FORMAL VERIFICATION OF DEEP REINFORCEMENT LEARNING AGENTS

By

EDOARDO BACCI

A thesis submitted to
the University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
September 2021

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 uncertain environments. In this thesis we develop novel verification techniques to give the user stronger guarantees over the performance of the trained agents that they would be able to obtain by testing, under different degrees and sources of uncertainty.

In particular, we tackle three different sources of uncertainty to the agent and offer different algorithms to provide strong guarantees to the user. The first one is input noise: sensors in the real world always provide imperfect data. The second source of uncertainty comes from the actuators: once an agent decides to take a specific action, faulty actuators and or hardware problems could still prevent the agent from acting upon the decisions given by the controller. The last source of uncertainty is the policy: the set of decisions the controller takes when operating in the environment. Agents may act probabilistically for a number of reasons, such as dealing with adversaries in a competitive environment or addressing partial observability of the environment.

In this thesis, we develop formal models of controllers executing under uncertainty, and propose new verification techniques based on abstract interpretation for their analysis. We cover different horizon lengths, i.e., the number of steps into the future that we analyse, and present methods for both finite-horizon and infinite-horizon verification. We perform both probabilistic and non-probabilistic analysis of the models constructed, depending on the methodology adopted. We implement and evaluate our methods on controllers trained for several benchmark control problems.

ACKNOWLEDGMENTS

I wish to express my gratitude to my supervisor Dave Parker for his endless support, advice and guidance through both the longest and quickest past four years. I am very grateful to my partner, Alessandra Venturoli for supporting me through my darkest moments and for never stopping encouraging me. I know I would not be where I am now without you. I would also like to thank my dear friends Olivia Dyke and Ilana Davis for the good moments together that let me push through difficulties with a lighter heart. In addition, I want to thanks Fatma Faruq who has been an invaluable colleague and friend and for always keeping me focused on the brighter side of things. Finally, I could not have done this without the help and support of my family who always believed in me and ultimately has allowed me to follow my passion.

Nobody does it alone and I am forever grateful to those that helped me.

Fundings. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 834115, FUN2MODEL).

Contents

List of Figures

List of Tables

List of Algorithms

Chapter One

Introduction

Machine learning is a field within Artificial Intelligence and Computer Science where algorithms improve their performance at solving a specific task through the processing of more and more data, effectively learning how to solve the task. One interesting field of application is *Reinforcement Learning* (RL) [189] where the machine learns to solve a task autonomously, without guidance, through trial and error, in a similar manner as infants do when growing up.

The shift of expertise required to solve the task, from the human to the machine, caused by the application of reinforcement learning, presents a great opportunity. Having the same algorithm being able to solve different problems without human intervention and even coming up with novel and unexpected solutions [19], provides the possibility to quickly and almost effortlessly automate an ever increasing number of tasks.

However, with autonomous learning a problem arises: within a complex and hard to understand system created without human intervention *how can we be sure that the behaviour learnt by the machine will not lead to unintended consequences?*

1.1 Motivation

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 [29, 180, 153, 115]. The criterion for training and evaluating RL agents is traditionally their performance, that is, how quickly and efficiently they solve their task. However, for real-world applications of these methods, especially in safety-critical scenarios such as autonomous driving, performance must meet safety: not only is it required that positive outcomes eventually happen, but also that negative ones do not [69, 134].

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 [108, 96, 74], 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 tackle safety in reinforcement learning. For example, safe reinforcement learning [73] 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 [30, 68, 85], restricting learning to a set of verified policies [105] or restricting the policy operation through safety shields [11, 21, 219, 119, 101]. More recently,

formal verification of deep reinforcement learning systems has been considered [113], although not in the context of probabilistic systems, by leveraging existing neural network verification methods.

Alongside the other methods, formal verification in the context of deep reinforcement learning would enable us to provide guarantees to the user over metrics such as quality, safety and generalisation capabilities of the agent during its future operation, helping to bridge the gap between theoretical and practical applications.

1.2 Challenges

Safety verification of neural network controllers presents many challenges. The first key challenge involves how to deal with continuous state spaces and reason about the safety of an infinite number of states. One methodology, called *abstract interpretation*, works by abstracting away some information non pertinent to the problem at hand in exchange for the ability to handle large continuous sections of the state space. This overapproximation often introduces estimation errors that may lead to situations where we cannot be sure whether the system is unsafe but we can be certain when the system is safe. This phenomenon is called *incompleteness* of the algorithm. To reduce the overapproximation errors it becomes essential to correctly tune the parameters to the problem. Complex dynamics of the environment can present a challenge when using abstract interpretation. While linear functions are easier to abstract, often both the environment and the agent present non-linearities in their definition. Abstracting non-linear dynamics requires different strategies to correctly manage the size of the overapproximations introduced.

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, is the occurrence of faults, e.g., in the hardware for actuators in the controller, ensuring that not only the agent chooses the right action but also that they account for a potential malfunctions by preferring a timely and conservative policy. Similarly, probabilistic policies, often used in partially observable environments or against competitive adversaries, need a different approach from deterministic ones: the actions that the agent will take are not clearly established. Given all the aforementioned sources of uncertainty, we need to work with a different type of guarantees, *probabilistic guarantees*, that represent whether or not the chances of bad behaviour falls below a defined threshold.

Our last challenge involves dealing with different time horizons: when trying to formally verify the safety of an agent, we are often constrained by the number of timesteps into the future that we are able to compute. The model of the system constructed to perform verification quickly grows with the number of timesteps we look ahead until it becomes an unfeasible problem. However, when looking for safety, we would ideally look for methods that scale to any time horizon but require a different approach to provide guarantees. This is where we can leverage *safe invariants*, sets of states which are safe that are guaranteed to create a loop with previously visited safe states [28, 67].

1.3 Outline

In this thesis we present three novel approaches to verify the safety of agents under different uncertainty sources: sensors, actuators and policies. These sources of uncertainty, paired with the number of possible actions and continuous state domain contribute to a rapid exponential growth of the number of states involved. To address this, we make use of abstraction to simplify the problem and make it more manageable to reason about. However, depending on the approach, we apply different steps to verify the safety of the controller.

In the first of our methods, following the reinforcement learning loop, we need to compute

successors for each abstract state: this requires restructuring the environment so that, similarly to the previous step, we are able to calculate the successors from each individual abstract state. We generate a Mixed Integer Linear Programming (MILP) model from the combination of the policy abstraction and the dynamics of the environment which we optimise, allowing us to compute abstract successors. Finally, we prove that the action sequence is part of a safe invariant and as a consequence, that the agent is guaranteed to never encounter unsafe states.

In the subsequent method, we construct an abstract model that separates abstract states according to their actions. Our method redefines the neural network so that it is possible to compute action scores for large regions of the state space, shifting from handling single datapoints to abstract regions. We construct a discrete-time Markov process (DTMP, defined in Chapter 5) from the combination between the policy actions and the abstraction of dynamics of the system. The analysis of the DTMP lets us measure probabilities associated with encountering unsafe states. From the computation of the worst case probabilities we can determine whether the agent is probabilistically safe.

In the final approach, we rely on extracting probability intervals from the controller when applied to abstract states. We use the intervals to construct an interval Markov decision process (IMDP) that models the operation of the controller in the environment. From the analysis of the IMDP we can measure the worst-case probability of failure and decide if the agent is probabilistically safe and which areas of the initial state are more prone to encounter unsafe states.

1.4 Thesis Organisation

Below is a brief description of each remaining chapter in the thesis:

- **Chapter 2:** We explain the basics of the concepts used in this thesis which are required to

understand our contribution.

- **Chapter 3:** We survey the recent progress in the literature regarding the work related to and building up to the topic of safety verification of deep reinforcement learning agents.
- **Chapter 4:** We present our first technical chapter where we tackle safety verification under input uncertainty and within infinite horizon.
- **Chapter 5:** We introduce our algorithm for performing probabilistic safety verification under actuator uncertainty within a finite time horizon.
- **Chapter 6:** We describe our approach when dealing with probabilistic safety verification of probabilistic policies within a finite time horizon.

Although the techniques presented in Chapters 4, 5 and 6 could be combined together, each one presents their own challenges and will be analysed independently due to computational complexity constraints.

1.5 Related Publications

The published work contributed to this thesis is listed below:

The work in Chapter 4 has been published as **Edoardo Bacci**, Mirco Giacobbe and David Parker, “Verifying Reinforcement Learning up to Infinity". In: *International Joint Conferences on Artificial Intelligence (IJCAI 2021)*.

The work in Chapter 5 has been published as **Edoardo Bacci** and David Parker, “Probabilistic Guarantees for Safe Deep Reinforcement Learning". In: *Formal Modeling and Analysis of Timed Systems (FORMATS 2020)*.

Chapter Two

Background

In this section we explain some of the foundamental concepts used in the rest of the thesis.

First, we describe deep reinforcement learning. We cover neural networks, some of their variations and the way that they are trained, and also reinforcement learning. We then describe in more depth what deep reinforcement learning (DRL) is, and some of the main algorithms for training DRL agents. Then, we explain the probabilistic models that we use for formal modelling and abstraction: discrete-time Markov processes, Markov decision processes and interval Markov decision processes.

2.1 Deep Reinforcement Learning

In this section we focus on DRL and its 2 components: neural networks and reinforcement learning.

2.1.1 Neural Networks Architecture and training.

Neural networks are an artificial intelligence technology used in machine learning algorithms. They are composed of computational units called neurons which are organised in layers that together form the network. Neural networks can have different architectures depending on the application and the desired output. One of the most common types of architecture is the “feed-forward” network, where each layer is stacked on top of another one, creating a chain that goes from input to the final output, as opposed to “recurrent” networks where the output is also fed back as an additional input to give the network a notion of “time sequence”.

Each neuron has a number of inputs which are aggregated through a weighted sum which is then passed through a non-linear function called an activation function. The corresponding weights for each input represent the variables which are assigned during the training process of the network. In addition to the weights, another term, named “bias", is added to the weighted sum. Whilst we could multiply each input with its corresponding weight individually, the weighted sum operation can be sped up through a matrix multiplication operation. In this way, weights of the entire layer are treated as a single matrix which is multiplied by the input vector and added to the bias vector before being passed to the activation function. Each forward pass can be further parallelised by stacking together multiple inputs in a single batch forming an input matrix. In this way multiple data can be processed in a single pass at the same time with noticeable performance gains.

The activation function used in each neuron is a non-linear function that will modify the weighted sum in the neuron. The non-linearity property of the activation function is strictly needed so that the network is able to approximate any desired function based on the network architecture. The type of function used forms another parameter to be chosen for the network. Recent work tends to prefer the rectified linear unit function (RELU) represented as $y = max(0, x)$. A property of the RELU unit is that it can be regarded as a function with two activation stages, the $x \leqslant 0$ part which is constant and the $x > 0$ part which is linear. Many verification algorithms exploit this dual

behaviour for proving safety properties about the network

We target neural networks with ReLU activation functions, m input neurons, l hidden layers with respectively $h_1, \ldots, h_l$ neurons, and Σ output neurons. The variable vectors $z_0, \ldots, z_{l+1}$ denote the values of the neurons at each layer. The input layer z_0 is assigned from the system observation x. The output of every hidden layer is determined according to the equation

$$z_i = \mathrm{ReLU}(W_i z_{i-1} + b_i), \qquad \text{for } i = 1, \ldots, l, \tag{2.1}$$

and the output layer according to $z_{l+1} = W_{l+1} z_l + b_{l+1}$. Each matrix W_i denotes the weights between any other $(i-1)$-th and i-th layers, and each vector b_i denotes the respective biases. The function ReLU applies $\max\{\cdot, 0\}$ element-wise to its h_i-dimensional argument. The output action is determined by the index of the output neuron whose value is the highest; in other words, the neural network defines the agent

$$\pi(z_0) = \underset{j \in |A|}{\arg\max} \, \langle e_j, z_{l+1} \rangle, \tag{2.2}$$

where e_j is the j-th standard unit vector of $\mathbb{R}^{\Sigma}$, $\langle \cdot, \cdot \rangle$ denotes scalar product and $|A|$ is the number of available actions. Altogether, the neural network acts as a classifier from observations to actions.

The most common way of training the neural network is called *stochastic gradient descent* (SGD). It consists of defining a cost function that describes how far the output of the network is from the true answer and modifying the weights associated with the connections between neurons to reduce the cost to 0. It is called stochastic because at each iteration a random sample is taken from the *training set* (the collection of input-output pairs that the network is supposed to memorise). During the training phase, the network is queried given a randomly sampled batch as input and the error is calculated using the cost function to estimate the difference of the output from the correct answer. The error, then, is back propagated through the network from the output layer towards the

input and each connection is modified depending on how much it contributed to the output. The direction in the multi-dimensional space of connection weights in which to move the network is determined by the gradient of the cost function (the weights are altered in the direction which brings the error closer to 0). The size of the step taken to change the weights is altered by a *learning rate* α. By reducing the size of each step we ensure that inputs with errors in the training set do not skew the global performance of the network. The problem with this approach is that by taking small steps the network will take more iterations to converge to a local minimimum while a big step size might not enable the network to converge at all, making the choice of the learning rate too critical for the success of the algorithm. For this reason, other extensions of SGD have been proposed to try to adjust the learning rate dynamically depending on the current situation. A widespread extension of SGD is called *Adam*.

Adam [117] takes its name from “adaptive moment estimation" and works by calculating the momentum of the gradient for each parameter. By introducing momentum, gradients that had a history of decreasing iterations will be encouraged to progress further along their path even when the gradients at newer iterations go against it, like a ball rolling down a slope that continues to rise for a bit even when it encounters a hill along its path.

2.1.2 Reinforcement learning

Reinforcement learning is an area of machine learning that deals with learning what to do in order to maximise a reward signal that describes how well the controller is performing. The learner is not directly told what actions are right, as in supervised learning, but instead, it must discover which actions yield the most reward by exploring the environment in which it acts. One of the main challenges in this area is the problem of delayed rewards: actions may affect not only the immediate future but they can also have an impact on long term return. These two characteristics: exploration vs exploitation and delayed reward, are the two most important distinguishing features

of reinforcement learning.

The *agent* is the entity guided by the controller, such as an autonomous car or a robot. The agent learns by changing the parameters regulating the controller in such a way as to maximise cumulative future rewards. In the current definition of the system, only the current situation is known to the agent to take decisions. The set of all decisions in the state space is called a *controller policy*. We will assume that the *state space* of the system is $S = \mathbb{R}^n$, using some vector of n real-valued state space variables, and the *actions* available to the controller policy are a finite set $A = \{a_1, \ldots, a_\Sigma\}$. In this thesis we mainly focus on discrete action space; work on continuous action spaces will be treated in future work.

Definition 1 (Environment)**.** An *environment* 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; although we also consider other sources, probabilistic behaviour due to failures is introduced separately (see Chapter 5).

2.1.3 Deep Reinforcement Learning

The use of deep neural networks to guide the decision making process in reinforcement learning is called *deep reinforcement learning*. Below we first define episodes, trajectories and returns, then, we will explain some of the algorithms from RL and DRL used in future chapters.

Definition 2 (Episode)**.** An *episode* is the repeated interaction of the agent with the environment through actions dictated by the controller and the transition to successor states dictated by the environment until a *terminal state* is reached. Some tasks do not have a predefined end and are called *continuing tasks*.

Definition 3 (Trajectory)**.** A *trajectory* is the potentially infinite sequence of alternating states and actions

$$s_0 a_0 s_1 a_1 s_2 ...$$

where each successor state $s_1, s_2, ...$ is determined by the policy π and the environment E.

Definition 4 (Return)**.** The *return* is the sum of all the rewards experienced by the agent interacting with the environment during an episode.

$$G_t = R_t + R_{t+1} + R_{t+2} + ... + R_T$$

where T is the final timestep of the episode. In case of continuing tasks future rewards are weighted with the use of the *discount rate* $\gamma \in [0, 1]$:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$$

In this case, although the return is a sum of an infinite number of terms, the result is finite.

Q-Learning

Temporal Difference algorithms (TD) are a family of algorithms that “bootstrap", which means they perform estimates on the current state based on other estimates (the estimated Q-value at timestep $t + 1$) plus some known information. One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning [204]. In Q-learning each state-action pair gets assigned a “quality" value Q that defines how good it is to take action a amongst all possible actions A in state s. S_t is defined as all possible states at timestep t while A_t

are the actions available in these states.

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

In this case, the bootstrapping starts from random values for the Q-function and then incorporates the new reward R received after taking some action a. The name off-policy means that the algorithm is capable of updating the value of each state-action pair, hence learning, no matter the actions the controller is following. On-policy algorithms, on the other hand, require the controller to follow its policy rigorously in order to improve it. The Q-values ("quality" of the state-action pairs) are stored in a table that contains every possible state configuration that gets queried every time we want to know the values of a state. These values get updated iteratively by the algorithm until eventually, they converge.

SARSA

The corresponding on-policy method to Q-learning is *Sarsa* [189]. Sarsa is another TD algorithm that learns the state-action value of following policy π. The estimation of the state action pairs in this case will no longer come from the reward at the current timestep plus the maximum achievable gain amongst any possible action, but from the reward added to the gain from the action that we will effectively take by following π. Following is the formula for the TD update step:

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

In order to estimate the TD error, which is the difference between the prediction Q at timestep t and the one step expected return, used to update the Q-values, the algorithm will need the states and the actions for the current and the next timesteps and the reward $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ forming the acronym *SARSA* from which the algorithm takes its name. Again, the Q-values will be stored in a

table that will determine the policy of the agent once executed.

Deep Q-Network (DQN)

The main drawback of Q-learning is that by using a table of Q-values we are limited by the memory of our machine. Besides, most of the state action pairs are rarely used because they contain combinations of the state space that are unlikely to be visited.

Deep Q Networks (DQN) try to address this problem by using a function approximator such as a neural network instead of a table of values. In this way, in addition to using considerably less memory, the neural network can be tuned to generalise to unseen states from the values of neighbouring states without the need to explore every state-action pair.

Using DQN [142], a controller was able to outperform humans in a variety of Atari games by learning the policy directly from pixels, without any external inputs.

Policy Gradient (PG)

The main idea behind the *Policy Gradient* [190] algorithm is to increase the probability of picking an action that leads to high returns in the past, and decrease the probability of picking an action that lead to low returns in the past. In the same way that DQN is an extension of Q-learning with the use of neural networks, PG is an extension of the Sarsa method described above. When updating the network weights, the size of the update will be directly proportional to both the probability of chosing the actions in the trajectory and the rewards that were collected along the way. The following is the update rule for the PG algorithm:

$$\theta_{k+1} = \theta_k + \alpha \underset{\tau \sim \pi_\theta}{\mathbf{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right]$$

where $A^{\pi_\theta}(s_t, a_t)$ is the advantage function which is the difference between the expected returns from the chosen action and the average expected return in the state.

Proximal Policy Optimisation (PPO)

A more recent improvement over PG comes from *Proximal Policy Optimisation* [178] which aims to reuse past experience (the trajectories experienced by the agent) as much as possible but also to limit the updates to the neural network weights to a small area surrounding their current configuration. One of the problems of PG is the great volatility in performance caused by the shift in action probabilities as the network changes. By restricting the nework updates and reusing past experiences, the overall growth in perfomance of the agent becomes more stable and reduces training time.

2.2 Markov Models

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 $\mathrm{supp}(\mu)$, is defined as $\mathrm{supp}(\mu) = \{x \in X \mid \mu(x) > 0\}$. 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.

2.2.1 Discrete-time Markov Process (DTMP)

We make particular use of three probabilistic models in this thesis: *discrete-time Markov processes* (DTMPs) for model controllers, and *Markov decision processes* (MDPs) and *interval Markov decision processes* (IMDPs) for abstractions of the underlying MDPs. We will use these to model the interactions of the agent with the environment considering different probabilistic settings and

abstraction levels depending on the focus of each chapter.

Definition 5 (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 $L : S \to AP$ 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 the following chapters.

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 [114], we can define a probability space Pr_s over $Path(s)$.

$$Pr_s^{=} \sum \prod$$

Atomic propositions from the set AP will be used to specify properties for verification; we write $s \models b$ for $b \in AP$ if $b \in L(s)$.

2.2.2 Markov Decision Process (MDP)

A commonly adopted control framework for discrete-time stochastic control problems is the Markov Decision Process (MDP). In MDP the current situation of the world is encoded in “states". A controller that wants to act in the chosen environment has access to the states and a set of available “actions". Each action has some probability of ending in one of the possible outcomes of taking

an action in a given state, becoming the next state. The assumption in this framework is that the outcome will only depend on the current state and the action chosen, as opposed to having the outcome to depend not only on the current state and action but also all the history of past states.

Definition 6 (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; $\mathbf{P} : S \times \mathbb{N} \times S \to [0, 1]$ 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 $L : S \to AP$ 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 $s_0 j_0 s_1 j_1 \ldots$ 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 π, again, we have a probability space Pr_s^π over the set of paths starting in state s, $Path(s)$, by fixing the actions according to the decisions of the policy, as done in the literature [114]. 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:

$$Pr_s^{\min}(\psi) = \inf_\pi Pr_s^\pi(\psi) \quad \text{and} \quad Pr_s^{\max}(\psi) = \sup_\pi Pr_s^\pi(\psi) \tag{2.3}$$

These value can be computed through standard techniques such as value iteration, policy iteration or linear programming[168].

2.2.3 Interval Markov Decision Processes (IMDPs)

When the probability of an event is difficult to measure, it can be abstracted by using interval ranges which allow us to express the uncertainty over our measurements. Interval Markov decision processes generalise MDPs by allowing transitions to be represented by intervals of probabilities.

Definition 7 (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 $\mathbb{I} = \{[a, b] \mid 0 \leqslant a \leqslant b \leqslant 1\}$, assigning either a probability interval or the exact probability of 0 to any transition; AP is a set of atomic propositions; and $L : S \to AP$ is a labelling function.

In addition to a policy π that resolves actions, like for MDPs, we have a so-called *environment policy* τ which selects probabilities for each transition that fall within the specified intervals. For a policy π and environment policy τ, we have a probability space $Pr_s^{\pi,\tau}$ over the set of paths starting in state s. If ψ is an event of interest defined by a measurable set of paths, we can compute, for example, lower and upper bounds on maximum probabilities, over the set of all allowable probability values:

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

This can be computed, for example, through *robust value iteration* [208].

Chapter Three

Literature Review

In this section we review similar and related works in the literature. We start by describing the progress in the context of formal verification of neural networks, mostly related to image classification problems. After that, we discuss some of the more recent methods for checking safety of deep reinforcement learning agents, similar to our approach. Subsequently, we analyse alternative methods such as shielding and safe reinforcement learning that, despite not providing formal guarantees, aim to achieve the same objective of obtaining a safe agent.

3.1 Verification of Neural Networks

In the next section we review the most recent verification algorithms applied to neural networks. Verification of neural networks is a new research field that gained popularity after the discovery of adversarial examples in 2013 [191, 27]. A summary table (Table 3.1) is provided at the end of the section to give the reader a better view of the current state of the research field.

Some algorithms in this section will be described as *exact* and/or *complete*. An algorithm described as “exact" is one that, once terminated, can provide a specific counterexample if the

Algorithm	Method	Complete?	Real networks?	Scales with
DLV	SMT solver	no	no	network
RELUplex	SMT solver	yes	no	network
MIP1	MIP	yes	no	network
MIP2	MIP	yes	no	network
Planet	MIP	yes	no	network
Sherlock	Gradient Descent	no	no	network
Ai2	Abstract Interpretation	no	yes	network
Fast-lin & Fast-lip	Linear approximation	no	yes	network
Branch and Bound	Global Optimisation	yes	yes	input
DeepGO	Global Optimisation	no	yes	input

Table 3.1: Comparison of different neural network verification algorithms. The method columns refers to the main strategy at the core of the algorithm. The complete column describe if the algorithm is always able to determine if the network is safe or not. The Real networks column refers to the ability of the algorithm to handle networks which are large in size (thousands of neurons). The last column outline which is the major factor that contributes to the time complexity of the algorithm

property which is being verified is satisfiable (e.g. an instance of unsafe behaviour is possible). An algorithm which is not exact, instead, is able to determine satisfiability but not to pinpoint a counterexample. In this case an interval is normally returned in which the adversarial examples exist.

On another note, a complete algorithm is an algorithm which guarantees by construction that it will be able to determine if the problem is satisfiable or not. Incomplete algorithms, on the other hand, may not be able to always determine the satisfiability of the property and return neither SAT nor UNSAT as a result.

3.1.1 SMT solvers

Deep Learning Verification (DLV) [95] is an automated verification framework based on Satisfiability Modulo Theories (SMT). The algorithm focuses on single images rather than providing a statistical analysis as in most of the literature for neural networks [64][23][201]. Due to the nature of the

problem of verifying images, the problem can be reduced in size by discretising the input to pixel values rather than using floating point values. The authors introduce the idea of a set of manipulations such as scratches, changes in lighting or rotations that make the network misclassify. The algorithm works by taking a point (image) and delimiting a region around it at a specified layer (a hidden layer can be selected) in which the user wants to verify that there is no change of class. The algorithm then proceeds by projecting, layer by layer, the region delimited by chosen constraints in the transformed hyper-plane represented in each layer and checking if there exists a manipulation amongst the ones allowed that will bring the image outside its delimiting region. Checking every possible manipulation would be infeasible, so there is a heuristic for feature extraction that helps to reduce the area to search. This method enables proving that at a given layer there are no possible manipulations that will cause the network to misclassify and will return an example if any adversarial perturbation is found.

