Probabilistic Guarantees for
Safe Deep Reinforcement Learning

Edoardo Bacci and David Parker

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

Abstract. Deep reinforcement learning has been successfully applied to many control tasks, but the application of such controllers in safety-critical scenarios has been limited due to safety concerns. Rigorous testing of these controllers is challenging, particularly when they operate in probabilistic environments due to, for example, hardware faults or noisy sensors. We propose MOSAIC, an algorithm for measuring the safety of deep reinforcement learning controllers in stochastic settings. Our approach is based on the iterative construction of a formal abstraction of a controller's execution in an environment, and leverages probabilistic model checking of Markov decision processes to produce probabilistic guarantees on safe behaviour over a finite time horizon. It produces bounds on the probability of safe operation of the controller for different initial configurations and identifies regions where correct behaviour can be guaranteed. We implement and evaluate our approach on controllers trained for several benchmark control problems.

1 Introduction

Deep reinforcement learning is the application of deep neural networks to solve reinforcement learning tasks. This technique has been shown to solve many complex control tasks successfully [5,31,28]. However, real-world applications of these methods, especially in safety-critical scenarios such as autonomous driving, is limited because it is difficult to establish guarantees on their safety.

Formal verification is a rigorous approach to checking the correctness of computerised systems. It is particularly appealing for systems that are based on neural networks, because the training process often yields models that are large, complex and opaque. Furthermore, the input space is typically too large to allow exhaustive testing, and there now exist a variety of approaches to construct adversarial attacks, i.e., small and imperceptible perturbations to the inputs of the neural network that cause it to produce erroneous outputs.

In recent years, there has been growing interest in verification techniques for neural networks [21,18,15], with a particular focus on the domain of image classification. These aim to prove the absence of particular classes of adversarial attack, typically those that are "close" to inputs for which the correct output is known. Methods proposed include mapping the verification to an SMT (satisfiability modulo theories) problem and the use of abstract interpretation.

There are also various approaches to tackling safety in reinforcement learning. For example, safe reinforcement learning [14] factors in safety objectives into the learning process. Using formal specifications of the objectives has also been proposed, such as maximising the probability of satisfying a temporal logic objective [6,13,17] or restricting learning to a set of verified policies [19]. More recently, formal verification of deep reinforcement learning systems has been considered [22], by leveraging existing neural network verification methods.

A further challenge for verifying the safe operation of controllers synthesised using deep reinforcement learning is the fact they are often developed to function in uncertain or unpredictable environments. This necessitates the use of stochastic models to train, and to reason about, the controllers. One source of probabilistic behaviour is dynamically changing environments and/or unreliable or noisy sensing. Another source, and the one we focus on here, is the occurrence of faults, e.g., in the hardware for actuators in the controller.

In this paper, we propose novel techniques to establish *probabilistic* guarantees on the safe behaviour of deep reinforcement learning systems which can be subject to faulty behaviour at runtime. Our approach, which we call MOSAIC (MOdel SAfe Intelligent Control) uses a combination of abstract interpretation and probabilistic verification to synthesise the guarantees.

Formally, we model the runtime execution of a deep reinforcement learning based controller as a continuous-space discrete-time Markov processes (DTMP). This is built from: (i) the neural network specifying the controller; (ii) a controller fault model characterising the probability with which faults occur when attempting to execute particular control actions; and (iii) a deterministic, continuous-space model of the physical environment, which we assume to be known.

We concern ourselves with finite-horizon safety specifications and consider the probability with which a failure state is reached within a specified number of time steps. More precisely, our main aim is to identify "safe" regions of the possible initial configurations of the controller, for which this failure probability is guaranteed to be below some specified threshold.

One key challenge to overcome, due to the continuous-space model, is that the number of initial configurations is infinite. We construct a finite-state abstraction as a Markov decision process (MDP), comprising abstract states (based on intervals) that represent regions of the state space of the concrete controller model. We then use standard probabilistic model checking techniques on the MDP abstraction, and show that this yields upper bounds on the step-bounded failure probabilities for different initial regions of the controller model.

A second challenge is that constructing the abstraction requires extraction of the controller policy from its neural network representation. We perform a symbolic analysis of the neural network, for which we design a branch-and-bound algorithm, and an abstraction process that explores the reachable abstract states of the environment. We also iteratively refine the abstraction to yield more accurate bounds on the failure probabilities. We evaluate our approach by applying it to deep reinforcement learning controllers for two benchmark control problems: a cartpole and a pendulum.

Related Work. As discussed above, various verification techniques for neural networks exist, including those based on abstract interpretation. Some use abstractions based on intervals [29,33,1], as we do; others use more sophisticated representations such as polyhedra and zonotopes [15]. Recently, correctness for Bayesian neural networks has been considered, using probabilistic notions of robustness, e.g., [9]. Mostly, these approaches focus on supervised learning, often for image classification, but they have also been built upon for verified deep reinforcement learning [22], where (non-probabilistic) safety and liveness properties are checked. Other, non-neural network based, reinforcement learning has also been verified, e.g., by extracting and analysing decision trees [3].

In the context of probabilistic verification, neural networks have been used to find POMDP policies with guarantees [11,10], but with recurrent neural networks and for discrete, not continuous, state models. Also related are techniques to verify continuous space probabilistic models, e.g., [25,32] which build finite-state abstractions as Markov chains or interval Markov chains. Finally, there is a large body of work on abstraction for probabilistic verification; ours is perhaps closest in spirit to the game-based abstraction approach for MDPs from [20].

