Formally-Sharp DAgger for MCTS: Lower-Latency Monte Carlo Tree Search using Data Aggregation with Formal Methods

Debraj Chakraborty
Université Libre de Bruxelles
Brussels, Belgium
debraj.chakraborty@ulb.be

Damien Busatto-Gaston
Univ. Paris Est Créteil, LACL, F-94010
Creteil, France
damien.busatto-gaston@u-pec.fr

Jean-François Raskin
Université Libre de Bruxelles
Brussels, Belgium
jean-francois.raskin@ulb.be

Guillermo A. Pérez
University of Antwerp
Antwerp, Belgium
guillermo.perez@uantwerpen.be

ABSTRACT

We study how to efficiently combine formal methods, Monte Carlo Tree Search (MCTS), and deep learning in order to produce high-quality receding horizon policies in large Markov Decision processes (MDPs). In particular, we use model-checking techniques to guide the MCTS algorithm in order to generate offline samples of high-quality decisions on a representative set of states of the MDP. Those samples can then be used to train a neural network that imitates the policy used to generate them. This neural network can either be used as a guide on a lower-latency MCTS online search, or alternatively be used as a full-fledged policy when minimal latency is required. We use statistical model checking to detect when additional samples are needed and to focus those additional samples on configurations where the learnt neural network policy differs from the (computationally-expensive) offline policy. We illustrate the use of our method on MDPs that model the Frozen Lake and Pac-Man environments — two popular benchmarks to evaluate reinforcement-learning algorithms.

KEYWORDS

Markov decision processes; Neural networks; Monte Carlo tree search; Model checking; Formal methods

ACM Reference Format:
Debraj Chakraborty, Damien Busatto-Gaston, Jean-François Raskin, and Guillermo A. Pérez. 2023. Formally-Sharp DAgger for MCTS: Lower-Latency Monte Carlo Tree Search using Data Aggregation with Formal Methods. In *Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), London, United Kingdom, May 29 – June 2, 2023*, IFAAMAS, 9 pages.

1 INTRODUCTION

Markov decision processes (MDPs) are frameworks to model sequential decision making. They are discrete-time stochastic models where an agent chooses actions based on the current state. The agent then receives a reward and the state of the MDP is updated based on a probabilistic transition function. Exact algorithms, or formal methods, for MDPs have been studied since the 1950s and efficient (and symbolic) versions of these algorithms have been implemented in probabilistic model-checking tools such as PRISM [23] and Storm [16]. The latter, as well as other tools, are regularly compared with respect to a large body of benchmarks from the Quantitative Verification Benchmark Set [7].

While tools like PRISM and Storm can handle very large systems, some applications arising from real-world systems and video games like Pac-Man are still out of reach. In contrast, novel (deep) reinforcement learning (RL) techniques or online heuristic search techniques, like Monte Carlo Tree Search (MCTS) [6], are able to produce policies for larger MDPs [3], albeit at the cost of either high sample complexity (*i.e.* they require much data to be trained), or high latency (*i.e.* they require much time before choosing a next action), and weaker performance guarantees.

In this work, we aim at combining exact methods, such as model checking, and MCTS to improve the quality of policies synthesized in large MDPs. Concretely, we make use of the MCTS algorithm with *symbolic advice* (coming from formal methods), as proposed in [8], to increase reliability of MCTS. Further, to improve the latency of MCTS augmented with advice, we propose to replace advice coming from exact algorithms with a neural network, trained on data from the exact advice, that we call *neural advice*. Finally, we also experiment with training a *surrogate* neural-network policy to imitate MCTS (with advice) altogether. Once more, to realize this efficiently, with respect to sample complexity, we leverage exact methods to obtain "perfect data" and we generate additional samples on demand when the performance of the learnt neural network does not match the quality of the policy computed offline. This step uses statistical model checking [13] instead of classical metrics from machine learning.

