Formal Verification for Safe Deep Reinforcement Learning in Trajectory Generation

Davide Corsi*, Enrico Marchesini*, Alessandro Farinelli, Paolo Fiorini
Department of Computer Science
University of Verona - 37134 Verona, Italy
*equal contribution: name.surname@univr.it

***Abstract*—We consider the problem of Safe Deep Reinforcement Learning (DRL) using formal verification in a trajectory generation task. In more detail, we propose an approach to verify whether a trained model can generate trajectories that are guaranteed to meet safety properties (e.g., operate in a limited work-space). We show that our verification approach based on interval analysis, provably guarantees whether a model meets pre-specified safety properties and it returns the input values that cause a violation of such properties. Furthermore, we show that an optimized DRL approach (i.e., using scaling discount factor and a mixed exploration policy based on a directional controller) can reach the target with millimeter precision while reducing the set of inputs that cause safety violations. Crucially, in our experiments, the number of undesirable inputs is so low that they can be directly removed with a simple controller.**

I. Introduction

The key to successfully apply Deep Reinforcement Learning in robotics is the ability to adapt to the surrounding environment by generalizing from the training experiences. Robotic applications, however, have to cope with the uncertainties of both the physical hardware and the operational environment, hence they usually require a huge number of trials to achieve reasonable performances. For this reason, devising learning approaches while reducing training time is a key issue for the application of DRL in robotics.

In particular, DRL methods have been recently applied to learn a variety of behavioral skills for different platforms [1], [2], [3], [4]. Typically, in these situations, several safety constraints (e.g., limited work-space) and high-cost hardware are involved, hence it is crucial to guarantee the correct behavior of a trained model before deploying the system in real applications. However, ensuring a provable behavior of a complex function approximator such as a neural network is still an open problem and the lack of such a guarantee represents one of the main issues that prevent wider use of Deep Neural Networks (DNNs) for building trustworthy commercial solutions.

In this project, we address both the sample efficiency and the formal verification of DRL models. For the former, starting from Double Deep Q-Network (DDQN) [5], we propose a DRL learning method designed to reduce the training phase, that can train a commercial redundant 7-DOF robotic arm to generate trajectories for the end effector. Crucially, we show that these optimizations lead to a safer model, our verification method finds fewer unsafe behaviors with respect to a model trained with the original DDQN algorithm.

We apply our approach to the Panda Franka Emika[1] (see Figure 1) and we use the Panda's Denavit-Hartenberg (D-H) parameters as a model. The key element to reduce the time required by the training phase is our scaling discount factor and the use of a mixed exploration policy, based on a directional controller [6]. These optimizations allow us to further increase the complexity of the scenario, reaching random targets in the whole work-space of the robot, within millimeter precision. In contrast, previous approaches on trajectory generation [3], [7] consider only random targets generated in a cube of size 0.2m centered around a fixed point, with a 5cm precision.

For what concerns formal verification of DRL models, several research efforts have been devoted to this problem, introducing different formal verification approaches for DNN [8], [9], [10] based on interval analysis [11]. However, all these techniques are not designed for directly verifying a DRL model that encodes decision-making policies (e.g., which joint has to move to keep the manipulator in its work-space), hence they focus on verifying whether the bound of a single output of the network lies in a given interval (e.g., a motor velocity never exceeds fixed bounds). In contrast, a DNN for decision-making typically requires the analysis of multiple outputs (e.g., choose the action that maximizes a return). Starting from the existing verification tools, our approach focuses on the verification of the behavior of our manipulator[2], formalized as a decision-making problem (see Section III). In detail, our method proposes the use of interval analysis to verify the relationships between two or more network outputs, allowing the verification of DRL models, where the output nodes correspond to actions and the DNN chooses the best one. This allows us to formally verify the safe behavior of a 7-DOF robotic arm.

Our results show that the use of certain optimizations (such as the scaling discount factor or the direction controller) during the training phase significantly reduces the error between the target point and the final position of the end-effector, while also diminishing the training time and the number of unsafe actions. Crucially, our extension of previous interval analysis based tools allows to formally prove that our DRL model

[1]https://www.franka.de/panda

[2]Among the variety of robotic platforms in both research and industry, we considered a manipulator for their wide utilization.

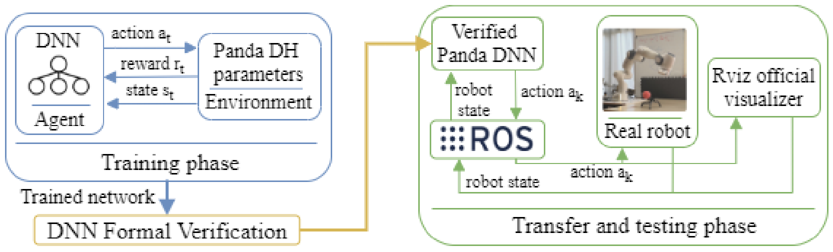


Fig. 1. Overall schematic of the formal verification process.

satisfies a set of constraints that ensure the safety of the trained network.