2   Preliminaries

We will use $Dist(X)$ to denote the set of discrete probability distributions over the set X, i.e., functions $\mu : X \to [0,1]$ where $\sum_{x \in X} \mu(x) = 1$. The support of μ, denoted $Supp(\mu)$, is defined as $Supp(\mu) = \{x \in X \mid \mu(x) > 0\}$. In some cases, we will use distributions where the set X is uncountable but where the support is finite. We also write $\mathcal{P}(X)$ to denote the powerset of X.

We use two probabilistic models: *discrete-time Markov processes* (DTMPs) to model controllers, and *Markov decision processes* (MDPs) for abstractions.

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 $\sum_{s' \in Supp(\mathbf{P}(s, \cdot))} \mathbf{P}(s, s') = 1$ for all $s \in S$; AP is a set of atomic propositions; and $L : S \to AP$ is a labelling function.*

The process starts in some initial state $s_0 \in S_0$ and then evolves from state to state in discrete time steps. When in state s, the probability of making a transition to state s' is given by $\mathbf{P}(s, s')$. We assume that the process is finite-branching, i.e., the number of possible successors of each state is finite, despite the continuous state space. This simplifies the representation and suffices for the probabilistic behaviour that we model in this paper.

A *path* is an infinite sequence of states $s_0 s_1 s_2 \ldots$ through the model, i.e., such that $\mathbf{P}(s_i, s_{i+1}) > 0$ for all i. We write $Path(s)$ for the set of all paths starting in a state s. In standard fashion [23], we can define a probability space Pr_s over $Path(s)$. Atomic propositions from the set AP will be used to specify properties for verification; we write $s \models b$ for $b \in AP$ if $b \in L(s)$.

Definition 2 (Markov decision process). *A* 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;* $\mathbf{P} : S \times \mathbb{N} \times S \rightarrow [0, 1]$ *is a transition probability function, where* $\sum_{s' \in S} \mathbf{P}(s, j, s') \in \{0, 1\}$ *for all* $s \in S, j \in \mathbb{N}$; AP *is a set of atomic propositions; and* $L : S \rightarrow AP$ *is a labelling function.*

Unlike discrete-time Markov processes above, we assume a finite state space. A transition in a state s of an MDP first requires a choice between (finitely-many) possible probabilistic outcomes in that state. Unusually, we do not use action labels to distinguish these choices, but just integer indices. Primarily, this is to avoid confusion with the use of actions taken by controllers, which do not correspond directly to these choices. The probability of moving to successor state s' when taking choice j in state s is given by $\mathbf{P}(s, j, s')$.

As above, a path is an execution through the model, i.e., an infinite sequence of states and indices $s_0 j_0 s_1 j_1 \dots$ such that $\mathbf{P}(s_i, j_i, s_{i+1}) > 0$ for all i. A *policy* of the MDP selects the choice to take in each state, based on the history of its execution so far. For a policy σ, we have a probability space Pr_s^σ over the set of paths starting in state s. If ψ is an event of interest defined by a measurable set of paths (e.g., those reaching a set of target states), we are usually interested in the minimum or maximum probability of the event over all policies:

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

3   Controller Modelling and Abstraction

In this section, we formalise the problem of modelling and verifying deep reinforcement learning controllers, and then describe the MDP abstraction that underlies our MOSAIC approach to performing the verification.

3.1   Controller Execution Model

We consider controllers acting over continuous state spaces systems with a discrete action space. We assume a set of n real-valued state space variables and denote the state space by $S = \mathbb{R}^n$. There is a finite set $A = \{a_1, \dots, a_m\}$ of m actions that can be taken by the controller. For simplicity, we assume that all actions are available in every state.

To describe the execution of a controller, we require three things: (i) a *controller policy*; (ii) an *environment model*; and (iii) a *controller fault model*. Each is described in more detail below.

Definition 3 (Controller policy). *A* controller policy *is a function* $\pi : S \rightarrow A$, *which selects an action* $\pi(s)$ *for the controller to take in each state* $s \in S$.

We restrict our attention to policies that are memoryless (choosing the same action in each state s) and deterministic (selecting a fixed single action, with

no randomisation). In this work, policies are represented by neural networks, and generated through deep reinforcement learning. However, for the purposes of this section, we treat the policy simply as a function from states to actions.

Definition 4 (Environment model). *An* environment model *is a function* $E : S \times A \to S$ *that describes the state* $E(s, a)$ *of the system after one time step if controller action* a *is (successfully) taken in state* s.

The environment represents the effect that each action executed by a controller has on the system. We assume a deterministic model of the environment; probabilistic behaviour due to failures is introduced separately (see below).

We also extend E to define the change in system state when a *sequence* of zero or more actions are executed, still within a single time step. This will be used below to describe the outcome of controller execution faults. Re-using the same notation, for state $s \in S$ and action sequence $w \in A^*$, we write $E(s, w)$ to denote the outcome of taking actions w in s. This can be defined recursively: for the empty action sequence ϵ, we have $E(s, \epsilon) = s$; and, for a sequence of k actions $a_1 \ldots a_k$, we have $E(s, a_1 \ldots a_k) = E(E(s, a_1 \ldots a_{k-1}), a_k)$.

