Verified Probabilistic Policies
for Deep Reinforcement Learning

Edoardo Bacci ⓘ and David Parker ⓘ

University of Birmingham, Birmingham, United Kingdom
{exb461,d.a.parker}@bham.ac.uk

Abstract. Deep reinforcement learning is an increasingly popular technique for synthesising policies to control an agent's interaction with its environment. There is also growing interest in formally verifying that such policies are correct and execute safely. Progress has been made in this area by building on existing work for verification of deep neural networks and of continuous-state dynamical systems. In this paper, we tackle the problem of verifying *probabilistic* policies for deep reinforcement learning, which are used to, for example, tackle adversarial environments, break symmetries and manage trade-offs. We propose an abstraction approach, based on interval Markov decision processes, that yields probabilistic guarantees on a policy's execution, and present techniques to build and solve these models using abstract interpretation, mixed-integer linear programming, entropy-based refinement and probabilistic model checking. We implement our approach and illustrate its effectiveness on a selection of reinforcement learning benchmarks.

1 Introduction

Reinforcement learning (RL) is a technique for training a policy used to govern the interaction between an agent and an environment. It is based on repeated explorations of the environment, which yield rewards that the agent should aim to maximise. *Deep reinforcement learning* combines RL and deep learning, by using neural networks to store a representation of a learnt reward function or optimal policy. These methods have been increasingly successful across a wide range of challenging application domains, including for example, autonomous driving [30], robotics [19] and healthcare [49].

In safety critical domains, it is particularly important to assure that policies learnt via RL will be executed safely, which makes the application of *formal verification* to this problem appealing. This is challenging, especially for deep RL, since it requires reasoning about multi-dimensional, continuous state spaces and complex policies encoded as deep neural networks.

There are several approaches to assuring safety in reinforcement learning, often leveraging ideas from formal verification, such as the use of temporal logic to specify safety conditions, or the use of abstract interpretation to build discretised models. One approach is *shielding* (e.g., [1]), which synthesises override mechanisms to prevent the RL agent from acting upon bad decisions; another is

constrained or *safe* RL (e.g. [17]), which generates provably safe policies, typically by restricting the training process to safe explorations.

An alternative approach, which we take in this paper, is to verify an RL policy's correctness after it has been learnt, rather than placing restrictions on the learning process or on its deployment. Progress has been made in the formal verification of policies for RL [6] and also for the specific case of deep RL [28,3,4], in the latter case by building on advances in abstraction and verification techniques for neural networks; [3] also exploits the development of efficient abstract domains such as *template polyhedra* [42], previously applied to the verification of continuous-space and hybrid systems [7,16].

A useful tool in reinforcement learning is the notion of a *probabilistic policy* (or *stochastic policy*), which chooses randomly between available actions in each state, according to a probability distribution specified by the policy. This brings a number of advantages (similarly to mixed strategies [39] in game theory and contextual bandits [34]), such as balancing the exploration-exploitation tradeoff [18], dealing with partial observability of the environment [40], handling multiple objectives [47] or learning continuous actions [38].

In this paper, we tackle the problem of verifying the safety of probabilistic policies for deep reinforcement learning. We define a formal model of their execution using (continuous-state, finite-branching) *discrete-time Markov processes*. We then build and solve sound abstractions of these models. This approach was also taken in earlier work [4], which used Markov decision process abstractions to verify deep RL policies in which actions may exhibit failures.

However, a particular challenge for probabilistic policies, as generated by deep RL, is that policies tend to specify very different action distributions across states. We thus propose a novel abstraction based on *interval Markov decision processes* (IMDPs), in which transitions are labelled with intervals of probabilities, representing the range of possible events that can occur. We solve these IMDPs, over a finite time horizon, which we show yields *probabilistic guarantees*, in the form of upper bounds on the actual probability of the RL policy leading the agent to a state designated to be unsafe.

We present methods to construct IMDP abstractions using template polyhedra as an abstract domain, and mixed-integer linear programming (MILP) to reason symbolically about the neural network policy encoding and a model of the RL agent's environment. We extend existing MILP-based methods for neural networks to cope with the softmax encoding used for probabilistic policies. Naive approaches to constructing these IMDPs yield abstractions that are too coarse, i.e., where the probability intervals are too wide and the resulting safety probability bounds are too high be useful. So, we present an iterative refinement approach based on sampling which splits abstract states via cross-entropy minimisation based on the uncertainty of the over-approximation.

We implement our techniques, building on an extension of the probabilistic model checker PRISM [32] to solve IMDPs. We show that our approach successfully verifies probabilistic policies trained for several reinforcement learning benchmarks and explore trade-offs in precision and computational efficiency.

2

Related work. As discussed above, other approaches to assuring safety in reinforcement learning include shielding [1,5,52,31,25] and constrained or safe RL [17,21,13,45,22,37,26,23]. By contrast, we verify policies independently, without limiting the training process or imposing constraints on execution.

Formal verification of RL, but in a *non-probabilistic* setting includes: [6], which extracts and analyses decision trees; [28], which checks safety and liveness properties for deep RL; and [3], which also uses template polyhedra and MILP to build abstractions, but to check (non-probabilistic) safety invariants.

In the *probabilistic* setting, perhaps closest is our earlier work [4], which uses abstraction for finite-horizon probabilistic verification of deep RL, but for non-probabilistic policies, thus using a simpler (MDP) abstraction, as well as a coarser (interval) abstract domain and a different, more basic approach to refinement. Another approach to generating formal probabilistic guarantees is [14], which, unlike us, does not need a model of the environment and instead learns an approximation and produces probably approximately correct (PAC) guarantees. Probabilistic verification of neural network policies on partially observable models, but for *discrete* state spaces, was considered in [10].

There is also a body of work on verifying continuous space probabilistic models and stochastic hybrid systems, by building finite-state abstractions as, e.g., interval Markov chains [33] or interval MDPs [36,11], but these do not consider control policies encoded as neural networks. Similarly, abstractions of discrete-state probabilistic models use similar ideas to our approach, notably via the use of interval Markov chains [15] and stochastic games [27].

2 Background

We first provide background on the two key probabilistic models used in this paper: *discrete-time Markov processes* (DTMPs), used to model RL policy executions, and *interval Markov decision processes* (IMDPs), used for abstractions.

Notation. We write $Dist(X)$ for the set of discrete probability distributions over a set X, i.e., functions $\mu : X \to [0,1]$ where $\sum_{x \in X} \mu(x) = 1$. The support of μ, denoted $\mathrm{supp}(\mu)$, is defined as $\mathrm{supp}(\mu) = \{x \in X \mid \mu(x) > 0\}$. We use the same notation where X is uncountable but where μ has finite support. We write $\mathcal{P}(X)$ to denote the powerset of X and v^i for the ith element of a vector v.

Definition 1 (Discrete-time Markov process). *A (finite-branching)* discrete-time Markov process *is a tuple* $(S, S_0, \mathbf{P}, AP, L)$, *where:* S *is a (possibly uncountably infinite) set of states;* $S_0 \subseteq S$ *is a set of initial states;* $\mathbf{P} : S \times S \to [0,1]$ *is a transition probability matrix, where* $\sum_{s' \in \mathrm{supp}(\mathbf{P}(s, \cdot))} \mathbf{P}(s, s') = 1$ *for all* $s \in S$; AP *is a set of atomic propositions; and* $L : S \to \mathcal{P}(AP)$ *is a labelling function.*

A DTMP begins in some initial state $s_0 \in S_0$ and then moves between states at discrete time steps. From state s, the probability of making a transition to state s' is $\mathbf{P}(s, s')$. Note that, although the state space of DTMPs used here is continuous, each state only has a finite number of possible successors. This is

always true for our models (where transitions represent policies choosing between a finite number of actions) and simplifies the model.

