# Formal Verification of Neural Networks for Safety-Critical Tasks in Deep Reinforcement Learning

**Davide Corsi**[1] **Enrico Marchesini**[1] **Alessandro Farinelli**[1]

[1] Department of Computer Science , University of Verona , 37135, Verona, Italy
name.surname@univr.it

## Abstract

In the last years, neural networks achieved groundbreaking successes in a wide variety of applications. However, for safety critical tasks, such as robotics and healthcare, it is necessary to provide some specific guarantees before the deployment in a real world context. Even in these scenarios, where high cost equipment and human safety are involved, the evaluation of the models is usually performed with the standard metrics (i.e., cumulative reward or success rate). In this paper, we introduce a novel metric for the evaluation of models in safety critical tasks, the *violation rate*. We build our work upon the concept of formal verification for neural networks, providing a new formulation for the safety properties that aims to ensure that the agent always makes rational decisions. To perform this evaluation, we present ProVe (Property Verifier), a novel approach based on the interval algebra, designed for the analysis of our novel *behavioral* properties. We apply our method to different domains (i.e., mapless navigation for mobile robots, trajectory generation for manipulators, and the standard ACAS benchmark). Results show that the violation rate computed by ProVe provides a good evaluation for the safety of trained models.

## 1 INTRODUCTION

In the last few years, Deep Reinforcement Learning (DRL) has been applied in several safety critical domains (e.g., robotics [Gu et al., 2017] and healthcare [Ruan et al., 2018]), where the search space becomes not manageable for traditional RL algorithms. In these tasks, where high-cost equipment and human safety are involved, the behavior of a Deep Neural Network (DNN) must be evaluated before the deployment in the real scenario to avoid undesirable and potentially dangerous situations. Nonetheless, even in such safety-critical contexts, the evaluation of DRL approaches is typically performed on standard metrics, such as the total reward or the success rate over independent episodes. However, Ian Goodfellow [2015] shows that human-imperceptible perturbations in the input space of DNNs, may result in a tremendous difference in the predicted output. This is a well-known problem in literature, and in particular in the field of computer vision [Madry et al., 2019]. Recently, Huang et al. [2017] discovered that neural network policies, generated by state of the art DRL algorithms, are also vulnerable to adversarial inputs. These particular input configurations are challenging to detect with empirical testing phases and consequently ignored by the standard metrics, underlining the limits of the traditional evaluation approaches.

Following the recent trend in formal verification for Neural Networks, we propose to design a set of safety properties, that encode different constraints on the behaviour of the agent, as a complementary metric for the reward. Ideally, given a safety property and a neural network, a verification framework should either guarantee that the property is always satisfied or return counterexamples [Liu et al., 2019]. The effectiveness of such methodology relies on the estimation accuracy of the output bounds [Xiang et al., 2018a] and has been successfully addressed by several recent studies [Wang et al., 2018a, Dutta et al., 2018]. One of the main limitations of these approaches concerns the design of the safety properties. State of the art methods has proven to be fast and efficient only if the input domain is strict and the target behaviour is well known [Bunel et al., 2020]. However, formalizing a set of properties that satisfy these requirements may not be possible without a deep prior knowledge of the environment and, forcing the agent to respect such strict requirements, may negatively affect the final policy generated by the DRL algorithm. In contrast, in this paper, we focus on the formal verification of properties that describe the general behaviour of the agent (i.e., *behavioral* properties), and that aim at ensuring that the policy makes rational decisions (e.g., if there is an obstacle close to the right never turn

*Accepted for the 37th Conference on Uncertainty in Artificial Intelligence* (UAI 2021).

right). Crucially, we show that a model that respects such soft constraints is overall safer than a model evaluated only on the long term reward. Notice that, properties of this form typically require large input domains, as they need to cover a wide variety of possible input configurations for the DDN (i.e., possible situations for the agent). As a consequence, they are rarely respected in the whole input domain and state of the art approaches typically return only SAT (i.e., the property is respected) or UNSAT (i.e, the property is violated with at least one input configuration), failing to provide useful information on the safety of the model. Against this background, we introduce ProVe a formal verification tool for DNN based on interval analysis [Wang et al., 2018a], designed to verify safety properties for decision-making tasks defined over large input spaces. While previous approaches provide verification tools that aim at verifying whether the bound of an output of the network lies in a given interval, in a DRL context, DNNs typically encode decision-making policies and require the analysis of multiple outputs, considering the relationships among them. To verify this kind of properties we need to modify the standard analysis of the output interval, in fact, as detailed in Section 2, even in an ideal scenario with a perfect estimation of the output intervals, state of the art tools can not always provide useful information on the relationships among the outputs (i.e., can not exploit the comparison rules of the intervals analysis [Moore, 1963]). To this end, in this paper, we propose a novel approach, that provides an accurate shape estimation of the output functions, by computing an iterative bisection of the input intervals. In detail, for a property that should be evaluated in the global domain $I$, we analyze independently a set of $n$ smaller domain $i_n$ (s.t., $I = \bigcup_{i=1}^{n} i_i$). This paves the possibility to handle properties encoded in the form described above. Moreover, we ca exploit the computation independence of the intervals to encode the process in a parallel fashion (e.g., multi-core CPU or GPU), improving the performances of the standard verification tools and hence handling very large input spaces. Crucially, with our approach, it is possible to compute the size of the domain that violates the safety property, providing a safety metric for the evaluation of a model (*violation rate*). Finally, we empirically evaluate ProVe on different domains including: (i) the airborne collision avoidance system (ACAS) [Owen et al., 2019], used in literature as a standard benchmark, (ii) a mapless navigation task for a TurtleBot3, which is a well-known task in DRL [Tai et al., 2017, Zhang et al., 2017, Wahid et al., 2019] and (iii) trajectory generation for the commercial Panda manipulator.

Summarizing, this work makes the following contribution to state of the art: (i) we introduce ProVe, a tool for the formal verification of trained networks that encode decision-making tasks for DRL; (ii) we propose a novel formalism for the safety properties, introducing a new metric to evaluate the safety of a policy (*violation rate*); and (iii) we empirically evaluate our approach on a set of standard benchmarking scenarios, showing that ProVe provides a good evaluation for the safety of trained models.