Definition 5 (Controller fault model). *A* controller fault model *is a function* $f : A \to Dist(A^*)$ *that gives, for each possible controller action, the sequences of actions that may actually result and their probabilities.*

This lets us model a range of controller faults. A simple example is the case of an action a failing to execute with some probability p: we have $f(a)(\epsilon) = p$, $f(a)(a) = 1{-}p$ and $f(a)(w) = 0$ for all other action sequences w. Another example, is a "sticky" action [26] a which executes twice with probability p, i.e., $f(a)(aa) = p$, $f(a)(a) = 1{-}p$ and $f(a)(w) = 0$ for any other w.

Now, given a controller policy π, an environment model E and a controller fault model f, we can formally define the behaviour of the execution of the controller within the environment. We add two further ingredients: a set $S_0 \subseteq S$ of possible *initial states*; and a set $S_{fail} \subseteq S$ of *failure states*, i.e., states of the system where we consider it to have failed. We refer to the tuple $(\pi, E, f, S_0, S_{fail})$ as a *controller execution*. Its *controller execution model* is a (continuous-space, finite-branching) discrete-time Markov process defined as follows.

Definition 6 (Controller execution model). *Given a controller execution* $(\pi, E, f, S_0, S_{fail})$, *the corresponding* controller execution model *describing its runtime behaviour 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* $s, s' \in S$:

$$\mathbf{P}(s, s') = \sum \left\{ f(\pi(s))(w) \mid w \in A^* \ s.t. \ E(s, w) = s' \right\}.$$

For each state s, the action chosen by the controller policy is $\pi(s)$ and the action sequences that may result are given by the support of the controller fault model distribution $f(\pi(s))$. For each action sequence w, the resulting state is $E(s, w)$. In the above, to define $\mathbf{P}(s, s')$ we have combined the probability of all such sequences w that lead to s' since there may be more than one that does so.

Recall the example controller fault models described above. For an action a that fails to be executed with probability p, the above yields $\mathbf{P}(s,s) = p$ and $\mathbf{P}(s, E(s,a)) = 1 - p$. For a "sticky" action a (with probability p of sticking), it yields $\mathbf{P}(s, E(E(s,a),a)) = p$ and $\mathbf{P}(s, E(s,a)) = 1 - p$.

3.2   Controller Verification

Using the model defined above of a controller operating in a given environment, our aim is to verify that it executes safely. More precisely, we are interested in the probability of reaching *failure states* within a particular time horizon. We write $Pr_s(\lozenge^{\leqslant k} fail)$ for the probability of reaching a failure state within k time steps when starting in state s, which can be defined as:

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

Since we work with discrete-time, finite-branching models, we can compute finite-horizon reachability probabilities recursively as follows:

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

For our controller execution models, we are interested in two closely related verification problems. First, for a specified probability threshold p_{safe}, we would like to determine the subset $S_0^{safe} \subseteq S_0$ of "safe" initial states from which the error probability is below the threshold:

$$S_0^{safe} = \{s \in S_0 \mid Pr_s(\lozenge^{\leqslant k} fail) < p_{safe}\}$$

Alternatively, for some set of states S', typically the initial state set S_0, or some subset of it, we wish to know the maximum (worst-case) error probability:

$$p_{S'}^+ = \sup\{Pr_s(\lozenge^{\leqslant k} fail) \mid s \in S'\}$$

This can be seen as a *probabilistic guarantee* over the executions that start in those states. In this paper, we tackle approximate versions of these problems, namely under-approximating S_0^{safe} or over-approximating $p_{S'}^+$.

3.3   Controller Execution Abstraction

A key challenge in tackling the controller verification problem outlined above is the fact that it is over a continuous-state model. In fact, since the model is finite-branching and we target finite-horizon safety properties, for a specific initial state, the k-step probability of a failure could be computed by solving a finite-state Markov chain. However, we verify the controller for a *set* of initial states, giving infinitely many possible probabilistic executions.

Our approach is to construct and solve an *abstraction* of the model of controller execution. The abstraction is a finite-state MDP whose states are *abstract*

states $\hat{s} \subseteq S$, each representing some subset of the states of the original concrete model. We denote the set of all possible abstract states as $\hat{S} \subseteq \mathcal{P}(S)$. In our approach, we use intervals (i.e., the "Box" domain; see Section 4).

In order to construct the abstraction of the controller's execution, we build on an abstraction $\hat{E}$ of the environment $E : S \times A \to S$. This abstraction is a function $\hat{E} : \hat{S} \times A \to \hat{S}$ which soundly over-approximates the (concrete) environment, i.e., it satisfies the following definition.

Definition 7 (Environment abstraction). *For environment model $E : S \times A \to S$ 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 $E(s,a) \in \hat{E}(\hat{s},a)$.*

Using interval arithmetic, we can construct $\hat{E}$ for a wide range of functions E. As for E, the environment abstraction $\hat{E}$ extends naturally to action sequences, where $\hat{E}(\hat{s},w)$ gives the result of taking a sequence w of actions in abstract state $\hat{s}$. It follows from Definition 7 that, for any abstract state $\hat{s} \in \hat{S}$, concrete state $s \in \hat{s}$ and action sequence $w \in A^*$, we have $E(s,w) \in \hat{E}(\hat{s},w)$.