A *path* through a DTMP is an infinite sequence of states $s_0 s_1 s_2 \ldots$ such that $\mathbf{P}(s_i, s_{i+1}) > 0$ for all i. The set of all paths starting in state s is denoted $Path(s)$ and we define a probability space Pr_s over $Path(s)$ in the usual way [29]. We use atomic propositions (from the set AP) to label states of interest for verification, e.g., to denote them as safe or unsafe. For $b \in AP$, we write $s \models b$ if $b \in L(s)$.

The probability of reaching a b-labelled state from s within k steps is:

$$Pr_s(\Diamond^{\leqslant k} b) = Pr_s(\{s_0 s_1 s_2 \cdots \in Path(s) \mid s_i \models b \text{ for some } 0 \leqslant i \leqslant k\})$$

which, since DTMPs are finite-branching models, can be computed recursively:

$$Pr_s(\Diamond^{\leqslant k} b) = \begin{cases} 1 & \text{if } s \models b \\ 0 & \text{if } s \not\models b \wedge k=0 \\ \sum_{s' \in \mathrm{supp}(\mathbf{P}(s,\cdot))} \mathbf{P}(s, s') \cdot Pr_{s'}(\Diamond^{\leqslant k-1} b) & \text{otherwise.} \end{cases}$$

To build abstractions, we use interval Markov decision processes (IMDPs).

Definition 2 (Interval Markov decision process). *An* interval Markov decision process *is a tuple* $(S, S_0, \mathbf{P}, AP, L)$*, where: S is a finite set of states; $S_0 \subseteq S$ are initial states; $\mathbf{P} : S \times \mathbb{N} \times S \to (\mathbb{I} \cup 0)$ is the interval transition probability function, where $\mathbb{I}$ is the set of probability intervals $\mathbb{I} = \{[a, b] \mid 0 < a \leqslant b \leqslant 1\}$, assigning either a probability interval or the probability exactly 0 to any transition; AP is a set of atomic propositions; and $L{:}S{\to}\mathcal{P}(AP)$ is a labelling function.*

Like a DTMP, an IMDP evolves through states in a state space S, starting from an initial state $s_0 \in S_0$. In each state $s \in S$, an action j must be chosen. Because of the way we use IMDPs, and to avoid confusion with the actions taken by RL policies, we simply use integer indices $j \in \mathbb{N}$ for actions. The probability of moving to each successor state s' then falls within the interval $\mathbf{P}(s, j, s')$.

To reason about IMDPs, we use *policies*, which resolve the nondeterminism in terms of actions and probabilities. A policy σ of the IMDP selects the choice to take in each state, based on the history of its execution so far. In addition, we have a so-called *environment policy* τ which selects probabilities for each transition that fall within the specified intervals. For a policy σ and environment policy τ, we have a probability space $Pr_s^{\sigma,\tau}$ over the set of infinite paths starting in state s. As above, we can define, for example, the probability $Pr_s^{\sigma,\tau}(\Diamond^{\leqslant k} b)$ of reaching a b-labelled state from s within k steps, under σ and τ.

If ψ is an event of interest defined by a measurable set of paths (e.g., $\Diamond^{\leqslant k} b$), we can compute (through *robust value iteration* [48]) lower and upper bounds on, e.g., maximum probabilities, over the set of all allowable probability values:

$$Pr_s^{\max \min}(\psi) = \sup_\sigma \inf_\tau Pr_s^{\sigma,\tau}(\psi) \quad \text{and} \quad Pr_s^{\max,\max}(\psi) = \sup_\sigma \sup_\tau Pr_s^{\sigma,\tau}(\psi)$$

3 Modelling and Abstraction of Reinforcement Learning

We begin by giving a formal definition of our model for the execution of a reinforcement learning system, under the control of a probabilistic policy. We also define the problem of verifying that this policy is executed safely, namely that the probability of visiting an unsafe system state, within a specified time horizon, is below an acceptable threshold.

Then we define abstractions of these models, given an abstract domain over the states of the model, and show how an analysis of the resulting abstraction yields probabilistic guarantees in the form of sound upper bounds on the probability of a failure occurring. In this section, we make no particular assumption about the representation of the policy, nor about the abstract domain.

3.1 Modelling and Verification of Reinforcement Learning

Our model takes the form of a controlled dynamical system over a continuous n-dimensional state space $S \subseteq \mathbb{R}^n$, assuming a finite set of *actions* A performed at discrete time steps. A (time invariant) *environment* $E : S \times A \to S$ describes the effect of executing an action in a state, i.e., if s_t is the state at time t and a_t is the action taken in that state, we have $s_{t+1} = E(s_t, a_t)$.

We assume a reinforcement learning system is controlled by a *probabilistic policy*, i.e., a function of the form $\pi : S \to Dist(A)$, where $\pi(s)(a)$ specifies the probability with which action a should be taken in state s. Since we are interested in verifying the behaviour of a particular policy, not in the problem of learning such a policy, we ignore issues of partial observability. We also do not need to include any definition of rewards.

Furthermore, since our primary interest here is in the treatment of probabilistic policies, we do not consider other sources of stochasticity, such as the agent's perception of its state or the environment's response to an action. Our model could easily be extended with other discrete probabilistic aspects, such as the policy execution failure models considered in [4].

Combining all of the above, we define an *RL execution model* as a (continuous-space, finite-branching) *discrete-time Markov process* (DTMP). In addition to a particular environment E and policy π, we also specify a set $S_0 \subseteq S$ of possible *initial states* and a set $S_{fail} \subseteq S$ of *failure states*, representing *unsafe* states.

Definition 3 (RL execution model). *Assuming a state space $S \subseteq \mathbb{R}^n$ and action set A, and given an environment $E : S \times A \to S$, policy $\pi : S \to Dist(A)$, initial states $S_0 \subseteq S$ and failure states $S_{fail} \subseteq S$, the corresponding* RL execution model *is the DTMP $(S, S_0, \mathbf{P}, AP, L)$ where $AP = \{fail\}$, for any $s \in S$, fail $\in L(s)$ iff $s \in S_{fail}$ and, for states $s, s' \in S$:*

$$\mathbf{P}(s, s') = \sum \{\pi(s)(a) \mid a \in A \ s.t. \ E(s, a) = s'\}.$$

The summation in Definition 3 is required since distinct actions a and a' applied in state s could result in the same successor state s'.

Then, assuming the model above, we define the problem of verifying that an RL policy executes safely. We consider a fixed time horizon $k \in \mathbb{N}$ and an error probability threshold p_{safe}, and the check that the probability of reaching an unsafe state within k time steps is always (from any start state) below p_{safe}.

Definition 4 (RL verification problem). *Given a DTMP model of an RL execution, as in Definition 3, a time horizon $k \in \mathbb{N}$ and a threshold $p_{safe} \in [0, 1]$, the* RL verification problem *is to check that $Pr_s(\lozenge^{\leqslant k} fail) \leqslant p_{safe}$ for all $s \in S_0$.*

In practice, we often tackle a *numerical* version of the verification problem, and instead compute the worst-case probability of error for any start state $p^+ = \inf\{Pr_s(\lozenge^{\leqslant k} fail) \mid s \in S_0\}$ or (as we do later) an upper bound on this value.

3.2 Abstractions for Verification of Reinforcement Learning

Because our models of RL systems are over continuous state spaces, in order to verify them in practice, we construct finite *abstractions*. These represent an over-approximation of the original model, by grouping states with similar behaviour into *abstract states*, belonging to some abstract domain $\hat{S} \subseteq \mathcal{P}(S)$.

Such abstractions are usually necessarily nondeterministic since an abstract state groups states with similar, but distinct, behaviour. For example, abstraction of a probabilistic model such as a discrete-time Markov process could be captured as a Markov decision process [4]. However, a further source of complexity for abstracting *probabilistic policies*, especially those represented as deep neural networks, is that states can also vary widely with regards to the probabilities with which policies select actions in those states.