Although the algorithm can guarantee the absence of adversarial examples under the right conditions, the degree of approximations due to discretisation and the number of assumptions needed for the guarantees to hold make DLV more suitable for finding adversarial examples rather than guaranteeing their absence.

RELUplex [108] is a state-of-the-art exact and complete verification tool for neural networks. It is based on the *Simplex* LP Solver combined with the DPLL algorithm (a SMT theory solver) for deterimining satisfiability which is then adapted for use on neural networks with RELU units (RELUplex = RELU with Simplex). One of the main problems of handling verification of deep neural networks is the nonlinearity of the activation function as it prevents the use of tools like linear programming or satisfiability modulo theories. A workaround that builds on DPLL is to consider the two possible activation phases of RELU independently with an operation called “split". This approach, however, brings an exponential number of combinations which is intractable for deep networks. However, the RELUplex algorithm works by first guessing the state of each activation function when trying to find a counterexample to the verified property (e.g. an adversarial example)

and backtracking later if the configuration leads to a contradiction.

The worst case time for the algorithm is exponential, however, in practice. Thanks to heuristics that allow discarding sections of the problem, it has reasonable solving time. A benefit of RELUplex is that, if a variable needs to be backtracked often, that is, its guess of activation state is frequently changing, then the algorithm performs the split of the function and continues solving for both cases at once. In addition to local adversarial robustness (proving that for a particular image there are no adversarial examples), RELUplex is also able to prove *global adversarial robustness*, robustness of the network for every input, by duplicating the given system and demonstrating that for any input value x and a given maximum perturbation δ the change in output will be less than ϵ. This problem, however, is much harder to solve and could only be applied to small networks.

3.1.2 Mixed Integer Programming.

Tjeng, Xiao, and Tedrake [196] propose a verification algorithm for piecewise-linear feed-forward neural networks that uses mixed-integer linear programming (MILP). Integer programming is an optimisation technique in which variables are restricted to integers. We talk about integer linear programming (ILP) when the objective function and the constraints of the problem are linear. When not all decision variables are integers, it is called mixed integer programming (MIP). Integer programming is an NP-complete problem, which means that techniques to reduce the size of the problem are needed to make solving large problems tractable. To this extent, the authors propose a new “presolve" algorithm. Presolve algorithms are designed to make a model smaller and easier to solve. The idea exploits the fact that the predicted label of the classifier is determined by the unit in the final layer with the maximum activation, so proving that a unit never has the maximum activation over all bounded perturbations solves part of the problem reducing the size of the model.

Another part of the problem is the nonlinearity introduced by the RELU activation function.

Since RELU is piecewise linear, it can be considered as a two-state linear function which means "when $x > 0$ then $y = x$ else $y = 0$". To speed up computation, the algorithm computes an upper (u) and lower (l) bound of the RELU output by keeping in consideration the input domain. If the algorithm can prove that $l \geqslant 0$ or $u \leqslant 0$ for some unit for the whole input domain during the presolve phase, then there will be no need to keep track of both states of the RELU when solving the MILP problem and the activation function can be substituted with a simple linear function. Tighter bounds mean faster solving time but require more time to compute them, so there is a tradeoff between maximum presolve complexity during which the model is being built and the solving complexity during which the safety of the network is assessed. Knowledge of the system allows the user to finely tune this trade-off or search for it in an iterative way. Results are compared to the Reluplex [108] algorithm, reaching 2-3 orders of magnitude increase in speed.

Cheng, Nuhrenberg, and Ruess [48] propose another MIP-based verification algorithm that can verify the stability of the neural network but without assessing just the point-wise robustness of it (point-wise robustness is measured by evaluating the robustness of single sample images to produce an estimate of the whole network). The algorithm is able to work with tanh and softmax activation functions in addition to the common RELU which is supported by other algorithms. Softmax cannot be directly encoded into a linear MIP constraint but, instead, it is sufficient to prove that the true class is the one with the strongest signal by using the following equivalence:

$$x_i^{(L)} \geqslant \alpha x_j^{(L)} \iff x_i^{(L-1)} \geqslant \ln(\alpha) + x_j^{(L-1)}$$

where $x_i^{(L)}$ and $x_j^{(L)}$ refer to the input value of the last layer L of the neural network, with $i, j \in 1, ..., d^{(L)}$, $i \neq j$ and $d^{(L)}$ is the size of the layer L. It follows that the true class value $x_{i_1}^{(L)}$ is bigger than any other class $x_{i_2}^{(L)}$ by a factor of α (where α is a parameter chosen by the user that represent the minimum desired offset of the true class from any other class) if the output from the previous node $x_{i_1}^{(L-1)}$ is greater than $\ln(\alpha) + x_{i_2}^{(L-1)}$ which can be proven by the MIP solver. The tanh function cannot be verified as easily as softmax, but it can be approximated with tight boundaries. The

boundaries will propagate further down the network, so it is of vital importance that they are as close as possible.

The resilience of the network is measured through perturbation bounds: in an m-classes classifier and given a constant $\alpha \geqslant 1$, the maximum perturbation Θ_m that does not cause misclassification can be computed. This measure is expressed in the form of an L_1 norm which consists of the absolute sum of all the perturbations ϵ for each pixel. Results, despite the effectiveness of the algorithm, are still not good enough. This approach can handle only small networks consisting of few hundreds of neurons before timing out, massively restricting its applicability in real-world scenarios.

Planet [62] is an exact algorithm for finding adversarial examples that uses a SAT solver in combination with linear programming. Planet aims to find adversarial examples within some given input domain which is defined as a set of constraints (e.g. the pixel values which are within ϵ from the target image). The algorithm tries to infer the phase of the RELU function (whether $x < 0$ or $x \geqslant 0$). 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 $y \leqslant \frac{u \cdot (x-l)}{u-l}$ which allow for the largest achievable output range of the RELU function considering the input boundaries of the layer where u and l are the upper and lower bound inferred from the boundaries of the input domain by considering the max and min values of each input variable.

At its core, Planet uses a customised SAT solver that employs the *elastic filtering* algorithm [52] for finding the minimal infeasible linear constraint set which are areas of the input domain which cannot contain an adversarial example. The reason for finding infeasible regions which are minimal is that if the specifications are too granular, the number of constraints that the solver will have to account for will constitute a bottleneck. Elastic filtering works by weakening constraints through the use of *slack variables*, which are variables added to make the constraints easier to meet.

Then, iteratively, the slack variable with the highest value is set to 0 making the corresponding instance infeasible. The infeasible combination is then remembered in the next iteration, ruling out parts of the domain, with the effect of speeding up the following searches for solutions.

Despite the improvements to search speed, Planet is only able to handle small networks of few hundred nodes in an acceptable time.

3.1.3 Gradient Descent.

Sherlock [59] is a verification algorithm for *range estimation*, that is, given an input domain, guaranteeing that the output will lie within a given range returned by the algorithm. This calculation can be used to guarantee robustness of adversarial examples in the case in which the output range for the true label is always greater than the range of any other labels. Sherlock works by iteratively ruling out local minima which are found through local gradient descent. Once the local minima are found, an MILP feasibility problem is solved to check if there exists a point which will return an output smaller than the current local minima. If such an example is found, a new local search is started to rule out the new local minima. If the MILP fails to find such an example it means that the current point is the global minimum. The algorithm, although much faster than algorithms such as RELUplex, still cannot handle medium to large networks with more than 250 nodes. In addition, Sherlock solves a different problem to other verification algorithms which is the range estimation problem. Although the range estimation problem can *in some cases* provide guarantees about the safety of the network, there are situations in which the ranges of different classes will overlap and nothing can be said about the output of the network.

3.1.4 Abstract interpretation.

Abstract interpretation consists of ways of abstracting the problem (in this case the neural network) such that we can still answer the important questions (whether the network is safe) while disregarding information which is not useful. By abstracting the network, the problem is simplified and made more manageable to solve. However, we relax the type of question it can be answered by allowing the algorithm to provide an incomplete answer, making the algorithm *incomplete*. An intuition of this phenomenon can be given, for example, in the case of approximating dogs as "animals with 4 legs": we can immediately say if an animal is not a dog by counting the number of legs, but we cannot tell for sure whether an animal with 4 legs is a dog without further analysis.

Abstract interpretation has seen a rise in popularity in the context of neural network verification. When exact verification methods become infeasible, abstraction allows us to scale to bigger networks and verify more complex properties. Some algorithms abstract inputs to intervals [171][202][13] and propagate the intervals across the layers until they get the output interval for further computation.

AI^2 [74] is a scalable but incomplete verification algorithm that works by leveraging the concept of proving safety through *abstract interpretation* that uses use a more sophisticated form of representation such as zonoedra and zonotopes.

In AI^2, each layer of the neural network is substituted by an *abstract transformer*, which is a layer that performs the same operations as the real layer but works in the abstract domain rather than the input domain. After each layer is replaced, the input region that we want to verify is projected in the abstract domain. To do so, the input domain is over-approximated through shapes expressible as a set of logical constraints. The numerical abstract domain classes considered in this paper [74] are: Box, Zonotope and Polyhedra. Box is the simplest one and works by simply defining an interval for each dimension; Zonotope uses zonotopes which are center-symmetric convex closed polyhedrons

defined by a set of linear boundaries; Polyhedra are convex closed polyhedra, where a polyhedron is captured by a set of linear constraints. The main difference between the zonotope and the polyhedra is that the zonotope exploits symmetry for reducing the number of constraints needed although at the cost of limiting the type of shapes it can represent. These domains go from the simplest but faster to compute to the more precise but computationally expensive, giving the user the choice between precision and scalability.

The experiments show that AI^2 manages to handle networks which are orders of magnitude larger than those allowed by exact methods (eg. RELUplex) and with less time. The amount of time increases less steeply as the size of the network increases. The number of properties that can be verified can be finely tuned by choosing a more precise abstract domain at the expense of computational time making this algorithm very adaptive depending on the specific task and needs. On the other hand, chosing the right tradeoff requires expertise in order to make an informed choice.

3.1.5 Linear approximation

In [205] the authors devise two methods for computing lower bounds on the minimum adversarial distortion (closest adversarial example) in a fraction of the time taken by exact methods like Reluplex. The two algorithms are called Fast-lin (fast linear approximator) and Fast-lip (fast Lipschitz approximator). The degree of speed up achieved, compared to other verification methods, allows the verification of very large networks in a matter of seconds. The lower bound β_L returned by both algorithms certifies that $\nexists_{x' \in \mathbb{R}^n} C(x') \neq C(x)$ where x and x' are datapoints close to each other such that $\|x - x'\| < \beta_L$ which means there is no change of class C (and no adversarial example) within the given boundary around x.

The two algorithms work in different ways: Fast-lin uses a linear approximation of the RELU to calculate the lower boundary while Fast-lip bounds the Lipschitz constant of the network

to provide maximum rate of change and therefore a lower boundary. While it has been proven that verifying the minimum adversarial distortion is NP-complete [108], Fast-lin & Fast-lip are in P. This improvement in complexity class comes at the price of an increase of inaccuracy as the number of neurons grows. This means that as the number of neurons grows in the network, the estimated lower bound will be further away from the true minimum.

Fast-lin works by approximating the RELU to their linear components if it can prove that the output always lies in one of the two activation stages ($x > 0$ or $x \leqslant 0$) and providing an upper and lower bound in the case of which the state of the RELU cannot be determined. The upper and lower bound are such that

$$\frac{u}{u-l}y \leqslant \delta(y) \leqslant \frac{u}{u-l}(y-l)$$

where u and l are the upper and lower bounds from the previous layer and $\delta(x)$ is the activation function (RELU).

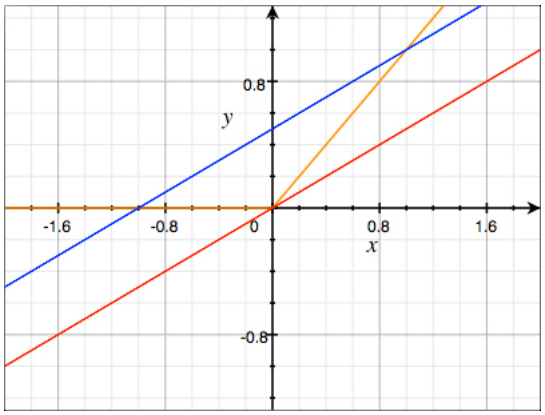


Figure 3.1: The upper (blue) and lower (red) bounds of the RELU function (orange) with input $x \in [-1, 1]$

Fast-lip works by bounding the local Lipschitz constant which is the maximum variation of the output of the network within the given input domain centered around a given image and bounded by the chosen L_p norm. The Lipschitz constant can be intended as the maximum norm of directional derivative, the maximum slope at a point x_0. This value can be calculated by checking

all the possible activation states of the network but the combinatorial explosion of each neuron in the network makes the problem impossible to solve just by brute-force. However, Fast-Lip approaches the problem by computing the worst-case pattern of each neuron, which is much simpler to compute (in case we cannot prove the RELU is always going to be inactive within the input domain, we consider it as if it were always active), and using it as Lipschitz constant. In addition, the computation is further sped up by considering that, in general, the maximum norm of a vector is always upper bounded by the norm of the maximum value of the components of the vector. This means that Fast-Lip does not need to calculate the norm of the whole vector of gradients, but can give an upper bound which consists of the maximum norm found within the vector. The quick calculation of the loose Lipschitz constant allows to predict whether or not is possible, in the worst case scenario, for a network to change decision within the given interval we intend to verify, allowing the user to provide certification that the network is robust.

3.1.6 Global Optimisation

In Bunel et al. [37], the authors reduce the satisfiability problem of finding adversarial examples to a global optimisation problem where the satisfiability of the property being verified is determined by checking the sign of the minimum. As an example, if the output of the network consists of checking that the probability of the k-th class $C(x)_k$ is greater than a value b, it will be sufficient to add a fully connected layer at the end of the network with weights $W = [1]_d$ (a vector of all 1s) and bias $= -b$ to prove the property. In this way, we can leverage an optimisation algorithm to solve this particular problem.

One of the algorithms used by the author is an adaptation of *Branch and Bound*. Branch and bound works by keeping track of the best upper and lower bound for a particular domain (in this case the area in proximity of a given image), iteratively splitting (“branching") the domain in two parts and then recomputing upper and lower bounds (“bound") for each part. If the lower bound of a

subdomain is higher than the minimum upper bound amongst all domains, then it is guaranteed that the subdomain cannot contain the global minimum and can be discarded. The algorithm continues until the difference between the global upper bound and the global lower bounds are within distance ϵ, then the algorithm returns the chosen domain. In this case, it is not necessary to wait for the algorithm to terminate: as soon as the global upper bound becomes $\leqslant 0$ the property is satisfied (an adversarial example has been found) and the algorithm can terminate, otherwise we can wait for termination in order to return a counterexample.

The authors adopt techniques for improving the efficiency of Branch and Bound such as smart branching or better bounding functions. In this way, the time to solve the problem almost halves. An advantage of BnB compared to other exact verification algorithms is that it is less susceptible to changes in the size of the network. This also means that while other algorithms can only handle small network with few layers, Branch and bound can work with much larger networks, getting close to real-world systems. Unfortunately, the downside is that when the size of the input of the network changes, the algorithm becomes very inefficient because it has to branch in many dimensions.

DeepGo [171] is an incomplete verification algorithm that uses techniques from *Global Optimisation* and leverages Lipschitz continuity to provide an output range analysis of the neural network. Since, generally, the last layer of a classifier consists of a softmax activation function, the problem of output range analysis can easily be generalised to *logit* range analysis (logits are the output of the network just before the softmax function). DeepGo differs from other verification algorithms that, rather than leveraging constraints applied to each layer, it is based on a refinement of the reachable ranges of output of the entire network. The algorithm starts by finding an upper bound and lower bound of the output. To find them, DeepGO requires a Lipschitz constant K, given as input, that will guarantee that the maximum rate of change around a point x_0 is less than K. The bigger the K the bigger the search space so the aim is to give the smallest admissible value known.

To find the global minimum, the algorithm uses another function $h(x, y)$ which serves as a lower bound of the original neural network function such that

$$\forall x, y \in [a, b]^n, h(x, y) \leqslant w(x) \;\; \& \;\; h(x, x) = w(x)$$

$h(x, y)$ is a function that, given two points within the interval, returns a value which is lower than the output and returns the value of the output if given the exact same two points. An example of such a function is given by

$$h(x, y) = w(y) - K|x - y|$$

which satisfies the above requirement. Once the estimation of the global minimum is calculated, the algorithm splits the input domain into two intervals and produces their respective lower bounds. The highest of the lower bounds (infimum) is the new estimated global minimum while the lowest output calculated on the points x and y is the upper bound. The algorithm then continues by working on the new set of intervals. As the number of intervals approaches infinity, the lower bound of the function gets closer and closer to the true value. This process of iterative refinement continues until the global minimum and the upper bound are close within an acceptable margin of error after which, the program terminates.

Having a better Lipschitz constant improves convergence so the authors implement a method for recalculating K for each interval depending on the observed values. This step speeds up the whole verification process at the negligible cost of keeping track of a different Lipschitz constant for each interval. In terms of speed, DeepGO, being an incomplete algorithm, results in much faster computation times than exact algorithms such as MIP or RELUplex. However, the authors show that the algorithm is NP-complete in the number of dimensions of the input. This is due to the fact that, by increasing the number of dimensions, the number of ways in which each dimension affects the others causes an exponential number of possibilities.

However, the main difference of DeepGo compared to other incomplete algorithms such

as Fast-lin & Fast-lip is that it scales with the number of changed dimensions rather than with the number of neurons. This property proves useful for neural networks with even millions of neurons but small number of inputs that might be infeasible for other algorithms.

3.1.7 Summary

We collect the above described methods in Table 3.1. Our focus is on verification of deep reinforcement learning, as opposed to just verification of neural network. For this reason we require the ability to reason about both the neural network and the modelled environment together. The necessity of repeating the same computation for multiple timesteps and the exponential state space growth requires a focus on fast computation rather than accuracy. For chapter 5 we leverage the Branch and Bound algorithm from Bunel et al. [37] and linearisations from Weng et al. [205] while for chapter 4 and 6 we adapt a combination of Mixed-integer Linear programming and Abstract Interpretation, similar to the work in Cheng, Nuhrenberg, and Ruess [48] and [74].

3.2 Assuring Safety in Reinforcement Learning

We now discuss other classes of techniques that aim to achieve safety of policies generated by reinforcement learning

3.2.1 Verification of RL

In the literature there has been some prior work where algorithms have been built for verifying deep reinforcement learning [113],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 [22].

In the context of probabilistic verification, neural networks have been used to find POMDP policies with guarantees [44, 43],but with recurrent neural networks and for discrete, not continuous, state models.

Also related are techniques to verify continuous space probabilistic models, e.g., [126, 187] 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 [107].

Zhao et al. [218] models continuous dynamical systems which use ordinary differential equations (ODE) to represent the agent. Such controllers are smaller, which helps for scalability, but not every system can be represented using an ODE. The paper aims to find an inductive invariant in the space of differential equations rather than in the space of solutions by building a property template and verifying if it is possible to violate it. The invariant will be approximated by a neural network and the safety property will be verified against it.

3.2.2 Shielding

Shielding is another safety mechanism applied to reinforcement learning. It is based around the construction of a *shield*, an override mechanism, that prevents the agent from acting upon bad decisions. Although not strictly a verification technique, providing guarantees from the system with this additional layer of safety becomes a trivial task because the agent acts safely by construction.

In Alshiekh et al. [11], the authors synthesize a shield by programmatically determining which actions are forbidden in some situations. The states in the MDP are abstracted using a safety automaton and if an action would lead to an unsafe abstract state, it gets automatically discarded

preventing any unsafe behaviour. The limitations of this approach are that we need to construct the safety automaton and map states to it which is not always possible: sometimes the outcomes are decided many timesteps earlier and the long term consequences of actions could be unknown in advance and need to be learnt by the machine.

Bastani [21] aims to tackle nonlinear dynamics in systems by using a model predictive controller (MPC) that approximates the dynamics of the system at every state and prevents the agent from reaching states that violate the safety property. In this way the agent is constrained to the regions of the safe space from which the MPC knows how to recover, ensuring safety. The success of this method hinges on the ability to construct the recovery policy and ensure its safety a priori.

The strategy adopted in Zhu et al. [219] largely differs from the previous ones: it synthesizes a deterministic program which is an approximation of the learnt neural network behaviour. The synthesized program is then refined with counterexamples that violate safety constraints and once the program has no more unsafe counterexamples it is added on top of the neural network as a shielding mechanism. The simplicity of the synthesized program allows the shielding to generalise to continuous action space. However, the program synthesis is based on counterexamples and on the reduction in complexity from the initial neural network, based on these two aspects, the performance of the agent could be negatively affected or the synthesis process could require an extremely large number of counterexamples.

Könighofer et al. [119] propose shielded learning, a precomputed shield that limits the exploration of the state space during the training phase. The agent will then learn to optimise its performance within the allowed actions hence guaranteeing correctness. After training, the shield is kept as an extra layer of safety but, thanks to its implementation during the training phase, the

algorithm ensures that the performance of the agent is minimally affected by the shield interferences. Again, this methodology requires the construction of an automaton to construct the shield, which is not always possible.

Jansen et al. [101] expands shielding to problems where there is some degree of uncertainty in the constructed MDP. They introduce probabilistic shields that allow the agent to be safe with high probability. The shield is applied during the learning phase, substantially decreasing the amount of time spent training.

3.2.3 Safe Reinforcement Learning

These algorithms focus on training the agent to be safe out of the box, without any external interference like with shielding. They also aim to provide ways to explore the state space without violating the safety constraints opening up the option to train agents in the real world.

Hasanbeig, Abate, and Kroening [87] train a vanilla RL agent by affecting the reward of the Q-function with an LTL automaton. Each state is abstracted using a discretisation technique and labelled according to the LTL automaton. Hence, by maximising the reward function using dynamic programming, the agent learns an optimal and safe policy.

Cheng et al. [51] aim to discover control barrier functions, safe areas of the state space delimited by a function, by deploying the agent in a completely unknown environment and employs a Gaussian Process (GP) to model the dynamics of the system. Since the environment is unknown, the agent adopts a very conservative policy during training that later gets relaxed as the performance

of the agent improves. The paper promise asymptotic improvement towards safety but does not guarantee that ultimately the agent will effectively be safe. The algorithm is better suited to continuous action problems where the dynamics of the system present no discontinuities and can be better approximated by the Gaussian process.

Srinivasan et al. [188] aim to train a safe agent by adding a safety critic to a Soft Actor Critic learning algorithm. In this case the safety specifications are not provided, the algorithm would need to learn how to be safe without the need for the user to define safety. Initially the agent learns the safety critic how does it do this? by sampling the state space and classifying it as either safe or unsafe based on the observation of the environment rewards and episode termination. Later the actor policy is trained while the safety critic monitors that actions will not lead to unsafe states. The work, unfortunately, relies on a number of assumptions that are not always applicable in every problem such as there always being an action that allows the system to go back to safety and many parameters that are problem specific. However, the paper constitutes a solid starting point for further improvements.

Hasanbeig, Abate, and Kroening [86] present another algorithm that operates with no supervision from the user that aims to safely explore the environment while training. The strategy behind the paper is to stick to safe well-known regions called *safe padding* when training and as the confidence of the agent about the dynamics of the environment increases, start to cautiously expand this region. The reward of the agent is shaped using an LTL formula by using a state-action mapping function. The algorithm works in the context of discretised state space with classical RL algorithms; whilst the paper provide a strong contribution to the topic, its application are limited by scalability constraints.

Ma et al. [135] aims to improve existing RL methods with the addition of a feasible actor-critic (FAC) in such a way that for a given set of initial states, the safety of the policy is guaranteed when possible. A notable difference from previous papers is that rather than using gradient descent during the training, the paper uses a Lagrangian-based approach that appear to be more efficient in complex control tasks. The algorithm guarantees that the agent will optimise its policy towards a safe behaviour asymptotically, but as with the other Safe-RL algorithms no hard guarantees can be provided with this method.

Jin et al. [103] is an abstraction-based training approach that trains the agent directly on the abstract states rather than the concrete ones. In this way, the policy network can determine the action to take for large consistent sections of the state space. The algorithm works by effectively discretising the concrete states and assigning them to interval-based abstract states. The verification of the LTL property looks at the discretised MDP generated by the transitions in the abstract space and returns whether the policy is safe. Being a discretisation based algorithm, the size of the abstraction granularity needs to be chosen in such a way that the abstract states do not overapproximate the system too much but also not too precisely in such a way that the number of states increases disproportionately.

Chapter Four

Verifying Deep Reinforcement Learning over Unbounded Time

4.1 Introduction

In this chapter, we present our first approach for verifying deep reinforcement learning systems. Traditionally, the criterion against which RL agents have been trained and evaluated has been their performance, that is, how quickly and efficiently they solve their task. However, for agents that interact with critical environments, performance must meet safety: not only is it required that positive outcomes eventually happen, but also that negative ones do not [69, 134]. Safety is subtle, because a system is truly safe only if it avoids danger regardless of how long it is left running. Determining whether an RL system is safe for *unbounded time* addresses both a formal verification question, providing stronger guarantees of correctness than bounded verification, and a machine learning question, indicating whether the learning algorithm has generalised a strategy beyond the length of the episodes used to train it. Verifying RL requires reasoning about the dynamics of the environments together with the learned agents which, in modern RL, are neural networks. For the first time, we treat the automated (and sound) time-unbounded verification of neural networks

interacting with dynamical systems.