Our abstraction is an MDP whose states are abstract states from the set $\hat{S} \subseteq \mathcal{P}(S)$. This represents an over-approximation of the possible behaviour of the controller, and computing the maximum probabilities of reaching failure states in the MDP will give upper bounds on the actual probabilities in the concrete model. The choices that are available in each abstract state $\hat{s}$ of the MDP are based on a partition of $\hat{s}$ into subsets $\{\hat{s}_1, \ldots, \hat{s}_m\}$. Intuitively, each choice represents the behaviour for states in the different subsets $\hat{s}_j$.

Definition 8 (Controller execution abstraction). *For a controller execution $(\pi, E, f, S_0, S_{fail})$, a set $\hat{S} \subseteq \mathcal{P}(S)$ of abstract states and a corresponding environment abstraction $\hat{E}$, the* controller execution abstraction *is defined as an MDP $(\hat{S}, \hat{S}_0, \hat{\mathbf{P}}, AP, \hat{L})$ 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 $\hat{s}$ that is* consistent *with the controller policy π (i.e., $\pi(s) = \pi(s')$ for any $s, s' \in \hat{s}_j$ for each j) and, for each $j \in \{1, \ldots, m\}$ we have:*

$$\hat{\mathbf{P}}(\hat{s}, j, \hat{s}') = \sum \left\{ f(\pi(\hat{s}_j))(w) \mid w \in A^* \text{ such that } \hat{E}(\hat{s}_j, w) = \hat{s}' \right\}$$

  where $\pi(\hat{s}_j)$ is the action that π chooses for all states $s \in \hat{s}_j$;
- *$AP = \{fail\}$ and $fail \in \hat{L}(\hat{s})$ iff $fail \in L(s)$ for some $s \in \hat{s}$.*

The idea is that each $\hat{s}_j$ within abstract state $\hat{s}$ represents a set of concrete states that have the same behaviour at this level of abstraction. This is modelled by the jth choice from $\hat{s}$, which we construct by finding the controller action $\pi(\hat{s}_j)$ taken in those states, the possible action sequences w that may arise when taking $\pi(\hat{s}_j)$ due to the controller fault model f, and the abstract states $\hat{s}'$ that result when applying w in $\hat{s}_j$ according to the abstract model $\hat{E}$ of the environment.

The above describes the general structure of the abstraction; in practice, it suffices to construct a fragment of at most depth k from the initial states. Once constructed, computing maximum probabilities for the MDP yields upper bounds on the probability of the controller exhibiting a failure. In particular, we have the following result (see [2] for a proof):

Theorem 1. *Given a state $s \in S$ of a controller model DTMP, and an abstract state $\hat{s} \in \hat{S}$ of the corresponding controller abstraction MDP for which $s \in \hat{s}$, we have $Pr_s(\Diamond^{\leqslant k} fail) \leqslant Pr_{\hat{s}}^{\max}(\Diamond^{\leqslant k} fail)$.*

This also provides a way to determine sound approximations for the two verification problems discussed in Section 3.2, namely finding the set S_0^{safe} of states considered "safe" for a particular probability threshold p_{safe}:

$$S_0^{safe} \supseteq \{s \in \hat{s} \mid \hat{s} \in \hat{S}_0 \text{ and } Pr_{\hat{s}}^{\max}(\Diamond^{\leqslant k} fail) < p_{safe}\}$$

and the worst-case probability $p_{S'}^{+}$ for a set of states S':

$$p_{S'}^{+} \leqslant \max\{Pr_{\hat{s}}^{\max}(\Diamond^{\leqslant k} fail) \mid \hat{s} \in \hat{S} \text{ such that } \hat{s} \cap S' \neq \varnothing\}$$

4   Policy Extraction and Abstraction Generation

Building upon the ideas in the previous section, we now describe the key parts of the MOSAIC algorithm to implement this. We explain the abstract domain used, how to extract a controller policy over abstract states from a neural network representation, and then how to build this into a controller abstraction. We also discuss data structures for efficient manipulation of abstract states.

Abstract domain. The abstraction described in Section 3.3 assumes an arbitrary set of abstract states $\hat{S} \subseteq \mathcal{P}(S)$. In practice, our approach assumes $S \subseteq \mathbb{R}^n$ and uses the "Box" abstract domain, where abstract states are conjunctions of intervals (or hyperrectangles), i.e., abstract states are of the form $[l_1, u_1] \times \cdots \times [l_n, u_n]$, where $l_j, u_i \in \mathbb{R}$ are lower and upper bounds for $1 \leqslant i \leqslant n$.

4.1   Neural Network Policy Extraction

Controller policies are functions $\pi : S \to A$, represented as neural networks. To construct an abstraction (see Definition 8), we need to divide abstract states into subregions which are *consistent* with π, i.e., those where $\pi(s)$ is the same for each state s in the subregion. Our overall approach is as follows. For each action a, we first modify the neural network, adding an *action layer* to help indicate the states (network inputs) where a is chosen. Then, we adapt a branch-and-bound style optimisation algorithm to identify these states, which builds upon methods to approximate neural network outputs by propagating intervals through it.