So, in this work we represent abstractions as *interval MDPs* (IMDPs), in which transitions are labelled with intervals, representing a range of different possible probabilities. We will show that solving the IMDP (i.e., computing the maximum finite-horizon probability of reaching a failure state) yields an upper bound on the corresponding probability for the model being abstracted.

Below, we define this abstraction and state its correctness, first focusing separately on abstractions of an RL system's environment and policy, and then combining these into a single IMDP abstraction.

Assuming an abstract domain $\hat{S} \subseteq \mathcal{P}(S)$, we first require an *environment abstraction* $\hat{E} : \hat{S} \times A \to \hat{S}$, which soundly over-approximates the RL environment $E : S \times A \to S$, as follows.

Definition 5 (Environment abstraction). *For environment $E : S \times A \to S$ and set of abstract states $\hat{S} \subseteq \mathcal{P}(S)$, an* environment abstraction *is a function $\hat{E} : \hat{S} \times A \to \hat{S}$ such that: for any abstract state $\hat{s} \in \hat{S}$, concrete state $s \in \hat{s}$ and action $a \in A$, we have $E(s, a) \in \hat{E}(\hat{s}, a)$.*

Additionally, we need, for any RL policy π, a *policy abstraction* $\hat{\pi}$, which gives a lower and upper bound on the probability with which each action is selected within the states grouped by each abstract state.

Definition 6 (Policy abstraction). *For a policy $\pi : S \rightarrow Dist(A)$ and a set of abstract states $\hat{S} \subseteq \mathcal{P}(S)$, a* policy abstraction *is a pair $(\hat{\pi}_L, \hat{\pi}_U)$ of functions of the form $\hat{\pi}_L : \hat{S} \times A \rightarrow [0,1]$ and $\hat{\pi}_U : \hat{S} \times A \rightarrow [0,1]$, satisfying the following: for any abstract state $\hat{s} \in \hat{S}$, concrete state $s \in \hat{s}$ and action $a \in A$, we have $\hat{\pi}_L(\hat{s}, a) \leqslant \pi(s, a) \leqslant \hat{\pi}_U(\hat{s}, a)$.*

Finally, combining these notions, we can define an *RL execution abstraction*, which is an IMDP abstraction of the execution of an policy in an environment.

Definition 7 (RL execution abstraction). *Let E and π be an RL environment and policy, DTMP $(S, S_0, \mathbf{P}, AP, L)$ be the corresponding RL execution model and $\hat{S} \subseteq \mathcal{P}(S)$ be a set of abstract states. Given also a policy abstraction $\hat{\pi}$ of π and an environment abstraction $\hat{E}$ of E, an* RL execution abstraction *is an IMDP $(\hat{S}, \hat{S}_0, \hat{\mathbf{P}}, AP, \hat{L})$ satisfying the following:*

- *for all $s \in S_0$, $s \in \hat{s}$ for some $\hat{s} \in \hat{S}_0$;*
- *for each $\hat{s} \in \hat{S}$, there is a partition $\{\hat{s}_1, \ldots, \hat{s}_m\}$ of $\hat{s}$ such that, for each $j \in \{1, \ldots, m\}$ we have $\hat{\mathbf{P}}(\hat{s}, j, \hat{s}') = [\hat{\mathbf{P}}_L(\hat{s}, j, \hat{s}'), \hat{\mathbf{P}}_U(\hat{s}, j, \hat{s}')]$ where:*

$$\hat{\mathbf{P}}_L(\hat{s}, j, \hat{s}') = \sum \left\{ \hat{\pi}_L(\hat{s}_j, a) \mid a \in A \text{ s.t. } \hat{E}(\hat{s}_j, a) = \hat{s}' \right\}$$
$$\hat{\mathbf{P}}_U(\hat{s}, j, \hat{s}') = \sum \left\{ \hat{\pi}_U(\hat{s}_j, a) \mid a \in A \text{ s.t. } \hat{E}(\hat{s}_j, a) = \hat{s}' \right\}$$

- *$AP = \{fail\}$ and $fail \in \hat{L}(\hat{s})$ iff $fail \in L(s)$ for some $s \in \hat{s}$.*

Intuitively, each abstract state $\hat{s}$ is partitioned into groups of states $\hat{s}_j$ that behave the same under the specified environment and policy abstractions. The nondeterministic choice between actions $j \in \{1, \ldots, m\}$ in abstract state $\hat{s}$, each of which corresponds to the state subset $\hat{s}_j$, allows the abstraction to overapproximate the behaviour of the original DTMP model.

Finally, we state the correctness of the abstraction, i.e., that solving the IMDP provides upper bounds on the probability of policy execution resulting in a failure. This is formalised as follows (see the appendix for a proof).

Theorem 1. *Given a state $s \in S$ of an RL execution model DTMP, and an abstract state $\hat{s} \in \hat{S}$ of the corresponding abstraction IMDP for which $s \in \hat{s}$:*

$$Pr_s(\lozenge^{\leqslant k} fail) \ \leqslant \ Pr_{\hat{s}}^{\max \max}(\lozenge^{\leqslant k} fail).$$

In particular, this means that we can tackle the RL verification problem of checking that the error probability is below a threshold p_{safe} for all possible start states (see Definition 4). We can do this by finding an abstraction for which $Pr_{\hat{s}}^{\max \max}(\lozenge^{\leqslant k} fail) \leqslant p_{safe}$ for all initial abstract states $\hat{s} \in \hat{S}_0$.

Although $Pr_{\hat{s}}^{\max \min}(\lozenge^{\leqslant k} fail)$ is not necessarily a *lower* bound on the failure probability, the value may still be useful to guide abstraction refinement.

7

4 Template-based Abstraction of Neural Network Policies

We now describe in more detail the process for constructing an IMDP abstraction, as given in Definition 7, to verify the execution of an agent with its environment, under the control of a probabilistic policy. We assume that the policy is encoded in neural network form and has already been learnt, prior to verification, and we use template polyhedra to represent abstract states.

The overall process works by building a k-step unfolding of the IMDP, starting from a set of initial states $\hat{S}_0 \subseteq S$. For each abstract state $\hat{s}$ explored during this process, we need to split $\hat{s}$ into an appropriate partition $\{\hat{s}_1, \ldots, \hat{s}_m\}$. Then, for each $\hat{s}_j \in \hat{s}$ and each action $a \in A$, we determine lower and upper bounds on the probabilities with which a is selected in states in $\hat{s}_j$, i.e., we construct a *policy abstraction* $(\hat{\pi}_L, \hat{\pi}_U)$. We also find the successor abstract state that results from executing a in $\hat{s}_j$, i.e., we build an *environment abstraction* $\hat{E}$. Construction of the IMDP then follows directly from Definition 7.

In the following sections, we describe our techniques in more detail. First, we give brief details of the abstract domain used: bounded polyhedra. Next, we describe how to construct policy abstractions via MILP. Lastly, we describe how to partition abstract states via *refinement*. We omit details of the environment abstraction since we reuse the symbolic post operator over template polyhedra given in [3], also performed with MILP. This supports environments specified as linear, piecewise linear or non-linear systems defined with polynomial and transcendental functions. The latter is dealt with using linearisation, subdividing into small intervals and over-approximating using interval arithmetic.

Further details of the algorithms in this section can be found in [2].

4.1 Bounded Template Polyhedra

Recall that the state space of our model $S \subseteq \mathbb{R}^n$ is over n real-valued variables. We represent abstract states using *template polyhedra* [42], which are convex subsets of $\mathbb{R}^n$, defined by constraints in a finite set of *directions* $\Delta \subset \mathbb{R}^n$ (in other words, the facets of the polyhedra are normal to the directions in Δ). We call a fixed set of directions $\Delta \subset \mathbb{R}^n$ a *template*.