Contribution. We consider our main contributions to be (1) an *expert imitation framework* to train a neural network in order to replace exact advice by lower-latency neural advice, or to imitate the *expert* policy that can be computed offline, and (2) this imitation framework relies on a data generation algorithm which leverages formal methods to obtain "perfect data" for our samples and to generate additional samples, as long as statistical model checking indicates that it is required to improve the quality of the imitation.

Imitating experts. Imitation can take different forms depending on the expert policy (or advice). In general, we define a ranking of actions for every state such that the maximally ranked elements

Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), A. Ricci, W. Yeoh, N. Agmon, B. An (eds.), May 29 – June 2, 2023, London, United Kingdom.

are those played by the policy. Intuitively, the ranking tells us how good every action is from the current state. We propose to train a neural network to learn such a ranking function as an offline step. This neural network can then be used as a full-fledged policy or as a neural advice to efficiently guide MCTS. The neural advice aims for an expected reward comparable with the expert advice, for a fraction of its online latency.

Data generation and aggregation. Recall that we propose to train a neural network to imitate an expert advice. The expert advice is usually implemented as an exact algorithm. In this case, given a set of inputs for the neural network, the original expert advice can be used to obtain (offline) a "perfect" set of corresponding outputs to train on. In contrast, when training a neural network to imitate the full MCTS-with-advice algorithm, the data can be noisy for one of two reasons: we are sampling from a randomized policy, and the expert policy we are imitating may not always match the optimal policy of the MDP. In both cases, a remaining challenge is to generate a representative set of inputs for the network to be trained on. We propose to enrich the set of data using formal methods to compare the behaviour of the trained neural network with that of the expert policy in what resembles a Counterexample-Guided Abstraction-Refinement loop [12]. Our experiments show that this data aggregation loop can speed up learning significantly.

Evaluating a neural network. In order to stop the data aggregation loop, we do not only rely on classical machine learning criteria to evaluate the quality of the generated policies, but also monitor the practical performance of the neural networks. Indeed, our setting requires taking decisions sequentially for many steps, so that small errors could accumulate over time. Thus, classical metrics such as computing a loss function on a testing dataset may not be representative of the expected reward a neural network will obtain when used as an advice or a full policy, e.g. a policy may make mistakes at crucial moments despite being almost always correct in its decisions. Instead, we use statistical model-checking to compute an approximation of the expected reward of our policies.

Related work. Our implementation of the MCTS algorithm with symbolic advice closely follows the approach described in [8]. However, while they relied on qualitative advice based on quantified Boolean formulae (QBF) and SAT solvers, we use more quantitative notions instead, based on probabilistic model checking and neural networks. Our approach also resembles the *shielding* framework [1, 21] used to add safety properties to RL algorithms. One difference is that our technique does not require one to construct the entire MDP, making our work scalable to larger MDPs.

Using deep learning to replace expert (but expensive) policies by learnt policies is known to be advantageous when the expert policy is unable to meet real-time (latency) constraints (see, e.g. [20, Section 5.2] and [18]). In order to obtain a satisfactory dataset to train on, we propose a sharp variant of the DAgger algorithm, a dataset aggregation technique introduced in [26, 27]. A notable difference is that we propose to use model checkers instead of human experts in order to get better-quality data. We also identify so-called *counterexample* configurations in order to guide the aggregation loop to the most interesting states. This is reminiscent of counterexample guided abstraction refinement (CEGAR) approaches for hybrid systems such as [14] that identify states violating a property then focus the deep learning procedures on such states.

Finally, we rely on statistical model checking [30] to efficiently evaluate particular policies for the system. This consists in running simulated executions of the MDP and computing statistics with confidence guarantees. However, such techniques are not known to find (or approximate) the optimal policies for our reward structures, as that would require using MCTS-like simulation techniques.

2 PRELIMINARIES

A *probability distribution* on a countable set S is a function $d : S \to [0, 1]$ such that $\sum_{s \in S} d(s) = 1$. We denote the set of all probability distributions on set S by $\mathcal{D}(S)$. The support of a distribution $d \in \mathcal{D}(S)$ is $\mathsf{Supp}(d) = \{s \in S \mid d(s) > 0\}$.

