, which is a hyperrectangle where $\forall i.(lb_i = 0) \wedge (ub_i = 1)$; an n-dimensional hypercube is denoted $[0, 1]^n$. A *halfspace-polytope* is a polytope that is represented as a set of linear inequality constraints, $Ax \leq b$, where $A \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$, k is the number of constraints and n is the size of the network input.

Using such encodings, researchers have specified a range of properties for neural networks. We classify these properties into several categories.

Open- vs Closed-Loop

First, we differentiate between open-loop and closed-loop properties.

Open-loop properties specify a relationship between network inputs and outputs at a single instance of time. For example, $\phi_{3.1}$ and $\phi_{3.2}$ in Figure 1.1 are open-loop specifications. $\phi_{3.1}$ states that all inputs should avoid two unsafe regions in the output space, while $\phi_{3.2}$ states that a subset of the input space map to a known safe region of the output space. In general, open-loop properties specify the sets of acceptable or unacceptable outputs for a given set of inputs.

Closed-loop properties specify how the environment state and network inputs and outputs change over time. For example, $\phi_{3.3}$ is a closed-loop property specification, which states that, after a given set of environment states is reached (i.e., those when the pendulum is in the upper left quadrant and moving clockwise), then another set of states should always be avoided (i.e., those states where the end of the pendulum is below the pivot). In general, closed-loop properties specify either a set of environment states that must be reached or that must be avoided as the network operates.

Open-Loop Property Taxonomy

Second, we distinguish three broad categories for open-loop property specfications: robustness, reachability, and differential properties. We provide this additional categorization due to the abundance of open-loop properties in the verification literature.

Robustness properties originated with the study of adversarial examples [123, 149]. They apply to classification models and specify that inputs from a specific region of the input space produce the same output class. Robustness properties can be further classified as either local or global; the former asserts robustness in a partial region of the input domain and the latter over the entire input domain. Detecting violations of robustness properties has been widely studied, and they are

a common type of property for evaluating verifiers [34, 115, 116, 125, 135].

Reachability properties define the post-condition using constraints over output values rather than output classes, and are thus not limited to classification models. Such properties have been used to evaluate verifiers [68, 135]. Reachability properties specify that inputs from a given region of the input space must produce outputs that lie in a given region of the output space. For example, a neural network model controlling the velocity of an autonomous vehicle may have a safety property specifying that the model never produces a desired velocity value greater than the maximum physical speed of the vehicle for any input in the input domain. Similarly to robustness, reachability properties can be further classified as local or global. For example, a global halfspace-polytope reachability (GHPR) property would specify a halfspace-polytope constraint on network output values that must hold for all inputs.

Differential properties are the most recently introduced neural network property type [96]. These properties specify a difference (or lack thereof) between outputs of multiple neural networks. One type of differential property is equivalence, which states that for every input, two neural network models produce the same output. Such a property can be used to check that neural network semantics are preserved after some modification, such as quantization or pruning. Differential properties can be supported by combining multiple neural networks into a single network and expressing properties over their combined input and output domains.

In addition to the three categories above, open-loop properties can also be classified by the form of their input pre-condition. *Global* properties have the most permissive pre-condition, enforcing the post-condition for any input in the input domain of the neural network. For example, a neural network that operates on images may accept values in $[0, 1]^n$. The pre-condition of a global property would not restrict this domain any further. *Local* properties only enforce the post-condition for inputs within a designated region of the input domain. For example, a local property for an image processing network may have the precondition that inputs are within distance ε of some given input x. This is especially common in robustness properties.

Closed-Loop Taxonomy

Prior work has distinguished 2 categories of closed-loop property specifications: must-reach and must-not-reach [107]. *Must-reach* properties specify that all paths beginning from a set of initial states eventually reach some state in a set of acceptable states. For example, a property for a robotic vacuum may specify that, starting from a set of initial conditions, the robot will eventually reach the charging base before its battery dies. *Must-not-reach* properties specify that no path beginning

from the set of initial states ever reaches an unsafe state. Equivalently, these specifications dictate that paths must always remain in a set of safe states. For example, a property for an adaptive cruise control system may specify that the vehicle being controlled always maintain a distance of 10 meters from the lead car. In other words, the system must never reach a state in which the follower is less than 10 meters from the lead vehicle.

2.2.2 Logic

Property specifications could be specified in many different forms. We will discuss 2 formal systems useful for this work: first-order logic and linear temporal logic.

Linear Temporal Logic

Linear Temporal Logic (LTL) is a fragment of first order logic designed to encode logical formulae over sequences of states. Formulae are constructed from a finite set of propositional variables, $\rho \in P$, logical operators, $\vee$ and $\neg$, and temporal operators $\mathbf{X}$ and $\mathbf{U}$, where $X\varphi_1$ means that φ_1 must be true in the next time step, and $\varphi_1 \,\mathbf{U}\, \varphi_2$ means that φ_1 must remain true until φ_2 becomes true. The LTL syntax is as follows:

$$\varphi := \rho \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\,\varphi_1 \mid \varphi_1 \,\mathbf{U}\, \varphi_2 \mid \mathbf{G}\,\varphi_1 \mid \mathbf{F}\,\varphi_1$$

The temporal operators G and F are also often defined, where $\mathbf{G}\,\varphi_1$ means that φ_1 must always remain true and $\mathbf{F}\,\varphi_1$ means that φ_1 must eventually become true. The formula $\mathbf{G}\,\varphi_1$ can be written using the standard variables and operators as $\neg((\rho \vee \neg\rho)\,\mathbf{U}\,\neg\varphi_1)$, while the formula $\mathbf{F}\,\varphi_1$ can be written as $(\rho \vee \neg\rho)\,\mathbf{U}\,\varphi_1$.

Given an infinite sequence, w, of truth assignments to the propositional variables in P, we can determine whether φ is satisfied, i.e., whether $w \models \varphi$, as follows:

$$\begin{aligned}
w &\models \rho \text{ if } \rho \in w_0 \\
w &\models \neg\varphi \text{ if } w \not\models \varphi \\
w &\models \varphi_1 \vee \varphi_2 \text{ if } w \models \varphi_1 \text{ or } w \models \varphi_2 \\
w &\models \mathbf{X}\,\varphi_1 \text{ if } w_{1..\infty} \models \varphi_1 \\
w &\models \varphi_1 \,\mathbf{U}\, \varphi_2 \text{ if } w \models \varphi_2 \text{ or } w \models \varphi_1 \text{ and } w_{1..\infty} \models \varphi_1 \,\mathbf{U}\, \varphi_2 \\
&\qquad \text{i.e., } \exists i \geq 0.(w_{i..\infty} \models \varphi_2) \wedge (\forall 0 \leq k < i.w_{k..\infty} \models \varphi_1)
\end{aligned}$$

where w_i is the i-th assignment in the sequence, and $w_{i..k}$ is the subsequence starting at index i and ending at index $k-1$. A sequence, w, *satisfies* an LTL formula φ if $w \models \varphi$, and φ is *satisfiable* if there exists some w such that $w \models \varphi$. A formula, φ, is *valid* if all w satisfy φ.

LTL Variant - LTL_τ For the work in Chapter 7 we use of variant of LTL which specifies properties over finite sequences with a maximum length of τ. The syntax of this variant, LTL_τ is the same as LTL, however the rules for satisfiability replace instances of ∞ with the length of the sequence, $|w|$, and the temporal operator $\mathbf{X}$ cannot be satisfied at the last time step, τ. Formally:

$$
\begin{aligned}
w &\models \rho \text{ if } \rho \in w_0 \\
w &\models \neg\varphi \text{ if } w \not\models \varphi \\
w &\models \varphi_1 \vee \varphi_2 \text{ if } w \models \varphi_1 \text{ or } w \models \varphi_2 \\
w &\models \mathbf{X}\,\varphi_1 \text{ if } |w| > 1 \text{ and } w_{1..|w|} \models \varphi_1 \\
w &\models \varphi_1\,\mathbf{U}\,\varphi_2 \text{ if } w \models \varphi_2 \text{ or } |w| > 1 \text{ and } w \models \varphi_1 \text{ and } w_{1..|w|} \models \varphi_1\,\mathbf{U}\,\varphi_2
\end{aligned}
$$

2.2.3 Verification and Falsification

Many methods have been developed to prove or falsify behavioral properties of neural networks. We consider two categories of approaches in this work: verification and falsification.

Verification

Verification approaches attempt to show that a specification is true for a network. They can also often determine whether a specification is false. A recent survey [80] describes several categories of algorithmic approaches for verifiying open-loop properties of neural networks. These include reachability methods, optimization methods, and search-based methods. Many existing tools can use a hybrid approach and use methods from multiple categories.

Reachability methods calculate approximations of the sets of values that can be computed as output of the network given an input constraint. Verification tools that use reachability methods include: MaxSens [143], DeepGo [104], CROWN [151], CNN-CERT [16], ERAN [114–117], VeriNet [49, 50], Debona [18], NNV [129], and nnenum [8]. *Optimization* methods formulate property violations as a threshold for an objective function and use optimization algorithms to determine failure to meet that threshold. Verification tools that use optimization methods include: MIPVerify [125], ILP [11], Duality [33], ConvDual [142], Certify [102], and PeregriNN [71]. *Search-based* methods

that explore regions of the input space where they then formulate reachability or optimization sub-problems. Verification tools that use search-based methods in conjunction with reachability methods include: Neurify [135], ReluVal [136], Fast-lin and Fast-lip [141], DLV [55], α-CROWN [147], and β-CROWN [137]. Verification tools that use search-based methods in conjunction with optimization-based methods include: Sherlock [32], Bab [20], Planet [34], Reluplex [68], and Marabou [69].

Closed-loop verification tools, particularly for continuous-time systems, almost exclusively use reachability-based algorithms to analyze the behavior of neural network control systems. These tools generally use Flow* [25] or CORA [4] to overapproximate the highly non-linear continuous dynamics of the system. Closed-loop verification tools that use reachability-based methods include: Sherlock [32], NNV [129], ReachNN* [36], JuliaReach [14, 107], and Verisig [59]. The closed-loop verifiers designed for discrete-time problems have slightly more algorithmic variety, but primarily focus on reachability and optimization. Reachability-based tools include SafeRL_Infinity [7], while optimization based methods include tools such as OVERT [112] and VenMAS [2].

Falsification

Falsifiers are a subset of verification approaches that seek to find examples that violate a given specification for a given model, rather than prove the specification. Two categories of falsification techniques that have been developed for neural are adversarial attacks and fuzzing.

Adversarial attacks are methods optimized to detect violations of robustness properties [1, 149]. They usually fall into the optimization category defined above. In general, adversarial attacks the take in a neural network and an initial input, and attempt to produce a small perturbation that, when applied to the input changes the class predicted by the given model. These perturbations are often also subject to additional constraints, such as remaining within some specified distance of the original input. A perturbed input, commonly known as an adversarial example is a violation to a local robustness property. Adversarial attacks can be classified based on characteristics of the attack, such as if they are white-box [45, 77, 85, 87, 123] or black-box [119]; targeted [45, 123] or untargeted [77, 87]; iterative [77, 85, 87] or one-shot [45, 123]; or by their perturbation constraint (e.g., L_0 [119], L_2 [23], or L_∞ [45, 123]). A more exhaustive taxonomy and description of existing adversarial attacks is available in the literature [1, 149].

Fuzzing involves randomly generating inputs within a given input region (often the full input space), and checking whether the outputs they produce violate a specified post-condition. Fuzzing is more general than adversarial attacks, in that it can support the falsification of more than robustness properties, but requires specifying input mutation functions and objective functions (i.e., an output

oracle), for every property of interest. Existing fuzzing techniques include TensorFuzz [90] and DeepHunter [145].

Chapter 3

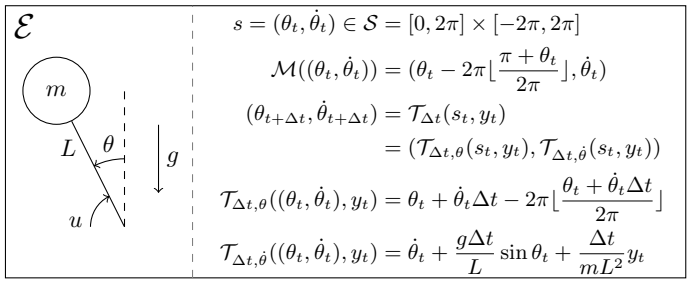
A Framework for Rewriting Neural Network Correctness Problems

We begin by defining the theoretical framework upon which the rest of our work builds. The goal of this framework is to establish a foundation for discussing and defining correctness problems, verification, problem rewritings, and properties of rewritings. We first provide a high level overview of rewriting, before formally defining the terms and concepts that we will build on in the succeeding chapters.

3.1 Overview

To ensure that a neural network behaves as expected, verification can be performed to check whether a given behavioral specification holds for the network operating in some specified environment. In general, verification takes in a neural network, behavioral specification, and model of the environment and returns whether, under the provided environment model, the behavioral specification is true or false for the given network. Behavioral specifications define a desired relationship between inputs and outputs of the neural network, while the environment model specifies how new input values are obtained from previous output values.

Take, for example, the problem originally depicted in Figure 1.1, and reproduced in Figure 3.1 using the formalisms defined later in this chapter. The problem consists of a single pendulum controlled by a neural network, with the goal of maintaining the pendulum in an upright position. In this example, the state of the environment can be completely described using a real-valued vector



(a) A model for a pendulum. The environment can be represented by the angle and angular velocity of the pendulum.

(b) A neural network to control the torque applied to the pendulum.

$$\forall x.(x \in \mathcal{X}) \implies (-1 \leq \mathcal{N}(x) \leq 1) \tag{$\phi_{3.1}$}$$

$$\forall x.(x \in \mathcal{X} \wedge \frac{\pi}{2} \leq x_0 \leq \pi) \implies (\mathcal{N}(x) \leq 0) \tag{$\phi_{3.2}$}$$

$$((0 \leq s_0 \leq \frac{\pi}{2}) \wedge (s_1 < 0)) \implies XG((s_0 < \frac{\pi}{2}) \vee (\frac{3\pi}{2} < s_0)) \tag{$\phi_{3.3}$}$$

(c) Three formal property specifications for the single pendulum control system.

Figure 3.1: Example definitions of the environment model, neural network, and properties for the pendulum system introduced in Chapter 1.

of 2 values, representing the angle and angular velocity of the pendulum. The environment also defines how future states are derived from the current state and network outputs. The pendulum is controlled by a neural network, which takes in an input vector derived from the state and outputs a control value representing the desired torque to apply to the pendulum. For now, the only assumption we will make about this network is that the final operation is a clip operation, which bounds the output values to some range by setting any values lower than a specified lower bound to be equal to that lower bound, and any values greater than a specified upper bound to be equal to that upper bound, i.e., $clip(x, L, U) = min(max(x, L), U)$. Finally, Figure 3.1c formally specifies 3 desired properties of the system behavior. The first two are open-loop properties, specifying how the network should respond to specific inputs, and the third is a closed-loop property, specifying how the environment is expected to change over time in response to the network.

Unfortunately, many of the constructs in this simple problem are not supported by various verifiers. For instance, open-loop verifiers do not support environment models, which are required to verify closed-loop properties, such as $\phi_{3.3}$. While closed-loop verifiers support some environment models, existing tools can struggle to return useful results due to significant overapproximation and incomplete algorithmic techniques, as we show in Chapter 7. Fortunately, in many cases, we can rewrite the problem such that the environment semantics are incorporated into the property specification and network, enabling open-loop verifiers to support problems with non-trivial environment

models. In addition to a lack of environment support, the property specifications themselves often have limited support among falsifiers and verifiers. The open-loop properties in this example are reachability properties, which cannot be directly checked by some techniques, such as adversarial attacks, which require robustness properties. Fortunately, we can encode the property semantics as operations in the network, and produce new robustness properties over the modified network, which when verified provide results for the original problem. We can similarly rewrite the closed-loop property, which enables verification using open-loop verifiers, as well as closed-loop verifers. While closed-loop verifiers may be able to parse closed-loop property specifications, they, in addition to the open-loop tools, often also struggle with supporting the necessary neural network operations. For example, in the problem described above, the clip operation in the neural network is unsupported by most verification tools. However, this function is piecewise-linear, which means it can be reformulated using a linear combination of other piecewise-linear functions such as matrix multiplication, vector addition, and ReLU operations, which are supported by most existing verifiers. For example, the clip operation can be rewritten as:

$$Clip(x, L, U) = \begin{bmatrix} I_n & -I_n \end{bmatrix} ReLU(\begin{bmatrix} I_n \\ I_n \end{bmatrix} x + \begin{bmatrix} -L_n \\ -U_n \end{bmatrix}) + L_n$$

where I_n is the identity matrix of size n, L_n and U_n are n-dimensional vectors containing the values L and U, respectively, and n is the length of the vector x. While existing verification tools may not support this example problem as originally stated, rewriting the problem can enable tools to provide useful results by verifying transformed versions of the problem.

3.2 A Framework for Rewriting Correctness Problems

We now formally define the components of a correctness problem, verification, and rewriting. We will use these definitions to formally specify the network, environment, and properties for our example inverted pendulum control problem, as shown in Figure 3.1.

3.2.1 Correctness Problems

Let $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$ be a *neural network* which approximates a partial function $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X} \subseteq \mathbb{R}^n$ and $\mathcal{Y} \subseteq \mathbb{R}^m$. $\mathcal{N}$ takes in an n-dimensional vector and produces an m-dimensional vector. We focus here on the single input and single output case without any loss of generality. A multi-input or -output network can be rewritten as an equivalent single-input or -output network through

concatenation, reshaping, and indexing of inputs and outputs. A neural network can be represented as an *operation graph*, which is a type of directed, acyclic graph in which nodes represent operations over data, and edges represent the flow of data between operations. An operation graph, $\mathcal{N} = \langle Q, q_0, q_*, \delta \rangle$, contains a set of operations, Q, a specified input operation, $q_0 \in Q$, an output operation, $q_* \in Q$, and a set of named, directed edges, $\delta \subset Q \times Q \times \Sigma$, where Σ is the set of possible edge names. We depict a simplified operation graph representation of the pendulum controller network in Figure 3.1b. In an operation graph, operations can represent arbitrary functions. The set of operations includes an *input* operation, q_0, and an *output* operation, q_*. An input operation accepts arbitrary user-provided data and an output operation produces the final result of the neural network computation. In figures, we will mark the input with a dashed outline and the output with a double outline. An edge exists between two operations if the output of the first operation is used by the second. Each edge is tagged with an identfier in order to correctly disambiguate inputs for operations with multiple incoming edges. The output of an operation graph can be computed by sorting the operation graphs in topological order and then executing operations from the inputs, to the outputs, passing data along their corresponding edges.

Let $\mathcal{E} = \langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle$ be an *environment model*, such as the model we have defined for the pendulum example in Figure 3.1a. The environment model includes a state space, $\mathcal{S}$, which is the set of all possible environment states. An environment state is some representation of the environment or context in which the neural network is operating. It can include, for example, the positions or velocities of objects in the environment, or any other information relevant to the function of the neural network or property to be checked. The environment model also requires a function, $\mathcal{M} : \mathcal{S} \to \mathbb{R}^n$, that maps from states to network inputs, as well as a function, $\mathcal{T} : \mathcal{S} \times \mathbb{R}^m \to \mathcal{S}$, that maps from a state and network output to a new state. $\mathcal{M}$ enables $\mathcal{S}$ to be different than the network input space. This decoupling provides more freedom to the modeller to choose a state space that is most convenient to their problem. For instance, while the neural network in this example takes in the angle of the pendulum in the range $[-\pi, \pi]$, it may be more convenient to model the system such that the angle is in the range $[0, 2\pi]$. $\mathcal{T}$ is necessary to enable checking behavior of networks over time. This function specifies how the environment changes in response to the network output. A common environment model (particularly for open-loop problems) is the trivial environment model, $\mathcal{E}^{(\top)} = \langle \mathbb{R}^n, x \mapsto x, (s, y) \mapsto s \rangle$, in which there is no distinction between states and network inputs, and the transition function maps states to themselves regardless of the network output. If no environment model has been defined, then we assume the trivial environment model is to be used.

Let ϕ be a neural network behavioral property. ϕ is a logical formula over environment states, network inputs, and network outputs. This formula specifies some desirable property between network inputs, outputs, and environment states. For example, we define 3 property specifications in Figure 3.1c. The first 2 are in first-order logic, and are only over network inputs and outputs. The third property is specified in linear temporal logic, and describes a relationship between values in the environment state as the neural network operates. Specifically, it states that if the angle of the pendulum is between 0 and $\frac{\pi}{2}$ and the angular velocity is negative (i.e., the pendulum is in the upper left quadrant and moving clockwise), then the angle of the pendulum should be less than $\frac{\pi}{2}$ or greater than $\frac{3\pi}{2}$ in all future states (i.e., it should remain above the pivot).

Finally, let $\psi = [\mathcal{E}, \mathcal{N}_1, \ldots, \mathcal{N}_k, \phi]$ be a *correctness problem*. A correctness problem consists of the environment, networks, and property to be verified. For example, the environment, network, and any of the 3 properties in Figure 3.1 compose a correctness problem. We may also refer to correctness problems as verification problems in this work. We denote the set of all possible correctness problems Ψ.

3.2.2 Verification and Falsification

Verification attempts to prove that a given behavioral property, ϕ, is valid under a given interpretation of the networks and environment models used in the property. If it is, then the verifier will output *holds*, otherwise it will output *violated*. Formally:

$$\mathcal{V}(\psi) = \mathcal{V}([\mathcal{E}, \mathcal{N}_1, \ldots, \mathcal{N}_k, \phi]) = \begin{cases} \textit{holds} & \text{if } \mathcal{E}, \mathcal{N}_1, \ldots, \mathcal{N}_k \models \phi \\ \textit{violated} & \text{if } \mathcal{E}, \mathcal{N}_1, \ldots, \mathcal{N}_k \models \neg\phi \end{cases} \tag{3.1}$$

While verifiers for traditional software systems tend to produce a witness or certificate of program correctness on a successful proof, this is not the case for existing verifiers for neural network properties. Production of witnesses for these types of proofs is still an emerging area of research [58]. Most existing verifiers only support properties over a single network, i.e., $\mathcal{V}(\mathcal{E}, \mathcal{N}, \phi)$, and open-loop verifiers generally implicitly use the trivial environment model, i.e., $\mathcal{V}(\mathcal{N}, \phi) = \mathcal{V}(\mathcal{E}^{(\top)}, \mathcal{N}, \phi)$. Additionally, some verifiers use incomplete decision procedures, which can return a value of *unknown* if the verifier cannot determine the validity of $\mathcal{E}, \mathcal{N} \models \phi$.

One subset of verification techniques is that of *falsification*, which, instead of attempting to prove validity of $\mathcal{E}, \mathcal{N} \models \phi$, attempts to prove satisfiability of $\mathcal{E}, \mathcal{N} \models \neg\phi$. Existing falsification techniques usually produce a witness of property invalidity in the form of a counter-example, i.e., a network

input that results in falsification of the property. While falsification methods can often be faster than full verification methods for invalid properties, they cannot prove validity of the property, only that it is false.

3.2.3 Rewriting

Verifiers and falsifiers often have restrictions on the types of neural networks, environment models, and properties that they support. For example, many verifiers do not support batch normalization, leaky ReLU, or max pooling operations, which are not uncommon in modern neural networks. Additionally, many tools only support verification of a single property type. Adversarial attacks, for instance, only support falsification of robustness properties. Finally, verifiers tend not to support explicit specification of environment models, unless they have been specifically designed for verification of closed-loop systems. In order to increase the applicability of existing tools to a broader range of property, unsupported networks, environments, and properties can be rewritten into sets of sub-problems that are supported by a given verifier.

We define a general framework for defining rewritings of correctness problems. A rewriting, $rewrite : \Psi \rightarrow P(\Psi)$, takes in a correctness problem and produces a set of correctness problems such that the new problems increase the applicability of a given verifier. We refer to each of the problems produced by rewriting as a sub-problem. In most cases, rewritings should also maintain some relationship between the original problem and sub-problems such that verifying the sub-problems provides insight into the original problem. We discuss two classifications of rewritings in this thesis, holds-preserving and violation-preserving.

If a rewrite is *holds-preserving*, then the original problem is guaranteed to be valid if all sub-problems are valid. Formally, we say that a rewrite, f, is holds-preserving if $(\bigwedge_{\psi' \in f(\psi)} \psi') \implies \psi$. If a rewrite is *violation-preserving*, then the original problem is guaranteed to have a violation if any sub-problem has a violation. Formally, we say that a rewrite, f, is violation-preserving if $(\bigvee_{\psi' \in f(\psi)} \psi') \implies \psi$. For a sequence of rewritings to be holds-preserving or violation-preserving, all rewritings in the sequence must also be holds-preserving or violation-preserving, respectively. If a rewriting is both holds-preserving and violation-preserving, we may also refer to it as semantics-preserving, since it exactly preserves the semantics of the original problem. The set of sub-problems produced by a semantics-preserving rewriting is said to be *equivalid* with the original problem.

We now define 3 general classes of rewritings for the instantiations introduced in this thesis; one for networks, one for environments, and one for property specifications. These rewritings take in a correctness problem and rewrite it into a new set of correctness problems, such that either the

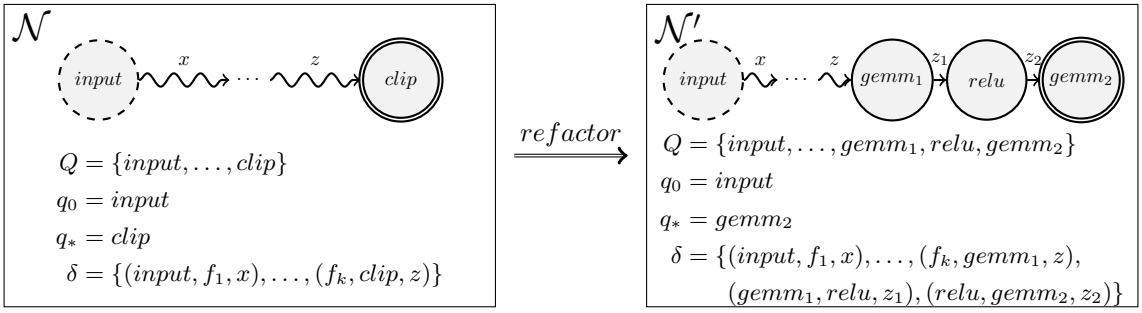


Figure 3.2: An example of network refactoring. The unsupported *clip* operation is replaced by an equivalent sequence of *gemm* and *relu* operations.

network, environment, or property becomes more verifiable in all sub-problems. Usually, the goal of rewriting is to increase verifiability of a given part of the correctness problem, but rewriting can also be performed in order to make a correctness problem meet the assumptions of another rewriting, as is the case for the environment rewritings that we introduce in Chapter 7. While correctness problems can be defined over multiple networks, we define these 3 classes of rewritings over the single network case. However, we also define a network rewriting which takes in any correctness problem and rewrites it to use a single network:

$$rewrite([\mathcal{E}, \mathcal{N}_1, \ldots, \mathcal{N}_k, \phi]) = \{[\mathcal{E}, \mathcal{N}', \phi']\} \tag{3.2}$$

An instantiation of this rewriting is fairly straightforward. Operation graphs can be combined by taking the union of their operation and edge sets, adding a new input operation that is the concatenation of the inputs of all operation graphs and replacing the existing input operations with slicing operations that selects the corresponding portion of the new input, and setting the output to be a new operation that is the concatenation of the outputs of all operation graphs. This rewriting is semantics-preserving, and we will assume throughout this work that it has been applied as necessary.

Network rewritings take in a correctness problem and transform the networks in some way. We call such rewritings, refactorings. Formally, we define:

$$refactor([\mathcal{E}, \mathcal{N}, \phi]) = \{[\mathcal{E}, \mathcal{N}', \phi]\} \tag{3.3}$$

This type of rewriting can be particularly useful for converting unsupported neural network operations into equivalent sets of supported operations. For example, we can rewrite the clip operation in the operation graph defined in Figure 3.1b using matrix multiplication, vector addition, and ReLU

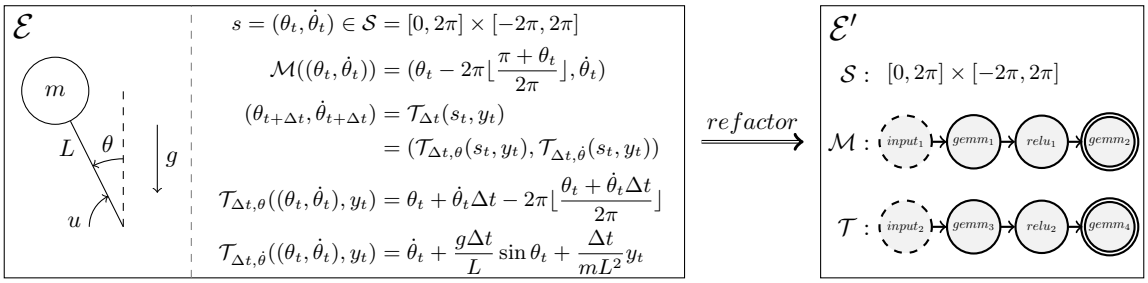


Figure 3.3: An example of environment rewriting. The state-to-input and transition functions are rewritten as neural networks to support further rewritings.

operations to get the rewritten operation graph shown in Figure 3.2. While we generally recommend defining network refactorings to be semantics-preserving, as we do in Chapter 5.1.1, non-semantics-preserving refactoring can also be useful as an early step to produce a new neural network model as a complete replacement for the original, as we do in Chapter 5.1.2.

Environment rewritings take in a correctness problem and transform the environment, networks, and property such that the resulting environment is supported by the desired verifier. Formally, we define:

$$remodel([\mathcal{E}, \mathcal{N}, \phi]) = \{[\mathcal{E}', \mathcal{N}', \phi']\} \tag{3.4}$$

The problem is rewritten to a single subproblem with a modified environment model, network, and property. One such rewriting can be used to rewrite an environment to use a neural network as the state-to-input and transition functions, as shown in Figure 3.3. In such a case, the network and property are generally not modified. This can be extremely useful for defining other rewritings, since it enables composition of the environment functions and the network being verified in future rewritings. Similar to refactorings, there are both semantics-preserving and non-semantics preserving environment rewritings. We introduce instantiations of environment rewritings in Chapters 6 and 7.

Property rewritings take in a correctness problem and transform the environment, networks, and property such that the resulting property specification is supported by the desired verifier. We call such rewritings, reductions. Formally, we define:

$$reduce([\mathcal{E}, \mathcal{N}, \phi]) = \{[\mathcal{E}'_1, \mathcal{N}'_1, \phi'], \ldots, [\mathcal{E}'_k, \mathcal{N}'_k, \phi']\} \tag{3.5}$$

One application of reduction is to enable the verification of non-robustness properties by verification tools that only support robustness. For example, we can rewrite the reachability specification $\phi_{3.2}$ in Figure 3.1c to a robustness specification by encoding the reachability constraints as operations

in the neural network which classify whether the property is satisfiable or not. This enables us to rewrite the property to specify that inputs in the desired region always map to the class representing satisfiability, a robustness property. These rewritings can also be used to encode the environment semantics as part of the neural network and property specification, resulting in correctness problems with the trivial environment model.

Chapter 4

Reducing Neural Network Properties

Many verification tools are designed to check a specific property of neural networks. For example, adversarial attacks, a common falsification method, are designed to find counter-examples to robustness properties, which state that the output class of a network is invariant to small perturbations of a given input. Likewise, the *MIPVerify* and *VeriNet* verifiers were designed to prove the absence of violations to robustness properties. Unfortunately, this means that many interesting properties can not be checked by existing off-the-shelf tools, as non-robustness property specifications are not explicitly supported.

However, we have identified the key insight that properties for neural networks can be reduced to more commonly supported forms. We build on this insight to develop an approach for reducing correctness problems with properties in an expressive, general form to a set of problems, equivalid with the original, that use local robustness properties, which have wide support among both falsifiers and verifiers. Such a rewriting has the potential to bring the existing techniques to bear on a much larger set of neural network properties.

For example, consider the network and property $\phi_{3.2}$ from the pendulum example in Figure 3.1, which we reproduce on the left side of Figure 4.1. While the network contains a *clip* operation rarely supported by verifiers, it is supported by common falsification methods, such as adversarial attacks. Unfortunately the property is a reachability property rather than a robustness property, stating that a given region of the input space reaches a specified region of the output space. Reduction can allow falsifiers to find a result for this property, by rewriting the problem such that the semantics of the

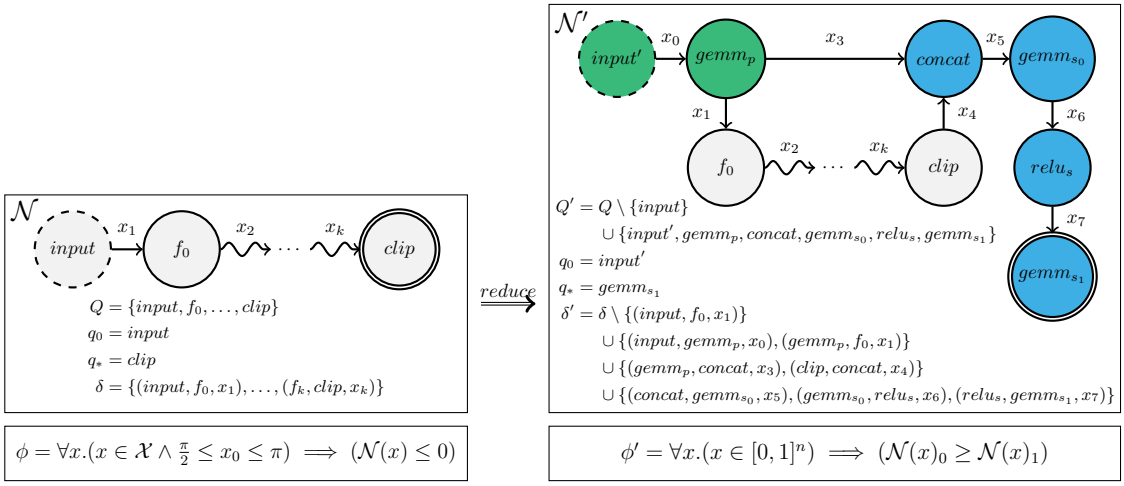


Figure 4.1: A correctness problem with a reachability property for the pendulum control example is reduced to an equivalid problem with a robustness property.

original property are encoded as operations in the network, transforming the network into a classifier of the satisfiability of the original property and replacing the original property with one that asserts robustness of the new network. Results for this rewritten problem can be directly mapped to a result for the original problem.

Property reduction has been exploited by the software engineering community for verification and program analysis for decades, such as the reduction from stateful safety properties to reachability properties. For example, the applicability of partial order reductions was broadened by reducing stateful properties to a form of deadlock [42], and both data flow analyses [91, 118] and SAT solving [13] can be applied to verify stateful properties by formulating the reachability of error states. Another use of reductions is to enable the application of more efficient algorithmic methods. For instance, the `-safety` option of the SPIN model checker enables it to use a significantly faster reachability algorithm [52]. Reduction is now considered standard in verification and program analysis, however, the lessons of such reductions have not yet taken root for the new domain of neural network falsification and verification.

In this chapter, we introduce an approach for reducing a neural network correctness problem into an equivalid set of correctness problems formulated with properties that can be processed by existing verification techniques. In particular, we will focus on reducing to local robustness properties, which have broad support among both verifiers and falsification approaches such as adversarial attacks. The approach is fully automated which allows developers to specify properties in a convenient form while leveraging the complementary strengths of falsification and verification algorithms.

The primary contributions of this work are:

1. An automated approach for reducing neural network correctness problems into a set of robustness problems which is equivalid with the original problem.
2. Two implementations, DNNF and DNNV, of our reduction approach that employ portfolios of falsifiers and verifiers.
3. A study demonstrating that property reduction to enable application of falsifiers yields cost-effective violations of general neural network correctness problems[1].

4.1 Approach

The primary goal of our approach is to amplify the power of verifiers and falsifiers by increasing their applicability. Our approach takes in a correctness problem comprised of a neural network and a property (we assume a trivial environment model), and encodes it as an set of correctness problems with robustness properties, which is equivalid with the original problem. This reduction enables us to run a portfolio of methods applicable to this restricted property type to obtain results for the original property.

4.1.1 Defining Property Reduction

As defined in Section 3.2, a *correctness problem*, $\psi = [\mathcal{E}, \mathcal{N}_1, \ldots, \mathcal{N}_k, \phi]$, consists of the environment, networks, and property to be verified, and the goal of verification is to determine whether the proeprty ϕ is satisfiable under the given interpretation of the networks and environment models, i.e., whether or not $\mathcal{E}, \mathcal{N}_1, \ldots, \mathcal{N}_k \models \phi$. In this work we assume that problems use trivial environment models, as defined in Section 3.2, and we will often omit them for brevity. Additionally, we will initially assume that properties are defined for a single network, and discuss how to apply the rewriting to properties over multiple networks in Section 4.1.4.

Reduction, $reduce : \Psi \to P(\Psi)$, aims to transform a correctness problem, $[\mathcal{N}, \phi] = \psi \in \Psi$, to an equivalid form, $reduce(\psi) = \{[\mathcal{N}_1, \phi_1], \ldots, [\mathcal{N}_k, \phi_k]\}$, in which property specifications define robustness properties:

$$\phi_i = \forall x. x \in [0,1]^n \to \mathcal{N}_i(x)_0 > \mathcal{N}_i(x)_1$$

and networks have input domains defined as unit hypercubes, and output domains consist of two values – indicating property satisfaction and falsification, i.e., $\mathcal{N}_i : [0,1]^n \to \mathbb{R}^2$.

[1] An analysis of DNNV is performed in the study in Chapter 5 after introducing network rewritings.

As we demonstrate in Section 4.2, reduction enables the application of a broad array of efficient neural network analyses to compute problem satisfiability and/or unsatisfiability.

As defined, reduction has two key properties. The first property is that the set of resulting problems is equivalid with the original correctness problem (a proof of this theorem is included in Appendix A).

Theorem 1. *The reduction, reduce* : $\Psi \rightarrow P(\Psi)$*, maps a correctness problem with properties constraining inputs and outputs to polytopes in the input and output space to an equivalid set of correctness problems.*

$$\mathcal{N} \models \psi \Leftrightarrow \forall(\mathcal{N}_i, \phi_i) \in reduce(\psi).\ \mathcal{N}_i \models \phi_i$$

The second property is that the resulting set of problems all use the same property type. In this case, we reduce to robustness, asserting that all inputs are classified as the same class, which we will refer to as *class 0*. Applying reduction enables verifiers or falsifiers to support a large set of correctness problems by implementing support for this single property type. We reduce to robustness properties here due to their broad support among existing falsifiers and verifiers, however the reduction described here can be trivially modified to a reachability property as described in Section 4.1.5.

4.1.2 Overview

To illustrate reduction, consider property $\phi_{3.2}$ for the pendulum control example, which we reproduce in Figure 4.1. Tools like MIPVerify or adversarial attacks cannot be used off-the-shelf to check this property, since it is not a robustness property.

To enable the application of these tools, we reduce the property to a set of correctness problems with robustness properties, such as the one shown in Figure 4.1. This particular example is reduced to a single correctness problem but, in general, multiple sub-problems can be produced by reduction. Each of the problems pair a robustness property (shown in the bottom right of Figure 4.1) with a modified version of the original network. The new neural network is created through two key transformations. First, we incorporate a prefix network (shown in green in Figure 4.1) to reduce the input domain to a unit-hypercube, which is necessary to enable the application of tools that require input constraints to be represented as hyperrectangles. This modification also facilitates property specification by ensuring that the properties for reduced problems can all use the same pre-condition. Second, we incorporate a suffix network (shown in blue in Figure 4.1) that takes in the inputs and outputs of the original network and classifies whether they constitute a violation of

Algorithm 1: Property Reduction

Input: Correctness problem $\langle \mathcal{N}, \phi \rangle$
Output: A set of robustenss problems $\{\langle \mathcal{N}_1, \phi_1 \rangle, ..., \langle \mathcal{N}_i, \phi_i \rangle\}$

```
1  begin
2      φ' ← DNF(¬φ)
3      Ψ ← ∅
4      for disjunct ∈ φ' do
5          hpoly ← disjunct_to_hpolytope(disjunct)
6          prefix ← construct_prefix(hpoly)
7          N' ← N' : x ↦ concat(N(x), x)
8          suffix ← construct_suffix(hpoly)
9          N'' ← suffix ∘ N' ∘ prefix
10         φ' ← ∀x.(x ∈ [0,1]^n ⟹ N''(x)_0 > N''(x)_1)
11         Ψ ← Ψ ∪ ⟨N'', φ'⟩
12     return Ψ
```

the original property. This suffix transforms the network into a classifier for which violations of a robustness property correspond to violations of the original property.

4.1.3 Reduction

We rely on two assumptions to define this property reduction. First, the constraints must be represented as a union of convex polytopes over the inputs and outputs of a network. And second, each convex polytope must be represented in halfspace-polytope form, i.e., as a conjunction of linear inequalities. The first of these assumptions enables the encoding of constraints as network operations, while the second assumption simplifies the definition of the algorithm. Properties specified using an alternative polytope representation can first be converted to the required form before applying the reduction. Complying with these assumptions still enables properties to retain a high degree of expressiveness as unions of polytopes are extremely general and subsume other geometric representations, such as intervals and zonotopes. Section 4.2.1 shows that these assumptions are sufficient to support existing neural network correctness problems.

Algorithm 1 defines the reduction at a high level. We present each step of the algorithm and describe their application to the inverted pendulum control example described above.

Reformat the Property

Reduction first negates the original property specification and converts it to disjunctive normal form (DNF) – line 2. Negating the specification means that a satisfying model falsifies the original property. The DNF representation allows us to construct a property for each disjunct, such that if

Algorithm 2: disjunct_to_hpolytope

Input: Conjunction of linear inequalities ϕ_i
Output: Halfspace polytope H

```
1  begin
2      H ← (A, b) where A is an (|φ_i|) × (m + n) matrix where columns 0 to m − 1 correspond to output
         variables N(x)_0 to N(x)_{m−1} and columns m to m + n − 1 correspond to input variables x_0 to x_{n−1}
3      for ineq_j ∈ φ_i do
4          if ineq_j uses ≥ then
5              swap lhs and rhs; switch inequality to ≤
6          else if ineq_j uses > then
7              swap lhs and rhs; switch inequality to <
8          move variables to lhs; move constants to rhs
9          if ineq_j uses < then
10             decrement rhs; switch inequality to ≤
11         A_j ← coefficients of variables on lhs
12         b_j ← rhs constant
13     return H
```

any are violated, the negated specification is satisfied and thus the original specification is falsified. For each of these disjuncts the approach defines a new robustness problem, as described below.