Summarizing, this paper makes the following contribution to the state of the art: (i) Our optimizations, i.e., the exploration policy and the adaptive discount factor, guarantee a high success rate (over 95%), accelerate the training phase (from 4 to 2 hours) and reduce both the error of the trajectory (from $5cm$ to $< 0.1cm$), and the set of property violations. (ii) Our verification approach, based on existing interval analysis tool [9], allows the formal verification of a DRL model in decision-making scenarios. (iii) We exploit the undesirable actions returned by our method to design a simple controller that corrects these wrong behaviors.

II. Background and Related Work

A typical feed-forward deep neural network is a function $f : X \to Y$ mapping an array of inputs $x_i \in X$ with $i \in [0, .., n]$ to outputs $y_j \in Y$ with $j \in [0, .., m]$.

The DNN function f is a sequence of parametric functions based on: (i) the weights of the network, (ii) the values of the inputs, and (iii) non-linear transformations Ω, known as activation functions. Among the variety of the latter (e.g., sigmoid, hyperbolic, tanh, ...), in this work we focus on Rectified Linear Unit (ReLU), one of the most used function in modern state-of-the-art DNN [8], which was previously considered by other neural network analysis methods [12].

In detail, a DNN is used as a function approximator in a trajectory generation problem, which can be formalized as a Reinforcement Learning problem over a Markov Decision Process (MDP). In a finite number of discrete time steps $t \in [0, T]$, the robot chooses and executes an action a_t according to the state s_t, reaching a new state s_{t+1} and receiving a reward $r(s_t, a_t)$. The process continues until T is reached or the agent ends up in a terminal state, hence terminating an episode. The goal is to maximize the total discounted reward (i.e., the return) from step t onward, given by $R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_t, a_t)$ with $0 < \gamma < 1$, where T is the time-step at which the experiment ends and γ is the discount factor. Given a policy π that maps states to actions, the action-value function $Q^\pi(s, a)$ describes the expected return from state s after taking action a and following policy π.

A. Learning for trajectory generation

DRL has been previously employed on 7-DOF robotic arms [7], [3]. These methodologies, however, mostly consider a limited target area or fixed targets during the test on the real robot (e.g., a door handle). An exhaustive evaluation of DRL algorithms to perform the trajectory generation of a 7-DOF manipulator is presented in [7]. In particular, authors consider only continuous action space algorithms. Moreover, their random target position is sampled uniformly from a cube of size 0.2m centered around a point, with a precision of 5cm. In contrast, authors in [3] show that it is possible to use discrete action space algorithms for the trajectory generation problem, but considering an error of 5cm between the target and the end-effector. Following this, we optimize DDQN to generate a trajectory in the whole considered work-space of the manipulator, and reducing the error between the target position and the end-effector to $< 0.1cm$, using our directional controller.

B. Formal Verification Analysis for Deep Neural Networks

Typical Safe DRL approaches [13], [14] address the safety of the model introducing constraints in the training phase or well-defined reward functions, but they are not designed to formally guarantee the correct behavior of the network. In contrast, several techniques have been developed in the area of Supervised Learning (e.g., regression and classification) where it is possible to formally ensure the correctness of the network, analyzing the output of the trained model. In detail, recent works focus on small perturbation on the input values that can fool state-of-the-art models [15]. As example, in the field of image recognition [16] a trained network is considered reliable only if it is evaluated against these perturbations. To address these challenges, previous methods propose the use of Satisfiability Modulo Theories (SMT) solvers, which application is strongly limited due to their computational efficiency [12] (that depends on the size of the network) or their imprecise domain representation [17]. Mixed Integer Linear Programming [18] and global optimization approach [19] have also been considered to address the scalability issues of SMT-based tools. However, such methods have been designed and tested mainly on Supervised Learning tasks and are not directly applicable to decision making problems such as the ones we consider here.

C. Interval analysis based verification tools

Interval analysis provides an alternative scalable strategy for verification in different domains [11], [20]. This method provides bounds on the solution of an equation and can represent an efficient strategy for the approximation of numerical optimization problems. Recent verification tools [8], [10] show how to extend the interval analysis for neural networks verification. In detail, they propose the evaluation of a property in the form:

$$\Theta_0\text{: } \textit{If } x_0 \in [a_0, b_0] \wedge ... \wedge x_n \in [a_n, b_n] \Rightarrow y_j \in [c, d]$$

where $\mathrm{x_0}$, ..., $\mathrm{x_n}$ are the inputs and $\mathrm{y_j}$ is a generic output of the network.

Specifically, given an input area and a property to verify, these methods propagate the input through the network to compute the bounds for each output node. The main problem of the computed bounds is related to the non-linear activation functions of the model (e.g., ReLU), that suffer from inherent overestimation caused by the input dependencies in the application of interval arithmetic to a complex function (i.e. a neural network). The imprecise estimation of the bounds provides poor information on the behavior of the model.