Given a (convex) abstract state $\hat{s} \subseteq \mathbb{R}^n$, a Δ-polyhedron of $\hat{s}$ is defined as the tightest Δ-polyhedron enclosing $\hat{s}$:

$$\cap\{\{s \colon \langle \delta, s \rangle \leqslant \sup\{\langle \delta, s \rangle \colon s \in \hat{s}\}\} \colon \delta \in \Delta\},$$

where $\langle \cdot, \cdot \rangle$ denotes scalar product. In this paper, we restrict our attention to *bounded* template polyhedra (also called *polytopes*), in which every variable in the state space is bounded by a direction of the template, since this is needed for our refinement scheme.

Important special cases of template polyhedra are *rectangles* (i.e., intervals) and *octagons*. Later, in Section 5, we will present an empirical comparison of these different abstract domains applied to our setting, and show the benefits of the more general case of template polyhedra.

4.2 Constructing Policy Abstractions

We focus first on the abstraction of the RL policy $\pi : S \to Dist(A)$, assuming there are k actions: $A = \{a_1, \ldots, a_k\}$. Let π be encoded by a neural network comprising n input neurons, l hidden layers, each containing h_i neurons ($1 \leqslant i \leqslant l$), and k output neurons, and using ReLU activation functions.

The policy is encoded as follows. We use variable vectors $z_0 \ldots, z_{l+1}$ to denote the values of the neurons at each layer. The current state of the environment is fed to the input layer z_0, each hidden layer's values are as follows:

$$z_i = \text{ReLU}(W_i z_{i-1} + b_i) \text{ for } i = 1, \ldots, l$$

and the output layer is $z_{l+1} = W_{l+1} z_l$, where each W_i is a matrix of weights connecting layers $i-1$ and i and each b_i is a vector of biases. In the usual fashion, $\text{ReLU}(z) = \max(z, 0)$. Finally, the k output neurons yield the probability assigned by the policy to each action. More precisely, the probability that the encoded policy selects action a_j is given by p_j based on a softmax normalisation of the output layer:

$$p_j = \text{softmax}(z_{l+1})^j = \frac{e^{z_{l+1}^j}}{\sum_{i=1}^k e^{z_{l+1}^i}}$$

For an abstract state $\hat{s}$, we compute the policy abstraction, i.e., lower and upper bounds $\hat{\pi}_L(\hat{s}, a_j)$ and $\hat{\pi}_U(\hat{s}, a_j)$ for all actions a_j (see Definition 6), via mixed-integer linear programming (MILP), building on existing MILP encodings of neural networks [46,12,9]. The probability bounds cannot be directly computed via MILP due to the nonlinearity of the softmax function so, as a proxy, we maximise the corresponding entry (the jth logit) of the output layer ($l+1$). For the upper bound (the lower bound is computed analogously), we optimise:

$$
\begin{aligned}
\text{maximize} \quad & z_{l+1}^j \\
\text{subject to} \quad & z_0 \in \hat{s}, \\
& 0 \leqslant z_i - W_i z_{i-1} - b_i \leqslant M z_i' \quad \text{for } i = 1, \ldots, l, \\
& 0 \leqslant z_i \leqslant M - M z_i' \quad \text{for } i = 1, \ldots, l, \\
& 0 \leqslant z_i' \leqslant 1 \quad \text{for } i = 1, \ldots, l, \\
& z_{l+1} = W_{l+1} z_l,
\end{aligned}
\tag{1}
$$

over the variables $z_0 \in \mathbb{R}^n$, $z_{l+1} \in \mathbb{R}^k$ and $z_i \in \mathbb{R}^{h_i}$, $z_i' \in \mathbb{Z}^{h_i}$ for $1 \leqslant i \leqslant l$.

Since abstract state $\hat{s}$ is a convex polyhedron, the initial constraint $z_0 \in \hat{s}$ on the vector of values z_0 fed to the input layer is represented by $|\Delta|$ linear inequalities. ReLU functions are modelled using a big-M encoding [46], where we add integer variable vectors z_i' and $M \in \mathbb{R}$ is a constant representing an upper bound for the possible values of neurons.

We solve $2k$ MILPs to obtain lower and upper bounds on the logits for all k actions. We then calculate bounds on the probabilities of each action by combining these values as described below. Since the exponential function in

softmax is monotonic, it preserves the order of the intervals, allowing us to compute the bounds on the probabilities achievable in $\hat{s}$.

Let $x_{lb,i}$ and $x_{ub,i}$ denote the lower and upper bounds, respectively, obtained for each action a_i via MILP (i.e., the optimised values z_{l+1}^i in (1) above). Then, the upper bound for the probability of choosing action a_j is $y_{ub,j}$:

$$y_{ub,j} = \text{softmax}(z_{ub,j}) \qquad \text{where} \qquad z_{ub,j}^i = \left\{ \begin{array}{ll} x_{ub,i} & \text{if } i = j \\ 1 - x_{lb,i} & \text{otherwise} \end{array} \right.$$

and where $z_{ub,j}$ is an intermediate vector of size k. Again, the computation for the lower bound is performed analogously.

4.3 Refinement of Abstract States

As discussed above, each abstract state $\hat{s}$ in the IMDP is split into a partition $\{\hat{s}_1, \ldots, \hat{s}_m\}$ and, for each $\hat{s}_i$, the probability bounds $\hat{\pi}_L(\hat{s}_i, a)$ and $\hat{\pi}_U(\hat{s}_i, a)$ are determined for each action a. If these intervals are two wide, the abstraction is too coarse and the results uninformative. To determine a good partition (i.e., one that groups states with similar behaviour in terms of the probabilities chosen by the policy), we use *refinement*, repeatedly splitting $\hat{s}_i$ into finer partitions.

We define the *maximum probability spread* of $\hat{s}_i$, denoted $\Delta_{\hat{\pi}}^{\max}(\hat{s}_i)$, as:

$$\Delta_{\hat{\pi}}^{\max}(\hat{s}_i) = \max_{a \in A}(\hat{\pi}_U(\hat{s}_i, a) - \hat{\pi}_L(\hat{s}_i, a))$$

and we refine $\hat{s}_i$ until $\Delta_{\hat{\pi}}^{\max}(\hat{s}_i)$ falls below a specified threshold ϕ. Varying ϕ allows us to tune the desired degree of precision.

When refining, our aim is minimise $\Delta_{\hat{\pi}}^{\max}(\hat{s}_i)$, i.e., to group areas of the state space that have similar probability ranges, but also to minimise the number of splits performed. We try to find a good compromise between improving the accuracy of the abstraction and reducing partition growth, which generates additional abstract states and increases the size of the IMDP abstraction.

Calculating the range $\Delta_{\hat{\pi}}^{\max}(\hat{s}_i)$ can be done by using MILP to compute each of the lower and upper bounds $\hat{\pi}_L(\hat{s}_i, a)$ and $\hat{\pi}_U(\hat{s}_i, a)$. However, this may be time consuming. So, during the first part of refinement for each abstract state, we sample probabilities for some states to compute an underestimate of the true range. If the sampled range is already wide enough to trigger further refinement, we do so; otherwise we calculate the exact range of probabilities using MILP to check whether there is a need for further refinement.

Each refinement step comprises three phases, described in more detail below: (i) sampling policy probabilities; (ii) selecting a direction to split; (iii) splitting. Figure 1 gives an illustrative example of a full refinement.