Transform into halfspace-polytopes

Constraints in each disjunct of the disjunctive normal form are then converted to halfspace-polytope constraints, defined over the concatenation of the input and output domains – *disjunct_to_hpolytope*() on line 5. This conversion is described in Algorithm 2, and simplifies the definition of future steps. A halfspace-polytope can be represented in the form $Ax \leq b$, where A is a matrix of k rows, where each row represents 1 constraint, and d columns, one for each variable. In this case, d is equal to $m + n$, the size of the output space, plus the size of the input space. This representation facilitates the transformation of constraints into network operations in subsequent steps. To build the matrix A and vector b, we first transform all inequalities in the conjunction to $\leq$ inequalities with variables on the left-hand-side and constants on the right-hand-side. The transformation first converts $\geq$ to $\leq$ and $>$ to $<$ – lines 4-7 of Algorithm 2. Then, all variables are moved to the left-hand-side and all constants to the right-hand-side – line 8. Next, $<$ constraints are converted to $\leq$ constraints by decrementing the constant value on the right-hand-side – lines 9-10. This transformation assumes that there exists a representable number with greatest possible value that is less than the right-hand-side, which holds for most numeric representations used in computation[2]. Finally, each inequality is converted to a row of A and value in b – lines 11-12. In the pendulum control example, DNF results in the single disjunct $(x \in \mathcal{X} \wedge x_0 \in [\frac{\pi}{2}, \pi] \wedge \mathcal{N}(x) \leq 0)$, which *disjunct_to_hpolytope*() transforms to

[2]We briefly discuss the validity of this assumption in Appendix A.

Algorithm 3: construct_prefix

```
   Input: Halfspace polytope H
   Output: A fully-connected layer P
1  begin
2      lb = [−∞, ..., −∞]
3      ub = [+∞, ..., +∞]
4      for constraint ∈ H do
5          if constraint is over only input variables then
6              for x_i ∈ x do
7                  lb_i ← max {min_{x_i} constraint, lb_i}
8                  ub_i ← min {max_{x_i} constraint, ub_i}
9      W ← diag(ub − lb)
10     b ← lb
11     GeMM_p ← (x ↦ Wx + b)
12     return GeMM_p
```

the halfspace polytope $\begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^\intercal \begin{bmatrix} x \\ \mathcal{N}(x) \end{bmatrix} \leq \begin{bmatrix} -\frac{\pi}{2} & \pi & 2\pi & 2\pi & 0 \end{bmatrix}^\intercal$.

Prefix Construction

Using the constructed halfspace-polytope, Algorithm 1 next constructs a prefix to the original network to ensure the input domain of the resulting network is $[0, 1]^n$, where n is the input dimensionality of the original network – *construct_prefix*() on line 6. The algorithm to construct the prefix is shown in Algorithm 3. The prefix is constructed by first extracting lower and upper bounds for every input variable – lines 2-8. This extracts the minimal axis-aligned bounding hyperrectangle. The lower and upper bounds can then be used to construct the prefix network, which is a single n-dimensional fully-connected layer, with no activation function, added as a GeMM operation (i.e., $\text{GeMM}(x) = Wx + b$), which has a diagonal weight matrix with values equal to the ranges of the input variables, and biases equal to the lower bounds of each input. For the pendulum control example, the diagonal of the weight matrix for the added GeMM operation is $\begin{bmatrix} \frac{\pi}{2} & 4\pi \end{bmatrix}$ and the bias vector is $\begin{bmatrix} \frac{\pi}{2} & -2\pi \end{bmatrix}^\intercal$. The prefix operates on unit hypercubes, reducing the input space to the correctness problems. This is necessary to enable the application of tools which require input constraints to be represented as hyperrectangles, which is common among the adversarial attack methods targeted by this work. It also simplifies the later property specification, as all modified networks operate over the same input domain, a unit hypercube. The prefix also encodes any interval constraints over the original input space, allowing them to be removed before suffix construction, which can simplify the suffix networks.

Algorithm 4: construct_suffix

Input: Halfspace polytope $H = (A, b)$
Output: A DNN with 2 fully connected layers S

1 **begin**
2 $\quad \text{GeMM}_{s_0} \leftarrow (x \mapsto Ax + -b)$
3 $\quad W \leftarrow \begin{bmatrix} 1 & 1 & ... & 1 \\ 0 & 0 & ... & 0 \end{bmatrix}$
4 $\quad \text{GeMM}_{s_1} \leftarrow (x \mapsto Wx + \vec{0})$
5 $\quad$ **return** $\text{GeMM}_{s_1} \circ ReLU \circ \text{GeMM}_{s_0}$

Forward Inputs

Next, the output of the prefix, i.e., the inputs to the original network, are forwarded to the end of the network and concatenated with the original output layer – line 7 of Algorithm 1. Because constraints will be encoded as a network suffix that classifies whether inputs are property violations, this step is necessary to enable the encoding of any constraints over the inputs.

Suffix Construction

Finally, we append a suffix to the end of the network that classifies whether inputs and their corresponding outputs satisfy the specification – *construct_suffix*() on line 8. The algorithm for constructing the suffix from the halfspace-polytope constraints is shown in Algorithm 4. The constructed suffix consists of three operations, a GeMM, followed by a ReLU, followed by a final GeMM operation. The output of the first GeMM is a vector of size k, the number of constraints in the halfspace-polytope defined by the disjunct. The final GeMM outputs a vector of size 2, one value for each of the 2 possible classes.

The first GeMM operation of the suffix has a weight matrix equal to the constraint matrix, A, of the halfspace-polytope representation, and a bias equal to $-b$ – line 2. With this construction, each value in the vector output by this operation will have a value greater than 0 if and only if the corresponding constraint is not satisfied, otherwise it will have a value less than or equal to 0, which becomes equal to 0 after the ReLU operation is applied. For the pendulum control example, we only have a single disjunct, with the halfspace-polytope defined above.

The output operation of the suffix produces a vector of 2 values. The first of these values is the sum of all values in the previous layer, and has a bias of 0. Because the values in the input to this operation each represent a constraint, and each of these values is 0 only when the constraint is satisfied, if the sum of all of these values is 0, then the conjunction of the constraints is satisfied, indicating that a violation has been found. The second of these output values is the constant 0 – all incoming weights and bias are 0. The resulting network will predict class 1 if the input satisfies the

corresponding disjunct and class 0 otherwise.

Correctness Problem Construction

Lines 9-11 of Algorithm 1 define the reduced subproblem comprised of the network that we have constructed and a robustness property. The robustness property specification is always the same and states that the network should classify all inputs in the d-dimensional hypercube as class 0 – no violations. If a violation is found to this property, then, according to Theorem 1, the original property is violated by the unreduced input that violated the robustness property. In the end, we have generated a set of correctness problems such that, if any of the problems is violated, then the original problem is also violated. This comes from our construction of a property for each disjunct in the DNF of the negation of the original property.

4.1.4 Properties Over Multiple Networks

While Algorithm 1 is defined over properties with a single network, it can easily be applied to properties over multiple networks, by combining those networks into a single large network. This is specially relevant to check for equivalence properties. This can be done by concatenating their input and output vectors, as described in Section 3.2.3. This results in a single large network with a computation path for each network. The transformation algorithm can then be applied as before.

4.1.5 Implementation

We have implemented instances of this reduction in 2 tools, one for falsifiers, which tend to have greater operation and architecture support for networks, and one for verifiers.

DNNF

The tool for falsifiers, which we call DNNF[3], implements the full reduction as described in Section 4.1. The tool is implemented in Python, and takes in a property specification specified in DNNP and a corresponding neural network in the ONNX format, and returns whether a violation is found. Due to the lack of a standard format for specifying DNN properties, we develop a Python-embedded DSL for DNN properties, which we call DNNP. DNNP is designed to be an expressive language and is independent of the network. We describe DNNP in more detail in Appendix B.

[3] `https://github.com/dlshriver/dnnf`

DNNV

The tool for verifiers, which we call DNNV[4], only partially implements this reduction, omitting the input forwarding step, since the non-sequential structure introduced by this step is rarely supported by verification tools. Because of this difference, the suffix cannot be used to check general polytope constraints over the inputs, restricting the space of supported properties to ones with input constraints that can be represented as a union of hyperrectangles. However, some verifiers allow specification of halfspace polytope constraints over the inputs, which the DNNV reduction can support by replacing the unit hypercube input constraint of the generated sub-property with the halfspace polytope constraint of the corresponding disjunct. This restricts DNNV to properties where all constraints apply to either inputs or outputs, i.e., the input and output spaces are disjoint. Both inputs and outputs can be constrained, but not to each other, e.g., $x > 0 \implies \mathcal{N}(x) > 0$ is allowed, but $\mathcal{N}(x) > x$ is prohibited. Additionally, the DNNV reduction allows reducing to reachability constraints, by dropping the constant output, and simply asserting that the single output value must always be greater than 0. The tool is implemented in Python, and takes in a property specification specified in DNNP and a corresponding neural network in the ONNX format, and returns whether a violation is found.

4.2 Empirical Evaluation

We now assess the cost-effectiveness of reducing neural network properties by applying it to a range of neural network property benchmarks that provide diversity in terms of property types and network complexity. This study only looks at DNNF. DNNV is evaluated in Chapter 5 after the introduction of network rewriting. Our evaluation will attempt to answer the following research questions:

- RQ1: How expressive are the properties supported by property reduction?
- RQ2: How cost-effective is DNNF at finding property violations?
- RQ3: How scalable is DNNF?

4.2.1 RQ1: On the Expressiveness of Reduction

We first evaluate whether the assumptions about the property specification required by reduction, namely that the original property is specified as a logical formula of linear inequalities, is expressive enough to support neural network correctness properties that have been proposed in existing work.

[4] `https://github.com/dlshriver/dnnv`

Setup

To evaluate the expressiveness of properties supported by our reduction, we analyze and catalog the benchmarks used by the five verifiers used in our later study, as well as the benchmarks of a recent neural network verifier competition, VNN-Comp 2020 [81]. Additionally, we surveyed published papers on neural network verification in 2019 and 2020 (the two years prior to the study), identifying 4 additional works [10, 35, 128, 129]. Finally, we include the 2 new benchmarks introduced in this work.

Results

We summarize the results in Table 4.1, which lists the benchmarks used in each work, the type and number of properties in the benchmark and whether the properties are supported by Algorithm 1. The property types use abbreviated names with the following encoding: the first symbol indicates whether the property is global (G) or local (L); the second symbol indicates whether the input constraint can be represented as a hyper-rectangle (□) or not (⊠); the third symbol indicates whether the property is a robustness (r) property, a reachability (R) property, or a differential (D) property. Each cell under a property type indicates the number of properties in the corresponding benchmark of that type. The bolded benchmarks are used later in the study for the evaluation of RQ2 and RQ3. We describe the details of these benchmarks in more detail below.

The first benchmark is ACAS Xu, introduced for the study of the Reluplex verifier [68], and used extensively since [10, 17, 20, 69, 81, 135, 136]. The benchmark consists of 10 properties. Properties ϕ_1, ϕ_2, ϕ_3, ϕ_4, ϕ_7 and ϕ_8 are reachability properties, while ϕ_5, ϕ_6, ϕ_9, and ϕ_{10} are traditional class robustness properties. All 10 properties have hyper-rectangles constraints over the inputs and are fully supported by our property reduction.

The next benchmark is from the evaluation of the Planet verifier. First is the Collision Avoidance benchmark, which consists of 500 safety properties that check the robustness of a network that classifies whether 2 simulated vehicles will collide, given their current state. All 500 properties are L□r properties, and are all fully supported. Second is a set of 7 properties on an MNIST network. The first 4 of these are G□R properties, while the next 2 are L□r properties, and the final property is an L⊠r property. In addition to restricting the amount of noise that can be added to each pixel in the input image, the final property constrains the difference in the noise between neighboring pixels. All 7 properties are fully supported by our property reduction.

The Neurify verifier was evaluated on the ACAS Xu benchmark and on properties of 4 MNIST networks, 3 android app malware classification networks, and 1 self-driving car network. The eval-

Table 4.1: Property types of existing benchmarks and their support by reduction. The property type names use the following encoding: the first symbol indicates a global (G) or local (L) property; the second symbol indicates whether the input constraint can be represented as a hyper-rectangle (□) or not (⊠); the third symbol indicates the property class as robustness (r), reachability (R), or differential (D). Bolded benchmarks are used later in the study to evaluate RQ2 and RQ3.

Benchmark	**# of Property of Each Type**								**Support**
	L□r	**L⊠r**	**G□R**	**L□R**	**L⊠R**	**G□D**	**L□D**	**Other**	
ACAS Xu [68]	4			6					10
Collision Avoidance [34]	500								500
Planet-MNIST [34]	2	1	4						7
Neurify-MNIST [135]	500								500
Neurify-Drebin [135]	500								500
Neurify-DAVE [135]				50	150				200
ERAN-MNIST [115]	1700								1700
ERAN-CIFAR [115]	1600								1600
ReluDiff ACAS [96]							14		14
ReluDiff-MNIST [96]							200		200
ReluDiff-HAR [96]							100		100
VNN-COMP-CNN [81]	300								300
VNN-COMP-PWL [81]	54			6					60
VNN-COMP-NLN [81]	32								32
ImageStars-MNIST [128]	900								900
ImageStars-ImageNet [128]		6							6
NNV-ACC [129]								2	0
GHPR			20						20
CIFAR-EQ						91	200		291

uation on MNIST used 500 L□r properties across 4 networks, all of which we support. Neurify was also evaluated on 3 networks trained on the Drebin dataset [6] to classify apps as benign or malicious. This benchmark also includes 500 L□r properties, which are fully supported. Finally, Neurify was evaluated on local reachability properties for a modified version of the DAVE self-driving car network [15]. This benchmark consists of 200 local reachability properties, with 4 different types of input constraints (50 properties of each type). The first type of input constraint is an L_∞ constraint, which is equivalent to a hyper-rectangle constraint. The second type of input constraint is an L_1 constraint, which can be written as a halfspace polytope constraint. The third and fourth type of input constraint is an image brightness and contrast, which can also be written as a halfspace polytope constraints. All 200 properties are fully supported by our property reduction.

The DeepZono abstract domain of the ERAN verifier used in our study [115], was evaluated on 3300 L□r properties applied to a set of 24 MNIST networks and 13 CIFAR10 networks. All of the properties in this benchmark are fully supported by our approach.

The ReluDiff verifier was designed to support differential properties in order to show equivalance between two networks [96]. The verifier was evaluated on L□D properties. Each property was defined over a network, $\mathcal{N}$ and a modified version of the same network with quantized weights, $\mathcal{N}'$. The property checked whether $|\mathcal{N}(x) - \mathcal{N}'(x)| < \epsilon$ held in a local region of the input space. 14 of these properties were verified over networks from the ACAS Xu benchmark [68], 200 properties on networks trained with the MNIST dataset, and 100 properties on a network trained for Human Activity Recognition [5]. All 314 differencing properties are fully supported by our approach.

The 2020 VNN-Comp competition used 3 benchmarks. The first is a benchmark with properties applied to networks with piecewise linear activation functions. This benchmark consists of the ACAS Xu benchmark [68] with 4 L□r properties and 6 L□R properties, as well as a set of 50 local robustness properties with hyper-rectangle input constraints applied to 3 MNIST networks. All of these properties are supported by our approach. The second is a set of 300 local robustness properties with hyper-rectangle input constraints applied to convolutional neural networks trained on MNIST and CIFAR10. All of these properties are supported by our approach. The final benchmark is a set of 32 local robustness properties with hyper-rectangle input constraints applied to neural networks with non-linear activation functions (*sigmoid* and *tanh*) trained on MNIST. All of these properties are supported by our approach.

Several neural network verifiers have recently been introduced. The nnenum verifier [10] and an abstraction-refinement approach for neural network verification [35] were evaluated on the ACAS Xu benchmark. The reachability set representation of ImageStars [128] was evaluated on two bench-

marks of local robustness properties applied to MNIST and ImageNet networks. The benchmark on the MNIST networks used a version of 900 local robustness where pixels could be independently darkened, enabling input constraints to be represented as hyper-rectangles. The benchmark on the ImageNet networks uses 6 properties created from an original image and a corresponding adversarial example. The properties specify that for a given region along the line between the original image and adversarial example, all inputs along the segment are classified as the correct class. Both the MNIST and ImageNet benchmarks are supported by our approach. The NNV verifier [129] also introduced a benchmark with an adaptive cruise control (ACC) system. It checks a temporal property not currently supported by Algorithm 1, however we present rewritings for such properties in Chapter 7.

Overall, we find that the property specifications accepted by Algorithm 1 are rich enough to express 7 of the 8 property types found in the explored benchmarks.

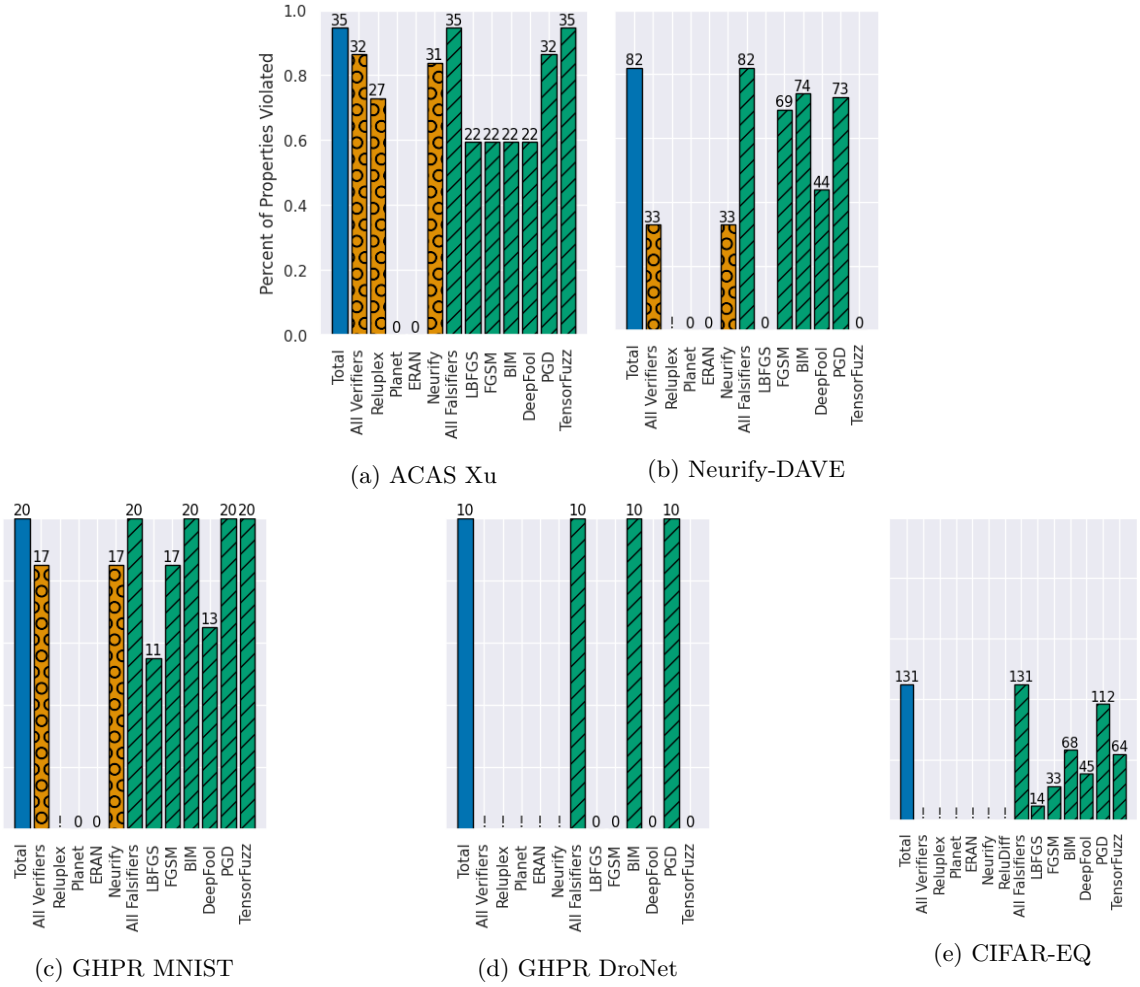
When considering the listed benchmarks, Algorithm 1 fully supports 16 of the 17 benchmarks. Our analysis also shows that the current space of neural network properties has limited diversity, with most benchmarks consisting primarily of local robustness properties. This points to the value added of the new benchmarks we introduce. It is also expected, as has happened in the verification community in the past, that as verification and falsification techniques improve, developers will want to apply them to reason about a broader range of correctness properties. The proposed algorithm will enable them to do that, even if verifiers and falsifiers do not directly support them.

4.2.2 RQ2: On the Cost-Effectiveness of Reduction-Enabled Falsification

To evaluate the cost-effectiveness of falsification enabled by the proposed reduction, we identify a set of falsifiers and verifiers to compare their complementary performance, problem benchmarks, and metrics that constitute the basis for the studies around RQ2 and RQ3.

Setup

Falsifiers. As falsification methods, we will use several common adversarial techniques, as well as a DNN fuzzing tool. For adversarial attacks, we choose a subset of the methods from two surveys [1, 149]. We select the methods common to both surveys with L_∞ input constraints (which matches our implementation) and with implementations available in the cleverhans tool [93]. The chosen adversarial attacks are LBFGS [123], FGSM [45], Basic Iterative Method (BIM) [77], and DeepFool [87]. Of these attacks, none use random initialization, and thus will produce the same



(a) ACAS Xu

(b) Neurify-DAVE

(c) GHPR MNIST

(d) GHPR DroNet

(e) CIFAR-EQ

Figure 4.2: The number of violations found by each falsifier and verifier, reduced by the total number of potentially falsifiable properties. The number above each bar gives the total number of violations found. An exclamation point indicates that a verifier could not be run on a property due to the structure of the network.

result over multiple runs. In order to observe the potential benefits of random initialization, we also include Projected Gradient Descent (PGD) [85], which was only included in one of the surveys. Therefore, we run each attack, except PGD, once, and if no adversarial example is found, we return an *unknown* result. For PGD, if no adversarial example is found, we try again, until one is found, or the given time limit is reached. We use the default parameters for each attack, as specified by cleverhans. For DNN fuzzing, we use TensorFuzz [90] for its easily accessible implementation [89]. TensorFuzz requires the definition of an oracle for recognizing property violations. We provide a

version of TensorFuzz with an oracle that identifies violations by checking whether $\mathcal{N}(x)_0 \leq \mathcal{N}(x)_1$[5].

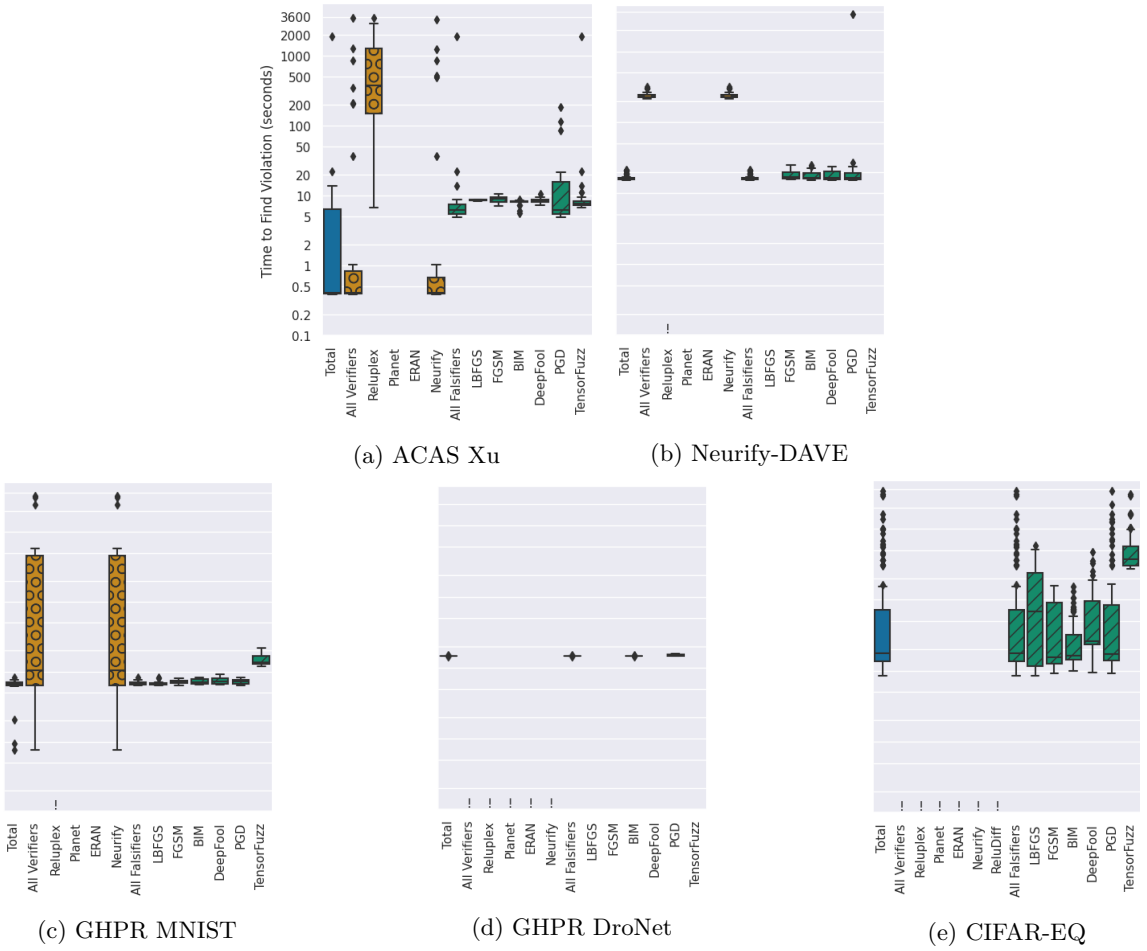
Verifiers. For comparison to verification, we select four verifiers: Reluplex [68], Planet [34], ERAN using the DeepZono abstract domain [115], and Neurify [135]. Neurify and ERAN have been shown to be fastest and most accurate in recent studies [146], and all four verifiers are supported by DNNV, which makes them easy to run and allows us to use a common property specification for all verifiers and falsifiers. For differential properties we also consider ReluDiff [96] since it is currently the only verifier built to handle such properties. However, ReluDiff only supports fully connected networks and so does not run on the chosen benchmarks.

Portfolios. In addition to the individual falsifiers and verifiers, we simulate portfolios of these methods, which run analyses in parallel and return the first result. We use 3 portfolios: *All Falsifiers*, which includes the 6 falsifiers described above; *All Verifiers* which includes all verifiers run on each benchmark; *Total* which includes all methods used in this study. To simulate running each portfolio, we take the union of the violations found by each method in the portfolio, and consider the time to find each violation to be the fastest time among the methods in the portfolio which found that violation.

Problem Benchmarks. We evaluate our approach on two common and representative benchmarks from the verification literature, and two created for this work to provide a range of networks and property types. Our selection criteria was meant to achieve two objectives. First, we wanted to select enough benchmarks to explore all property types with hyper-rectangle input constraints. Second, we wanted to select benchmarks with networks that varied in both size and structure since these factors have been shown to affect verifier performance [146].

From the verification literature, we select *ACAS Xu*, the most commonly used benchmark, and a slightly modified version of the *Neurify-DAVE* benchmark. For Neurify-DAVE, we select the 50 L□R properties supported by our current implementation, and we augment the benchmark with an additional network. The new network is the original DAVE DNN on which the smaller network in the benchmark was based. While the small DNN has 10277 neurons, the original DAVE network that we add has 82669 neurons, which will allow us to explore the scalability of reduction and falsification. The two networks in this benchmark are convolutional networks and are much larger than the networks in the ACAS Xu benchmark.

[5] https://github.com/dlshriver/tensorfuzz



(a) ACAS Xu

(b) Neurify-DAVE

(c) GHPR MNIST

(d) GHPR DroNet

(e) CIFAR-EQ

Figure 4.3: The times, in seconds, to find violations for each verifier and falsifier. An exclamation point indicates that a verifier could not be run on a property due to the structure of the network.

We developed 2 new benchmarks to cover property types that are not yet covered by existing benchmarks. The *GHPR* benchmark is a new DNN property benchmark that contains G□R properties applied to several network architectures of varying size and structure. It consists of 30 correctness problems, 20 of which are 10 GHPR properties applied to 2 MNIST networks, and 10 of which are GHPR properties applied to the DroNet DNN described previously. The DroNet DNN is one of the largest in our study, with more than 475,000 neurons. The MNIST properties are of the form: for all inputs, the output values for classes a and b are closer to one another than either is to the output value of class c. The DroNet properties are of the form: for all inputs, if the probability of collision is between p_{min} and p_{max}, then the steering angle is within d degrees of 0. These properties are described in more detail in Appendix C.

The *CIFAR-EQ* benchmark is a new DNN property benchmark with differential properties applied to large networks with complex structures. It contains a mix of both global and local equivalence properties. It is the only benchmark to contain G□D properties, which were absent in the property benchmarks that we found. It consists of 291 properties: 91 global equivalence properties and 200 local equivalence properties. Of the global properties, 1 is untargeted, while the other 90 are targeted equivalence properties. Of the 200 local properties, 20 are untargeted, while the other 180 are targeted equivalence properties. The properties are applied to a pair of neural networks trained on the CIFAR dataset [75]. The first network is a large convolutional network with 62,464 neurons and the second is a ResNet-18 network with over 588,000 neurons. These properties are described in more detail in Appendix C. Because the verifiers do not support the multiple computation path structure formed during network composition, we do not run them on this benchmark.

Metrics. For each verification and falsification approach, we will measure the number of property violations found and the total time to find each violation. The total time to find a violation includes both the time to transform the property, as well as to run the falsifier on the resulting properties.

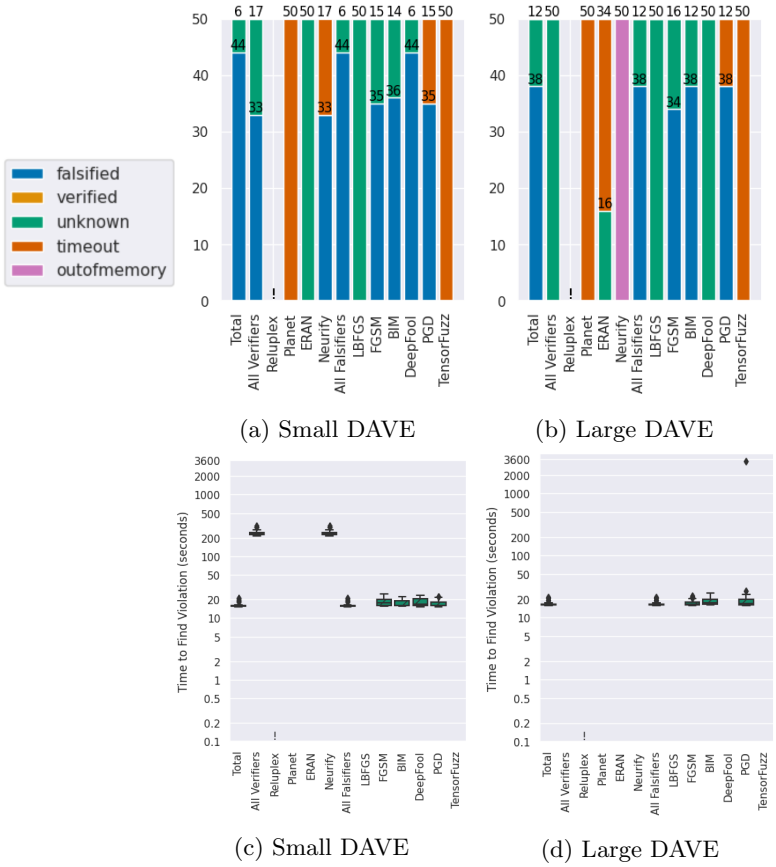
Computing resources. Experiments were run on compute nodes with Intel Xeon Silver 4214 processers at 2.20 GHz and 512GB of memory. Jobs were allowed to use up to 8 processor cores, 64GB of memory, with a time limit of 1 hour.

Results

Figure 4.2 shows the number of violations found by each verifier and falsifier method on the five benchmarks. The y-axis is the proportion of non-verified properties for which the techniques could find violations. We eliminated correctness problems that were known to be unfalsifiable. For ACAS this leaves 37 correctness problems, and does not reduce any of the other benchmarks. The number above each bar in the plots indicates the number of violations found. An exclamation point indicates that the verifier could not be run due to the architecture of the networks being verified.

The ACAS Xu benchmark with its simple and small DNN models, often used in verifier evaluation, showcases where verifiers perform best today. However, even in this benchmark we notice that falsification can complement verification, finding an additional 3 violations.

On the Neurify-DAVE benchmarks, the verifiers find only 33 violations from the 100 DNN correctness problems, while the falsifiers find 82 violations, subsuming the 33 violations from the verifiers. The best performing falsification method on this benchmark was BIM, with 74 violations found. PGD and FGSM follow closely with 73 and 69 violations found, respectively. TensorFuzz, the top



(a) Small DAVE (b) Large DAVE

(c) Small DAVE (d) Large DAVE

Figure 4.4: The number of violations and time to find each violation for the Neurify-DAVE benchmark. The number above each bar gives the total number of property check results in the bar below it. An exclamation point indicates that a verifier could not be run on a property due to the structure of the network.

performing falsification method on the ACAS benchmark, does not find any violations. We conjecture that this is due to the much larger input space. While the ACAS Xu networks have an input dimension of 5, the DAVE networks have an input dimension of 30000, which is more difficult to cover by random fuzzing.

On the GHPR MNIST benchmark, the verifiers can find 17 violations for the 20 properties, while 3 falsifiers, BIM, PGD, and TensorFuzz, can find violations for every property.

On the GHPR DroNet benchmark, the verifiers cannot find any violations, due to not supporting the residual block structures present in the DroNet network. Many of the falsifiers also struggle on these properties, except for PGD and BIM, which can find violations to all 10 properties.

Finally, on the CIFAR-EQ benchmark, the verifiers did not find any violations because they could not be run. Reluplex, Planet, ERAN, and Neurify do not support properties over multiple

networks or networks with multiple computation paths, while ReluDiff is limited to networks with only fully-connected layers. Additionally, while PGD finds the most violations, it is complemented by the other falsification approaches, with BIM, DeepFool, and TensorFuzz each finding violations for at least 1 unique property. We conjecture that much of PGD's success is due to its random initialization, which allows it to be run multiple times with different results, increasing the chance of finding a violation.

Note that the Planet and ERAN verifiers find no violations for any benchmark. For these benchmarks ERAN cannot find violations since its algorithmic approach focuses on proving that a property holds, which suggests its complementarity with falsification methods. Planet fails to find violations due to the complexity of the problem, and internal tool errors that cause Planet to crash on almost 20% of the correctness problems. We also see that the Reluplex verifier only finds violations for the ACAS Xu benchmark. It cannot find violations on the other benchmarks, since it does not support the architectures of the networks in those benchmarks.

Overall, we find that falsifiers can detect many property violations usually complementing those found by verification, that applying them in a parallel portfolio can leverage their unique strengths, and that they successfully scale to more complex benchmarks.

Box plots of the distributions of time to find violations for each method are shown in Figure 4.3. Figure 4.3a shows that the verifiers can be effective on the ACAS Xu benchmark, with Neurify often out performing the falsifiers. This is likely due to the extremely small size of the ACAS Xu networks enabling verification to run efficiently. These plots also show the efficiency struggle of the verifiers as the network get larger. For example, on the Neurify-DAVE benchmark, even when the verifiers can find a property violation, the falsifiers can find violations an order of magnitude faster. For more complex benchmarks, the verifiers cannot find violations within the timeout, so we only report the time for the falsifiers.

We find that falsification can efficiently find property violations even for the most complex benchmarks, with a median time to find a violation across all benchmarks and falsifiers of 16 seconds.

Figures 4.2 and 4.3 also reveal that no single falsifier always outperforms the others. While PGD performs well for the benchmarks studied here, we can still increase the number of violations by running mutliple falsifiers. Additionally, the falsifiers that find the most violations, do not necessarily always find them the fastest. Based on these two observations, we recommend using a portfolio approach, running many falsifiers in parallel and stopping as soon as a violation is found

by any technique, such as the *All Falsifiers* method shown in the previous figures. This approach finds all the violations found by the verifiers as quickly as the fastest falsifier. We also recommend using falsifiers in conjunction with verifiers, since while falsifiers can often quickly find violations, they cannot prove when a property holds.

4.2.3 RQ3: On the Scalability of Reduction-Enabled Falsification

Setup

To explore the scalability of falsification with reduction, we want to evaluate a set of properties across networks that vary in size. To do this we applied the Neurify-DAVE properties to both the small DAVE network [135] and the original larger DAVE network [15]. This will allow us to see how performance of the verifiers and falsifiers change with respect to the size of the network being verified.

Results

We present the results of checking the properties in Figures 4.4a and 4.4b, as well as the box plots of the times to find violations for each method in Figures 4.4c and 4.4d.

On the smaller DAVE network, the verifiers struggle to verify the properties. Reluplex does not run at all, due to its lack of support for convolutional layers, while Planet does run, but reaches the timeout for all properties. The ERAN verifier does not timeout on the small network, but cannot verify any of the properties. Neurify was the only verifier that returned accurate results on the small network, successfully falsifying 33 of the 50 properties, and reaching the time limit on the other 17. While only a single verifier was able to falsify any properties, 4 of the 6 falsification approaches were able to falsify properties, all of them finding more violations than Neurify. The falsifiers were also faster than Neurify, finding violations almost an order of magnitude faster than Neurify on the small DAVE network.

While one verifier was able to find violations on the smaller network, none of the verifiers were able to find violations on the larger DAVE network, which has more than 8 times more neurons. Similar to the small network, Reluplex does not support the network structure, while Planet reaches the time limit for all properties. However, ERAN and Neurify both perform slightly differently. While ERAN was previously able to finish its analysis on the small network, it reaches the time limit for 34 properties on the large network, indicating that it could not scale to the larger network size. Similarly, while Neurify previously found property violations for the small network, it reaches the

memory limit on the large network before any violations are found. The falsifiers on the other hand still perform well, with 3 of the 6 verifiers finding property violations. Surprisingly, the DeepFool falsifier goes from 44 violations found on the small network, to 0 violations on the large DAVE network. We conjecture that this may be due to the use of the default parameters for DeepFool, and that adjusting these parameters may yield better results. Additionally, the falsifiers show only a minor increase in the time needed to find a violation, from a median time of 20.2 seconds to 20.7 seconds.

Overall we find that, on the benchmarks explored here, DNN property reduction scales well to larger networks, and enables the application of scalable falsification approaches such as adversarial example generation.

4.3 Conclusion

In this chapter we have presented an approach for reducing neural network correctness problems to facilitate the application of falsifiers and verifiers to complex property specifications. We implement our approach for both falsifiers and verifiers and apply it to a range of correctness problem benchmarks and find that 1) the reduction approach covers a rich set of properties, 2) reducing problems enables falsifiers to find property violations, and 3) since falsifiers tend to have different strengths, a portfolio approach can increase the violation finding ability.

Chapter 5

Refactoring Neural Networks for Verification

Despite significant progress in the neural network verification, the state-of-the-art tools are still far from adequate for addressing the scale and complexity of DNNs used in autonomous systems. End-to-end networks for autonomous driving [15, 28] and flying [66, 82] are often too complex for cost-effective verification. These systems are generally developed with an exclusive focus on achieving a low error, which incentivizes large complex architectures. Unfortunately, this can prohibit usage of certain tools or substantially increase verification time.

For example, consider the network for the pendulum control example introduced in Figure 3.1, which we reproduce on the left side of Figure 5.1. This network contains operations and structures rarely supported by verifiers of neural networks. Specifically, the network makes use of a clip operation and has a non-sequential structure, using the output of an operation as inputs to multiple subsequent operations. One possible solution is to simply train a new network without this unsupported operation, but this risks significantly changing the behavior of the network. We assume that the original network architecture was selected using domain knowledge such that it learned the desired behavior to an acceptable precision. Our goal is to rewrite this trained network to one that is amenable to verifiers while preserving as much of its behavior as possible.

Our insight is that we can adapt the concept of refactoring to neural networks. Code refactoring [38] typically seeks to (a) restructure a software system to facilitate subsequent development activities, while (b) preserving the behavior of the original software system. Fowler's original focus was on improving maintainability and extensibility, but researchers have adapted the concept in

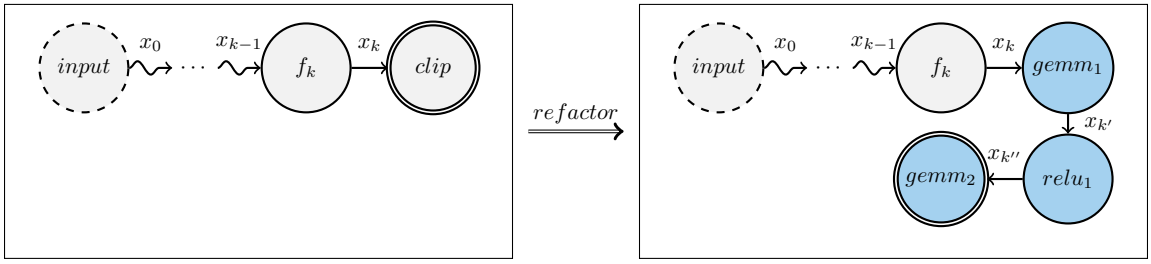


Figure 5.1: A neural network controller for the pendulum control example is refactored to increase its verifiability by replacing the clip operation with an equivalent formulation using GeMM and ReLU operations.

myriad ways. For example, to enhance the testability [27, 47] and verification [148] of software. In this chapter we introduce two approaches for refactoring neural networks for verification, a semantics preserving approach and a non-semantics preserving approach.

We first introduce semantics preserving refactorings which can identify specific sub-graphs of an operation graph, and replace them with sub-graphs of equivalent behavior. These refactorings are implemented, along with property reduction, as part of the DNNV tool introduced in Section 4.1.5. By replacing sub-graphs containing unsupported operations with sub-graphs containing supported operations, we can increase the verifiability of a network. For example, the Clip operation in the network on the left side of Figure 5.1 can be replaced by a sub-graph containing a sequence of GeMM and ReLU operations which encode the same behavior, as shown on the right side of the figure. This type of refactoring requires coming up with encodings of operations as combinations of other operations which are more generally supported by verifiers. While this can have a high one time cost for development, it can significantly increase the applicability of verifiers without requiring any additional user intervention, such as providing a transformation specification or replacing the original network.