Recently, [8] proposes a $bisection$ based method that addresses this issue; the idea is to split the input area and re-perform the propagation on smaller inputs. After many propagation phases, authors perform the union of the generated sub-intervals to obtain a more accurate estimation of the output bounds. For an overview of this methodology, we refer to the original implementation [8].

Crucially, as mentioned before, all previous methods typically aim at computing absolute bounds (e.g., to define the bound for each output given a bound on the input). In contrast, our extension aims at verifying decision making tasks where we do not need the absolute value that each output assumes, instead we must verify the output value relations (i.e., how the value of an output differs from the value of another output, in a specific input area). In contrast to previous methods that merge the generated sub-intervals to compute the output bound of the entire input area, the idea is to compute the bounds of the different sub-areas, analyzing the output trends in the generated sub-intervals to obtain a more accurate overview of the output values (more details in Section IV).

III. Training

In this section, we address the sample efficiency problem, starting with the setup of the network and our optimizations to accelerate the training phase.

A. Problem Encoding

Given the manufacturer specifications, we encode the manipulator work-space considering a range of ± 120 degree for the first six joints (with the possibility for further scaling), excluding the last one in the wrist, which is responsible for the grasping phase that we do not consider. A computation of the network represents an action that moves one joint of $\omega = 2$ degrees. Given the capability of generalization of the agent, once the network is trained, it is possible to generate a trajectory with different ω. A key factor that allows us to use the DDQN [5] (which considers discrete action spaces) algorithm and its extensions [21], [22] (in contrast to the continuous action space algorithms evaluated in [7]) is the functions provided by the Franka APIs that allows us to interpolate the discrete joint steps, to create a smooth trajectory with a desired joint velocity. We considered a discrete action space as recently, [23], [24], [25] show that discrete action space algorithms, with high discretization in the output space, are a competitive alternative over continuous methods in locomotion benchmarks while requiring shorter training time.

During the training phase, the target position coordinates are represented as a triple <x, y, z> $\in \mathbf{R}^3$ sampled uniformly in the whole work-space of the robot.

1) Network architecture: the input layer of the network contains 9 nodes, one for each considered joint and the last three for the target. These values are normalized in range $[0, 1]$. Multiple trials with different random seeds and network sizes [26] led us to use two hidden layers with 64 neurons each and ReLU activation function. Finally, we use 12 linear nodes in the output layer (each joint is represented by 2 nodes to decide if it should move of ω degrees clockwise or anti-clockwise), for a straightforward verification process.

2) Reward function: a natural choice for a dense reward is the distance from the end-effector to the target. However, we noticed that there might be configurations that are close to the goal but can not reach the desired final configuration. Giving a high reward to such configurations would provide false positive information, ruining the training phase. To avoid this, the agent receives a sparse reward in case of reaching the target, and for timeouts:

$$reward = \begin{cases} -1 & \text{timeout is reached} \\ 0 & \text{intermediate step} \\ 1 & \text{end-effector has reached the target} \end{cases} \tag{1}$$

3) Discount factor: the joint steps that compose the trajectory tend to be minimized in the advanced stages of the training (the network learns to reach the target as fast as possible to get the positive reward). Authors in [27] consider to upscale the discount, starting to learn by maximizing rewards on a short-term horizon and progressively giving more weights to delayed rewards; our approach works in the opposite way. Initially, we give more importance to delayed rewards and then to short-term ones (linearly scaling downwards the discount). This is due to our sparse reward function. With this method, the success rate results 10-15% better with respect to the fixed discount (Figure 2A), keeping the same trajectory dimension in terms of joint steps.

4) Directional controller for mixed exploration: The mixed exploration policy replaces part of the random choices of the ϵ-greedy strategy, with actions that are correct, providing the network more good samples. For example, given j_0 the current position of $joint_0$ and j_{0f} its final pose when generating the target, if $j_0 < j_{0f}$ then a clockwise action is selected. Furthermore, training the network using the controller as policy when the end-effector reaches $\tau = 2cm$ from the target, enables us to reach a millimeter precision $< 0.1cm$ between the end-effector and the target. This improves results