2.1 Markov chain

Definition 2.1 (Markov chain). A (discrete-time) Markov chain (MC) is a tuple $M = (S, P, AP, L)$, where S is a countable set of states, P is a mapping from S to $\mathcal{D}(S)$, AP is a finite set of atomic proposition and L is the labelling function from S to 2^{AP}.

For states $s, s' \in S$, $P(s)(s')$ denotes the probability of moving from state s to state s' in a single transition and we denote this probability $P(s)(s')$ as $P(s, s')$. We say that the atomic proposition a holds in a state s if $a \in L(s)$. For a Markov chain M, a *finite path* $p = s_0 s_1 \dots s_i$ of length $i \geq 0$ is a sequence of $i + 1$ consecutive states such that for all $t \in [0, i-1]$, $s_{t+1} \in \mathsf{Supp}(P(s_t))$. Similarly, An infinite path is an infinite sequence $p = s_0 s_1 s_2 \dots$ of states such that for all $t \in \mathbb{N}$, $s_{t+1} \in \mathsf{Supp}(P(s_t))$. For a finite or infinite path $p = s_0 s_1 \dots$, we denote its $(i+1)^{th}$ state by $p[i] = s_i$. Let $p = s_0 s_1 \dots s_i$ and $p' = s'_0 s'_1 \dots s'_j$ be two paths such that $s_i = s'_0$. Then, $p \cdot p'$ denotes $s_0 s_1 \dots s_i s'_1 \dots s'_j$. For an MC M, the set of all finite paths of length i (resp. infinite paths) is denoted by Paths^i_M (resp. Paths^ω_M). We denote the set of all finite paths in M by Paths_M and the set of finite paths of length at most H by $\mathsf{Paths}^{\leq H}_M$. For $p \in \mathsf{Paths}_M$, let $\mathsf{Paths}^\omega_M(p)$ denote the set of paths p' in Paths^ω_M such that there exists $p'' \in \mathsf{Paths}^\omega_M$ with $p' = p \cdot p''$. $\mathsf{Paths}^\omega_M(p)$ is called the cylinder set of p.

The σ-algebra associated with the MC M is the smallest σ-algebra that contains the cylinder sets $\mathsf{Paths}^\omega_M(p)$ for all $p \in \mathsf{Paths}_M$. For a state s in S, a measure is defined for the cylinder sets as

$$\mathbb{P}_{M,s}(\mathsf{Paths}^\omega_M(s_0 s_1 \dots s_i)) = \begin{cases} \prod_{t=0}^{i-1} P(s_t)(s_{t+1}) & \text{if } s_0 = s \\ 0 & \text{otherwise.} \end{cases}$$

We also have $\mathbb{P}_{M,s}(\mathsf{Paths}^\omega_M(s)) = 1$ and $\mathbb{P}_{M,s}(\mathsf{Paths}^\omega_M(s')) = 0$ for $s' \neq s$. Using Carathéodory's extension theorem [2, section 1.3.10], this can be extended to a unique probability measure $\mathbb{P}_{M,s}$ on the aforementioned σ-algebra. In particular, if $C \subseteq \mathsf{Paths}_M$ is a set of finite paths forming pairwise disjoint cylinder sets, then $\mathbb{P}_{M,s}(\cup_{p \in C} \mathsf{Paths}^\omega_M(p)) = \sum_{p \in C} \mathbb{P}_{M,s}(\mathsf{Paths}^\omega_M(p))$. Moreover, if $\Pi \in \mathsf{Paths}^\omega_M$ is the complement of a measurable set Π', then $\mathbb{P}_{M,s}(\Pi) = 1 - \mathbb{P}_{M,s}(\Pi')$.