In some cases, a network may contain operations or architectures for which a semantics-preserving refactoring has not been developed. For example, a network may contain a non-linear Sigmoid operator, while the desired verifier only supports ReLU operators. In other cases, the network structure may be supported by a given verifier, but be too large to be efficiently analyzed. For such networks, we introduce the non-semantics preserving *refactoring for verification* (R4V), in Section 5.1.2. This refactoring first transforms the network architecture to both remove unsupported operations and decrease the network complexity. We choose to transform the original architecture rather than develop a completely new model under the assumption that the original architecture encodes some domain knowledge of the designer and that in many cases, only small transformations

will be necessary to make a network more verifiable. For the example network in the left half of Figure 5.1, this transformation might simply be to drop the Clip operation from the network. Next, R4V uses knowledge distillation to transfer the behavior of the original model to the transformed model. This process results in a more verifiable network with behavior highly similar to the original. After refactoring, the original network should be discarded, and the new model should replace the original.

This chapter makes the following contributions:

1. Development of both semantics preserving and non-semantics preserving automated refactoring methods for neural networks to increase verifiability.

2. An analysis of semantics-preserving refactoring in conjunction with reduction across a larges space of verification benchmarks, demonstrating its potential to increase the applicability of many existing verifiers, off-the-shelf.

3. A study of non-semantics preserving refactoring on 3 case studies, demonstrating its ability to increase both verifier applicability and efficiency.

5.1 Approach

We introduce two refactoring approaches for neural networks: a semantics preserving approach, DNNV, and a non-semantics preserving approach, R4V.

5.1.1 Semantics Preserving Refactoring

We first introduce a semantics preserving approach to refactor networks with the goal of increasing the applicability of verifiers. These refactorings produce networks that are more amenable to verification by a given verifier, and for which, for all network inputs, the refactored network produces the same output as the original. Formally, we define semantics preserving refactorings as:

$$\mathit{refactor}([\mathcal{E}, \mathcal{N}, \phi]) = \{[\mathcal{E}, \mathcal{N}', \phi]\} \tag{5.1}$$

where the rewritten properties must satisfy:

$$\forall [\mathcal{E}', \mathcal{N}', \phi'] \in \mathit{refactor}([\mathcal{E}, \mathcal{N}, \phi]).\ \forall x \in \mathcal{X}.\ \mathcal{N}'(x) = \mathcal{N}(x) \tag{5.2}$$

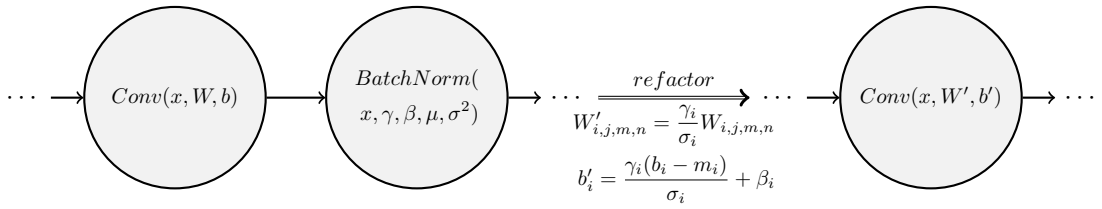


Figure 5.2: Batch normalization operations can be refactored out of a network by combining them with a preceeding convolution operation.

We implement several semantics preserving refactorings as part of DNNV, which also implements property reduction (introduced in Section 4.1.5). DNNV takes in a network and property, applies semantics preserving rewritings, and runs a user specified verifier on the rewritten sub-problems. It currently includes 6 semantics preserving refactorings, which we describe below. These refactorings were selected based on our experience verifying neural networks and are sufficient for the networks we have studied, but many additional refactorings are possible and will be investigated and implemented in future work.

Batch Normalization Refactoring. *Batch normalization refactoring* removes batch normalization operations from a network by combining them with a preceding convolution operation or generalized matrix multiplication (GeMM) operation. This is possible since batch normalization, convolution, and GeMM operations are all affine operations. The refactoring of a batch normalization operation following a convolution operation is shown in Figure 5.2. If no applicable preceding layer exists, the batch normalization layer is converted into an equivalent convolution or GeMM operation, depending on the shape of the inputs to the operation, as shown in (5.3). This refactoring enables the application of verifiers without explicit support for batch normalization operations, such as Neurify and Reluplex, to networks with these operations.

$$BatchNorm(x; \gamma, \beta, \mu, \sigma^2) \rightarrow \begin{cases} Conv(x; reshape(diag(\frac{\gamma}{\sigma}), (c, c, 1, 1)), (\beta - \frac{\gamma\mu}{\sigma})) \\ GeMM(x; diag(\frac{\gamma}{\sigma}), (\beta - \frac{\gamma\mu}{\sigma})) \end{cases} \tag{5.3}$$

Identity Refactoring. *Identity refactoring* removes sequences of operations that result in no change to the input to the sequence. Such sequences only add complexity to the model and have no effect on the behavior, and can be safely removed. For example, explicit Identity operations (which may be added by tools when exporting models to ONNX) simply output their inputs and are removed by this refactoring. In addition, this refactoring checks for Concat operations with

a single input operation, Flatten operations applied to already flat tensors, and ReLU operations applied to vectors that are guaranteed to contain positive values (such as the outputs of another ReLU or Sigmoid operation). If any of these operation sequences are found they are removed by this refactoring.

Affine Refactoring. The *fully-connected refactoring* converts MatMul and Add operations into equivalent GeMM operations, as follows:

$$Add(MatMul(x; W); b) \rightarrow GeMM(x, W, b) \tag{5.4}$$

$$Add(x; b) \rightarrow GeMM(x, I, b) \tag{5.5}$$

$$MatMul(x; W) \rightarrow GeMM(x; W, \mathbf{0}) \tag{5.6}$$

This refactoring does not necessarily increase verifier support, but it simplifies the specification of other refactorings, since it enables them to be defined only over the GeMM case.

Operation Combination Refactoring. The *operation combination refactoring* combines certain pairs of consecutive operations with a single, semantically-equivalent operation. For example, because GeMM operations are affine operations, consecutive GeMMs can be combined into a single GeMM operation as shown in Figure 5.3. Additionally, consecutive Conv operations can be combined into a single Conv operation if the initial Conv has a kernel size of 1, with a diagonal weight matrix, and default values for all other parameters according to version 11 of the ONNX operator specification[1]. In this scenario, we can combine the two Conv operations as follows:

$$Conv(Conv(x; W^{(1)}, b^{(1)}); W^{(2)}, b^{(2)}) \rightarrow Conv(x; W, b) \tag{5.7}$$
$$\text{where } W_{i,j,k,l} = W^{(2)}_{i,j,k,l} * W^{(1)}_{j,j,0,0}$$
$$\text{and } b_i = b^{(2)}_i + \sum_{j,k,l} (W^{(2)}_{i,j,k,l} * b^{(1)}_j)$$

This configuration of the Conv operation can occur when we convert BatchNorm operations to Conv operations. Finally, explicit Pad operations preceding a Conv or pooling (e.g., MaxPool or AveragePool) can be combined with the padding parameters of the Conv or pool operation, if padding parameters match those of the Conv or pool. For example, if the Pad operation pads zeros to its input, then it can be combined with a succeeding Conv operation, but not necessarily with a

[1] `https://github.com/onnx/onnx/blob/main/docs/Operators.md#Conv`

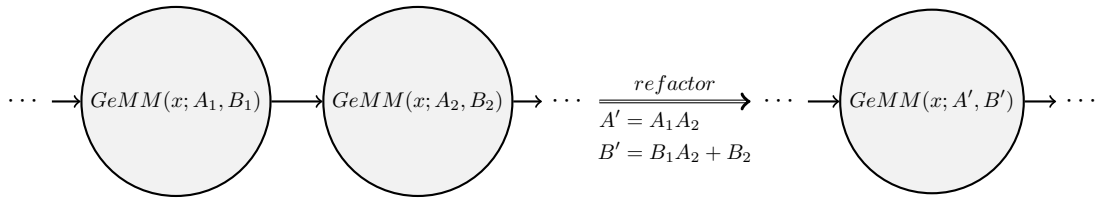


Figure 5.3: Consecutive GeMM operations can be refactored to a single GeMM operation.

succeeding MaxPool operation, which assumes that padding values will never be larger than values in the input.

Activation Placement Refactoring. *Activation placement refactoring* re-orders operations such that activation functions (e.g., Relu, Sigmoid, Tanh) do not come after reshaping operations (e.g., Transpose, Reshape, Flatten). If this refactoring finds an activation function after a reshape operation, it moves the activation operation backwards to the first non-reshape operation. This is possible since activation operations are element-wise operations, and reshaping operations do not change element values, only position. This refactoring facilitates operation graph pattern matching and other refactorings, such as identity refactoring, since it will ensure that activation functions are not separated by only reshaping operations.

Reluify MaxPool Refactoring. The *Reluify MaxPool refactoring* transforms MaxPool operations into an equivalent sequence of Conv and ReLU operations to enable verifiers without support for MaxPool operations to verify networks with these operations. This refactoring takes advantage of the equivalence $max(a, b) = relu(a-b)+relu(b)-relu(-b)$. The refactoring halves one dimension of the MaxPool kernel, and replaces it with an equivalent sequence of Conv, ReLU, Conv, which computes this max equivalence between pairs of input values. The output of this sequence has a new MaxPool operation applied to it, with the kernel size halved. This process is repeated until a MaxPool with a 1×1 kernel is obtained, which can be omitted as it is effectively an identity

operation. One refactoring step can be defined as follows,

$$MaxPool(x; k_1 \times k_2) \rightarrow MaxPool(Conv(ReLU(Conv(x; W^{(1)}, b^{(1)})); W^{(2)}, b^{(2)}); \frac{k_1}{2} \times k_2) \quad (5.8)$$

$$\text{where } W^{(1)}_{\cdot,j,2k,l} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 1 & 0 & 0 \\ -1 & 1 & -1 & \cdots & -1 & 1 & -1 \end{bmatrix}, \text{for } 0 \leq k < \frac{k_1}{2}$$

$$W^{(2)}_{i,\cdot,0,0} = \begin{bmatrix} 1 & 1 & -1 & \cdots & 1 & 1 & -1 \end{bmatrix}$$

$$b^{(1)} = b^{(2)} = 0$$

where the MaxPool is refactored in the first kernel dimension. Other dimensions can be refactored similarly. The above formulation also assumes that kernel dimensions are always even, however extension to odd dimensions is straightforward and can be accomplished by adding only a single row to $W^{(1)}$ for the last value of k when k_1 is odd that is the sum of the two rows inserted for all other values of k. This process takes $log_2(K)$ steps, where K is the original kernel size, and adds three operations at each step, however the consecutive Conv operations that appear from recursively applying this refactoring can be merged using the operation combination refactoring defined in (5.7). This refactoring can significantly increase the network size for large MaxPool kernels, however it enables verifiers to run on networks which they previously would not support. Due to the potential cost, this refactoring must be explicitly enabled by a user of DNNV.

Implementation

We have implemented these semantics-preserving refactorings in a tool called DNNV. DNNV takes in networks in the ONNX format, and properties in a custom DSL named DNNP (described in more detail in Appendix B). It applies semantics-preserving network refactoring and property reduction (see Section 4.1) and translates the rewritten subproblems to the input format of any of 13 verification techniques supported by the tool.

DNNV is publicly available at `https://github.com/dlshriver/dnnv`.

5.1.2 Non-Semantics Preserving Refactoring For Verification

While semantics preserving refactoring can produce equivalent networks, it is not always possible to replace unsupported operations with supported ones in an equivalent manner. To address this, we introduce a non-semantics preserving approach with the goal of increasing the applicability and scalability of verification, which we call R4V. An overview of R4V is depicted in Figure 5.4. The

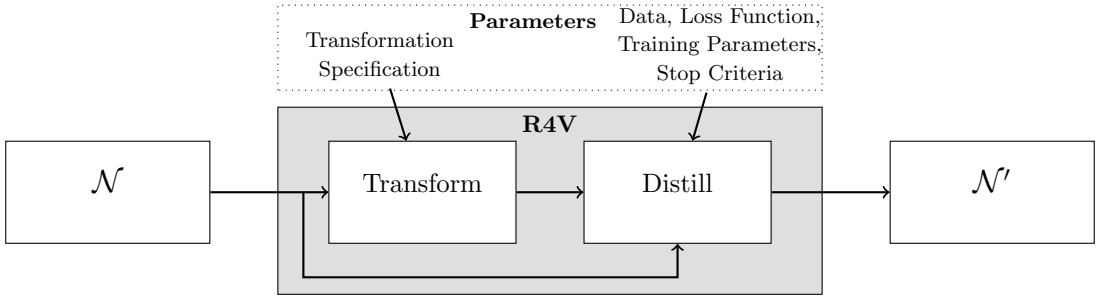


Figure 5.4: An overview of R4V.

original network is first transformed according to a user provided specification, designating which operations to drop, scale, or transform to meet verifier requirements. This transformation proceeds by propagating inter-operation constraints to produce a well-defined, but untrained, network that is less complex than the original and thereby facilitates verification. This transformed architecture is trained using knowledge distillation which seeks to match the accuracy of the original, trained network [51]. Formally, we specify this refactoring as:

$$refactor_{R4V}([\mathcal{E}, \mathcal{N}, \phi]) = \{[\mathcal{E}, \mathcal{N}', \phi]\} \tag{5.9}$$

where $\mathcal{N}'$ satisfies $\forall x \in \mathcal{X}. \mathcal{N}'(x) \sim \mathcal{N}(x)$ for some measure of similarity, $\sim$, provided by the developer. In the rest of this section we describe these two components, transformation and distillation, in more detail and describe our implementation and recommendations for use.

Transformations

Many different strategies can be used to reduce the complexity of a neural network. Here, we define a simple set of transformations over network operations, focused on those that can reduce the complexity of networks. These transformations permit dropping operations and scaling certain operation types, since these can reduce the size of networks or remove unsupported operations. We also define a utility operation to apply transformations to operations that satisfy a given predicate.

Drop. The drop transformation (Eq. 5.10) takes in a set of operations $\mathcal{O}$, and removes those operations from the network.

$$drop(\mathcal{O}) : \forall o \in \mathcal{O}.\ o' = \bot \tag{5.10}$$

After dropping an operation, the output shape of the preceding operation is used to update successive operations, to ensure that the new architecture is valid. The drop transformation can be applied to operations that perform some computation on their input. We prohibit its application to the output operation to ensure that the transformed architecture produces outputs with the same dimensions as the original architecture. We also prohibit dropping reshaping operations, such as *Transpose* and *Flatten*, since they are required to ensure that the input to an operation is the correct shape.

Scale. The scale transformation (Eq. 5.11) takes in a set of operations $\mathcal{O}$, and a scale factor f, and scales the output size of each specified operation by the factor f.

$$scale(\mathcal{O}, f) : \forall o \in \mathcal{O}\ (o \neq \bot) \rightarrow (\#o' = \lfloor f \cdot \#o \rfloor) \tag{5.11}$$

where # denotes the output size of an operation. After scaling an operation, its new output shape is used to update the parameterization of successive operations, to ensure that the new architecture is valid. Our current framework supports the scale transformation on the input operation, GeMM operations, and Convolution operations. GeMM operations are scaled by changing the size of their weight matrix and bias vector. Convolution operations are scaled by changing the number of kernels. The input operation is scaled by changing its specified size.

Forall. We also define a utility transformation *forall* (Eq. 5.12), that applies a partially parameterized transformation λ over all operations of a neural network $\mathcal{N} = \langle Q, q_0, q_*, \delta \rangle$, that satisfy a predicate φ.

$$forall(\langle Q, q_0, q_*, \delta \rangle, \varphi, \lambda) = \lambda(\{o \in Q \mid \varphi(o)\}) \tag{5.12}$$

We have applied and investigated these transformations individually and in combination over several networks, including the ones that we report on in Section 5.3. We have considered extending the transformation framework with additional capabilities, for example, to *Add* an operation. To date, however, we have focused on capabilities that reduce the complexity of refactored networks – adding operations generally works against that goal – and that are necessary to support existing verifiers for neural networks.

Distillation

After transforming the architecture, we must transfer the knowledge of the original model to the untrained transformed network. To do this we employ *knowledge distillation*, which was introduced by Hinton et al. [51] as a general approach to addressing the differing requirements in training and deploying neural networks (e.g., less computation, lower energy consumption). Distillation trains a *student* network to match the accuracy of a trained *teacher* model. It has proven useful in addressing varied deployment requirements [24, 53, 103]. In R4V, we employ distillation to address the requirement that neural networks be verified prior to deployment.

Distillation is a flexible, highly parameterizable framework for training neural networks, and we identify three degrees of freedom to adapt distillation for R4V: datasets, training parameters, and stopping rules. Once parameterized, R4V trains the student network automatically. The resulting network can be assessed for accuracy and subjected to property verification.

Datasets. Separate datasets can be specified for training, validation, and testing. Additionally, while the same datasets will generally be used for the teacher and student, R4V supports the utilization of different datasets for each, enabling enabling data to be pre-processed offline to accelerate distillation.

Training Parameters. As with other neural network training approaches, there is a large hyperparameter space to tune distillation performance. For refactoring classification networks, R4V follows prior work beginning with Hinton et al. [51], incorporating the output of the teacher as target vectors, however the true labels can also be used to augment computation of the loss. For refactoring regression networks, we distill strictly using the outputs of the teacher as the target vectors.

Stopping Rules. In addition to establishing a timeout for distillation, R4V can terminate distillation when the relative student-teacher error is below a given threshold. For example, in a regression setting, the MSE between the teacher and student outputs may be used to measure relative accuracy.

Implementation

Our implementation of R4V consists of two main components: a transformation component, and a distillation component. Both components are parameterized using a single configuration file, specified using the TOML language [100], such as the one shown in Listing 1.

The transformation component is parameterized with a teacher architecture and a set of transformation strategies. The teacher model is specified using the Open Neural Network Exchange (ONNX) format [92], which can be produced from many of the most popular machine learning frameworks. The configuration file in Listing 1 will transform the teacher architecture (specified in lines 22-25) by dropping layers 2 and 3 (lines 26-27), linearizing all residual blocks in the network (lines 28-30), and scaling the first layer by a factor 0.5 (lines 31-33). After transforming the architecture, the R4V tool will automatically begin distillation.

```
1  [distillation]
2  maxmemory = "32G"
3  threshold = 0.01
4  type = "regression"
5  [distillation.parameters]
6  epochs = 100
7  optimizer = "adam"
8  loss = "MSE"
9  [distillation.data]
10 format = "dronet"
11 batchsize = 32
12 [distillation.data.transform]
13 grayscale = true
14 [distillation.data.train]
15 shuffle = true
16 student.path = "artifacts/dronet.200/training"
17 teacher.path = "artifacts/dronet.200/training"
18 [distillation.data.validation]
19 shuffle = false
20 student.path = "artifacts/dronet.200/validation"
21 teacher.path = "artifacts/dronet.200/validation"
22 [distillation.teacher]
23 model = "networks/dronet/model.onnx"
24 input_shape = [1, 200, 200, 1]
25 input_format = "NHWC"
26 [[distillation.strategies.drop_layer]]
27 layer_id = [2, 3]
28 [[distillation.strategies.forall]]
29 layer_type = "ResidualConnection"
30 strategy = "linearize"
31 [[distillation.strategies.scale_layer]]
32 layer_id = [0]
33 factor = 0.5
```

Listing 1: An example R4V configuration file.

The distillation component was implemented in Python, with PyTorch as the underlying machine learning framework [95] used for training. The default distillation settings are based on the knowledge distillation method of Hinton et al. [51], and can be configured through the configuration file provided by the user. R4V allows users to choose the training algorithm and loss function (lines 7 and 8), as well as set hyper parameters such as the learning rate, and the batch size (line 11). We also extend distillation with a relative-performance-based stopping criteria (such as the error threshold specified in line 3), as well as a memory limit (line 2) and timeout. These permit developers to enforce a distillation budget to allow for fast partial distillation training to rapidly explore the space of refactorings; something we plan to explore in more depth in future work. Our distillation procedure also allows for different datasets to be specified for the student and the teacher (lines 14-21). This allows the student to be trained on smaller input sizes or on inputs that have been pre-processed differently than the teacher. For example, in the second case study for R4V, in Section 5.3.3, we transform the DroNet network to operate on smaller input sizes, so we update lines 16 and 20 to point to a pre-processed dataset with the smaller image sizes. The distilled student model is saved

Table 5.1: Verifier benchmarks.

Key	Name	Uses	#P	#$\mathcal{N}$	¬HR	Features C	R	¬ReLU
AX	ACAS Xu	[10, 20, 68, 69, 135]	10	45				
CD	Collision Detection	[20, 34, 69]	500	1				
PM	*Planet* MNIST	[34]	7	1	✓	✓		
TS	TwinStream	[19]	1	81				
PCA	PCAMNIST	[20]	12	17				
MM	*MIPVerify* MNIST	[125]	10000	5		✓		
MC	*MIPVerify* CIFAR10	[125]	10000	2		✓	✓	
NM	*Neurify* MNIST	[49, 135]	500	4		✓		
NDB	*Neurify* Drebin	[135]	500	3				
NDV	*Neurify* DAVE	[135]	200	1	✓	✓		
DZM	*DeepZono* MNIST	[115]	1700	10		✓	✓	✓
DZC	*DeepZono* CIFAR10	[115]	1700	5		✓		✓
DPM	*DeepPoly* MNIST	[49, 116]	1500	8		✓		✓
DPC	*DeepPoly* CIFAR10	[116]	800	5		✓		
RZM	*RefineZono* MNIST	[117]	800	8		✓		
RZC	*RefineZono* CIFAR10	[117]	200	2		✓		
RPM	*RefinePoly* MNIST	[114]	600	6		✓		
RPC	*RefinePoly* CIFAR10	[114]	300	3		✓	✓	
VC	*VeriNet* CIFAR10	[49]	250	1		✓		

in the ONNX format.

Our implementation is publicly available at `https://github.com/dlshriver/R4V`.

5.2 Study: Semantics Preserving Refactorings

We now examine the applicability of verifiers to existing verification benchmarks with and without DNNV. A verification benchmark consists of a set of verification problems which are used to evaluate the performance of a verifier. A problem is made of a neural network and a property specification and asks whether the property is valid for the given neural network. We consider a verifier to support a benchmark if it can be run on that benchmark out of the box. We consider a verifier to have support for a benchmark through DNNV if it would be possible to run DNNV on that benchmark with networks specified using ONNX and properties specified in DNNP, by reducing the properties, refactoring the networks, and translating the problem to work with the target verifier. Note that, in addition to network refactoring, DNNV also performs property reduction as introduced in Section 4.1.5.

Table 5.2: Benchmark support by each verifier. The left half of the circle is black if the verifier can support the benchmark out of the box, and is white otherwise. The right half is black if the verifier supports the benchmark through DNNV, and is white otherwise. An absent circle indicates that the verifier can not be made to support some aspect of the benchmark.

	Benchmark																		
Verifier	AX	CD	PM	TS	PCA	MM	MC	NM	NDB	NDV	DZM	DZC	DPM	DPC	RZM	RZC	RPM	RPC	VC
Reluplex	●	◐	○	●	●	○	○	○	◑	○				○	○	○	○	○	○
Planet	●	◐	●	●	●	◑	◑	◑	◑	◑				◑	◑	◑	◑	◑	◑
BaB	●	◐	●	●	●	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
BaBSB	●	◐	●	●	●	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
MIPVerify	◑	○	○	◑	◑	●	◐	○	◑	○				○	○	○	○	○	○
Neurify	●	○	◑	◑	◑	◑	○	●	●	●				◑	◑	◑	◑	○	◑
DeepZono	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
DeepPoly	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
RefineZono	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
RefinePoly	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
Marabou	●	◐	◑	●	●	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
nnenum	●	○	◑	◑	◑	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
VeriNet	◑	○	○	◑	◑	◑	○	●	◑	○	○	◑	◑	◑	◑	◑	◑	○	●

5.2.1 Benchmarks

To evaluate benchmark support, we collected the benchmarks used by each of the 13 verifiers supported by DNNV, and determined whether each verifier could run on the benchmark out of the box, and also whether they would be able to run on the benchmark when DNNV is applied. The verification benchmarks are shown in Table 5.1 and are also described in more detail in Appendix D. Each row of the table corresponds to a benchmark, to which we assign a short key for identification. For each benchmark, we give the name, some of the verifiers it evaluated, the number of properties (#P) and networks ($\#\mathcal{N}$), and features that can make it challenging for verifiers. These features include whether any properties cannot represent their input constraints using hyper-rectangles (¬HR), whether any network in the benchmark contains convolution operations (C), whether any network contains residual structures (R), and whether any network uses any non-ReLU activation functions (¬ReLU).

5.2.2 Results

The support of verifiers for each benchmark is shown in Table 5.2. Each row of this table corresponds to one of the 13 verifiers supported by DNNV, and each column corresponds to one of the 19

benchmarks identified in Table 5.1. Each cell of the table may contain a circle that identifies the support of the verifier for the benchmark. The left half of the circle is black if the verifier can support the benchmark out of the box, and is white otherwise. The right half is black if the verifier supports the benchmark through DNNV, and white otherwise. An absent circle indicates that the verifier can not be made to support some aspect of the benchmark. For the benchmarks shown here, this is always due to the presence of non-ReLU activation functions in some of the networks in the benchmarks and indicates a potential use case for the non-semantics preserving refactoring R4V.

As shown in Table 5.2, DNNV can dramatically increase the support of verifiers for benchmarks. For example, the Planet verifier would originally be able to run on only 5 of the 19 benchmarks, but would be able to run on 16 using DNNV. Similarly, the nnenum verifier, could originally only be run on 1 of the existing benchmarks, but would be able to run on 13 using DNNV. **Of the 223 pairs of verifiers and benchmarks for which support may be possible, 166 of them are currently supported by DNNV, an increase of over 2.4 times the 68 pairs supported without DNNV.**

5.3 Study: Non-Semantics Preserving Refactorings

We explore the potential of supporting neural network verification with R4V through 3 case studies on verifier applicability, verifier efficiency, and error-verifiability trade-offs.

The first case study focuses on Verifier Applicability. Verifiers are constantly playing catch-up to the latest operations and architectural features introduced for neural networks. As a result, a favorite verifier will often not support the features of a network of interest. For example, some verifiers, such as CROWN [151] and ReluVal [136], are restricted to fully-connected layers. Other verifiers, such as MIPVerify [125] and DeepGo [104], add support for Convolution and MaxPool operations, but cannot handle more complex operation graphs such as those with residual connections. In this case study, we showcase how R4V can help verifiers overcome this kind of limitation by refactoring the network to one that satisfies the verifier's feature constraints.

The second case study focuses on Verifier Efficiency. When faced with modern complex networks, even when verifiers can support the operations and structures in a network, they often struggle to provide results within an actionable time frame. In such cases R4V can refactor the network into one that can be verified significantly more efficiently.

The third case study focuses on Error-Verifiability Trade-offs. While the first two studies investigate R4V under a small number of refactorings, this study provides a much broader exploration

of the refactoring space. It demonstrates a binary search for refactored versions of the original network that lie in the *complexity sweet spot* where the error and verification time meet acceptable thresholds, providing concrete evidence of the trade-offs illustrated earlier in the chapter.

We explore these three case studies using two large neural networks, four verifiers, and multiple refactorings, described in more detail below.

5.3.1 Setup

Here we describe the correctness problems, verifiers, and methodology used in the three studies.

Correctness Problems

We evaluate R4V using correctness problems for two neural networks for both autonomous ground and aerial control. Both artifacts apply local reachability properties to large neural network models.

DAVE2 is a network trained to predict steering angles from color images from a front-facing camera to control an autonomous ground vehicle [15]. Because the original DAVE-2 model is not publicly available, we use the network trained by other researchers [97] on the Udacity self-driving car dataset [131]. This model was trained on color images which have been mean-centered and scaled to a height and width of 100 by 100. We use the same input pre-processing when refactoring the network. The network before refactoring has a MSE of 0.047 when run on the test set.

Following the process of Neurify [135], we randomly generated 10 safety properties for DAVE-2. The properties specify that images within an L_{inf} ball with radius 2 centered at a given test image must produce a steering angle within 15 degrees of the angle predicted for the original test image. Different from Neurify [135], we restricted the output to a tighter range (from ± 30 to ± 10 degrees) to make it more realistic for the problem domain.

DroNet is a network trained to provide control signals for an autonomous quadrotor [82]. It consumes a 200x200 grayscale image and outputs a steering angle, as well as a probability that the drone is about to collide with an obstacle. The model is trained on a version of the Udacity self-driving car dataset [131] augmented with a set of images taken from a camera mounted to a bicycle as it is ridden on the streets of a city. The Udacity dataset provides the data for the steering prediction, while the bicycle data provide data for the probability of collision prediction. The resulting network has a MSE of 0.013 for the steering angle and an accuracy of 0.93 for the collision prediction when evaluated on the test set.

We randomly generated 20 safety properties for the DroNet networks, 10 for the steering angle regression task, and 10 for the collision probability classification task. The steering properties specify that images within an L_{inf} ball with radius 2 centered at a given test image must produce a steering angle within 15 degrees of the angle predicted for the original. The collision probability properties specify that images within an L_{inf} ball with radius 2 centered at a given test image must produce a collision classification that is the same as the test image, e.g., if the probability was greater than 0.5 for the test image, it must also be greater than 0.5 for all images within the L_{inf} ball.

Verifiers

We selected four verifiers: ERAN [116], Neurify [135], Reluplex [68], and Planet [34]. These tools represent a variety of verification techniques. Reluplex is based on an adaptation of the simplex method that provides support for fully-connected networks with ReLU activation functions. ERAN uses abstract interpretation to overapproximate the reachable output region. We use ERAN with the DeepPoly abstract domain, which supports convolutional, fully-connected, and maxpooling layers with ReLU, tanh, or Sigmoid activation functions [116]. Neurify combines symbolic interval analysis with symbolic linear relaxation, and supports convolutional and fully-connected layers with ReLU activations. Planet blends techniques from SMT and search, and supports layers that can be specified as a linear combination of neurons, as well as ReLU activations and maxpooling layers. All these verifiers also have freely available open source implementations.

Unfortunately each of these tools has it's own input format, so, for each verifier, we wrote a translation tool to convert networks in the ONNX format and properties in a Python-embedded DSL (the output format of R4V) to the specific input format required by each verifier[2]. For some of the verifiers we had to make small modifications to get the tool to work on the artifacts used in this study. For Reluplex, we use the version of the tool modified by [20] to support generic properties. This version of the tool works by checking whether the output neuron is less than 0. Properties can be specified by encoding them as a set of layers at the end of the network being verified. Because the Neurify tool is hard-coded to check a constant set of properties, we modified the DAVE2 version of the tool to be more general by allowing a user specified input and output interval to be specified at runtime.

[2]This toolchain later became the initial version of the DNNV framework: `https://github.com/dlshriver/dnnv/commit/e516a51b49f127a723162cd5e1461cbee8245d66`

Methodology

For each case study, we use R4V to refactor the original neural networks. The transformations in each scenario are meant to demonstrate how R4V can be applied to increase verifiability. Distillation parameters are set to correspond to the training parameters reported for the original networks, including the number of epochs (50 for DAVE2 and 100 for DroNet), optimization method (Adadelta [150] for DAVE2 and Adam [72] for DroNet), and batch size (256 for DAVE2 and 32 for DroNet). After training, we select the model from the epoch with the best mean squared error (MSE) performance (relative to the teacher), evaluated on the validation set. For DAVE2, the refactored performance is measured as the MSE of the output steering angle with respect to the original network, evaluated on the test set. Because DroNet has 2 outputs, we report the refactored performance as 2 values: the MSE of the steering angle with respect to the original network, and the accuracy with respect to the original probability of collision, evaluated on the test set.

For each scenario in each case study, we verify the correctness problems described above. The first study pairs the DAVE2 problems with the Reluplex verifier and the DroNet problems with the ERAN verifier. The second study pairs the DAVE2 problems with the Neurify verifier and the DroNet problems with the Planet verifier. The third study uses only the DAVE2 problems, paired with the Neurify verifier. For each scenario we report the verification results (true, false, unknown, "out of resources" (OOR)) as well as the mean verification time.

Distillation took around 11 hours on average to complete all the epochs. We made no effort to optimize this time. As an example of the potential for optimization, the average best epoch for the accuracy-verifiability trade-off study below was 37 (out of 50 total), suggesting that early stopping could have reduced distillation time by up to 20%. More aggressive training optimizations, such as those used in neural architecture search [99], apply a strict training budget, e.g., 5% of the epochs, when making preliminary assessments about the suitability of an architecture and then only using larger budgets for very promising architectures. This could work well for the approach described in the third case study and would reduce the time for most distillation runs to a matter of minutes.

Computing Resources

Distillation tasks were run on GPU compute nodes with 64GB of memory, and using NVIDIA 1080Ti GPUs. Verification tasks were run on Linux compute nodes with 2.3GHz Xeon processors, 64GB of memory, and a time limit of 24 hours.

Table 5.3: Results for the Verifier Applicability study.

	Result Count				Time (sec.)		
Artifact/Verifier	T	F	U	OOR	T	F	U
DAVE-2/Reluplex	-	-	-	-	-	-	-
R4V(DAVE-2)/Reluplex	10	0	0	0	52	-	-
DroNet/ERAN	-	-	-	-	-	-	-
R4V(DroNet)/ERAN	8	12	0	0	519	525	-

5.3.2 Results: Verifier Applicability

This case study pairs the DAVE-2 and DroNet correctness problems with the Reluplex and ERAN verifiers, respectively. The pairing is designed to showcase the potential value of R4V to overcome a verifier's limited support for certain features of a network architecture. The original DAVE-2 network has several convolutional layers, which are not supported by Reluplex. The original DroNet network has complex residual blocks which are not supported by ERAN when using the DeepPoly domain.

Refactoring. Since Reluplex only supports fully-connected layers, DAVE2 must be refactored to remove its convolutional layers. We apply the transformation $forall(\mathcal{N}, isConvLayer, drop)$, where $isConvLayer$ is a predicate that determines whether an operation is part of a convolutional layer, resulting in a network consisting of only the final 5 fully-connected layers. R4V automatically adjusts the first of these layers to accept vectors with the same size as the input vector. To enable ERAN (using the DeepPoly domain) to run on DroNet, we first drop the convolution and MaxPool operations at the beginning of the DroNet network, as well as the last 2 residual blocks using the transformation $forall(\mathcal{N}, isNth(Conv, 1) \vee isNth(MaxPool, 1) \vee isNth(ResidualBlock, 2) \vee isNth(ResidualBlock, 3), drop)$, where $isNth(t, i)$ returns a predicate that returns whether an operation is the ith operation of type t in the network, counting from the inputs. We then linearize the remaining residual connection with $linearize(\mathcal{N}) = forall(\mathcal{N}, isResidualConnection, drop)$, where $isResidualConnection$ is a predicate identifies if an operation is part of a residual connection. The distillation step for both artifacts uses the parameters described in the previous section, resulting in a refactored DAVE2 network with an MSE of 0.062 relative to the original (other refactorings of DAVE2 that render much smaller errors are presented in the last case study), and a refactored DroNet with a relative steering angle MSE of 0.020 and a relative accuracy on the collision avoidance of 0.979.

Findings. The results from this case study are shown in Table 5.3. The first column of this table is the paired artifact and verifier, the next four columns are the number of checked properties that resulted in either proving (T) or falsifying (F) the property, returning unknown (U), or running out of resources (OOR). The next three columns show the mean verification time across all properties for each result type. We do not report verification times for OOR results.

While Reluplex can not be applied to the original DAVE2 artifact, and ERAN could not be applied to the original DroNet artifact, both verifiers could be applied to the refactored networks. In the case of DAVE2 and Reluplex, all 10 properties where shown to hold for the refactored network in less than a minute per property. In the case of DroNet and ERAN, the refactoring enabled the verification of all 20 properties, eight were deemed true and 12 deemed false for the refactored network, taking on the order of 10 minutes to complete verification of each property.

The most significant finding of this case study is that R4V enables the application of previously inapplicable verifiers.

5.3.3 Results: Verification Efficiency

This case study pairs the DAVE-2 and DroNet correctness problems with the Neurify and Planet verifiers, respectively. While these verifiers can run on these verification problems, they take a considerable amount of time. Using the original networks, Neurify takes an average of 5.6 hours to check a single DAVE-2 property, and Planet cannot verify any DroNet property within 24 hours.

Refactoring. To improve the efficiency of Neurify on the DAVE2 network, we apply R4V to refactor the network by applying the transformation $\mathit{forall}(\mathcal{N}, \mathit{isLayer}(\{2,3,4,7,9\}), \mathit{drop})$, where $\mathit{isLayer}(\mathcal{I})$ is a predicate which identifies whether an operation is part of the layer at position $i \in \mathcal{I}$, which matches the one manually generated by Wang et al. in the evaluation of Neurify [135]. To increase the efficiency of Planet on DroNet, we apply R4V to remove the first convolutional and maxpooling layers, as well as the last two residual blocks ($\mathit{forall}(\mathcal{N}, \mathit{isLayer}(\{0,2,3\}), \mathit{drop})$). We also scale the size of the input layer to a 10 by 10 grayscale image ($\mathit{forall}(\mathcal{N}, \mathit{isInput}, \mathit{scale}(\cdot, 0.05))$). This last step is necessary to enable Planet to verify any of the properties within the time limit.

Findings. The results from this case study are summarized in Table 5.4. The first column of this table is the artifact and verifier pair, the next four columns are the number of properties that were checked that resulted in either proving or falsifying the property, or returning unknown or running out of resources. The next three columns show the mean verification time for each kind of returned

Table 5.4: Results for the Verification Efficiency study.

	Result Count				Time (sec.)		
Artifact/Verifier	T	F	U	OOR	T	F	U
DAVE-2/Neurify	0	0	10	0	-	-	20318*
R4V(DAVE-2)/Neurify	10	0	0	0	2449	-	-
DroNet/Planet	0	0	0	20	-	-	-
R4V(DroNet)/Planet	1	2	0	17	1155	45423	-

result. We do not report times for OOR results.

While Neurify could mostly handle the original DAVE2 network, its results were flaky for some properties, alternating between segmentation errors and verification completions. Hence, the reported numbers on the original DAVE2 with Neurify correspond to the best of up to five tries. Even when Neurify completed its execution after an average of 339 minutes per property, the complexity of the original network introduced significant imprecision into the analysis, causing all 10 property checks to return an *unknown* result. In contrast, Neurify was able to prove all properties were true of the refactored network, taking about 41 minutes on average to verify a property - an 8 times speedup over results on the original.

Similarly, although Planet can accept the original DroNet architecture, it could not complete any of the property checks within one day. After applying R4V to refactor the network, Planet was able to finish checking three out of the 10 properties, taking just under 20 minutes to verify a property and a little over 16.5 hours to falsify a property.

The most significant finding of this case study is that R4V can speed up the verification time and enable verifiers to provide more useful results.

5.3.4 Results: Error-Verifiability Trade-Offs

In this case study, we explore the error-verifiability trade-offs using DAVE-2 and the Neurify verifier. We generate multiple refactored versions of the DAVE-2 network to explore the trade-offs.

Refactoring. We start our exploration of the refactoring space using the coarser granularity transformation *drop*, and performing a binary search to determine the layers to drop based on the refactored network error and verification time. Because it rarely makes sense to drop a convolution or fully-connected (e.g., GeMM) operation without also droping its corresponding ReLU operation we treat an operation followed by a ReLU as a single layer to be dropped. This results in 11 layers,

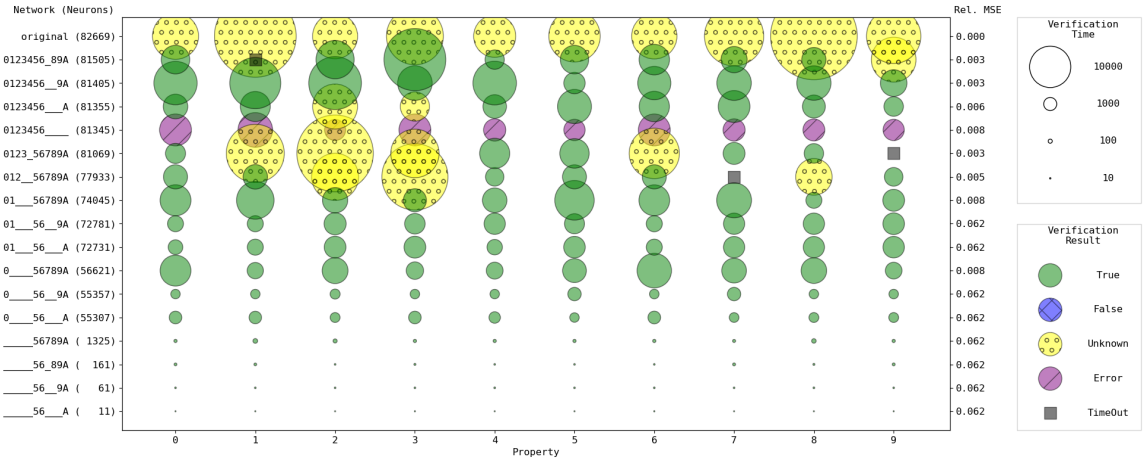


Figure 5.5: Results for the Error-Verifiability Trade-Offs study. The horizontal axis lists the 10 DAVE2 safety properties. The vertical axis lists 17 networks (original plus 16 refactorings). Each shape in the plot corresponds to a verification result, where the size indicates the time and the color and pattern indicate the verification result.

where 9 are droppable: 5 convolutions and their corresponding ReLUs, and 4 fully-connected operations with their corresponding ReLUs. The sixth and seventh layers correspond to reshaping operations and cannot be dropped. Each of the 11 layers is labelled with a hexidecimal digit in the range 0-A.

As stopping criteria for the search we set desired thresholds on the error and verifiability of the refactored network. We arbitrarily set the acceptable error in terms of the MSE relative to the original network to be under 0.01. In terms of verifiability, we require for all the properties to be verified as true or falsified in under three hours per property.

Findings. To provide a characterization of the space of trade-offs involved with refactoring we use R4V to refactor the original network into 16 networks. The results are depicted in Figure 5.5. Each row in this figure is a refactored network, and each column is a DAVE2 safety property. The network names and number of neurons appear in the first and second column along the left vertical axis. The networks are sorted by the number of neurons, so networks towards the bottom of the plot are less complex by this measure of complexity. The size of each circle represents the time to verify the corresponding network and property, and the color and pattern of the circle indicate the verification result. The relative MSE of each network appears on the right of the plot.