## 2 PRELIMINARIES

Formal verification of deep neural network can be addressed in many ways. Typically, a verification framework should check whether an input-output relation respects some given constraints. Given a neural network function $f_\theta(x)$ (with parameters $\theta$) with an input domain $\mathcal{D}_x \subseteq \mathbb{R}^{k_i}$ and output domain $\mathcal{D}_y \subseteq \mathbb{R}^{k_o}$, where $k_i$ is the number of input nodes and $k_o$ the number of output nodes, solving the verification problem requires to formally show that a property in the following form holds:

$$\mathbf{x} \in \mathcal{X} \Rightarrow \mathbf{y} = f_\theta(\mathbf{x}) \in \mathcal{Y} \tag{1}$$

where, $\mathcal{X} \subseteq \mathcal{D}_x$ and $\mathcal{Y} \subseteq \mathcal{D}_y$. The input set $\mathcal{X}$ can have different geometries, a common representation is based on *hyperrectangle*, which correspond to multi-dimensional rectangle with a defined center $c \in \mathbb{R}^{k_i}$ and $r_0 \in \mathbb{R}^{k_i}$:

$$\mathcal{X}_h = \{\mathbf{x} : \|\mathbf{x} - c\|_2 \leq r_0\} \subseteq \mathcal{D}_\mathbf{x} \tag{2}$$

More in general the input domain of a safety properties is represented with polytopes, defined as *halfspace-polytopes* in the following form:

$$\mathcal{X}_p = Cx \leq d \tag{3}$$

where $C \in \mathbb{R}^{k \times k_i}$, $d \in \mathbb{R}^k$ and $k$ is the number of inequalities that define the polytope. In literature [Liu et al., 2019], verification approaches are subdivided in two different categories: (i) *optimization* approaches, that use optimization methods (e.g., linear programming [Bastani et al., 2016] or mixed integer linear programming [Lomuscio and Maganti, 2017, Tjeng et al., 2018]) to falsify an assertion; and (ii) *reachability* approaches which, given the input domain for the property $\mathcal{X}$, try to compute the corresponding output domain [Wang et al., 2018b, Weng et al., 2018]. For the first class of problem, frameworks try to falsify the assertion searching for a counter example $\tilde{x} \in \mathcal{X}$ such that:

$$f(\tilde{x}) \notin \mathcal{Y} \tag{4}$$

If no counter example is found, the given property is respected and the solver returns SAT, otherwise the solver return UNSAT or *unknown* if a timeout is reached.

### 2.1 REACHABILITY APPROACHES

In contrast to optimization approaches, a reachability framework does not directly returns SAT or UNSAT. Instead, it try to compute the output reachable set, formally, given the neural network function $f_\theta(x)$ and the property input domain $\mathcal{X}$, the reachability set is defined as:

$$\Gamma(\mathcal{X}, f_\theta) := \{\mathbf{y} : \mathbf{y} = f_\theta(\mathbf{x}),\ \forall \mathbf{x} \in \mathcal{X}\} \tag{5}$$

In this scenario, a property is violated if $\Gamma(\mathcal{X}, f_\theta) \nsubseteq \mathcal{Y}$. However, even state of the art approaches [Wang et al., 2018a, Weng et al., 2018] struggle to compute the exact reachable set, succeeding only in finding an *overestimation* of the real set $\tilde{\Gamma}(\mathcal{X}, f_\theta)$ . Research, in recent years has thus been focused on finding different strategies to reduce the *overestimation* and find a $\tilde{\Gamma}$ as close as possible to $\Gamma$.

**Input Area Propagation** In the last years, previous works show how to extend the Interval Analysis [Moore, 1963] in the field of DNNs verification, partially mitigating the computational limitation of the optimization approaches, to compute the reachability set. In detail, these methods propagate layer by layer the input domain represented as one bound for each input node (input *area*). Naive approaches [Xiang et al., 2018b] computes the bound ($[l_{new}, u_{new}]$) for each nodes of the network in an independent fashion, applying node wise the following linear mapping:

$$\begin{aligned} l_{new} &= \max(\theta, 0) * l + \min(\theta, 0) * u \\ u_{new} &= \max(\theta, 0) * u + \min(\theta, 0) * l \end{aligned} \tag{6}$$

adding the biases if required and propagating the obtained bound through the activation function. In the last years has been shown that some more sophisticated approaches based on the *symbolic propagation* [Wang et al., 2018b] and the *linear relaxation* [Wang et al., 2018a] can provide more accurate estimations of $\tilde{\Gamma}(\mathcal{X}, f_\theta)$ drastically reducing the computation time, while we refer the interested reader to the original works for further details, it is crucial to highlight that these methods can not be easily parallelized and therefore can not take full advantage of the growing computational power. Moreover, these approaches requires the piecewise linearity of the activation functions while the naive method can be applied to any networks with monotone functions.

To describe the input area propagation, we considered an explanatory example with a network with two inputs ($x_0$, $x_1$), one output ($y_0$) and an input area ($[0, 2]$, $[1, 5]$) to propagate

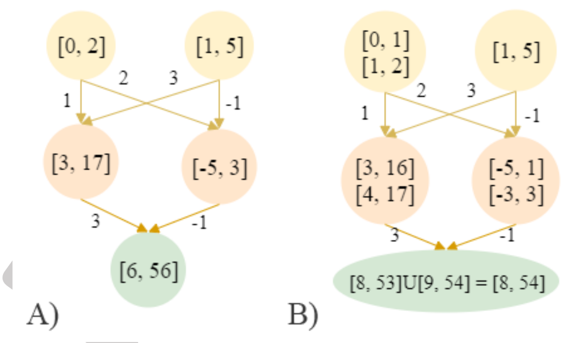

Figure 1: Area propagation, iterative refinement and area bisection. A possible subdivision of area ($[0, 2]$, $[1, 5]$), results in two sub-areas: ($[0, 1]$, $[1, 5]$), ($[1, 2]$, $[1, 5]$)

through the network (Figure 1A). By performing interval multiplications, additions and applying the linear activation considered in this example, we compute the output bound. This method ignores the interdependencies of the input variables, resulting in a very loose estimation (*overestimation problem*). The overestimation problem has been addressed by the *iterative refinement* [Wang et al., 2018b]. This technique obtains a more accurate estimation of the output bound, leveraging the fact that the dependency error for Lipschitz continuous functions decreases as the width of the interval decreases [Weng et al., 2018]. In detail, iterative refinement subdivides the input area in smaller *sub-areas*, computing the corresponding output bound for each of them. The union of the output bounds results in a more accurate bound of the original area. Figure 1B shows an example of iterative refinement, where the process results in a better estimation of the output bound (i.e., $[8, 54]$ in contrast to the $[6, 56]$ estimation of the mere input area propagation).