Branch and bound. Branch and bound (BaB) is an optimisation algorithm which aims to minimise (or maximise) a given objective function. It works iteratively, starting from the full domain of possible inputs. BaB estimates a maximum and minimum value for the domain using estimator functions, which are
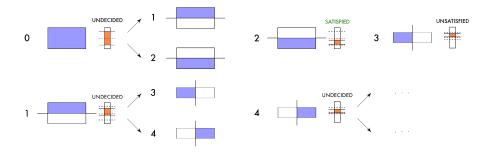
Fig. 1: Illustrating branch-and-bound to identify actions. Each box represents an abstract state and the bar on the right represents upper and lower bounds on the output of the network. 0) The upper and lower bounds of the domain do not give a definite answer, the domain is split into two subregions; 1) The boundaries are tighter than in the previous iteration but the subregion is still undecided; 2) The upper bound is < 0, the property "action taken is a" is always true in this subregion; 3) The lower bound is > 0, the property "action taken is a" is always false in this subregion; 4) The interval between upper and lower bound still contains 0, the action taken in this interval is still unknown so we continue to branch.

quick to compute and approximate the real objective function by providing an *upper bound* (UB) and a *lower bound* (LB) between which the real function lies. The chosen bounding functions must be admissible, meaning we can guarantee that the real function will always lie within those boundaries.

At each iteration of BaB, the domain is split (or "branched") into multiple parts. In the absence of any additional assumptions about the objective function, the domain is split halfway across the largest dimension. For each part, the upper and lower bounds are calculated and regions whose lower bounds are higher than the current global minimum upper bound (the minimum amongst all regions' upper bounds) are discarded because, thanks to the admissibility property of the approximate functions, they cannot ever have a value lower than the global minimum upper bound.

The algorithm proceeds by alternating the branching phase and the bounding phase until the two boundaries converge or the difference between the bounds is less than an acceptable error value. After that, the current region is returned as a solution to the optimisation problem, and the algorithm terminates.

Finding consistent regions. In order to frame the problem of identifying areas of the domain that choose an action a as an optimisation problem, we construct an additional layer that we call an "action layer", and append it on top of the neural network architecture. This is built in such a way that the output is strictly negative if the output is a, and strictly positive value if not. We adopt the construction from [8], which uses a layer to encode a correctness property to be verified on the output of the network.

Algorithm 1: Finding subregions of abstract state $\hat{s}$ for action a

---

1 function find_action_subregions(net, a, $\hat{s}$):
2 | $queue = \{\hat{s}\}$, $sat = \{\ \}$, $unsat = \{\ \}$
3 | $mod_net = $ add_action_layer (net, a)
4 | **while** $queue \neq \varnothing$ **do**
5 | | $curr\_domain = queue.\text{pop}()$
6 | | $UB = $ compute_UB (mod_net, $curr\_domain$)
7 | | $LB = $ compute_LB (mod_net, $curr\_domain$)
8 | | **if** $UB < 0$ **then**
9 | | | $sat.\text{append}(curr\_domain)$
10 | | **else if** $LB > 0$ **then**
11 | | | $unsat.\text{append}(curr\_domain)$
12 | | **else**
13 | | | dom_1, $dom_2 = $ split ($curr\_domain$)
14 | | | $queue.\text{append}(dom_1)$
15 | | | $queue.\text{append}(dom_2)$
16 | | **return** sat,$unsat$

The techniques of [8] also adapt branch-and-bound algorithms, using optimisation to check if a correctness property is true. But our goal is different: identifying areas within abstract states where action a is chosen, so we need a different approach. Rather than minimising the modified output of the neural network, we continue splitting domains until we find areas that consistently either do or do not choose action a or we reach a given precision. We do not keep track of the global upper or lower bound since we only need to consider the local ones to determine which actions are taken in each subregion. In the modified branch-and-bound algorithm, after calculating upper and lower bounds for an interval, we have 3 cases:

– $UB > LB > 0$: the controller will never choose action a for the interval;
– $0 > UB > LB$: the controller will always choose action a;
– $UB > 0 > LB$: the outcome of the network is still undecided, so we split the interval and repeat for each sub-interval.

At the end of the computation, we will have a list of intervals which satisfy the property "the controller always take action a" and intervals which always violate it. From these two lists we can summarise the behaviour of the controller within the current region of the state space.

Algorithm 1 shows pseudocode for the overall procedure of splitting an abstract state $\hat{s}$ into a set of subregions where an action a is always taken, and a set where it is not. Figure 1 illustrates the algorithm executing for a 2-dimensional input domain. The blue subregions are the ones currently being considered; the orange bar indicates the range between computed lower and upper bounds for the output of the network, and the red dashed line denotes the zero line.

Approximating neural network output. The branch-and-bound algorithm requires computation of upper and lower bounds on the neural network's output

for a specific domain (`compute_UB` and `compute_LB` in Algorithm 1). To approximate the output of the neural network, we use the *Planet* approach from [12]. The problem of approximating the output of the neural network lies in determining the output of the non-linear layers, which in this case are composed of ReLU units. ReLU units can be seen as having 2 phases: one where the output is a constant value if the input is less than 0 and the other where the unit acts as the identity function. The algorithm tries to infer the phase of the ReLU function (whether $x < 0$ or $x \geqslant 0$) by constraining the range of values from the input of the previous layers. In the case of the algorithm not being able to determine the phase of the activation function, some linear over-approximation boundaries are used to constrain the output of each ReLU within the section. The constraints used are $y > 0$, $y > x$ and $y \leqslant (u \cdot (x-l))/(u-l)$ where u and l are the upper and lower bounds inferred from the boundaries of the input domain by considering the maximum and minimum values of each input variable.