Starting from the top, we notice that the original network takes over 10,000 seconds to verify every single property, and for the second property as much as 39306 seconds, and in all cases it returns *unknown*. When R4V is applied to refactor the original network by removing the fully-

connected layer labeled as "7" resulting in `0123456_89A` (second row in the graph), we observe a 2.1x speedup in verification time and also more useful returned results (eight properties were shown to be true, one unknown, and one timed-out). The third row shows network `0123456__9A`, which has another fully-connected layer removed, resulting in a speedup of 3.6x in verification time and all properties checked as true. The following two rows of networks keep dropping fully-connected layers, and when all fully-connected layers are dropped we notice that Neurify returns an error as it cannot deal with the resulting architecture of `0123456____` which is correct but has no fully connected layers, which was unexpected by the verifier. The following rows start the removal of convolutional layers, resulting in another noticeable reduction in network complexity. Network `01___56789A` renders a speedup of 6.3x in verification time while retaining a MSE of 0.008 relative to the original network. That is also the case with `0____56789A`, with 9.4x speedup and still an acceptable error rate. The rows below have too large of an error to be deemed viable.

In total, 3 of the 16 refactored DAVE2 networks in Figure 5.5 meet the error-verifiability criteria, suggesting that the complexity sweet spot for DAVE2 contains a variety of acceptable networks.

The most significant finding of this case study is that, without prior domain knowledge, R4V can enable the effective exploration of the error and verifiability tradeoffs and converge towards a network that meets both criteria.

5.4 Conclusion

This chapter introduces automated support for refactoring neural networks to facilitate verification while retaining behavior of the original network. We introduce both non-semantics preserving refactorings (R4V), which transform the network to increase verifiability and retrain the network to achieve similar (but not necessarily equivalent) error to the original, and semantics preserving refactorings (DNNV), which transform the network to one with equivalent behavior and increased verifiability.

Chapter 6

Distribution Models for Verification

Joint work with fellow UVA PhD student Felipe Toledo.

A neural network, $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$, is trained to approximate a target function, $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X} \subseteq \mathbb{R}^n$ and $\mathcal{Y} \subseteq \mathbb{R}^m$. In many common use cases, such as for images, the domain of definition – referred to as the data distribution $\mathcal{X}$ – is an infinitesimal portion of the full domain, $\frac{|\mathcal{X}|}{|\mathbb{R}^n|} \approx 0$. This becomes more common as n grows large. Take for example, an instantiation of the pendulum system introduced in Chapter 1 in which the network takes in images of the pendulum in order to predict a torque, as depicted in Figure 6.1. One common image representation is a vector of values between 0 and 1 (i.e., $[0, 1]^n$), where each value in the vector corresponds to the intensity of one pixel in the image. However, the set of vectors that make up the space of images corresponding to pendulum images is vanishingly small within this space as the image size grows. A neural network is trained to operate as desired on this small space of pendulum images, but its behavior on inputs outside of this restricted subspace is undefined.

Unfortunately, validation and verification of neural networks often ignores the partial definition of a network with significant negative consequences. First, existing test generation techniques [90, 97, 120, 124, 145] have been shown to produce a majority of inputs that lie outside the data distribution [12, 31]. Second, white-box coverage criteria [84, 97] do not take the distribution into account which can drive coverage-directed test generators to give misleading reports of the coverage achieved [31]. Third, faults detected for inputs outside the distribution constitute false

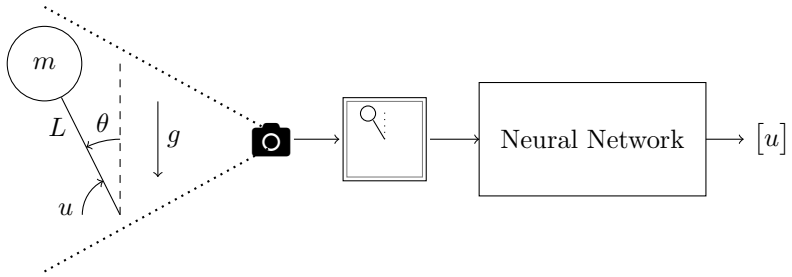


Figure 6.1: An example instantiation of the environment model and neural network for the pendulum system introduced in Chapter 1 in which the network takes in images of the pendulum to produce a control signal.

reports [12, 31] which can lead to wasted effort in fault triage, localization, and repair.

While recent work has explored leveraging models of the data distribution for testing neural networks [21, 31], we present the first approach to use such models to support verification and falsification. We draw inspiration from research exploiting environmental models of the feasible input domain for software systems to focus Verification and Validation (V&V). These models are typically built from the system requirements and can be expressed in a variety of forms, such as simulations [130], state-machines [26], or logical specifications [70]. Such *environment models* [126] have become an essential component of V&V approaches for software systems [22, 30, 39, 41, 76], which has led to their adoption in several domains [86].

To be amenable for V&V, environment models must satisfy three requirements. First, they must be *accurate* in defining the set of feasible inputs, i.e., they should not include too many inputs off the data distribution or exclude many inputs on the distribution. For example, for an under-approximating analysis, e.g., [76], an underapproximating model is required to guarantee feasible counter-examples; dually an overapproximating analysis requires an overapproximating environment model. Second, they must be *generative*, providing the ability to be executed, interpreted, or solved to generate feasible inputs. This can enable, for example, generating feasible counter-examples when verifiers or falsifiers detect property violations [94]. Third, for verification, they must be *amenable to constraint-based encoding* in a form that can be leveraged by the verification algorithm. For example, for a SMT-based verification method, e.g., [76], an environment model must be convertible to logical formulae in a supported theory. For abstract interpretation, e.g., [46], an environment model must be convertible to supported abstract domains.

In this chapter, we adapt the concept of an environment model to support existing verification and falsification techniques for neural networks. To do this, we must address the potential intractability of specifying an accurate model of the feasible inputs for a complex neural network – like those that

process images (such as the one in Figure 6.1). A key insight of this work is that we can leverage the rich body of research that the machine learning community has developed for learning generative models of a data distribution, which we exploit to define environment models. By defining an environment model with a generative model to map from states to network inputs, we can develop rewritings to incorporate this environment model into the property. This rewriting modifies the property such that the generative model is used as a prefix to the network under analysis, focusing V&V tools on the state space of the environment model rather than the network input space. Because the generative model only maps states to network inputs within the distribution, this method of environment modelling and rewriting focuses V&V on the data distribution.

Our *distribution-based falsification and verification* (DFV) approach transforms a correctness problem into a new problem focused on the data distribution by leveraging generative models for defining environments and introducing rewritings to enable V&V tools to support problems with this type of non-trivial environment model. DFV supports the reporting of feasible on-distribution counter-examples when property violations are detected, and that subsets of the data distribution are free of violations when verifiers are able to discharge such proofs. We evaluate DFV on networks for classifying images of clothing [144] and controlling a drone from image data [82], for a range of challenging property specifications. We find clear evidence that DFV enables existing V&V tools to produce counter-examples much more representative of the data distribution than are computed otherwise – both visually and in terms of quantitative similarity measures. While scaling of verification techniques is challenging, we find evidence that distribution models can enable them to prove properties over the data distribution. Our results can be used to guide the development of models to support DFV.

The primary contributions of this work are the:

1. Formulation of the first model-based verification and falsification method for neural networks.
2. Demonstration that distribution models yield substantially better counter-examples from verification and falsification.
3. Exploration of different models of the data distribution and their trade-offs.

6.1 Approach

Figure 6.2 depicts the general process taken by DFV. Given a correctness problem with an environment model in which the state-to-input mapping, $\mathcal{M}$ is specified as a neural network, such as

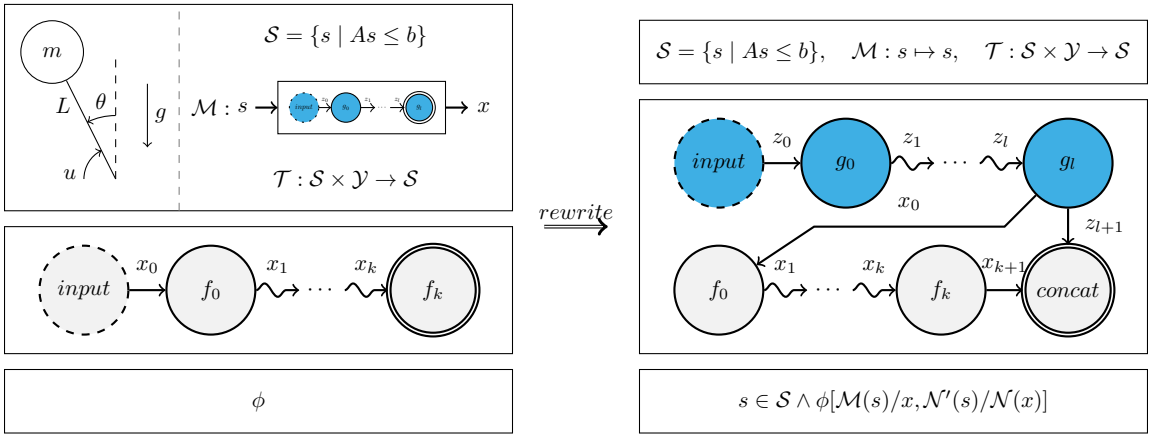


Figure 6.2: An overview of DFV.

a generative model, DFV rewrites the problem such that the new network composes $\mathcal{M}$ with the original network, and the new property adds a constraint over the inputs of $\mathcal{M}$ and rewrites constraints over the original network input to constraints over the output of $\mathcal{M}$. More formally, this environment rewriting takes the following form:

$$rewrite_{DFV}([\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle, \mathcal{N} : \mathbb{R}^n \rightarrow \mathbb{R}^m, \phi]) = \{[\langle \mathcal{S}, s \mapsto s, \mathcal{T} \rangle, \mathcal{N}', \phi']\} \tag{6.1}$$

A correctness problem is rewritten to a single correctness problem with equivalent semantics to the original problem, such that the state-to-input mapping of the environment model in the modified problem is the identity function, and the new network and property encode the semantics of the original state-to-input mapping. We discuss this rewriting in more detail below.

6.1.1 Rewriting

This rewriting relies on a few assumptions to enable transformation of the original problem. First, we assume that the state-to-input mapping function, $\mathcal{M}$, of the environment model is modelled as a neural network which maps states to network inputs on the data distribution. Second, we assume that the statespace, $\mathcal{S}$, is defined by a bounded polytope. We provide a more detailed analysis of these two assumptions and their implications in Section 6.1.2. Finally, similar to previous work [109], we assume that properties are defined in a subset of first-order logic in which variables are universally quantified and constraints are linear inequalities over network input and output values. This form facilitates property manipulation while maintaining expressiveness [108, 109].

Algorithm 5 presents a high-level overview of this rewriting. The $rewrite_{DFV}$ algorithm, which

Algorithm 5: $rewrite_{DFV}$

Input: A correctness problem,
$\psi = [\mathcal{E} = \langle \mathcal{S}, \mathcal{M} = \langle Q^{(\mathcal{M})}, q_0^{(\mathcal{M})}, q_*^{(\mathcal{M})}, \delta^{(\mathcal{M})} \rangle, \mathcal{T} \rangle, \mathcal{N} = \langle Q, q_0, q_*, \delta \rangle, \phi]$.

Output: A singleton containing a semantically equivalent correctness problem,
$\psi' = [\mathcal{E}', \mathcal{N}', \phi']$, with trivial environment model.

1 **begin**
2 $q_0' \leftarrow q_0^{\mathcal{M}}$
3 $q_*' \leftarrow Concat(x, y)$
4 $Q' \leftarrow Q \cup Q^{(\mathcal{M})} \cup \{q_*'\} \setminus \{q_0\}$
5 $\delta' \leftarrow \{(q_*^{(\mathcal{M})}, q_*', x), (q_*, q_*', y)\} \cup \delta^{(\mathcal{M})} \cup \{ \begin{cases} (q_*^{(\mathcal{M})}, op_2, \sigma) & \text{if } op_1 = q_0 \\ (op_1, op_2, \sigma) & \text{otherwise} \end{cases} \mid (op_1, op_2, \sigma) \in \delta \}$
6 $\mathcal{N}' \leftarrow \langle Q', q_0', q_*', \delta' \rangle$
// $\mathcal{N}'(s)_{\mathcal{N}}$ returns the output values associated with $\mathcal{N}$,
// while $\mathcal{N}'(s)_{\mathcal{M}}$ returns the output values associated with $\mathcal{M}$.
7 $\phi' \leftarrow (s \in \mathcal{S} \implies \phi[\mathcal{N}'(s)_{\mathcal{N}} / \mathcal{N}(x), \mathcal{N}'(s)_{\mathcal{M}} / x])$
8 $\mathcal{E}' \leftarrow \langle \mathcal{S}, s \mapsto s, \mathcal{T} \rangle$
9 $\psi' \leftarrow [\mathcal{E}', \mathcal{N}', \phi']$
10 **return** $\{\psi'\}$

we will refer to more generally as DFV, takes in a correctness problem, and outputs a semantically equivalent problem which incorporates the semantics of the original environment as part of the modified network and property. The output of this algorithm can then be processed by additional rewritings, such as the reduction introduced in Chapter 4 before being checked by existing V&V tools for neural networks.

The rewriting transformation occurs in four steps.

First, DFV constructs a new neural network (lines 2-6), which is the composition of the original network, $\mathcal{N}$ and the state-to-input mapping, $\mathcal{M}$. The output of $\mathcal{M}$ is connected to the inputs of $\mathcal{N}$ and also forwarded to the end of the network where they are concatenated with the outputs of $\mathcal{N}$. This concatenation enables constraints to be specified over both inputs and outputs. Without this step we could not specify constraints over network inputs, since they would now be in the middle of the composed network. This new network, $\mathcal{N}'$, is a function from environment states to network inputs and outputs. Because $\mathcal{M}$ is assumed to only produce inputs on the data distribution this has the effect of focusing $\mathcal{N}$ on inputs within this distribution since this construction forces $\mathcal{N}$ to only operate on outputs of $\mathcal{M}$. This composition can be simplified for problems with properties that do not constrain network inputs by omitting the input forwarding and output concatenation since those forwarded values would not be used in the final specification.

Second, DFV constructs a new property specification (line 7) by substituting the expression $\mathcal{N}'(s)_{\mathcal{N}}$ for usages of $\mathcal{N}(x)$ and $\mathcal{N}'(s)_{\mathcal{M}}$ for usages of x and using the updated specification as the

consequent to an implication with $s \in \mathcal{S}$ as the antecedent. The expression $\mathcal{N}'(s)_{\mathcal{N}}$ returns the subset of the output values that correspond to outputs of $\mathcal{N}$. If the output of $\mathcal{N}$ has size m, this corresponds to the rightmost m values of the output of $\mathcal{N}'$. Similarly, the expression $\mathcal{N}'(s)_{\mathcal{M}}$ returns the subset of the output values that correspond to outputs of $\mathcal{M}$. If the output of $\mathcal{M}$ (i.e., the input of $\mathcal{N}$) has size n, this corresponds to the leftmost n values of the output of $\mathcal{N}'$. After substitution, this new property captures the semantics of the original property specification in conjunction with the environment.

Third, we specify a new environment model (line 8) in which $\mathcal{M}$ is replaced by an identity mapping. This is necessary to preserve the semantics of the correctness problem, since the rewritten network now operates on environment states rather than over the original network input space. While this step may be able to be skipped if DFV is the last rewriting applied before verification and the target verifier does not take in an environment model, it ensures that $rewrite_{DFV}$ preserves the problem semantics for future rewritings.

Finally, we construct a correctness problem (line 9) from the modified network, property, and environment and return a single element set containing this problem to the user. This new correctness problem is semantically equivalent to the original, but uses an environment with an identity $\mathcal{M}$ and a network which operates only on inputs from the modelled distribution.

6.1.2 Environment Modelling

Algorithm 5 assumes both that the state-to-input mapping function, $\mathcal{M}$, of the environment model is modelled as a neural network which maps states to network inputs on the data distribution, and that the statespace, $\mathcal{S}$, is defined by a bounded polytope. However, for many domains, such as images, it is difficult to explicitly define such an $\mathcal{S}$ and $\mathcal{M}$. Fortunately, the field of machine learning has long understood the importance of modelling the data distribution and many techniques have been developed to generate unseen samples from the data distribution, generally known as *generative models*. There are three broad classes of generative models: variational autoencoders (VAE) [73], generative adversarial networks (GAN) [44], and autoregressive models such as PixelCNN++ [105]. Among these, VAEs and GANs can be classified as *latent variable* models since they make explicit the mathematical structure of the learned latent space which models the distribution. While they cannot produce a perfectly accurate characterization of the data distribution, we argue, and later demonstrate in Section 6.2.2, that generative models learn a close enough abstraction to obtain useful results, and the state-of-the-art techniques continue to produce more accurate models.

In this work, we advocate the use of a latent variable generative model, $\mathcal{G} : \mathbb{R}^d \to \mathbb{R}^n$, where

$d \ll n$, for defining the statespace and state-to-input mapping of an environment model, $\mathcal{E} = \langle \mathcal{S}, \mathcal{G}, \cdot \rangle$. The generative model maps from a vector of latent variable values to a network input. The key here is that $\mathcal{G}$ is designed to be a total function, i.e., every point in $\mathbb{R}^d$ is a valid input, meaning that $\mathcal{G}$ outputs only network inputs from the learned distribution. The input space of $\mathcal{G}$ can be thought of as a learned statespace where the model has learned some (often uninterpretable) state representation, and how to map from the learned representation to data from the desired distribution. Ideally we could model $\mathcal{E}$ with $\mathcal{S} = \mathbb{R}^d$, however our definition of environment models (in Section 3.2) requires the statespace be a bounded set[1]. Fortunately, the training of $\mathcal{G}$ generally enforces some structure on its input space, such as being multivariate Gaussian. This structure can enable the specification of meaningful bounds. For instance, in a d-dimensional Gaussian space, the L_2 ball of radius c contains all points within c standard deviations of the mean. Given such a space, c can be specified to contain an arbitrarily large portion of the data distribution, e.g., $c = 5$ covers 99.99994% of the distribution. Unfortunately, many existing V&V tools do not support the non-linear constraints necessary for defining an L_2 ball, so we formulate approximations using convex polytope, e.g., an L_∞ ball. This enables specification of the environment statespace as $\mathcal{S} = \{s \in \mathbb{R}^d \mid As \leq b\}$, where $As \leq b$ specifies the convex polytope bounding the desired subset of the input space.

The rewriting introduced in this chapter assumes that the environment model has been specified with a bounded state space and a neural network as the state-to-input mapping. While we advocate use of generative models, we leave modelling of the environment to the user due to the many degrees of freedom when training $\mathcal{G}$, and the specific needs of the user. For example, for simple VAEs, such as those used in Section 6.2.2, the accuracy of the learned model depends on many factors, including the architecture (e.g., number, type, and configuration of layers) and the training parameters (e.g., optimizer, batch size, and learning rate). Independent of the model and training process, the goal is for $\mathcal{G}$ to approximate the distribution. We note that the development of distribution models that have high precision and recall is an active area of ML research [3, 88], and that recent research has defined high-precision VAEs [29]. We also note, however, that models with lower levels of precision can still be quite valuable. As we show in Section 6.2, rather simple VAE models can yield much more meaningful counter-examples than those produced without using a distribution model. We leave to future work a broader study of how model accuracy impacts the cost and benefit of DFV.

[1] Additionally, most V&V tools require input spaces be bounded.

(a) FashionMNIST training images.

(b) DroNet training images.

Figure 6.3: Samples from FashionMNIST and DroNet training sets.

6.2 Evaluation

Here we assess the cost-effectiveness and scalability of DFV by applying it in conjunction with multiple falsifiers and verifiers. We conduct experiments to evaluate the following research questions:

1. How cost-effective is falsification and verification with DFV?
2. How does DFV scale with input domain complexity?

6.2.1 Design

In this section, we describe the correctness problem benchmarks, generative models, V&V tools, metrics, and computing resources that we use in the experiments of our study. We describe the experimental procedure and configuration of these components for each experiment under the results section for the corresponding research question.

Correctness Problem Benchmarks

GHPR-FMNIST is a new correctness problem benchmark introduced in this work, which is based on the GHPR-MNIST benchmark from the evaluation of DNNF [109]. The benchmark consists of 20 global reachability properties applied to a small FashionMNIST [144] network. A sample of images from the FashionMNIST training set is shown in Figure 6.3a. The network used is based on the architecture of the small MNIST network from the evaluation of the Neurify verifier [135]. There are 2 formulations of properties in this benchmark. The first 10 properties specify a relation

between output values conditioned on the input being classified as a certain class. We refer to these as conditional output relational (COR) properties, and they have the form: for all inputs, if class a has the maximal value, then the output values for classes a and b are closer to one another than the output values for classes a and c. Formally:

$$\forall x \in \mathcal{X} : (argmax(\mathcal{N}(x)) = a) \implies (|\mathcal{N}(x)_a - \mathcal{N}(x)_b| < |\mathcal{N}(x)_a - \mathcal{N}(x)_c|)$$

The intuition behind this property is that, if a model classifies the image as some class, then similar classes should be ranked higher than dissimilar classes. For example, one of the properties states that for all inputs, if the network predicts the class *sneaker*, then the class *sandal* will be ranked higher than class *shirt*. The other 10 properties are unconditional output relational (UOR) properties, and are weaker variations that drop the maximal value constraint. Formally:

$$\forall x \in \mathcal{X} : (|\mathcal{N}(x)_a - \mathcal{N}(x)_b| < |\mathcal{N}(x)_a - \mathcal{N}(x)_c|)$$

These properties capture some intuition of structure in the rankings, specifying that similar classes should always be ranked nearer to each other than to a given dissimilar class.

GHPR-DroNet, introduced in DNNF [109], consists of 10 global reachability properties applied to the DroNet network [82], which predicts a steering angle, $N(x)_{steer}$, and probability of collision, $\mathcal{N}(x)_{P_{coll}}$, for a quadrotor from 200 by 200 black and white images. A sample of images from the DroNet training set is shown in Figure 6.3b. The properties are of the form: for all inputs, if the probability of collision is between p_{min} and p_{max}, then the steering angle is within d degrees of 0. Formally:

$$\forall x \in \mathcal{X} : (p_{min} \leq \mathcal{N}(x)_{P_{coll}} \leq p_{max}) \implies (|N(x)_{steer}| < d^{\circ})$$

As p_{min} increases, so does d, capturing the intuition that if the probability of collision is low, then the quadrotor vehicle should not make sharp turns, encoding one notion of safety in the highway driving and flying context for which this network was trained. We specify 10 such properties where p_{min} ranges from 0.0 to 0.9 in steps of size 0.1, p_{max} is always 0.1 greater than p_{min}, and d is 100 times larger than p_{min}, except for $p_{min} = 0$ where we set $d = 5$ to allow some degree of steering.

Environment Models

We consider two types of latent variable generative models to learn the data distributions and model the environments in our study – VAEs and GANS. We selected these models because: 1) they

meet the requirements of the approach, 2) they are among the most popular unsupervised learning approaches to encode a data distribution, and 3) they work in different ways and provide different tradeoffs. Given the number of variables involved in our experiments, we chose VAEs for RQ1 and incorporated GANs for RQ3. We explored a total of 3 environment models, 1 to characterize the distribution of *GHPR-FMNIST* and 2 to characterize the distribution of *GHPR-DroNet*. Details for the configuration of those models is provided in the experimental procedures for each of the research questions.

Falsifiers and Verifiers

For the falsifiers, we use four common adversarial techniques included in the DNNF tool [109]. DNNF reduces correctness problems to adversarial robustness problems to allow them to be falsified by off-the-shelf adversarial attacks. We chose to use FGSM [45], Basic Iterative Method (BIM) [77], DeepFool [87], and Projected Gradient Descent (PGD) [85] as they were the top performing falsifiers in the DNNF study. We use the default parameters for each adversarial attack method, as used in that study.

For the verifiers, we use three tools included in DNNV [108]: Neurify [135], VeriNet [49], and nnenum [10]. These tools have performed well in recent benchmarks [81, 146], and, importantly, each of these verifiers is able to return counter-examples.

Metrics

For each run of the falsifiers and verifiers we report the number of counter-examples found and the time to find each counter-example. To judge the quality of counter-examples we compute the mean reconstruction similarity (MRS) which adapts encoder-stochastic reconstruction error (ESRE) to use the SSIM metric. ESRE is computed as the mean of $\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|$, where $\hat{\boldsymbol{x}}$ is the output of a VAE for an input $\boldsymbol{x}$, for many samples of $\boldsymbol{x} \in \mathcal{X}$ [132]. We adapt ESRE to use the structural similarity index measure (SSIM) [140] to assess the quality of generated image data. Given a reference VAE, $\mathcal{V}$, MRS computes for a given input, $\boldsymbol{x}$, the expected similarity for a set of reconstructed inputs, $MRS(\boldsymbol{x}, \mathcal{V}) = \frac{1}{N}\sum_{i=1}^{N} SSIM(\mathcal{V}(\boldsymbol{x}), \boldsymbol{x})$. In this work, we estimate the mean using a sample size, N, of 100 reconstructions.

For each problem benchmark we require a separate VAE model, independent of falsification and verification, for measuring the MRS. For FashionMNIST we trained a fully-connected VAE model, VAE_{MRS}, with a 100-dimensional latent space, and symmetric encoder and decoder, each with two hidden layers, one of 256 neurons and one of 512 neurons, and ReLU activations. The decoder

uses a Sigmoid activation so that output values are in the range 0 to 1. We chose to use a model significantly larger than those used for DFV for evaluating MRS under the assumption that a larger model would be able to better model the distribution and thus provide accurate MRS measures for all models tested. For DroNet we trained a convolutional VAE model, Conv-VAE_{DroNet}, with symmetric encoder and decoder, and a 512 dimensional latent space. The decoder consists of 8 blocks, each composed of a convolutional transpose operation followed by batch normalization and an ELU activation, except for the final block, which uses a Sigmoid activation so that output values are in the range 0 to 1. We chose to use this model as the baseline for MRS, since we expected a convolutional model to perform well on the complex image data of the DroNet benchmark.

Computing Resources

Experiments were run on Linux compute nodes with Intel Xeon Silver 4214 processors at 2.20 GHz and 512GB of memory.

6.2.2 Results: RQ1 - Efficacy

In this experiment we quantitatively and qualitatively assess the effectiveness of DFV and its costs when applied in conjunction with 4 falsifiers and 3 verifiers.

Experimental Procedure

We use the GHPR-FMNIST benchmark in this experiment. We run the verifiers and falsifiers on this benchmark, both with and without DFV. When using DFV, we use generative model VAE_{RQ1}, which was designed to be supported by all existing tools by constraining its size and type of activation functions. More specifically, we design VAE_{RQ1} with a single hidden layer of size 24 in the decoder, and instead of a sigmoid activation on the output, it uses a piecewise-linear approximation of the sigmoid function with ReLU activations, since, of the verifiers explored in this work, only VeriNet supported non-ReLU activation functions. The approximation used is $Sigmoid(x) \approx ReLU(-ReLU(-0.25 * x + 0.5) + 1)$. We run all tools 5 times on each problem to account for random noise and we record the number of problems that return a *violated* result, as well as the MRS of each counter-example found. Properties generated by DFV used a radius of 3 in the latent space. Each verification or falsification job was allowed to use a single processor core and had a time limit of 1 hour.

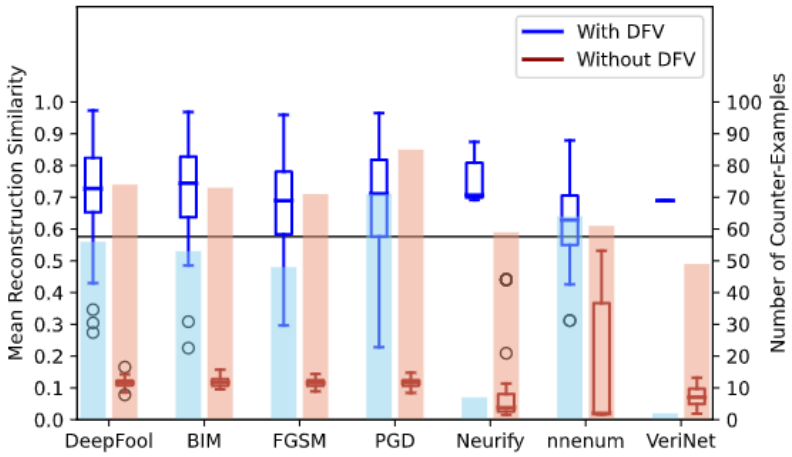


Figure 6.4: MRS for counter-examples across falsifiers and verifiers, both with (blue) and without (red) DFV. The solid horizontal line indicates the median MRS of the test set images. The shaded bars, measured on the right vertical axis, represent the number of counter-examples found.

Analysis and Findings

We first examine the MRS of counter-examples. Figure 6.4 shows box plots representing the distribution of the MRS of the counter-examples found by each of the 7 tools (x-axis) when applied to the original problems (red) and the problems using the decoder of VAE_{RQ1} as generated by DFV (blue). We find that, across all tools, the use of DFV results in counter-examples with a higher MRS than those found without DFV, i.e., counter-examples generated using DFV are reconstructed better by VAE_{MRS}. Indeed, the median MRS for the counter-examples found for the original benchmark problems is less than 0.1, while the median MRS when using DFV is greater than 0.6. This implies that counter-examples produced using DFV are closer to the distribution learned by VAE_{MRS} and thus they may be closer to the true input distribution. A statistical analysis of variance with the Kruskal-Wallis method[2] confirmed that the differences between using and not using DFV on any given tool are significant at $p = 0.05$.

Figure 6.4 also includes a horizontal line representing the median MRS of the FashionMNIST test set, providing another guideline to judge the quality of counter-examples. As shown in the figure, counter-examples found without DFV are well below the median MRS of the test data, indicating that they are reconstructed poorly, likely due to being far from the distribution. Counter-examples found with DFV tend to have an MRS higher than the median of the test set, indicating that they likely come from the distribution.

[2]We performed the non-parametric Kruskal-Wallis test given the different standard deviations observed across the distributions.

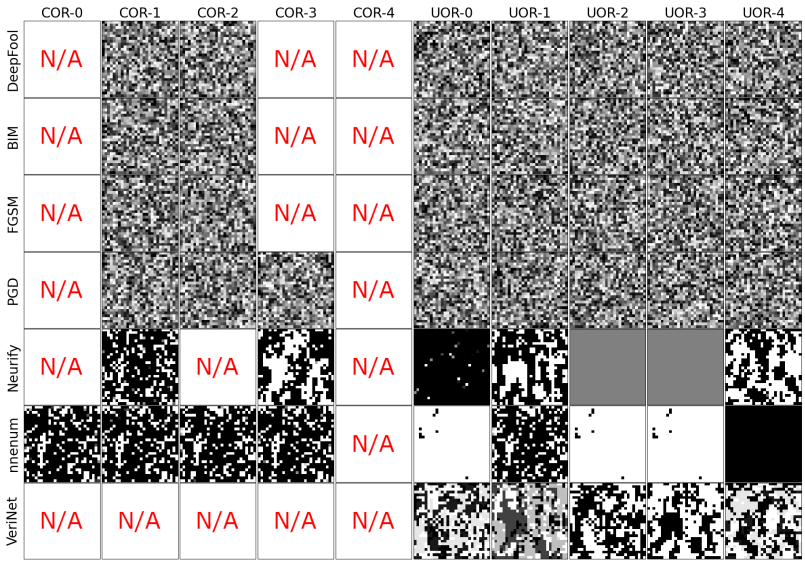


Figure 6.5: Counter-examples with highest MRS found for GHPR-FMNIST without DFV. Rows correspond to properties while columns correspond to tools.

The shaded bars in Figure 6.4, measured on the right vertical axis, show the number of counter-examples found. The data show that, as expected, the number of counter-examples found when using DFV decreases as irrelevant parts of the input space are pruned. For example, DeepFool found 74 counter-examples without DFV and 56 when applying DFV[3].

This portion of the study also revealed an interesting opportunity for verifiers. Based on the property design, we highly expected COR properties to be false and we expected all UOR properties to be false for the original correctness problem, which was indeed the case. However, when we used DFV to focus verification to a representation of the data distribution, the nnenum verifier was able to prove 25 problems true for the reduced input space encoded by the VAE_{RQ1}. This observation points to an opportunity for enabling verification to prove properties that may not hold over the whole input space but may hold over the relevant input space encoded by the training distribution. In order for such an approach to be effective, further studies are needed to guarantee that the generative model encodes a faithful model of the input distribution. We discuss this further in future work.

We now qualitatively examine the counter-examples generated with and without DFV. The tabulated images in Figure 6.5 are the counter-examples with the highest MRS produced without

[3]One exception to this trend was nnenum, which reported a floating point error for many properties without DFV but did not do the same with DFV. We conjecture that this is because DFV may be steering the the tool away from inputs that cause the failure.

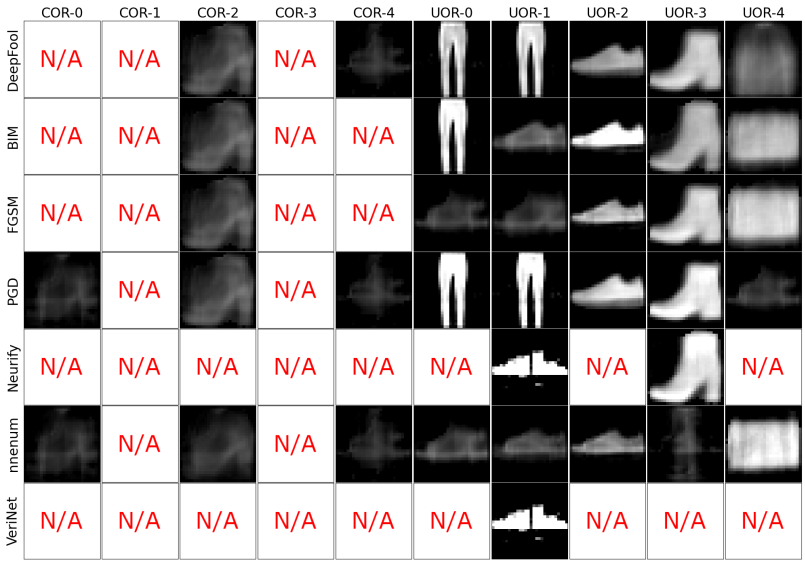


Figure 6.6: Counter-examples with highest MRS found for GHPR-FMNIST with DFV. Rows correspond to properties while columns correspond to tools. When applied with DFV, the counter-examples appear to be much better aligned with the training distribution.

DFV, and the ones in Figure 6.6 with DFV. Each row corresponds to one of the V&V tools used and each column a property. Without DFV (in Figure 6.5) the counter-examples generated by the falsifiers look like random noise, while the counter-examples generated by the verifiers have a bit more structure, with larger blocks of similarly valued pixels, but still have little discernible pattern. On the other hand, most counter-examples generated when using DFV (in Figure 6.6) bear some resemblance to the training images (e.g., boots, pants, sandals), and some of them are clearly identifiable. We argue that when no counter-examples are found for a given property property when using DFV, but they are found without DFV, like for Property A-1 and A-3, those counter-examples are likely to be spurious as they reside outside the data distribution. By the same token, when counter-examples are found with DFV but not found without DFV, like for Property A-4, we argue that DFV enables tools to explore the pruned space more extensively, enabling their generation.

Last, we briefly examine the time distribution for each tool to generate the counter-examples. Figure 6.7 presents box plots for each of the tools, and we again plot the number of counter-examples on the y2-axis. As expected, falsifiers are faster than verifiers. Looking at the upper quartiles of the times spent by the different tools, we can see that all falsifiers took under 1.5 seconds, while the verifiers took up to 1444.5 seconds. PGD detected the most counter-examples, 85 without DFV and 71 with DFV, while its median execution time was just over a second. When comparing the boxes within a tool, we find that incorporating DFV did not have a major impact on the time taken by

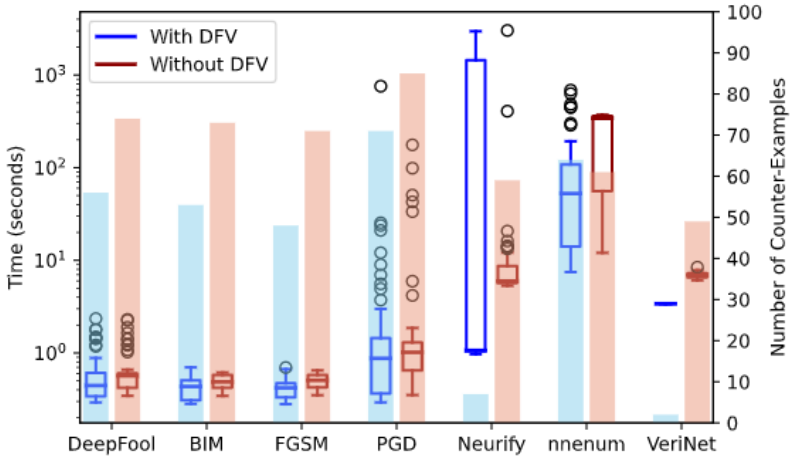


Figure 6.7: Times to find counter-examples by each tool. Blue box plots represent the times when using DFV, while red box plots represent the times without DFV. The shaded columns, measured on the y2-axis, represent the number of counter-examples found.

any of the tools[4].

Major Findings: Tools applied in conjunction with DFV generate fewer counter-examples, with four times higher MRS, negligible time penalty, and more visual similarity to the training distribution.

6.2.3 Results: RQ2 - Scalability

In this experiment, we assess the scalability of DFV by applying it to a large neural network for autonomous UAV control using 2 different input distribution models.

Experimental Procedure.

We use the larger and more complex GHPR-DroNet benchmark and apply the PGD falsifier, both as is, and using DFV with both a VAE and a GAN as the generative models. We train a fully-connected VAE, FC-VAE_{DroNet} with a symmetric encoder and decoder. The decoder of FC-VAE_{DroNet} has 6 hidden layers in the decoder with sizes 512, 512, 512, 512, 1024, and 2048, all with ELU activations, except the final layer which uses a Sigmoid activation. For GAN_{DroNet} we train a DCGAN [101] model on the DroNet dataset [82] with a Sigmoid on the final layer. Both models use a 512 dimensional latent space. As before, we run each falsifier 5 times to account for random noise and we record the number of counter-examples found and the time to find each counter-example. Each job

[4]The larger variation for Neurify can be attributed to the smaller number of counter-examples it generated.

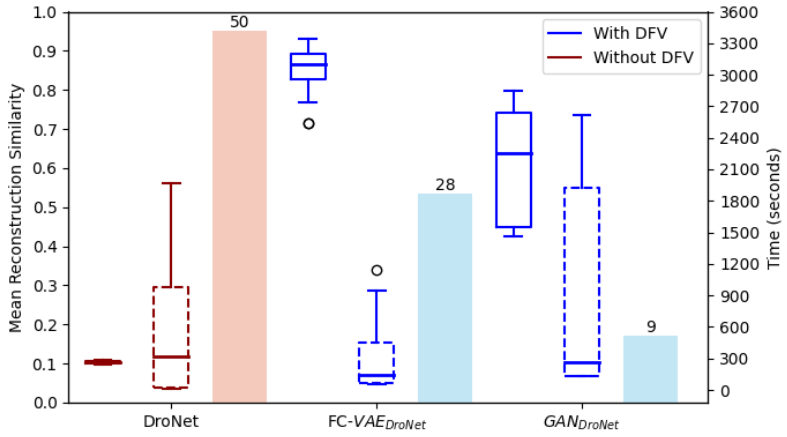


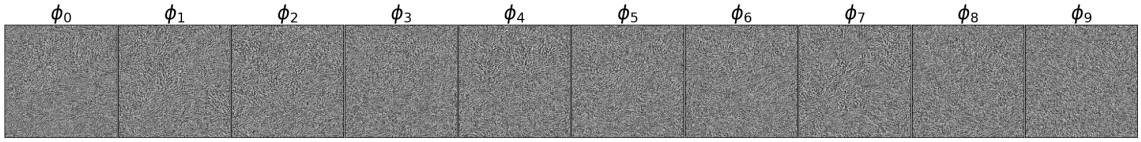
Figure 6.8: The mean reconstruction similarity (solid box), time (dashed box), and numbers of counter-examples (color bar) to the DroNet properties.

had a timeout of 1 hour.

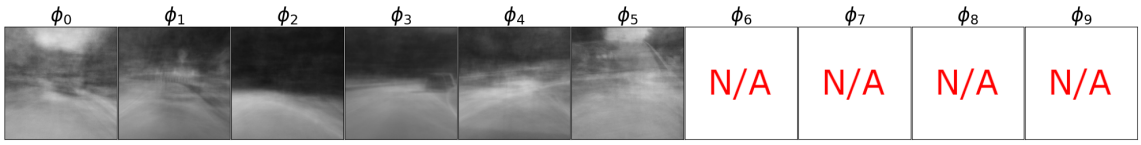
Analysis and Findings.

Figure 6.8 shows box plots with solid outlines for the distributions of the reconstruction similarities of counter-examples found using PGD on the DroNet DNN without DFV, as well as using DFV with the decoder of FC-VAE_{DroNet} and the generator of GAN_{DroNet}. Figure 6.8 also shows the number of counter-examples found using each model using bars with the count labeled above each bar. We find that, for DFV with both models, while fewer counter-examples are found, they clearly have higher reconstruction similarities than those found using the DroNet model alone. Indeed, the MRS differences between DroNet, FC-VAE_{DroNet}, GAN_{DroNet} are shown to be statistically significant overall by a Kruskal-Wallis test with p=0.05, and so do their pairwise differences. Corroborating the previous findings, this implies that the counter-examples found using DFV are closer to the distribution learned by Conv-VAE_{DroNet}, the model used to compute the MRS values, and thus may be closer to the actual input distribution. Without DFV, violations were found for all 10 properties across all 5 seeds. Using FC-VAE, 28 violations were found for 6 properties. Using GAN, 9 violations were found across 2 properties. While the lowest MRS for a counter-example found using DFV was 0.42, the MRS without DFV never exceeded 0.11.

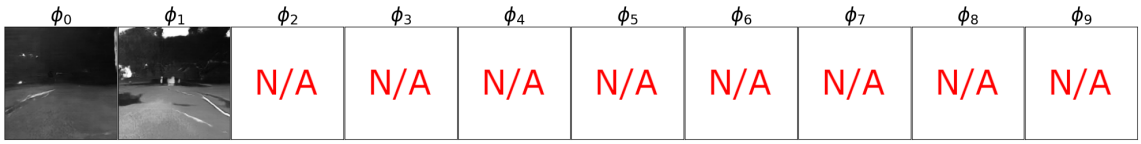
We now proceed to visually examine the counter-examples generated with and without DFV for 5 properties. Figure 6.9 shows the counter-examples with highest MRS generated by PGD. All generated counter-examples are provided in Appendix F. The images generated without DFV look



(a) Without DFV.