Sampling the neural network policy. We first generate a sample of the probabilities chosen by the policy within the abstract state. Since this is a convex region, we sample state points within it randomly using the Hit & Run method [44]. We then obtain, from the neural network, the probabilities of picking actions at each sampled state. We consider each action a separately, and
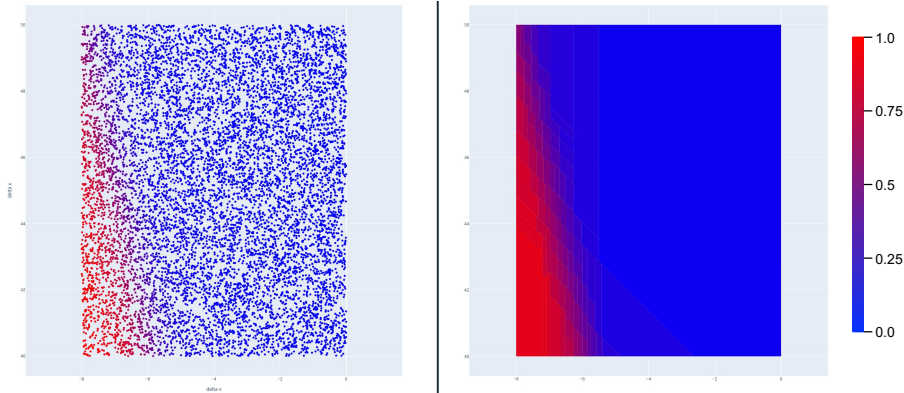
Fig. 1: Sampled policy probabilities for one action in an abstract state (left) and the template polyhedra partition generated through refinement (right).

then later split according to the most promising one (i.e., with the widest probability spread across all actions). The probabilities for each a are computed in a *one-vs-all* fashion: we generate a point cloud representing the probability of taking that action as opposed to any other action.

The number of samples used (and hence the time needed) is kept fixed, rather than fixing the density of the sampled points. We sample 1000 points per abstract state split but this parameter can be tuned depending on the machine and the desired time/accuracy tradeoff. This ensures that ever more accurate approximations are generated as the size of the polyhedra decreases.

Choosing candidate directions. We refine abstract states (represented as template polyhedra) by bisecting them along a chosen direction from the set Δ used to define them. Since the polyhedra are bounded, we are free to pick any one. To find the direction that contributes most to reducing the probability spread, we use cross-entropy minimisation to find the optimal boundary at which to split each direction, and then pick the direction that yields the lowest value.

Let $\tilde{S}$ be the set of sampled points and $\tilde{Y}_s$ denote the true probability of choosing action a in each point $s \in \tilde{S}$, as extracted from the probabilistic policy. For a direction δ, we project all points in $\tilde{S}$ onto δ and sort them accordingly, i.e., we let $\tilde{S} = \{s_1, \ldots, s_m\}$, where $m = |\tilde{S}|$ and index i is sorted by $\langle \delta, s_i \rangle$. We determine the optimal boundary for splitting in direction δ by finding the optimal index k that splits $\tilde{S}$ into $\{s_1, \ldots, s_k\}$ and $\{s_{k+1}, \ldots, s_m\}$. To do so, we first define the function $Y_i^{k,\delta}$ classifying the ith point according to this split:

$$Y_i^{k,\delta} = \begin{cases} 1 \text{ if } i \leqslant k \\ 0 \text{ if } i > k \end{cases}$$

and then minimise, over k, the binary cross entropy loss function:

$$H(Y^{k,\delta}, \tilde{Y}) = -\frac{1}{m} \sum_{i=1}^{m} \left(Y_i^{k,\delta} \log(\tilde{Y}_{s_i}) + (1 - Y_i^{k,\delta}) \log(1 - \tilde{Y}_{s_i}) \right)$$

11

which reflects how well the true probability for each point $\tilde{Y}_s$ matches the separation into the two groups.

One problem with this approach is that, if the distribution of probabilities is skewed to strongly favour some probabilities, a good decision boundary may not be picked. To counter this, we perform sample weighting by grouping the sampled probabilities into small bins, and counting the number of samples in each bin to calculate how much weight to give to each sample.

Abstract state splitting. Once a direction δ and bisection point s_k are chosen, the abstract state is split into two with a corresponding pair of constraints that splits the polyhedron. Because we are constrained to the directions of the template, and the decision boundary is highly non-linear, sometimes the bisection point falls close to the interval boundary and the resulting slices are extremely thin. This would cause the creation of an unnecessarily high number of polyhedra, which we prevent by imposing a minimum size of the split relative to the dimension chosen. By doing so we are guaranteed a minimum degree of progress and the complex shapes in the non-linear policy space which are not easily classified (such as non-convex shapes) are broken down into more manageable regions.

5   Experimental Evaluation

We evaluate our approach by implementing the techniques described in Section 4 and applying them to 3 reinforcement learning benchmarks, analysing performance and the impact of various configurations and optimisations.

5.1   Experimental Setup

Implementation. The code is developed in a mixture of Python and Java. Neural network manipulation is done through Pytorch [51], MILP solution through Gurobi [20], graph analysis with `networkX` [50] and cross-entropy minimisation with Scikit-learn [41]. IMDPs are constructed and solved using an extension of PRISM [32] which implements robust value iteration [48]. The code is available from https://github.com/phate09/SafeDRL.

Benchmarks. We use the following three RL benchmark environments:

(i) Bouncing ball [24]: The agent controls a ball with height p and vertical velocity v, choosing to either hit the ball downward with a paddle, adding speed, or do nothing. The ball accelerates while falling and bounces on the ground losing 10% of its energy; it eventually stops bouncing if its height is too low and it is out of reach of the paddle. The initial heights and speed vary. In our experiments, we consider two possible starting regions: "large" ($S_0 = L$), where $p \in [5, 9]$ and $v \in [-1, 1]$, and "small" ($S_0 = S$), where $p \in [5, 9]$ and $v \in [-0.1, 0]$. The safety constraint is that the ball never stops bouncing.

(ii) Adaptive cruise control [3]: The problem has two vehicles $i \in \{lead, ego\}$, whose state is determined by variables x_i and v_i for the position and speed of

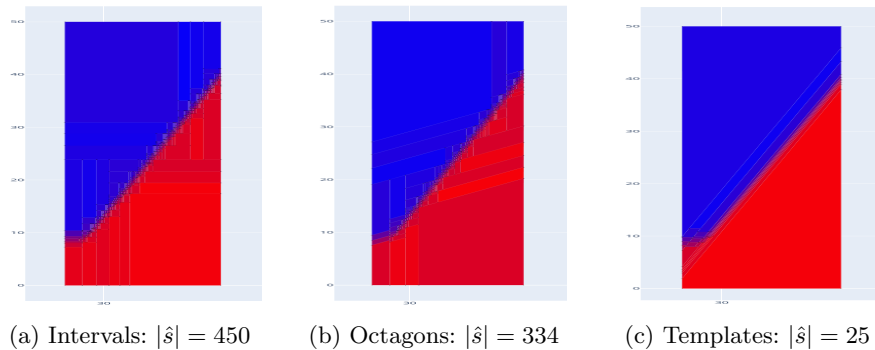(a) Intervals: $|\hat{s}| = 450$ (b) Octagons: $|\hat{s}| = 334$ (c) Templates: $|\hat{s}| = 25$

Fig. 2: Policy abstractions for an abstract state from the adaptive cruise control benchmark, using different abstract domains (see Figure 1 for legend).

each car, respectively. The lead car proceeds at constant speed $(28\,\mathrm{m\,s^{-1}})$, and the agent controls the acceleration $(\pm 1\,\mathrm{m\,s^{-2}})$ of *ego* using two actions. The range of possible start states allows a relative distance of $[3, 10]$ metres and the speed of the ego vehicle is in $[26, 32]$ m/s. Safety means preserving $x_{lead} \geqslant x_{ego}$.

(iii) Inverted pendulum: This benchmark is a modified (discrete action) version of the "Pendulum-v0" environment from the OpenAI Gym [8] where an agent applies left or right rotational force to a pole pivoting around one of its ends, with the aim of balancing the pole in an upright position. The state is modelled by 2 variables: the angular position and velocity of the pole. We consider initial conditions of an angle $[-0.05, 0.05]$ and speed $[-0.05, 0.05]$. Safety constitutes remaining within a range of positions and velocities such that an upright position can be recovered. This benchmark is more challenging than the previous two: it allows 3 actions (noop, push left, push right) and the dynamics of the system are highly non-linear, making the problem more complex.