2.2 Probabilistic computation tree logic

Probabilistic computation tree logic or *PCTL* is a branching temporal logic which formulates conditions on a Markov chain. PCTL state

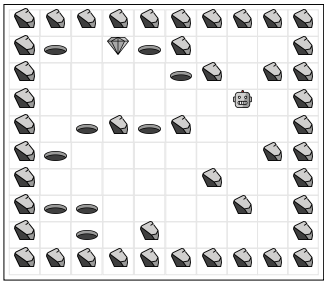

Figure 1: A 10×10 layout for Frozen-Lake

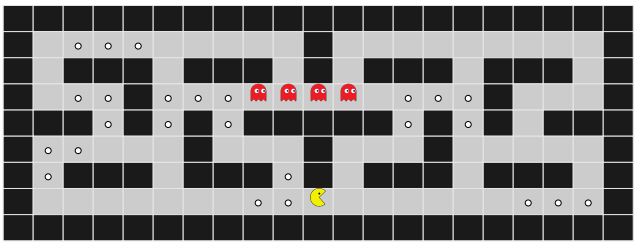

Figure 2: A 21×9 layout for Pac-Man

formulae over a set of atomic propositions AP are defined according the following grammar:

$$\Phi := true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathbb{P}_J(\varphi)$$

where $a \in AP$, Φ_1 and Φ_2 are state formulae, φ is a path formula and $J \subseteq [0,1]$ is an interval with rational bounds. PCTL path formulae are defined according the following grammar:

$$\varphi := \bigcirc\Phi \mid \Phi_1 \mathcal{U} \Phi_2 \mid \Phi_1 \mathcal{U}^{\leq n} \Phi_2$$

where Φ_1 and Φ_2 are state formulae and $n \in \mathbb{N}$.

The satisfaction relation $\models$ between an infinite path $p = s_0 s_1 \ldots$ and a PCTL path formula is defined as follows:

- $p \models \bigcirc\Phi$ if $p[1] \models \Phi$.
- $p \models \Phi_1 \mathcal{U} \Phi_2$ if $\exists i \in \mathbb{N}$ s.t. $s_i \models \Phi_2$ and $\forall j < i$, $p[j] \models \Phi_1$.
- $p \models \Phi_1 \mathcal{U}^{\leq n} \Phi_2$ if $\exists i \leq n$ s.t. $s_i \models \Phi_2$ and $\forall j < i$, $p[j] \models \Phi_1$.

We define the probability of a path formula φ holding at $s \in S$ by

$$\mathbb{P}_M(s \models \varphi) = \mathbb{P}_{M,s}(\{p \in \mathrm{Paths}^{\omega}_M(s) \mid p \models \varphi\})$$

The satisfaction relation $\models$ between a state $s \in S$ and a PCTL state formula is defined inductively: $s \models true$, $s \models a$ if $a \in L(s)$, and

- $s \models \Phi_1 \wedge \Phi_2$ if $s \models \Phi_1$ and $s \models \Phi_2$.
- $s \models \neg\Phi$ if $s \not\models \Phi$.
- $s \models \mathbb{P}_J(\varphi)$ if $\mathbb{P}_M(s \models \varphi) \in J$.

Using the Boolean connectives $\wedge$ and $\neg$, we can define other Boolean connectives such as $\vee$, $\rightarrow$, $\leftrightarrow$. The $\mathcal{U}$ operator (and its bounded version) also allows us to define other useful operators such as $\Diamond$ that expresses reachablility or $\Box$ that expresses safety:

$$\Diamond\Phi = true\, \mathcal{U} \Phi \quad \text{and} \quad \Box\Phi = \neg\Diamond\neg\Phi$$
$$\Diamond^{\leq n}\Phi = true\, \mathcal{U}^{\leq n} \Phi \quad \text{and} \quad \Box^{\leq n}\Phi = \neg\Diamond^{\leq n}\neg\Phi$$

2.3 Markov decision process

Definition 2.2 (Markov decision process). A Markov decision process (MDP) is a tuple $M = (S, A, P, R, R_T, AP, L)$, where S and A are finite sets of states and actions, respectively, A is a finite set of actions, P is a mapping from $S \times A$ to $\mathcal{D}(S)$, R is a mapping from $S \times A$ to $\mathbb{R}$, R_T is a mapping from S to $\mathbb{R}$, AP is a finite set of atomic proposition and L is the labelling function from S to 2^{AP}.