(b) With DFV, using FC-VAE_{DroNet}



(c) With DFV, using GAN_{DroNet}

Figure 6.9: Counter-examples to DroNet properties with 3 distinct input models. The counter-examples shown are those with the highest MRS across 5 runs of the falsifier on each of the 10 properties. When applied in conjunction with DFV, whether using a VAE or a GAN, the generated counter-examples visually appear to be much better aligned with the training distribution.

like random noise, while the images generated with DFV, independent of the chosen model, have structure and contain features seen in the training images such as roads, trees, or horizon lines. The model used for DFV has an impact on the images produced. While the VAE model tended to produce blurrier counter-examples, the GAN model produced counter-examples with sharper lines, but fewer recognizable road features.

Finally, Figure 6.8 shows box plots, with dashed outlines, of the time to generate each counter-example using each model. The median time to falsify DroNet alone was 321 seconds, while DFV with FC-VAE_{DroNet} took 146 seconds and DFV with GAN_{DroNet} took 259 seconds, but there is enough performance variance that those differences are not deemed statistically significant.

Major Findings: DFV can be applied with various models without significant time penalty, while also producing counter-examples with up to a nine-fold improvement in reconstruction similarity.

6.3 Threats to Validity

External Validity. Three threats to the generalization of our findings are our choice of tools, benchmarks, and generative models to evaluate DFV. We mitigate the concern about tools generality by selecting multiple falsifiers and verifiers as part of the first research question. For the next question we traded generality across tools for more insights about the performance of DFV under different models, which implied that we had to drop DNN verifiers from the rest of the assessment because they did not scale to the networks and models we were targeting. Regarding benchmarks, we selected ones from different domains, one a classification task, and the other being a regression task, with very different architectures and training data. Still, more benchmarks are needed to more broadly explore the cost-effectiveness of DFV. To mitigate the threat about model selection, we explored the use of both VAEs and GANs. Still, the examination of more generative models and their parameterization is part of the future work.

Construct Validity. Our choice of MRS as a quality measure and our personal qualitative judgment of generated counter-examples pose a threat in that the relevance of a counter-example could be judged by many means. We mitigated this threat by basing MRS on a popular measure, ESRE, and specializing it to images with SSIM. We also provide results using ESRE in the appendix, and we will explore additional measures including those for outlier detection in the future and perform studies with users to help us judge the counter-examples quality.

Internal Validity. Our training processes for the networks and the models constitute a threat to the internal validity of the study as their correctness could have affect the findings. We have documented and programmed those processes when possible through scripts to facilitate their reproduction. We also mitigate this threat by making our data and scripts for running our experiments and analyzing our results publicly available. Another threat to validity is the randomness involved in training of networks and models, and in the tools' performance. We mitigated that threat by running falsifiers multiple times and showing their variability.

6.4 Conclusion

This chapter introduces DFV, an approach enabling existing verification and falsification techniques for neural networks to target the data distribution. DFV composes learned latent variable generative distribution models with the network under analysis, rewriting the problem so that generated counter-examples are on the data distribution. We explore different data distribution models and find that using even simple models yield substantially better counter-examples across a range of verification and falsification techniques for two different benchmarks. These findings along with recent work on distribution-aware testing [21, 31], suggest that models of the data distribution can play an important role in V&V of neural networks. In future work, we plan to explore how performance metrics for latent variable generative models that assess their precision and recall [3], can guide the development of distribution models customized to best suit different V&V activities for neural networks.

Chapter 7

Enabling Verification of Closed-Loop Systems with Neural Network Components

Neural networks are increasingly used in systems that interact with the environment in which they are used, such as autonomous ground [28, 78, 98] and aerial [63, 67, 82] vehicles. These systems often have a high cost of failure and it is essential to assure their safety and correctness. Several tools have been introduced to verify behavioral properties of neural networks as they interact with their environment, often referred to as verifiers for closed-loop systems with neural network controllers [2, 14, 32, 36, 59, 107, 111, 112, 129]. However, these tools often struggle to provide a result as the reachability analyses they tend to favor lead to overapproximations that can neither prove the property nor identify a counter-example. For example, as we descibe in our evaluation, in a friendly competition consisting of a total of 6 participants across 2 years, among 8 benchmark problems consisting of closed-loop systems, no tool was able to return a result for more than 3 benchmarks without significantly altering the problem definition [60, 61].

Complementary to verification of closed-loop systems components is that of verification of input/output properties of networks, which aims to prove properties about network behavior at a single instance of time, such as adversarial robustness. Such properties are referred to as open-loop, since they do not consider the feedback from network to environment. Many tools with diverse algorithmic approaches for verifying open-loop properties of neural networks have recently been

introduced [8, 16–18, 20, 33, 34, 40, 50, 55, 68, 69, 96, 102, 104, 114–117, 125, 133, 135, 136, 141–143, 151]. Unfortunately these tools are not designed to verify properties of a closed-loop system, as they have no model of how the neural network interacts with the environment over time.

Recent work has shown a general set of open-loop properties that use polytope constraints can be rewritten to adversarial robustness properties, enabling verification tools to support a much broader set of properties than originally designed [109]. We build on this insight to develop three new rewritings for closed-loop correctness problems consisting of a discrete-time environment model, behavior specification, and neural network, which transform the closed-loop problem into a set of sub-problems which can be solved by existing open-loop verification tools. The result of verifying these open-loop sub-problems provides a solution to the original closed-loop problem, enabling open-loop verifiers to be used to solve closed-loop problems. We show that this approach enables tool users to obtain twice as many verification results compared to using existing closed-loop verifiers alone.

We assume we are given a discrete-time model of the environment with a function that provides the next state given the current state and network output, a property specified in linear temporal logic over finite-length paths, and a neural network. We introduce rewritings that: (a) transform the environment transition function into a neural network that overapproximates the true next state with high probability, (b) unroll the property specification for a bounded number of time steps based on the temporal logic specification, and (c) transform the unrolled networks into sequential multi-layer perceptrons which are supported by existing verifiers.

Our work is conceptually similar to 2 other closed-loop verification approaches. OVERT [112] first approximates non-linear environment dynamics with piecewise-linear bounds, and formulates the problem as piecewise linear relations which can be solved by mixed-integer programming. Our approach differs in our treatment of the environment model and neural network, both of which we treat as black boxes in contrast to the white box approach taken by OVERT. This means we are not limited to environment models with defined bounding approximations and that we can make use of off-the-shelf verifiers without modifying them for our approach. VenMAS [2] trains neural networks to approximate non-linear functions in the environment model, however they do not account for model error, which can lead to unsound results. Our approach rewrites the whole environment model, rather than individual non-linear functions, and accounts for the error of the trained model.

The contributions of this work are:

1. A new rewriting approach which can transform closed-loop correctness problems to a set of open-loop sub-problems.

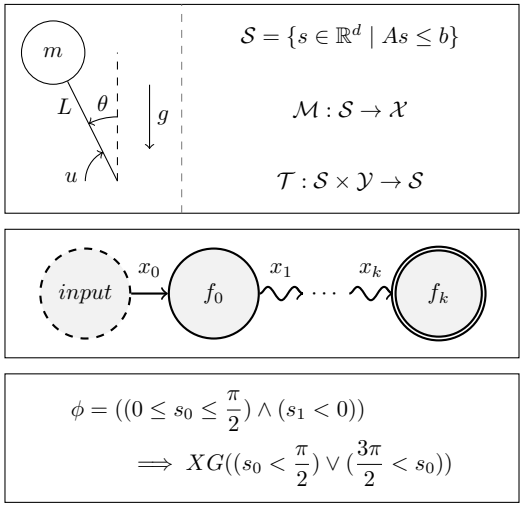


Figure 7.1: An example closed-loop problem specification for the inverted pendulum system.

2. An implementation of these rewritings in a publicly available tool.

3. A demonstration the effectiveness of rewriting by applying it to several closed-loop correctness problems and show that open-loop verifiers, after problem rewriting, doubles the number of results obtained compared to state-of-the-art closed-loop verifiers.

7.1 Approach

To enable the application of open-loop verifiers to closed-loop verification problems, we must develop several new rewritings which take into account a model of the environment in which, and on which, a neural network operates. Take for example, the verification problem introduced in Figure 3.1 and reproduced in Figure 7.1. This problem defines a property in a temporal logic, which incorporates knowledge of how the environment changes over time. This knowledge is encoded in the given environment model. To check this problem with open-loop tools, we must rewrite the problem to one with an open-loop property and a trivial environment model. We can do this through the use of several new rewritings.

7.1.1 Assumptions

We assume that we are given an environment model, with explicitly defined state-to-input and transition functions, and a bounded state space, as described in Section 3.2. Additionally, we assume a discrete-time model. This is not the case for many problems, including most of the benchmarks in

$$\lambda_t(x) \to \mathcal{M}(\lambda_t(s)) \tag{7.1}$$
$$\lambda_t(y) \to \mathcal{N}(\lambda_t(x)) \tag{7.2}$$
$$\lambda_t(s) \to \begin{cases} s & \text{if } t = 0 \\ \mathcal{T}(\lambda_{t-1}(s), \lambda_{t-1}(y)) & \text{otherwise} \end{cases} \tag{7.3}$$
$$\lambda_t(\neg\varphi) \to \neg\lambda_t(\varphi) \tag{7.4}$$
$$\lambda_t(\varphi_1 \vee \varphi_2) \to \lambda_t(\varphi_1) \vee \lambda_t(\varphi_2) \tag{7.5}$$
$$\lambda_t(X\varphi) \to \begin{cases} \bot & \text{if } t = \tau \\ \lambda_{t+1}(\varphi) & \text{otherwise} \end{cases} \tag{7.6}$$
$$\lambda_t(\varphi_1 U_\tau \varphi_2) \to \begin{cases} \lambda_\tau(\varphi_2) & \text{if } t = \tau \\ \lambda_t(\varphi_2) \vee (\lambda_t(\varphi_1) \wedge \lambda_{t+1}(\varphi_1 U_\tau \varphi_2)) & \text{otherwise} \end{cases} \tag{7.7}$$
$$\lambda_t(\phi) \to \phi[s/\lambda_t(s), x/\lambda_t(x), y/\lambda_t(y)] \tag{7.8}$$

Figure 7.2: Property rewriting rules.

our study (Section 7.2). However, a continuous-time system can be discretized to obtain a discrete-time model that approximates the continuous one. Unfortunately, continuous and discrete models are fundamentally different, and results do not directly transfer from one to the other. However, a violation for the discrete time problem can be checked against the continuous time specification, and proving a property holds for a discrete time model can provide some confidence in the correctness of the continuous time model, although not a guarantee. In addition to having a discrete-time model, we assume that property is specified in the time bounded linear temporal logic described in Section 2.2.2.

7.1.2 Property Rewritings

We first define $rewrite_\varphi : \Psi \mapsto \Psi$, which takes in correctness problems where $\mathcal{E}$ is specified such that $\mathcal{M}$ and $\mathcal{T}$ are represented as neural networks (justified in Section 7.1.3) and φ is specified in the restricted LTL defined in Section 2.2.2, and returns a new problem with an equivalent property, free of temporal operators.

Given ψ with $\mathcal{E}$ and φ satisfying those assumptions, and a maximum time bound τ determined from φ, we can rewrite the problem to one in which the semantics of $\mathcal{E}$ are included in the network $\mathcal{N}$ by unrolling the property through time. We do this through the property rewriting rules in Figure 7.2. We introduce a rewriting function $\lambda : \Phi \times \mathbb{N} \to \Phi$, which takes in a formula and a parameter indicating the time step and recursively rewrites sub-expressions, effectively unrolling the

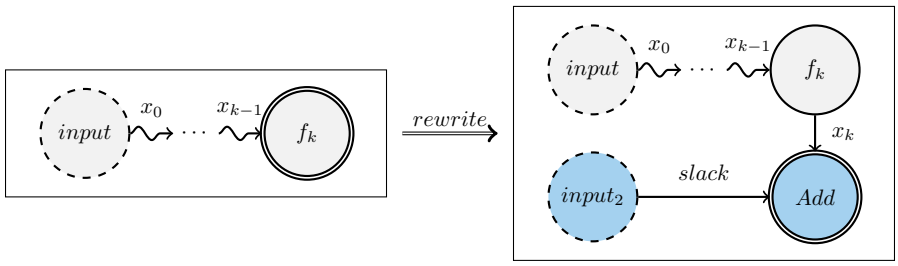


Figure 7.3: After training a network approximation for $\mathcal{M}$ or $\mathcal{T}$, $rewrite_{\mathcal{E}}$ modifies the trained network with a new input that is added added element-wise with the original output.

temporal property up to a user specified bound τ. We list rewritings for the basic LTL operators. Additional operators such as F, G, and $\wedge$ can be supported by first rewriting them using the basic operators. This rewriting process is semantics preserving and results in a new property formula free of temporal operators.

7.1.3 Environment Rewritings

Next, we define $rewrite_{\mathcal{E}} : \Psi \mapsto \Psi$, which takes in a correctness property and rewrites it such that the environment is specified such that $\mathcal{M}$ and $\mathcal{T}$ are represented as neural networks, as required by $rewrite_{\varphi}$. This rewriting simply trains a neural network for any non-linear $\mathcal{M}$ and $\mathcal{T}$ functions. However, because environment models may be quite complex, the trained network may be a poor approximation of the original function, so the rewriting also modifies these trained networks such that, with high probability, they overapproximate the original environment.

This rewriting takes in the environment model, and trains a neural network for any non-piecewise-linear $\mathcal{M}$ or $\mathcal{T}$. Any linear $\mathcal{M}$ or $\mathcal{T}$ is exactly encoded as a neural network with a single GeMM operation. The rewriting generates training data by sampling inputs from the domain of the target function, i.e., $\mathcal{M}$ or $\mathcal{T}$, and computing the ground-truth outputs using that function. This results in a neural network that approximates the target function, but is not exact. For example, if the target function is the dashed black line in Figure 7.4, one approximation could be the solid red line also shown in that figure. This approximation can output values lesser or greater than the target value and these errors can compound over time causing verification results for the approximation to be unsound for the original model.

To achieve soundness, we must ensure that the network overapproximates the ground truth function. To do so, we estimate the error for outputs of the trained model by sampling inputs and computing error bounds for the outputs. With the computed error bounds, we modify the trained network with an additional input vector whose values are constrained to be in the computed

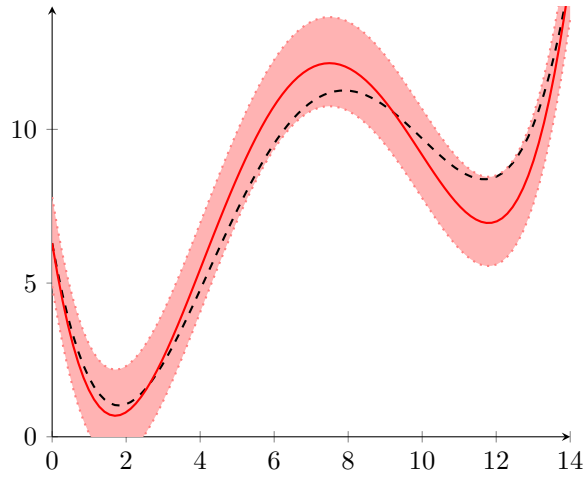


Figure 7.4: An example of added slack (shown in red shaded area) allowing the approximation (solid red line) to overapproximate the target function (dashed black line).

error intervals, as depicted in Figure 7.3. We term this error vector the *slack*. The network with added slack can output any value in the pink shaded area in Figure 7.4. A verifier can select any value within the error bounds for the slack, ensuring that the true next state is reachable by the trained model. However, it also overapproximates the reachable states, increasing the chances for spurious counter-examples. A neural network that better approximates the ground truth will have a lower error, thus a smaller overapproximation, and thus a lower chance of producing spurious counter-examples.

7.1.4 Network Rewritings

Finally, we define $rewrite_{\mathcal{N}} : \Psi \mapsto \Psi$, which takes in a problem where $\mathcal{N}$ is assumed to be piecewise-linear, a common assumption for neural network verifiers, and rewrites it such that the network is a single sequential path of operations, a requirement for most existing open-loop verifiers. This is necessary because $rewrite_{\varphi}$ introduces complex structure into the neural network. In particular, the network constructed by property rewriting rule (7.3) has 2 parallel computation paths, one to get the previous state, and one to get the control input. This branching structure can be repeated many times based on the bounds in the temporal operators, leading to large complex network structures.

To remove the complex structure, $rewrite_{\mathcal{N}}$, transforms the network into a multilayer perceptron with ReLU operations using the rewriting rules in Figure 7.5. Starting from the last network operation we work backwards, combining parallel paths into a single path, until we reach the beginning of the network. The result of this network rewriting is a sequential neural network with alternating

$$z = A(x,y) \rightarrow [z' = C(x,y), \quad z = G(z', \begin{bmatrix} I_n \\ I_n \end{bmatrix}, 0)] \tag{7.9}$$

$$z = C(x,x) \rightarrow [z = G(x, \begin{bmatrix} I_n & I_n \end{bmatrix}, 0)] \tag{7.10}$$

$$\begin{aligned} z = C(x,y) \rightarrow [&z' = C(x',y'), \\ &z = R(z')] \\ \text{where } x = R(x') & \\ \text{and } y = R(y') & \end{aligned} \tag{7.11}$$

$$\begin{aligned} z = C(x,y) \rightarrow [&z' = C(x',y'), \\ \text{where } x = G(x', A_1, b_1) \quad &z = G(z', \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}, \begin{bmatrix} b_1 \\ b_2 \end{bmatrix})] \\ \text{and } y = G(y', A_2, b_2) & \end{aligned} \tag{7.12}$$

$$\begin{aligned} z = C(x,y) \rightarrow [&z' = C(x',y), \\ \text{where } x = G(x', A, b) \quad &z = G(z', \begin{bmatrix} A & 0 \\ 0 & I_n \end{bmatrix}, \begin{bmatrix} b \\ 0 \end{bmatrix})] \end{aligned} \tag{7.13}$$

$$\begin{aligned} z = C(x,y) \rightarrow [&z_x = G(x, I_n, 0), \\ \text{where } x = R(x') \quad &z''_y = G(y, \begin{bmatrix} I_n & -I_n \end{bmatrix}, 0), \\ &z'_y = R(z''_y), \\ &z_y = G(z'_y, \begin{bmatrix} I_n \\ -I_n \end{bmatrix}, 0), \\ &z = C(z_x, z_y)] \end{aligned} \tag{7.14}$$

Figure 7.5: Network rewriting rules using A (Add), C (Concat), G (GeMM), and R (ReLU). 7.13 and 7.14 have symmetric rewritings when the order of Concat inputs is swapped.

GeMM and ReLU operations, and with behavior equivalent to the original network, i.e., the rewriting is semantics preserving. This network architecture is supported by all known open-loop verifiers for neural networks.

7.1.5 Rewriting Correctness Problems

With these three rewritings we can transform correctness problems with closed-loop properties to ones with open-loop properties, enabling application of open-loop verifiers. We define $rewrite_{cl}(\psi) = \{rewrite_{\mathcal{N}}(\psi') \mid \psi' \in reduce(rewrite_{\varphi}(rewrite_{\mathcal{E}}(\psi)))\}$, where $reduce : \Psi \mapsto P(\Psi)$ reduces the property to one supported by the desired verifier, $\mathcal{V}$, as described in Chapter 4. This rewriting produces a set of correctness problems such that for all $\psi' \in rewrite_{cl}(\psi)$, ψ' is supported by $\mathcal{V}$, and $(\bigwedge_{\psi' \in rewrite_{cl}(\psi)} \mathcal{V}(\psi') = holds) \Rightarrow (\mathcal{E}, \mathcal{N} \models \phi)$, assuming that the rewritten environment overapproximates the true environment model.

7.1.6 Implementation

We have implemented DNNV+LTL as extensions to DNNV, a tool for rewriting open-loop neural network verification problems to equivalid sets of problems which can be solved by several backend verifiers (discussed in Chapters 4 and 5).

We extended the DNNP property specification language used by DNNV to support LTL operators, and we added functions for constructing neural networks from a set of linear functions to allow explicit specification of linear environment models. We added rewritings from LTL expressions to first order logic formula using the rewriting rules in Figure 7.2, and rewritings from piecewise-linear operation graphs to multilayer perceptrons with ReLUs using the rules in Figure 7.5.

We implemented the environment rewriting as a separate tool, which trains neural networks given an environment model. We chose to implement this rewriting separately since DNNV contains only equivalid rewritings, while this environment rewriting does not completely preserve the original problem semantics. This tool allows users to specify hyperparameters for training, including the network architecture, training batch size and learning rate. Additionally, training requires the ability to randomly sample from the input space of the target function, and we assume such a sampling procedure is provided.

All tools were written in Python 3.7+, and will be publicly available at `https://github.com/dlshriver/dnnv`[1].

7.2 Study

To evaluate our approach to enabling verification of closed-loop properties of neural networks by off-the-shelf open-loop verifiers, we will explore the following research questions:

1. How effective is our approach at enabling verification of stateful properties by existing stateless verifiers?

2. How does the neural network configuration of the rewritten environment model affect verification?

7.2.1 Study Design

We perform 2 experiments, one for each research question. The details of each experiment are provided in this section.

[1] The environment rewriting tool is provided in the `tools/` directory.

Benchmarks

We will evaluate our approach across the 8 problems included in the 2020 and 2021 ARCH-COMP AINNCS competitions [60]. This is a friendly, open competition for verifiers of closed-loop systems with neural network controllers. The systems have well defined environment models and neural networks on which to study our approach. Detailed descriptions of the benchmarks are provided in Appendix G. The competition consisted of 8 problems, 7 with continuous-time environments, and 1 discrete-time model (VCAS), each with 2 to 12 state values. The neural network controllers vary in size from 1 to 5 hidden layers and 50 to 500 total hidden units. All networks use ReLU activations. The second experiment explores the effect of the network architecture of the rewritten environment transition function using several modified versions of one of the more challenging AINNCS benchmarks.

Rewriting Parameters.

For the environment rewriting, the state-to-input mappings are linear for all benchmarks so we need only rewrite the transition functions. For each benchmark in the first experiment we train a linear model to approximate the transition function. We explore the effect of different model architectures in experiment 2, where we train 4 additional models for the SB9 benchmark with single hidden layers of size 64, 128, 256, and 512. We trained each model for $100,000$ iterations with batches of 100 randomly generated input samples per iteration. We used the AdamW optimizer [83] with an exponentially decaying learning rate starting from 0.001 and ending at 0.000001. Models were evaluated using a randomly generated set of $50,000,000$ sample points. Training times are reported in Appendix H.1.1, but we summarize them here. For experiment 1, the mean training time across the 8 benchmarks was 721.0 seconds, with a standard deviation of 1116.33[2]. For experiment 2, the mean training time across the 5 models was 294.4 seconds, with a standard deviation of 6.9. We then perform property and network rewriting and apply 4 verifiers for open-loop properties of neural networks to the rewritten problems.

Verifiers

We evaluate DNNV+LTL using 4 open-loop verifiers: ERAN [116], Marabou [69], nnenum [8], and Neurify [135]. These were chosen because they are supported by DNNV, upon which we implemented our approach, and represent a diverse set of algorithmic approaches. We will provide the reported

[2]The AP benchmark was an outlier at 3247.0 seconds. Removing this model, the mean drops to 300 seconds with a standard deviation of 81.33.

results from the closed-loop verifiers that competed in ARCH-COMP AINNCS as a baseline. These are JuliaReach [14], NNV [129], Verisig [59], ReachNN* [36], OVERT [112], and VenMAS [2]. We assume the reported results represent a best case performance of the tools given the tuning performed by the authors for each problem in the competition. Similar to our approach, OVERT and VenMAS support discrete time systems, while the other 4 tools support continuous time systems.

Metrics

For each verification problem in each benchmark we will record the verification result from each verifier, as well as the verification time. Each problem will also have several environment rewritings, for each of which we will report the error bounds on the outputs of the approximated environment models.

Computing Resources

Environment rewriting was performed on a Linux machine with an AMD Ryzen Threadripper 2970WX 24-Core Processor, 128 GB of memory, and 2 NVIDIA GeForce RTX 2080 Ti graphics cards. All other rewritings and verification used Linux compute nodes with Intel Xeon Silver 4214 processers at 2.20 GHz and 512GB of memory. Verifiers were allowed to use 48 logical processor cores, 64 GB of memory, and up to 30 minutes.

7.3 Results

We present the results of our 2 experiments in this section.

7.3.1 RQ1: Effectiveness of Rewriting

We evaluate the effectiveness of rewriting on the ARCH-COMP AINNCS benchmarks by performing rewriting, running the 4 open-loop verifiers on the rewritten problems, and recording the verification results. We compare the verification results to those reported by the AINNCS competitors.

The results are shown in Table 7.1, along with the results reported by the participants in ARCH-COMP AINNCS. The top section of the table shows the participating verifiers by competition year (descending), then alphabetically. The bottom section of the table shows the results using open-loop verifiers after applying our rewriting approach. Each column is one of the benchmark problems and each row is a verifier. Each cell indicates whether the corresponding verifier returned the expected result (i.e., *holds* or *violated*) on the corresponding benchmark without changing the problem. We

Table 7.1: ARCH-COMP AINNCS benchmarks that can be verified, without modification, by AINCSS 2020 and 2021 closed-loop verifiers (above the line) and by open-loop verifiers in conjunction with property, environment, and network rewriting.

	Verifier	ACC	AP	DPLR	DPMR	SB9	SB10	SP	VCAS	Total
Closed-loop	**JuliaReach**	✓				✓	✓			3
	NNV	✓							✓	2
	Verisig									0
	ReachNN*									0
	OVERT					✓		✓		2
	VenMAS	✓							✓	2
Open-loop	**Rewrite+ERAN**	✓							✓	2
	Rewrite+Marabou		✓	✓	✓			✓		4
	Rewrite+Neurify	✓		✓	✓			✓	✓	5
	Rewrite+nnenum	✓	✓	✓	✓			✓	✓	6

note that competition participants were permitted to modify problems to facilitate the use of their tool, including using knowledge of a counter-example to change the initial conditions to target a smaller input region, or using a simplification of the controller. We do not count results which were obtained on modified problems, as they either require knowledge of the result or do not provide results for the original problem. The one exception we make is for discretization of the environment models, which was employed by OVERT and VenMAS, in addition to our work. We note that *holds* results may not directly apply to the continuous time system, however *violated* results, such as those for AP, DPLR, DPMR, SP, and VCAS, can be checked against the original model.

Table 7.1 clearly demonstrates the ability of rewriting to expand the scope of closed-loop verification problems that can be verified. ARCH-COMP AINNCS problem support is quite limited among closed-loop verifiers, with the most productive verifier, JuliaReach, returning results for 3 of the 8 unmodified benchmarks. Only 5 of the benchmarks were able to be solved by any closed-loop verifiers, with AP, DPLR, and DPMR unable to be solved by any verifier without property modification. Problem rewriting provides a more general solution, and enables open-loop verifiers to solve as many as 6 of the 8 unmodified benchmarks. Additionally, rewriting allows the open-loop verifiers to find counter-examples for the AP, DPLR, and DPMR benchmarks, which previously could not be done without modifying the property. We provide a more in depth analysis of the results for each tool in Appendix H.1.1.

Two benchmarks stand out as not being solved by the rewriting approach, SB9 and SB10. SB9 was a difficult problem for the open-loop verifiers, with 2 verifiers encountering errors, 1 returning unknown, and 1 running out of time (the next experiment studies variants of SB9 that vary in difficulty). SB10 was difficult for our rewriting. It was the only problem in which the property

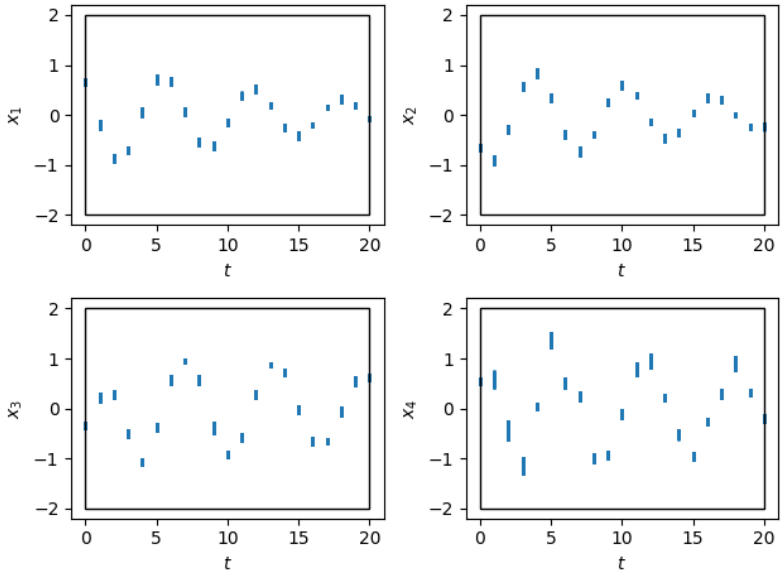


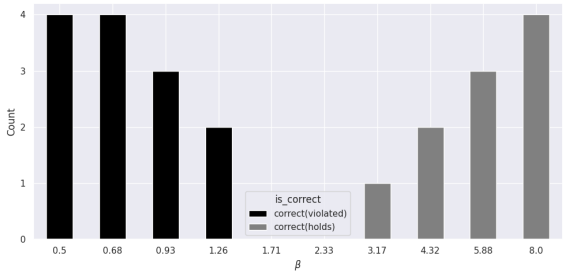
Figure 7.6: Plots of 50000 simulations of the SB9 benchmark. Each plot shows the value of 1 state variable (vertical axes) over time (horizontal axes) within control periods. The black rectangles indicate the bounds of each state variable as specified in the original property: $s_0 \in [0.6, 0.7] \times [-0.7, -0.6] \times [-0.4, -0.3] \times [0.5, 0.6] \Rightarrow \forall t \in [1, 20] : s_t \in [-2, 2]^4$.

stated that all initial states must eventually reach some region. This is difficult for our rewriting as the property reduction method results in a set of sub-problems, each specifying that at some time step, at least one path has not reached the target region. This results in $(4k)^\tau$ sub-problems, where k is the size of the state vector and τ is the number of time steps. We conjecture that one way to improve this reduction is to first reduce stronger versions of the property that imply the original. If none of these problems are proven to hold, then we can fall back to reducing the original property. We will explore this approach further in future work.

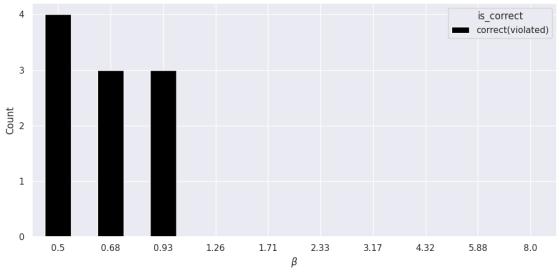
7.3.2 RQ2: Effect of Environment Architecture

Environment rewriting trains a neural network to overapproximate the true environment model. The degree of overapproximation depends on the accuracy of the trained network, which we hypothesize is affected by the network architecture. To evaluate the effect of the network architecture of the rewritten environment on verification, we performed rewriting to 5 different architectures, and used these environments for verification of 10 versions of the SB9 benchmark, modified to provide a range of problem difficulties.

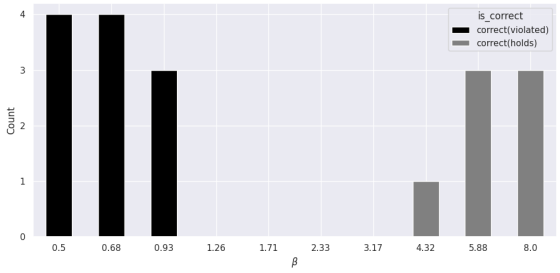
For this experiment we selected a single verification benchmark from ARCH-COMP AINNCS,



(a) Linear model.



(b) Non-linear model with size 128.



(c) Non-linear model with size 512.

Figure 7.7: Per bound verification accuracy for 3 models. Each bar represents 1 correctness problem executed on the 4 verifiers. Problems on the left have violations, problems on the right hold, and those near $\beta = 2$ are more challenging for the verifiers.

SB9, for which the original version was a difficult for our approach and that we know the expected result to be *holds*. The property specifies that the region that the state must stay within a hypercube centered at the origin. Figure 7.6 shows a simulation of this benchmark with black rectangles indicating a hypercube radius of 2 for the original property; as can be observed the property *holds* for the simulation results.

We varied the difficulty of the problem by scaling the radius up or down to get more *holds* or *violated* results, respectively. Intuitively, the larger the radius, the easier it should be for tools to prove the property, and the smaller the radius, the easier it should be for tools to falsify the property. We created 10 versions of this property, of varying difficulty, by changing the radius of the hypercubes. Radii values, which we denote β, were selected on an exponential scale from 0.5 to 8 to obtain a range of correctness problems, from those that should be easily falsified, to those that should be easily verified. This range was chosen to be roughly centered around a radii of 2, the original radius. This modifies the property from the one shown in Figure 7.6 by replacing $s_t \in [-2, 2]$ with $s_t \in [-\beta, \beta]^4$, where $\beta \in \{0.5, 0.68, 0.93, 1.26, 1.71, 2.33, 3.17, 4.32, 5.88, 8.0\}$.

We present the results for 3 of the 5 environment models in Figure 7.7. Bars depict the *correct* results, which can be either *violated* or *holds*, produced for the problems. We use the same 4 verifiers

as in experiment 1. The appendix includes a more detailed set of plots for all 5 environment models and explain why verifiers fail to produce correct results.

The nonlinear model with hidden layer of size 128 (Figure 7.7b) performed worst, with verifiers only returning results for the easier violated properties. This model cannot accurately capture the environment, resulting in large overapproximations which make it difficult for verifiers to produce *holds* results. The nonlinear model with a hidden layer of 512 appears better able to capture the environment, allowing for tighter slack intervals and enabling more accurate results, including *holds* results. The linear model performs best, producing the most results on these problem. We hypothesize that this is due to 2 reasons. First, the linear model adds the least amount of complexity into the rewritten problems for the verifiers, reducing the amount of work that they must perform. Second, the linear model has the tightest slack intervals of the 3 models even though it has the highest mean average error on the validation set. We conjecture that this is due to it having a higher bias but lower variance.

The results of this experiment indicate that a linear model is a good choice to approximate the environment behavior, but if it cannot accurately capture the non-linear dynamics of the environment then a larger non-linear model may be able to support accurate verification.

7.4 Case Study: ACAS Xu

Our study showed that rewriting closed-loop problems for open-loop verifiers was effective for a set of competitive benchmark problems. To demonstrate the power of rewriting, in this section we take the lessons learned from that study and apply them to a closed-loop property for a set of networks for aircraft collision avoidance.

7.4.1 Problem Description

Airborne collision avoidance systems are essential to ensure the safe operation of aircraft. One such system, ACAS Xu, outputs horizontal maneuver advisories based on the relative states of an ownship and intruder aircraft [74]. Recently, neural network representations have been explored as a potential replacement for the memory-intensive lookup table based approach [63, 64]. One of these network based representations uses an array of 45 neural networks [63]. This is the neural network representation which we will use in this work.

We model the system with two aircraft, the ownship and intruder, where the ownship has an aircraft collision avoidance system, ACAS Xu, which issues turn rate advisories once per second. We

choose to model the state with a vector: $[\phi_{own}, x_{own}, y_{own}, v_{own}, \phi_{int}, x_{int}, y_{int}, v_{int}, \dot{\phi}_{int}, \tau, a_{prev}]$, where ϕ is the heading of the ownship or intruder (*own* and *int*, respectively), x and y are the positions of the ownship or intruder, v is the speed of the ownship or intruder, $\dot{\phi}_{int}$ is the turn rate of the intruder, τ is the time to loss of vertical separation, and a_{prev} is the last turn advisory. To simplify the modelling, we will assume that both the ownship and intruder have constant speed, and the intruder is turning at a constant rate.

While we model the system using the absolute positions and headings of the aircraft to facilitate the transition function specification, the ACAS Xu neural networks take as input the relative distance and headings between the aircraft. The neural networks take as input a vector, $[\rho, \theta, \psi, v_{own}, v_{int}, \tau, a_{prev}]$, where ρ is the distance between ownship and intruder, θ is the angle to the intruder relative to the ownship heading, and ψ is the heading of the intruder relative to the ownship heading. This requires us to specify a state-to-input function as follows:

$$\rho = \sqrt{(x_{own} - x_{int})^2 + (y_{own} - y_{int})^2}$$
$$\theta = arccos(\frac{y_{int} - y_{own}}{\rho}) * sign(x_{own} - x_{int}) - \phi_{own}$$
$$\psi = \phi_{int} - \phi_{own}$$

The other 4 input values are simply identity mappings from their corresponding state variable.

Based on the given input, each network can produce any of 5 possible advisories, each with an associated turn rate:

0) Clear of Conflict (COC), 0.0

1) Weak Left (WL), 1.5 deg

2) Weak Right (WR), -1.5 deg

3) Strong Left (SL), 3.0 deg

4) Strong Right (SR), -3.0 deg

The network has 2 outputs, the advisory index a, and the turn rate u.

Given the current state, and the network output, this problem can be modelled as a discrete

time system with a transition function that defines the following state variable updates:

$$
\begin{aligned}
(\phi_{own})_{t+1} &= (\phi_{own})_t + u\Delta T & (\phi_{int})_{t+1} &= (\phi_{int})_t \\
(x_{own})_{t+1} &= (x_{own})_t - ((v_{own})_t * \sin((\phi_{own})_t))\Delta T & (x_{int})_{t+1} &= (x_{int})_t - ((v_{int})_t * \sin((\phi_{int})_t))\Delta T \\
(y_{own})_{t+1} &= (y_{own})_t + ((v_{own})_t * \cos((\phi_{own})_t))\Delta T & (y_{int})_{t+1} &= (y_{int})_t + ((v_{int})_t * \cos((\phi_{int})_t))\Delta T \\
(v_{own})_{t+1} &= (v_{own})_t & (v_{int})_{t+1} &= (v_{int})_t \\
(\tau)_{t+1} &= (\tau)_t & (\dot{\phi}_{int})_{t+1} &= (\dot{\phi}_{int})_t \\
(a_{prev})_{t+1} &= (a) + t
\end{aligned}
$$

where $\Delta T = 1.0$ is the discrete time step size, $(a)_t$ is the advisory predicted by the control network, and $(u)_t$ is the corresponding turn rate. Additionally, we define the statespace to be $\mathcal{S} = [-\pi, \pi] \times [-50000, 50000]^2 \times [100, 1200] \times [-\pi, \pi] \times [-50000, 50000]^2 \times [100, 1200] \times [-6°, 6°] \times [0, 0] \times [0, 4]^2 \times [-6°, 6°]$. These state bounds were selected based on the bounds on the network inputs specified in the network controller specifications.

Given the ACAS Xu networks and environment model defined above, we specify a property based on a requirement for detect and avoid (DAA) systems. Specifically the requirement, "for single-intruder, non-accelerating encounters, horizontal DAA guidance to regain [DAA well-clear] DWC shall (261) have at most one reversal" [106]. We encode this requirement in LTL as follows:

$$
\begin{aligned}
(S_0 \times \mathcal{X} \times \mathcal{Y}) \implies (& \\
& (\mathcal{S} \times \mathcal{X} \times left \wedge X(\mathcal{S} \times \mathcal{X} \times right)) \implies XXG\neg(\mathcal{S} \times \mathcal{X} \times left) \\
& \wedge \\
& (\mathcal{S} \times \mathcal{X} \times right \wedge X(\mathcal{S} \times \mathcal{X} \times left)) \implies XXG\neg(\mathcal{S} \times \mathcal{X} \times right) \\
&)
\end{aligned}
$$

where $S_0 = [0, 0]^3 \times [100, 1200] \times [-\pi, \pi] \times [-50000, 50000]^2 \times [100, 1200] \times [-3°, 3°] \times [0, 0] \times [0, 4]^2 \times [-3°, 3°]$, $left = \{(a, u) \in \mathcal{Y} \mid a = 1 \vee a = 3\}$, $right = \{(a, u) \in \mathcal{Y} \mid a = 2 \vee a = 4\}$, and $\mathcal{X}$ and $\mathcal{Y}$ are the full input and output sets of the network, respectively. Intuitively, this property specification states that, from the initial conditions, if the network predicts a left turn, followed immediately by a right turn, then the network should not predict another left turn, and similarly for turns in the opposite direction. The initial conditions, S_0 are a subset of the full statespace, $\mathcal{S}$. Because our state representation uses absolute positions and headings, but the network takes in relative values,

the initial position and heading of the ownship are irrelevant to verification. Therefore we specify that it is at the origin with a heading of 0 degrees. We will attempt to verify this property with two different maximum time bounds, first with a bound of 3 states (2 state transitions), the minimum number of steps at which a counter example could occur, and second with a bound of 4 states to demonstrate the effect of the path length on verification. Additionally, in the 4 state case, we manually split the property into 2 properties to better parallelize verification. We split the property on the conjunction in the consequent of the outer implication, producing one subproperty in which the initial turn advisory must be to the left and another in which the initial turn advisory must be to the right.

7.4.2 Rewriting Parameters

Based on the results from Section 7.2 we choose to use a linear architecture for both environment functions as the linear models tended to perform quite well, and offer the best chance at getting a result from the verifier. In addition, we kept the batch size and number of training iterations the same as that study, at 100 and $100,000$, respectively. We used the AdamW optimizer [83] with an exponentially decaying learning rate starting from 0.001 and ending at 0.000001. Models were evaluated using a randomly generated set of $50,000,000$ sample points.

7.4.3 Verifiers

Due to the much larger problem size, and longer expected verification time, we select a single verifier. We initially selected nnenum [8] as it is the most recently released verifier supported by DNNV and it performed well on the ARCH-COMP AINNCS benchmarks. However it reported internal floating point errors after around 30 minutes, so instead we use Marabou [69] as it is the second most recently released tool, and tends to perform better than ERAN [116] and Neurify [135] on larger networks in our experience.

7.4.4 Computing Resources

Both rewriting and verification were performed on a Linux machine with an AMD Ryzen Threadripper 2970WX 24-Core Processor, 128 GB of memory, and 2 NVIDIA GeForce RTX 2080 Ti graphics cards. Verification jobs were allowed to run for up to 48 hours.

Table 7.2: Training time and mean absolute error for the rewritten environment functions for ACAS Xu.