Our results (Section 5) show that, with a sufficient number of subdivision it is possible to obtain a reachable set compatible with state of the art approaches (e.g., [Wang et al., 2018a]). Crucially, while with a standard computation this approach would lead to a significant overheard on the required computational time, in our parallel setup this method drastically reduce the wall-clock time required by the analysis, paving the possibility to compute $\tilde{\Gamma}$ online inside the training loop.

## 3 BEHAVIORAL PROPERTIES

Verification of DNN for decision-making requires the comparison between the outputs and this may be difficult to achieve with previous approaches. In particular Figure 2 visualizes different scenarios of our output analysis as a 2d graph, to simplify its understanding. Notice that a network with $n > 1$ requires a multi-dimensional graph, however, we assume that each point on the x-axis represents a tuple of $n$ inputs ($x \in X$) in an arbitrary order (as it does not affect the analysis), we represent the outputs ($f(X)$) on the y-axis. Figure 2a shows the typical result of previous verifiers: a generic representation of a single output function. This is also an ideal scenario where the overestimation problem is solved (i.e., the output bounds [a, b] matches minimum and maximum of the output function). However, figure 2b clearly shows the limitations of such methods in the verification of decision-making tasks, where $y_0$ and $y_1$ represent the output functions generated by two nodes of a generic network. In particular, it is not possible to infer which output action will be selected as they only compute the output bounds, without considering the shape (and the relationship) of the output functions. The main insight of ProVe is to compute an accurate estimation of the output function shape, subdividing the input area into multiple subareas and computing the previous operations for each

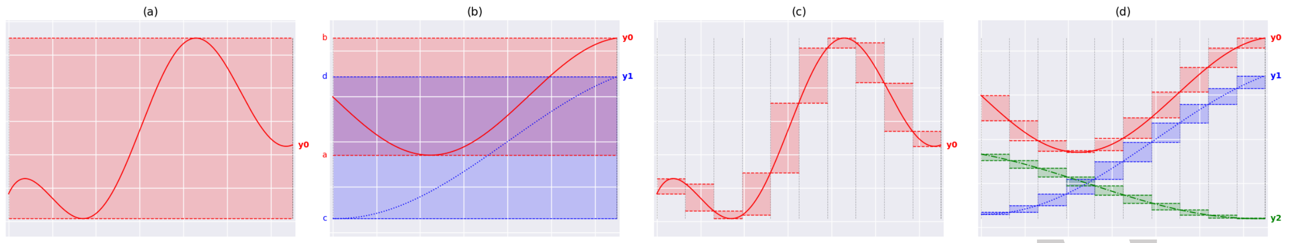

Figure 2: Explanatory output analysis of: (a) one output function with one subdivision; (b) decision-making problem with two outputs and one subdivision. (c) Estimation of an output function shape, using multiple subdivisions. (d) Output analysis with three outputs and multiple subdivisions.

subarea. Figure 2c shows that, through this process, it is possible to obtain a better estimation of an output shape. Finally, Figure 2d considers multiple outputs and subareas, allowing ProVe to state that output $y_0$ is the output that the network will choose for the given input area (i.e., $y_0$ always returns a higher value than the other outputs). This analysis of the relationships among the network outputs is necessary to formally verify a decision-making property.

### 3.1 DEFINITION OF SAFETY PROPERTIES FOR DECISION-MAKING

Following the formulation provided by Liu et al. [2019] (and adopted as standard in previous state of the art works), a safety property for a neural network formalizes an input-output relationship. In detail, a safety property can be formalized in the following form:

$$\Theta : \text{If } x_0 \in [a_0, b_0] \wedge ... \wedge x_n \in [a_n, b_n] \Rightarrow y_j \in [c, d] \quad (7)$$

where $x_k \in X$, with $k \in [0, n]$ and $y_j$ is a generic output.

A property in this form aims at verifying if an output of a network lies in a specific interval. This formulation can be applied to many problems related to robotics and deep learning in general (e.g. the velocity limit of a motor or the probability in a classification task). However, even if it is possible to adapt this formulation to verify simple decision-making properties, it requires to manually modify the input network and introduces overhead in the verification process.

In contrast, we propose a different formulation, specifically designed for decision making problems:

$$\Theta : \text{If } x_0 \in [a_0, b_0] \wedge ... \wedge x_n \in [a_n, b_n] \Rightarrow y_j > y_i \quad (8)$$

We refer to these properties as *safe-decision properties* as they can be used to ensure that a given action (e.g., $y_j$) is always preferred over the others for a given input configuration.

Following the insights of the previous section, we exploit this proposition to prove (or deny) a variety of properties in the form of Prop. 8. In detail, ProVe compares the computed bounds, verifying if the values of one of them is strictly lower than the others, which means that the DNN never selects the action related to the output with the lower value. As an example, to verify properties for a simplified navigation scenario encoded by a DNN with: (i) inputs $x_i \in [0, 1]$ with $i = 0, .., 3$, representing the normalized distance from an obstacle in the four cardinal directions (1 translates in a distance $\geq 1m$ in that direction), where $x_0$ is the *right* distance and $x_1$ is the *left* distance. (ii) outputs $y_0$ = $right$, $y_1$ = $left$, representing the directions where the agent can turn. We could be interested in a property as: $\Theta$*: If an obstacle is close to the right and other directions are obstacle-free, always turn left*. Assume we measured the minimum distance from an obstacle ($0.07m$) that allows our autonomous drone to avoid a collision when turning in the opposite direction, supposing the worst case, where the robot is moving at its maximum speed. We can exploit this constraint to formalize the safety property in the form:

$$\Theta : \text{If } x_0 \in [0, 0.07] \wedge x_1, x_2, x_3 \in \mathcal{D} \Rightarrow y_0 < y_1 \quad (9)$$

where $\mathcal{D} = (0.07, 1]$.