Safety analysis for neural networks has been studied before for bounded settings. One example is classification, whose well-known vulnerability to adversarial attacks has been analysed using gradient descent, mixed-integer linear programming (MILP), and satisfiability modulo theories (SMT) [143, 97, 61, 36, 111]. Search-based algorithms of this kind are inherently bounded, unlike *abstract interpretation* methods. Abstract interpretation computes a representation of the set of reachable states and checks whether it avoids a set of bad states. Methods for the abstract interpretation of neural networks have borrowed from the analysis of numerical programs, and have been applied to adversarial attacks [75, 184], output range analysis [212, 58], and time-bounded verification of RL [199, 16]. Time-unbounded verification is more difficult because it requires that the abstraction is both *safe*, i.e., disjoint from the bad states, and *invariant*, i.e., no other states are reachable from it; none of the available approaches, as is, have been demonstrated to achieve both requirements on RL problems.

We present the first technique for verifying whether a neural network controlling a dynamical system maintains the system within a safe region for unbounded time. For this purpose, we overapproximate the reach set using *template polyhedra*, i.e., polyhedra whose shape is determined by a set of directions, the template [175]. Traditional interval and octagonal abstractions have rigid shapes which often produce abstractions that are too coarse to be safe or too tight to be invariant. By contrast—with an appropriate choice of directions—template polyhedra can be adapted to the verification problem making the abstraction tight only where necessary and thus facilitating the identification of safe invariants [28, 67].

We formulate the problem of computing template polyhedra as an optimization problem. For this purpose, we introduce an MILP encoding for a sound abstraction of neural networks with ReLU activation functions acting over discrete-time systems. We support linear, piecewise linear and non-linear systems defined with polynomial and transcendental functions. For the latter, we

combine MILP with interval arithmetic.

We propose a safety verification workflow where agents trained with any, possibly model-free, RL technique are verified against a model of the environment. Every model is accompanied with user-defined templates which, as we experimentally demonstrate, suffice to verify multiple agents. Upon every successful verification result we thus certify that an agent is safe w.r.t. a model, which determines our problem specific safety specification (discussed below in sections 4.4.1, 4.4.2 and 4.4.3). Ultimately, we provide formal guarantees that are equivalent to (or stronger than) those of agents that are trained or enforced to be safe [10, 50, 87, 129],yet without imposing constraints on the agent or the RL process.

We demonstrate that our method effectively verifies agents trained over three benchmark control problems [100, 199, 33]. We additionally show that an alternative time-unbounded verification approach built upon range analysis fails in all cases.

4.2 Safety Analysis of Reinforcement Learning

4.2.1 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 $A = \{a_1, \ldots, a_\Sigma\}$ of Σ actions that can be taken by the controller. For simplicity, we assume that all actions are available in every state.

Often the agent can only experience a small part of the system called the *observation*. The observation differs from the current state so that it contains less information with lower precision, making the decison process harder.

Definition 8 (Observation function)**.** The *observation function* if a function $O : S \to X$, with $S \in \mathbb{R}^n$ and $X \in \mathbb{R}^m$, and $m \in \mathbb{Z}$, that given the current state in the system, returns the observation available to the agent distorted by some observation noise.

A time-invariant *controlled dynamical system* with discrete actions and over discrete time consists of an n-dimensional vector of real-valued state variables s and an m-dimensional vector of real-valued observable variables x. The system dynamics are determined by a difference equation

$$s_{t+1} = E(s_t, a_t) + c_t, \qquad c_t \in C, \qquad s_0 \in S_0, \tag{4.1}$$

where $s_t \in \mathbb{R}^n$, $a_t \in A$, and c_t respectively denote state, input action, and control disturbance at time t. The set $C \subset \mathbb{R}^n$ is the space of control disturbances, $S_0 \subset \mathbb{R}^n$ is the space of initial conditions, and $E\colon \mathbb{R}^n \times A \to \mathbb{R}^n$ is the update function. An observation function $O\colon \mathbb{R}^n \to \mathbb{R}^m$ and a space of observation disturbances $D \subset \mathbb{R}^m$ determine the observable values at time t from a state:

$$x_t = O(s_t) + d_t, \qquad d_t \in D. \tag{4.2}$$

As defined in Definition 3, a trajectory of the system is an infinite sequence of states and actions in alternation

$$s_0 a_0 s_1 a_1 s_2 \dots \tag{4.3}$$

every state $s_1, s_2, \dots$ is determined by Eq. (4.1); every action $a_1, a_2, \dots$ is determined by the *controller policy* $\pi\colon \mathbb{R}^m \to A$ from the observation at the current step, i.e.,

$$a_t = \pi(x_t). \tag{4.4}$$

For the purpose of training an agent using RL, we augment the system with the probability distributions λ_{X_0}, λ_C, and λ_D for the set of initial observations X_0, and the sets C and D, respectively. We require that $\mathrm{supp}(\lambda_{S_0}) = S_0$, $\mathrm{supp}(\lambda_C) = C$, and $\mathrm{supp}(\lambda_D) = D$, where

$\text{supp}(\lambda) = \{x \in : \lambda(s) > 0\}$ is the support of distribution λ over set S (where $x_t = O(s_t) + d_t$ as described in Equation 4.2). This induces a discrete-time *partially observable Markov decision process* (POMDP) with finite actions and possibly uncountable state space and branching.

Definition 9 (Observation disturbance)**.** *Observation disturbance* λ_D is a distortion from the correct value in the observation received by the agent. We model the magnitude of the noise within some distortion boundary D.

Definition 10 (Control disturbance)**.** *Control disturbance* λ_C is a distortion applied to the controller from the action chosen by the agent. We model the magnitude of the noise within some distortion boundary C.

The decisions taken by a controller are represented by a *policy*.

Definition 11 (Controller policy)**.** A *controller policy* is a function $\pi : S \to Dist(A)$, which, for each action $a \in A$, returns the probability of selecting it.

With the above definition we can explain both *deterministic policies* and *probabilistic policies* by changing the type of probability distribution. Probabilistic policies are going to be sampled according to the probability distribution while deterministic ones just choose the action with the highest score, hence they will have probability of 1 for that action and 0 everywhere else.

For simplicity, we also use the following style to access the probability of action a being chosen

$$\pi(s, a) = \pi(s)(a) \tag{4.5}$$

We restrict our attention to policies that are memoryless (whose output depends only on the value of s and not on the previously visited states) in order to preserve the Markov assumption. In addition, in this chapter we focus on deterministic controller policies to be able to find invariants in the model when performing safety verification.

Figure 4.1: Adaptive cruise control: a good and a bad state.

Finally, a reward function $R\colon \mathbb{R}^m \to \mathbb{R}$ maps observations to reward values. We discuss in Sect. 4.4 how we design rewards functions for obtaining performant and safe agents using RL.

4.2.2 Neural Network Policies

Agents are given in the form of neural networks with a focus on RELU activation functions and deterministic output. We train our agents over the POMDP induced by the distributions over initial and disturbance sets. Then, we verify the safety of the dynamical system controlled by the obtained network. We tackle time-unbounded safety verification; for this purpose, we introduce a technique for constructing coarse yet safe abstractions of the reach sets of these neurally controlled dynamical systems.

4.2.3 Safety Verification

We target the *safety verification* question for controlled dynamical systems. Let $B \subset \mathbb{R}^n$ be a set of bad states. Verifying the safety of a system consists of deciding whether, for every trajectory $s_0 a_1 s_1 a_2 s_2 \ldots$, we have that $s_t \notin B$ for all $t = 0, \ldots, \infty$. Dually, it consists of determinining whether there exists a finite prefix $s_0 a_0 s_1 \ldots a_{k-1} s_k$ such that $s_k \in B$. In the former case we say that the system is safe; in the latter we say that it is unsafe.

Example 1. Adaptive cruise control is a paradigmatic example for the safety of an RL system [57, 199]. In its simplest form, it consists of two vehicles, ego and lead, moving in a straight line. An

agent should control ego so that it stays at some close and safe distance from lead. State variables s_v, s'_v, and s''_v resp. determine position, speed, and acceleration of each vehicle $v = \text{ego}, \text{lead}$. The observation function exposes the vehicles' distance $s_{\text{lead}} - s_{\text{ego}}$ and the speed of ego s'_{ego}; both observables are subject to a disturbance. The lead vehicle proceeds at a constant speed of $28\,\mathrm{m\,s^{-1}}$ and, at every step, the agent can either decelerate (action 1) or accelerate ego by $1\,\mathrm{m\,s^{-2}}$ (action 2). Update and observation functions are formally defined in Sect. 4.4.2. The agent is safe only if the distance is positive along every trajectory (Fig. 4.1a); every other condition indicates that the vehicles have crashed (Fig. 4.1b). The set of bad states is thus defined by the constraint $s_{\text{lead}} \leqslant s_{\text{ego}}$. Trivially, an agent that always decelerates is safe; however, safety must coexist with performance, which rewards the agent for keeping ego close to lead.

4.3 Template-based Polyhedral Abstractions for Neurally Controlled Dynamical Systems

We employ abstract interpretation for constructing a sound overapproximation of the reach set of the system. Specifically, we compute a sequence of abstract sets of states in $\mathbb{R}^n$

$$\hat{S}_{t+1} = \mathsf{post}(\hat{S}_t) \tag{4.6}$$

for increasing $t \geqslant 0$, where post—the *post operator*—ensures that $\hat{S}_{t+1}$ overapproximates the states that are reachable after one step from $\hat{S}_t$. Time-unbounded safety verification succeeds if our procedure finds a finite $k \geqslant 1$ such that the sequence of states up to k contains $\hat{S}_t$.

invariant $\hat{S}_t \subseteq \cup\{\hat{S}_0, \ldots, \hat{S}_{k-1}\}$ and

safe $\cup\{\hat{S}_0, \ldots, \hat{S}_k\} \cap B = \varnothing$.

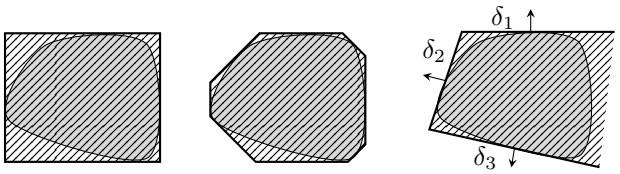


Figure 4.2: Template polyhedra (hatched areas) of a set (gray area).

The procedure computes $\hat{S}_t$ iteratively for increasing t and checks both conditions at each step. If both are satisfied the procedure terminates concluding that the system is safe; if safety is violated it terminates with an inconclusive answer. This procedure may, in the worst case, not terminate. We present a post operator that computes $\hat{S}_t$ in the form of finite unions of template polyhedra; as our experiments show (Sect. 4.4), we repeatedly compute the set of abstract successor states until every successor is contained within the set of previously visited abstract states, which display some practical examples of safe and invariant abstractions.

We call a finite set of directions $\Delta \subset \mathbb{R}^n$ a *template*. A Δ-polyhedron is a polyhedron whose facets are normal to the directions in Δ. The Δ-*polyhedron of* S, where S is a convex set in $\mathbb{R}^n$, is the tightest Δ-polyhedron enclosing S:

$$\cap \{\{s \colon \langle \delta, s \rangle \leqslant \rho_S(\delta)\} \colon \delta \in \Delta\}, \tag{4.7}$$

where $\rho_S(\delta) = \sup\{\langle \delta, s \rangle \colon s \in S\}$ is the support function of S. Special cases of template polyhedra are rectangles (i.e., intervals) and octagons (Fig. 4.2a and b), which are determined by specific templates. In addition, by using fewer, well-chosen directions, template polyhedra let us construct sufficiently tight yet unbounded polyhedral abstractions (Fig. 4.2c).

We compute template polyhedra over a symbolic representation of the post. We split the post computation into a partitioning P_t (a set of sets in $\mathbb{R}^n$) that overapproximates the states that are

reachable after one step from S_t, i.e.,

$$\cup P_t \supseteq \{E(s, \pi(O(s) + d)) + c \colon c \in C, d \in D, s \in S_t\}. \tag{4.8}$$

As we show below, we build the partitioning from the piecewise structure of the system and represent its elements symbolically. Then, for every symbolic representation we construct a template polyhedron by optimising in the directions of Δ. Our post is the union of these template polyhedra:

$$\mathsf{post}(S_t) = \cup\{\Delta\text{-polyhedron of } P' \colon P' \in P_t\}, \tag{4.9}$$

The post produces a union of convex polyhedral overapproximations.

Neurally controlled dynamical systems often have piecewise dynamics because of the large number of problems that support a discrete number of actions (e.g. thermostat [71], braking system [46], collision avoidance in aircrafts [108], videogames [142]). The discrete action space naturally induces a case split in the update function. Also, some systems may have dynamics that switch between two or more behaviours according to guard conditions over the state (see, e.g., Sect. 4.4.1). Formally, each case split is a partial function from a set $F \subset (\mathbb{R}^n \times A \to \mathbb{R}^n)$ s.t. $f = \cup F$ and f is total. Likewise, this case split and the encoding below also applies to the observation function O; for simplicity, we only refer to f.

We compute $\mathsf{post}(S_t)$ using optimisation. We express an encoding for every combination of action $a \in A$, case split $f' \in F$ of the update function, and convex polyhedron S' from the finite union of convex polyhedra S_t; each combination induces an element P' of P_t. For a direction

$\delta \in \Delta$, we solve the following problem:

$$\begin{aligned} \text{maximize} \quad & \langle \delta, p' \rangle \\ \text{subject to} \quad & \pi(x) = a, \\ & p' = f'(s', a) + c,\ s' \in \mathrm{dom}(f'(\cdot, a)) \\ & x = O(s') + d, \\ & c \in C, \quad d \in D, \quad s' \in S', \end{aligned} \tag{4.10}$$

over the variables $c, s', p' \in \mathbb{R}^n$ and $x, d \in \mathbb{R}^m$. The solution provides the value of $\rho_{\mathrm{conv}\, P'}(\delta)$ which, computed over all $\delta \in \Delta$, yields the Δ-polyhedron of P' (see Eq. (4.7)); in turn, the polyhedron yields an element of the post (see Eq. (4.9)).

The optimisation problem consists of a linear objective function and constraints for, respectively, the action chosen by the neural network, update and observation functions, and disturbance and input sets. We assume that the disturbance sets C and D are convex polyhedra, and that the initial set s_0 is a union of convex polyhedra similarly to all other S_t for $t \geqslant 1$. For partial functions $f'(\cdot, a)$, we assume that the domains of definition are given as convex polyhedra. Consequently, the constraints for C, D, S', and $\mathrm{dom}(f'(\cdot, a))$ are expressed with systems of linear inequalities. Our encoding for the constraint over the neural network $\pi(s) = a$ and our overapproximation of non-linear functions introduce integer variables, as we show below. The optimisation problem thus results in an MILP.

The network selects action a when the value of the a-th output neuron is larger than the value of all other output neurons (Eq. (2.2)). Since MILP cannot optimise directly over Eq. (2.2), we reframe the problem such that the constraint for $\pi(s) = a$ is

$$\langle e_j - e_a, z_{l+1} \rangle \leqslant 0 \quad \text{for } j = 1, \ldots, \Sigma, \tag{4.11}$$

where $z_{l+1} \in \mathbb{R}^{\Sigma}$ is a variable for the value of the output layer. For each hidden layer i, we add

to the optimisation problem a real variable $z_i \in \mathbb{R}^{h_i}$ for the values of the neurons in the layer, plus an integer variable $z'_i \in \mathbb{Z}^{h_i}$ for the activation status of the ReLU function. We encode the ReLU function using a big-M encoding [195], a technique for handling the different phases of the activation function. For the hidden layers, we add constraints

$$\begin{aligned} &0 \leqslant z_i - W_i z_{i-1} - b_i \leqslant M z'_i \quad \text{for } i = 1, \ldots, l, \\ &0 \leqslant z_i \leqslant M - M z'_i \qquad\qquad \text{〃} \\ &0 \leqslant z'_i \leqslant 1 \qquad\qquad \text{〃} \end{aligned} \tag{4.12}$$

For output and input layers, we add $z_{l+1} = W_{l+1} z_l$ and $z_0 = y$. Constant $M \in \mathbb{R}$ is an upper bound for the values a neuron can take, which we set to a sufficiently large value so as not to constrain the system's states (such value can either be based on reachable boundaries or found empirically).

Linear update (and observation) functions are encoded directly into the MILP using linear equalities. For non-linear constraints defined with polynomials or transcendental functions, such as with the cart-pole problem (in Sec. 4.4.3), we introduce an overapproximation based on interval arithmetic. Constraint $p' = f'(s', a) + c$ is an n-dimensional system of equalities. We identify the equations within the system that are non-linear and let p'_{N}, f'_{N}, and c_{N} be the corresponding projection for resp. p', and f', and c. Moreover, we let s'_{N} be the largest subset of variables in s' that appear in these non-linear equations. The non-linear components thus form the reduced system

$$p'_{\mathsf{N}} = f'_{\mathsf{N}}(s'_{\mathsf{N}}, a) + c_{\mathsf{N}}. \tag{4.13}$$

We encode the remaining linear components exactly, using linear equalities, whereas we overapproximate Eq. (4.13). First, we construct a bounding box of S'_{N}; note that we ensure beforehand that S'_{N} is bounded with an appropriate template choice (see Sect. 4.4.3). Then, we partition the bounding box into a grid of intervals $[\underline{\xi}_1, \bar{\xi}_1], \ldots, [\underline{\xi}_\kappa, \bar{\xi}_\kappa]$ by splitting it in half. For every element $i = 1, \ldots, \kappa$, we compute using interval arithmetic an output interval $[\underline{\pi}_i, \bar{\pi}_i]$ for the image of $[\underline{\xi}_i, \bar{\xi}_i]$

though $f'_{\mathsf{N}}(\cdot, a)$. For example, for the cart-pole problem (Section 4.4.3), we compute the output boundaries of the dynamics of the environment given intervals for both the θ and θ' variables, with each interval being of size $\bar{\xi}_i - \underline{\xi}_i \leqslant \epsilon$ where ϵ is a parameter chosen by the user. As a result, we obtain a lookup table that associates input intervals to output intervals. We encode this table by adding to the MILP the integer variables $\zeta_1, \ldots \zeta_\kappa \in \mathbb{Z}$, each of which represents an active interval, and the following constraints:

$$\begin{aligned}
&\textstyle\sum_{i=1}^{\kappa} \underline{\xi}_i - \underline{\xi}_i \cdot \zeta_i \leqslant s'_{\mathsf{N}} \leqslant \sum_{i=1}^{\kappa} \bar{\xi}_i - \bar{\xi}_i \cdot \zeta_i \\
&\textstyle\sum_{i=1}^{\kappa} \underline{\pi}_i - \underline{\pi}_i \cdot \zeta_i \leqslant p'_{\mathsf{N}} - c_{\mathsf{N}} \leqslant \sum_{i=1}^{\kappa} \bar{\pi}_i - \bar{\pi}_i \cdot \zeta_i \\
&\textstyle\sum_{i=1}^{\kappa} \zeta_i = \kappa - 1 \\
&0 \leqslant \zeta_i \leqslant 1 \qquad \text{for } i = 1, \ldots, \kappa.
\end{aligned} \tag{4.14}$$

We tune the precision of the overapproximation by fixing a desired granularity for output intervals, a maximal diameter, and iteratively split the input intervals until that is attained.

Example 2. We trained a neural network for adaptive cruise control (Ex. 1) by rewarding the agent for keeping a safety distance of $10\,\mathrm{m}$; the vehicles start from a range of distances between 20 and $40\,\mathrm{m}$. We employed our method for analysing its safety using three different abstraction templates: rectangles, octagons and a custom template designed for this system which considers the most important linear combinations of variables and discards unnecessary information in order to help finding a safe invariant (see Sect. 4.4.2). Rectangles produce an excessively coarse abstraction which hit distance zero: the bad state (Fig. 4.3a). Unlike rectangles, octagons keep track of the vehicles' distance and thus avoid the bad state; however, their abstraction is too tight to identify an invariant, inducing an infinite sequence of polyhedra along the vehicles' position (Fig. 4.3b). Our custom template keeps track of vehicles' distance, while abstracting away absolute position; this yields a safe and invariant abstraction of the reach set (Fig. 4.3c).

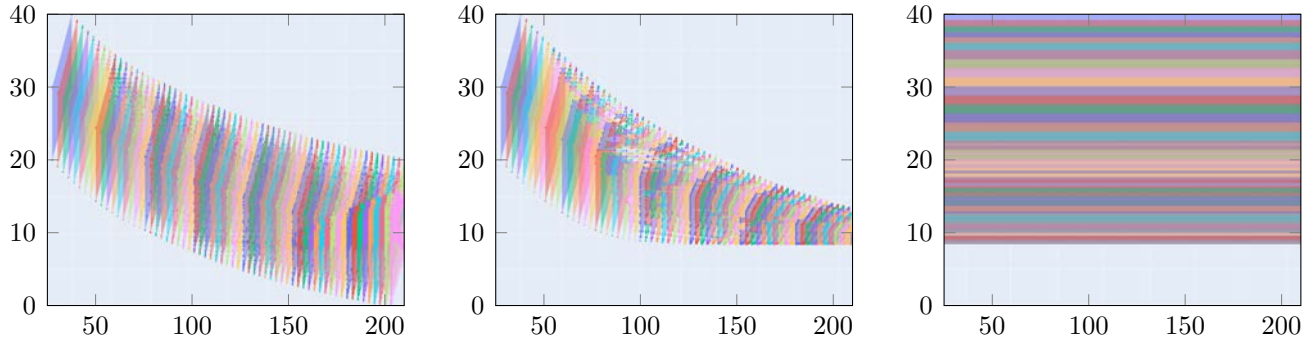


Figure 4.3: Abstract reach sets of a neural network for adaptive cruise control using different templates (Ex. 2). Plots are projected onto vehicles distance (y-axis) and position of lead (x-axis) and constrained within a window, as shown; different colours correspond to different time steps.

4.4 Experimental Evaluation

We evaluate our method over multiple agents for 3 benchmark control problems: a bouncing ball, automated cruise control, and cart-pole. We selected a range of loss functions and hyperparameters and verified, using our method, which setups produce safe behaviour. We trained RL agents using proximal policy optimisation (PPO) [179] with Adam optimiser. We used standard feed forward architectures with 2 hidden layers of size 64 (32 for the bouncing ball), and ReLU activation functions; we used a learning rate of $5e^{-4}$.

We built a prototype[1] and verified the safety of these networks with rectangular and octagonal abstractions and, when necessary, custom templates[2] which we discuss in Sect. 4.4.2 and 4.4.3. In addition, we also compared our method with an alternative approach built upon range analysis (Sect. 4.4.4).

We ran our experiments on a 4-core 4.2GHz with 64GB RAM. Results are shown in Tab. 4.1 and discussed in Sect. 4.4.4.

[1] https://github.com/phate09/SafeRL_Infinity
[2] For readability, we present direction δ by displaying $\langle \delta, x \rangle$.

4.4.1 Bouncing Ball

Environment. The system consists of a ball, whose height from the ground is determined by a variable s and whose vertical velocity is determined by a variable s' [100]. Under normal conditions, position is given by the equation $s_{t+1} = s_t + \tau \cdot s'_t$ and velocity is given by $s'_{t+1} = s'_t - \tau \cdot g$, where g denotes gravitational acceleration and $\tau = 0.1$ indicates our time step. Every time the ball hits the ground, i.e., $s_t \leqslant 0$, the ball bounces back after losing 10% of its energy, i.e., $s'_{t+1} = -0.9 \cdot s'_t - \tau \cdot g$ and $s_{t+1} = 0$. At every timestep, the agent can either hit the ball downward with a piston by adding $-4\,\mathrm{m\,s^{-1}}$ to its velocity, or do nothing. Overall, this results in a piecewise linear system.

Training. The goal is to ensure that the ball keeps bouncing indefinitely, while using the piston as little as possible. We reward the agent with value 1 for each time step that the ball's absolute velocity is above the minimal velocity of $1\,\mathrm{m\,s^{-1}}$. Additionally, we discourage the agent from overactivating the piston by punishing it with reward -1 every time it is activated. We trained 11 agents using different initial seeds and with episodes of at most 1000 timesteps, after which we forcefully terminate. We terminate training either when our agent reaches a mean reward of 900 or after 5M training steps. The parameters have been chosen in order to provide a sensible time cap to the training phase whilst also providing abundant time for the agent to converge to an optimal solution.

Verification. As initial condition, we consider the set of initial ball heights $s_0 \in [7, 10]$ and initial velocities $s'_0 \in [0, 0.1]$. The agent becomes unsafe if the ball stops bouncing ($s_0 = 0$ and $s'_0 = 0$) . We use traditional rectangular and octagonal abstractions, that is, for rectangles we use the directions $s, -s, s', -s'$ and for octagons add the extra directions $s + s', -s + s', s - s', -s - s'$. Notably, all agents have been successfully verified with both rectangles and octagons with no notable difference in performance.

4.4.2 Adaptive Cruise Control

Environment. The problem consists of two vehicles, lead and ego, whose state is determined by variables s_v, s'_v and s''_v, respectively, for position, speed, and acceleration of $v = \text{ego}, \text{lead}$ (see Ex. 1). 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 either of two actions. Its dynamics are given by

$$s_{v,t+1} = s_{v,t} + \tau \cdot s'_{v,t} \qquad \text{for } v = \text{ego}, \text{lead}, \tag{4.15}$$

$$s'_{\text{ego},t+1} = s_{\text{ego},t} + \tau \cdot s''_{\text{ego},t} \qquad s'_{\text{lead},t} = 28, \tag{4.16}$$

$$s''_{\text{ego},t+1} = \begin{cases} -1 & \text{if } a_t = 0, \\ 1 & \text{if } a_t = 1. \end{cases} \tag{4.17}$$

The observation function exposes vehicle distance x_{dis} and the velocity of ego x_{vel} with an additional observation disturbance of radius ϵ, determined by the following equations:

$$x_{\text{dis},t} = s_{\text{lead},t} - s_{\text{ego},t} + d_{\text{dis},t}, d_{\text{dis},t} \in [-\epsilon, +\epsilon] \tag{4.18}$$

$$x_{\text{vel},t} = s'_{\text{ego},t} + d_{\text{vel},t}, d_{\text{vel},t} \in [-\epsilon, +\epsilon]. \tag{4.19}$$

We consider a case with $\epsilon = 0$ and another case with $\epsilon = 0.05$, and use $\tau = 0.1$. Altogether, when an action is given this is a linear system with disturbances.