Model	**Time (seconds)**	**MAE**
$\mathcal{M}_{lin}$	346	0.123
$\mathcal{T}_{lin}$	423	146.661

7.4.5 Results

We first rewrite the environment model. The complex ACAS Xu environment required training neural networks for both $\mathcal{M}$ and $\mathcal{T}$ during the environment remodelling step. We trained linear networks, $\mathcal{M}_{lin}$ and $\mathcal{T}_{lin}$, for both functions. We report the training times and mean absolute error (MAE) of each network in Table 7.2. The MAE is computed on a randomly generated set of 100,000 network inputs. For both $\mathcal{M}_{lin}$ and $\mathcal{T}_{lin}$, the training time was under 10 minutes. As seen in column 3 of Table 7.2, the MAE of $\mathcal{M}_{lin}$ was several orders of magnitude lower than $\mathcal{T}_{lin}$. This is due to the much larger output ranges of $\mathcal{T}$, particularly for the aircraft positions. The max output range of $\mathcal{T}$ is about 4 orders of magnitude larger than the max output range of $\mathcal{M}$.

After rewriting the environment, we apply our extended version of DNNV for closed-loop rewritings, along with the Marabou verifier.

On the 2 state transition case, Marabou ran for a total of 24.5 hours before returning a *holds* result, indicating that the ACAS Xu networks will not switch the direction of the turn advisory twice within 2 state transitions. Rewriting enabled Marabou to both run and return a result for a closed-loop verification problem, when it previously could not. While we were able to obtain results for the desired property, it took over 24 hours, and only explored up to 2 state transitions. However, this result demonstrates that *rewriting enables open-loop verifiers to prove important closed-loop properties of safety-critical systems.*

On the 3 state transition case, Marabou ran for a total of 15.98 hours before running out of memory on the subproperty that started with a right turn and 15.78 hours on the subproperty starting with a left turn. While Marabou could not prove the full property over 3 state transitions, it was able to prove several subproblems within that time, providing some confidence in the original property. In both cases Marabou was able to prove 10 out of 16 subproblems, the first 8 of which are sufficient to prove the 2 state transition case. Unfortunately, while rewriting has enabled verifiers to run on more problems than ever before, we are still reliant on performant tools to get results. Fortunately, rewriting is independent of any specific verifier, and simply increases the number of tools which can run on a given problem. As more efficient tools are developed, rewriting, in conjunction

with these more performant verifiers, will be able to solve more problems than ever before.

7.5 Conclusion

In this work introduced rewritings to transform correctness problems for closed-loop systems into open-loop problems, enabling application of open-loop verification tools for neural networks. We applied these rewritings to a set of 8 benchmark problems and showed that our rewriting, in conjunction with off-the-shelf open-loop verifiers can solve up to twice as many problems as existing closed-loop verifiers. In future work we will explore more efficient rewritings for specifications that specify states that must eventually be reached, as well as explore methods for rewriting continuous time systems.

Chapter 8

Conclusions and Future Work

In this thesis we have defined a new framework for rewriting correctness problems to enable the application of a broader verification and falsification tools. Under this framework we have introduced several rewriting instantiations for properties, networks, and environment models to increase their support among existing verifiers. The property reduction introduced in Chapter 4 enables tools, such as the adversarial attacks studied in that chapter, to be applied to any property that can be specified using halfspace-polytope constraints, an extremely general and expressive property form. The network refactorings introduced in Chapter 5 increase the application of tools by transforming networks. Semantics preserving refactorings can enable the verification of many networks at verification time, but they currently support a limited set of network operations. On the other hand, non-semantics preserving refactoring is not limited by the operations in the network, but it does not produce an equivalent network and must be performed separate to verification, with the new network replacing the original in all usages. The environment remodelling approach in Chapter 6 enables open-loop verifiers to take advantage of an environment model and can decrease the number of spurious counter-examples found by keeping verifiers and falsifiers focused on the data distribution. Chapter 7 brings together all 3 types of rewriting to enable open-loop verifiers to support closed-loop correctness problems. We introduce an environment remodelling to transform functions to neural networks which likely overapproximate the original behavior. Given a remodelled environment, we introduce a property reduction to transform temporal specifications into non-temporal ones. And finally, we introduce a network refactoring to remove the complex operation graph structures that are introduced by the temporal property reduction. These 3 rewritings are sufficient to enable open-loop verifiers to check closed-loop problems, and we show that, on a set of competition benchmarks,

they can find more property violations than state-of-the-art closed-loop verifiers.

This work has resulted in several publications [108–110, 127], each with publicly available tools and artifacts (i.e., R4V, DNNF, DNNV, and DFV) to facilitate the spread and application of the ideas presented in this work. These tools are utilized by a variety of users, from researchers, to students, to ML practitioners. At the time of this writing, DNNV and DNNF have a combined 50 stars on GitHub, and almost 20 unique users interacting with the tool repositories on GitHub through issues or pull requests. Our work has inspired others to push the limits of our tools, and propose improvements to overcome those limits and support more complex problems than before [134].

Our rewritings are enabling the current generation of verifiers to support more problems than ever before. The network rewritings included in DNNV, especially the MaxPool nework rewriting, were relied on by state-of-the-art verifiers such as PerigriNN [71], nnenum [8], VeriNet [50], MN-BaB [37] and alpha-beta-CROWN [137, 147] in VNN-COMP 2022 [9] to support complex networks. These include the 4 highest placing tools in that competition and 5 of the top 6 participants (out of 11 tools).

8.1 Broader Impacts

Rewriting increases the applicability of verifiers, enabling machine learning developers and testers to assure that their systems behave as expected. Our approach enables verification of more machine learning models, significantly reducing the gap between the current state-of-the-art neural networks and those supported by verifiers. As network complexity continues to increase, our rewritings will enable verifiers to keep up. Rewriting also enables tools to check more complex behaviors than ever before, including properties of the system and environment interaction. As networks become more utilized in domains with high-cost consequences, our rewritings in conjunction with verification will enable us to assure the correctness of these systems. Enabling this assurance will reduce the likelihood of system failures, and lead to safer, more trustworthy systems.

8.2 Future Work

In this section we suggest several lines of future research for each of the three rewriting types introduced in Chapter 3, as well as rewritings for closed-loop properties in particular. In addition to the research directions presented here, there is still significant work to be done on specifying useful behavioral properties (both what and how to specify) and environment modelling, such as how to

develop more accurate generative models.

8.2.1 Reduction

The property reduction introduced in Chapter 4 produces a set of independent subproblems that, together, are equivalid with the original problem. The original problem, and generated subproblems are restricted to linear constraints over the input and outputs of the network. Two directions for future work include reductions that produce non-independent subproblems and reductions for properties that include free variables, i.e., variables that are not inputs or outputs to a network.

Non-Independent Subproblems

A new property reduction could produce sets of non-independent subproblems, where proving one or several of the subproblems removes the need to verify others. For example, given a problem with the property $\forall x \in \mathcal{X}.\mathcal{N}(x)_0 > 0 \implies (\mathcal{N}(x)_1 > -1 \wedge \mathcal{N}(x)_1 < 1)$ the reduction introduced in Chapter 4 will produce two subproblems, one to check $\forall x \in \mathcal{X}.\mathcal{N}(x)_0 > 0 \implies (\mathcal{N}(x)_1 > -1)$ and another for $\forall x \in \mathcal{X}.\mathcal{N}(x)_0 > 0 \implies (\mathcal{N}(x)_1 < 1)$. However, it may be better to generate additional subproblems, such as a subproblem that encodes $\forall x \in \mathcal{X}.\mathcal{N}(x)_0 \leq 0$ as well as those produced by the original reduction. This is a much simpler specification, and it implies the original property (and its subproblems), so, if it can be proved quickly, then verification can stop without having to obtain a result from the verifier for the other subproblems. Instead of needing to prove 2 subproblems, we only needed to prove 1 to get a result for the original problem.

Free Variables

In this thesis, properties have been defined to be specified over the inputs and outputs of neural networks, and environment states in the case of closed-loop properties. However, it can often be easier to write logical specifications after introducing an extra variable. For example, an image brightness constraint can be specified using only constraints over network inputs, such as $\forall x \in [0,1]^n.\forall i \in [n].\forall j \in [i].((x[i] - x_0[i] = x[j] - x_0[j]) \vee (x[i] = 0) \vee (x[i] = 1)) \implies \mathcal{N}(x) = \mathcal{N}(x_0)$. This constrains all pixels to change by the same value (or saturate at 0 or 1) by constraining the difference in changes between pairs of pixels to be equal. An equivalent formulation using a free variable, ϵ, is $\forall x \in [0,1]^n.\forall i \in [n].(|x[i] - x_0[i]| = \epsilon \vee (x[i] = 0) \vee (x[i] = 1)) \implies \mathcal{N}(x) = \mathcal{N}(x_0)$. A property reduction could be developed to support such properties to further reduce the burden on users when writing specifications.

8.2.2 Refactoring

The semantics preserving refactorings introduced in Chapter 5 provide a small set of refactorings that can enable verification for many networks. However, more refactorings are still needed. For example, we still don't have refactorings for ConvTranspose operations, which are relatively common in VAEs and GANs for images. This is primarily an engineering effort to increase refactoring support for these operations.

There are several interesting directions for future research in refactoring neural networks, one of which we present here, overapproximations for holds preserving refactorings.

The refactorings introduced in Chapter 5 all assume a deterministic network as input and produce a deterministic network as output. However, the environment modelling method introduced in Chapter 7 proposes the idea of using non-determinism to force verifiers to overapproximate neural network behavior. This could enable verifiers without support for smooth network operations, such as Sigmoid and Tanh, to verify properties of networks with these operations by first approximating them with piecewise-linear operations and then adding random noise to overapproximate the desired operation. The level of overapproximation could be controlled by the precision of the piecewise-linear approximation. A more precise approximation would lead to a tighter overapproximation, but it would also likely introduce more non-linearity which verifiers struggle with. This approach could even be paired with a refinement step if a spurious counter-example is found, which would produce a tighter overapproximation by using a more precise piecewise-linear approximation.

8.2.3 Remodelling

The environment rewritings in Chapters 6 and 7 generally assume that the target verifier does not accept an explicit representation of the environment. Future work on verifiers could look at both developing open-loop tools which do take in a model of the environment, as well as developing rewritings to support such tools. Additionally, the environment remodelling in Chapter 7 requires training neural networks to replace components of the environment, which can be time consuming and result in highly overapproximated models when poorly configured. Future work should explore rewritings which can automatically construct neural networks from these components without training, and with error guarantees. One way to do this would be through picewise-linear approximations, which can approximate continuous functions arbitrarily well.

8.2.4 Closed-Loop Rewriting

The rewritings for closed-loop problems introduced in Chapter 7 were able to support must-not-reach properties for discrete-time closed-loop systems. However, these rewritings do not directly support continuous-time systems and perform poorly on must-reach properties, opening at least two directions for future work.

Continuous-Time Systems

The rewritings in Chapter 7 all assume a discrete-time system. While a continuous-time system can be approximated with a discrete-time system, this requires additional work by the user to perform this remodelling, and *holds* results for the discrete-time system do not necessarily map to continuous-time system. Ideally rewritings would directly support the continuous-time system. One possible direction to explore is to develop environment rewritings which make use of a representation for the time such that verifiers could explore the statespace at all possible values of time rather than at discrete time steps.

State-Reachability Property Support

The study in Chapter 7 showed that the rewriting approach performed poorly on properties that specified states that should eventually be reached by the system, such as the one for the SB10 benchmark. This was due to the property reduction used, which produced a number of subexpressions exponential in the number of time steps for this property type. To better support these properties, we must explore new, more efficient reductions. One possible direction to address this issue may be to rewrite to non-independent subproblems, as described in Section 8.2.1. For example, we could generate a subproblem that specifies that all states must be in the set of safe states at the last time step. Then, if this subproblem holds, the original problem holds, but if it doesn't, then we must check the other subproblems. Additionally, our current reductions do not differentiate between properties and make no attempt to optimize subproblem generation. However, it could be beneficial to explore more property-specific reductions which are tailored to certain property types in order to produce either easier or fewer subproblems. For example, a reduction tailored to must-reach property could be designed to produce dependent subproblems, which may not be necessary for must-not-reach properties.

Bibliography

[1] N. Akhtar and A. Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430, 2018.

[2] Michael E. Akintunde, Elena Botoeva, Panagiotis Kouvaros, and Alessio Lomuscio. Formal verification of neural agents in non-deterministic environments. *Autonomous Agents and Multi-Agent Systems*, 36(1):6, Nov 2021.

[3] Ahmed Alaa, Boris van Breugel, Evgeny S. Saveliev, and Mihaela van der Schaar. How faithful is your synthetic data? sample-level metrics for evaluating and auditing generative models. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 290–306. PMLR, 2022.

[4] Matthias Althoff. An introduction to cora 2015. In Goran Frehse and Matthias Althoff, editors, *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*, volume 34 of *EPiC Series in Computing*, pages 120–151. EasyChair, 2015.

[5] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. A public domain dataset for human activity recognition using smartphones. In *21st European Symposium on Artificial Neural Networks, ESANN 2013, Bruges, Belgium, April 24-26, 2013*, 2013.

[6] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[7] Edoardo Bacci, Mirco Giacobbe, and David Parker. Verifying reinforcement learning up to infinity. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 2154–2160. International Joint Conferences on Artificial Intelligence Organization, 8 2021. Main Track.

[8] Stanley Bak. nnenum: Verification of relu neural networks with optimized abstraction refinement. In *NASA Formal Methods Symposium*, pages 19–36. Springer, 2021.

[9] Stanley Bak, Changliu Liu, Taylor T. Johnson, Christopher Brix, and Mark Müller. Vnn-comp 2022. https://sites.google.com/view/vnn2022.

[10] Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T. Johnson. Improved geometric path enumeration for verifying relu neural networks. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 66–96, Cham, 2020. Springer International Publishing.

[11] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 2621–2629, USA, 2016. Curran Associates Inc.

[12] David Berend, Xiaofei Xie, Lei Ma, Lingjun Zhou, Yang Liu, Chi Xu, and Jianjun Zhao. Cats are not fish: Deep learning testing calls for out-of-distribution awareness. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1041–1052. IEEE, 2020.

[13] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320, 1999.

[14] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Kostiantyn Potomkin, and Christian Schilling. Juliareach: A toolbox for set-based reachability. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '19, page 39–44, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[16] Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. In *AAAI*, Jan 2019.

[17] Elena Botoeva, Panagiotis Kouvaros, Jan Kronqvist, Alessio Lomuscio, and Ruth Misener. Efficient verification of relu-based neural networks via dependency analysis. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 3291–3299. AAAI Press, 2020.

[18] Christopher Brix and Thomas Noll. Debona: Decoupled boundary network analysis for tighter bounds and faster adversarial robustness proofs. *CoRR*, abs/2006.09040, 2020.

[19] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. Piecewise linear neural network verification: A comparative study. *CoRR*, abs/1711.00455v1, 2017.

[20] Rudy R. Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and Pawan Kumar Mudigonda. A unified view of piecewise linear neural network verification. In *NeurIPS*, pages 4795–4804, 2018.

[21] Taejoon Byun and Sanjai Rayadurgam. Manifold for machine learning assurance. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, page 97–100, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[23] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. *CoRR*, abs/1608.04644, 2016.

[24] Guobin Chen, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. Learning efficient object detection models with knowledge distillation. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 742–751. Curran Associates, Inc., 2017.

[25] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 258–263, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[26] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (E). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440. IEEE, 2015.

[27] Mel Ó Cinnéide, Dermot Boyle, and Iman Hemati Moghadam. Automated refactoring for testability. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 437–443. IEEE, 2011.

[28] commaai. openpilot. `https://github.com/commaai/openpilot`, 2022.

[29] Bin Dai and David P. Wipf. Diagnosing and enhancing VAE models. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[30] Philippe Dhaussy, Jean-Charles Roger, and Frederic Boniol. Reducing state explosion with context modeling for model-checking. In *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*, pages 130–137. IEEE, 2011.

[31] Swaroopa Dola, Matthew B. Dwyer, and Mary Lou Soffa. Distribution-Aware Testing of Neural Networks Using Generative Model. In *Proceedings of the International Conference on Software Engineering*, 2021.

[32] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In *NFM*, volume 10811 of *Lecture Notes in Computer Science*, pages 121–138. Springer, 2018.

[33] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy Mann, and Pushmeet Kohli. A dual approach to scalable verification of deep networks. In *Proceedings of the Thirty-Fourth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-18)*, pages 162–171, Corvallis, Oregon, 2018. AUAI Press.

[34] Rüdiger Ehlers. 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*, pages 269–286, 2017.

[35] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. An abstraction-based framework for neural network verification. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 43–65. Springer, 2020.

[36] Jiameng Fan, Chao Huang, Xin Chen, Wenchao Li, and Qi Zhu. Reachnn*: A tool for reachability analysis of neural-network controlled systems. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis*, pages 537–542, Cham, 2020. Springer International Publishing.

[37] Claudio Ferrari, Mark Niklas Müller, Nikola Jovanovic, and Martin T. Vechev. Complete verification via multi-neuron relaxation guided branch-and-bound. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[38] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[39] Mikhail R Gadelha, Felipe R Monteiro, Jeremy Morse, Lucas C Cordeiro, Bernd Fischer, and Denis A Nicole. Esbmc 5.0: an industrial-strength c model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 888–891, 2018.

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

[41] Dimitra Giannakopoulou, Corina S Pasareanu, and Jamieson M Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Proceedings. 26th International Conference on Software Engineering*, pages 211–220. IEEE, 2004.

[42] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *International Conference on Computer Aided Verification*, pages 332–342. Springer, 1991.

[43] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[44] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[45] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[46] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The seahorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer, 2015.

[47] Mark Harman. Refactoring as testability transformation. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 414–421. IEEE, 2011.

[48] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[49] P. Henriksen and A. Lomuscio. Efficient neural network verification via adaptive refinement and adversarial search. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2513–2520. IOS Press, 2020.

[50] Patrick Henriksen and Alessio Lomuscio. DEEPSPLIT: an efficient splitting method for neural network verification via indirect effect analysis. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 2549–2555. ijcai.org, 2021.

[51] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[52] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[53] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[54] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2261–2269. IEEE Computer Society, 2017.

[55] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *CAV (1)*, volume 10426 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2017.

[56] Paul Hudak. Modular domain specific languages and tools. In *Proceedings. Fifth International Conference on Software Reuse*, pages 134–142. IEEE, 1998.

[57] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[58] Omri Isac, Clark Barrett, Min Zhang, and Guy Katz. Neural network verification with proof production, 2022.

[59] Radoslav Ivanov, Taylor J. Carpenter, James Weimer, Rajeev Alur, George J. Pappas, and Insup Lee. Verisig 2.0: Verification of neural network controllers using taylor model preconditioning. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 249–262. Springer, 2021.

[60] Taylor T. Johnson, Diego Manzanas Lopez, Luis Benet, Marcelo Forets, Sebasti\'an Guadalupe, Christian Schilling, Radoslav Ivanov, Taylor J. Carpenter, James Weimer, and Insup Lee. Arch-comp21 category report: Artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In Goran Frehse and Matthias Althoff, editors, *8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21)*, volume 80 of *EPiC Series in Computing*, pages 90–119. EasyChair, 2021.

[61] Taylor T Johnson, Diego Manzanas Lopez, Patrick Musau, Hoang-Dung Tran, Elena Botoeva, Francesco Leofante, Amir Maleki, Chelsea Sidrane, Jiameng Fan, and Chao Huang. Archcomp20 category report: Artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants. In Goran Frehse and Matthias Althoff, editors, *ARCH20. 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20)*, volume 74 of *EPiC Series in Computing*, pages 107–139. EasyChair, 2020.

[62] Kyle D. Julian and Mykel J. Kochenderfer. A reachability method for verifying dynamical systems with deep neural network controllers. *CoRR*, abs/1903.00520, 2019.

[63] Kyle D. Julian, Mykel J. Kochenderfer, and Michael P. Owen. Deep neural network compression for aircraft collision avoidance systems. *Journal of Guidance, Control, and Dynamics*, 42(3):598–608, 2019.

[64] Kyle D. Julian, Jessica Lopez, Jeffrey S. Brush, Michael P. Owen, and Mykel J. Kochenderfer. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–10, 2016.

[65] P. Julian, A. Desages, and O. Agamennoni. High-level canonical piecewise linear representation using a simplicial partition. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 46(4):463–480, 1999.

[66] Sunggoo Jung, Sunyou Hwang, Heemin Shin, and David Hyunchul Shim. Perception, guidance, and navigation for indoor autonomous drone racing using deep learning. *IEEE Robotics and Automation Letters*, 3(3):2539–2544, 2018.

[67] Sunggoo Jung, Sunyou Hwang, Heemin Shin, and David Hyunchul Shim. Perception, guidance, and navigation for indoor autonomous drone racing using deep learning. *IEEE Robotics and Automation Letters*, 3(3):2539–2544, 2018.

[68] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 97–117, 2017.

[69] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. The Marabou framework

for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452. Springer, 2019.

[70] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Testera: A tool for testing java programs using alloy specifications. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 608–611. IEEE, 2011.

[71] Haitham Khedr, James Ferlez, and Yasser Shoukry. Effective formal verification of neural networks using the geometry of linear regions. *CoRR*, abs/2006.10864, 2020.

[72] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[73] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[74] Mykel J Kochenderfer and JP Chryssanthacopoulos. Robust airborne collision avoidance through dynamic programming. *Massachusetts Institute of Technology, Lincoln Laboratory, Project Report ATC-371*, 130, 2011.

[75] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images, 2009.

[76] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[77] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017.

[78] Sampo Kuutti, Richard Bowden, Yaochu Jin, Phil Barber, and Saber Fallah. A survey of deep learning applications to autonomous vehicle control. *IEEE Transactions on Intelligent Transportation Systems*, 22(2):712–733, 2021.

[79] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits.

[80] Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark W. Barrett, and Mykel J. Kochenderfer. Algorithms for verifying deep neural networks. *CoRR*, abs/1903.06758, 2019.

[81] Changliu Liu and Taylor T. Johnson. Vnn-comp. https://sites.google.com/view/vnn20/vnncomp.

[82] Antonio Loquercio, Ana Isabel Maqueda, Carlos R. Del Blanco, and Davide Scaramuzza. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 2018.

[83] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[84] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. Deepgauge: multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 120–131, 2018.

[85] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[86] Matthias Markthaler, Stefan Kriebel, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving model-based testing in automotive software engineering. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 172–180. IEEE, 2018.

[87] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2574–2582. IEEE Computer Society, 2016.

[88] Muhammad Ferjad Naeem, Seong Joon Oh, Youngjung Uh, Yunjey Choi, and Jaejun Yoo. Reliable fidelity and diversity metrics for generative models. In *International Conference on Machine Learning*, pages 7176–7185. PMLR, 2020.

[89] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. TensorFuzz, 2019. https://github.com/brain-research/tensorfuzz.

[90] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. TensorFuzz: Debugging neural networks with coverage-guided fuzzing. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4901–4911, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[91] Kurt M. Olender and Leon J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Trans. Softw. Eng. Methodol.*, 1(1):21–52, January 1992.

[92] ONNX. Open Neural Network Exchange. `https://github.com/onnx/onnx`, 2017.

[93] Nicolas Papernot, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, Alexander Matyasko, Vahid Behzadan, Karen Hambardzumyan, Zhishuai Zhang, Yi-Lin Juang, Zhi Li, Ryan Sheatsley, Abhibhav Garg, Jonathan Uesato, Willi Gierke, Yinpeng Dong, David Berthelot, Paul Hendricks, Jonas Rauber, and Rujun Long. Technical report on the cleverhans v2.1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768*, 2018.

[94] Corina S Păsăreanu, Matthew B Dwyer, and Willem Visser. Finding feasible counter-examples when model checking abstracted java programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 284–298. Springer, 2001.

[95] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[96] Brandon Paulsen, Jingbo Wang, and Chao Wang. Reludiff: Differential verification of deep neural networks. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020*, 2020.

[97] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 1–18, 2017.

[98] Zi Peng, Jinqiu Yang, Tse-Hsun (Peter) Chen, and Lei Ma. *A First Look at the Integration of Machine Learning Models in Complex Autonomous Driving Systems: A Case Study on Apollo*, page 1240–1250. Association for Computing Machinery, New York, NY, USA, 2020.

[99] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 4092–4101, 2018.

[100] Tom Preston-Werner. TOML. `https://github.com/toml-lang/toml`, 2013.

[101] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[102] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified defenses against adversarial examples. In *ICLR*. OpenReview.net, 2018.

[103] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[104] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. Reachability analysis of deep neural networks with provable guarantees. In *IJCAI*, pages 2651–2659. ijcai.org, 2018.

[105] Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P Kingma. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*, 2017.

[106] RTCA. SC-228. *DO-365, Minimum Operational Performance Standards (MOPS) for Detect and Avoid (DAA) Systems*. RTCA, Incorporated, 2017.

[107] Christian Schilling, Marcelo Forets, and Sebastián Guadalupe. Verification of neural-network control systems by integrating taylor models and zonotopes. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(7):8169–8177, Jun. 2022.

[108] David Shriver, Sebastian G. Elbaum, and Matthew B. Dwyer. DNNV: A framework for deep neural network verification. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Los Angeles, CA, USA, July 18-23, 2021, Proceedings, Part I*, 2021.

[109] David Shriver, Sebastian G. Elbaum, and Matthew B. Dwyer. Reducing DNN properties to enable falsification with adversarial attacks. In *Proceedings of the International Conference on Software Engineering*, 2021.

[110] David Shriver, Dong Xu, Sebastian G. Elbaum, and Matthew B. Dwyer. Refactoring neural networks for verification. *CoRR*, abs/1908.08026, 2019.

[111] Chelsea Sidrane and Mykel J. Kochenderfer. OVERT: Verification of nonlinear dynamical systems with neural network controllers via overapproximation. *Safe Machine Learning workshop at ICLR*, 2019.

[112] Chelsea Sidrane, Amir Maleki, Ahmed Irfan, and Mykel J. Kochenderfer. Overt: An algorithm for safety verification of neural network control policies for nonlinear systems. *Journal of Machine Learning Research*, 23(117):1–45, 2022.

[113] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[114] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin T. Vechev. Beyond the single neuron convex barrier for neural network certification. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 15072–15083, 2019.

[115] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10802–10813. Curran Associates, Inc., 2018.

[116] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. An abstract domain for certifying neural networks. *PACMPL*, 3(POPL):41:1–41:30, 2019.

[117] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. Boosting robustness certification of neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[118] Robert E. Strom and Daniel M Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering*, 19(5):478–485, 1993.

[119] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *CoRR*, abs/1710.08864, 2017.

[120] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 109–119, 2018.

[121] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 1–9. IEEE Computer Society, 2015.

[122] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2818–2826. IEEE Computer Society, 2016.

[123] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[124] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 303–314, 2018.

[125] Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2019.

[126] Oksana Tkachuk, Matthew B Dwyer, and Corina S Pasareanu. Automated environment generation for software model checking. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 116–127. IEEE, 2003.

[127] Felipe Toledo, David Shriver, Sebastian G. Elbaum, and Matthew B. Dwyer. Distribution models for falsification and verification of DNNs. In *International Conference on Automated Software Engineering, ASE 2021*, 2021.

[128] Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T. Johnson. Verification of deep convolutional neural networks using imagestars. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 18–42. Springer, 2020.

[129] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2020.

[130] Cumhur Erkan Tuncali, Georgios Fainekos, Hisahiro Ito, and James Kapinski. Simulation-based adversarial test generation for autonomous vehicles with machine learning components. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1555–1562. IEEE, 2018.

[131] Udacity. Self driving car. `https://github.com/udacity/self-driving-car`, 2016.

[132] Aleksei Vasilev, Vladimir Golkov, Marc Meissner, Ilona Lipp, Eleonora Sgarlata, Valentina Tomassini, Derek K. Jones, and Daniel Cremers. q-space novelty detection with variational autoencoders. In Elisenda Bonet-Carne, Jana Hutter, Marco Palombo, Marco Pizzolato, Farshid Sepehrband, and Fan Zhang, editors, *Computational Diffusion MRI*, pages 113–124, Cham, 2020. Springer International Publishing.

[133] Joseph A. Vincent and Mac Schwager. Reachable polyhedral marching (rpm): A safety verification algorithm for robotic systems with deep neural network components, 2021.

[134] Meriel von Stein and Sebastian Elbaum. Finding property violations through network falsification: Challenges, adaptations and lessons learned from openpilot. In *IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[135] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. In *NeurIPS*, pages 6369–6379, 2018.

[136] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *USENIX Security Symposium*, pages 1599–1614. USENIX Association, 2018.

[137] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 29909–29921, 2021.

[138] Shuning Wang, Xiaolin Huang, and Khan M. Junaid. Configuration of continuous piecewise-linear neural networks. *IEEE Transactions on Neural Networks*, 19(8):1431–1445, 2008.

[139] Shuning Wang and Xusheng Sun. Generalization of hinging hyperplanes. *IEEE Transactions on Information Theory*, 51(12):4425–4431, 2005.

[140] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[141] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane S. Boning, and Inderjit S. Dhillon. Towards fast computation of certified robustness for relu networks. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 5273–5282. PMLR, 2018.

[142] Eric Wong and J. Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 5283–5292. PMLR, 2018.

[143] W. Xiang, H. Tran, and T. T. Johnson. Output reachable set estimation and verification for multilayer neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(11):5777–5783, Nov 2018.

[144] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[145] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. Deephunter: A coverage-guided fuzz testing framework for deep neural networks. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[146] Dong Xu, David Shriver, Matthew B. Dwyer, and Sebastian G. Elbaum. Systematic generation of diverse benchmarks for DNN verification. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 97–121. Springer, 2020.

[147] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui. Hsieh. Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. *International Conference on Learning Representation (ICLR)*, 2021.

[148] Xiang Yin, John Knight, and Westley Weimer. Exploiting refactoring in formal verification. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 53–62. IEEE, 2009.

[149] X. Yuan, P. He, Q. Zhu, and X. Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE Transactions on Neural Networks and Learning Systems*, 30(9):2805–2824, 2019.

[150] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[151] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. *Advances in Neural Information Processing Systems*, 31:4939–4948, 2018.

Appendix A

On the Equisatisfiability of Reduction

Lemma 2. *Let ϕ be a conjunction of linear inequalities over the variables x_i for i from 0 to $n-1$. We can construct an H-polytope $H = (A, b)$ with Alg. 2 s.t. $(Ax \leq b) \Leftrightarrow (x \models \phi)$.*

Proof. Let $f(x) = \sum_0^{n-1} a_i x_i$ We first show that every lin. ineq. in the conjunction can be reformulated to the form $f(x) \leq b$. It is trivial to show the ineq. can be manipulated to have variables on lhs and a constant on rhs, that $\geq$ can be manipulated to an equivalent form with $\leq$, and $>$ can be manipulated to become $<$. The $<$ comparison can be changed to a $\leq$ comparison by decrementing the rhs constant from b to b' where b' is the largest representable number less than b. We prove ineq. with $<$ can be reformulated to use $\leq$ by contradiction. Assume either $f(x) < b$ and $f(x) > b'$ or $f(x) \geq b$ and $f(x) \leq b'$. Either $b' < f(x) < b$, a contradiction, since $f(x)$ cannot be both larger than the largest representable number less than b and also less than b.[1] Or $b \leq f(x) \leq b'$, a contradiction, since $b' < b$ by definition.

Given a conjunction of lin. ineq. in the form $f(x) \leq b$, Alg. 2 constructs A and b with a row in A and value in b corresponding to each conjunct. There are two cases: $(Ax \leq b) \rightarrow (x \models \phi)$ and $(x \models \phi) \rightarrow (Ax \leq b)$.

We prove case 1 by contradiction. Assume $(Ax \leq b)$ and $(x \not\models \phi)$. By construction of H in Alg. 2, each conjunct of ϕ is exactly 1 constraint in H. If $Ax \leq b$, then all constraints in H must be satisifed, and thus all conjuncts in ϕ must be satisfied and $x \models \phi$, a contradiction.

We prove case 2 by contradiction. Assume $(x \models \phi)$ and $(Ax \not\leq b)$. By construction of H in

[1] We discuss the assumption that such a number exists in Appendix A.1.

Alg. 2, each conjunct of ϕ is exactly 1 constraint in H. If $x \models \phi$, then all conjuncts in ϕ must be satisfied, and thus all constraints in H must be satisifed and $Ax \leq b$, a contradiction. □

Lemma 3. *Let $H = (A, b)$ be an H-polytope s.t. $Ax \leq b$. Alg. 4 constructs a DNN, $\mathcal{N}_s$, that classifies whether inputs satisfy $Ax \leq b$. Formally, $x \in H \Leftrightarrow \mathcal{N}_s(x)_0 \leq \mathcal{N}_s(x)_1$.*

Proof. There are 2 cases:

1. $x \in H \rightarrow \mathcal{N}_s(x)_0 \leq \mathcal{N}_s(x)_1$
2. $\mathcal{N}_s(x)_0 \leq \mathcal{N}_s(x)_1 \rightarrow x \in H$

We prove case 1 by contradiction. Assume $x \in H$ and $\mathcal{N}_s(x)_0 > \mathcal{N}_s(x)_1$. From Alg. 4, each neuron in the hidden layer of $\mathcal{N}_s$ corresponds to one constraint in H. The weights of each neuron are the values in the corresponding row of A, and the bias is the negation of the corresponding value of b. If input x satisfies the constraint, then the neuron value will be at most 0, otherwise it will be greater than 0. After the ReLU, each neuron will be equal to 0 if the corresponding constraint is satisfied by x and greater than 0 otherwise. The first output neuron sums all neurons in the hidden layer, while the second has a constant value of 0. If $x \in H$, then all neurons in the hidden layer after activation must have a value of 0 since all constraints are satisfied. However, if all neurons have value 0, then their sum must also be 0, and therefore $\mathcal{N}_s(x)_0 = \mathcal{N}_s(x)_1$, a contradiction.

We prove case 2 by contradiction. Assume $\mathcal{N}_s(x)_0 \leq \mathcal{N}_s(x)_1$ and $x \notin H$. If $x \notin H$, at least one neuron in the hidden layer must have a value greater than 0 after the ReLU since at least one constraint is not satisfied. Because some neuron has a value greater than 0, their sum must also be greater than 0, and therefore $\mathcal{N}_s(x)_0 > \mathcal{N}_s(x)_1$, a contradiction. □

Lemma 4. *Let $H = (A, b)$ be an H-polytope s.t. $Ax \leq b$. Alg. 3 constructs a DNN, $\mathcal{N}_p$, that maps values from the n-dim. unit hypercube to the axis aligned hyperrectangle that minimally bounds H. The range of this mapping does not exclude any x s.t. $Ax \leq b$. Formally, $\forall x \in H. \exists z \in [0, 1]^n. x = \mathcal{N}_p(z)$.*

Proof. The proof is by contradiction. Let the axis aligned hyperrectangle that minimally bounds H be specified by lower bounds $\vec{lb}$ and upper bounds $\vec{ub}$ s.t. $\forall x \in H. \forall i. x_i \in [lb_i, ub_i]$. Alg. 3 constructs a DNN, $\mathcal{N}_p$, that computes $\mathcal{N}_p(z) = Wz + b$, where $W = \text{diag}(ub - lb)$ and $b = lb$. This function is invertible: $\mathcal{N}_p^{-1}(x) = W^{-1}(x - b) = W^{-1}x - W^{-1}b$. Assume $\exists x \in H. \exists i. (z = \mathcal{N}_p^{-1}(x)) \wedge ((z_i < 0) \vee (z_i > 1))$. From the def. of $\mathcal{N}_p^{-1}$, we get $\mathcal{N}_p^{-1}(lb)_i \leq z_i \leq \mathcal{N}_p^{-1}(ub)_i$ and $W_{i,i}^{-1}(lb_i) - W_{i,i}^{-1}(lb_i) = 0 \leq z_i \leq W_{i,i}^{-1}(ub_i) - W_{i,i}^{-1}(lb_i) = (\frac{1}{ub_i - lb_i}(ub_i) - \frac{1}{ub_i - lb_i}(lb_i)) = 1$. Therefore $(lb_i \leq x_i \leq ub_i) \rightarrow (0 \leq z_i \leq 1)$, a contradiction. □

Theorem 5. *Let $\psi = \langle \mathcal{N}, \phi \rangle$ be a correctness problem with property defined as disjunctions and conjunctions of linear inequalities over the inputs and outputs of $\mathcal{N}$. Property Reduction (Alg. 1) maps ψ to an equivalid set of correctness problems $reduce(\psi) = \{\langle \mathcal{N}_1, \phi_1 \rangle, \ldots, \langle \mathcal{N}_k, \phi_k \rangle\}$.*

$$\mathcal{N} \models \psi \Leftrightarrow \forall \langle \mathcal{N}_i, \phi_i \rangle \in reduce(\psi). \mathcal{N}_i \models \phi_i$$

Proof. A model that satisfies any disjunct of $DNF(\neg\phi)$ falsifies ϕ. If ϕ is falsifiable, then at least one disjunct of $DNF(\neg\phi)$ is satisfiable.

Alg. 1 constructs a correctness problem for each disjunct. For each disjunct, Alg. 1 constructs an H-polytope, H, which is used to construct a prefix network, $\mathcal{N}_p$, and suffix network, $\mathcal{N}_s$. The algorithm then constructs networks $\mathcal{N}'(x) = concat(\mathcal{N}(x), x)$ and $\mathcal{N}''(x) = \mathcal{N}_s(\mathcal{N}'(\mathcal{N}_p(x)))$. Alg. 1 pairs each constructed network with the property $\phi = \forall x.x \in [0,1]^n \rightarrow \mathcal{N}''(x)_0 > \mathcal{N}''(x)_1$. A violation occurs only when $\mathcal{N}''(x)_0 \leq \mathcal{N}''(x)_1$. By Lemmas 2, 3, and 4, we get that $\mathcal{N}''(x)_0 \leq \mathcal{N}''(x)_1$ if and only if $\mathcal{N}'(x) \in H$. If $\mathcal{N}'(x) \in H$ then $\mathcal{N}'(x)$ satisfies the disjunct and is therefore a violation of the original property. □

A.1 On Existance of a Bounded Largest Representable Number

Our proof that property reduction generates a set of robustness problems equivalid to an arbitrary problem relies on the assumption that strict inequalities can be converted to non-strict inequalities. To do so we rely on the existance of a largest representable number that is less than some given value. While this is not necessarily true for all sets of numbers (e.g., $\mathbb{R}$), it is true for for most numeric representations used in computation (e.g., IEEE 754 floating point arithmetic).

Appendix B

DNNP

A property specification defines the desired behavior of a neural network in a formal language. DNNF and DNNV use a custom Python-embedded DSL for writing property specifications, which we call DNNP. Embedding DNNP in Python allows for the rich ecosystem of the host language to be used in writing specifications [56]. However, DNNP is a work-in-progress, so some expressions (such as star expressions) are not yet supported by our property parser. We are still working to fully support all Python expressions, but the current version supports the most common use cases.

Figure 2 shows the definition of the DNNP grammar. The general structure of a property specification is as follows:

1. A set of python module imports
2. A set of variable definitions
3. A property expression

B.1 Imports

Imports have the same syntax as Python import statements, and they can be used to import arbitrary Python modules and packages. This allows re-use of datasets or input pre-processing code. For example, the Python package `numpy` can be imported to load a dataset. Inputs can then be selected from the dataset, or statistics, such as the mean data point, can be computed on the fly.

```
<property> ::= <python-imports> <assignment-list> <expr>

<python-imports> ::= `'
    | <python-imports> `import' <id>
    | <python-imports> `import' <id> `as' <id>
    | <python-imports> `from' <id> `import' <id>

<assignment-list> ::= `'
    | <assignment-list> <assignment>

<assignment> ::= <id> `=' <expr>

<expr> ::= `Forall(' <id> `,' <expr> `)'
    | `And(' <expr-seq> `)'
    | `Or(' <expr-seq> `)'
    | `Implies(' <expr> `,' <expr> `)'
    | `Parameter(' <id> `, type=' <id> `)'
    | ...
    | <python-expr>

<expr-seq> ::= <expr> | <expr-seq> `,' <expr>
```

Listing 2: Subset of the grammar for DNNP.

B.2 Definitions

After any imports, DNNP allows a sequence of assignments to define variables that can be used in the final property specification. For example, `i = 0`, will define the variable `i` to a value of 0.

These definitions can be used to load data and configuration parameters, or to alias expressions that may be used in the property formula. For example, if the `torchvision.datasets` package has been imported, then `data = datasets.MNIST("/tmp")` will define a variable `data` referencing the MNIST dataset from this package. Additionally, the `Parameter` class can be used to declare parameters that can be specified at run time. `eps = Parameter("epsilon", type=float)`, will define the variable `eps` to have type float and will expect a value to be specified at run time. This value can be specified to DNNF or DNNV with the option `--prop.epsilon`.

Definitions can also assign expressions to variables to be used in the property specification later. For example, `x_in_unit_hyper_cube = 0 <= x <= 1` can be used to assign an expression specifying that the variable x is within the unit hyper cube to a variable. This could be useful for more complex properties with a lot of redundant sub-expressions.

A network can be defined using the `Network` class. `N = Network("N")`, specifies a network with the name `N` (which is used at run time to concretize the network with a specific network). All

networks with the same name refer to the same model.

B.3 Property Expression

Finally, the last part of the property specification is the property formula itself. It must appear at the end of the property specification. All statements before the property formula must be either import or assignment statements.

The property formula defines the desired behavior of the neural network in a subset of first-order-logic. It can make use of arbitrary Python code, as well as any of the expressions defined before it.

DNNP provides many functions for defining logical expressions. The function `Forall( symbol, expression)` can be used to specify that the provided expression is valid for all values of the specified symbol. The function `And(*expression)`, specifies that all of the expressions passed as arguments to the function must be valid. `And(expr1, expr2)` can be equivalently specified as `expr1 & expr2`. The function `Or(*expression)`, specifies that at least one of the expressions passed as arguments to the function must be valid. `Or(expr1, expr2)` can be equivalently specified as `expr1 | expr2`. The function `Implies(expression1, expression2)`, specifies that if `expression1` is true, then `expression2` must also be true. The `argmin` and `argmax` functions can be used to get the argmin or argmax value of a network's output, respectively.

In property expressions, networks can be called like functions to get the outputs for the network for a given input. Networks can be applied to symbolic variables (such as universally quantified variables), as well as numpy arrays.

Appendix C

Evaluation Benchmarks for Chapter 4

C.1 ACAS Xu Property Benchmark

The ACAS Xu problem benchmark consists of 10 DNN properties, each applied to a subset of 45 small networks. This benchmark is described in detail in Appendix VI of [68]. Each of the 45 fully-connected networks in this benchmark have 5 input values and 5 output values with 6 hidden layers of 50 neurons each and relu activations. For completeness, we provide formal definitions of the 10 ACAS Xu properties.