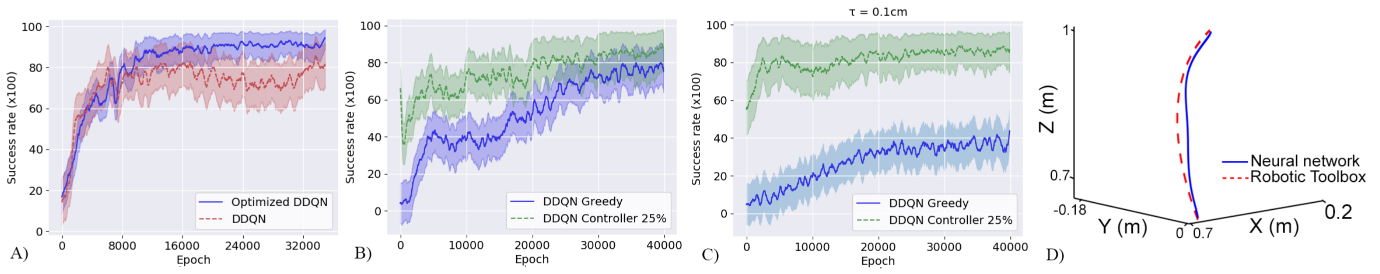


Fig. 2. (A) Average success rate between our optimized DDQN and the original DDQN. (B) Average success rate with our mixed exploration phase ($\tau = 2cm$) and (C) with the directional controller for the last part of the trajectory ($\tau < 0.1cm$) (D) Explanatory trajectory generated by our model and Matlab.

of continuous DRL for trajectory generation [7], [3], where $\tau = 5cm$ was considered.

B. Training Results

First, we evaluate the proposed training optimization, verifying the generalization ability of the DNN for the target position, starting joint position, and step size. Data are collected on an i7-8550U and a GeForce MX150, using the implementation previously described. If not specified, we considered the same set of hyper-parameters provided in the original algorithmic implementation. In order to get reproducible and consistent results, we performed multiple training phases with different random seeds.

In every experiment, we consider the success rate (reported on the x-axis): how many correct trajectories (i.e., a sequence of joint positions that lead the end-effector to the target position with error τ) are generated on a batch of one hundred episodes (reported on the y-axis). An episode ends when a correct trajectory is generated or a fixed timeout of actions is reached (300 in our experiments).

1) Quantitative results in the training phase: Figure 2A shows that the scaling discount factor offers more stability during the training phase with respect to the original DDQN [5]. This result considers $\tau = 5cm$ and the ϵ-greedy exploration policy.

The training stabilizes at over 95% of success rate after about 20000 iterations (i.e., 4 hours). The following results will consider only training performed with this optimization. Figure 2B shows the performance with the mixed policy based on the directional controller. Experimental evaluation shows the best results using the controller actions in 25% of the ϵ cases (a greater value of controller actions causes a drop in performance). This result offers over 90% of success rate after about 30000 iterations. By providing more correct trajectories in the early stages of the training we obtain superior performance faster (i.e., 2 hours). As explained in Section III the controller allows to train the network with $\tau = 2cm$, achieving a similar success rate obtained in the training with $\tau = 5cm$.

When the controller is used in the last part of the trajectory (i.e., when $\tau = 2cm$ is reached), we can train the same network to reach the target with $\tau < 0.1cm$ (Figure 2C). Figure 2D also shows the similarity between an exemplary trajectory to the same target position: one generated by our trained network and the other one computed with the inverse kinematic solver of the Robotic Toolbox[3].

IV. Verification

In this section, we focus on the verification of decision-making models in a DRL scenario, where a neural network is typically used to make a decision. In this formulation, we are not interested in the value that a specific output node assumes, but rather in which action the network selects. To obtain this behavior, our verification approach aims at verifying if a value is greater or lower than another one. In contrast, as detailed in Section II, in a real decision-making application (e.g., trajectory generation), previous techniques for network verification [12], [19] can not directly verify the behavior of the agent in every temporal step.

A. Problem Definition

We define an *area* as a set of bounds, representing for each input node the values that it can assume (limited by an upper and lower bound). As example, in a network with n inputs normalized in range [0, 1], the input area that covers all possible input is formalized as: $area = (a_0, ..., a_{n-1})$ where $a_i = [0, 1]$. A *sub-area* is a further subdivision of the original area and the union of all the sub-areas returns the former area. The propagation of an area (or a sub-area) through a DNN, generates an *output-bound*, which is a set of bounds for each output node that limits the values that the network assumes if the inputs belong to the input area. Figure 3 shows an example of these concepts, where area = $([a, b], [a', b'])$ (i.e., the input area for the verification method). Two possible sub-areas of this are $([a, (a + b)/2], [a', b'])$ and $([(a + b)/2, b], [a', b'])$, notice that the union of the sub-areas, always return the former area. Figure 3 also shows an example of two different bound representations: (i) a simple network with two inputs and one output (Figure 3A). (ii) The corresponding graphical overview (Figure 3B), where the area is bounded by values [a, b] on the first input and by [a', b'] on the second one; the Y curve is the representation of the network output $f(X)$ and [c, d] is the bound for the values that the function can assume.

[3]http://petercorke.com/wordpress/toolboxes/robotics-toolbox

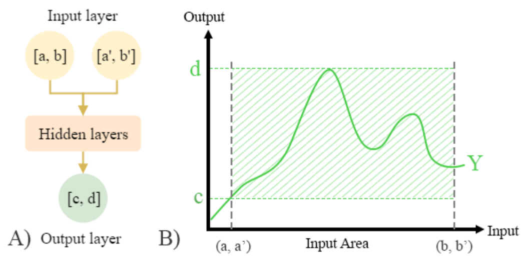


Fig. 3. Example of bound analysis of a generic neural network with 2 inputs and 1 output.