To verify the relation between two (or more) outputs, ProVe relies on the interval algebra of Moore [1963]. In particular, supposing $y' = [a, b]$ and $y'' = [c, d]$ we have the preposition:

$$b < c \Rightarrow y' < y'' \quad (10)$$

To better explain the key problem of previous interval analysis based approaches, that prevent a direct application to decision-making, in Figure 2 we show a simplified visual example of an output analysis for a decision-making property. In a typical scenario we often have that $max(y_1) > min(y_0)$, hence we can not assert anything on the property. Figure 2b shows an example of this behavior, where $d \not< a$, (i.e. the bounds overlaps). In this scenario verification frameworks can not formally verify the property (i.e., we do not have enough information to state if the property condition is true or false). ProVe directly addresses this problem by computing the propagation for a subset of the input area to obtain a more accurate estimation of the output function shape (Figure 2c). This leads to Figure 2d, where $y_1(x) < y_0(x)$ for any $x \in X$ ($X$ is the set of the possible

**Algorithm 1** ProVe

**Input:** network NET, input area matrix AREAS, property PRP, precision EPS.

**Output:** VIOLATION RATE (%), if 0 the property is verified on the entire input area.

```
1: mul-matrix ← generate-mul-matrix(NET)
2: AREAS ← split-area(AREAS, EPS, mul-matrix)
3: for sub-area in AREAS do
4:     output-bound-matrix ← get-out-bound(NET, AREAS)
5: end for
6: denied-areas ← []
7: proved-areas ← []
8: for output-bound in output-bound-array do
9:     test ← check-property(output-bound, PRP)
10:    if test is VIOLATED then
11:        append sub-area to violated-areas
12:        update-violation-rate()
13:    end if
14:    if test is PROVED then
15:        append sub-area to proved-areas
16:    end if
17:    remove [denied-areas, proved-areas] from AREAS
18: end for
19: if len(AREAS) not empty then
20:    return: ProVe(NET, AREAS, PRP, EPS)
21: end if
22: return: DONE, denied-areas, violation-rate
```

inputs), which translates, de facto, in $y_1 < y_0$ (i.e. the network always choose the action represented by $y_0$ inside the input area specified by the property). Furthermore, $y_2 \nless y_1$ (the agent can choose $y_2$ in that input domain).

## 4 PROVE

In this section, we address the huge amount of memory and time required to compute and verify the output bounds for decision-making properties. Furthermore, we introduce a violation rate metric to measure the reliability of a model with respect to a property, showing that in some scenarios, we can design a simple controller to ensure the correct behavior of a DNN in the entire input area.

Considering the pseudo-code in Algorithm 1, ProVe takes as input: (i) a trained DNN; (ii) the input area encoded as a matrix, (iii) the property to verify (expressed in the form described in Section 3.1), and (iv) a discretization value $\epsilon$ (Section 4.1). ProVe proceeds by performing an iterative recursive splitting process using a matrix encoding. In more detail, ProVe generates the multiplication matrix (Section 4.1) for the iterative splitting of the input area, performing the first input split. Then, ProVe propagates the input to compute the output bounds for each unverified sub-areas (a subdivision of the former area, as described in Section 2). This is the most computationally demanding part of ProVe (the number of sub-areas is exponential in the recursion depth). However, this procedure is easily parallelizable on GPU, given that the propagation of each sub-area is independent and the computed output bounds can be analyzed individually, sequentially loading on the GPU memory a subset of the former sub-areas. After these preliminary operations, ProVe evaluates the safety property for each output-bound, returning three possible outputs: (i) the property is violated; (ii) the property holds or (iii) we can not conclude anything on the property in this area (i.e. the bounds overlap as detailed in Section 2). In the first two cases, the property is *verified* (proved or denied) and we remove the verified area from the matrix. In the third case, the area matrix is not empty, and we recursively call the algorithm with the remaining unverified sub-area as input. Moreover, during all the main loop the violation rate is constantly kept updated (Section 4.2). In general, a property could require an uncountable number of splits to be verified. For this reason, we introduce a discretization value $\epsilon$ to create an upper bound on the possible number of iterations (Section 4.1). If the area matrix is empty, we return all the violated areas as counterexample along with the *violation rate* that represents an overestimation of the probability for a property to fail in a real execution. If the violation rate equals to 0, the property is formally verified in the given input area (i.e. the property is true).

### 4.1 MATRIX ENCODING

The verification of decision making DNNs may require a large number of area splitting, hence to use this technique in practical applications the iterative splitting process must be very efficient. The core idea behind ProVe is to exploit matrix operations for the iterative splitting process. Matrix multiplication is a widely used operation for several optimizations and inference tasks, it is known to be highly parallelizable and there exist several dedicated and efficient implementations. We encode the area splitting by using a matrix $A^0$ of size $m \times 2n$, where $m$ is the number of entries (i.e. the sub-areas to split) and $n$ the number of input nodes (i.e. the bounds for each node, which are couple of values). To split the *k-th* input node, we first generate a *multiplication-matrix* $B_{2n \times 4n}$. These are structured as:

$$A^0_{m \times 2n} = \begin{bmatrix} [a_{0,0}, b_{0,0}] & ... & [a_{0,n}, b_{0,n}] \\ ... & ... & ... \\ [a_{m,0}, b_{m,0}] & ... & [a_{m,n}, b_{m,n}] \end{bmatrix}$$

$$B_{2n \times 4n} = \begin{bmatrix} \widetilde{I_{2n}}' & \widetilde{I_{2n}}'' \end{bmatrix}$$

In detail, the matrix $B$ is formed by two identity matrices. Both these matrices maintain the values of an identity matrix $2n \times 2n$, except for the following elements:

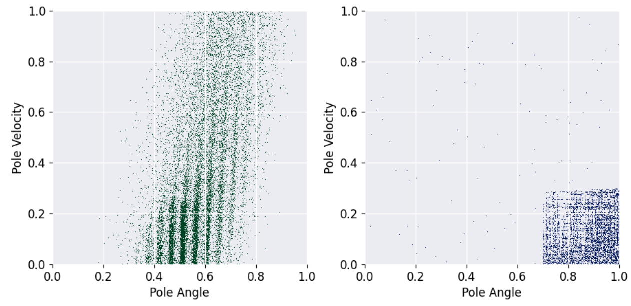

Figure 3: Overview of (left) the most frequent input configuration encountered in a typical execution of the *CartPole* environment and (right) the configurations found by ProVe, that cause a property violation.