Property ϕ_1

$$\forall x.((55947.691 \leq x_0 \leq 60760) \wedge (-\pi \leq x_1 \leq \pi) \wedge (-\pi \leq x_2 \leq \pi)$$
$$\wedge(1145 \leq x_3 \leq 1200) \wedge (0 \leq x_4 \leq 60)) \rightarrow (\mathcal{N}(x)_0 \leq 1500)$$

Property ϕ_2

$$\forall x.((55947.691 \leq x_0 \leq 60760) \wedge (-\pi \leq x_1 \leq \pi) \wedge (-\pi \leq x_2 \leq \pi)$$
$$\wedge(1145 \leq x_3 \leq 1200) \wedge (0 \leq x_4 \leq 60)) \rightarrow (argmax(\mathcal{N}(x)) \neq 0)$$

Property ϕ_3

$$\forall x.((1500 \leq x_0 \leq 1800) \wedge (-0.06 \leq x_1 \leq 0.06) \wedge (3.10 \leq x_2 \leq \pi)$$
$$\wedge(980 \leq x_3 \leq 1200) \wedge (960 \leq x_4 \leq 1200)) \rightarrow (argmin(\mathcal{N}(x)) \neq 0)$$

Property ϕ_4

$$\forall x.((1500 \leq x_0 \leq 1800) \wedge (-0.06 \leq x_1 \leq 0.06) \wedge (0 \leq x_2 \leq 0)$$
$$\wedge(1000 \leq x_3 \leq 1200) \wedge (700 \leq x_4 \leq 800)) \rightarrow (argmin(\mathcal{N}(x)) \neq 0)$$

Property ϕ_5

$$\forall x.((250 \leq x_0 \leq 400) \wedge (0.2 \leq x_1 \leq 0.4) \wedge (-\pi \leq x_2 \leq -\pi + 0.005)$$
$$\wedge(100 \leq x_3 \leq 400) \wedge (0 \leq x_4 \leq 400)) \rightarrow (argmin(\mathcal{N}(x)) = 4)$$

Property ϕ_6

$$\forall x.(((12000 \leq x_0 \leq 62000) \wedge (0.7 \leq x_1 \leq \pi) \wedge (-\pi \leq x_2 \leq -\pi + 0.005)$$
$$\wedge(100 \leq x_3 \leq 1200) \wedge (0 \leq x_4 \leq 1200))$$
$$\vee(12000 \leq x_0 \leq 62000) \wedge (-\pi \leq x_1 \leq -0.7) \wedge (-\pi \leq x_2 \leq -\pi + 0.005)$$
$$\wedge(100 \leq x_3 \leq 1200) \wedge (0 \leq x_4 \leq 1200)) \rightarrow (argmin(\mathcal{N}(x)) = 0)$$

Property ϕ_7

$$\forall x.((0 \leq x_0 \leq 60760) \wedge (-\pi \leq x_1 \leq \pi) \wedge (-\pi \leq x_2 \leq \pi)$$
$$\wedge(100 \leq x_3 \leq 1200) \wedge (0 \leq x_4 \leq 1200)) \rightarrow (argmin(\mathcal{N}(x)) \neq 4)$$

Property ϕ_8

$$\forall x.((0 \leq x_0 \leq 60760) \wedge (-\pi \leq x_1 \leq -0.75\pi) \wedge (-0.1 \leq x_2 \leq 0.1)$$
$$\wedge(600 \leq x_3 \leq 1200) \wedge (600 \leq x_4 \leq 1200)) \rightarrow ((argmin(\mathcal{N}(x)) = 0) \vee (argmin(\mathcal{N}(x)) = 1))$$

Property ϕ_9

$$\forall x.((2000 \leq x_0 \leq 7000) \wedge (-0.4 \leq x_1 \leq -0.14) \wedge (-\pi \leq x_2 \leq -\pi + 0.01)$$
$$\wedge(100 \leq x_3 \leq 150) \wedge (0 \leq x_4 \leq 150)) \rightarrow (argmin(\mathcal{N}(x)) = 3)$$

Property ϕ_{10}

$$\forall x.((36000 \leq x_0 \leq 60760) \wedge (0.7 \leq x_1 \leq \pi) \wedge (-\pi \leq x_2 \leq -\pi + 0.01)$$
$$\wedge(900 \leq x_3 \leq 1200) \wedge (600 \leq x_4 \leq 1200)) \rightarrow (argmin(\mathcal{N}(x)) = 0)$$

C.2 Neurify-DAVE Property Benchmark

The Neurify-DAVE benchmark, introduced in [135], is a set of local interval-reachability properties applied to a network that predicts steering angles for a self-driving car. The original benchmark applied these properties to a smaller version of the original DAVE DNN. The networks take 100x100 color images as input and produce a single value, y, which is converted to a value between $-\pi$ and π with the function $f(x) = 2 * arctan(x)$. While the smaller network has an input domain of $[0, 1]^{30000}$, the original network uses an input domain of $[-103.939, 103.939]^{10000} \times [-116.779, 116.779]^{10000} \times [-123.68, 123.68]^{10000}$, due to mean centering of inputs originally in the interval $[0, 255]^{30000}$.

The small version of DAVE has 2 convolutional layers with relu activations, and 24 and 36 5x5 kernels, respectively. Both of these layers use strides of 5 and have no padding. These are followed by 2 fully-connected layers, the first of which has a size of 100 and relu activations, and the second of which has a single neuron and no activation. This network has 10277 neurons and 81261 parameters. In addition to this small network, we include the original DAVE network as part of this benchmark to help demonstrate the scalability of analyses. The original DAVE networks has five

convolutional layers with 24, 26, 48, 64, and 64 convolutional kernels, respectively. The first 3 layers use 5x5 kernels, with strides of 2, while the next two use 3x3 kernels with strides of 1. All of the convolutional layers use relu activations and have no padding. The convolutional layers are followed by 5 fully-connected layers with sizes 1164, 100, 50, 10, 1, respectively. The first four of these have relu activations. The original DAVE network has 82669 neurons and 2116983 parameters.

The properties for the Neurify-DAVE benchmark all have the following form: for all inputs within distance ε from input x, the output value must be within 15 degrees of $\mathcal{N}(x)$. Formally, this can be stated as:

$$\forall x'.((x' \in [x-\varepsilon, x+\varepsilon]) \wedge (x' \in \mathcal{X})) \rightarrow (\mathcal{N}(x) - 15^\circ \leq \mathcal{N}(x') \leq \mathcal{N}(x) + 15^\circ)$$

where $\mathcal{X}$ is the appropriate input domain, described above. This benchmark uses $\varepsilon \in \{1, 2, 5, 8, 10\}$ for the original DAVE network, and $\varepsilon \in \{\frac{1}{255}, \frac{2}{255}, \frac{5}{255}, \frac{8}{255}, \frac{10}{255}\}$ for the small network to adjust for input domain.

C.3 GHPR Problem Benchmark

The global halfspace-polytope reachability (GHPR) problem benchmark, is made up of 2 sets of properties, one of which is defined over MNIST networks, and one of which is defined over the DroNet network. Each property sets consists of 10 properties. Within the benchmark, the 10 MNIST properties are each applied to 2 networks, drawn from benchmark used for the ERAN verifier [115]. We chose to use a small convolutional network and a medium convolutional network, both with relu activations. The 10 DroNet properties are applied to the DroNet network [82], which has a ResNet based architecture. The properties are described in more detail below.

C.3.1 MNIST

The networks used as part of the GHPR-MNIST benchmark are the `convSmallRELU__Point.pyt` and `convMedGRELU__Point.pyt` models from the ERAN-MNIST benchmark[1]. The small network has 2 convolutional layers with 16 and 32 4x4 kernels respectively, and strides of 2 and no padding. The convolutional layers are followed by 2 fully-connected layers with dimensions 100 and 10, respectively. The network has 4398 neurons and 89608 parameters. The medium network has 2 convolutional layers with 16 and 32 4x4 kernels respectively, and strides of 2 and uses zero padding. The convo-

[1] Available at `https://github.com/eth-sri/eran#neural-networks-and-datasets`

lutional layers are followed by 2 fully-connected layers with dimensions 1000 and 10, respectively. The network has 6498 neurons and 1587508 parameters.

The MNIST properties are of the form: for all inputs, the output values for classes a and b are closer to one another than either is to the output value of class c. The values of a, b, and c are selected from the confusion matrix of the medium convolutional network on the MNIST test set, shown in Table C.1 with the diagonal values removed. We select the 10 pairs of a and b with the most confusion. We then select a value for c, such that images of digit a were never classified as c, and images of digit b were never classified as c. As an example, we would select 4 and 9 for a and b, since images of fours were classified as nines 13 times, more than any other pair. We then select the value 8 for c, since no images of fours or nines were ever misclassified as eights. This results in 10 properties, defined formally below.

	Predicted Label									
True Label	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	*	1	1	0	1	0	0	0	2	1
1	0	*	1	3	0	1	0	0	0	0
2	1	2	*	1	0	0	1	2	0	0
3	0	0	0	*	0	1	0	2	1	4
4	0	0	1	0	*	0	4	2	0	13
5	2	0	1	10	0	*	1	1	1	1
6	7	3	0	1	2	3	*	0	0	0
7	1	4	7	1	0	0	0	*	1	3
8	4	0	5	10	0	4	0	2	*	5
9	2	3	0	2	4	2	0	3	0	*

Table C.1: The confusion matrix of the medium convolutional DNN on the MNIST test set.

Property ϕ_0.

$$\forall x.(x \in [0,1]^n) \rightarrow ((|\mathcal{N}(x)_4 - \mathcal{N}(x)_9| < |\mathcal{N}(x)_4 - \mathcal{N}(x)_8|) \\ \wedge (|\mathcal{N}(x)_4 - \mathcal{N}(x)_9| < |\mathcal{N}(x)_9 - \mathcal{N}(x)_8|))$$

Property ϕ_1.

$$\forall x.(x \in [0,1]^n) \rightarrow ((|\mathcal{N}(x)_3 - \mathcal{N}(x)_8| < |\mathcal{N}(x)_3 - \mathcal{N}(x)_1|) \\ \wedge (|\mathcal{N}(x)_3 - \mathcal{N}(x)_8| < |\mathcal{N}(x)_8 - \mathcal{N}(x)_1|))$$

Property ϕ_2.

$$\forall x.(x \in [0,1]^n) \rightarrow ((|\mathcal{N}(x)_5 - \mathcal{N}(x)_3| < |\mathcal{N}(x)_5 - \mathcal{N}(x)_4|) \\ \wedge(|\mathcal{N}(x)_5 - \mathcal{N}(x)_3| < |\mathcal{N}(x)_3 - \mathcal{N}(x)_4|))$$

Property ϕ_3.

$$\forall x.(x \in [0,1]^n) \rightarrow ((|\mathcal{N}(x)_6 - \mathcal{N}(x)_0| < |\mathcal{N}(x)_6 - \mathcal{N}(x)_7|) \\ \wedge(|\mathcal{N}(x)_6 - \mathcal{N}(x)_0| < |\mathcal{N}(x)_0 - \mathcal{N}(x)_7|))$$

Property ϕ_4.

$$\forall x.(x \in [0,1]^n) \rightarrow ((|\mathcal{N}(x)_7 - \mathcal{N}(x)_2| < |\mathcal{N}(x)_7 - \mathcal{N}(x)_5|) \\ \wedge(|\mathcal{N}(x)_7 - \mathcal{N}(x)_2| < |\mathcal{N}(x)_2 - \mathcal{N}(x)_5|))$$

Property ϕ_5.

$$\forall x.(x \in [0,1]^n) \rightarrow ((|\mathcal{N}(x)_8 - \mathcal{N}(x)_9| < |\mathcal{N}(x)_8 - \mathcal{N}(x)_6|) \\ \wedge(|\mathcal{N}(x)_8 - \mathcal{N}(x)_9| < |\mathcal{N}(x)_9 - \mathcal{N}(x)_6|))$$

Property ϕ_6.

$$\forall x.(x \in [0,1]^n) \rightarrow ((|\mathcal{N}(x)_8 - \mathcal{N}(x)_2| < |\mathcal{N}(x)_8 - \mathcal{N}(x)_4|) \\ \wedge(|\mathcal{N}(x)_8 - \mathcal{N}(x)_2| < |\mathcal{N}(x)_2 - \mathcal{N}(x)_4|))$$

Property ϕ_7.

$$\forall x.(x \in [0,1]^n) \rightarrow ((|\mathcal{N}(x)_7 - \mathcal{N}(x)_1| < |\mathcal{N}(x)_7 - \mathcal{N}(x)_6|) \\ \wedge(|\mathcal{N}(x)_7 - \mathcal{N}(x)_1| < |\mathcal{N}(x)_1 - \mathcal{N}(x)_6|))$$

Property ϕ_8.

$$\forall x.(x \in [0,1]^n) \rightarrow ((|\mathcal{N}(x)_3 - \mathcal{N}(x)_9| < |\mathcal{N}(x)_3 - \mathcal{N}(x)_2|) \\ \wedge(|\mathcal{N}(x)_3 - \mathcal{N}(x)_9| < |\mathcal{N}(x)_9 - \mathcal{N}(x)_2|))$$

Property ϕ_9.

$$\forall x.(x \in [0,1]^n) \rightarrow ((|\mathcal{N}(x)_8 - \mathcal{N}(x)_5| < |\mathcal{N}(x)_8 - \mathcal{N}(x)_4|) \\ \wedge(|\mathcal{N}(x)_8 - \mathcal{N}(x)_5| < |\mathcal{N}(x)_5 - \mathcal{N}(x)_4|))$$

C.3.2 DroNet

The network used for the GHPR-DroNet benchmark is the DroNet network[2] [82] for autonomous quadrotor control. This network is based on a ResNet type architecture, with 3 residual blocks. It is comprised of 475131 neurons and 320226 parameters.

The properties for DroNet codify the desired behavior that, if the probability for collision is low, the system should not make sharp turns. The DroNet properties are of the form: for all inputs, if the probability of collision is between p_{min} and p_{max}, then the steering angle is within d degrees of 0.

Property ϕ_0.

$$\forall x.((x \in [0,1]^n) \wedge (0 < \mathcal{N}(x)_P \leq 0.1)) \rightarrow (-5^\circ \leq \mathcal{N}(x)_S \leq 5^\circ)$$

Property ϕ_1.

$$\forall x.((x \in [0,1]^n) \wedge (0.1 < \mathcal{N}(x)_P \leq 0.2)) \rightarrow (-10^\circ \leq \mathcal{N}(x)_S \leq 10^\circ)$$

Property ϕ_2.

$$\forall x.((x \in [0,1]^n) \wedge (0.2 < \mathcal{N}(x)_P \leq 0.3)) \rightarrow (-20^\circ \leq \mathcal{N}(x)_S \leq 20^\circ)$$

Property ϕ_3.

$$\forall x.((x \in [0,1]^n) \wedge (0.3 < \mathcal{N}(x)_P \leq 0.4)) \rightarrow (-30^\circ \leq \mathcal{N}(x)_S \leq 30^\circ)$$

Property ϕ_4.

$$\forall x.((x \in [0,1]^n) \wedge (0.4 < \mathcal{N}(x)_P \leq 0.5)) \rightarrow (-40^\circ \leq \mathcal{N}(x)_S \leq 40^\circ)$$

[2] `https://github.com/uzh-rpg/rpg_public_dronet`

Property ϕ_5.

$$\forall x.((x \in [0,1]^n) \wedge (0.5 < \mathcal{N}(x)_P \leq 0.6)) \rightarrow (-50^\circ \leq \mathcal{N}(x)_S \leq 50^\circ)$$

Property ϕ_6.

$$\forall x.((x \in [0,1]^n) \wedge (0.6 < \mathcal{N}(x)_P \leq 0.7)) \rightarrow (-60^\circ \leq \mathcal{N}(x)_S \leq 60^\circ)$$

Property ϕ_7.

$$\forall x.((x \in [0,1]^n) \wedge (0.7 < \mathcal{N}(x)_P \leq 0.8)) \rightarrow (-70^\circ \leq \mathcal{N}(x)_S \leq 70^\circ)$$

Property ϕ_8.

$$\forall x.((x \in [0,1]^n) \wedge (0.8 < \mathcal{N}(x)_P \leq 0.9)) \rightarrow (-80^\circ \leq \mathcal{N}(x)_S \leq 80^\circ)$$

Property ϕ_9.

$$\forall x.((x \in [0,1]^n) \wedge (0.9 < \mathcal{N}(x)_P \leq 1.0)) \rightarrow (-90^\circ \leq \mathcal{N}(x)_S \leq 90^\circ)$$

C.4 CIFAR-EQ Property Benchmark

The CIFAR-EQ problem benchmark, is made up of a set of 291 equivalence properties defined over 2 networks trained on CIFAR10. The benchmark has a 91 global equivalence properties, the first of which is an untargeted equivalence property specifying that the two networks must predict the same class for every input.

$$\forall x.(x \in [0,1]^n) \rightarrow (argmax(\mathcal{N}_1(x)) = argmax(\mathcal{N}_2(x)))$$

The other 90 properties are targeted equivalence properties, specifying that if the first network predicts class A, then the second network cannot predict class B, and vice versa, where A and B are different classes. We create a property for each possible pair of output classes for a total of 90 properties.

$$\forall x.(x \in [0,1]^n) \rightarrow ((argmax(\mathcal{N}_1(x)) \neq A) \vee (argmax(\mathcal{N}_2(x)) \neq B))$$

The next 200 properties are local properties, created from the first 10 images from the CIFAR10 test set that are correctly classified by both networks. Each local property is specified with an L_∞ ϵ-ball around the original input. In this work, we use the epsilon values of $\frac{1}{255}$ and $\frac{10}{255}$. The first 20 properties are untargeted equivalence properties, specifying that all inputs within the ϵ-ball are classified as the same class by both networks. This results in 20 properties, 2 for each of the 10 inputs.

$$\forall x'.(x' \in [x-\epsilon, x+\epsilon]^n) \rightarrow (argmax(\mathcal{N}_1(x')) = argmax(\mathcal{N}_2(x')))$$

The final 180 properties are targeted equivalence properties, specifying that if either network classifies the input to the correct class, C, then the other network should not classify it as class D, different from the correct class. This results in 180 properties, 18 for each of the 10 inputs.

$$\begin{aligned}\forall x'.(x' \in [x-\epsilon, x+\epsilon]^n) \rightarrow (((argmax(\mathcal{N}_1(x')) = C) \rightarrow (argmax(\mathcal{N}_2(x')) \neq D)) \\ \wedge\ ((argmax(\mathcal{N}_2(x')) = C) \rightarrow (argmax(\mathcal{N}_1(x')) \neq D)))\end{aligned}$$

Appendix D

Benchmarks for Analysis of DNNV

We examine the benchmarks used to evaluate each of the 13 verifiers supported by DNNV, and determine whether each verifier can run on the benchmark out of the box, and also whether they could be run on the benchmark when DNNV is applied. Here we provide a short description of each of the 19 verification benchmarks that we have identified. A short summary of some of the features of each verifier relevant to DNNV are shown in Table 5.1. These features include whether any properties cannot represent their input constraints using hyper-rectangles (¬HR), whether any network in the benchmark contains convolution operations (C), whether any network contains residual structures (R), and whether any network uses any non-ReLU activation functions (¬ReLU).

The ACAS Xu (AX) benchmark, introduced for *Reluplex* [68], is one of the most used verification benchmarks [10, 20, 69, 135]. The benchmark consists of 10 properties. Property ϕ_1 is a reachability property, specifying an upper bound on one of the 5 output variables. Properties ϕ_5, ϕ_6, ϕ_9, and ϕ_{10} are all traditional class robustness properties, specifying the desired class for the given input region. Properties ϕ_3, ϕ_4, ϕ_7 and ϕ_8 are reachability properties, specifying a set of acceptable classes for the input region. Properties ϕ_2 is also a reachability property, specifying that a given output value cannot be greater than all others. Each of the properties are applied to a subset of 45 networks trained on an aircraft collision avoidance dataset, with 5 inputs, 5 output classes and 6 layers of 50 neurons each. The original benchmark included networks in *Reluplex*-NNET format, and a custom version of *Reluplex* was written for each property. Later uses of the benchmark translated the verification problems into RLV format, which is used by *Planet*, *BaB*, and *BaBSB*, as well as translating the networks into ONNX. The benchmark in ONNX and DNNP format is fully supported by DNNP.

The Collision Detection (CD) benchmark [34], intoduced for the evaluation of *Planet*, consists of 500 local robustness properties for an 80 neuron network with a fully connected layer and max pooling layer that classifies whether 2 simulated vehicles will collide, given their current state. The verification problems, in RLV format, are supported by *Planet*, *BaB*, and *BaBSB*. The problems have also been modified to convert max pooling operations to a sequence of fully-connected layers with ReLU activations, and then translated to *Reluplex*-NNET format, enabling off the shelf support by *Marabou*, and a generalized version of *Reluplex*. This benchmark is one of the few that is not supported by R4V, since the network contains structures that are not easily supported by ONNX. In particular, the max-pooling operation in the original network, applied to a flat tensor, cannot be encoded by ONNX from their original format.

The *Planet* MNIST (PM) benchmark [34] is a set of 7 properties over a convolutional network trained on the MNIST dataset [79]. The first 4 of these are reachability properties with hyper-rectangle input constraints, while the next 2 are local robustness properties with hyper-rectangle input constraints, and the final property is an local robustness property with halfspace-polytope input constraints. The original benchmark was provided in RLV format. The first 6 of these properties are currently supported by R4V, while the final property could be supported by R4V with additional engineering effort.

The TwinStream (TS) benchmark [19] consists of 1 property applied to 81 networks that output a constant value. The property asserts that for all inputs, the output of the network is positive. The original benchmark was provided in RLV format. This benchmark is fully supported by R4V for all verifiers.

The PCAMNIST (PCA) benchmark [20] consists of 12 reachability properties applied to 17 networks trained on modified versions of the MNIST dataset to predict the parity of the digit represented by the first k components of the PCA decomposition of an image. The original benchmark was provided in RLV format. This benchmark is fully supported by R4V for all verifiers.

MIPVerify MNIST (MM) consists of 10000 local robustness properties applied to 5 networks trained on the MNIST dataset. The networks have varied structures: 2 networks are fully connected and 3 are convolutional. We could not find an available version of the benchmark used by *MIPVerify* to evaluate its original input format. This benchmark is fully supported by R4V for all verifiers except *Reluplex*, which does not support convolution operations.

MIPVerify CIFAR (MC) consists of 10000 local robustness properties applied to 2 networks trained on the CIFAR10 dataset [75]. One of these networks is a convolutional network and the other is a residual network. We could not find an available version of the benchmark used by

MIPVerify to evaluate its original input format. This benchmark is supported by R4V for verifiers that can support residual connections, including: *Planet*, *DeepZono*, *DeepPoly*, *RefineZono*, and *RefinePoly*. While the benchmark is supported by the version of *MIPVerify* used in its study, it is not supported through R4V, since the publicly available version of *MIPVerify* does not support residual connections.

The *Neurify* MNIST (NM) benchmark [135] consists of 500 L_∞ local robustness properties across 4 MNIST networks, 3 fully connected networks with 58, 110, and 1034 neurons respectively, and a convolutional network with 4814 neurons. The original benchmark was provided in *Neurify*-NNET format, with properties hard-coded into the verifier. R4V enables almost all verifiers to run on this benchmark. *Reluplex* cannot be run due to the presence of convolutional layers, which are not supported. *MIPVerify* cannot be run due to the presence of non-hypercube input constraints. While this limitation of the verifier can be satisfied with a property reduction for fully-connected networks, R4V does not currenly support such a reduction for convolutional networks.

The *Neurify* Drebin (NDB) benchmark [135] consists of 500 L_∞ local robustness properties across 3 fully connected Drebin [6] networks with 102, 212, and 402 neurons each. The original benchmark was provided in *Neurify*-NNET format, with properties hard-coded into the verifier. This benchmark is fully supported by R4V for all verifiers.

The *Neurify* DAVE (NDV) benchmark [135] consists of 200 local reachability properties, with 4 different types of input constraints (50 properties of each type). The first type of input constraint is an L_∞ constraint, which is equivalent to a hyper-rectangle constraint. The second type of input constraint is an L_1 constraint, which can be written as a halfspace polytope constraint. The third and fourth type of input constraint are image brightning and contrast, which can be written as halfspace polytope constraints. The properties are applied to a convolutional network for an autonomous vehicle, with 10276 neurons. The original benchmark was provided in *Neurify*-NNET format, with properties hard-coded into the verifier. Similar to the *Neurify* MNIST benchmark, R4V enables almost all verifiers to run on this benchmark. *Reluplex* cannot be run, due to the presence of convolutional layers, which are not supported, and *MIPVerify* cannot be run due to the presence of non-hypercube input constraints.

The *DeepZono* MNIST (DZM) benchmark [115] consists of 1700 local robustness properties, subsets of which are applied to 10 networks trained on the MNIST dataset. The networks have varied structures and activation functions: 3 networks are fully connected, 1 of which uses ReLU activations, 1 with Tanh activations, and 1 with Sigmoid activations; 6 are convolutional, 4 of which have ReLU activations, 1 with Tanh activations, and 1 with Sigmoid activations; and 1 is a residual

network. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. R4V does not increase the support for this benchmark due to the presence of both a residual network and non-ReLU activation functions.

The *DeepZono* CIFAR10 (DZC) benchmark [115] consists of 1700 local robustness properties, subsets of which are applied to 5 networks trained on the CIFAR10 dataset. The networks have varied structures and activation functions: 3 networks are fully connected, 1 of which uses ReLU activations, 1 with Tanh activations, and 1 with Sigmoid activations; and 2 are convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. R4V enables *VeriNet* to run on this benchmark. Other verifiers are not supported due to the non-ReLU activation functions.

The *DeepPoly* MNIST (DPM) benchmark [116] consists of 1500 local robustness properties, subsets of which are applied to 8 networks trained on the MNIST dataset. The networks have varied structures and activation functions: 5 networks are fully connected, 3 of which uses ReLU activations, 1 with Tanh activations, and 1 with Sigmoid activations; and 3 are convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. R4V enables *VeriNet* to run on this benchmark. Other verifiers are not supported due to the non-ReLU activation functions.

The *DeepPoly* CIFAR10 (DPC) benchmark [116] consists of 800 local robustness properties, subsets of which are applied to 5 networks trained on the CIFAR10 dataset. The networks have varied structures: 3 networks are fully connected with ReLU activations; and 2 are convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. R4V enables several additional verifiers to support this benchmark. In particular, it enables most verifiers that can be applied to convolutional networks with relu activations.

The *RefineZono* MNIST (RZM) benchmark [117] consists of 800 local robustness properties, subsets of which are applied to 8 networks trained on the MNIST dataset. 5 networks are fully connected with ReLU activations and 3 are convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. R4V enables several additional verifiers to support this benchmark. In particular, it enables most verifiers that can be applied to convolutional networks with relu activations.

The *RefineZono* CIFAR10 (RZC) benchmark [117] consists of 200 local robustness properties, subsets of which are applied to 2 networks trained on the CIFAR10 dataset. One of the networks is

fully connected with ReLU activations and the other is convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. R4V enables several additional verifiers to support this benchmark. In particular, it enables most verifiers that can be applied to convolutional networks with relu activations.

The *RefinePoly* MNIST (RPM) benchmark [114] consists of 600 local robustness properties, subsets of which are applied to 6 networks trained on the MNIST dataset. 4 networks are fully connected with ReLU activations and 2 are convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. R4V enables several additional verifiers to support this benchmark. In particular, it enables most verifiers that can be applied to convolutional networks with relu activations.

The *RefinePoly* CIFAR10 (RPC) benchmark [114] consists of 300 local robustness properties, subsets of which are applied to 3 networks trained on the MNIST dataset. Two of the networks are convolutional with ReLU activations and the third is a residual network with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. R4V enables the *Planet* verifier to support this benchmark. In particular, it enables most verifiers that can be applied to convolutional networks with relu activations. Other verifiers do not support the residual structure of one of the networks.

The *VeriNet* CIFAR10 (VC) benchmark [49] consists of 250 local robustness properties applied to 1 convolutional network with ReLU activations. The networks were provided in ONNX format, with hard-coded properties. R4V enables support of this benchmark by most of the integrated verifiers. *Reluplex* does not support convolutional networks, and *MIPVerify* does not support properties with input constraints that are not hyper-cubes.

Appendix E

Correctness Problem Benchmarks for Chapter 6

In this section we describe the benchmarks used in our study of DFV.

E.1 FashionMNIST

E.1.1 Properties

The properties for FashionMNIST consists in comparing different pieces of clothes in a way that the difference between clothes with similar shapes are smaller that others with different shapes. E.g. the difference between a t-shirt/top and a shirt should be smaller than the difference between a t-shirt/top and a sneaker. There are two types of properties:

Conditional output relational (COR) properties specify a relation between output values and require that the predicted output class must be one of the output classes being compared.

Property $\phi_{COR,0}$.

$$\forall x.((x \in [0,1]^n) \wedge (argmax(\mathcal{N}(x)) = 7)) \rightarrow \\ (|\mathcal{N}(x)_7 - \mathcal{N}(x)_6| > |\mathcal{N}(x)_7 - \mathcal{N}(x)_5|)$$

Property $\phi_{COR,1}$.

$$\forall x.((x \in [0,1]^n) \wedge (argmax(\mathcal{N}(x)) = 6)) \rightarrow (|\mathcal{N}(x)_6 - \mathcal{N}(x)_9| > |\mathcal{N}(x)_6 - \mathcal{N}(x)_2|)$$

Property $\phi_{COR,2}$.

$$\forall x.((x \in [0,1]^n) \wedge (argmax(\mathcal{N}(x)) = 5)) \rightarrow (|\mathcal{N}(x)_5 - \mathcal{N}(x)_8| > |\mathcal{N}(x)_5 - \mathcal{N}(x)_7|)$$

Property $\phi_{COR,3}$.

$$\forall x.((x \in [0,1]^n) \wedge (argmax(\mathcal{N}(x)) = 4)) \rightarrow (|\mathcal{N}(x)_4 - \mathcal{N}(x)_1| > |\mathcal{N}(x)_4 - \mathcal{N}(x)_6|)$$

Property $\phi_{COR,4}$.

$$\forall x.((x \in [0,1]^n) \wedge (argmax(\mathcal{N}(x)) = 3)) \rightarrow (|\mathcal{N}(x)_3 - \mathcal{N}(x)_7| > |\mathcal{N}(x)_3 - \mathcal{N}(x)_0|)$$

Property $\phi_{COR,5}$.

$$\forall x.((x \in [0,1]^n) \wedge (argmax(\mathcal{N}(x)) = 9)) \rightarrow (|\mathcal{N}(x)_9 - \mathcal{N}(x)_0| > |\mathcal{N}(x)_9 - \mathcal{N}(x)_7|)$$

Property $\phi_{COR,6}$.

$$\forall x.((x \in [0,1]^n) \wedge (argmax(\mathcal{N}(x)) = 2)) \rightarrow (|\mathcal{N}(x)_2 - \mathcal{N}(x)_1| > |\mathcal{N}(x)_2 - \mathcal{N}(x)_4|)$$

Property $\phi_{COR,7}$.

$$\forall x.((x \in [0,1]^n) \wedge (argmax(\mathcal{N}(x)) = 5)) \rightarrow (|\mathcal{N}(x)_5 - \mathcal{N}(x)_2| > |\mathcal{N}(x)_5 - \mathcal{N}(x)_9|)$$

Property $\phi_{COR,8}$.

$$\forall x.((x \in [0,1]^n) \wedge (argmax(\mathcal{N}(x)) = 0)) \rightarrow \\ (|\mathcal{N}(x)_0 - \mathcal{N}(x)_8| > |\mathcal{N}(x)_0 - \mathcal{N}(x)_6|)$$

Property $\phi_{COR,9}$.

$$\forall x.((x \in [0,1]^n) \wedge (argmax(\mathcal{N}(x)) = 1)) \rightarrow \\ (|\mathcal{N}(x)_1 - \mathcal{N}(x)_7| > |\mathcal{N}(x)_1 - \mathcal{N}(x)_3|)$$

Unconditional output relational (UOR) properties do not require a specific output class but do specify a relationship between output values.

Property $\phi_{UOR,0}$.

$$\forall x.(x \in [0,1]^n) \rightarrow (|\mathcal{N}(x)_7 - \mathcal{N}(x)_6| > |\mathcal{N}(x)_7 - \mathcal{N}(x)_5|)$$

Property $\phi_{UOR,1}$.

$$\forall x.(x \in [0,1]^n) \rightarrow (|\mathcal{N}(x)_6 - \mathcal{N}(x)_9| > |\mathcal{N}(x)_6 - \mathcal{N}(x)_2|)$$

Property $\phi_{UOR,2}$.

$$\forall x.(x \in [0,1]^n) \rightarrow (|\mathcal{N}(x)_5 - \mathcal{N}(x)_8| > |\mathcal{N}(x)_5 - \mathcal{N}(x)_7|)$$

Property $\phi_{UOR,3}$.

$$\forall x.(x \in [0,1]^n) \rightarrow (|\mathcal{N}(x)_4 - \mathcal{N}(x)_1| > |\mathcal{N}(x)_4 - \mathcal{N}(x)_6|)$$

Property $\phi_{UOR,4}$.

$$\forall x.(x \in [0,1]^n) \rightarrow (|\mathcal{N}(x)_3 - \mathcal{N}(x)_7| > |\mathcal{N}(x)_3 - \mathcal{N}(x)_0|)$$

Property $\phi_{UOR,5}$.

$$\forall x.(x \in [0,1]^n) \rightarrow (|\mathcal{N}(x)_7 - \mathcal{N}(x)_2| > |\mathcal{N}(x)_7 - \mathcal{N}(x)_9|)$$

Property $\phi_{UOR,6}$.

$$\forall x.(x \in [0,1]^n) \rightarrow (|\mathcal{N}(x)_6 - \mathcal{N}(x)_5| > |\mathcal{N}(x)_6 - \mathcal{N}(x)_4|)$$

Property $\phi_{UOR,7}$.

$$\forall x.(x \in [0,1]^n) \rightarrow (|\mathcal{N}(x)_5 - \mathcal{N}(x)_1| > |\mathcal{N}(x)_5 - \mathcal{N}(x)_7|)$$

Property $\phi_{UOR,8}$.

$$\forall x.(x \in [0,1]^n) \rightarrow (|\mathcal{N}(x)_4 - \mathcal{N}(x)_8| > |\mathcal{N}(x)_4 - \mathcal{N}(x)_2|)$$

Property $\phi_{UOR,9}$.

$$\forall x.(x \in [0,1]^n) \rightarrow (|\mathcal{N}(x)_3 - \mathcal{N}(x)_9| > |\mathcal{N}(x)_3 - \mathcal{N}(x)_0|)$$

E.2 DroNet

E.2.1 Network

The network used for the GHPR-DroNet benchmark is the DroNet network[1] [82] for autonomous quadrotor control. This network is based on a ResNet type architecture, with 3 residual blocks. It is comprised of 475131 neurons and 320226 parameters.

E.2.2 Properties

The properties for DroNet codify the desired behavior that, if the probability for collision is low, the system should not make sharp turns. The DroNet properties are of the form: for all inputs, if the probability of collision is between p_{min} and p_{max}, then the steering angle is within d degrees of 0.

[1] `https://github.com/uzh-rpg/rpg_public_dronet`

Property ϕ_0.

$$\forall x.((x \in [0,1]^n) \wedge (0 < \mathcal{N}(x)_P \leq 0.1)) \rightarrow (-5^\circ \leq \mathcal{N}(x)_S \leq 5^\circ)$$

Property ϕ_1.

$$\forall x.((x \in [0,1]^n) \wedge (0.1 < \mathcal{N}(x)_P \leq 0.2)) \rightarrow (-10^\circ \leq \mathcal{N}(x)_S \leq 10^\circ)$$

Property ϕ_2.

$$\forall x.((x \in [0,1]^n) \wedge (0.2 < \mathcal{N}(x)_P \leq 0.3)) \rightarrow (-20^\circ \leq \mathcal{N}(x)_S \leq 20^\circ)$$

Property ϕ_3.

$$\forall x.((x \in [0,1]^n) \wedge (0.3 < \mathcal{N}(x)_P \leq 0.4)) \rightarrow (-30^\circ \leq \mathcal{N}(x)_S \leq 30^\circ)$$

Property ϕ_4.

$$\forall x.((x \in [0,1]^n) \wedge (0.4 < \mathcal{N}(x)_P \leq 0.5)) \rightarrow (-40^\circ \leq \mathcal{N}(x)_S \leq 40^\circ)$$

Property ϕ_5.

$$\forall x.((x \in [0,1]^n) \wedge (0.5 < \mathcal{N}(x)_P \leq 0.6)) \rightarrow (-50^\circ \leq \mathcal{N}(x)_S \leq 50^\circ)$$

Property ϕ_6.

$$\forall x.((x \in [0,1]^n) \wedge (0.6 < \mathcal{N}(x)_P \leq 0.7)) \rightarrow (-60^\circ \leq \mathcal{N}(x)_S \leq 60^\circ)$$

Property ϕ_7.

$$\forall x.((x \in [0,1]^n) \wedge (0.7 < \mathcal{N}(x)_P \leq 0.8)) \rightarrow$$
$$(-70^\circ \leq \mathcal{N}(x)_S \leq 70^\circ)$$

Property ϕ_8.

$$\forall x.((x \in [0,1]^n) \wedge (0.8 < \mathcal{N}(x)_P \leq 0.9)) \rightarrow$$
$$(-80^\circ \leq \mathcal{N}(x)_S \leq 80^\circ)$$

Property ϕ_9.

$$\forall x.((x \in [0,1]^n) \wedge (0.9 < \mathcal{N}(x)_P \leq 1.0)) \rightarrow$$
$$(-90^\circ \leq \mathcal{N}(x)_S \leq 90^\circ)$$

Appendix F

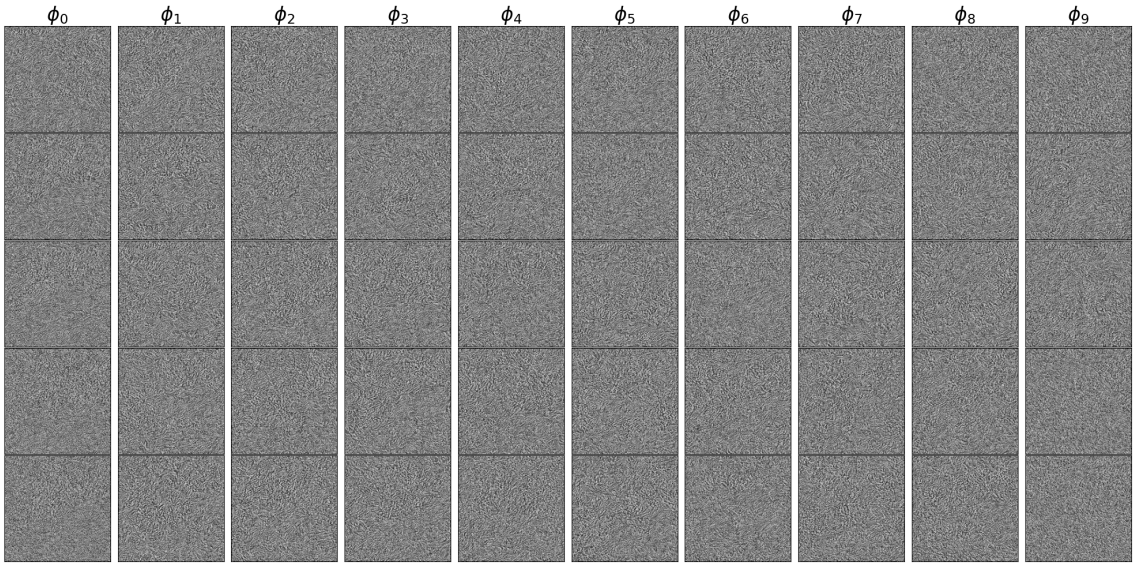
Additional Experimental Data for Chapter 6

In this section we provide some additional data for the experiments conducted in our study of DFV.

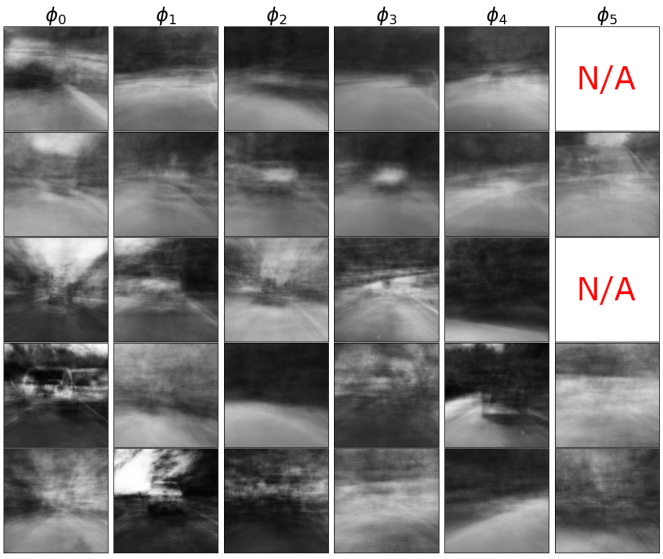
F.1 Additional Data: RQ2 - Scalability

All counter-examples found for the DroNet benchmark in the experiment for RQ2 in Section 6.2.3 are shown in Figure F.1. Each subfigure represents 1 of the treatments: falsification without DFV (Figure F.1a), with DFV using a VAE (Figure F.1b), and with DFV using a GAN (Figure F.1c). We omit columns for the treatments using DFV which did not return any counter-examples to save space and fit the figure on a single page.

Figure F.1 demonstrates the ability of DFV to produce inputs closer to the data distribution by using a model of the environment, even when the network under test and environment models are quite large. Without DFV, the PGD falsifier finds a counter-example for every property on every run. However, these counter-examples are likely useless to testers and developers, as they are far from the data distribution on which the network would be expected to operate correctly, and they provide little insight into a potential fault.



(a) Without DFV



(b) With DFV, using FC-VAE_{DroNet}

(c) With DFV, using GAN_{DroNet}

Figure F.1: Counter-examples to DroNet properties with 3 distinct input models. Each row of a grid corresponds to 1 of 5 runs of the falsifier, and each column corresponds to 1 of the 10 properties. When applied in conjunction with DFV, whether using a VAE or a GAN, the generated counter-examples visually appear to be much better aligned with the training distribution.

Appendix G

Benchmarks for Closed-Loop Problem Rewriting

This section describes the benchmarks used in our evaluation. The benchmarks are originally from the 2020 and 2021 ARCH-COMP AINNCS competition for closed-loop verification tools [60, 61]. We provide definitions of the problems, both with continuous and discrete time environment models. Our paper and much of this appendix makes use of short names for the benchmarks, and a mapping from short names to full names, as used in AINNCS, is provided in Table G.1.