We visualize the DNN function as a 2-dimensional graph, with the inputs on the x-axis (note that a network generally has more than 1 input so we represent a tuple with one value for each input and the order is required only for visualization purpose). The output values corresponding to the input are on the y-axis. Figure 4A shows an example of a bound estimation of a generic output function, where x_0, x_i, x_k, x_n are tuples of size n (where n is the number of input). The typical analysis of previous approaches on the input area from x_0 to x_n returns the bounds (a and b) for the output function y_0.

B. Decision Making Safety Properties

Suppose a simplified navigation scenario and a neural network [4] configured as: (i) *2 outputs*: representing the 2 directions where the agent can move: $y_0 = right$, $y_1 = left$; (ii) *4 inputs*: representing the normalized distance from an obstacle for each direction. Each input could assume a value $x \in [0, 1]$, where 1 means that in the corresponding direction no obstacles are perceived (i.e., distance greater or equal to 1m). Given this configuration we could be interested in a property of the following form:

Θ: If there is an obstacle close to right and other directions are free, between left and right the agent always chooses left.

Considering an obstacle close to the agent (distance is less than $0.07m$), we can formally write this preposition in the following form:

$$\Theta\text{: If } \; x_0 \in [0, 0.07] \wedge x_1 \in \mathcal{D} \wedge x_2 \in \mathcal{D} \wedge x_3 \in \mathcal{D} \\ \Rightarrow y_0 < y_1, \textit{ where } \mathcal{D} = [0.5, 1]$$

In general, Figure 4B shows an example of the bound analysis of a neural network with two outputs y_0, y_1 and n inputs $x_0, ..., x_n$. Assuming that we want to verify a condition in the form:

$$\Theta_1 : \text{If } x_0 \in [a_0, b_0] \wedge ... \wedge x_n \in [a_n, b_n] \Rightarrow y_0 < y_1$$

from the interval algebra presented in [11], we know that:

$$y_0 = [a, b] < y_1 = [c, d] \Rightarrow b < c \tag{2}$$

We can exploit this preposition for our verifier; knowing that the outputs of a network are conditioned by the inputs. To solve the property in the form of Θ_1 we have to ensure that, if the inputs are bounded in the specified range, the values that the outputs could assume always respect the provided condition.

[4]Number and size of the hidden layers, and the choice of the activation function, are not relevant for this example.

Using the input propagation [8], we can compute an overestimation of the bound for the values that the outputs could assume. With this information, our solver can compare the obtained bounds of each output and verify if the values of one of them are strictly lower than another one, which means that the neural network never selects the action represented by the output with the lower values.

The key problem of previous approaches based on the interval analysis [8], [10], is that the output bound estimation is based on the values that an output could assume inside the entire input area and this prevents the application of previous verifiers to a decision-making problem. Indeed, previous verification frameworks applied in a decision-making task, typically find a scenario where $max(y_0) > min(y_1)$, which means that (from the interval analysis) we can not assert anything. In contrast, if we have that $y_0(x) < y_1(x)$ for any $x \in X$ where X is the set of the possible input; which means that, de facto, $y_0 < y_1$.

C. Output Bound Analysis

From the interval algebra [11], $y_1 = [c, d] < y_0 = [a, b] \Rightarrow d < a$. Performing the input area propagation, we find an overestimation of the output bound for each output node that we use to verify a decision making property. However, even if we reduce the overestimation using the bisection method, we showed in the previous section that we compute bounds that are not informative. Figure 4B, shows an example of this situation (note that the real shape of the two output function is impossible to obtain and here we show two explanatory curves). Even with a perfect estimation of the bound and if $y_1 < y_0$ in the given input area, $b \not< c$ so, with (2), we can not formally conclude if the decision making property is proved or denied. To address this, we also exploit the bisection method, but we compute the bounds for each sub-area: (i) to reduce the overestimation (we do not need an accurate estimation in terms of absolute values but reducing the range of the bound is important to make the separation between the output functions more clear) and (ii) to obtain an estimation of the curves. Figure 4C shows this process, where increasing the number of subdivisions we obtain an arbitrary precise estimation of the function. Figure 4D shows how we perform the evaluation. For each generated sub-area we check if the property is respected. For this reason, even with an approximate shape of the output functions, we can assert if a property is respected or not (i.e. if the values of a specific output became greater or lower inside the input area).

1) Combined Properties: Finally, a realistic application of a DRL trained model typically involves more than one output for the model (e.g, in a manipulation task, usually there is at least one output for each joint). Usually, the outputs are related, so we must check a condition over multiple variables.