$$\widetilde{I_{2n}}'[2k][2k+1] = 0.5$$
$$\widetilde{I_{2n}}'[2k+1][2k+1] = 0.5 - \epsilon,$$
$$\widetilde{I_{2n}}''[2k][2k] = 0.5 \qquad \widetilde{I_{2n}}''[2k+1][2k] = 0.5 + \epsilon$$

where $\epsilon$ is required to solve the infinite splitting loop that we analyze in the next section. The result of matrix multiplication $A^1 = A^0 \times B$ is a matrix $m \times 4n$, where a row contains two parts of the corresponding input area, splitted on the desired *k-th* node. Finally, in time $\mathcal{O}(1)$ we reshape the matrix $A^1$ to the desired matrix $A^1_{2m\times 2n}$:

$$A^1_{2m\times 2n} = \begin{bmatrix} [a_{0,0}, b_{0,0}] & ... & [a_{0,n}, b_{0,n}] \\ ... & ... & ... \\ [a_{2m,0}, b_{2m,0}] & ... & [a_{2m,n}, b_{2m,n}] \end{bmatrix}$$

**Input Discretization** The continuous domain of the input space makes it always possible to split an input area, hence ProVe could, in the some specific worst-cases, loop infinitely on the iterative refinement process. To address this, we introduce a discretization value $\epsilon$ to limit the input precision and ensure the convergence of our algorithm in a finite number of steps. Crucially, the $\epsilon$ parameter could be as small as required by the application, however this will have an impact on the worst case memory and computation required by the approach. In particular, both time and space complexity of the algorithm are exponential in $\epsilon$, as we show in the previous analysis. Notice that, $\epsilon$ can be considered as a lever to address the trade-off between analysing input with higher precision and maintaining the computation manageable. For several applications, and specifically for robotics, $\epsilon$ can be tuned considering the precision of the sensing system. In our mapless navigation task we set $\epsilon$ to be the precision of the lidar system. With this setting ProVe was able to verify the application within an acceptable amount of time on a standard computational architecture (Section 5 for more details).

## 4.2 VIOLATION RATE

As we discussed in Section 2, while standard optimization methods can only return SAT or UNSAT, a key component of our approach is to quantify the number of violations compared to the complete reachable set. We introduce a violation rate metric to infer how a trained DNN performs with respect to the given properties. We define this metric as the percentage of the input area that causes a violation, to compute this value at each step we normalize the size of the area that violates the property with respect to the size of the original input area. Crucially, the violation rate is an upper bound for the actual probability of failure as visually shown in Figure 3. In detail, the left figure shows the states distribution over 10000 episodes of the CartPole scenario [Sutton et al., 1983], while the right one shows the input configurations that cause a violation of a safety property on that environment. Clearly, the states where failures occur are on rarely encountered input. This confirms that an empirical evaluation would most probably not encounter those states, ignoring errors that may appear during the deployment in a real world context.

**Definition 4.1** (Violation Rate)**.** *Given a set of behavioral properties* $\Pi$ *with input domain* $\mathcal{X}$*, a neural network function* $f_\theta(x)$ *and the corresponding estimated reachability set* $\Gamma(\mathcal{X}, f_\theta) := \{\mathbf{y} : \mathbf{y} = f_\theta(\mathbf{x}),\ \forall \mathbf{x} \in \mathcal{X}\}$*. Defining* $\mathcal{X}_{UNSAT}$ *as a subset of the original domain* $\mathcal{X}$ *such that* $\Gamma(\mathcal{X}_{UNSAT}, f_\theta) \cap \mathcal{Y} = \emptyset$*, we define the violation rate as follow:* $v = \frac{|\mathcal{X}_{UNSAT}|}{|\mathcal{X}|}$*,*

**Definition 4.2** (Safe Rate)**.** *Given a set of behavioral properties* $\Pi$ *with input domain* $\mathcal{X}$*, a neural network function* $f_\theta(x)$ *and the corresponding estimated reachability set* $\Gamma(\mathcal{X}, f_\theta) := \{\mathbf{y} : \mathbf{y} = f_\theta(\mathbf{x}),\ \forall \mathbf{x} \in \mathcal{X}\}$*. Defining* $\mathcal{X}_{SAT}$ *as a subset of the original domain* $\mathcal{X}$*, such that* $\Gamma(\mathcal{X}_{SAT}, f_\theta) \subseteq \mathcal{Y}$*, we define the safe rate as follow:* $s = \frac{|\mathcal{X}_{SAT}|}{|\mathcal{X}|}$*.*

Finally, in Section 5 we show how to exploit the violation rate to design a *simple controller* to check at run-time if the current network input causes a violation, formally guaranteeing the safety related to the desired properties.

## 5 EMPIRICAL EVALUATION

We empirically evaluate ProVe[1] on three domains: ACAS, trajectory generation for a robotic manipulator, and mapless navigation. For these domains, we train the DNN by using the Rainbow algorithm [Hessel et al., 2018]. Data are collected on a commercial desktop computer (equipped with a GPU NVIDIA2070, an 8-core CPU, and 16gb RAM), with a *c++* code based on CUDA11.

[1]code available at: https://github.com/d-corsi/ProVe

Table 1: Comparison between the real collision probability and the violation rate of our *behavioural* properties. The table also shows the number of collisions for a model that respects all the original 15 ACAS properties.

| Model | Properties | V. Rate (%) | Collision (%) |
|---|---|---|---|
| *acas_50* | $\theta_L, \theta_R$ | $\approx 50$ | 14.21 |
| *acas_40* | $\theta_L, \theta_R$ | $\approx 40$ | 9.34 |
| *acas_30* | $\theta_L, \theta_R$ | $\approx 30$ | 8.22 |
| *acas_20* | $\theta_L, \theta_R$ | $\approx 20$ | 5.04 |
| *acas_10* | $\theta_L, \theta_R$ | $\approx 10$ | 3.71 |
| *acas_05* | $\theta_L, \theta_R$ | $\approx 5$ | 0.72 |
| *acas_og* | $\theta_0$, ..., $\theta_{15}$ | $\emptyset$ | 0.56 |

## 5.1 ACAS XU COMPARISON

The airborne collision avoidance system (ACAS), designed to prevent the collision between aircrafts, has been widely adopted by previous verification approaches [Owen et al., 2019, Katz et al., 2017]. This system consists of a set of networks, each one with five inputs: (i) distance between ownship and intruders, (ii) heading of ownship with respect to intruder, (iii) heading of the intruder relative to ownship, (iv) speed of ownship and (v) speed of the intruder; and five outputs to encode the action to take to avoid the collision: (i) clear of conflict (which means that no action is needed), (ii) weak right, (iii) strong right, (iv) weak left and (v) strong left. For our evaluation, we rely on the *HorizontalCAS* implementation of Julian and Kochenderfer [2019], which provides a set of trained networks and an extensive dataset of safety critical situations.