Policy training. All agents have been trained using proximal policy optimisation (PPO) [43] in actor-critic configuration with Adam optimiser. The training is distributed over 8 actors with 10 instances of each environment, managing the collection of results and the update of the network with `RLlib` [35]. Hyperparameters have been mostly kept unchanged from their default values except the learning rate and batch size which have been set to 5×10^{-4} and 4096, respectively. We used a standard feed forward architecture with 2 hidden layers (size 32 for the bouncing ball and size 64 for the adaptive cruise control and inverted pendulum problems) and ReLU activation functions.

Abstract domains. The abstraction techniques we present in Section 4 are based on the use of template polyhedra as an abstract domain. As special cases, this includes rectangles (intervals) and octagons. We use both of these in our experiments, but also the more general case of arbitrary bounded template polyhedra. In the latter case, we choose a set of directions by sampling a representative portion of the state space where the agent is expected to operate, and

13

Benchmark environment	k	Abs. dom.	ϕ	Contain. check	Num. poly.	Num. visited	IMDP size	Prob. bound	Runtime (min.)
Bouncing ball ($S_0 = $ S)	20	Rect	0.1	✓	337	28	411	0.0	1
	20	Oct	0.1	✓	352	66	484	0.0	2
Bouncing ball ($S_0 = $ L)	20	Rect	0.1	✓	1727	5534	7796	0.63	30
	20	Oct	0.1	✓	2489	3045	6273	0.0	33
	20	Rect	0.1	✗	18890	0	23337	0.006	91
	20	Oct	0.1	✗	13437	0	16837	0.0	111
Adaptive cruise control	7	Rect	0.33	✓	1522	4770	10702	0.084	85
	7	Oct	0.33	✓	1415	2299	6394	0.078	60
	7	Temp	0.33	✓	2440	2475	9234	0.47	70
	7	Rect	0.5	✓	593	1589	3776	0.62	29
	7	Oct	0.5	✓	801	881	3063	0.12	30
	7	Temp	0.5	✓	1102	1079	4045	0.53	34
	7	Rect	0.33	✗	11334	0	24184	0.040	176
	7	Oct	0.33	✗	7609	0	16899	0.031	152
	7	Temp	0.33	✗	6710	0	14626	0.038	113
	7	Rect	0.5	✗	3981	0	8395	0.17	64
	7	Oct	0.5	✗	2662	0	5895	0.12	52
	7	Temp	0.5	✗	2809	0	6178	0.16	48
Inverted pendulum	6	Rect	0.5	✓	1494	3788	14726	0.057	71
	6	Rect	0.5	✗	5436	0	16695	0.057	69

Table 1: Verification results for the benchmark environments

choosing appropriate slopes for the directions to better represents the decision boundaries. The effect of the choice of different template can be seen in Fig 2 where we show a representative abstract state and how the refinement algorithm is affected by the choice of template: as expected, increasing the generality of the abstract domain results in a smaller number of abstract states.

Containment checks. Lastly, we describe an optimisation implemented for construction of IMDP abstractions, whose effectiveness we will evaluate in the next section. When calculating the successors of abstract states to construct an IMDP, we sometimes find that successors that are partially or fully contained within previously visited abstract states. Against the possible trade-off of decreasing the accuracy of the abstraction, we can attempt to reduce the total size of the IMDP that is constructed by aggregating together states which are fully contained within previously visited abstract states.

5.2 Experimental Results

Table 1 summarises the experimental results across the different benchmark environments; k denotes the time horizon considered. We use a range of configurations, varying: the abstract domain used (rectangles, octagons or general template polyhedra); the maximum probability spread threshold ϕ and whether the containment check optimisation is used.

The table lists, for each case: the number of independent polyhedra generated, the number of instances in which polyhedra are contained in previously visited abstract states and aggregated together; the final size of the IMDP abstraction (number of abstract states); the generated upper bound on the probability of encountering an unsafe state from an initial state; and the runtime of the whole process. Experiments were run on a 4-core 4.2 GHz PC with 64 GB RAM.

Verification successfully produced probability bounds for all environments considered. Typically, the values of k shown are the largest time horizons we could check, assuming a 3 hour timeout for verification. The majority of the runtime is for constructing the abstraction, not solving the IMDP.

As can be seen, the various configurations result in different safety probability bounds and runtimes for the same environments, so we are primarily interested in the impact that these choices have on the trade-off between abstraction precision and performance. We summarise findings for each benchmark separately.

Bouncing ball. These are the quickest abstractions to construct and verify due to the low number of variables and the simplicity of the dynamics. For both initial regions considered, we can actually verify that it is fully safe (maximum probability 0). However, for the larger one, rectangles (particular with containment checks) are not accurate enough to show this.

Two main areas of the policy are identified for refinement: one where it can reach the ball and should hit it and one where the ball is out of reach and the paddle should not be activated to preserve energy. But even for threshold $\phi = 0.1$ (lower than used for other benchmarks), rectangular abstractions resulted in large abstract states containing most of the other states visited by the agent, and which ultimately overlapped with the unsafe region.

Adaptive cruise control. On this benchmark, we use a wider range of configurations. Firstly, as expected, for smaller values of the maximum probability spread threshold ϕ, the probability bound obtained is lower (the overestimation error from the abstraction decreases, making it closer to the true maximum probability) but the abstraction size and runtime increase. Applying the containment check for previously visited states has a similar effect: it helps reduce the computation time, but at the expense of overapproximation (higher bounds)

The choice of abstract domain also has a significant impact. Octagons yield more precise results than rectangles, for the same values of ϕ, and also produce smaller abstractions (and therefore lower runtime). On the other hand, general template polyhedra (chosen to better approximate the decision boundary) do not appear to provide an improvement in time or precision on this example, instead causing higher probability bounds, especially when combined with the containment check. Our hypothesis is that this abstract domains groups large areas of the state space (as shown in Fig. 2) and this eventually leads to overlaps with the unsafe region.

Inverted pendulum. This benchmark is more challenging and, while we successfully generate bounds on the probability of unsafe behaviour, for smaller values of ϕ and other abstract domains, experiments timed out due to the high
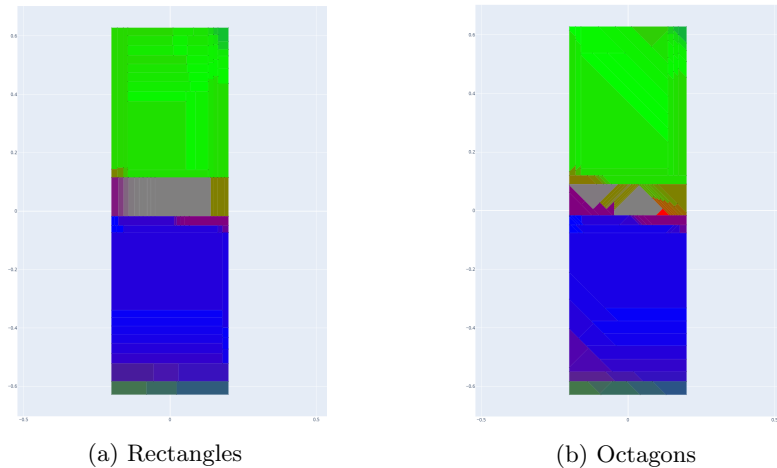
(a) Rectangles   (b) Octagons

Fig. 3: Refined policy abstractions from the inverted pendulum benchmark

number of abstract states generated and the time needed for MILP solution. The abstract states generated were sufficiently small that the containment check could be used to reduce runtime without increasing the probability bound.

Figure 3 illustrates abstraction applied to a state space fragment from this benchmark using both rectangles and octagons. It shows the probability of choosing one of three actions, coded by RGB colour: *noop* (red), *right* (green) and *left* (blue), The X axis represents angular speed and the Y axis represents the angle of the pendulum in radians. Notice the grey area towards the centre where all 3 actions have the same probability, the centre right area with yellow tints (red and green), and the centre left area with purple tints (red and blue). Towards the bottom of the heatmap, the colour fades to green as the agent tries to push the pendulum so that it spins and balances once it reaches the opposite side.