4.2   Building the Abstraction

Section 3.3 describes our approach to defining an abstract model of controller execution, as an MDP, and Definition 8 explains the structure required of this MDP such that it can be solved to produce probabilistic guarantees, i.e., upper bounds on the probability of a failure occurring within some time horizon k. Here, we provide more details on the construction of the abstraction.

Algorithm 2 shows pseudo code for the overall procedure. We start from the initial abstract states $\hat{S}_0$, which are the initial states of the MDP, and then repeatedly explore the "frontier" states, whose transitions have yet to be constructed, stopping exploration when either depth k (the required time horizon) or an abstract state containing a failure state is reached. For each abstract state $\hat{s}$ to be explored, we use the techniques from the previous section to split $\hat{s}$ into subregions of states for which the controller policy selects the same action.

Determining successor abstract states in the MDP uses the environment abstraction $\hat{E}$ (see Definition 7). Since we use the "Box" abstract domain, this means using interval arithmetic, i.e., computing the successors of the corner points enclosing the intervals while the remaining points contained within them are guaranteed to be contained within the enclosing successors. The definitions of our concrete environments are therefore restricted to functions that are extensible to interval arithmetic.

4.3   Refining the Abstraction

Although the MDP constructed as described above yields upper bounds on the finite-horizon probability of failure, we can improve the results by *refining* the abstraction, i.e., further splitting some of the abstract states. The refinement step aims to improve the precision of states which are considered unsafe (assuming some specified probability threshold p_{safe}), by reducing the upper bound closer to the real probability of encountering a failure state.

Algorithm 2: Build MDP

---

1 function build_mdp(net, $\hat{S}_0$)**:**

2 $\hat{S}_{frontier} = \hat{S}_0$, $t = 0$

3   **while** $t < k$ **do**

4   **foreach** $\hat{s} \in \hat{S}_{frontier}$ **do**

5   **foreach** $a \in A$ **do**

6 $\hat{S}_a, \hat{S}_{\overline{a}} =$ find_action_subregions $(net, a, \hat{s})$

7   **foreach** $\hat{s}_j \in \hat{S}_a$ *and* $p_i{:}w_i$ *in* $f(a)$ **do**

8 $\hat{s}' = \hat{E}(\hat{s}_j, w_i)$

9   store $(\hat{s}, p_i, \hat{s}')$ in MDP

10   add $\hat{s}'$ to $\hat{S}_{frontier}$ unless $\hat{s}' \cap \mathit{fail} \neq \varnothing$

11 $t = t + 1$

Regions of initial abstract states that are considered unsafe are split into smaller subregions and we then recreate the branches of the MDP abstraction from these new subregions in the same way as described in Algorithm 2. This portion of the MDP is then resolved, to produce a more accurate prediction of their upper bound probability of encountering a failure state, potentially discovering new safe subregions in the initial abstract state. The refinement process is executed until either there are no more unsafe regions in the initial state or the maximum size of the intervals are less than a specified precision ϵ.

4.4 Storing and Manipulating Abstract States

Very often abstract states have a topological relationship with other abstract states encountered previously. One abstract state could completely encapsulate or overlap with another, but simply comparing all the possible pairs of states would be infeasible. For this reason we need a data structure capable of reducing the number of comparisons to just the directly neighbouring states. A tree-like structure is the most appropriate and significant progress has been made on tree structures capable of holding intervals. However, most of them do not scale well for n-dimensional intervals with $n > 3$.

R-tree [16] is a data-structure that is able to deal with n-dimensional intervals, used to handle GIS coordinates in the context of map loading where only a specific area needs to be loaded at a time. This data structure allows us to perform "window queries" which involve searching for n-dimensional intervals that intersect with the interval we are querying in $\mathrm{O}(\log_n(m))$ time, where m is the number of intervals stored. R-tree organises intervals and coordinates in nested "subdirectories" so that only areas relevant to the queried area are considered when computing an answer.

Here, we use an improved version of R-tree called R*-tree [4] which reduces the overlapping between subdirectories at the cost of higher computational cost of $\mathrm{O}(n\log(m))$. This modification reduces the number of iterations required during the queries effectively speeding up the calculation of the results. When an

abstract domain is queried for the actions the controller would choose, only the areas which were not previously visited get computed.

5   Experimental Results

We have implemented our MOSAIC algorithm, described in Sections 3 and 4, and evaluated it on deep reinforcement learning controllers trained on two different benchmark environments from OpenAI Gym [7], a pendulum and a cartpole, modified to include controller faults. For space reasons, we consider only "sticky" actions [26]: each action is erroneously executed twice with probability $p = 0.2$.

Implementation. Our implementation uses a combination of Python and Java. The neural network architecture is handled through the Pytorch library [38], interval arithmetic with `pyinterval` [37] and graph analysis with `networkX` [35]. Abstract domain operations are performed with `Rtree` [39], building on the library `libspatialindex` [34]. Constructing and solving MDPs is done using PRISM [24], through its Java API, built into a Python wrapper using `py4j` [36].