In the previous sections, we introduced the problem of the formalization of the safety properties. Design a set of properties that cover all the possible violations could be hard and requires a deep prior knowledge of the environment. In some cases (e.g., real world application or robotics) this process is unfeasible. To support our claims, in Table 1 we show that, even the 15 properties of ACAS, used as standard metric to evaluate the safety in the state of the art works [Wang et al., 2018a, Katz et al., 2017], are not informative enough to guarantee the collision avoidance. In contrast, we propose to design a small set of *behavioural* safety properties, that aims to ensure that the agent makes rational decisions, instead to verify the complete set of unsafe possible actions. We exploited ProVe to obtain the *violation rate* on only two fundamental properties:

$\theta_L$**:** If there is an intruder close on the left, never turn left.
$\theta_R$**:** If there is an intruder close on the right, never turn right.

Table 1 shows our results. We compare six models that achieve the maximum cumulative reward for the task, but with significantly different *violation rates*. To obtain an estimation of the real collision we rely on the safety-critical configurations provided by Julian and Kochenderfer [2019], comparing the behaviour of our models with the correct actions of the dataset. More in detail, from the multitude of trained models that achieve similar rewards, we select a subset with a violation rate ranging from 50 to 4.8 (our best result) with a step of 10. Notice that, in contrast to the standard formulation, for our *behavioural* properties it is difficult to obtain a *violation rate* of zero, having to cover a huge amount of possible situations.

Our results show a strong correlation between the *violation rate* and the real collision estimation, highlighting the limits of the reward for the evaluation of safety-critical tasks and the relevance of our novel metric. Finally, Table 1 shows that the models that obtain the lowest violation rate on our properties (*acas_05*) result in a similar collision number to the one that respect all the 15 original properties (*acas_og*). This further motivates our claims and the hypothesis that our *behavioural* properties, designed without a deep prior knowledge of the environment, provide a good indicator of the safety of a model.

## 5.2 MAPLESS NAVIGATION

In mapless navigation a robot must reach a given target without a map of the surrounding environment, using only local observation to avoid obstacles. This is a challenging scenario for DRL that has attracted significant attention in recent literature [Zhang et al., 2017, Wahid et al., 2019]. In detail, we consider a Turtlebot3 navigating with constant linear velocity, which is a widely used platform in several previous works focusing on DRL for navigation [Tai et al., 2017, 2016]. Here, we use Unity as simulator [Juliani et al., 2018] (previous work [Corsi et al., 2020] demonstrates that this is an efficient and realistic simulation of the behavior of the robot).

In our setup, the network has twelve inputs: (i) the laser sensor is used to collect a sparse 11-dimensional scan values $x_0, .., x_{10}$ normalized $\in [0, 1]$ and sampled between -90 and 90 $deg$ in a fixed angle distribution. The lidar sensor precision is the manufacturer specification used to set the $\epsilon$ discretization value; (ii) the heading of the target with respect to the robot heading ($x_{11}$ normalized $\in [0, 1]$); and three outputs for the angular velocity (i.e., [-90, 0, 90] deg/s).

We evaluate the robustness of the network with three different properties, that represent important behaviors that the agent must respect to be considered reliable in navigating in a polygonal map without collision with the surrounding walls.

$\Theta_{T,0}$**:** If the target is in front of the robot and no obstacle is detected, go straight.
$\Theta_{T,1}$**:** If there is an obstacle close to the left never turn left.
$\Theta_{T,2}$**:** If there is an obstacle close to the right never turn right.

Table 2: Execution time comparison between Neurify and ProVe on the standard properties of the ACAS Xu dataset. For each group of properties table shows the mean computation time (seconds) and the average speed up (x).

| Property | Neurify (sec) | ProVe (sec) | Gain (X) |
|---|---|---|---|
| $\Theta_{A,1}$ | 1037.37 | 345.12 | 3.01 |
| $\Theta_{A,2}$ | 19352.12 | 339.72 | 56.08 |
| $\Theta_{A,3}$ | 1359.89 | 159.22 | 8.54 |
| $\Theta_{A,4}$ | 113.62 | 132.31 | 0.86 |
| $\Theta_{A,5}$ | 22.07 | 5.16 | 4.27 |
| $\Theta_{A,6}$ | 4.91 | 11.5 | 0.42 |
| $\Theta_{A,7}$ | 1278.7 | 64.84 | 19.72 |
| $\Theta_{A,8}$ | 412.29 | 11.21 | 36.78 |
| $\Theta_{A,9}$ | 643.8 | 42.97 | 14.98 |
| $\Theta_{A,10}$ | 60.01 | 10.08 | 5.95 |
| $\Theta_{A,11}$ | 0.51 | 5.79 | 0.08 |
| $\Theta_{A,12}$ | 8.82 | 8.86 | 0.99 |
| $\Theta_{A,13}$ | 34.76 | 10.34 | 3.36 |
| $\Theta_{A,14}$ | 23.74 | 8.09 | 2.93 |
| $\Theta_{A,15}$ | 1136.55 | 8.22 | 16.61 |
| **Total:** | **25526.52** | **1163.43** | **21.94** |