$P(s,a)(s')$ denotes the probability that action a in state s leads to state s' and we denote this probability $P(s,a)(s')$ as $P(s,a,s')$. $R(s,a)$ defines the reward obtained for taking action a from state s and R_T assigns a terminal reward to each state in S.

Example 2.3 (Frozen Lake). We can represent the game Frozen Lake [15] as an MDP. In this game, a robot moves in a slippery grid. It has to reach the target while avoiding holes in the grid. Each state in the MDP represents the current position of the robot in the grid. The states representing the target and the holes can be assumed to be sink states, i.e., the robot cannot move to any other positions from this state. Part of the grid contains walls and the robot cannot move into it. The frozen surface of the lake being slippery, when the robot tries to move by picking a cardinal direction, the next state is determined randomly over the four neighbouring positions of the robot, according to the following distribution weights: the intended direction gets a weight of 10, and other directions that are not a wall and not the reverse direction of the intended one get a weight of 1, the distribution is then normalized so that weights sum up to 1. There are no rewards, and the terminal reward is 1 when the robot reaches the target and 0 otherwise.

*Example 2.4 (*Pac-Man*).* We can represent the multiagent game Pac-Man as a Markov decision process. In this game Pac-Man has to eat food pills in an enclosed grid as fast as possible while avoiding the ghosts. The agents (Pac-Man and the ghosts) can travel in the four cardinal directions unless they are blocked by the walls in the grid. Moreover, the ghosts cannot reverse their direction of travel, and are moving uniformly at random among the directions that are left. In the MDP, the states encode a position for each agent[1] and for the food pills in the grid, while the actions encode individual Pac-Man moves, and while the next state is chosen according to the probabilistic models of the ghosts. The reward decreases by 1 at each step, and increases by 10 whenever Pac-Man eats a food pill. A win (when the Pac-Man eats all the food pills in the grid) increases the reward by 500. Similarly, a loss (when the Pac-Man makes contact with a ghost), decreases the reward by 500.

The definitions and notations used for paths in Markov chain can be extended to the case of MDPs. In an MDP, a *path* is a sequence of states and actions. For a Markov decision process M, a (probabilistic) *policy* is a function $\sigma : \mathrm{Paths}_M \rightarrow \mathcal{D}(A)$ that maps a path p to a probability distribution in $\mathcal{D}(A)$. A policy σ is *deterministic* if the support of the probability distributions $\sigma(p)$ has size 1. A policy σ is *memoryless* if $\sigma(p)$ depends only on $\mathrm{last}(p)$, i.e. if σ satisfies that for all $p, p' \in \mathrm{Paths}_M$, $\mathrm{last}(p) = \mathrm{last}(p') \Rightarrow \sigma(p) = \sigma(p')$.

An MDP M and a policy σ define an MC M_σ. Intuitively, this is obtained by unfolding M, using the policy σ and the probabilities in M

[1]The last action played by ghosts should be stored as well, as they are not able to reverse their direction.

to define the transition probabilities and ignoring the rewards. Formally $M_\sigma = (\mathsf{Paths}_M, P_\sigma, AP, L_\sigma)$ where for all paths $p \in \mathsf{Paths}_M$, $P_\sigma(p)(p \cdot as) = \sigma(p)(a) \cdot P(\mathsf{last}(p), a)(s)$ and $L_\sigma(p) = L(\mathsf{last}(p))$. Thus a finite path p in $\mathsf{Paths}_M(\sigma)$ uniquely *matches* a finite path p' in M_σ when $\mathsf{last}(p') = p$. This way when a policy σ and a state s is fixed, the probability measure $\mathbb{P}_{M_\sigma,s}$ defined in M_σ is also extended for paths in $\mathsf{Paths}_M(\sigma)$. For ease of notation, we write $\mathbb{P}_{M_\sigma,s}$ as $\mathbb{P}^\sigma_s$. We write the expected value of a random variable X with respect to the probability distribution $\mathbb{P}^\sigma_s$ as $\mathbb{E}^\sigma_s(X)$.