Training. We train our agents using two 2 different reward functions. A "simple" function only rewards the agent for each timestep it survives without crashing, that is, $R(x_{\text{dis}}, x_{\text{vel}}) = 1$ if $x_{\text{dis}} > 0$; a "complex" function additionally punishes the agent from being away from a predefined distance $x^\star_{\text{dis}}$, specifically, $R(x_{\text{dis}}, x_{\text{vel}}) = 1 - 0.02 \cdot (x_{\text{dis}} - x^\star_{\text{dis}})^2$. We cap each episode at 1000 timesteps.

From the definition of the simple cost function above, we can periodically pre-test the safety of the agent by disabling the exploration and requiring an average score of 1000 before attempting the verification step. For the complex cost function it is more difficult to estimate what a safe score should be, so we empirically determined that before attempting to verify the neural network, the agent needs to reach an average score of at least -20. As an additional stopping condition we terminate the training after 20M training steps. We ran our algorithm over 22 agents trained with different initialisation seeds and two modes of input perturbation ($\epsilon = 0$ and $\epsilon = 0.05$) for up to 300 seconds.

Verification. We consider the initial region enclosed within the constraints $s_{\text{lead},0} \in [40, 50]$, $s_{\text{ego},0} \in [0, 10]$, $s'_{\text{ego},0} = 36$. Using standard rectangular or octagonal abstractions that verification procedure fails by either returning a spurious counterexamples or timing out. To effectively verify this systems, we designed a template with the following directions: s''_{ego}, $-s''_{\text{ego}}$, s'_{lead}, $-s'_{\text{lead}}$, $(s'_{\text{lead}} - s'_{\text{ego}})$, $-(s'_{\text{lead}} - s'_{\text{ego}})$, $(s_{\text{lead}} - s_{\text{ego}})$, $-(s_{\text{lead}} - s_{\text{ego}})$. This allows us to keep track of the distance between the two vehicles and easily spot if the agent encounters an unsafe state, while enabling the identification of an invariant. The agent is considered safe if it can maintain positive distance from the leading car without touching it $(s_{\text{lead}} - s_{\text{ego}}) > 0$. Agents could be proven to be safe in most but not all of the cases within our time constraints (300s), showing a higher degree of difficulty compared to the previous problem. When testing the agents on the perturbed environment, only a few of the agents that were proven safe in the previous experiment retained safety, demonstrating that the problem the agent had to solve is much harder.

4.4.3 Cart-pole

Environment. The cart-pole problem is a very well known control problem in the RL literature; for our experiments, we refer to the OpenAI Gym implementation of CartPole-v1 [33]. The state

variables are angle θ and angular velocity θ' of the pole, together with horizontal position x and velocity x' of the cart. The agent has two actions for pushing the cart to either the left or the right which, together with θ and θ', internally determine horizontal and angular accelerations x'' and θ''. The values of θ'' and x'' are determined according to non-linear equations defined with transcendental functions and whose arguments are the action and variables θ and θ'. The update rule for angle θ, position x, and velocities θ' and x' follow a linear Euler integration rule with timestep τ. All variables x, x', θ, and θ' are observable.

Training. The objective for an agent is to keep the pole upright; we consider the system unsafe whenever $\theta > 12°$, according to the OpenAI Gym termination condition. We train agents using three cost functions. A "simple" version only rewards the agent for surviving; a "complex" version discourages it from having high values of θ and θ', i.e., $R(x, x', \theta, \theta') = 1 - 0.5 * \theta^2 - 0.5 * (\theta')^2 - 0.1 * (x')^2$; a third one limits the complex cost function to only giving positive rewards. For every cost function, we trained two agents using $\tau = 0.02$ and $\tau = 0.001$, thus obtaining 6 agents. We capped each episode to 8000 timesteps. For all cost functions, we terminate training when the mean reward of the last 50 episodes reaches 7950 (i.e., sufficiently close to the maximum of 8000) or after 20M training steps. We use curriculum learning [26], a technique that trains the agent starting from easier variation of the problem and then slowly increasing difficulty, to improve training: when the mean episode return reaches 6500 the initial states get sampled from bigger intervals with $\theta \in [-0.2, 0.2]$ and $\theta' \in [-0.5, 0.5]$.

Verification. We use the starting region of OpenAI Gym, i.e., all variables are initialised from the interval $[-0.05, 0.05]$. However, we remove the constraints imposed on x and let the cart-pole move freely to any position. The safety specification determines that the agent is safe only if it remains within the range $-12° < \theta < 12°$. Rectangles and octagons failed to prove safety on all instances. Thus, we designed a custom template that forms an octagon over θ and θ' only, determined by the

following directions: θ, $-\theta$, θ', $-\theta'$, $\theta + \theta'$, $-\theta + \theta'$, $\theta - \theta'$, $-\theta - \theta'$. The rationale behind this choice is that the position variable x does not contribute to proving or disproving the safety of the system and can be excluded from the template. In addition, by bounding the space of θ and θ', which are the variables affecting the angle of the pole in the non-linear equations of the system we are able to keep track of the range of values that have a real impact on the balance (and then safety) of the cart-pole. This lets us use interval arithmetic for encoding the non-linear equations in our MILP (see Sect. 4.3). We verified our agents against both versions of the environment, with $\tau = 0.02$ and with $\tau = 0.001$ to test the how it would affect safety. The agents that did run on environments at $\tau = 0.001$ during the evaluation found an invariant quicker and in less timesteps.

4.4.4 Results

Table 4.1 reports, for each environment and hyperparameter, the number of solved instances, the average timestep of invariant detection, the number of template polyhedra in the abstract reach set after pruning redundant ones and the runtime of the model construction.

The time required to find whether the agent is safe increases as the number of variables in the problem increases (BB has 2, ACC has 6 and CP has 4) and on the type of abstraction. Templates enable us to find invariants on problems that would not converge otherwise (ACC and CP). Once we introduce a small observation perturbation on ACC such as in adversarial examples, only a small fraction of the agents remain safe negatively impacting safety.

From our results, the cost function used does not strongly correlate with the safety of the agent hence it is omitted in the table. Conversely, shorter timesteps contribute positively to reducing the time required to verify an agent, promoting a higher chance to find a safe invariant in early timesteps.

Additionally, we verified our agents using a naive time-unbounded approach (based on range

Env.	Abs.	Safe	Avg k	Avg poly.	Avg runtime
BB	Rect	11/11	237	477	40s
BB	Oct	11/11	203	411	47s
ACC ($\epsilon = 0$)	Temp	20/22	467	610	171s
ACC ($\epsilon = .05$)	Temp	5/22	226	337	124s
CP ($\tau = .001$)	Temp	4/6	27	18	67s
CP ($\tau = .02$)	Temp	3/6	100	125	174s

Table 4.1: Verification results by environment, i.e, bouncing ball (BB), adaptive cruise control (ACC) and cart-pole (CP), hyperparameters ϵ and τ (where they apply), abstraction, i.e., rectangular, octagonal, or template-based, and number of agents determined to be safe within 300s. For successful outcomes, we report average timestep of fixpoint detection k, number of final template polyhedra, and runtime.

analysis) that constructs, from the network in isolation, ranges of observables for which an action is enabled; then, it uses these ranges as guards for the dynamical system. This alternative approach failed on all instances by producing inconclusive answers (unsafe abstractions) or reaching time-out. Notably, existing verification methods for neural networks are incomparable as they only support time-bounded problems such as robustness to adversarial attacks or finite-horizon safety analysis of RL [199, 75].

4.5 Conclusion

We presented the first method for verifying the safety of RL agents up to infinite time. To this end, our method constructs coarse, yet precise enough, abstractions using template polyhedra. We demonstrated the efficacy of our method over multiple case studies. Our technique yields stronger formal guarantees than previous time-bounded methods, and also indicates which RL setups generalise well beyond the length of their training episodes. Our result poses the basis for future research, both in machine learning and formal verification. Our method can be used to make informed decisions about architectures and hyperparameters, and also to guide an RL procedure that

trains for safety. Also, our method lends itself to extensions towards multi-agent systems, systems over continuous time, continuous actions and automated abstraction refinement.

Chapter Five

Probabilistic Guarantees for Safe Deep Reinforcement Learning

5.1 Introduction

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 chapter, 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. This is in contrast with the work in the previous chapter because, since we want to calculate the probability boundaries of encountering an unsafe state, we can no longer calculate safety invariants and are now forced to use a different approach that requires a time boundary on our analysis.

One key challenge to overcome, due to the continuous-space model, is that the number of initial configurations is infinite due the set of initial states being represented as a section of the state space. 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.

5.2 Controller Execution Model

We now describe our approach to formally modelling and verifying the execution of a controller, and the process of defining an abstraction of this model.

5.2.1 Controller Execution

To describe the execution of a controller, we require three things: (i) a *controller policy* π (from definition 11); (ii) an *environment model* E (from definition 1); and (iii) a *controller fault model*, that we are going to describe more in detail 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 12 (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.

As with the policy, for simplicity we can access the probability of a specific a and w combination:

$$f(a, w) = f(a)(w) \tag{5.1}$$

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 [136] 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 below.

In this chapter, we will restrict our attention to deterministic controller policies (probabilistic policies are considered later in Chapter 6). Here, we will slightly abuse notation and use $\pi(s)$ to denote the single action selected by π in a state s.

Definition 13 (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 $s, s' \in S$:

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

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 $\mathbf{P}(s, E(s, a)) = 1-p$.

5.2.2 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. Since in this chapter we are focusing on a different source of probabilistic behaviour, unlike in the previous one we do not consider the *observation function* and we replace it with an identity function every time we want to access the underlying state. We write $Pr_s(\Diamond^{\leqslant k} fail)$ for the probability of reaching a failure state within k time steps when starting in state s, which can be defined as:

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

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

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

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:

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

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:

$$p^+_{S'} = \sup\{Pr_s(\Diamond^{\leqslant k} fail) \mid s \in S'\}$$

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 $p^+_{S'}$.

5.2.3 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. This is caused by the infinite set of initial states; in comparison, if we only had a finite number of initial states the problem would become much easier and could be solved by enumerating every possible successor up to timestep k.

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 5.3).

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 14 (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 14 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 15 (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 $\hat{E}$, a *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:

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 $Pr_s(\Diamond^{\leqslant k} fail) \leqslant Pr^{\max}_{\hat{s}}(\Diamond^{\leqslant k} fail)$.

A proof of Theorem 1 is provided in the Appendix A.1.

This also provides a way to determine sound approximations for the two verification problems discussed in Section 5.2.2, namely finding the set S_0^{safe} of states considered "safe" for a particular probability threshold p_{safe}:

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

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

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

5.3 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 5.2.3 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 $l_i, u_i \in \mathbb{R}$ are lower and upper bounds for $1 \leqslant i \leqslant n$.

Box intervals will satisfy definition 14 and will easily partition each state a into homogeneous regions $a_j \subseteq a$ such that $\uplus_{j=1}^{m} a_j = a$ compared to other means of abstraction such as the one discussed in [74]. The disjoint property stated in the definition ensures that no point is contained in two different subregions: this is important because if a state was to be included in two different subregions the probabilities of the constructed abstract MDP would not represent a mathematically correct overapproximation of the real probabilities in the concrete one. In addition it helps keeping the number of successors down by ensuring that no region is propagated more than once, limiting the state space explosion phenomenon.

5.3.1 Neural Network Policy Extraction

Controller policies are functions $\pi : S \to A$, represented as neural networks. To construct an abstraction (see Definition 15), 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 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 [37], which uses a layer to encode a correctness property to be verified on the output of the network.

The techniques of [37] 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

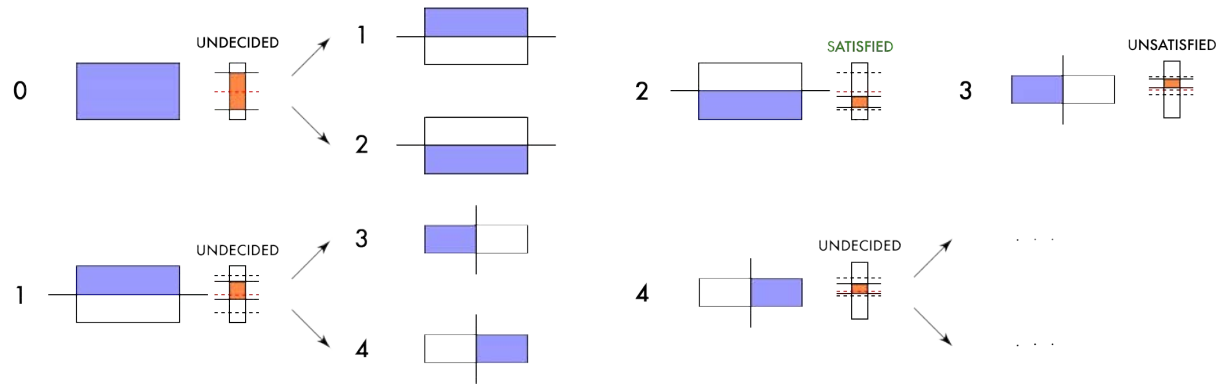


Figure 5.1: Illustrating branch-and-bound to identify actions. Each box represents an abstract state and the bar on the right represents upper and lower bounds on the output of the network. 0) The upper and lower bounds of the domain do not give a definite answer, the domain is split into two subregions; 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.

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

```
function find_action_subregions(net, a, ŝ):
    queue = {ŝ}, sat = { }, unsat = { }
    mod_net = add_action_layer (net, a)
    while queue ≠ ∅ do
        curr_domain = queue.pop()
        UB = compute_UB (mod_net, curr_domain)
        LB = compute_LB (mod_net, curr_domain)
        if UB < 0 then
            sat.append(curr_domain)
        else if LB > 0 then
            unsat.append(curr_domain)
        else
            dom_1, dom_2 = split (curr_domain)
            queue.append(dom_1)
            queue.append(dom_2)
    return sat,unsat
```

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 5.1 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 [62]. 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 $y \leqslant (u \cdot (x - l))/(u - l)$, as presented in *Planet* [62], to represent the range of output obtainable from the ReLU activation function, 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.

5.3.2 Building the Abstraction

Section 5.2.3 describes our approach to defining an abstract model of controller execution, as an MDP, and Definition 15 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 (Paragraph 5.3.1) to split $\hat{s}$ into subregions of states for which the controller policy selects the same action. Each action a is then translated to a distribution over the sequence of action w by the controller fault model f with the corresponding probability p associated. This information is then used to create the various transitions and the corresponding successor abstract states. In algorithm 2 this is expressed as $f(a)$ returning a sequence of pairs of probabilities over sequences of actions denoted $p : w$.

Determining successor abstract states in the MDP uses the environment abstraction $\hat{E}$ (see Definition 14). 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

Algorithm 2: Build MDP

function `build_mdp` (net, $\hat{S}_0$) **:**
  $\hat{S}_{frontier} = \hat{S}_0$, $t = 0$
  while $t < k$ **do**
    foreach $\hat{s} \in \hat{S}_{frontier}$ **do**
      foreach $a \in A$ **do**
        $\hat{S}_a, \hat{S}_{\overline{a}}$ = `find_action_subregions` (net, a, $\hat{s}$)
        foreach $\hat{s}_j \in \hat{S}_a$ *and* $p_i{:}w_i$ *in* $f(a)$ **do**
          $\hat{s}' = \hat{E}(\hat{s}_j, w_i)$
          store $(\hat{s}, p_i, \hat{s}')$ in MDP
          add $\hat{s}'$ to $\hat{S}_{frontier}$ unless $\hat{s}' \cap fail \neq \varnothing$
    $t = t + 1$

definitions of our concrete environments are therefore restricted to functions that are extensible to interval arithmetic.

5.3.3 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.

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.

5.3.4 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 [82] 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 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 [24] which reduces the overlapping between subdirectories at the cost of higher computational cost of 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.

5.4 Experimental Evaluation

We have implemented our MOSAIC algorithm, described in Sections 5.2 and 5.3, and evaluated it on deep reinforcement learning controllers trained with DQN[142] and Adam optimiser on two different benchmark environments from OpenAI Gym [32], an inverted pendulum and a cartpole, modified to include controller faults. For space reasons, we consider only "sticky" actions [136] which provide a reasonable model of old malfunctioning actuators as opposed to random actions: 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 [6], interval arithmetic with `pyinterval` [2] and graph analysis with `networkX` [4]. Abstract domain operations are performed with `Rtree` [1], building on the library `libspatialindex` [5]. Constructing and solving MDPs is done using PRISM [125], through its Java API, built into a Python wrapper using `py4j` [3].

5.4.1 Benchmarks and Policy Learning

Inverted Pendulum. The inverted 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

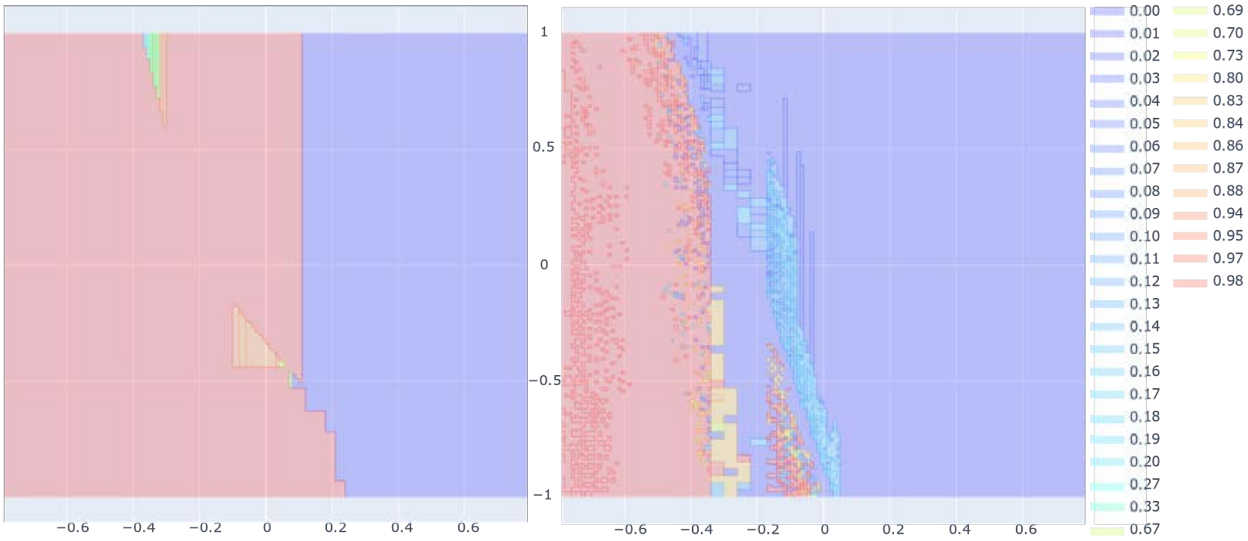


Figure 5.2: Heatmaps of failure probability upper bounds for subregions of initial states for the pendulum benchmark (x/y-axis: pole angle/angular velocity). Left: the initial abstraction; Right: the abstraction after 50 refinement steps.

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 [142] with prioritised experience replay [177], 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 [142] and [177].

5.4.2 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 5.2 (left) shows a heatmap of the probabilities for various subregions of the initial states of the inverted pendulum benchmark, within a time horizon of 7 steps. Any time horizon longer than that timed out due to the exponential state space explosion which limits the number of time steps that can be handled. The slope dividing the safe and unsafe regions is created by the trained agent favouring the left action and preferring to remain with the pole slightly tilted to the right (x-axis > 0). Figure 5.2 (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 5.4 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 5.3, a histogram showing how the probabilities are distributed across the volume of the subregions of the initial states. The plots show that the majority of the initial states considered for this environment have an upper probability of failure concentrated around 0 with few outliers that reach a probability of failure of 0.33. 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. The agent has a good overall performance but might need further refinement depending on the desired safety threshold.

Scalability and efficiency. Lastly, we briefly discuss the scalabilty and efficiency of our prototype

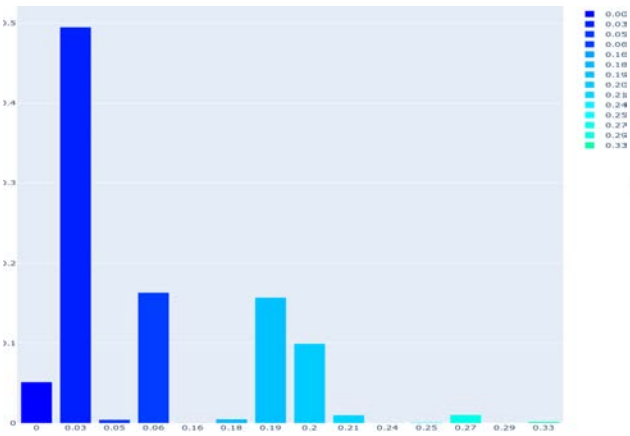


Figure 5.3: Cartpole: Histogram plot of the volume occupied by the initial state subregions, grouped by their maximum failure probability.

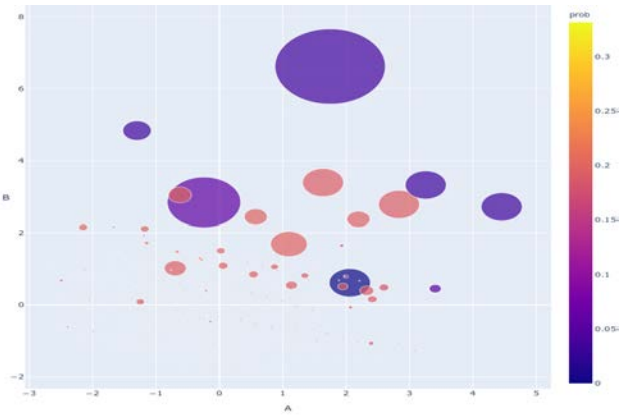


Figure 5.4: Cartpole: probability bounds for initial state subregions (the axes A and B are 2D projections from the 4D space; size denotes the volume occupied by the interval). We see that large sections of the state space have maximum probability close to 0.

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 inverted 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 cartpole problem, the number of states after 7 time-steps ranged up to approximately 75,000 states. A plot of the increasing number of states at different timesteps compared to enumerating every state up to 3 decimal digits is provided in Figure 5.5. The time required was roughly 50 minutes and 30 minutes for the two benchmarks, respectively.

5.5 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

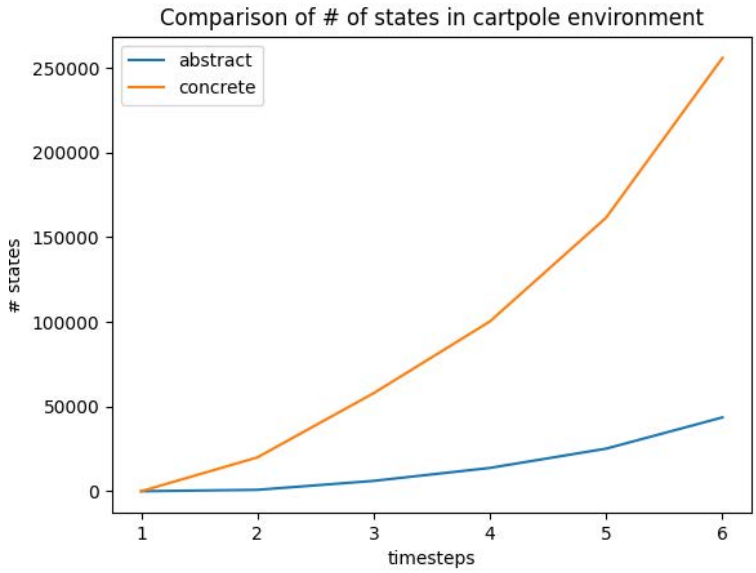


Figure 5.5: A comparison of the number of states required by the MDP using the abstraction method vs enumerating every state with a given precision ϵ of 3 decimal digits at different horizon values. The comparison covers only the first 6 timesteps because enumerating every single state becomes prohibitively expensive to compute for longer time horizons (computation time > 5 hours).

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.

Chapter Six

Verifying Probabilistic Policies with Entropy Minimisation

6.1 Introduction

Probabilistic policies are a popular tool used in reinforcement learning. This approach to policies brings a number of advantages such as managing uncertain information in the environment, dealing with adversaries in a competitive environment which might learn a counter-strategy to the current learnt policy, dealing with partial observability of the environment by breaking symmetries in the observation space, or balancing the exploration-exploitation tradeoff during training (useful for continuously learning agents) and monitoring for changes in the underlying environment (e.g.noticing that a path that was previously deemed inefficient has now become the optimal path).

However, due to their stochastic nature, it is usually possible to have at least a non-zero probability of encountering an unsafe situation and the agent eventually failing. For this reason, we want instead to establish *probabilistic* guarantees on the safe operation of the controller at runtime (as we have previously seen in Chapter 5).

As in previous chapters, we make use of abstract interpretation to reason about the dynamics of the model with groups of continuous states. In this work, though, we construct an Interval Markov Decision Process (IMDP) that models the operation of the abstracted controller in the environment, and abstract the associated action probabilities with intervals that represent the best and worst case probabilities of each action. This is achieved by combining the abstraction of the neural network output given an abstract state, the range of probabilities associated with each action (the intervals used within each transition of the IMDP) and the deterministic model of the physical environment.