## 5.3 TRAJECTORY GENERATION FOR A ROBOTIC MANIPULATOR

We chose safety verification for robotic manipulators because they are critical assets for industrial environments. Our task is to rotate each joint of the robot to generate a real-time trajectory to reach a target (a similar task is considered in [Gu et al., 2017, Marchesini et al., 2019]). In our setup, the input layer of the network contains 9 nodes normalized in range $[0, 1]$: (i) one for each considered joint and (ii) the last three to encode the target coordinate. We use 12 nodes in the output layer: each joint is represented by 2 nodes to decide if it should move $\omega$ degrees clockwise or anti-clockwise. This encoding of the output allows a straightforward verification process for our tool (i.e., one node represents only one specific action). Furthermore, operating the manipulator in the Cartesian space, we can use $\omega$ as the $\epsilon$ value. Hence, to formally verify if the manipulator operates inside its work-space, we consider properties in the following form: if the current angle of a joint $j_i$ is equal to one of its domain limits, whatever is the configuration of the other joints and whatever is the position of the target, the robot must not rotate $j_i$ in the wrong direction (i.e. an action that rotates $j_i$ causes the robot to exit from the work-space). Considering the network architecture and our task formalization, we design safety properties as:

$\Theta_{P,0}$**:** If the first joint current rotation is close to the left limit of the work-space, never rotate that joint on the left.

Property $\Theta_{P,0L}$ represents a configuration where the angle of joint $j_0$ equals to its limit on the left (i.e., a normalized

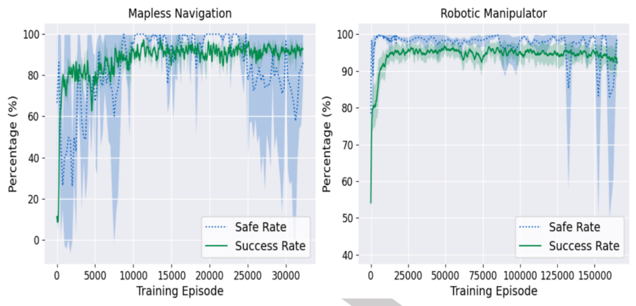

Figure 4: Comparison between the success rate and the safe rate in our environments.

value 1) and whatever values the other inputs of the network assume, the output value corresponding to the action *rotate left*, must be lower than at least one of the others. For each joint $j_i$ we consider two properties, one for the left limit ($\Theta_{P,iL}$) and one for the right limit ($\Theta_{P,iR}$).

## 5.4 DISCUSSION

In order to collect statistically significant data, we performed different training phases for each task, using different random seeds [Colas et al., 2019]. We report mean and standard deviation (smoothed over one hundred episodes) for each task, considering (i) the success rate (i.e., the number of successful episodes over 100 sequential steps) and (ii) the safe rate (the complementary metric of the violation rate, used for a clearer visualization of the graphs).

Figure 4 shows that the *violation rate* is not directly correlated to the success rate (or cumulative reward). In particular, in the early stage of the training, the safe rate is surprisingly high (i.e., the model is overall safe), we motivate that because the agent has not yet learned the task, consequently, it tends to stay still or move around the same point. Afterward, the safe rate starts to follow the trend of the success rate, reaching the best values, this is the fundamental phase, where the agent starts to learn the policy with a good generalization for unseen situations. Moreover, Figure 4 highlights the problem described in the previous sections, the safe rate becomes unstable. A multitude of models with the same performance for the standard metric, obtains significantly different values from the safety point of view. Moreover, our results show an overall drop in the safety of the models in the last stage of the training, in general, the agents starts to learn shorter paths, taking more dangerous actions to maximize the reward. This further motivates the need to use additional evaluation metrics, before the deployment in a real world scenario. Finally, Table 2 shows that ProVe can also be applied on the standard properties, providing a performance comparison between ProVe and Neurify [Wang et al., 2018a]. On average, ProVe achieves a speedup up of *20x* over Neurify. In particular, we found a huge improvement on the most time-demanding properties (i.e. $\theta_1$, $\theta_2$,

$\theta_{15}$) while we obtain slightly worse performance on the simplest ones. We motivate this as our approach requires to load all the data on the GPU memory before the operations, and this is a bottleneck on simple (and fast to verify) properties. An additional analysis of the time and space complexity, a convergence proof of the algorithm and the mathematical formulation of our safety properties can be found in the supplementary materials.

**Simple Controller** Due to the low violation rate of our properties, it is possible to design a simple controller to guarantee the correct behavior of the network. To illustrate this, we describe the process to decide whether the controller can be designed for the trajectory generation environment. From the manipulator data-sheet, a step of 2 degrees (i.e. the $\omega$ value of our controller) requires $\approx 0.01$s to be executed by the arm and, with the violation rate presented in Figure 4, a complete search through the array of the sub-areas that cause a violation, always requires less than 0.01s. Ideally, this means that we can verify if the input state leads to a violation at each iteration, without lags in the robot operations (notice that this depends on the hardware). In contrast, we computed that with a violation rate of $\approx 12\%$, a complete search requires approximately 1.02s, making our solution unfeasible without operational lags. Clearly, the application of a simple controller to guarantee the correct behavior under a certain property is limited by many factors, such as the nature of the task and the characteristic of the agent.

## 6 RELATED WORK

Safety for DRL can be addressed in many ways. A wide branch of literature proposes the use of well-designed reward functions or aim at constraining the exploration [Garcıa and Fernández, 2015, Li et al., 2018]. These methods, minimize undesirable behaviors but can not provide formal guarantees. Since the input space is effectively infinite, it is not possible to test all the input configurations and it is well-known that DNNs are vulnerable to specific adversarial attacks or to an incorrect generalization to new situations [Papernot et al., 2016]. To overcome these limitations, in the last years a novel research direction aims at providing some formal guarantees on the network. Two of the first attempts in this direction are NeVer [Pulina and Tacchella, 2010], which shows that it is possible to abstract the verification problem with the interval algebra [Moore, 1963]), and ILP [Bastani et al., 2016], which is the first method that proposes to apply linear programming approaches to verify some properties on the neural network. However, both of these early attempts suffer from scalability problems and can not be applied to a general neural network with heterogeneous activation functions. Nevertheless, these two works paved the way for the two different branches for the formal analysis of neural networks [Liu et al., 2019], respectively *reachability* approaches and *optimization*.