G.1 ACC: Adaptive Cruise Controller

This benchmark models an adaptive cruise control system in which the ego vehicle has a target velocity, but must also maintain a safe distance behind a lead vehicle. The state of this system can

Table G.1: Benchmark Names

Short Name	**Full Name**
ACC	Adaptive Cruise Controller
AP	Airplane
DPLR	Double Pendulum (Less Robust)
DPMR	Double Pendulum (More Robust)
SB9	Sherlock-Benchmark-9 (TORA)
SB10	Sherlock-Benchmark-10 (Unicycle Car)
SP	Single Pendulum
VCAS	VCAS

be represented by the vector $[x_{lead}, v_{lead}, \gamma_{lead}, x_{ego}, v_{ego}, \gamma_{ego}]$, where x_{lead}, v_{lead}, and γ_{lead} are the position, velocity, and acceleration of the lead vehicle, and x_{ego}, v_{ego}, and γ_{ego} are the position, velocity, and acceleration of the ego vehicle.

The continuous time dynamics are defined by:

$$\begin{aligned}
\dot{x}_{lead} &= v_{lead} \\
\dot{v}_{lead} &= \gamma_{lead} \\
\dot{\gamma}_{lead} &= -2\gamma_{lead} + 2a_{lead} - uv_{lead}^2 \\
\dot{x}_{ego} &= v_{ego} \\
\dot{v}_{ego} &= \gamma_{ego} \\
\dot{\gamma}_{ego} &= -2\gamma_{ego} + 2a_{ego} - uv_{ego}^2
\end{aligned}$$

where $u = 0.0001$ is a friction parameter, and $a_{lead} = -2$ and a_{ego} are the acceleration control inputs of the lead and ego vehicles, respectively.

A discrete model of the environment can be specified as:

$$\begin{aligned}
(x_{lead})_{t+1} &= (x_{lead})_t + (v_{lead})_t \Delta T \\
(v_{lead})_{t+1} &= (v_{lead})_t + (\gamma_{lead})_t \Delta T \\
(\gamma_{lead})_{t+1} &= (\gamma_{lead})_t + (2a_{lead} - 2(\gamma_{lead})_t - 2u(v_{lead})_t)\Delta T \\
(x_{ego})_{t+1} &= (x_{ego})_t + (v_{ego})_t \Delta T \\
(v_{ego})_{t+1} &= (v_{ego})_t + (\gamma_{ego})_t \Delta T \\
(\gamma_{ego})_{t+1} &= (\gamma_{ego})_t + (2(a_{ego})_t - 2(\gamma_{ego})_t - 2u(v_{ego})_t)\Delta T
\end{aligned}$$

, where ΔT is the discrete time step size.

A neural network is used to control the acceleration, a_{ego}, of the ego vehicle. The network takes in a vector $[v_{set}, T_{gap}, v_{ego}, D_{rel}, v_{rel}]$, where $v_{set} = 30$ is the target velocity, $T_{gap} = 1.4$, $D_{rel} = x_{lead} - x_{ego}$ is the distance between the vehicles, and $v_{rel} = v_{lead} - v_{ego}$ is the relative velocity. The network is queried at a control period of 0.1 seconds.

To define the environment in the form $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle$, we require a bounded state space. We choose to use $\mathcal{S} = ([0, 300] \times [0, 100] \times [-20, 20])^2$, which bounds the positions to be positive values less than 300, the velocities to be positive values less than 100, and the accelerations to be between negative and positive 20. While the acceleration can be negative, we assume the velocity will never be negative, which we believe to be a reasonable assumption for a cruise control system, as the

vehicle should always be moving forward. We define $\mathcal{M}$ as:

$$\mathcal{M}(s) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & - & 0 \end{bmatrix} s + \begin{bmatrix} 30 \\ 1.4 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

. We define the transition function, $\mathcal{T}$ as a function that takes in $s \in \mathcal{S}$ and the output of the network, y, and applies the discrete environment model 100 times, with $\Delta T = 0.001$ and $(a_{ego})_t = y$.

Property: Given the environment and network defined above, the system must satisfy the following property:

$$(S_0 \times \mathcal{X} \times \mathcal{Y}) \implies XG_{50}(S_{safe} \times \mathcal{X} \times \mathcal{Y})$$

where the set of initial states is $S_0 = [90, 110] \times [32, 32.2] \times \{0\} \times [10, 11] \times [30, 30.2] \times \{0\}$, and $S_{safe} = \{[x_{lead}, v_{lead}, \gamma_{lead}, x_{ego}, v_{ego}, \gamma_{ego}] \in \mathcal{S} \mid x_{lead} - x_{ego} \geq 10 + 1.4 v_{ego}\}$ is the set of safe states, and $\mathcal{X}$ and $\mathcal{Y}$ are the full input and output sets of the network, respectively.

G.2 AP: Airplane

This benchmark consists of a simple airplane model with a controller that outputs forces, (F_x, F_y, F_z), and moments, (M_x, M_y, M_z), in three dimensions. The state can be modelled by the position of the airplane, (x, y, z), the velocity in each of those dimensions (u, v, w), the body rotation rates, (p, q, r) and Euler angles (ϕ, θ, ψ).

The continuous time dynamics can be specified as:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{z}\end{bmatrix} = A_1 \begin{bmatrix}u\\ v\\ w\end{bmatrix}$$
$$\begin{bmatrix}\dot{u}\\ \dot{v}\\ \dot{w}\end{bmatrix} = \begin{bmatrix}F_x - qw + rw - \sin(\theta)\\ F_y - ru + pw + \cos(\theta)\sin(\phi)\\ F_z - pv + qu + \cos(\theta)\cos(\phi)\end{bmatrix}$$
$$\begin{bmatrix}\dot{\phi}\\ \dot{\theta}\\ \dot{\psi}\end{bmatrix} = A_2 \begin{bmatrix}p\\ q\\ r\end{bmatrix}$$
$$\begin{bmatrix}\dot{p}\\ \dot{q}\\ \dot{r}\end{bmatrix} = \begin{bmatrix}M_x\\ M_y\\ M_z\end{bmatrix}$$

where

$$T_\psi = \begin{bmatrix}\cos(\psi) & -\sin(\psi) & 0\\ \sin(\psi) & \cos(\psi) & 0\\ 0 & 0 & 1\end{bmatrix}$$
$$T_\theta = \begin{bmatrix}\cos(\theta) & 0 & \sin(\theta)\\ 0 & 1 & 0\\ -\sin(\theta) & 0 & \cos(\theta)\end{bmatrix}$$
$$T_\phi = \begin{bmatrix}1 & 0 & 0\\ 0 & \cos(\phi) & -\sin(\phi)\\ 0 & \sin(\phi) & \cos(\phi)\end{bmatrix}$$
$$A_1 = T_\psi T_\theta T_\phi$$
$$A_2 = \begin{bmatrix}1 & \tan(\theta)\sin(\phi) & \tan(\theta)\cos(\phi)\\ 0 & \cos(\phi) & -\sin(\phi)\\ 0 & \sec(\theta)\sin(\phi) & \sec(\theta)\cos(\phi)\end{bmatrix}$$

A discrete time model of the system can be specified as:

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \\ z_{t+1} \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} + ((A_1)_t \begin{bmatrix} u_t \\ v_t \\ w_t \end{bmatrix})\Delta T$$

$$\begin{bmatrix} u_{t+1} \\ v_{t+1} \\ w_{t+1} \end{bmatrix} = \begin{bmatrix} u_t \\ v_t \\ w_t \end{bmatrix} + \begin{bmatrix} (F_x)_t - q_t w_t + r_t w_t - \sin(\theta_t) \\ (F_y)_t - r_t u_t + p_t w_t + \cos(\theta_t)\sin(\phi_t) \\ (F_z)_t - p_t v_t + q_t u_t + \cos(\theta_t)\cos(\phi_t) \end{bmatrix} \Delta T$$

$$\begin{bmatrix} \phi_{t+1} \\ \theta_{t+1} \\ \psi_{t+1} \end{bmatrix} = \begin{bmatrix} \phi_{t+1} \\ \theta_{t+1} \\ \psi_{t+1} \end{bmatrix} + ((A_2)_t \begin{bmatrix} p_t \\ q_t \\ r_t \end{bmatrix})\Delta T$$

$$\begin{bmatrix} p_{t+1} \\ q_{t+1} \\ r_{t+1} \end{bmatrix} = \begin{bmatrix} p_t \\ q_t \\ r_t \end{bmatrix} \begin{bmatrix} (M_x)_t \\ (M_y)_t \\ (M_z)_t \end{bmatrix} \Delta T$$

where

$$(T_\psi)_t = \begin{bmatrix} \cos(\psi_t) & -\sin(\psi_t) & 0 \\ \sin(\psi_t) & \cos(\psi_t) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$(T_\theta)_t = \begin{bmatrix} \cos(\theta_t) & 0 & \sin(\theta_t) \\ 0 & 1 & 0 \\ -\sin(\theta_t) & 0 & \cos(\theta_t) \end{bmatrix}$$

$$(T_\phi)_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi_t) & -\sin(\phi_t) \\ 0 & \sin(\phi_t) & \cos(\phi_t) \end{bmatrix}$$

$$(A_1)_t = (T_\psi)_t (T_\theta)_t (T_\phi)_t$$

$$(A_2)_t = \begin{bmatrix} 1 & \tan(\theta_t)\sin(\phi_t) & \tan(\theta_t)\cos(\phi_t) \\ 0 & \cos(\phi_t) & -\sin(\phi_t) \\ 0 & \sec(\theta_t)\sin(\phi_t) & \sec(\theta_t)\cos(\phi_t) \end{bmatrix}$$

where ΔT is the discrete time step.

To define the environment in the form $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle$, we require a bounded state space. We choose to use $\mathcal{S} = [-25, 25]^3 \times [-20, 20]^3 \times [-\frac{\pi}{2}, \frac{\pi}{2}]^3 \times [-1, 1]^3$, which were estimated by simulating the environment with the controller for 20 timesteps and selecting bounds which overapproximated the visited states. We chose to use the same bounds for semantically similar state variables. We define $\mathcal{M}(s) = s$, since the state and network input representations are the same. We define the transition function, $\mathcal{T}$ as a function that takes in $s \in \mathcal{S}$ and the output of the network, y, and applies the discrete environment model 100 times, with $\Delta T = 0.001$ and $(F_x, F_y, F_z, M_x, M_y, M_z) = y$.

A neural network is used to control the forces and their moments of the airplane. The network takes in a vector $[x, y, z, u, v, w, \phi, \theta, \psi, r, p, q]$ that is the same as the state representation, and outputs 6 control signals, $[F_x, F_y, F_z, M_x, M_y, M_z]$. The network is queried at a control period of 0.1 seconds.

Property: Given the environment and network defined above, the system must satisfy the following property:

$$(S_0 \times \mathcal{X} \times \mathcal{Y}) \implies XG_{20}(S_{safe} \times \mathcal{X} \times \mathcal{Y})$$

where $S_0 = \{0\}^3 \times [0, 1]^6 \times \{0\}^3$, and $S_{safe} = [-\infty, \infty]^6 \times [-1, 1]^3 \times [-\infty, \infty]^3$, and $\mathcal{X}$ and $\mathcal{Y}$ are the full input and output sets of the network, respectively.

G.3 DPLR: Double Pendulum (Less Robust)

This benchmark uses a neural network to control an inverted double (two-link) pendulum, which consists of 2 balls, each of mass $m = 0.5$, attached to the ends of massless rods of length $L = 0.5$. Each rod is actuated with a separate torque u_1 and u_2 for each link. The state can be modelled by the angles of the pendulum, θ_1 and θ_2, from the upward vertical axis and the angular velocities, $\dot{\theta}_1$ and $\dot{\theta}_2$.

The continuous time dynamics are defined by:

$$2\ddot{\theta}_1 + \ddot{\theta}_2 \cos(\theta_2 - \theta_1) - \dot{\theta}_2^2 \sin(\theta_2 - \theta_1) - \frac{2\sin(\theta_1)}{L} = \frac{u_1}{mL^2}$$
$$\ddot{\theta}_2 + \ddot{\theta}_1 \cos(\theta_2 - \theta_1) + \dot{\theta}_1^2 \sin(\theta_2 - \theta_1) - \frac{\sin(\theta_2)}{L} = \frac{u_2}{mL^2}$$

A discrete model of the environment can be specified as:

$$
\begin{aligned}
(\theta_1)_{t+1} &= (\theta_1)_t + (\dot{\theta}_1)_t \Delta T \\
(\theta_2)_{t+1} &= (\theta_2)_t + (\dot{\theta}_2)_t \Delta T \\
(\dot{\theta}_1)_{t+1} &= (\dot{\theta}_1)_t + ((\ddot{\theta}_1)_t) \Delta T \\
(\dot{\theta}_2)_{t+1} &= (\dot{\theta}_2)_t + ((\ddot{\theta}_2)_t) \Delta T \\
&\text{where} \\
(\ddot{\theta}_1)_t &= \frac{(c_{rel})_t (\dot{\theta}_1)_t^2 (s_{rel})_t}{(c_{rel})_t^2 - 0.5} \\
&\quad - \frac{(c_{rel})_t^2 (\frac{(s_1)_t}{L} - \frac{(\dot{\theta}_2)_t^2 (s_{rel})_t}{2} + \frac{(u_1)_t}{2mL^2})}{(c_{rel})_t^2 - 0.5} \\
&\quad + \frac{\frac{(s_2)_t}{L} + \frac{(u_2)_t}{mL^2}}{(c_{rel})_t^2 - 0.5} \\
&\quad - \frac{(\dot{\theta}_2)_t^2 (s_{rel})_t}{2} + \frac{\sin((\theta_1)_t)}{L} + \frac{(u_1)_t}{2mL^2} \\
(\ddot{\theta}_2)_t &= \frac{(\dot{\theta}_1)_t^2 (s_{rel})_t}{0.5((c_{rel})_t^2 - 2)} \\
&\quad \frac{(c_{rel})_t^2 (\frac{(s_1)_t}{L} - \frac{(\dot{\theta}_2)_t^2 (s_{rel})_t}{2} + \frac{(u_1)_t}{2mL^2})}{0.5((c_{rel})_t^2 - 2)} \\
&\quad \frac{\frac{(s_2)_t}{L} + \frac{(u_2)_t}{mL^2}}{0.5((c_{rel})_t^2 - 2)} \\
(c_i)_t &= \cos((\theta_i)_t), \text{where } i \in \{rel, 1, 2\} \\
(s_i)_t &= \sin((\theta_i)_t), \text{where } i \in \{rel, 1, 2\} \\
(\theta_{rel})_t &= (\theta_1)_t - (\theta_2)_t
\end{aligned}
$$

, where ΔT is the discrete time step size.

To define the environment in the form $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle$, we require a bounded state space. We choose to use $\mathcal{S} = [-\pi, \pi]^4$, which allow the angles to cover a full rotation of the pendulum and bounds the angular velocity to reasonably high values. We define $\mathcal{M}(s) = s$, since the state and network input representations are the same. We define the transition function, $\mathcal{T}$ as a function that takes in $s \in \mathcal{S}$ and the output of the network, y, and applies the discrete environment model 100 times, with $\Delta T = 0.0005$ and $u_t = y$.

A neural network is used to control the torque applied to the two pendulums, with the goal of keeping the pendulum inverted, i.e., with $\theta_1 = \theta_2 = 0$. The network takes in a vector $[\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]$ that is the same as the state representation. The network is queried at a control period of 0.05 seconds.

Property: Given the environment and network defined above, the system must satisfy the following property:

$$(S_0 \times \mathcal{X} \times \mathcal{Y}) \implies XG_{20}(S_{safe} \times \mathcal{X} \times \mathcal{Y})$$

where $S_0 = [1, 1.3]^4$, and $S_{safe} = [-1, 1.7]^4$, and $\mathcal{X}$ and $\mathcal{Y}$ are the full input and output sets of the network, respectively.

G.4 DPMR: Double Pendulum (More Robust)

This problem uses the same environment description as DPLR, except with $\Delta T = 0.0002$ in the definition of $\mathcal{T}$.

A neural network is used to control the torque applied to the two pendulums, with the goal of keeping the pendulum inverted, i.e., with $\theta_1 = \theta_2 = 0$. The network takes in a vector $[\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]$ that is the same as the state representation. The network is queried at a control period of 0.02 seconds.

Property: Given the environment and network defined above, the system must satisfy the following property:

$$(S_0 \times \mathcal{X} \times \mathcal{Y}) \implies XG_{20}(S_{safe} \times \mathcal{X} \times \mathcal{Y})$$

where $S_0 = [1, 1.3]^4$, and $S_{safe} = [-0.5, 1.5]^4$, and $\mathcal{X}$ and $\mathcal{Y}$ are the full input and output sets of the network, respectively.

G.5 SB9: Sherlock-Benchmark-9 (TORA)

This benchmark models a system with rotational actuators. The environment consists of a cart on a frictionless surface, which is attached to a wall with a spring. The cart contains a weight attached to an arm, which can be rotated to move the cart. A network is used to control the actuation of this rotating arm.

The continous time dynamics can be defined by:

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = 0.1\sin(x_3) - x_1$$
$$\dot{x}_3 = x_4$$
$$\dot{x}_4 = u$$

, where u is the control signal.

The discrete time system can be specified as:

$$(x_1)_{t+1} = (x_1)_t + (x_2)_t \Delta T$$
$$(x_2)_{t+1} = (x_2)_t + (0.1\sin((x_3)_t) - (x_1)_t)\Delta T$$
$$(x_3)_{t+1} = (x_3)_t + (x_4)_t \Delta T$$
$$(x_4)_{t+1} = (x_4)_t + u_t \Delta T$$

, where ΔT is the discrete time step size.

To define the environment in the form $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle$, we require a bounded state space. We choose to use $\mathcal{S} = [-3, 3]^4$. We define $\mathcal{M}(s) = s$, since the state and network input representations are the same. We define the transition function, $\mathcal{T}$ as a function that takes in $s \in \mathcal{S}$ and the output of the network, y, and applies the discrete environment model 100 times, with $\Delta T = 0.01$ and $u_t = y$.

A neural network is used to control the actuation of the rotating arm, with the goal of stabilizing the cart at $x = 0$. The network takes in a vector $x = [x_1, x_2, x_3, x_4]$ that is the same as the state representation. The output of the network must be normalized by subtracting a value of 10, i.e., $u = \mathcal{N}(x) - 10$. The control period is 1.0 second.

Property: Given the environment and network defined above, the system must satisfy the following property:

$$(S_0 \times \mathcal{X} \times \mathcal{Y}) \implies XG_{20}(S_{safe} \times \mathcal{X} \times \mathcal{Y})$$

where $S_0 = [0.6, 0.7] \times [-0.7, -0.6] \times [-0.4, -0.3] \times [0.5, 0.6]$, and $S_{safe} = [-2, 2]^4$, and $\mathcal{X}$ and $\mathcal{Y}$ are the full input and output sets of the network, respectively.

G.6 SB10: Sherlock-Benchmark-10 (Unicycle Car)

This benchmark uses a neural network to control a unicycle model of a car, which can be described by the 2-d postion of the car, x_1 and x_2, and the yaw, x_3 and velocity x_4.

The continous time dynamics can be defined as:

$$\dot{x}_1 = x_4 \cos(x_3)$$
$$\dot{x}_2 = x_4 \sin(x_3)$$
$$\dot{x}_3 = u_2$$
$$\dot{x}_4 = u_1$$

, where u_1 and u_2 are the speed and yaw control signals, respectively.

A discrete model of the environment can be specified as:

$$(x_1)_{t+1} = (x_1)_t + (x_4)_t \cos((x_3)_t)\Delta T$$
$$(x_2)_{t+1} = (x_2)_t + (x_4)_t \sin((x_3)_t)\Delta T$$
$$(x_3)_{t+1} = (x_3)_t + (u_2)_t \Delta T$$
$$(x_4)_{t+1} = (x_4)_t + (u_1)_t \Delta T$$

, where ΔT is the discrete time step size.

To define the environment in the form $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle$, we require a bounded state space. We choose to use $\mathcal{S} = [-1.141, 10.907] \times [-5.125, 0.427] \times [-0.175, 2.824] \times [-1.607, 3.404]$, which was derived from the bounds on the data in the training set for the control network by computing the min and max values of each state variable, and then adding five percent of the range of each variable to its lower and upper bounds to ensure full coverage of the state space. We define $\mathcal{M}(s) = s$, since the state and network input representations are the same. We define the transition function, $\mathcal{T}$ as a function that takes in $s \in \mathcal{S}$ and the output of the network, y, and applies the discrete environment model 100 times, with $\Delta T = 0.002$ and $((u_1)_t, (u_2)_t) = y$.

A neural network is used to control the speed and yaw of the vehicle. The network takes in a vector $x = [x_1, x_2, x_3, x_4]$ that is the same as the state representation. The output of the network must be normalized by subtracting a value of 10, i.e., $(u_1, u_2) = \mathcal{N}(x) - 20$. The control period is 0.2 seconds.

Property: Given the environment and network defined above, the system must satisfy the following property:

$$(S_0 \times \mathcal{X} \times \mathcal{Y}) \implies XF_{50}(S_{goal} \times \mathcal{X} \times \mathcal{Y})$$

where $S_0 = [9.5, 9.55] \times [-4.5, -4.45] \times [2.1, 2.11] \times [1.5, 1.51]$, and $S_{goal} = [-0.6, 0.6] \times [-0.2, 0.2] \times [-0.06, 0.06] \times [-0.3, 0.3]$, and $\mathcal{X}$ and $\mathcal{Y}$ are the full input and output sets of the network, respectively.

G.7 SP: Single Pendulum

This benchmark uses a neural network to control an inverted pendulum. A ball of mass $m = 0.5$ is attached to a massless rod of length $L = 0.5$, which is actuated with a torque u. The state can be modelled by the angle of the pendulum, θ, from the upward vertical axis and the angular velocity, $\dot{\theta}$.

The continuous time dynamics are defined by:

$$\ddot{\theta} = \frac{1}{L}\sin(\theta) + \frac{u}{mL^2}$$

A discrete model of the environment can be specified as:

$$\theta_{t+1} = \theta_t + \dot{\theta}_t \Delta T$$
$$\dot{\theta}_{t+1} = \dot{\theta}_t + (\frac{1}{L}\sin(\theta_t) + \frac{u_t}{mL^2})\Delta T$$

, where ΔT is the discrete time step size.

To define the environment in the form $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle$, we require a bounded state space. We choose to use $\mathcal{S} = [-\pi, \pi]^2$, which allow the angle to cover a full rotation and bounds the angular velocity to reasonably high values. We define $\mathcal{M}(s) = s$, since the state and network input representations are the same. We define the transition function, $\mathcal{T}$ as a function that takes in $s \in \mathcal{S}$ and the output of the network, y, and applies the discrete environment model 100 times, with $\Delta T = 0.0005$ and $u_t = y$.

A neural network is used to control the torque applied to the pendulum, with the goal of keeping the pendulum inverted, i.e., with $\theta = 0$. The network takes in a vector $[\theta, \dot{\theta}]$ that is the same as the state representation. The network is queried at a control period of 0.05 seconds.

Property: Given the environment and network defined above, the system must satisfy the following property:

$$(S_0 \times \mathcal{X} \times \mathcal{Y}) \implies XXXXXXXXXXG_{20}(S_{safe} \times \mathcal{X} \times \mathcal{Y})$$

where $S_0 = [1, 1.2] \times [0, 0.2]$, and $S_{safe} = [0, 1] \times (-\infty, \infty)$, and $\mathcal{X}$ and $\mathcal{Y}$ are the full input and output sets of the network, respectively.

G.8 VCAS

This benchmark models a system involving two aircraft, the ownship and intruder, in which the ownship has an aircraft collision avoidance system, VerticalCAS [62], which issues climb rate advisories once per second. The goal is to avoid near mid-air collisions, which occur when the ownship are separated by less than 100ft vertically and 500ft horizontally. The ownship is assumed to have a constant horizontal speed, and the intruder is assumed to follow a constant horizontal trajectory. The state can be modelled by the intruder altitude relative to the ownship, h, the ownship vertical climb rate, $\dot{h}_o$, the time to loss of vertical separation, τ, and the previous advisory, a_p.

There are 9 possible advisories, each with an associated set of acceptable horizontal accelerations (advisory, accelerations):

0) COC, $\{0\}$

1) DNC, $\{-10.7333, -9.3917, -8.05\}$

2) DND, $\{8.05, 9.3917, 10.7333\}$

3) DES1500, $\{-10.7333, -9.3917, -8.05\}$

4) CLI1500, $\{8.05, 9.3917, 10.7333\}$

5) SDES1500, $\{-10.7333\}$

6) SCLI1500, $\{10.7333\}$

7) SDES2500, $\{-10.7333\}$

8) SCLI2500, $\{10.7333\}$

The competition allowed tools to choose from 2 strategies for selecting accelerations. Either they could select the worst-case acceleration that takes the ownship closer to the intruder, or they could always select the middle acceleration. Here we always select the middle acceleration.

The discrete time system is modelled as:

$$\begin{aligned} h_{t+1} &= h_t - (\dot{h_{ot}} + 0.5(u_2)_t \Delta T)\Delta T \\ (\dot{h}_o)_{t+1} &= (\dot{h}_o)_t + (u_2)_t \Delta T \\ \tau_{t+1} &= \tau - \Delta T \\ a_{t+1} &= (u_1)_t \end{aligned}$$

, where $\Delta T = 1.0$ is the discrete time step size, $(u_1)_t$ is the advisory predicted by the control network, and $(u_2)_t$ is the corresponding acceleration.

To define the environment in the form $\langle \mathcal{S}, \mathcal{M}, \mathcal{T} \rangle$, we require a bounded state space. We choose to use $\mathcal{S} = [-8000, 8000] \times [-100, 100] \times [0, 40] \times [0, 8]$, which was derived from the bounds specified in the network controller specifications. The neural networks require normalized input values, so we define $\mathcal{M}$ as:

$$\mathcal{M}(s) = \begin{bmatrix} 0.0000625 & 0 & 0 & 0 \\ 0 & 0.0002 & 0 & 0 \\ 0 & 0 & 0.025 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -0.5 \\ 0 \end{bmatrix}$$

. Because the system is defined as a discrete time system with $\Delta T = 1$, the transition function is linear, and can be exactly defined as:

$$\mathcal{T}((h, \dot{h}_o, \tau, a), (u_1, u_2)) = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & -0.5 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} h \\ \dot{h}_o \\ \tau \\ a \\ u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \end{bmatrix}$$

This problem is unique in that it uses 9 control networks, one of which is selected at each step by using the previous advisory specified in the state and network input. The control period is 1.0 seconds.

Property: Given the environment and network defined above, the system must satisfy the following property:

$$(S_0 \times \mathcal{X} \times \mathcal{Y}) \implies XG_{10}(S_{safe} \times \mathcal{X} \times \mathcal{Y})$$

where $S_0 = [-133, -129] \times \{-19.5, -22.5, -25.5, -28.5\} \times \{25\} \times \{0\}$, and $S_{safe} = ((-\infty, -100) \cup (100, \infty)) \times (-\infty, \infty)^3$, and $\mathcal{X}$ and $\mathcal{Y}$ are the full input and output sets of the network, respectively.

Appendix H

Additional Experimental Data for Chapter 6

H.1 Environment Rewriting Data

Here we present data on the environment rewriting step, including the time to train and the error of the trained models. We trained each model for 100000 iterations with batches of 100 randomly generated input samples per iteration. We used the AdamW optimizer with an exponentially decaying learning rate starting from 0.001 and ending at 0.000001. The final mean average error (MAE) and slack bounds evaluated using a set of 50000000 randomly sampled points, independent of the training data.

H.1.1 Time to Rewrite Environments

The times to train each model are shown in Tables H.6 and H.7. Table H.6 shows the time in seconds of the linear models used in Experiment 1 of our evaluation. VCAS uses a linear environment model, and so did not require training as it can simply be directly converted to a neural network. All models except that of the AP benchmark took less than 10 minutes to train. The environment model for the AP problem is complex, with 12 state variables, 6 control variables, and is highly non-linear, and computing the ground truth labels at every iteration was an expensive step, leading to increased time cost.

Table H.1: Mean absolute error (MAE) for each model in Experiment 1. All models are linear.

Benchmark	**MAE**
ACC	0.0020063617
AP	0.43135616
DPLR	0.36752835
DPMR	0.14504883
SB9	0.027825627
SB10	0.08174728
SP	0.01977753
VCAS	NA

Table H.2: Mean absolute error (MAE) for each SB9 model.

Model	**MAE**
Linear	0.027953
Non-linear (64 hidden units)	0.002881
Non-linear (128 hidden units)	0.002061
Non-linear (256 hidden units)	0.001648
Non-linear (512 hidden units)	0.002437

H.1.2 Fidelity of Rewritten Environments

Here we present the error of each rewritten model, as both the MAE and the slack bounds.

Tables H.1 and H.2 present the MAE of each model for Experiment 1 and 2, respectively. No model was trained for VCAS because both $\mathcal{M}$ and $\mathcal{T}$ were linear and could be trivially encoded as a network using a single GeMM operation. The MAE values are not necessarily between benchmarks as each model has different size input and output vectors with different scales.

Tables H.3, H.4, and H.5 present the slack intervals for the state values of each model. The model for the AP benchmark in Experiment 1 is presented separately in Table H.4 due to space considerations, since it has double the number of state variables as the benchmark with the next largest state vector. The slack bounds tend to be relatively low for most benchmarks, showing that these models can capture the environment model. The linear model does quite poorly for the AP benchmark, where 2 variables have slack bounds with magnitudes around 20000. We believe this is due to the extreme nonlinearity of the environment and the large input space.

H.2 Experiment 1

In this section we present detailed accounts of the verification results reported by ARCH-COMP AINNCS participants, as well as the verification results obtained by our rewriting approach with

Table H.3: Measured slack intervals for the trained linear environment models in Experiment 1, excluding AP, which is presented separately due to space considerations.

State	ACC	DPLR	DPMR	SB9	SB10	SP
x_1	$[-0.001, 0.001]$	$[-0.089, 0.090]$	$[-0.015, 0.014]$	$[-0.661, 0.693]$	$[-0.079, 0.079]$	$[-0.003, 0.003]$
x_2	$[-0.002, 0.000]$	$[-0.113, 0.113]$	$[-0.018, 0.018]$	$[-0.887, 0.420]$	$[-0.147, 0.147]$	$[-0.122, 0.122]$
x_3	$[-0.019, 0.006]$	$[-3.657, 3.685]$	$[-1.474, 1.457]$	$[-0.128, 0.124]$	$[-0.000, 0.000]$	
x_4	$[-0.001, 0.001]$	$[-4.644, 4.632]$	$[-1.836, 1.835]$	$[-0.103, 0.100]$	$[-0.128, 0.097]$	
x_5	$[-0.001, 0.001]$					
x_6	$[-0.018, 0.006]$					

Table H.4: Measured slack intervals for the trained linear environment model used for the AP benchmark in Experiment 1.

State	AP
x_1	$[-3.841, 3.808]$
x_2	$[-4.548, 4.547]$
x_3	$[-3.787, 3.801]$
x_4	$[-4.220, 4.163]$
x_5	$[-4.254, 4.299]$
x_6	$[-4.203, 4.346]$
x_7	$[-20103.314, 21869.316]$
x_8	$[-0.229, 0.235]$
x_9	$[-19719.666, 21869.324]$
x_{10}	$[-0.000, 0.000]$
x_{11}	$[-0.000, 0.000]$
x_{12}	$[-0.000, 0.000]$

open-loop verifiers. We present the results of the verifiers in Table H.8. The top section of the table shows the participating verifiers by competition year (descending), then alphabetically. The bottom section of the table shows the results using open-loop verifiers after applying our rewriting approach. Each column is one of the benchmark problems and each row is a verifier. Each cell indicates the reported result of each tool with a symbol indicating our classification as one of the following:

- Correct result for the original problem (✓)

Table H.5: Measured slack intervals for each SB9 model.

	State Variable			
Model	x_1	x_2	x_3	x_4
Linear	$[-0.079, 0.079]$	$[-0.15, 0.15]$	$[-2.1e-5, 2.0e-5]$	$[-0.13, 0.097]$
64 hidden units	$[-0.13, 0.23]$	$[-0.20, 0.082]$	$[-0.34, 0.23]$	$[-0.32, 0.38]$
128 hidden units	$[-0.16, 0.15]$	$[-0.22, 0.20]$	$[-0.38, 0.19]$	$[-0.19, 0.20]$
256 hidden units	$[-0.068, 0.051]$	$[-0.12, 0.093]$	$[-0.15, 0.13]$	$[-0.093, 0.094]$
512 hidden units	$[-0.056, 0.082]$	$[-0.10, 0.10]$	$[-0.073, 0.081]$	$[-0.086, 0.12]$

Table H.6: Time to train a linear environment model for each benchmark problem.

Benchmark	**Time (seconds)**
ACC	431
AP	3247
DPLR	321
DPMR	310
SB9	305
SB10	243
SP	190
VCAS	NA

Table H.7: Time to train each SB9 model.

Model	**Time (seconds)**
Linear	305
Non-linear (64 hidden units)	286
Non-linear (128 hidden units)	292
Non-linear (256 hidden units)	294
Non-linear (512 hidden units)	295

- Spurious violation for the original problem (✓)
- Incorrect holds result for the original problem (X)
- Tool ran on, but produced no result for, the original problem (U)
- Tool ran on, but produced an error on, the original problem (E)
- Correct result on modified problem with reduced input space (✓′)
- Returned a result on a modified problem (?′)

.

H.2.1 Results for ARCH-COMP AINNCS Participants

The JuliaReach [14] verifier fully supports benchmarks ACC, SB9 and SB10. It does not support the VCAS benchmark due to the use of multiple controllers and a discrete-time system. On benchmarks AP, DPLR, DPMR, and SP, the tool authors modified the input space of the problem to contain a single point using knowledge of a known counter-example to focus their tool and ensure a counter-example was found.

Table H.8: Verification results on the ARCH-COMP AINNCS benchmarks by AINCSS 2020 and 2021 closed-loop verifiers (above the line) and by open-loop verifiers in conjunction with property, environment, and network rewriting.

	Verifier	ACC	AP	DPLR	DPMR	SB9	SB10	SP	VCAS	Total
Closed-loop	**JuliaReach**	✓	✓′	✓′	✓′	✓	✓	✓′		3
	NNV	✓	U	✓′	✓′	U	U	✓′	✓	2
	Verisig	?′								0
	ReachNN*	U	U			U				0
	OVERT	U				✓	U	✓		2
	VenMAS	✓				?′	?′	?′	✓	2
Open-loop	**Rewriting+ERAN**	✓	U	U	U	U	U	U	✓	2
	Rewriting+Marabou	U	✓	✓	✓	U	U	✓	X	4
	Rewriting+Neurify	✓	✓	✓	✓	E	U	✓	✓	5
	Rewriting+nnenum	✓	✓	✓	✓	E	U	✓	✓	6

NNV [129] supports, and was able to return results for the ACC and VCAS benchmarks. While NNV was able to run on the SB9 and AP benchmarks, it could not provide a result due to the overapproximation of the reachability analysis used. Additionally, NNV could reportedly run on SB10, but ran out of memory (32GB) and did not provide a result. Finally, on benchmarks DPLR, DPMR, and SP, the tool authors modified the input space of the problem using knowledge of a known counter-example to focus their tool and ensure a counter-example was found.

Verisig [59] supports only smooth activation functions, limiting its application to the benchmarks. Verisig was run on a different version of the ACC benchmark, using both a different controller and a property with different initial conditions.

ReachNN* [36] was able to run on the ACC and SB9 benchmarks, but had large overapproximation errors. In the case of large overapproximation errors, ReachNN* distills new control networks and attempts to verify the problem with those instead of the original controllers. In this case, both of these benchmarks could be verified by ReachNN* using the distilled controllers, however the verification result is not meaningful for the original problem. ReachNN* could also reportedly run on the AP benchmark, but was unable to return a result after running out of memory (32GB). ReachNN* does not support SB10 due to the use of multiple controller outputs, and it does not support VCAS due to the discrete-time dynamics. No results were reported for the DPLR, DPMR, or SP benchmarks.

The OVERT [112] verifier was designed to verify discrete-time systems, and so they first discretize the continuous-time dynamics of the 4 benchmarks on which they evaluate their tool, ACC, SB9, SB10, SP. While this means that they are verifying a fundamentally different system than the continuous-time version, we do not penalize them for this change in this analysis. After discretization,

OVERT runs and returns results on the SP and SB9 benchmarks. While OVERT was able to run on the discretized versions of the ACC and SB10 benchmarks, it was not able to check the property up to the full time bound.

Similar to OVERT, VenMAS [2] is designed for discrete-time systems, and discretized continuous-time dynamics when necessary. Additionally, to support non-linear terms in the dynamics model, it trained neural network approximations of some functions, such as $\sin(x)$, $x\sin(y)$, and $x\cos(y)$. VenMAS fully supports the VCAS benchmark, and it supports the ACC benchmark after discretization. While VenMAS was run on the SB9, SB10, and SP benchmarks, the tool authors first had to convert several non-linear operations in the environment models to neural network approximations, which produces a new problem that can lead to incorrect results. In particular, the use of neural network approximations on SB10 for $x\sin(y)$ and $x\cos(y)$ led to an incorrect proof of the result to be reported. In particular, VenMAS shows that the vehicle reaches the target region in 24 time steps, but using the true environment, it doesn't reach the target region until step 50. VenMAS was not run on the AP, DPLR, and DPMR benchmarks due to their highly non-linear models.

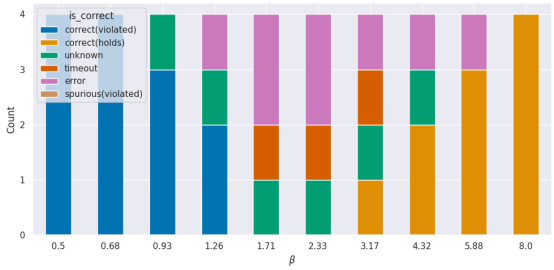
H.2.2 Results for Probelm Rewriting plus Open-Loop Verification

Our rewriting approach enabled the application of many open-loop verifiers, 4 of which we explore in this work.

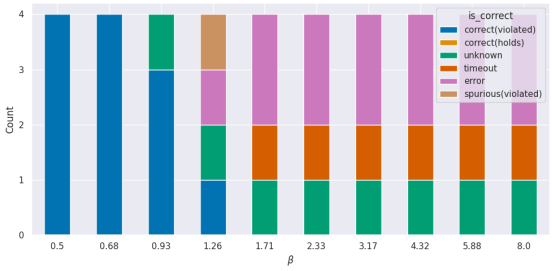
After rewriting ERAN was able to return results for 2 benchmarks, ACC and VCAS. Because ERAN uses a reachability method it frequently reported *unknown* results. It was, however, able to prove that the property was true for the discrete version of ACC, and was able to prove that the VCAS property was violated.

After rewriting Marabou was able to return results for 4 benchmarks, finding verified violations for the AP, DPLR, DPMR, and SP benchmarks. While it did run on ACC, SB9, and SB10, it ran out of memory on ACC, and exceeded the time limit on both SB9 and SB10. Additionally, Marabou incorrectly reported a *holds* result for the VCAS benchmark. We plan to open a bug report to the maintainers of this tool with this example.

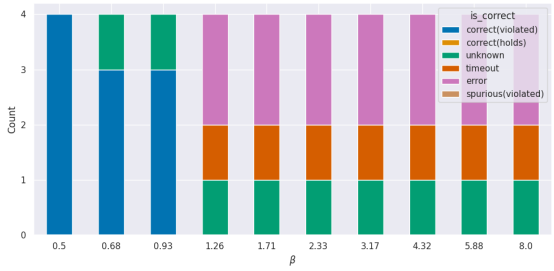
After rewriting Neurify was able to return results for 5 benchmarks, finding verified counter-examples for DPLR, DPMR, SP, and VCAS. It also correctly reported a *holds* result for the discrete time ACC benchmark. Neurify also returned a counter-example for the AP benchmark, which was determined to be spurious after checking it against the ground truth environment model. Neurify was not able to return a result on SB10 due to exceeding the 30 minute time limit. On SB9, the Neurify tool encountered an error and did not return a result.



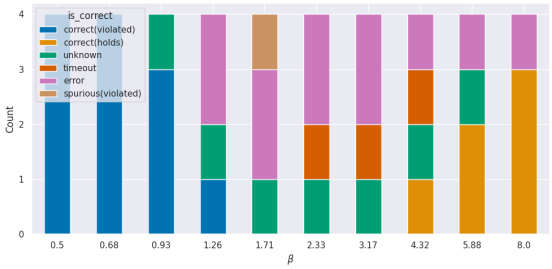
(a) Linear model.



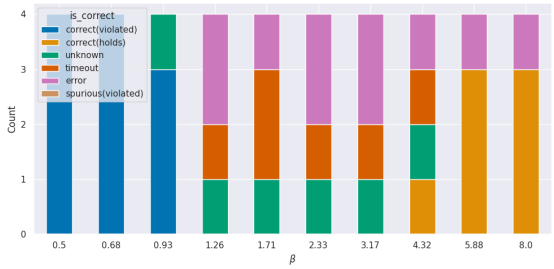
(b) Non-linear model with hidden layer of size 64.



(c) Non-linear model with hidden layer of size 128.



(d) Non-linear model with hidden layer of size 256.



(e) Non-linear model with hidden layer of size 512.

Figure H.1: Verification results per problem bound. Problems on the left should have violations and problems on the right should hold. In between, a transition occurs. Problems nearer the transition are more challenging.

After rewriting nnenum returned results for 6 of the 8 benchmarks. It found verified counter-examples for AP, DPLR, DPMR, SP, and VCAS, and correctly returned *holds* for the discrete time ACC benchmark. Similar to Neurify, nnenum exceeded the time limit on SB10, and encountered an error on SB9, causing it to exit before returning a result.

H.3 Experiment 2

The full results of the open-loop verifiers on the rewritten closed-loop problems using the 5 environment models described in Experiment 2 are shown in Figures H.1. Results are classified as either

correctly being a *holds* result or *violated* result, being a spurious violation (when checked agains the ground truth environment), or being one of *unknown*, *timeout*, *error*. An error result indicates that the verifiers began running, but encountered an issue internally and terminated before returning any result.

As shown in the paper, the linear environment model performs the best, returning the most correct results. Of the non-linear models, the larger models tended to perform better in terms of the number of correct results. These results also show that as the problems move from sat to unsat (left to right), the verifiers return fewer results, with the middle problems returning the fewest correct results. The problems in the middle are more difficult for the verifiers, and lead to more timeout and error results.