5.1   Benchmarks and Policy Learning

Pendulum. The pendulum environment consists of a pole pivoting around a point at one of its ends. The controller can apply a rotational force to the left or to the right with the aim of balancing the pole in its upright position. The pole is underactuated which means that the controller can only recover to its upright position when the pole is within a certain angle. For this reason, if the pole goes beyond a threshold from which it cannot recover, the episode terminates and the controller is given a large negative reward. Each state is composed of 2 variables: the angular position and velocity of the pole.

Cartpole. The cartpole environment features a pole being balanced on top of a cart that can either move left or right. The cartpole can only move within fixed bounds and the pole on top of it cannot recover its upright state after its angle exceeds a given threshold. In this problem the size of each state is 4 variables: the position of the cart on the x-axis, the speed of the cart, the angle of the pole and the angular velocity of the pole.

Policy construction. We train our own controller policies for the benchmarks, in order to take into account the controller failures added. For the policy neural networks, we use 3 fully connected layers of size 64, followed by an output layer whose size equals the number of controller actions in the benchmark. The training is performed by using the Deep Q-network algorithm [27] with prioritised experience replay [30], which tries to predict the action value in each state and choosing the most valuable one. For both environments, we train the controller for 6000 episodes, limiting the maximum number of timesteps for each episode to 1000. We linearly decay the epsilon in the first 20% of the total episodes up to a minimum of 0.01 which we keep constant for the rest of the training. The remaining hyperparameters remain the same as suggested in [27] and [30].
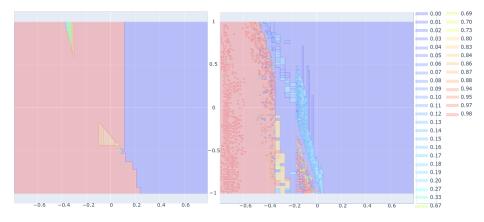
Fig. 2: Heatmaps of failure probability upper bounds for subregions of initial states for the pendulum benchmark (x/y-axis: pole angle/angular velocity). Left: the initial abstraction; Right: the abstraction after 50 refinement steps.

5.2   Results

We have run the MOSAIC algorithm on the benchmark controller policies described above. We build and solve the MDP abstraction to determine upper bounds on failure probabilities for different parts of the state space. Figure 2 (left) shows a heatmap of the probabilities for various subregions of the initial states of the pendulum benchmark, within a time horizon of 7 steps. Figure 2 (right) shows the heatmap for a more precise abstraction, obtained after 50 steps of refinement. We do not fix a specific probability threshold p_{safe} here, but the right-hand part (in blue) has upper bound zero, so is "safe" for any $p_{safe} > 0$. The refined abstraction discovers new areas which are safe due to improved (i.e., lower) upper bounds in many regions.

Results for the cartpole example are harder to visualise since the state space has 4 dimensions. Figure 4 shows a scatterplot of failure probability bounds within 7 time steps for the subregions of the initial state space; the intervals have been projected to two dimensions using principal component analysis, the size of the bubble representing the volume occupied by the interval. We also plot, in Figure 3, a histogram showing how the probabilities are distributed across the volume of the subregions of the initial states. For a given value p_{safe} on the x-axis, our analysis yields a probabilistic guarantee of safety for the sum of all volumes shown to the left of this point.

Scalability and efficiency. Lastly, we briefly discuss the scalabilty and efficiency of our prototype implementation of MOSAIC. Our experiments were run on a 4-core 4.2 GHz PC with 64 GB RAM running Ubuntu 18.04. We successfully built and solved abstractions up to time horizons of 7 time-steps on both benchmark environments. For the pendulum problem, the size of the MDP built ranged up to approximately 160,000 states after building the initial abstraction, reaching approximately 225,000 states after 50 steps of refinement. For the cart-
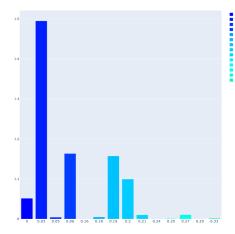
Fig. 3: Cartpole: Histogram plot of the volume occupied by the initial state subregions, grouped by their maximum failure probability.
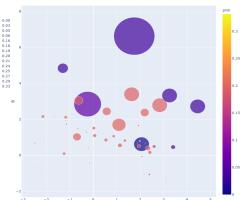

Fig. 4: Cartpole: probability bounds for initial state subregions (projection using principal component analysis; size denotes the volume occupied by the interval). We see that large sections of the state space have maximum probability close to 0.

pole problem, the number of states after 7 time-steps ranged up to approximately 75,000 states. The time required was roughly 50 minutes and 30 minutes for the two benchmarks, respectively.

6   Conclusions

We have presented a novel approach called MOSAIC for verifying deep reinforcement learning systems operating in environments where probabilistic controller faults may occur. We formalised the verification problem as a finite-horizon analysis of a continuous-space discrete-time Markov process and showed how to use a combination of abstract interpretation and probabilistic model checking to compute upper bounds on failure probabilities. We implemented our techniques and successfully applied them to two benchmark control problems.

Future work will include more sophisticated refinement and abstraction approaches, including the use of lower bounds to better measure the precision of abstractions and to guide their improvement using refinement. We also aim to improve scalability to larger time horizons and more complex environments, for example by investigating more efficient abstract domains.