One key challenge to this approach is posed by the abstraction of the probabilities for each action, which can result in extremely large over-approximating intervals. To address this, we propose a novel sampling-based approach for refinement based on entropy minimisation. The algorithm will iteratively identify areas where probability intervals are tighter and subdivide the abstract state accordingly ensuring a minimum degree of quality of the abstraction, based on the uncertainty of the over-approximation.

We evaluate our approach by applying it to deep reinforcement learning controllers trained on three benchmark environments: bouncing ball, adaptive cruise control and inverted pendulum. We also experiment by changing type of abstraction, minimum quality thresholds and abstract state aggregation to see how each element contributes to the precision and computational efficiency of the algorithm.

6.2 Controller Modelling and Abstraction

We begin by giving a formal definition of our model for the execution of a probabilistic controller, and of the abstractions that we construct to analyse these models.

6.2.1 Controller Model and Verification

As in Chapter 5, we model the execution of a probabilistic policy in a given environment as a (continuous-space, finite-branching) *discrete-time Markov process* (DTMP). We again define this for a *controller execution*, which now takes the form (π, E, S_0, S_{fail}) comprising controller policy π, an environment model E, set $S_0 \subseteq S$ of *initial states* and set $S_{fail} \subseteq S$ of *failure states*. The policy is now probabilistic, i.e., a function $\pi : S \to Dist(A)$. For simplicity, we assume that this is the only source of probabilistic behaviour, and ignore the previously used fault model f. This approach differs from the previous modelling of probabilistic behaviour because the probabilities of each action being chosen, for example due to a fault, are not consistent throughout the system but are unknown a priori and change dynamically for each state.

Definition 16 (Probabilistic controller execution model)**.** Given a controller execution (π, E, S_0, S_{fail}), the corresponding *probabilistic 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 $s, s' \in S$:

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

This is a similar style to Definition 13, but removes the effect of the fault model f and takes into account the probabilistic nature of the policy π.

On this model, we tackle the same verification problem as in the previous chapter, namely computing (a bound on) the maximum probability of reaching *failure states* within some time horizon k, $Pr_s(\Diamond^{\leqslant k} fail)$ assuming that the system may start in any initial state $s \in S_0$.

6.2.2 Controller Abstraction

We use a finite-state abstraction to compute a probabilistic guarantee. Like in the previous chapter, the states of this abstraction are *abstract states* from a set $\hat{S} \subseteq \mathcal{P}(S)$, and the model represents an over-approximation of the possible behaviour of the controller. However, to deal with the probabilistic nature of the policy, and the fact that it may choose different probabilities for actions in every state, we now use an *interval MDP* (IMDP) as our abstraction. The choices that are available in each abstract state $\hat{s}$ of the IMDP are based on the upper and lower bounds on probabilities of choosing actions in states $s \in \hat{s}$.

To define the abstraction, we again use an *environment abstraction* $\hat{E} : \hat{S} \times A \to \hat{S}$ (see Definition 14), which has the property 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 a *policy abstraction*, which gives a lower and upper bound on the probability with which each action is selected within an abstract state.

Definition 17 (Policy abstraction)**.** For a policy $\pi : S \to \mathit{Dist}(A)$ 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 $\hat{\pi}_L : \hat{S} \times A \to [0, 1]$ and $\hat{\pi}_U : \hat{S} \times A \to [0, 1]$, 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 $\hat{\pi}_L(\hat{s}, a) \leqslant \pi(s, a) \leqslant \hat{\pi}_U(\hat{s}, a)$.

In other words, for every action $a \in A$ the probability of a being selected in any concrete state within the abstract state is bounded by the upper and lower probability bounds. For convenience, we will also use $\hat{\pi}$ to refer to the bounds for a state, i.e., $\hat{\pi}(\hat{s}, a) = [\hat{\pi}_L(\hat{s}, a), \hat{\pi}_U(\hat{s}, a)]$.

Combining these notions, we can defined the IMDP abstraction of a controller execution.

Definition 18 (Probabilistic controller execution abstraction)**.** For a (probabilistic) controller execution $(\pi, E, S_0, S_{\mathit{fail}})$, a set $\hat{S} \subseteq \mathcal{P}(S)$ of abstract states and corresponding policy abstraction $\hat{\pi}$ and

environment abstraction $\hat{E}$, the *probabilistic controller execution abstraction* is defined as 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 $\hat{\mathbf{P}}(\hat{s}, j, \hat{s}') = [\hat{\mathbf{P}}_L(s, j, \hat{s}'), \hat{\mathbf{P}}_U(\hat{s}, j, \hat{s}')]$ where:

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

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

As in the previous chapter, an analysis of the IMDP then yields upper bounds on the probability of a controller exhibiting a failure within k steps ($\Diamond^{\leqslant k} fail$). This is formalised as follows (see Appendix A.2 for a proof).

Theorem 2. Given a state $s \in S$ of a probabilistic controller model DTMP, and an abstract state $\hat{s} \in \hat{S}$ of the corresponding controller abstraction IMDP for which $s \in \hat{s}$, we have:

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

where $Pr_{\hat{s}}^{\max\max}(\Diamond^{\leqslant k} fail)$ represents the maximum upper bound of the probability of failure over the non-deterministic choices of actions within k timesteps when starting from the abstract state $\hat{s}$.

6.3 Abstraction Construction

As in the previous chapter, we construct and then solve an abstract model represented by an IMDP (as opposed to MDP) using abstract interpretation. One of the main differences is the challenge of determining the abstract policy from its neural network encoding. In this chapter we explain the steps required for the abstraction of the policy.

6.3.1 Bounded Template Polyhedra Abstraction

We abstract states by using template polyhedra, convex shapes constrained within a fixed set of directions Δ (previously defined in Equation 4.7).

Template polyhedra can be bounded or unbounded depending whether every variable in the state space is bounded by the direction of the template. We are going to focus on *bounded* template polyhedra (also called *polytopes*) which are required for the refinement function later in the chapter.

6.3.2 Layer Encoding

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.

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 17), via mixed-integer linear programming (MILP), building on existing MILP encodings of neural networks [196, 48, 37]. 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:

$$\begin{aligned}
&\text{maximize} && z_{l+1}^{j} \\
&\text{subject to} && z_0 \in \hat{s}, \\
& && 0 \leqslant z_i - W_i z_{i-1} - b_i \leqslant M z_i' && \text{for } i = 1, \ldots, l, \\
& && 0 \leqslant z_i \leqslant M - M z_i' && \text{for } i = 1, \ldots, l, \\
& && 0 \leqslant z_i' \leqslant 1 && \text{for } i = 1, \ldots, l, \\
& && z_{l+1} = W_{l+1} z_l,
\end{aligned} \tag{6.1}$$

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 [196], 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 (6.1) above). Then, the upper bound for the probability of choosing action a_j is $y_{ub,j}$:

$$y_{ub,j} = \mathrm{softmax}(z_{ub,j}) \qquad \text{where} \qquad z_{ub,j}^i = \begin{cases} x_{ub,i} & \text{if } i = j \\ 1 - x_{lb,i} & \text{otherwise} \end{cases}$$

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

6.3.3 Abstract State Containment Check

When calculating the successors of abstract states we sometimes find successors which are partially or fully contained within previously visited states. In Chapter 4 we used containment for finding invariants, as, in the case of deterministic policies, the choice of action within the abstract state was known, and this allowed us to achieve unbounded time safety verification. However, with probabilistic policies, for each state there is always a small chance of choosing each different action so it is highly unlikely that we are able to find an invariant.

On the other hand, the branching factor caused by the number of actions in the probabilistic policy scenario is constant and this can cause a premature explosion of the number of states the algorithm has to keep track. To this extent, as a trade-off for introducing additional approximation errors, we can considerably reduce the number of independent abstract states by aggregating together states which are fully contained within previously visited abstract states (as shown in Fig. 6.1), hence reducing the branching factor.

6.3.4 Maximum Probability Spread

By using MILP to model both the environment and the neural network policy we can extract the range of probabilities for each action. Depending on the size of the abstract state we are analysing, the range of probabilities can get extremely wide, sometimes as wide as $\hat{\pi}(\hat{s}_i, a) = [0, 1]$ for any action a, which becomes rather uninformative about the actions chosen by the policy. This usually happens when the region covered by the polyhedron contains areas with completely different behaviours.

We define the maximum probability spread as the maximum difference between upper and

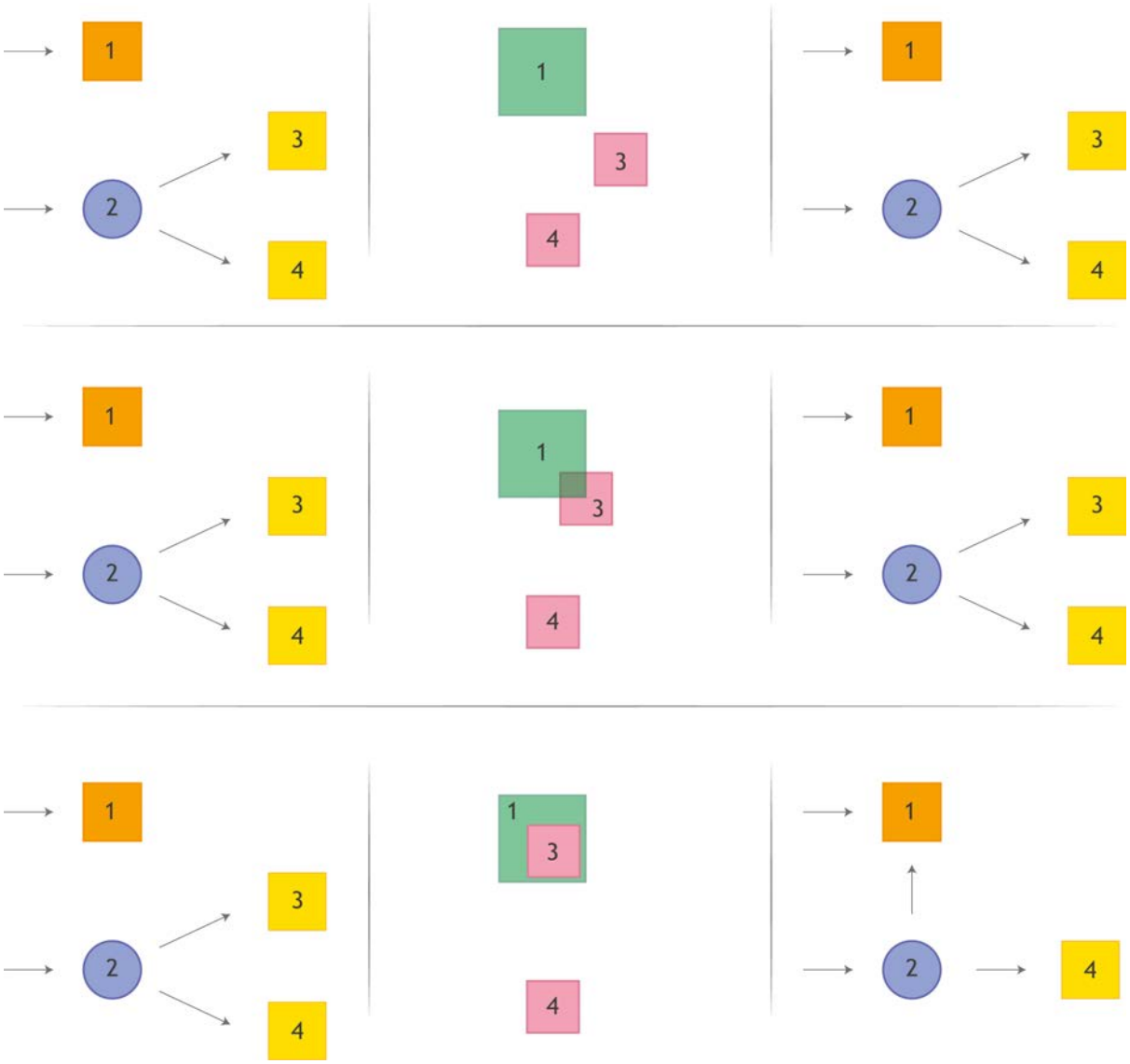


Figure 6.1: While constructing the graph, when a successors 3 and 4 are computed from a state 2 we check for containment in previously visited state 1 (green). If there is a full containment between the states (last row) we aggregate them together in the IMDP (right column).

lower bound for the abstract policy probability among any action:

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

We can tune the desired degree of precision by changing the maximum probability spread ϕ allowed for each abstract state: when the $\Delta_{\hat{\pi}}^{\max}(\hat{s}_i)$ exceeds the threshold, we refine the abstract state in smaller chunks that satisfy the maximum probability spread constraint. This step ensures a minimum quality of our probability estimates at the expense of an increase in the number of states processed.

6.4 Refinement

In order to refine our estimates we aim to split the abstract state into sub-regions in such a way that the difference $\Delta_{\hat{\pi}}^{\max}(\hat{s}_i)$ is as small as possible resulting in smaller areas with more concentrated probability ranges. The aim of the partitioning is to both group areas of the state space having similar probability ranges whilst also keeping the number of splits performed to a minimum. We try to find a good compromise between accuracy of the abstraction and number of splits because by constantly splitting the polyhedron, we may end up creating too many successors increasing the branching factor

Calculating the range of probabilities in an abstract space using MILP can be very time consuming: for this reason we take a sample of the probabilities that will underestimate the true range of probabilities when deciding whether we want to split an abstract state or not. If the sampled range of probabilities is already wide enough to trigger the split we can continue directly with the next step, otherwise we calculate the exact range of probabilities using MILP to ensure that there is no need for further refinement.

Whenever an abstract state has a Δ-probability greater than a chosen hyperparameter threshold ϕ we apply our refinement method which consists in 3 steps: sampling datapoints, choosing direction candidates, and splitting. By enforcing $\Delta_{\hat{\pi}}^{\max}(\hat{s}_i) < \phi$ we ensure that the probability range for a given path is always shrinking at a desired pace. This is because for a path of k timesteps with maximum probability range $< \phi$ at each timestep the final probability range for the path will be $< \phi^k$. This step can be found in algorithm 4 between line 8 and 10. The function `calculate_probabilities` is time consuming to compute so we use the range of sampled probabilities $\tilde{\pi}(\hat{s})$ (obtaining by performing the foward pass `forward` of the network over the sampled points) as a proxy when it exceeds the threshold ϕ. In case where $\tilde{\pi}(\hat{s})$ does not exceed the threshold, we compute the true range of probabilities $\hat{\pi}(\hat{s})$ from the abstract state when deciding whether to split the abstract state. Below we present a description of each step in detail.

6.4.1 Sampling the neural network policy

In order to group together areas of the state space according to probabilities we sample the points inside a given convex region by using the Hit & Run method [185]. After generating individual points contained in an abstract space, we obtain from the neural network the true probability distributions of picking an action associated with each point. The probability of each action is computed in a *one vs all* fashion, where we consider the probability of action a against every other action, and the action we pick for the sampling is chosen based on the widest probability range among all actions.

For each action a, we then generate a point cloud representing the probability of taking that action as opposed to any other action. Since we are focusing on a single action a at a time and a small section of the state space, the range of probabilities is smaller than the entire probability space $0 < \hat{\pi}_U(\hat{s}_i, a) < \hat{\pi}_L(\hat{s}_i, a) < 1$ (this because every action has a non-zero probability of being chosen) and might be concentrated within a small range. To improve the partitioning, we break the

tight distribution of probabilities by normalising them to the range $[0, 1]$ before continuing.

For this step we choose a fixed number of samples rather that a number based on the desired density of the datapoints: the rationale behind it is that by keeping the number of samples fixed this step takes a fixed amount of time with the density of the points increasing as the polytope being analysed becomes smaller, producing a better estimate of the policy as the algorithm generates child partitions. Figure 6.2 shows the process of sampling policy probabilities and the partitions generated by the algorithm.

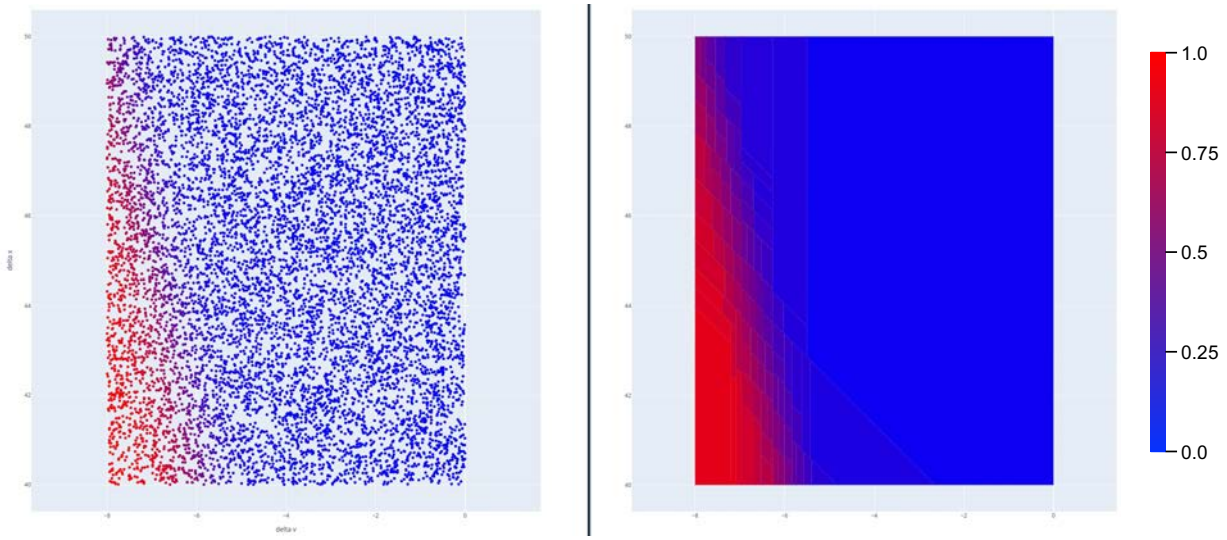


Figure 6.2: Sampled policy probabilities for one action for the adaptive cruise control in an abstract state (left) and the template polyhedra partition generated through refinement (right). X axis represents Δ-speed, Y axis represents Δ-distance.

6.4.2 Choosing Direction Candidates

When aiming to split the abstract state according to probability distribution we first have to decide how it is going to be partitioned. Since we are using bounded template polyhedra, we already have a fixed set of directions across which we can split each abstract state. By requiring a bounded template we can pick any direction to bisect the abstract state.

However, not every direction contributes equally to the reduction of the probability spread:

we will iterate through every direction and project each sampled point onto it forming a line with the probabilities of picking action a associated to each point. We will split the projection in two parts at the point that minimises a given cost function, representing the best way to partition the sequence of points.

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 each direction δ_j, we project all points in $\tilde{S}$ onto δ_j 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 $\langle \delta_j, s_i \rangle$. In addition, we want a sorted representation of the probabilities, so we define $\tilde{Y}_{s_i}^{\delta_j}$ which is a pointer to the true probability of choosing action a relative to the ith point $s \in \tilde{S}$ sorted by $\langle \delta_j, s_i \rangle$. We determine the optimal boundary for splitting in direction δ_j 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 classifying the ith point according to this split:

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

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

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

where $\tilde{Y}_{s_i}^{\delta_j}$ is the true probability of state s_i sorted according to the projection $\langle \delta_j, s_i \rangle$; which reflects how well the true probability for each point $\tilde{Y}_s^{\delta_j}$ matches the separation into the two groups. Out of all the directions δ_j we then pick the one that minimises the entropy cost function.

One of the problems with this approach is that if the distribution of probabilities is skewed to strongly favour some probabilities (e.g the probabilities are concentrated around a point with few outliers), this method does not correctly pick a good decision boundary. To counter this issue

we perform sample weighting by grouping the distribution of sampled probabilities in small bins, counting the number of samples in each bin to calculate how much weight to give to each sample. In algorithm 5 this operation is performed by the function `binning` on line 2 which returns an importance weight for each sampled point. We do not use adaptive bin size based on the number of elements in each bin because we are interested in weighting accurately each sample rather that creating bins which will weight each group the same. In this way, overly common probabilities will not outweight the least sampled ones providing a better decision boundary.

6.4.3 Abstract State Partitioning

Once thedividing value $p = \langle \delta, s_k \rangle$ at index k and the direction δ of the split are chosen, we simply add the constraints $\langle \delta, \hat{s} \rangle \leqslant p$ and $\langle \delta, \hat{s} \rangle > p$ respectively to generate the subregions. By being constrained to the directions of the template, and because the decision boundary is highly non-linear, sometimes no direction seems to provide a good candidate for the split and consequently the slices are extremely thin (the optimisation algorithm chooses a midpoint close to the interval boundaries). This causes the creation of an unnecessarily high number of successors 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.

We include the pseudocode of the entire process broken down in Algorithms 3,4 and 5.

Algorithm 3: Refine Abstract State

function `state_partitioning`(net, $\hat{S}_0$, $template$)**:**
  $\hat{S}_{frontier} = \hat{S}_0$
  $partitions = \varnothing$
  while $\hat{S}_{frontier} \neq \varnothing$ **do**
    $\hat{s} = \texttt{pop}(\hat{S}_{frontier})$
    $S_{sample} \sim \hat{s}$
    $\tilde{\pi}(\hat{s}) = \texttt{forward}(net, S_{sample})$ `// use sampled probabilities to save time`
    if $\Delta_{\tilde{\pi}(\hat{s})} \leqslant \phi$ **then**
      $\hat{\pi}(\hat{s}) = \texttt{calculate\_probabilities}(net, \hat{s})$
      if $\Delta_{\hat{\pi}(\hat{s})} \leqslant \phi$ **then**
        $partitions = partitions + \hat{s}$ `// no split`
        `continue`
    $\hat{s}_1, \hat{s}_2 = \texttt{split}(net, \hat{s}, template)$
    $\hat{S}_{frontier} = \hat{S}_{frontier} + \hat{s}_1, \hat{s}_2$
  return $partitions$

Algorithm 4: Split Abstract State

function `split`(net, $\hat{s}$, $template$)**:**
  $cost_{min} = \infty, p_{best} = \varnothing, \delta_{best} = \varnothing$
  foreach $\delta \in template$ **do**
    $projected = \langle S_{sample}, \delta \rangle$
    $\tilde{\pi}(\hat{s}) = \texttt{forward}(net, S_{sample})$
    $cost, p = \texttt{optimise\_cost}(projected, \tilde{\pi}(\hat{s}))$
    if $cost < cost_{min}$ **then**
      $cost_{min} = cost$
      $p_{best} = p$
      $\delta_{best} = \delta$
      $\hat{s}_1, \hat{s}_2 = \texttt{split\_milp}(p_{best}, \delta_{best}, \hat{s})$
  return $\hat{s}_1, \hat{s}_2$

6.5 Experimental Evaluation

In the following section we evaluate the performance of our approach for deep reinforcement learning agents on 3 different environments encountered in the previous chapters: Bouncing Ball, Stopping Car and Inverted Pendulum. All the agents have been trained using PPO with the same hyperparameters used in chapter 4, and return probabilistic policies. We are interested in

Algorithm 5: Optimise Cost

function `optimise_cost` ($projected$, $\tilde{\pi}(projected)$)**:**
  $sample\_weight = \texttt{binning}(projected, \tilde{\pi}(projected))$
  $cost_{min} = \infty, p_{best} = \varnothing$
  $yp = \tilde{\pi}(projected)$
  foreach $p \in projected$ **do**
    $yt = [projected < p ? 1 : 0]$ `// creates an array of 0 and 1`
    $cost = -\frac{1}{n}\sum_{i=0}^{n}(yt_i \log(yp_i) + (1 - yt_i)\log(1 - yp_i)) \cdot sample\_weight$
    if $cost < cost_{min}$ **then**
      $cost_{min} = cost$
      $p_{best} = p$
  return $cost_{min}, p_{best}$

calculating bounds on the worst-case probability within k steps, that is the maximum probability $Pr^{\max}_{\hat{s}}(\Diamond^{\leqslant k} fail)$, which areas of the state space are more prone to be unsafe and how some hyperparameters influence the result.