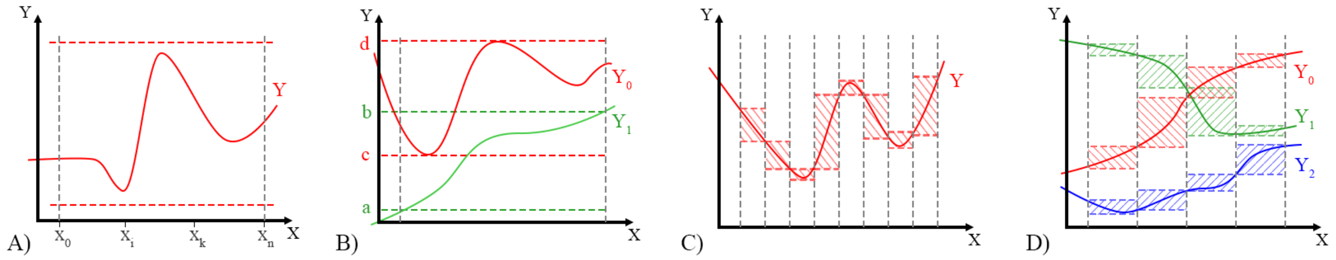


Fig. 4. (A) Bound analysis of a general output function with one cut. (B) Analysis of a decision making problem of two outputs with one cut. (C) Estimation of the shape of a curve with multiple cuts. (D) Decision making analysis with multiple cuts.

In a common scenario we want to prove that the model never selects a specific action over a set of others. Assuming to have 4 outputs (y_0, y_1, y_2, y_3), we could be interested in a property formalized as:

$$\Theta : \text{If } x_0 \in A^n \wedge ... \wedge x_n \in A^n \Rightarrow y_0 < [y_1, y_2, y_3] \text{ , } \textit{where } A^n \textit{ is the input area.}$$

To verify this combined property is necessary that, for each sub-area, one lower bound of y_1, y_2 or y_3 is greater than the upper bound of y_0. The idea is that it is enough that one of the other actions has a greater value in any sub-area, so the agent never chooses the wrong output.

It is also possible to prove that the model selects a specific action over a set of others. Using the previously described scenario, we can formalize this problem as:

$$\Theta : \text{If } x_0 \in A^n \wedge ... \wedge x_n \in A^n \Rightarrow [y_1, y_2, y_3] < y_0$$

In this case, we split the initial property (Θ) in different simple sub-properties ($\Theta^0, \Theta^1, \Theta^2$):

$$\Theta^0 : \text{If } x_{1,...,n} \in A^n \Rightarrow y_1 < y_0$$
$$\Theta^1 : \text{If } x_{1,...,n} \in A^n \Rightarrow y_2 < y_0$$
$$\Theta^2 : \text{If } x_{1,...,n} \in A^n \Rightarrow y_3 < y_0$$

the property is verified if and only if all the sub-properties are verified.

D. Implementation

To solve a property in the form of Θ we subdivide the input area in an arbitrary number of sub-areas and propagate these bounds to finally perform an evaluation for each of them. Algorithm 1 shows a pseudo-code description of our approach. Given as input: (i) the trained neural network to test, (ii) the input area to perform the analysis, (iii) the property to verify, and (iv) the number of subdivisions, in Line 1 we perform the input area splitting. This function uses a simple heuristic that splits the input area with the largest bound. In our evaluation, it performs better than a random strategy. However, this is a crucial step to reduce the number of the subdivision to perform, therefore this is an interesting area for future research. From Line 2 to 5, we perform the propagation described in the previous sections. This is the most computational demanding section of our tool, due to the number of sub-areas that we generate (in our experiment we generate more than 10 million sub-areas to verify the properties). However, each propagation is strongly independent from the others and we parallelize this process on GPU[5]. In Line 7, the tool performs the evaluation of the property for each generated sub-area. This evaluation can produce 3 different results: (i) the property is verified; (ii) the property is denied or (iii) in this area we can not conclude anything on the property and our tool returns *false* (i.e., more subdivisions are required to solve the property). Finally, in Line 15 we return *true* and our tool concludes with a response: if the returned array *denied-areas* is empty we know that the property is verified and no violations are found in the input area, otherwise, we obtain the array with the configuration that causes a wrong evaluation from the network (as discussed in the next section, we can exploit these values to handle undesirable behaviors, obtaining a provably trained model).