References

1. Anderson, G., Pailoor, S., Dillig, I., Chaudhuri, S.: Optimization and Abstraction: A Synergistic Approach for Analyzing Neural Network Robustness. In: Proc. 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19). p. 731–744 (2019)
2. Bacci, E., Parker, D.: Probabilistic guarantees for safe deep reinforcement learning (2020), arXiv preprint arXiv:2005.07073
3. 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)
4. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: Proc. 1990 ACM SIGMOD International Conference on Management of Data. pp. 322–331 (1990)
5. Bougiouklis, A., Korkofigkas, A., Stamou, G.: Improving Fuel Economy with LSTM Networks and Reinforcement Learning. In: Proc. International Conference on Artificial Neural Networks (ICANN'18). pp. 230–239 (2018)
6. Brázdil, T., Chatterjee, K., Chmelík, M., Forejt, V., Křetínský, J., Kwiatkowska, M., Parker, D., Ujma, M.: Verification of Markov decision processes using learning algorithms. In: Proc. 12th Intl. Symposium on Automated Technology for Verification and Analysis (ATVA'14). LNCS, vol. 8837, pp. 98–114. Springer (2014)
7. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: OpenAI gym. arXiv preprint arXiv:1606.01540 (2016)
8. 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)
9. Cardelli, L., Kwiatkowska, M., Laurenti, L., Paoletti, N., Patane, A., Wicker, M.: Statistical guarantees for the robustness of Bayesian neural networks. In: Proc. International Joint Conference on Artificial Intelligence (IJCAI-19) (2019)
10. Carr, S., Jansen, N., Topcu, U.: Verifiable RNN-based policies for POMDPs under temporal logic constraints. In: Proc. IJCAI'20 (2020), to appear
11. Carr, S., Jansen, N., Wimmer, R., Serban, A.C., Becker, B., Topcu, U.: Counterexample-guided strategy improvement for POMDPs using recurrent neural networks. In: Proc. IJCAI'19. pp. 5532–5539 (2020)
12. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: Proc. 15th International Symposiumon Automated Technology for Verification and Analysis (ATVA'17). LNCS, vol. 10482, pp. 269–286. Springer (2017)
13. Fu, J., Topcu, U.: Probably approximately correct MDP learning and control with temporal logic constraints. In: Proceedings of Robotics: Science and Systems (2014)
14. Garcıa, J., Fernandez, F.: A comprehensive survey on safe reinforcement learning. Journal of Machine Learning Research 16, 1437–1480 (2015)
15. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: Proc. 2018 IEEE Symposium on Security and Privacy (S&P). pp. 3–18. IEEE Computer Society (2018)
16. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data. p. 47–57. SIGMOD '84, ACM (1984)
17. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Omega-regular objectives in model-free reinforcement learning. In: Vojnar, T., Zhang,

L. (eds.) Proc. 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19). LNCS, vol. 11427, pp. 395–412. Springer (2019)

18. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Proc. 29th International Conference on Computer Aided Verification (CAV'17). Springer (2017)

19. Junges, S., Jansen, N., Dehnert, C., Topcu, U., Katoen, J.P.: Safety-constrained reinforcement learning for mdps. In: Chechik, M., Raskin, J.F. (eds.) Proc. 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16). LNCS, vol. 9636, pp. 130–146. Springer (2016)

20. 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)

21. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: Proc. 29th International Conference on Computer Aided Verification (CAV'17). LNCS, vol. 10426, pp. 97–117. Springer (2017)

22. 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)

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

24. 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)

25. Lahijania, M., Andersson, S.B., Belta, C.: Formal verification and synthesis for discrete-time stochastic systems. IEEE Transactions on Automatic Control 60(8), 2031–2046 (2015)

26. Machado, M.C., Bellemare, M.G., Talvitie, E., Veness, J., Hausknecht, M., Bowling, M.: Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. Journal of Artificial Intelligence Research 61, 523–562 (2018)

27. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature 518(7540), 529–533 (feb 2015)

28. Ohn-Bar, E., Trivedi, M.M.: Looking at Humans in the Age of Self-Driving and Highly Automated Vehicles. IEEE Trans. Intelligent Vehicles 1(1), 90–104 (2016)

29. Ruan, W., Huang, X., Kwiatkowska, M.: Reachability analysis of deep neural networks with provable guarantees. In: Proc. 27th International Joint Conference on Artificial Intelligence (IJCAI'18) (2018)

30. Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized experience replay. arXiv preprint arXiv:1511.05952 (2015)

31. Shalev-Shwartz, S., Shammah, S., Shashua, A.: Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving. arXiv preprint arXiv:1610.03295 (2016)

32. Soudjani, S., Gevaerts, C., Abate, A.: FAUST^2: Formal abstractions of uncountable-state stochastic processes. In: Proc. 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15). LNCS, vol. 9035, pp. 272–286. Springer (2015)

33. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proc. 27th USENIX Security Symposium. pp. 1599–1614 (2018)
34. libspatialindex. `https://libspatialindex.org/`, accessed: 2020-05-07
35. Networkx - network analysis in python. `https://networkx.github.io/`, accessed: 2020-05-07
36. Py4j - a bridge between python and java. `https://www.py4j.org/`, accessed: 2020-05-07
37. Pyinterval — interval arithmetic in python. `https://pyinterval.readthedocs.io/en/latest/`, accessed: 2020-05-07
38. Pytorch. `https://pytorch.org/`, accessed: 2020-05-07
39. Rtree: Spatial indexing for python. `https://rtree.readthedocs.io/en/latest/`, accessed: 2020-05-07