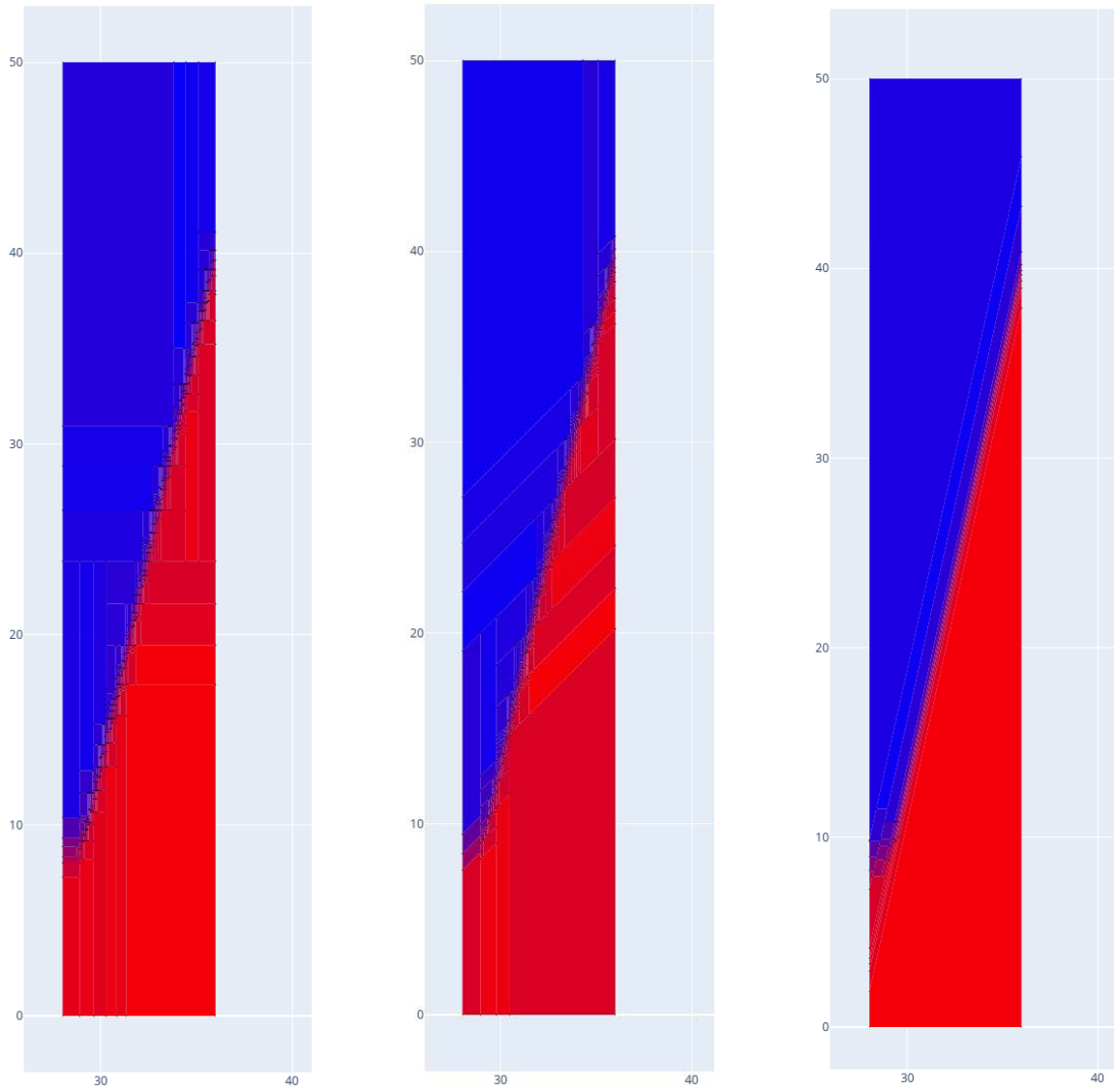
Implementation. Our implementation uses a combination of Python and Java. The neural network architecture is handled through the Pytorch library [6], the MILP modelling through Gurobi [81], and graph analysis with `networkX` [4]. The optimisation of the cost function is done through the Scikit-learn python library [162]. Constructing and solving IMDPs is done using a prototype extension of PRISM [125], through its Java API, built into a Python wrapper using `py4j` [3].

6.5.1 Type of Template

As discussed in the previous chapters, the set of directions δ used during the construction of the template affect how well the abstract states approximate the true probability values of sections of the state space. Depending on the problem, the choice of intervals or octagons can contribute to a high level of fragmentation when abstracting the state space, causing a premature explosion in the number of states.

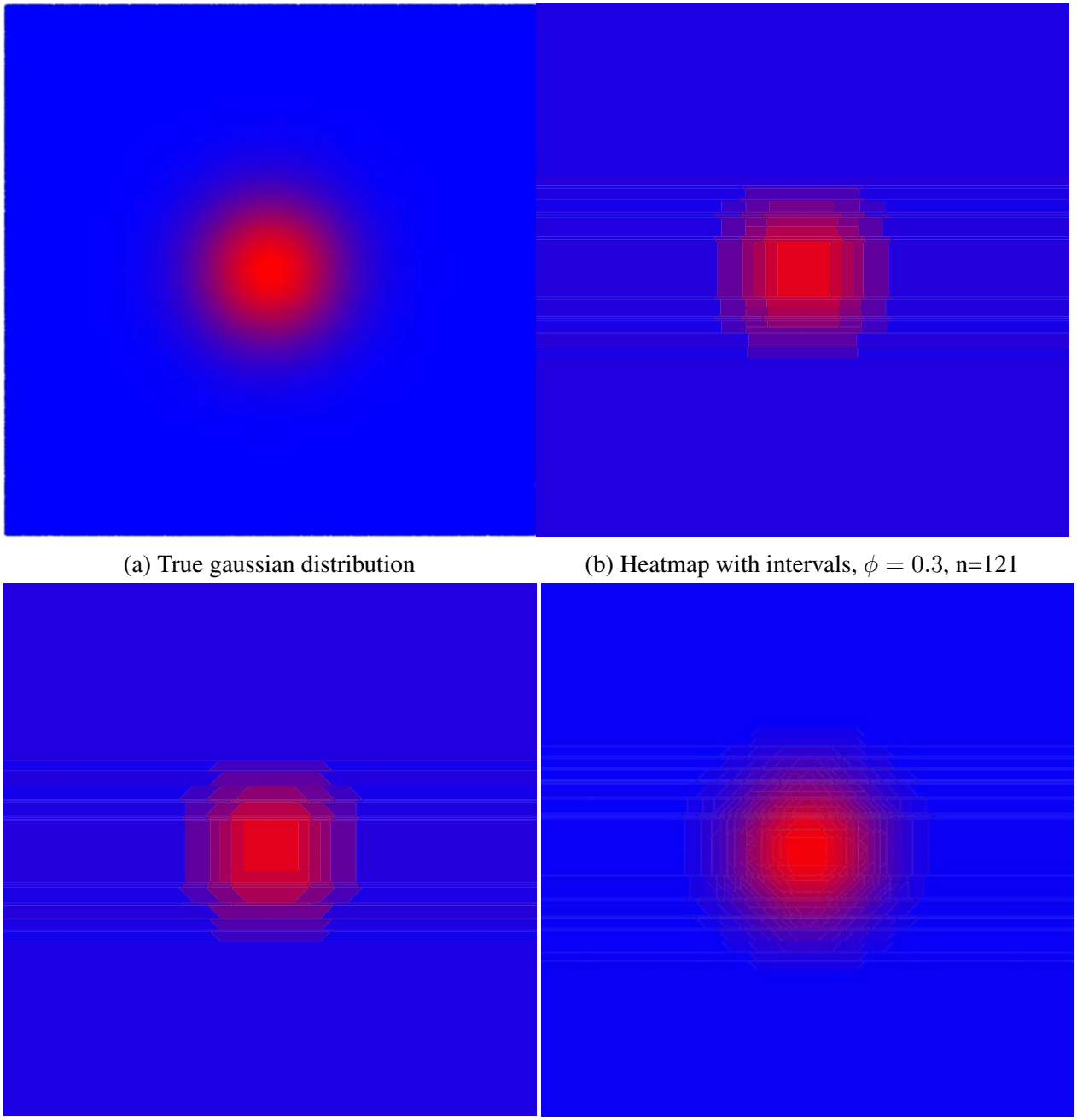
To prevent that, we sample a representative portion of the state space where the agent is

expected to operate and modify the template by choosing appropriate slopes for the directions, to better represents the decision boundaries. An example of this process is shown in Figure 6.3. By doing so, we aim to reduce the number of individual abstract states in the IMDP and speed up the model construction process.



(a) Interval: 450 items (b) Octagon: 334 items (c) Adjusted Octagon: 25 items

Figure 6.3: Heatmap of a state space section for a trained neural network policy representing the average probability of choosing the acceleration action (red) in the stopping car environment. X axis represents Δ-speed, Y axis represents Δ-distance. The type of template affects the number of abstract states created. The slope of the template in (c) is based on the neural network decision boundary in order to minimise the number of abstract states.



(a) True gaussian distribution

(b) Heatmap with intervals, $\phi = 0.3$, n=121

(c) Heatmap with octagons, $\phi = 0.3$, n=111

(d) Heatmap with octagons, $\phi = 0.1$, n=573

Figure 6.4: Example of the abstraction process applied to a toy example (Gaussian distribution) with abstracted results generated by sampling. By increasing the complexity of the template we reduce the number of abstract states as we have more directions across which we can choose how to split. By reducing the maximum probability range ϕ the number of abstract states increase exponentially whilst each abstract state gives a better representation of the true distribution.

Env.	Abs.	Max k	Contain	Num poly.	Num visited	Graph size	$Pr_{\hat{s}}^{\max}(\Diamond^{\leqslant k} fail)$	Avg runtime
BB ($\phi = 0.1, S_0 = \text{L}$)	Rect	20	✓	1727	5534	7796	0.63	30 min
BB ($\phi = 0.1, S_0 = \text{S}$)	Rect	20	✓	337	28	411	0.0	1 min
BB ($\phi = 0.1, S_0 = \text{L}$)	Oct	20	✓	2489	3045	6273	0.0	33 min
BB ($\phi = 0.1, S_0 = \text{S}$)	Oct	20	✓	352	66	484	0.0	2 min
BB ($\phi = 0.1, S_0 = \text{L}$)	Rect	20	✗	18890	0	23337	0.006	91 min
BB ($\phi = 0.1, S_0 = \text{L}$)	Oct	20	✗	13437	0	16837	0.0	111 min

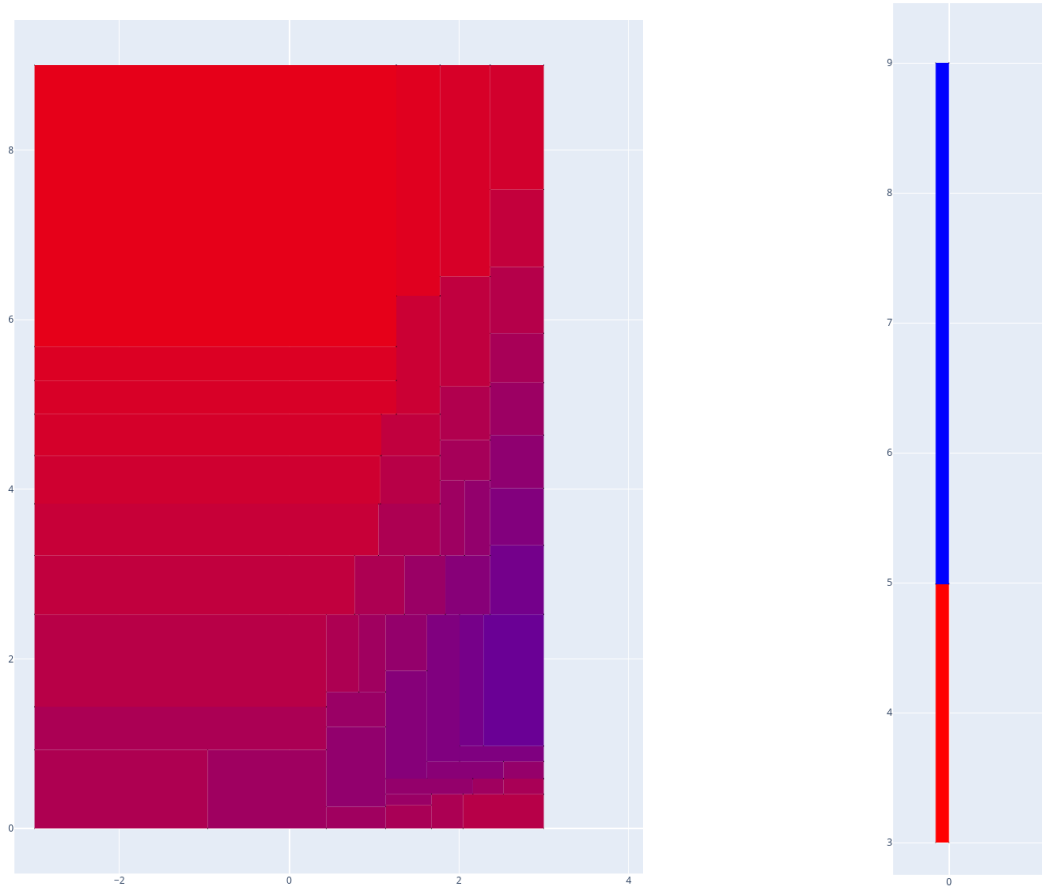
Table 6.1: Verification results for the bouncing ball (BB) environment. The results include the number of independent polytopes generated, the number of instances in which polytopes that are contained in previously visited abstract states and get aggregated together, the worst case probability of encountering an unsafe state from the initial state, and the runtime required for the IMDP construction. We experimented with different starting state (Small and Large), contain check and type of abstraction.

6.5.2 Environments

Bouncing Ball

We run our method on the Bouncing Ball environment, described previously (Section 4.4.1). We consider the starting set of states as the ball starting from a range of different heights with almost no change to the initial velocity. The ball accelerates while falling and bounces on the ground. When dropped from a height close to the ground and out of reach from the paddle the ball eventually stops and violates the safety constraint of never stopping bouncing. The agent learnt not to use the paddle when out of reach in order to preserve energy (signalled to the agent by a small negative cost).

The heatmap in Fig. 6.5 represents the maximum (worst-case) probability of encountering an unsafe state which is very high for regions out of reach from the piston, dropped from low height and no momentum. We run experiments with both a large and a small starting region: the large starting region was the area obtained by the intersection $p \in [9, 5]$ and $v \in [1, -1]$ where p and v are the position and velocity of the ball, while the small starting region was the intersection $p \in [9, 5]$ and $v \in [0, -0.1]$.



(a) Heatmap with Intervals, $\phi = 0.1$

(b) Max probability of unsafety within 20 time steps

Figure 6.5: Heatmap of the neural network policy and plot of bounded probabilistic safety for the bouncing ball environment. The X and Y axes represent the speed and position of the ball. The red area in (b) indicates regions that are out of reach of the piston and are bound to fail no matter the action of the agent.

Adaptive Cruise Control

In the Adaptive Cruise Control problem, described in Section 4.4.2, we expand on the size of the initial abstract state and customize the directions of the template to better approximate the decision boundaries present in the controller (as shown in Fig. 6.3).

Env.	Abs.	Max k	Contain	Num poly.	Num visited	Graph size	$Pr_{\hat{s}}^{\max}(\Diamond^{\leqslant k} fail)$	Avg runtime
ACC ($\phi = 0.33$)	Rect	7	✓	1522	4770	10702	0.084	85 min
ACC ($\phi = 0.33$)	Oct	7	✓	1415	2299	6394	0.078	60 min
ACC ($\phi = 0.33$)	Temp	7	✓	2440	2475	9234	0.47	70 min
ACC ($\phi = 0.5$)	Rect	7	✓	593	1589	3776	0.62	29 min
ACC ($\phi = 0.5$)	Oct	7	✓	801	881	3063	0.12	30 min
ACC ($\phi = 0.5$)	Temp	7	✓	1102	1079	4045	0.53	34 min
ACC ($\phi = 0.33$)	Box	7	✗	11334	0	24184	0.040	176 min
ACC ($\phi = 0.33$)	Oct	7	✗	7609	0	16899	0.031	152 min
ACC ($\phi = 0.33$)	Temp	7	✗	6710	0	14626	0.038	113 min
ACC ($\phi = 0.5$)	Box	7	✗	3981	0	8395	0.17	64 min
ACC ($\phi = 0.5$)	Oct	7	✗	2662	0	5895	0.12	52 min
ACC ($\phi = 0.5$)	Temp	7	✗	2809	0	6178	0.16	48 min

Table 6.2: Verification results for the adaptive cruise control (ACC) environment, with different types of abstraction, i.e., rectangular, octagonal, or customised template-based. The results include the number of independent polytopes generated, the number of instances in which polytopes that are contained in previously visited abstract states and get aggregated together, the worst case probability of encountering an unsafe state from the initial state, and the runtime required for the IMDP construction. We experiment with different components: the type of abstraction, the maximum probability range ϕ and the checking for containment in previously visited states.

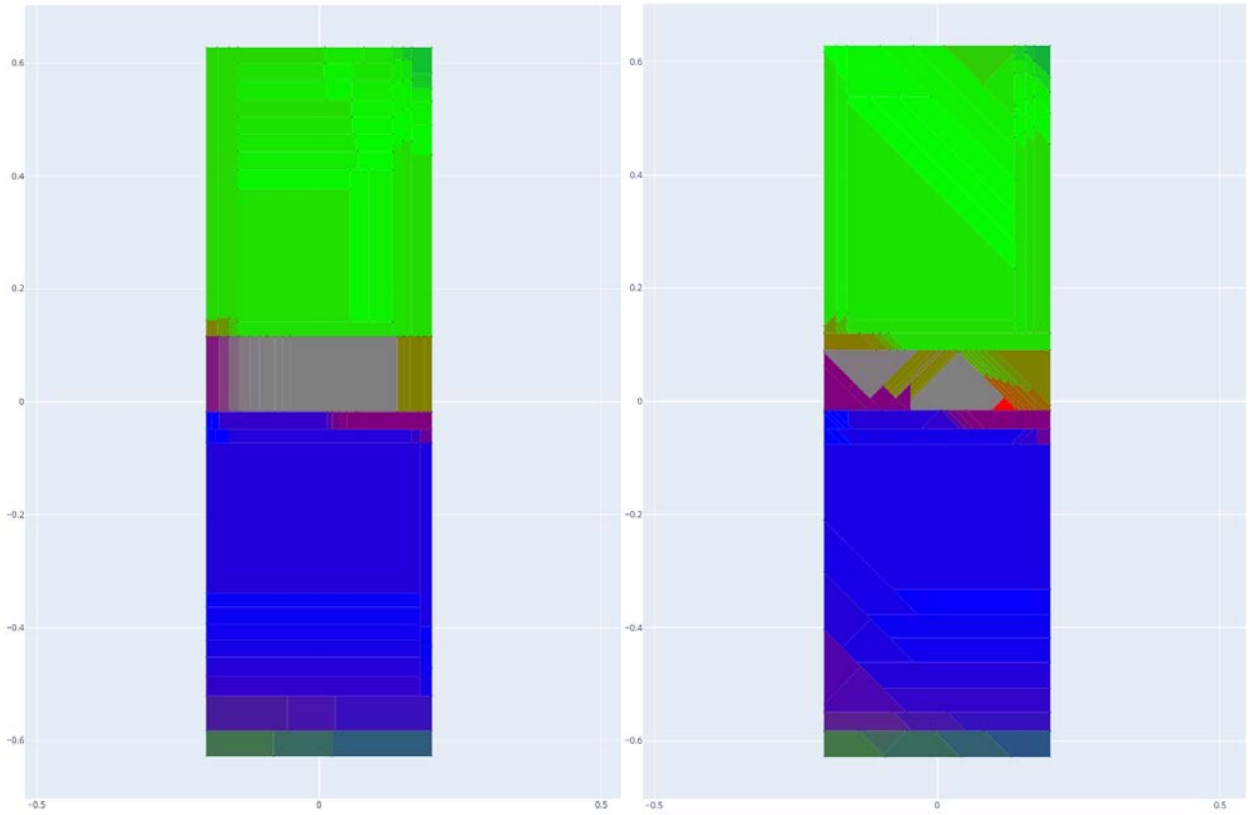
Env.	Abs.	Max k	Contain	Num poly.	Num visited	Graph size	$Pr_{\hat{s}}^{\max}(\Diamond^{\leqslant k} fail)$	Avg runtime
IP ($\phi = 0.5$)	Rect	6	✓	1494	3788	14726	0.057	71 min
IP ($\phi = 0.5$)	Rect	7	✓	na	na	na	na	timeout
IP ($\phi = 0.5$)	Oct	6	✓	na	na	na	na	timeout
IP ($\phi = 0.5$)	Box	6	✗	5436	0	16695	0.057	69 min
IP ($\phi = 0.5$)	Box	7	✗	na	0	na	na	timeout
IP ($\phi = 0.5$)	Oct	6	✗	na	0	na	na	timeout

Table 6.3: Verification results by for the inverted pendulum (IP) environment, with different types of abstraction i.e., rectangular, octagonal, or customised template-based. The results include the number of independent polytopes generated, the number of instances in which polytopes that are contained in previously visited abstract states and get aggregated together, the worst case probability of encountering an unsafe state from the initial state, and the runtime required for the IMDP construction. We limited the execution to 3h for each experiment.

Inverted Pendulum

The Inverted Pendulum problem is different from the previous problems because rather than using 2 actions as in the previous ones, the environment allows for 3 actions (noop, push left, push right). In addition, the dynamics of the system are highly non-linear making the problem more complex.

For a 3 action environment, the choice of action matters when deciding which direction to split as opposed to the previous 2 cases. When dealing with non-linearities, we adopted the same approach as in the Cartpole environment in Chapter 4 which involved subdividing the area contained in the abstract state in small intervals and mapping them to their corresponding successor after applying the linearised dynamics. In Figure 6.6 we show a heatmap of the abstract policy probability for each action to show how the choice of template affects the refinement of the abstract states.



(a) Interval

(b) Octagon

Figure 6.6: Heatmap of a state space section for a trained neural network policy representing the average probability of choosing the noop action (red), right (green) and the left action (blue) in RGB in the inverted pendulum environment. 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), the centre left area with purple tints (red and blue). Towards the bottom of the heatmap the colour fades to green as the agent try to push the pendulum to cause it to spin and balance once it reaches the opposite side.

6.5.3 Results

We have run our algorithm on the controller policies presented above. We build and solve the IMDP abstraction and are able to calculate the upper and lower bounds of failure probabilities for different subregions of the initial abstract state. The branching factor is heavily influenced by both the choice of template and the precision of the probability range ϕ that we want to maintain (shown in Fig 6.4).

Our experiments were run on a 4-core 4.2 GHz PC with 64 GB RAM running Ubuntu 18.04. We used a time horizon of 20 time-steps for the bouncing ball problem and 7 time-steps for both the adaptive cruise control and the inverted pendulum.

The bouncing ball environment is the quickest experiment to construct and verify due to the low number of variables and the simplicity of the dynamics. The policy identified 2 main areas for refining the initial state: one where it could reach the ball and should hit it and one where the ball was out of reach and the paddle should have not been activated to preserve energy. However, despite keeping ϕ very small, we noticed the creation of big abstract states when using box abstractions that ended up containing most of the other states visited by the agent. When those regions encountered an unsafe state, the value of $Pr_{\hat{s}}^{\max}(\Diamond^{\leqslant k} fail)$ increased substantially. Unfortunately those large abstract states are not broken down by the refinement algorithm because the probability range $\Delta_{\hat{\pi}}^{\max}(\hat{s}_i)$ was still within threshold. Reducing the threshold would help address this problem but at the expense of greater computational load due to the increasing number of abstract state splits.

On the adaptive cruise control problem, we run different variations. As expected, with increasingly smaller ϕ, the runtime increases and the worst case probability decreases (the overestimation error from the abstraction decreases making it closer to the true maximum probability). The checking for containing previously visited states helps reduce the computation time at the expense of some overapproximation. Modifying the template to better approximate the decision boundary in a sample of the state space does not seem to contribute to any improvement in time or precision

and instead causes a spike in the worst case probability when combined with the containment check: our explanation is that, in this configuration, abstract states tend to cover big areas of the state space uninterrupted (shown in Fig. 6.3), and hence it becomes easy to act as a sink state for smaller surrounding abstract states. When one of these macro states encountered violates safety, the maximum probability propagates back throughout the graph causing a large overapproximation of the probability.

On the inverted pendulum, most of the experiments timed out due to the high number of abstract states generated when splitting each successor and the overall speed of solving the MILP operations. Although only remotely similar, we can see that the throughput of using Mixed Integer Linear Programming is order of magnitudes lower compared to the network abstraction used in Chapter 5. In this case the use of containment check did not affect the precision of our estimates because the abstract states are sufficiently small and do not cause the overapproximations errors seen in the previous experiments.

6.6 Conclusion

We presented a novel approach to verifying probabilistic neural network policies based on MILP and abstract interpretation, extending on the work presented in the previous chapters. We propose a new refinement method based on cross-entropy aimed at breaking areas with large probability intervals into smaller regions with more concentrated ranges. We also experimented with multiple hyperparameters to improve the computational speed and reduce the overapproximation errors in the abstraction.

Chapter Seven

Conclusions

7.1 Summary

In this thesis we focused on developing novel verification techniques to provide the user with strong guarantees over the performance of neural network based controllers. In particular, we considered 3 different sources of uncertainty affecting deep reinforcement learning agents: sensory input noise, faulty actuators and probabilistic policies.

For each of these sources of uncertainty we offer a different algorithm based on abstract interpretation (and probabilistic model checking) to perform safety verification. We tackled:

- Deterministic policies in environments with noisy sensors and infinite horizon
- Deterministic policies with noisy actuators and bounded horizon
- Probabilistic policies with deterministic environments and bounded horizon

Each verification algorithm has been implemented and applied to a group of controllers trained on benchmark problems taken from the literature, experimenting with multiple different configurations and reporting results for each task.

7.2 Future Work

The work in this thesis can be extended in a number of directions. Below we present possible extensions associated to each approach.

Verifying Deep Reinforcement Learning over Unbounded Time

A possible extension to the presented work would include the application to continuous action spaces. This task brings some challenges such as the *wrapping effect* [151] where the overapproximations caused by the template build up at each timestep, which is magnified by the continuous action space.

Probabilistic Guarantees for Safe Deep Reinforcement Learning

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.

Verifying Probabilistic Policies with Entropy Minimisation

There are a number of future modifications that could improve the algorithm: (i) creating an automated way to find the important template directions and customising based on the sampling of the state space and minimising the number of abstract states, (ii) remove partial overlap between abstract states, (iii) refining abstract states based on the maximum and minimum probability of encountering an unsafe state.