**Reachability** This class of methods compute the output reachable set $\Gamma(\mathcal{X}, f_\theta)$ exploiting the interval algebra to perform a layer-by-layer analysis. ExactReach [Xiang et al., 2018a] computes, and maintains, an independent reachable set for every linear segments of the piecewise activation function. However, the number of segments grows exponentially and become intractable for a realistic neural network. To overcome this limitation Ai2 [Gehr et al., 2018] proposes an evolution of the previous method by approximating the reachable set, joining the results from different linear segment and obtaining a smaller number of set to propagate. Unfortunately, this method is only applicable to piecewise activation functions and can only provide a large overestimation of the actual reachable set. In recent years, ReluVal [Wang et al., 2018b] proposes a *symbolic propagation* approach to reduce the overestimation, obtaining a more accurate estimation of the reachable set. Weng et al. [2018] and Wang et al. [2018a] propose an evolution of the previous methods based on the *linear relaxation* to handle different types of nonlinear activation functions.

**Optimization** These methods propose to look at the verification as an optimization problem, representing the neural network as a constraint. A keystone for this class of problem is Reluplex [Katz et al., 2017], which proposes the use of the simplex algorithm to find counterexamples that falsify the given properties. MIPVerify [Tjeng et al., 2018] encodes the ReLu activation functions as integer constraints for a mixed integer linear programming problem. Built upon these approaches, Sherlock [Bastani et al., 2016] and BaB [Bunel et al., 2018] propose a hybrid approach for the estimation of the reachable set.

## 7 CONCLUSION

In this paper, we highlight the limits of the standard evaluation for safety critical tasks in DRL. To overcome this limitation, we introduce a novel metric to measure the reliability of a trained model, the *violation rate*. We build this evaluation strategy upon the concept of formal verification for DDN, introducing a different encoding for the safety properties, specifically designed to ensure that the agents always make rational decisions. Crucially, we present ProVe, an efficient novel framework designed to formally evaluate these properties for real-world decision-making problems. We evaluate ProVe on various tasks of interest in DRL literature, the ACAS dataset and two robotic scenarios, mapless navigation, and trajectory generation for a commercial manipulator. This paper paves the way for several important research directions, which include exploiting the information given by the property analysis to perform a safe-oriented training phase. Moreover, to guarantee that the network is trained to respect a set of safety properties, it is possible to exploit ProVe in the early stages of the training to guarantee a safe exploration process.

## References

Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Conference on Neural Information Processing Systems*, 2016.

Rudy Bunel, Ilker Turkaslan, Philip HS Torr, Pushmeet Kohli, and M Pawan Kumar. A unified view of piecewise linear neural network verification. In *Conference on Neural Information Processing Systems*, 2018.

Rudy Bunel, Oliver Hinder, Srinadh Bhojanapalli, et al. An efficient nonconvex reformulation of stagewise convex optimization problems. In *Conference on Neural Information Processing Systems*, 2020.

Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. A hitchhiker's guide to statistical comparisons of reinforcement learning algorithms, 2019.

Davide Corsi, Enrico Marchesini, Alessandro Farinelli, and Paolo Fiorini. Formal verification for safe deep reinforcement learning in trajectory generation. In *International Conference on Robotic Computing*, 2020.

Souradeep Dutta, Susmit Jha, Sriram Sanakaranarayanan, and Ashish Tiwari. Output range analysis for deep neural networks. In *NASA Formal Methods*, 2018.

Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. In *Journal of Machine Learning Research*, 2015.

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

S. Gu, E. Holly, T. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *International Conference on Robotics and Automation*, 2017.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Conference on Artificial Intelligence*, 2018.

Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. *International Conference on Learning Representations*, 2017.

Christian Szegedy Ian Goodfellow, Jonathon Shlens. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

Kyle D Julian and Mykel J Kochenderfer. Guaranteeing safety for neural network-based aircraft collision avoidance systems. In *Digital Avionics Systems Conference (DASC)*, 2019.

Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. In *arXiv*, 2018.

Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *Computer Aided Verification*, 2017.

Zhaojian Li, Uroš Kalabić, and Tianshu Chu. Safe reinforcement learning: Learning with supervision using a constraint-admissible set. In *Annual American Control Conference*, 2018.

Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark Barrett, and Mykel J. Kochenderfer. Algorithms for verifying deep neural networks. In *Foundations and Trends in Optimization*, 2019.

Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. In *arXiv*, 2017.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks, 2019.

E. Marchesini, D. Corsi, A. Benfatti, A. Farinelli, and P. Fiorini. Double deep q-network for trajectory generation of a commercial 7dof redundant manipulator. In *International Conference on Robotics and Automation*, 2019.

Ramon Edgar Moore. Interval arithmetic and automatic error analysis in digital computing. In *Stanford University*, 1963.

M. P. Owen, A. Panken, R. Moss, L. Alvarez, and C. Leeper. Acas xu: Integrated collision avoidance and detect and avoid capability for uas. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, 2019.

Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *European symposium on security and privacy*, 2016.

Luca Pulina and Armando Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *International Conference on Computer Aided Verification*, 2010.

Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. Reachability analysis of deep neural networks with provable guarantees. In *International Joint Conference on Artificial Intelligence*, 2018.

R. S. Sutton, A. G. Barto, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. In *IEEE Transactions on Systems, Man, and Cybernetics*, 1983.

L. Tai, G. Paolo, and M. Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *International Conference on Intelligent Robots and Systems*, 2017.

Lei Tai, Shaohua Li, and Ming Liu. A deep-network solution towards model-less obstacle avoidance. In *International conference on Intelligent Robots and Systems (IROS)*, 2016.

Vincent Tjeng, Kai Y Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2018.

Ayzaan Wahid, Alexander Toshev, Marek Fiser, and Tsang-Wei Edward Lee. Long range neural navigation policies for the real world. In *International Conference on Intelligent Robots and Systems*, 2019.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. In *Conference on Neural Information Processing Systems*, 2018a.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In $USENIX$ *Security Symposium*, 2018b.

Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning*, 2018.

Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. Reachable set computation and safety verification for neural networks with relu activations. In *Annual American Control Conference*, 2018a.

Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. Output reachable set estimation and verification for multilayer neural networks. In *IEEE transactions on neural networks and learning systems*, 2018b.

J. Zhang, J. T. Springenberg, J. Boedecker, and W. Burgard. Deep reinforcement learning with successor features for navigation across similar environments. In *International Conference on Intelligent Robots and Systems*, 2017.