Algorithm 1

Input: model to test NET, area where verify the property IA, property to verify PRP, subdivisions to perform N.
Output: TRUE if PRP is verified/denied with the sub-area, FALSE if greater N needed.

```
 1: sub-areas-array ← split-area(IA, N)
 2: denied-areas ← []
 3: for sub-area in sub-areas-array do
 4:     output-bound-array ← get-out-bound(NET, sub-area)
 5: end for
 6: for output-bound in output-bound-array do
 7:     test ← check-property(output-bound, PRP)
 8:     if test is false then
 9:         append sub-area to denied-areas
10:     end if
11:     if PRP is unknown on output-bound then
12:         return: False
13:     end if
14: end for
15: return: True, denied-areas
```

[5]Our results are related to computations performed on an NVIDIA2070 with a *c++* code based on CUDA10.1

E. *Simple controller for property violations:*

After the verification phase, we obtain an array of configurations (i.e., denied-areas) that cause a property violation. We can exploit these values to design a *simple controller* that checks if the current state is in the denied-areas and, in that case, it prevents the undesirable action. 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. As example, we describe the process to decide whether the controller can be designed for the presented task. From the manipulator data-sheet, a step of 2 degrees (i.e. the ω value of our controller) requires ≈ 0.01s to be executed by the arm and, with the violation rate presented in Table 1, a complete search through the array of the sub-areas that cause a violation always requires less than 0.01s. Ideally, it 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 and 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.

V. Experimental Evaluation

In this section, we are interested in proving that our trained model does not lead the manipulator outside its work-space. In a real world scenario, this is a typical condition that robotic arms must respect[6].

A. *Experimental Setup*

To ensure that the manipulator operates inside its work-space, we require that properties describe this behavior: if the current angle of a joint j_i is equal to one of its domain limits, regardless the configuration of the other joints and 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). Based on the network architecture and our problem formalization (described in Section III) we design our properties as follows:

$$\Theta_{0L}\text{: If } \; x_0 \in [1,1] \land x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 \in \mathcal{D} \Rightarrow y_0 < [y_1, y_2, ..., y_{11}], \textit{ where } \mathcal{D} = [0,1)$$

Property Θ_{0L} represents a configuration where the angle of joint j_0 equals to its limit on the left (i.e., a normalized 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 (Θ_{iL}) and one for the right limit (Θ_{iR}). Afterward, we evaluate these properties on two trained networks: one trained without our optimizations (Model A) and one with the scaling discount and the mixed exploration (Model B).

[6]Despite the focus on a trajectory generation task, it is possible to check any properties formalized as a decision-making problem.

Property	**Fail Model A (%)**	**Fail Model B (%)**
Θ_{0L}	0.00	0.17
Θ_{0R}	78.76	0.00
Θ_{1L}	0.00	0.12
Θ_{1R}	0.01	0.00
Θ_{2L}	0.31	0.01
Θ_{2R}	0.00	0.03
Θ_{3L}	68.33	0.50
Θ_{3R}	0.27	0.28
Θ_{4L}	0.04	0.10
Θ_{4R}	0.04	0.03
Θ_{5L}	0.09	0.08
Θ_{5R}	0.08	0.12
TOTAL	12.33	0.12

TABLE I
PROPERTY VERIFICATION RESULTS. *For each property we show the violation rate found by our verification approach. Failure A is related to the standard model, Failure B is related to our optimized approach.*

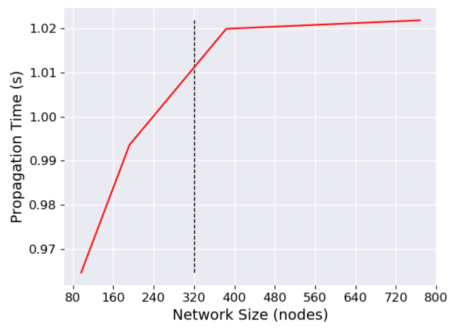
B. *Results*

To evaluate the properties of our models, we designed a metric, which we refer to as *failure rate*, to understand how a trained network performs with respect to a property. In particular, this value represents how many sub-areas cause a violation on the total number of sub-division generated during our formal analysis. Crucially, the failure rate is an upper bound for the rate of failure in the actual execution, and, as such, it can be considered as a worst-case analysis. This is because our analysis assumes an equally distributed probability for a state to belong to one of the sub-areas. However, typically, when executing the trajectory, there are states that appear more frequently while property violations are usually restricted to the limits of the work-space. Summarizing, during the testing phase of a trained model it is possible that a violation will never appear because undesirable behaviors belong to a restricted sub-set of input configurations that rarely occur. Table I shows the collected data and the network trained with our optimizations is overall safer than the other one. In detail, the latter shows a significantly higher percentage of failures for Θ_{0R} and Θ_{3L} (failure rate is $> 65\%$). Furthermore, even if the optimized model has a higher failure rate in some properties, the total number of undesirable behaviors is significantly lower (0.12% in contrast to 12.33%). Crucially, the total failure rate of the optimized model allows the simple controller, described in Section IV, to guarantee that the manipulator never exits from its work-space, without any delay in the robot functioning. In contrast, the same check on the denied-areas for the other model is not feasible without lags in the robot operations[7].

C. *Scalability of the verifier*

Finally, we varied the following parameters: (ii) hidden layers and (iii) neurons, to test the scalability of the proposed verification framework for networks that encode decision-making policies (as discussed in Section II, other verification

[7]From the Panda data-sheet a step of 2 degrees requires ≈ 0.01s and, with our setup, the check of the denied-areas array requires on average less than 0.01s with the optimized model and approximately 1.02s on the other.