Automating the discovery of important template dimension could further reduce the number

of directions employed in each template speeding up the MILP calculations and improving the overall time efficiency of the algorithm. Removing partial overlap is also a promising direction for reducing the overall number of timesteps needed for finding invariants and loops within the graph; by concentrating the computational efforts to subsections of the abstract states whose results are yet to be discovered, the algorithm should be able to keep the number of states in the IMDP low and this should allow us to extend the horizon to include states further in the future. Finally, by adopting a similar approach to the refinement of our probability estimates (as in Chapter 5) when the upper and lower bound of the probability of encountering an unsafe state dirverge, we can improve the precision of the algorithm and pinpoint with greater accuracy which areas of the state space have a high probability of failing. This requires adopting an additional strategy for breaking down abstract states into smaller subregions that is not only based on the probabilities of the controller but possibly driven by the trajectories obtained by sampling data within the abstract state. This would prevent large abstract states from forming in regions neighbouring unsafe areas leading to large overestimations in the probability of failing.

Appendix One

Appendix

A.1 Proof of Theorem 1

We give here a proof of Theorem 1, from Section 5.2.3, which states that:

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:

$$Pr_s(\Diamond^{\leqslant k} fail) \leqslant Pr^{\max}_{\hat{s}}(\Diamond^{\leqslant k} fail)$$

By the definition of $Pr^{\max}_{\hat{s}}(\cdot)$, it suffices show that there is *some* policy σ in the MDP such that:

$$Pr_s(\Diamond^{\leqslant k} fail) \leqslant Pr^{\sigma}_{\hat{s}}(\Diamond^{\leqslant k} fail) \tag{A.1}$$

Recall that, in the construction of the MDP (see Definition 15), 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 $s \in \hat{s}_{j_s}$.

The probabilities $Pr^{\sigma}_{\hat{s}}(\Diamond^{\leqslant k} fail)$ for this policy, starting in abstract state $\hat{s}$, are defined similarly to those for discrete-time Markov processes (see Section 5.2.2):

$$Pr^{\sigma}_{\hat{s}}(\Diamond^{\leqslant k} fail) = \begin{cases} 1 & \text{if } \hat{s} \models fail \\ 0 & \text{if } \hat{s} \not\models fail \wedge k{=}0 \\ \sum\limits_{\hat{s}' \in \mathrm{supp}(\hat{\mathbf{P}}(\hat{s}, j_s, \cdot))} \hat{\mathbf{P}}(\hat{s}, j_s, \hat{s}') \cdot Pr^{\sigma}_{\hat{s}'}(\Diamond^{\leqslant k-1} fail) & \text{otherwise.} \end{cases} \tag{A.2}$$

Since this is defined recursively, we prove (A.1) by induction over k.

First, for the case $k = 0$, from the computation of $Pr_s(\Diamond^{\leqslant k} fail)$ (see Section 5.2.2), we have that $Pr_s(\Diamond^{\leqslant 0} fail)$ is equal to 1 if $s \models fail$ and 0 otherwise. The same holds for $Pr^{\sigma}_{\hat{s}}(\Diamond^{\leqslant 0} fail)$, as stated above. From Definition 15, $s \models fail$ implies $\hat{s} \models fail$. Therefore, $Pr_s(\Diamond^{\leqslant 0} fail) \leqslant Pr^{\sigma}_{\hat{s}}(\Diamond^{\leqslant 0} fail)$.

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

$$\begin{array}{lll}
& Pr^{\sigma}_{\hat{s}}(\Diamond^{\leqslant k} fail) & \\
= & \sum_{\hat{s}' \in \mathrm{supp}(\hat{\mathbf{P}}(\hat{s}, j_s, \cdot))} \hat{\mathbf{P}}(\hat{s}, j_s, \hat{s}') \cdot Pr_{\hat{s}'}(\Diamond^{\leqslant k-1} fail) & \text{by defn. of } \sigma \text{ and } Pr^{\sigma}_{\hat{s}}(\Diamond^{\leqslant k} fail) \\
= & \sum_{w \in A^*} f(\pi(\hat{s}_j))(w) \cdot Pr_{\hat{E}(\hat{s}_j, w)}(\Diamond^{\leqslant k-1} fail) & \text{by defn. of } \hat{\mathbf{P}}(\hat{s}, j, \hat{s}') \\
= & \sum_{w \in A^*} f(\pi(s))(w) \cdot Pr_{\hat{E}(\hat{s}_j, w)}(\Diamond^{\leqslant k-1} fail) & \text{since } s \in \hat{s}_j \\
\geqslant & \sum_{w \in A^*} f(\pi(s))(w) \cdot Pr_{E(s,w)}(\Diamond^{\leqslant k-1} fail) & \text{by induction and since, by} \\
& & \text{Defn. 14, } E(s,w) \in \hat{E}(\hat{s}_j, w) \\
= & \sum_{s' \in \mathrm{supp}(\mathbf{P}(s, \cdot))} \mathbf{P}(s, s') \cdot Pr_{s'}(\Diamond^{\leqslant k-1} fail) & \text{by defn. of } \mathbf{P}(s, s') \\
= & Pr_s(\Diamond^{\leqslant k} fail) & \text{by defn. of } Pr_s(\Diamond^{\leqslant k} fail)
\end{array}$$

which completes the proof.

A.2 Proof of Theorem 2

We give here a proof of Theorem 2, from Section 6.2, which states that:

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

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

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:

$$Pr_s(\Diamond^{\leqslant k} fail) \leqslant Pr_{\hat{s}}^{\sigma,\tau}(\Diamond^{\leqslant k} fail) \tag{A.3}$$

Recall that, in the construction of the IMDP (see Definition 18), 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 $\hat{\mathbf{P}}_\tau(\hat{s}, j_s, \hat{s}') = \hat{\mathbf{P}}_U(\hat{s}, j, \hat{s}')$ for any $\hat{s}, j, \hat{s}'$.

The probabilities $Pr_{\hat{s}}^{\sigma,\tau}(\Diamond^{\leqslant k} fail)$ for these policies, starting in abstract state $\hat{s}$, are defined recursively in the same way as in (A.2) above. So, as there, we prove (A.3) by induction over k. The case $k = 0$ proceeds identically to show that $Pr_s(\Diamond^{\leqslant 0} fail) \leqslant Pr_{\hat{s}}^{\sigma,\tau}(\Diamond^{\leqslant 0} fail)$.

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

$Pr_s(\Diamond^{\leqslant k} fail)$. Otherwise we have:

$$
\begin{array}{lll}
 & Pr^{\sigma,\tau}_{\hat{s}}(\Diamond^{\leqslant k} fail) & \\
= & \sum_{\hat{s}' \in \mathrm{supp}(\hat{\mathbf{P}}_\tau(\hat{s}, j_s, \cdot))} \hat{\mathbf{P}}_\tau(\hat{s}, j_s, \hat{s}') \cdot Pr_{\hat{s}'}(\Diamond^{\leqslant k-1} fail) & \text{by defn. of } \sigma \text{ and } Pr^{\sigma,\tau}_{\hat{s}}(\Diamond^{\leqslant k} fail) \\
= & \sum_{\hat{s}' \in \mathrm{supp}(\hat{\mathbf{P}}_U(\hat{s}, j_s, \cdot))} \hat{\mathbf{P}}_U(\hat{s}, j_s, \hat{s}') \cdot Pr_{\hat{s}'}(\Diamond^{\leqslant k-1} fail) & \text{by defn. of } \tau \\
= & \sum_{a \in A} \pi_U(\hat{s}, a) \cdot Pr_{\hat{E}(\hat{s}_j, a)}(\Diamond^{\leqslant k-1} fail) & \text{by defn. of } \hat{\mathbf{P}}_U(\hat{s}, j, \hat{s}') \\
\geqslant & \sum_{a \in A} \pi(s, a) \cdot Pr_{\hat{E}(\hat{s}_j, a)}(\Diamond^{\leqslant k-1} fail) & \text{since } s \in \hat{s} \text{ and by Defn.17} \\
\geqslant & \sum_{a \in A} \pi(s, a) \cdot Pr_{E(s,a)}(\Diamond^{\leqslant k-1} fail) & \text{by induction and since, by} \\
 & & \text{Defn. 14, } E(s, w) \in \hat{E}(\hat{s}_j, w) \\
= & \sum_{s' \in \mathrm{supp}(\mathbf{P}(s, \cdot))} \mathbf{P}(s, s') \cdot Pr_{s'}(\Diamond^{\leqslant k-1} fail) & \text{by defn. of } \mathbf{P}(s, s') \\
= & Pr_s(\Diamond^{\leqslant k} fail) & \text{by defn. of } Pr_s(\Diamond^{\leqslant k} fail)
\end{array}
$$

which completes the proof.

References

[1] *Rtree: Spatial indexing for Python*. https://rtree.readthedocs.io/en/latest/. Accessed: 2020-05-07.

[2] *PyInterval — Interval Arithmetic in Python*. https://pyinterval.readthedocs.io/en/latest/. Accessed: 2020-05-07.

[3] *Py4J - A Bridge between Python and Java*. https://www.py4j.org/. Accessed: 2020-05-07.

[4] *NetworkX - Network Analysis in Python*. https://networkx.github.io/. Accessed: 2020-05-07.

[5] *libspatialindex*. https://libspatialindex.org/. Accessed: 2020-05-07.

[6] *PyTorch*. https://pytorch.org/. Accessed: 2020-05-07.

[7] Martin Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *None* (2015). ISSN: 0270-6474. DOI: 10.1038/nn.3331. arXiv: 1603.04467.

[8] Naveed Akhtar, Jian Liu, and Ajmal Mian. *Defense against Universal Adversarial Perturbations*. 2017. URL: http://arxiv.org/abs/1711.05929v3%20http://arxiv.org/pdf/1711.05929v3.

[9] Amin Alibakshi. "Strategies to develop robust neural network models: Prediction of flash point as a case study". In: *Analytica Chimica Acta* 1026 (2018), pp. 69–76. DOI: 10.1016/j.aca.2018.05.015. URL: http://linkinghub.elsevier.com/retrieve/pii/S0003267018306044.

[10] Mohammed Alshiekh et al. “Safe Reinforcement Learning via Shielding”. In: *AAAI*. AAAI Press, 2018, pp. 2669–2678.

[11] Mohammed Alshiekh et al. “Safe reinforcement learning via shielding”. In: *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*. 2018, pp. 2669–2678. ISBN: 9781577358008. arXiv: 1708.08611. URL: www.aaai.org.

[12] Stephen F Altschul et al. “Gapped BLAST and PSI-BLAST: a new generation of protein database search programs”. In: *Nucleic acids research* 25.17 (1997), pp. 3389–3402.

[13] Greg Anderson et al. “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)*. 2019, pp. 731–744.

[14] Anish Athalye et al. “Synthesizing Robust Adversarial Examples”. In: (2018), pp. 284–293. URL: http://proceedings.mlr.press/v80/athalye18b.html%20http://arxiv.org/abs/1707.07397.

[15] Edoardo Bacci and David Parker. *Probabilistic Guarantees for Safe Deep Reinforcement Learning*. arXiv preprint arXiv:2005.07073. 2020. arXiv: 2005.07073 [cs.AI].

[16] Edoardo Bacci and David Parker. “Probabilistic Guarantees for Safe Deep Reinforcement Learning”. In: *FORMATS*. Springer, 2020, pp. 231–248.

[17] Wenjun Bai, Changqin Quan, and Zhiwei Luo. “Alleviating adversarial attacks via convolutional autoencoder”. In: *Proceedings - 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2017*. IEEE, June 2017, pp. 53–58. ISBN: 9781509055043. DOI: 10.1109/SNPD.2017.8022700. URL: http://ieeexplore.ieee.org/document/8022700/.

[18] Stanley Bak et al. “Improved Geometric Path Enumeration for Verifying ReLU Neural Networks”. In: *CAV (1)*. Vol. 12224. LNCS. Springer, 2020, pp. 66–96.

[19] Bowen Baker et al. “Emergent Tool Use From Multi-Agent Autocurricula”. In: (Sept. 2019). arXiv: 1909.07528. URL: https://arxiv.org/abs/1909.07528v2.

[20] Stephen Baker et al. “A novel linear plasmid mediates flagellar variation in Salmonella Typhi”. In: *PLoS Pathog* 3.5 (2007), e59.

[21] Osbert Bastani. “Safe Reinforcement Learning with Nonlinear Dynamics via Model Predictive Shielding”. In: (2019). arXiv: 1905.10691. URL: http://arxiv.org/abs/1905.10691.

[22] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. “Verifiable Reinforcement Learning via Policy Extraction”. In: *Proc. 2018 Annual Conference on Neural Information Processing Systems (NeurIPS'18)*. 2018, pp. 2499–2509.

[23] Osbert Bastani et al. “Measuring Neural Net Robustness with Constraints”. In: *NIPS* (2016), pp. 2613–2621. ISSN: 10495258. arXiv: 1605.07262. URL: https://papers.nips.cc/paper/6339-measuring-neural-net-robustness-with-constraints%20http://arxiv.org/abs/1605.07262.

[24] Norbert Beckmann et al. “The R*-tree: an efficient and robust access method for points and rectangles”. In: *Proc. 1990 ACM SIGMOD International Conference on Management of Data*. 1990, pp. 322–331.

[25] Vahid Behzadan and Arslan Munir. “Vulnerability of Deep Reinforcement Learning to Policy Induction Attacks”. In: Springer, Cham, 2017, pp. 262–275. URL: http://link.springer.com/10.1007/978-3-319-62416-7_19.

[26] Yoshua Bengio et al. “Curriculum learning”. In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 41–48.

[27] Battista Biggio et al. “Evasion Attacks against Machine Learning at Test Time”. In: Springer, Berlin, Heidelberg, 2013, pp. 387–402. URL: http://link.springer.com/10.1007/978-3-642-40994-3_25.

[28] Sergiy Bogomolov et al. “Counterex-ample-Guided Refinement of Template Polyhedra”. In: *TACAS (1)*. 2017, pp. 589–606.

[29] Andreas Bougiouklis, Antonis Korkofigkas, and Giorgos Stamou. “Improving Fuel Economy with LSTM Networks and Reinforcement Learning”. In: *Proc. International Conference on Artificial Neural Networks (ICANN’18)*. 2018, pp. 230–239.

[30] T. Brázdil et al. “Verification of Markov Decision Processes using Learning Algorithms”. In: *Proc. 12th Intl. Symposium on Automated Technology for Verification and Analysis (ATVA’14)*. Vol. 8837. LNCS. Springer, 2014, pp. 98–114.

[31] Greg Brockman et al. “OpenAI Gym”. In: (June 2016). arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540.

[32] Greg Brockman et al. “OpenAI Gym”. In: *arXiv preprint arXiv:1606.01540* (2016).

[33] Greg Brockman et al. “OpenAI Gym”. In: *arXiv:1606.01540* (2016).

[34] Sebastien Bubeck, Eric Price, and Ilya Razenshteyn. “Adversarial examples from computational constraints”. In: (2018). URL: http://arxiv.org/abs/1805.10204.

[35] Rudy R Bunel et al. “A Unified View of Piecewise Linear Neural Network Verification”. In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 4790–4799. URL: http://papers.nips.cc/paper/7728-a-unified-view-of-piecewise-linear-neural-network-verification.pdf.

[36] Rudy Bunel et al. “A Unified View of Piecewise Linear Neural Network Verification”. In: *NeurIPS*. 2018, pp. 4795–4804.

[37] Rudy Bunel et al. “A unified view of piecewise linear neural network verification”. In: *Proc. 32nd International Conference on Neural Information Processing Systems (NIPS’18)*. 2018, pp. 4795–4804.

[38] Luca Cardelli et al. “Statistical Guarantees for the Robustness of Bayesian Neural Networks”. In: *Proc. International Joint Conference on Artificial Intelligence (IJCAI-19)*. 2019.

[39] Nicholas Carlini and David Wagner. *Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods*. 2017. URL: http://arxiv.org/abs/1705.07263v2%20http://arxiv.org/pdf/1705.07263v2.

[40] Nicholas Carlini and David Wagner. "Defensive Distillation is Not Robust to Adversarial Examples". In: (2016). URL: http://arxiv.org/abs/1607.04311.

[41] Nicholas Carlini and David Wagner. "Towards Evaluating the Robustness of Neural Networks". In: (2016). URL: http://arxiv.org/abs/1608.04644.

[42] Nicholas Carlini et al. "Ground-Truth Adversarial Examples". In: *ICLR 2018* (Feb. 2018), pp. 1–12. arXiv: 1709.10207. URL: https://openreview.net/forum?id=Hki-ZlbA-%20http://arxiv.org/abs/1709.10207.

[43] Steven Carr, Nils Jansen, and Ufuk Topcu. "Verifiable RNN-Based Policies for POMDPs Under Temporal Logic Constraints". In: *Proc. IJCAI'20*. To appear. 2020.

[44] Steven Carr et al. "Counterexample-Guided Strategy Improvement for POMDPs Using Recurrent Neural Networks". In: *Proc. IJCAI'19*. 2020, pp. 5532–5539.

[45] Dario Cattaruzza et al. "Unbounded-Time Analysis of Guarded LTI Systems with Inputs by Abstract Acceleration". In: *SAS*. Vol. 9291. LNCS. Springer, 2015, pp. 312–331.

[46] Hyunmin Chae et al. "Autonomous braking system via deep reinforcement learning". In: *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*. Vol. 2018-March. Institute of Electrical and Electronics Engineers Inc., Mar. 2018, pp. 1–6. ISBN: 9781538615256. DOI: 10.1109/ITSC.2017.8317839. arXiv: 1702.02302.

[47] *ZOO: Zeroth Order Optimization Based Black-box Attacks to Deep Neural Networks without Training Substitute Models*. New York, New York, USA: ACM Press, 2017. URL: http://dx.doi.org/10.1145/3128572.3140448.

[48] Chih-Hong Cheng, Georg Nuhrenberg, and Harald Ruess. "Maximum Resilience of Artificial Neural Networks". In: (2017). URL: http://arxiv.org/abs/1705.01040.

[49] Chih-Hong Cheng et al. “Verification of Binarized Neural Networks via Inter-Neuron Factoring”. In: (2017). URL: http://arxiv.org/abs/1710.03107.

[50] Richard Cheng et al. “End-to-End Safe Reinforcement Learning through Barrier Functions for Safety-Critical Continuous Control Tasks”. In: *AAAI*. AAAI Press, 2019, pp. 3387–3395.

[51] Richard Cheng et al. “End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks”. In: *33rd AAAI Conference on Artificial Intelligence, AAAI 2019, 31st Innovative Applications of Artificial Intelligence Conference, IAAI 2019 and the 9th AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019*. Vol. 33. 01. AAAI Press, July 2019, pp. 3387–3395. ISBN: 9781577358091. DOI: 10.1609/aaai.v33i01.33013387. arXiv: 1903.08792. URL: https://ojs.aaai.org/index.php/AAAI/article/view/4213.

[52] John W. Chinneck and Erik W. Dravnieks. *Locating Minimal Infeasible Constraint Sets in Linear Programs*. May 1991. DOI: 10.1287/ijoc.3.2.157. URL: http://pubsonline.informs.org/doi/abs/10.1287/ijoc.3.2.157.

[53] Moustapha Cisse et al. “Parseval Networks: Improving Robustness to Adversarial Examples”. In: (May 2017). ISSN: 1938-7228. arXiv: 1704.08847. URL: http://arxiv.org/abs/1704.08847.

[54] Ekin D. Cubuk et al. “Intriguing Properties of Adversarial Examples”. In: (2017). URL: http://arxiv.org/abs/1711.02846.

[55] George Dantzig. *Linear programming and extensions*. Princeton university press, 2016. URL: https://books.google.com/books?hl=en%5C&lr=%5C&id=hUWPDAAAQBAJ%5C&oi=fnd%5C&pg=PP1%5C&dq=Linear+Programming+and+Extensions%5C&ots=k9cYhPJGbd%5C&sig=FBLhkEvtws4aq0uwV-rBx-4Mx6c.

[56] "Defense-GAN: Protecting Classifiers Against Adversarial Attacks Using Generative Models". In: *ICLR 2018* (Feb. 2018). eprint: 1805.06605. URL: https://openreview.net/forum?id=BkJ3ibb0-%20http://arxiv.org/abs/1805.06605.

[57] Charles Desjardins and Brahim Chaib-draa. "Cooperative Adaptive Cruise Control: A Reinforcement Learning Approach". In: *IEEE Trans. Intell. Transp. Syst.* 12.4 (2011), pp. 1248–1260.

[58] Souradeep Dutta et al. "Output Range Analysis for Deep Feedforward Neural Networks". In: *NFM*. Springer, 2018, pp. 121–138.

[59] Souradeep Dutta et al. "Output Range Analysis for Deep Neural Networks". In: (2017). URL: http://arxiv.org/abs/1709.09130.

[60] Krishnamurthy Dvijotham et al. "A Dual Approach to Scalable Verification of Deep Networks". In: (2018). URL: http://arxiv.org/abs/1803.06567.

[61] Rüdiger Ehlers. "Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks". In: *ATVA*. Springer, 2017, pp. 269–286.

[62] Rüdiger Ehlers. "Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks". In: *Proc. 15th International Symposiumon Automated Technology for Verification and Analysis (ATVA'17)*. Vol. 10482. LNCS. Springer, 2017, pp. 269–286.

[63] Ivan Evtimov et al. "Robust Physical-World Attacks on Deep Learning Models". In: (2017). URL: http://arxiv.org/abs/1707.08945.

[64] Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. *Analysis of classifiers' robustness to adversarial perturbations*. 2015. URL: http://arxiv.org/abs/1502.02590v4%20http://arxiv.org/pdf/1502.02590v4.

[65] Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. "Analysis of classifiers' robustness to adversarial perturbations". In: (2015). URL: http://arxiv.org/abs/1502.02590.

[66] Volker Fischer et al. “Adversarial Examples for Semantic Image Segmentation”. In: (2017). URL: http://arxiv.org/abs/1703.01101.

[67] Goran Frehse, Mirco Giacobbe, and Thomas A. Henzinger. “Space-Time Interpolants”. In: *CAV (1)*. Springer, 2018, pp. 468–486.

[68] Jie Fu and Ufuk Topcu. “Probably Approximately Correct MDP Learning and Control With Temporal Logic Constraints”. In: *Proceedings of Robotics: Science and Systems*. 2014. DOI: 10.15607/RSS.2014.X.039.

[69] Nathan Fulton and André Platzer. “Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning”. In: *AAAI*. AAAI Press, 2018, pp. 6485–6492.

[70] Angus Galloway, Graham W. Taylor, and Medhat Moussa. “Predicting Adversarial Examples with High Confidence”. In: (2018). URL: http://arxiv.org/abs/1802.04457.

[71] Jim Gao. “Machine learning applications for data center optimization”. In: (2014).

[72] Javier Garcia and Fernando Fernandez. “A comprehensive survey on safe reinforcement learning”. In: *J. Mach. Learn. Res.* 16 (2015), pp. 1437–1480.

[73] Javier Garcıa and Fernando Fernandez. “A comprehensive survey on safe reinforcement learning”. In: *Journal of Machine Learning Research* 16 (2015), pp. 1437–1480.

[74] Timon Gehr et al. “AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation”. In: *Proc. 2018 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2018, pp. 3–18.

[75] Timon Gehr et al. “AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation”. In: *IEEE SP*. IEEE Computer Society, 2018, pp. 3–18.

[76] Justin Gilmer et al. *Adversarial Spheres*. 2018. URL: http://arxiv.org/abs/1801.02774v2%20http://arxiv.org/pdf/1801.02774v2.

[77] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. “Explaining and Harnessing Adversarial Examples”. In: (2014). URL: http://arxiv.org/abs/1412.6572.

[78] Ian Goodfellow et al. “Generative Adversarial Nets”. In: (2014), pp. 2672–2680. URL: https://papers.nips.cc/paper/5423-generative-adversarial-nets.

[79] Shixiang Gu and Luca Rigazio. “Towards Deep Neural Network Architectures Robust to Adversarial Examples”. In: (2014). URL: http://arxiv.org/abs/1412.5068.

[80] P. Gupta and J. Schumann. “A tool for verification and validation of neural network based adaptive controllers for high assurance systems”. In: *Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings.* (2004), pp. 277–278. DOI: 10.1109/HASE.2004.1281757. URL: http://ieeexplore.ieee.org/document/1281757/.

[81] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2021. URL: https://www.gurobi.com.

[82] Antonin Guttman. “R-Trees: A Dynamic Index Structure for Spatial Searching”. In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’84. Boston, Massachusetts: ACM, 1984, pp. 47–57. ISBN: 0897911288.

[83] Antonin Guttman. “R-Trees: A Dynamic Index Structure for Spatial Searching”. In: *SIGMOD Rec.* 14.2 (June 1984), pp. 47–57. ISSN: 0163-5808. DOI: 10.1145/971697.602266. URL: https://doi.org/10.1145/971697.602266.

[84] Tuomas Haarnoja et al. “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”. In: *35th International Conference on Machine Learning, ICML 2018* 5 (Jan. 2018), pp. 2976–2989. arXiv: 1801.01290. URL: https://arxiv.org/abs/1801.01290v2.

[85] Ernst Moritz Hahn et al. “Omega-Regular Objectives in Model-Free Reinforcement Learning”. In: *Proc. 25th International Conference on Tools and Algorithms for the Construction*

and Analysis of Systems (TACAS'19). Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11427. LNCS. Springer, 2019, pp. 395–412.

[86] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. "Cautious Reinforcement Learning with Logical Constraints". In: *AAMAS*. International Foundation for Autonomous Agents and Multiagent Systems, 2020, pp. 483–491.

[87] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. "Logically-Constrained Neural Fitted Q-iteration". In: *AAMAS*. IFAAMAS, 2019, pp. 2012–2014.