6   Conclusion

We presented an approach for verifying probabilistic policies for deep reinforcement learning agents. This is based on a formal model of their execution as continuous-space discrete time Markov process, and a novel abstraction represented as an interval MDP. We propose techniques to implement this framework with MILP and a sampling-based refinement method using cross-entropy minimisation. Experiments on several RL benchmarks illustrate its effectiveness and show how we can tune the approach to trade off accuracy and performance.

Future work includes automating the selection of an appropriate template for abstraction and using lower bounds from the abstraction to improve refinement.

Appendix: Proof of Theorem 1

We provide here a proof of Theorem 1, from Section 3, which states that:

Given a state $s \in S$ of an RL execution model DTMP, and abstract state $\hat{s} \in \hat{S}$ of the corresponding controller abstraction IMDP for which $s \in \hat{s}$, we have:

$$Pr_s(\lozenge^{\leqslant k} fail) \;\leqslant\; Pr_{\hat{s}}^{\max\max}(\lozenge^{\leqslant k} fail)$$

By the definition of $Pr_{\hat{s}}^{\max\max}(\cdot)$, it suffices to show that there is *some* policy σ and *some* environment policy τ in the IMDP such that:

$$Pr_s(\lozenge^{\leqslant k} fail) \;\leqslant\; Pr_{\hat{s}}^{\sigma,\tau}(\lozenge^{\leqslant k} fail) \tag{2}$$

Recall that, in the construction of the IMDP (see Definition 7), an abstract state $\hat{s}$ is associated with a partition of subsets $\hat{s}_j$ of $\hat{s}$, each of which is used to define the j-labelled choice in state $\hat{s}$. Let σ be the policy that picks in each state s (regardless of history) the unique index j_s such that $s \in \hat{s}_{j_s}$. Then, let τ be the environment policy that selects the upper bound of the interval for every transition probability. We use function $\hat{\mathbf{P}}_\tau$ to denote the chosen probabilities, i.e., we have $\hat{\mathbf{P}}_\tau(\hat{s}, j_s, \hat{s}') = \hat{\mathbf{P}}_U(\hat{s}, j_s, \hat{s}')$ for any $\hat{s}, j_s, \hat{s}'$.

The probabilities $Pr_{\hat{s}}^{\sigma,\tau}(\lozenge^{\leqslant k} fail)$ for these policies, starting in $\hat{s}$, are defined similarly to those for discrete-time Markov processes (see Section 2):

$$Pr_{\hat{s}}^{\sigma,\tau}(\lozenge^{\leqslant k} fail) = \begin{cases} 1 & \text{if } \hat{s} \models fail \\ 0 & \text{if } \hat{s} \not\models fail \wedge k=0 \\ \displaystyle\sum_{\hat{s}' \in \mathrm{supp}(\hat{\mathbf{P}}(\hat{s}, j_s, \cdot))} \hat{\mathbf{P}}(\hat{s}, j_s, \hat{s}') \cdot Pr_{\hat{s}'}^{\sigma,\tau}(\lozenge^{\leqslant k-1} fail) & \text{otherwise.} \end{cases}$$

Since this is defined recursively, we prove (2) by induction over k. For the case $k = 0$, the definitions of $Pr_s(\lozenge^{\leqslant 0} fail)$ and $Pr_{\hat{s}}(\lozenge^{\leqslant 0} fail)$ are equivalent: they equal 1 if $s \models fail$ (or $\hat{s} \models fail$) and 0 otherwise. From Definition 7, $s \models fail$ implies $\hat{s} \models fail$. Therefore, $Pr_s(\lozenge^{\leqslant 0} fail) \;\leqslant\; Pr_{\hat{s}}^{\sigma,\tau}(\lozenge^{\leqslant 0} fail)$.

Next, for the inductive step, we will assume, as the inductive hypothesis, that $Pr_{s'}(\lozenge^{\leqslant k-1} fail) \;\leqslant\; Pr_{\hat{s}'}^{\sigma,\tau}(\lozenge^{\leqslant k-1} fail)$ for $s' \in S$ and $\hat{s}' \in \hat{S}$ with $s' \in \hat{s}'$. If $\hat{s} \models fail$ then $Pr_{\hat{s}}^{\sigma,\tau}(\lozenge^{\leqslant k} fail) = 1 \;\geqslant\; Pr_s(\lozenge^{\leqslant k} fail)$. Otherwise we have:

$$
\begin{aligned}
&Pr_{\hat{s}}^{\sigma,\tau}(\lozenge^{\leqslant k} fail) \\
&= \textstyle\sum_{\hat{s}' \in \mathrm{supp}(\hat{\mathbf{P}}_\tau(\hat{s}, j_s, \cdot))} \hat{\mathbf{P}}_\tau(\hat{s}, j_s, \hat{s}') \cdot Pr_{\hat{s}'}(\lozenge^{\leqslant k-1} fail) && \text{by defn. of } \sigma \text{ and } Pr_{\hat{s}}^{\sigma,\tau}(\lozenge^{\leqslant k} fail) \\
&= \textstyle\sum_{\hat{s}' \in \mathrm{supp}(\hat{\mathbf{P}}_U(\hat{s}, j_s, \cdot))} \hat{\mathbf{P}}_U(\hat{s}, j_s, \hat{s}') \cdot Pr_{\hat{s}'}(\lozenge^{\leqslant k-1} fail) && \text{by defn. of } \tau \\
&= \textstyle\sum_{a \in A} \pi_U(\hat{s}, a) \cdot Pr_{\hat{E}(\hat{s}_j, a)}(\lozenge^{\leqslant k-1} fail) && \text{by defn. of } \hat{\mathbf{P}}_U(\hat{s}, j, \hat{s}') \\
&\geqslant \textstyle\sum_{a \in A} \pi(s, a) \cdot Pr_{\hat{E}(\hat{s}_j, a)}(\lozenge^{\leqslant k-1} fail) && \text{since } s \in \hat{s} \text{ and by Defn.6} \\
&\geqslant \textstyle\sum_{a \in A} \pi(s, a) \cdot Pr_{E(s,a)}(\lozenge^{\leqslant k-1} fail) && \text{by induction and since, by} \\
&&& \text{Defn. 5, } E(s, w) \in \hat{E}(\hat{s}_j, w) \\
&= \textstyle\sum_{s' \in \mathrm{supp}(\mathbf{P}(s, \cdot))} \mathbf{P}(s, s') \cdot Pr_{s'}(\lozenge^{\leqslant k-1} fail) && \text{by defn. of } \mathbf{P}(s, s') \\
&= Pr_s(\lozenge^{\leqslant k} fail) && \text{by defn. of } Pr_s(\lozenge^{\leqslant k} fail)
\end{aligned}
$$

which completes the proof.