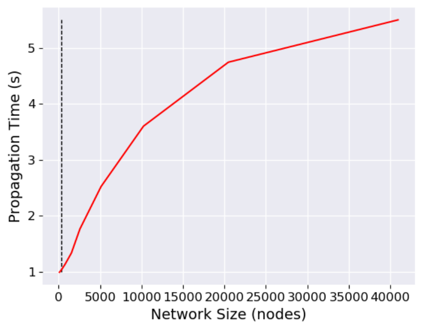


Fig. 5. Time required for the propagation of an input area through network with different sizes, on the x-axis the total number of neurons.

approaches does not scale well with the network size). In detail, we perform the input area propagation phase through networks of different sizes (this is the only process in the method that requires a computation of the input network). Figure 5 shows the results of this analysis, where the vertical line represents the largest network that we used for our task, i.e., 5 hidden layers with 64 neurons each (previous work on DRL [28] states that this is a reasonable size to solve the majority of RL tasks). As expected, the propagation process scales well with the network sizes and does not limit the application on bigger networks.

VI. Discussion

We presented a neural network formal verification method for decision-making problems and further optimizations to improve the performance of the original DDQN algorithm (i.e., a scaling discount factor and a mixed exploration policy). We evaluate our methodologies in a trajectory generation for a commercial robotic manipulator that represents a real-world application of Safe DRL. Crucially, our results show that our approach can verify whether a trained DRL model respects a set of safety properties. Moreover, our empirical evaluation shows that a model trained with the proposed optimization results in lower failure rates, hence allowing a simple controller to guarantee safe execution in real time. This work paves the way for several interesting research directions in the field of Safe DRL, which include exploiting the verification tool in the training phase, to guarantee that the trained model respects the safety properties.

References

[1] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," in *IROS*, 2017.

[2] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-Real: Learning Agile Locomotion For Quadruped Robots," in *RSS*, 2018.

[3] 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 *IRC*, 2019.

[4] O. Vinyals, I. Babuschkin, W. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. Agapiou, M. Jaderberg, and D. Silver, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, 2019.

[5] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," in *AAAI*, 2016.

[6] L. Xie, S. Wang, S. Rosa, A. Markham, and N. Trigoni, "Learning with Training Wheels: Speeding up Training with a Simple Controller for Deep Reinforcement Learning," in *ICRA*, 2018.

[7] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *ICRA*, 2017.

[8] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in *27th {USENIX} Security Symposium*, 2018.

[9] ——, "Efficient formal safety analysis of neural networks," in *NIPS*, 2018.

[10] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proc. ACM Program. Lang.*, 2019.

[11] R. E. Moore, "Interval arithmetic and automatic error analysis in digital computing," Ph.D. dissertation, 1963.

[12] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *Computer Aided Verification*, 2017.

[13] J. Garcıa and F. Fernández, "A comprehensive survey on safe reinforcement learning," *JMLR*, 2015.

[14] R. Cheng, G. Orosz, R. M. Murray, and J. W. Burdick, "End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks," in *AAAI*, 2019.

[15] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *ICLR*, 2015.

[16] L. Engstrom, D. Tsipras, and L. Schmidt, "A rotation and a translation suffice: Fooling cnns with simple transformations," *arXiv*, 2017.

[17] 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 *IEEE SP*, 2018.

[18] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari, "Output range analysis for deep feedforward neural networks," in *arXiv*, 2018.

[19] W. Ruan, X. Huang, and M. Kwiatkowska, "Reachability analysis of deep neural networks with provable guarantees," in *IJCAI*, 2018.

[20] D. Corsi, E. Marchesini, and A. Farinelli, "Evaluating the safety of deep reinforcement learning models using semi-formal verification," in *arXiv*, 2020.

[21] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," in *ICLR*, 2016.

[22] R. Fox, A. Pakman, and N. Tishby, "Taming the Noise in Reinforcement Learning via Soft Updates," in *AUAI*, 2015.

[23] E. Marchesini and A. Farinelli, "Genetic deep reinforcement learning for mapless navigation," in *AAMAS*, 2020.

[24] A. Tavakoli, F. Pardo, and P. Kormushev, "Action Branching Architectures for Deep Reinforcement Learning," in *Association for the Advancement of Artificial Intelligence*, 2018.

[25] E. Marchesini and A. Farinelli, "Discrete deep reinforcement learning for mapless navigation," in *ICRA*, 2020.

[26] C.-T. Chen and W.-D. Chang, "A feedforward neural network with function shape autotuning," in *Neural Networks*, 1996.

[27] V. François-Lavet, R. Fonteneau, and D. Ernst, "How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies," in *NIPS*, 2015.

[28] A. Juliani, V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," in *arXiv*, 2018.