[88] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. "Logically-constrained neural fitted Q-iteration". In: *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*. Vol. 4. 2019, pp. 2012–2014. ISBN: 9781510892002. arXiv: 1809.07823.

[89] Mohammadhosein Hasanbeig et al. "Reinforcement Learning for Temporal Logic Control Synthesis with Probabilistic Satisfaction Guarantees". In: *CDC*. IEEE, 2019, pp. 5338–5343.

[90] Jamie Hayes and George Danezis. *Learning Universal Adversarial Perturbations with Generative Models*. 2017. URL: http://arxiv.org/abs/1708.05207v3%20http://arxiv.org/pdf/1708.05207v3.

[91] Warren He et al. "Adversarial Example Defenses: Ensembles of Weak Defenses are not Strong". In: (2017). URL: http://arxiv.org/abs/1706.04701.

[92] Geoffrey E. Hinton, Alex Krizhevsky, and Sida D. Wang. "Transforming auto-encoders". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 6791 LNCS. PART 1. Springer, Berlin, Heidelberg, 2011, pp. 44–51. ISBN: 9783642217340. DOI: 10.1007/978-3-642-21735-7_6. arXiv: 9605103 [cs]. URL: http://link.springer.com/10.1007/978-3-642-21735-7%7B%5C_%7D6.

[93] Weiwei Hu and Ying Tan. *Black-Box Attacks against RNN based Malware Detection Algorithms*. 2017. URL: http://arxiv.org/abs/1705.08131v1%20http://arxiv.org/pdf/1705.08131v1.

[94] Weiwei Hu and Ying Tan. *Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN*. 2017. URL: http://arxiv.org/abs/1702.05983v1%20http://arxiv.org/pdf/1702.05983v1.

[95] Xiaowei Huang et al. "Safety Verification of Deep Neural Networks". In: *CoRR* abs/1610.0 (2016), pp. 3–29. DOI: 10.1007/978-3-319-63387-9_1. URL: http://arxiv.org/abs/1610.06940.

[96] Xiaowei Huang et al. "Safety Verification of Deep Neural Networks". In: *Proc. 29th International Conference on Computer Aided Verification (CAV'17)*. Springer, 2017.

[97] Xiaowei Huang et al. "Safety Verification of Deep Neural Networks". In: *CAV (1)*. Springer, 2017, pp. 3–29.

[98] *Verification and validation of neural networks for safety-critical applications*. IEEE, 2002, 4789–4794 vol.6. URL: http://ieeexplore.ieee.org/document/1025416/.

[99] Andrew Ilyas et al. "Black-box Adversarial Attacks with Limited Queries and Information". In: (2018), pp. 2142–2151. URL: http://proceedings.mlr.press/v80/ilyas18a.html.

[100] Manfred Jaeger et al. "Teaching Stratego to Play Ball: Optimal Synthesis for Continuous Space MDPs". In: *ATVA*. Springer, 2019, pp. 81–97.

[101] Nils Jansen et al. "Safe reinforcement learning using probabilistic shields". In: *Leibniz International Proceedings in Informatics, LIPIcs*. Vol. 171. 2020, pp. 31–316. ISBN: 9783959771603. DOI: 10.4230/LIPIcs.CONCUR.2020.3. arXiv: 1807.06096. URL: http://shieldrl.nilsjansen.org.

[102] Saumya Jetley, Nicholas A. Lord, and Philip H. S. Torr. *With Friends Like These, Who Needs Adversaries*. 2018. URL: http://arxiv.org/abs/1807.04200v3%20http://arxiv.org/pdf/1807.04200v3.

[103] Peng Jin et al. "Learning on Abstract Domains: A New Approach for Verifiable Guarantee in Reinforcement Learning". In: (June 2021). arXiv: 2106.06931. URL: https://arxiv.org/abs/2106.06931v1%20http://arxiv.org/abs/2106.06931.

[104] J.H. Johnson, P.D. Picton, and N.J. Hallam. "Safety-critical neural computing: explanation and verification in knowledge augmented neural networks". In: *Artificial Intelligence in Engineering* 8 (1993), pp. 307–313. DOI: 10.1016/0954-1810(93)90015-8. URL: http://linkinghub.elsevier.com/retrieve/pii/0954181093900158.

[105] Sebastian Junges et al. "Safety-Constrained Reinforcement Learning for MDPs". In: *Proc. 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*. Ed. by M. Chechik and J-F. Raskin. Vol. 9636. LNCS. Springer, 2016, pp. 130–146.

[106] Danny Karmon, Daniel Zoran, and Yoav Goldberg. "LaVAN: Localized and Visible Adversarial Noise". In: (2018), pp. 2512–2520. URL: http://proceedings.mlr.press/v80/karmon18a.html.

[107] M. Kattenbelt et al. "A Game-based Abstraction-Refinement Framework for Markov Decision Processes". In: *Formal Methods in System Design* 36.3 (2010), pp. 246–280.

[108] Guy Katz et al. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *Proc. 29th International Conference on Computer Aided Verification (CAV'17)*. Vol. 10426. LNCS. Springer, 2017, pp. 97–117.

[109] Guy Katz et al. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *CAV (1)*. Vol. 10426. LNCS. Springer, 2017, pp. 97–117.

[110] *Reluplex: An efficient SMT solver for verifying deep neural networks*. Vol. 10426 LNCS. 2017, pp. 97–117. URL: http://arxiv.org/abs/1702.01135.

[111] Guy Katz et al. "The Marabou Framework for Verification and Analysis of Deep Neural Networks". In: *CAV (1)*. Springer, 2019, pp. 443–452.

[112] Guy Katz et al. "Towards Proving the Adversarial Robustness of Deep Neural Networks". In: *Electronic Proceedings in Theoretical Computer Science* 257 (2017), pp. 19–26. DOI: 10.4204/EPTCS.257.3. URL: http://arxiv.org/abs/1709.02802%20http://dx.doi.org/10.4204/EPTCS.257.3%20http://arxiv.org/abs/1709.02802v1.

[113] Yafim Kazak et al. "Verifying Deep-RL-Driven Systems". In: *Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM'19*. ACM, 2019, pp. 83–89.

[114] J. Kemeny, J. Snell, and A. Knapp. *Denumerable Markov Chains*. 2nd. Springer-Verlag, 1976.

[115] Alex Kendall et al. "Learning to Drive in a Day". In: *ICRA*. IEEE, 2019, pp. 8248–8254.

[116] Valentin Khrulkov and Ivan Oseledets. *Art of singular vectors and universal adversarial perturbations*. 2017. URL: http://arxiv.org/abs/1709.03582v2%20http://arxiv.org/pdf/1709.03582v2.

[117] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. URL: http://arxiv.org/abs/1412.6980v9%20http://arxiv.org/pdf/1412.6980v9.

[118] Diederik P Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: (2013). URL: http://arxiv.org/abs/1312.6114.

[119] Bettina Könighofer et al. "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. Springer, Cham, Oct. 2020, pp. 290–306. ISBN: 9783030613617. DOI: 10.1007/978-3-030-61362-4_16. URL: https://link.springer.com/chapter/10.1007/978-3-030-61362-4_16.

[120] Jernej Kos, Ian Fischer, and Dawn Song. “Adversarial examples for generative models”. In: (2017). URL: http://arxiv.org/abs/1702.06832.

[121] Mandar Kulkarni and Aria Abubakar. “Siamese networks for generating adversarial examples”. In: (2018). URL: http://arxiv.org/abs/1805.01431.

[122] Orna Kupferman and Moshe Y Vardi. “Model checking of safety properties”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 1633. 1999, pp. 172–183. ISBN: 3540662022. DOI: 10.1007/3-540-48683-6_17.

[123] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. “Adversarial examples in the physical world”. In: (2016). URL: http://arxiv.org/abs/1607.02533.

[124] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. “Adversarial Machine Learning at Scale”. In: (2016). URL: http://arxiv.org/abs/1611.01236.

[125] M. Kwiatkowska, G. Norman, and D. Parker. “PRISM 4.0: Verification of Probabilistic Real-time Systems”. In: *Proc. 23rd International Conference on Computer Aided Verification (CAV’11)*. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[126] Morteza Lahijania, Sean B. Andersson, and Calin Belta. “Formal Verification and Synthesis for Discrete-Time Stochastic Systems”. In: *IEEE Transactions on Automatic Control* 60.8 (2015), pp. 2031–2045.

[127] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist* (2010).

[128] Hyeungill Lee, Sungyeob Han, and Jungwoo Lee. “Generative Adversarial Trainer: Defense to Adversarial Perturbations with GAN”. In: (May 2017). arXiv: 1705.03387. URL: http://arxiv.org/abs/1705.03387.

[129] Shuo Li and Osbert Bastani. “Robust Model Predictive Shielding for Safe Reinforcement Learning with Stochastic Dynamics”. In: *ICRA*. IEEE, 2020, pp. 7166–7172.

[130] Yen-Chen Lin et al. “Tactics of Adversarial Attack on Deep Reinforcement Learning Agents”. In: (Mar. 2017). arXiv: 1703.06748. URL: http://arxiv.org/abs/1703.06748.

[131] Yanpei Liu et al. *Delving into Transferable Adversarial Examples and Black-box Attacks*. 2016. URL: http://arxiv.org/abs/1611.02770v3%20http://arxiv.org/pdf/1611.02770v3.

[132] Alessio Lomuscio and Lalit Maganti. “An approach to reachability analysis for feed-forward ReLU neural networks”. In: (2017). URL: https://arxiv.org/abs/1706.07351.

[133] Jiajun Lu, Hussein Sibai, and Evan Fabry. “Adversarial Examples that Fool Detectors”. In: (2017). URL: http://arxiv.org/abs/1712.02494.

[134] Matt Luckcuck et al. “Formal Specification and Verification of Autonomous Robotic Systems: A Survey”. In: *ACM Comput. Surv.* 52.5 (2019), 100:1–100:41.

[135] Haitong Ma et al. “Feasible Actor-Critic: Constrained Reinforcement Learning for Ensuring Statewise Safety”. In: (2021). arXiv: 2105.10682. URL: http://arxiv.org/abs/2105.10682.

[136] Marlos C Machado et al. “Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents”. In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 523–562.

[137] *Towards Deep Learning Models Resistant to Adversarial Attacks*. 2018. URL: https://openreview.net/forum?id=rJzIBfZAb.

[138] Alireza Makhzani et al. “Adversarial Autoencoders”. In: (2015). URL: http://arxiv.org/abs/1511.05644.

[139] Jan Hendrik Metzen et al. *Universal Adversarial Perturbations Against Semantic Image Segmentation*. 2017. URL: http://arxiv.org/abs/1704.05712v3%20http://arxiv.org/pdf/1704.05712v3.

[140] *Differentiable Abstract Interpretation for Provably Robust Neural Networks*. Vol. International Conference on Machine Learning. 2018, pp. 3575–3583. URL: http://proceedings.mlr.press/v80/mirman18b/mirman18b.pdf.

[141] Volodymyr Mnih et al. “Asynchronous methods for deep reinforcement learning”. In: *33rd International Conference on Machine Learning, ICML 2016*. Vol. 4. International Machine Learning Society (IMLS), Feb. 2016, pp. 2850–2869. ISBN: 9781510829008. arXiv: 1602.01783. URL: https://arxiv.org/abs/1602.01783v2.

[142] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 14764687. DOI: 10.1038/nature14236.

[143] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. “DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks”. In: *CVPR*. IEEE Computer Society, 2016, pp. 2574–2582.

[144] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. “DeepFool: a simple and accurate method to fool deep neural networks”. In: (2015). URL: http://arxiv.org/abs/1511.04599.

[145] Seyed-Mohsen Moosavi-Dezfooli et al. *Analysis of universal adversarial perturbations*. 2017. URL: http://arxiv.org/abs/1705.09554v1%20http://arxiv.org/pdf/1705.09554v1.

[146] Seyed-Mohsen Moosavi-Dezfooli et al. *Universal adversarial perturbations*. 2016. URL: http://arxiv.org/abs/1610.08401v3%20http://arxiv.org/pdf/1610.08401v3.

[147] Konda Reddy Mopuri, Aditya Ganeshan, and R. Venkatesh Babu. *Generalizable Data-free Objective for Crafting Universal Adversarial Perturbations*. 2018. URL: http://arxiv.org/abs/1801.08092v3%20http://arxiv.org/pdf/1801.08092v3.

[148] Konda Reddy Mopuri, Utsav Garg, and R. Venkatesh Babu. *Fast Feature Fool: A data independent approach to universal adversarial perturbations*. 2017. URL: http://arxiv.org/abs/1707.05572v1%20http://arxiv.org/pdf/1707.05572v1.

[149] Konda Reddy Mopuri, Phani Krishna Uppala, and R. Venkatesh Babu. *Ask, Acquire, and Attack: Data-free UAP Generation using Class Impressions*. 2018. URL: http://arxiv.org/abs/1808.01153v1%20http://arxiv.org/pdf/1808.01153v1.

[150] Nina Narodytska et al. “Verifying Properties of Binarized Deep Neural Networks”. In: (2017). URL: http://arxiv.org/abs/1709.06662.

[151] Arnold Neumaier. “The wrapping effect, ellipsoid arithmetic, stability and confidence regions”. In: *Validation numerics*. Springer, 1993, pp. 175–190.

[152] Anh Nguyen, Jason Yosinski, and Jeff Clune. “Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images”. In: (2014). URL: http://arxiv.org/abs/1412.1897.

[153] Eshed Ohn-Bar and Mohan Manubhai Trivedi. “Looking at Humans in the Age of Self-Driving and Highly Automated Vehicles”. In: *IEEE Trans. Intelligent Vehicles* 1.1 (2016), pp. 90–104. ISSN: 2379-8904. DOI: 10.1109/TIV.2016.2571067.

[154] OpenAI. *OpenAI Five*. 2018. URL: https://blog.openai.com/openai-five/ (visited on).

[155] Tianyu Pang et al. “Towards Robust Detection of Adversarial Examples”. In: (2017). URL: http://arxiv.org/abs/1706.00633.

[156] Nicolas Papernot and Patrick McDaniel. “On the Effectiveness of Defensive Distillation”. In: (2016). URL: http://arxiv.org/abs/1607.05113.

[157] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. *Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples*. 2016. URL: http://arxiv.org/abs/1605.07277v1%20http://arxiv.org/pdf/1605.07277v1.

[158] Nicolas Papernot et al. “Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks”. In: (2015). URL: http://arxiv.org/abs/1511.04508.

[159] Nicolas Papernot et al. “Practical Black-Box Attacks against Machine Learning”. In: (2016). URL: http://arxiv.org/abs/1602.02697.

[160] Nicolas Papernot et al. *The Limitations of Deep Learning in Adversarial Settings*. 2015. URL: http://arxiv.org/abs/1511.07528v1%20http://arxiv.org/pdf/1511.07528v1.

[161] Shashank Pathak, Luca Pulina, and Armando Tacchella. "Verification and repair of control policies for safe reinforcement learning". In: *Applied Intelligence* (2017), pp. 1–23. DOI: 10.1007/s10489-017-0999-8.

[162] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[163] Arnu Pretorius, Steve Kroon, and Herman Kamper. "Learning Dynamics of Linear Denoising Autoencoders". In: (2018), pp. 4138–4147. URL: http://proceedings.mlr.press/v80/pretorius18a.html.

[164] Luca Pulina and Armando Tacchella. "An Abstraction-Refinement Approach to Verification of Artificial Neural Networks". In: *CAV*. Vol. 6174. LNCS. Springer, 2010, pp. 243–257.

[165] Luca Pulina and Armando Tacchella. "Challenging SMT Solvers to Verify Neural Networks". In: *AI Commun.* 25 (2012), pp. 117–135. URL: http://dl.acm.org/citation.cfm?id=2350156.2350160.

[166] Luca Pulina and Armando Tacchella. "Challenging SMT solvers to verify neural networks". In: *AI Commun.* 25.2 (2012), pp. 117–135.

[167] Luca Pulina and Armando Tacchella. "NeVer: a tool for artificial neural networks verification". In: *Annals of Mathematics and Artificial Intelligence* 62 (2011), pp. 403–425. DOI: 10.1007/s10472-011-9243-0. URL: http://link.springer.com/10.1007/s10472-011-9243-0.

[168] Martin L Puterman. "Markov decision processes". In: *Handbooks in operations research and management science* 2 (1990), pp. 331–434.

[169] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. "Certified Defenses against Adversarial Examples". In: (2018). URL: http://arxiv.org/abs/1801.09344.

[170] Andras Rozsa, Manuel Gunther, and Terrance E. Boult. "Towards Robust Deep Neural Networks with BANG". In: (2016). URL: http://arxiv.org/abs/1612.00138.

[171] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. “Reachability Analysis of Deep Neural Networks with Provable Guarantees”. In: *Proc. 27th International Joint Conference on Artificial Intelligence (IJCAI’18)*. 2018.

[172] Lukas Ruff et al. “Deep One-Class Classification”. In: (2018), pp. 4390–4399. URL: http://proceedings.mlr.press/v80/ruff18a.html.

[173] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. “Dynamic Routing Between Capsules”. In: *NIPS* (2017), pp. 3856–3866. ISSN: 10495258. DOI: 10.1371/journal.pone.0035195. arXiv: 1710.09829. URL: https://papers.nips.cc/paper/6975-dynamic-routing-between-capsules.

[174] Sriram Sankaranarayanan, Thao Dang, and Franjo Ivancic. “Symbolic Model Checking of Hybrid Systems Using Template Polyhedra”. In: *TACAS*. Vol. 4963. LNCS. Springer, 2008, pp. 188–202.

[175] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. “Scalable Analysis of Linear Systems Using Mathematical Programming”. In: *VMCAI*. Springer, 2005, pp. 25–41.

[176] Kevin Scaman and Aladin Virmaux. “Lipschitz regularity of deep neural networks: analysis and efficient estimation”. In: *arXiv preprint arXiv:1805.10965* (2018). URL: https://arxiv.org/pdf/1805.10965.

[177] Tom Schaul et al. “Prioritized experience replay”. In: *arXiv preprint arXiv:1511.05952* (2015).

[178] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: (July 2017). arXiv: 1707.06347. URL: https://arxiv.org/abs/1707.06347.

[179] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *arXiv:1707.06347* (2017).

[180] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. "Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving". In: *arXiv preprint arXiv:1610.03295* (2016). arXiv: 1610.03295.

[181] *Accessorize to a Crime: Real and Stealthy Attacks on State-of-the-Art Face Recognition*. New York, New York, USA: ACM Press, 2016, pp. 1528–1540. URL: http://dl.acm.org/citation.cfm?doid=2976749.2978392.

[182] Shiwei Shen et al. "APE-GAN: Adversarial Perturbation Elimination with GAN". In: (July 2017). arXiv: 1707.05474. URL: https://arxiv.org/abs/1707.05474.

[183] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nat.* 550.7676 (2017), pp. 354–359.

[184] Gagandeep Singh et al. "An abstract domain for certifying neural networks". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 41:1–41:30.

[185] Robert L. Smith. "Efficient Monte Carlo Procedures for Generating Points Uniformly Distributed Over Bounded Regions". In: *Operations Research* 32.6 (1984), pp. 1296–1308. ISSN: 0030364X, 15265463. URL: http://www.jstor.org/stable/170949.

[186] Yang Song et al. "Generative Adversarial Examples". In: (2018). URL: http://arxiv.org/abs/1805.07894.

[187] Sadegh Soudjani, Caspar Gevaerts, and Alessandro Abate. "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)*. Vol. 9035. LNCS. Springer, 2015, pp. 272–286.

[188] Krishnan Srinivasan et al. "Learning to be Safe: Deep RL with a Safety Critic". In: (2020). arXiv: 2010.14603. URL: http://arxiv.org/abs/2010.14603.

[189] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[190] Richard S Sutton et al. “Policy Gradient Methods for Reinforcement Learning with Function Approximation”. In: ().

[191] Christian Szegedy et al. “Intriguing properties of neural networks”. In: (2013). URL: http://arxiv.org/abs/1312.6199.

[192] Christian Szegedy et al. “Intriguing properties of neural networks”. In: *ICLR (Poster)*. 2014.

[193] Pedro Tabacof, Julia Tavares, and Eduardo Valle. “Adversarial Images for Variational Autoencoders”. In: (Dec. 2016). arXiv: 1612.00155. URL: https://arxiv.org/abs/1612.00155.

[194] *Ai2: Safety and robustness certification of neural networks with abstract interpretation*. 2018.

[195] Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. “Evaluating Robustness of Neural Networks with Mixed Integer Programming”. In: *ICLR (Poster)*. OpenReview.net, 2019.

[196] Vincent Tjeng, Kai Xiao, and Russ Tedrake. *Evaluating Robustness of Neural Networks with Mixed Integer Programming*. 2017. URL: http://arxiv.org/abs/1711.07356v2%20http://arxiv.org/pdf/1711.07356v2.

[197] Florian Tramèr et al. “Ensemble Adversarial Training: Attacks and Defenses”. In: (2017). URL: http://arxiv.org/abs/1705.07204.

[198] Florian Tramèr et al. “The Space of Transferable Adversarial Examples”. In: (2017). URL: http://arxiv.org/abs/1704.03453.

[199] Hoang-Dung Tran et al. “NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems”. In: *CAV (1)*. Springer, 2020, pp. 3–17.

[200] Hoang-Dung Tran et al. “Verification of Deep Convolutional Neural Networks Using ImageStars”. In: *CAV (1)*. Vol. 12224. LNCS. Springer, 2020, pp. 18–42.

[201] Vladimir Vapnik. “Principles of risk minimization for learning theory”. In: *Advances in neural information processing systems*. 1992, pp. 831–838. ISBN: 1-55860-222-4. URL: https://papers.nips.cc/paper/506-principles-of-risk-minimization-for-learning-theory%20http://papers.nips.cc/paper/506-principles-of-risk-minimization-for-learning-theory.

[202] Shiqi Wang et al. “Formal Security Analysis of Neural Networks using Symbolic Intervals”. In: *Proc. 27th USENIX Security Symposium*. 2018, pp. 1599–1614.

[203] Tao Wang et al. “Reading digits in natural images with unsupervised feature learning”. In: *NIPS* (2011). DOI: 10.2118/18761-MS.

[204] Christopher J. C. H. Watkins and Peter Dayan. “Q-learning”. In: *Machine Learning* 8.3-4 (May 1992), pp. 279–292. ISSN: 0885-6125. DOI: 10.1007/bf00992698.

[205] Tsui-Wei Weng et al. “Towards Fast Computation of Certified Robustness for ReLU Networks”. In: (2018), pp. 5273–5282. URL: http://arxiv.org/abs/1804.09699%20http://proceedings.mlr.press/v80/weng18a.html.

[206] K A Wetterstrand. *DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP)*. 2016. URL: www.genome.gov/sequencingcosts.

[207] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. “Feature-Guided Black-Box Safety Testing of Deep Neural Networks”. In: (2017). URL: http://arxiv.org/abs/1710.07859.

[208] E. Wolff, U. Topcu, and R. Murray. “Robust Control of Uncertain Markov Decision Processes with Temporal Logic Specifications”. In: *Proc. 51th IEEE Conference on Decision and Control (CDC’12)*. 2012, pp. 3372–3379.

[209] Eric Wong and J. Zico Kolter. “Provable defenses against adversarial examples via the convex outer adversarial polytope”. In: (2017). URL: http://arxiv.org/abs/1711.00851.

[210] G. R. Wood and B. P. Zhang. “Estimation of the Lipschitz constant of a function”. In: *Journal of Global Optimization* 8.1 (1996), pp. 91–103. DOI: 10.1007/BF00229304. URL: https://doi.org/10.1007/BF00229304.

[211] Xi Wu et al. “Reinforcing Adversarial Robustness using Model Confidence Induced by Adversarial Training”. In: (2018), pp. 5330–5338. URL: http://proceedings.mlr.press/v80/wu18e.html.

[212] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. “Output Reachable Set Estimation and Verification for Multilayer Neural Networks”. In: *IEEE Trans. Neural Networks Learn. Syst.* 29.11 (2018), pp. 5777–5783.

[213] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. “Reachable Set Computation and Safety Verification for Neural Networks with ReLU Activations”. In: (2017). URL: http://arxiv.org/abs/1712.08163.

[214] Chaowei Xiao et al. “Generating Adversarial Examples with Adversarial Networks”. In: (2018). URL: http://arxiv.org/abs/1801.02610.

[215] Han Xiao, Kashif Rasul, and Roland Vollgraf. “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms”. In: *CoRR* abs/1708.07747 (2017). arXiv: 1708.07747. URL: http://arxiv.org/abs/1708.07747.

[216] Weilin Xu, David Evans, and Yanjun Qi. “Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks”. In: (2017). DOI: 10.14722/ndss.2018.23198. URL: http://arxiv.org/abs/1704.01155%20http://dx.doi.org/10.14722/ndss.2018.23198.

[217] *Verification of a trained neural network accuracy*. Vol. 3. IEEE, 2001, pp. 1657–1662. URL: http://ieeexplore.ieee.org/document/938410/.

[218] Hengjun Zhao et al. “Synthesizing barrier certificates using neural networks”. In: *HSCC 2020 - Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control ,part of CPS-IoT Week*. Vol. 20. New York, NY, USA: ACM, 2020. ISBN: 9781450370189. DOI: 10.1145/3365365.3382222. URL: https://doi.org/10.1145/3365365.3382222.

[219] He Zhu et al. “An inductive synthesis framework for verifiable reinforcement learning”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, June 2019, pp. 686–701. ISBN: 9781450367127. DOI: 10.1145/3314221.3314638. arXiv: 1907.07273.