References

1. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: Proc. 32nd AAAI Conference on Artificial Intelligence (AAAI'18). pp. 2669–2678 (2018)
2. Bacci, E.: Formal Verification of Deep Reinforcement Learning Agents. Ph.D. thesis, School of Computer Science, University of Birmingham (2022)
3. Bacci, E., Giacobbe, M., Parker, D.: Verifying reinforcement learning up to infinity. In: Proc. 30th International Joint Conference on Artificial Intelligence (IJCAI'21). pp. 2154–2160 (2021)
4. Bacci, E., Parker, D.: Probabilistic guarantees for safe deep reinforcement learning. In: Proc. 18th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'20). LNCS, vol. 12288, pp. 231–248. Springer (2020)
5. Bastani, O.: Safe Reinforcement Learning with Nonlinear Dynamics via Model Predictive Shielding. In: Proceedings of the American Control Conference. pp. 3488–3494 (2021)
6. Bastani, O., Pu, Y., Solar-Lezama, A.: Verifiable reinforcement learning via policy extraction. In: Proc. 2018 Annual Conference on Neural Information Processing Systems (NeurIPS'18). pp. 2499–2509 (2018)
7. Bogomolov, S., Frehse, G., Giacobbe, M., Henzinger, T.A.: Counterexample-guided refinement of template polyhedra. In: TACAS (1). pp. 589–606 (2017)
8. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: OpenAI Gym (6 2016)
9. Bunel, R., Turkaslan, I., Torr, P., Kohli, P., Kumar, P.: A unified view of piecewise linear neural network verification. In: Proc. 32nd International Conference on Neural Information Processing Systems (NIPS'18). pp. 4795–4804 (2018)
10. Carr, S., Jansen, N., Topcu, U.: Task-aware verifiable RNN-based policies for partially observable Markov decision processes. Journal of Artificial Intelligence Research **72**, 819–847 (2021)
11. Cauchi, N., Laurenti, L., Lahijanian, M., Abate, A., Kwiatkowska, M., Cardelli, L.: Efficiency through uncertainty: Scalable formal synthesis for stochastic hybrid systems. In: 22nd ACM International Conference on Hybrid Systems: Computation and Control (2019)
12. Cheng, C.H., Nührenberg, G., Ruess, H.: Maximum resilience of artificial neural networks. In: Proc. International Symposium on Automated Technology for Verification and Analysis (ATVA'17). LNCS, vol. 10482, pp. 251–268 (2017)
13. Cheng, R., Orosz, G., Murray, R.M., Burdick, J.W.: End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks. In: AAAI. pp. 3387–3395. AAAI Press (2019)
14. Delgrange, F., Ann Nowe, G.A.P.: Distillation of RL policies with formal guarantees via variational abstraction of markov decision processes. In: Proc. 36th AAAI Conference on Artificial Intelligence (AAAI'22) (2022)
15. Fecher, H., Leucker, M., Wolf, V.: Don't know in probabilistic systems. In: Valmari, A. (ed.) Proc. SPIN'06. LNCS, vol. 3925, pp. 71–88. Springer (2006)
16. Frehse, G., Giacobbe, M., Henzinger, T.A.: Space-time interpolants. In: CAV (1). pp. 468–486. Springer (2018)
17. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In: AAAI. pp. 6485–6492. AAAI Press (2018)

18. García, J., Fernández, F.: Probabilistic policy reuse for safe reinforcement learning. ACM Transactions on Autonomous and Adaptive Systems **13**(3), 1–24 (2018)

19. Gu, S., Holly, E., Lillicrap, T.P., Levine, S.: Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In: Proc. 2017 IEEE International Conference on Robotics and Automation (ICRA'17). pp. 3389–3396 (2017)

20. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2021)

21. Hasanbeig, M., Abate, A., Kroening, D.: Logically-constrained neural fitted q-iteration. In: AAMAS. pp. 2012–2014. IFAAMAS (2019)

22. Hasanbeig, M., Abate, A., Kroening, D.: Cautious reinforcement learning with logical constraints. In: AAMAS. pp. 483–491. International Foundation for Autonomous Agents and Multiagent Systems (2020)

23. Hunt, N., Fulton, N., Magliacane, S., Hoang, T.N., Das, S., Solar-Lezama, A.: Verifiably safe exploration for end-to-end reinforcement learning. In: Proc. 24th International Conference on Hybrid Systems: Computation and Control (HSCC'21) (2021)

24. Jaeger, M., Jensen, P.G., Larsen, K.G., Legay, A., Sedwards, S., Taankvist, J.H.: Teaching Stratego to play ball: Optimal synthesis for continuous space MDPs. In: ATVA. pp. 81–97. Springer (2019)

25. Jansen, N., Könighofer, B., Junges, S., Serban, A., Bloem, R.: Safe reinforcement learning using probabilistic shields. In: Proc. 31st International Conference on Concurrency Theory (CONCUR'20). vol. 171, pp. 31–316 (2020)

26. Jin, P., Zhang, M., Li, J., Han, L., Wen, X.: Learning on Abstract Domains: A New Approach for Verifiable Guarantee in Reinforcement Learning (jun 2021)

27. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. Formal Methods in System Design **36**(3), 246–280 (2010)

28. Kazak, Y., Barrett, C.W., Katz, G., Schapira, M.: Verifying deep-RL-driven systems. In: Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM'19. pp. 83–89. ACM (2019)

29. Kemeny, J., Snell, J., Knapp, A.: Denumerable Markov Chains. Springer-Verlag, 2nd edn. (1976)

30. Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J., Lam, V., Bewley, A., Shah, A.: Learning to drive in a day. In: ICRA. pp. 8248–8254. IEEE (2019)

31. Könighofer, B., Lorber, F., Jansen, N., Bloem, R.: Shield Synthesis for Reinforcement Learning. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 12476 LNCS, pp. 290–306. Springer, Cham (oct 2020)

32. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Proc. 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, vol. 6806, pp. 585–591. Springer (2011)

33. Lahijania, M., Andersson, S.B., Belta, C.: Formal verification and synthesis for discrete-time stochastic systems. IEEE Transactions on Automatic Control **60**(8), 2031–2045 (2015)

34. Langford, J., Zhang, T.: The epoch-greedy algorithm for contextual multi-armed bandits. Advances in neural information processing systems **20**(1), 96–1 (2007)

35. Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J., Jordan, M., Stoica, I.: RLlib: Abstractions for distributed reinforcement learning. In: Dy, J., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 80, pp. 3053–3062. PMLR (10–15 Jul 2018)

36. Lun, Y.Z., Wheatley, J., D'Innocenzo, A., Abate, A.: Approximate abstractions of markov chains with interval decision processes. In: Proc. 6th IFAC Conference on Analysis and Design of Hybrid Systems (2018)

37. Ma, H., Guan, Y., Li, S.E., Zhang, X., Zheng, S., Chen, J.: Feasible Actor-Critic: Constrained Reinforcement Learning for Ensuring Statewise Safety (2021)

38. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: Balcan, M.F., Weinberger, K.Q. (eds.) Proc. 33rd International Conference on Machine Learning. vol. 48, pp. 1928–1937. PMLR (2016)

39. Osborne, M.J., et al.: An introduction to game theory, vol. 3. Oxford university press New York (2004)

40. Papoudakis, G., Christianos, F., Albrecht, S.V.: Agent modelling under partial observability for deep reinforcement learning. In: Proceedings of the Neural Information Processing Systems (NeurIPS) (2021)

41. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)

42. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: VMCAI. pp. 25–41. Springer (2005)

43. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv:1707.06347 (2017)

44. Smith, R.L.: Efficient Monte Carlo procedures for generating points uniformly distributed over bounded regions. Operations Research **32**(6), 1296–1308 (1984)

45. Srinivasan, K., Eysenbach, B., Ha, S., Tan, J., Finn, C.: Learning to be Safe: Deep RL with a Safety Critic (2020)

46. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating Robustness of Neural Networks with Mixed Integer Programming (2017)

47. Vamplew, P., Dazeley, R., Barker, E., Kelarev, A.V.: Constructing stochastic mixture policies for episodic multiobjective reinforcement learning tasks. In: Proc. Australasian Conference on Artificial Intelligence. LNCS, vol. 5866, pp. 340–349. Springer (2009)

48. Wolff, E., Topcu, U., Murray, R.: Robust control of uncertain Markov decision processes with temporal logic specifications. In: Proc. 51th IEEE Conference on Decision and Control (CDC'12). pp. 3372–3379 (2012)

49. Yu, C., Liu, J., Nemati, S., Yin, G.: Reinforcement learning in healthcare: A survey. ACM Computing Surveys **55**(1), 1–36 (2021)

50. Networkx - network analysis in python. https://networkx.github.io/, accessed: 2020-05-07

51. Pytorch. https://pytorch.org/, accessed: 2020-05-07

52. Zhu, H., Magill, S., Xiong, Z., Jagannathan, S.: An inductive synthesis framework for verifiable reinforcement learning. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 686–701. Association for Computing Machinery (jun 2019)