Our goal is to maximize the expected rewards obtained by a policy. Classically, this can mean maximizing the sum of rewards up to a finite horizon, or maximizing infinite-horizon metrics such as average reward or discounted sum. In our experiments on Frozen Lake and Pac-Man, we optimize for the total reward objective after fixing a horizon at which the game ends in a draw.

Definition 2.5 (Total reward). The total reward of horizon h for a path $p = s_0 a_0 \ldots$ in M is defined as $\mathsf{Reward}^h_M(p) = \sum_{i=0}^{h-1} R(s_i, a_i) + R_T(s_h)$. The *expected total reward* of a policy σ in an MDP M, starting from state s and for a finite horizon $h \in \mathbb{N}$, is defined as

$$\mathsf{Val}^h_M(s, \sigma) = \mathbb{E}^\sigma_s\left[\mathsf{Reward}^h_M\right].$$

The optimal expected total reward of horizon h, starting from s, over all policies σ in the MDP M is $\mathsf{Val}^h_M(s) = \sup_\sigma \mathsf{Val}^h_M(s, \sigma)$.

One can show that there is a deterministic policy that achieves this supremum [25, Theorem 4.4.1.b]. Such optimal policies may not be memoryless, as one can change their behaviour as the horizon h approaches for example. As the choice of h is arbitrary, we would like to find policies that achieve a good expected total reward independently of h (*i.e.* for every h that is big enough). We will focus our search on (randomized) memoryless policies as a result.[2]

3 EXPERT POLICIES

We describe different policies computed using a combination of formal methods, heuristic search algorithms and machine learning that all aim for the optimal expected total reward.

3.1 Formal methods

Model checking. Exact methods can be used to compute a policy that reaches the optimal expected total reward, *e.g.* with dynamic programming (value iteration) [25, Section 4.5]. They have been efficiently implemented in probabilistic model-checkers such as Storm [16], that offer support for a large range of specifications. More specifically, given a model (Markov chain or MDP), a reward structure and a specification such as a PCTL formula as defined in Section 2.2, Storm can determine whether the input model conforms to the specification and compute expected rewards for a range of finite or infinite horizon metrics such as total or average reward. For MDPs, probabilistic model-checkers can also output an optimal policy associated with the optimal expected reward that they compute. Such tools have been designed with performance in mind and can typically handle models of size up to 10^8 states. Exact methods are thus applicable for smaller MDPs such as the MDP obtained for Frozen Lake in Example 2.3, but not for larger models such as Pac-Man (the MDP represented in Figure 2 already has approximately 10^{16} states). For larger MDPs, formal methods offer alternative techniques *e.g.* based on sampling.

Symbolic MDP. The MDP can be described symbolically in the PRISM [23] language, a guarded command language where one only needs to specify abstract rules that the transitions must satisfy.

Statistical model checking. Computations can make use of statistical model-checking techniques to find good approximations of the expected reward of a policy. By relying on running simulations and computing statistics, it offers confidence guarantees on the quality of the approximated expected reward. We also use the Storm model-checker in this context, as it is capable of producing simulated paths efficiently for an MDP in PRISM format.

Scalability. In both cases, one can scale to larger models by focusing on smaller horizons or sub-objectives for which the MDP can be abstracted further. The idea is that for simple parts of the specification, the relevant aspects of the model may define a much smaller MDP. For example, if we focus on a safety objective in Pac-Man (not being eaten is a necessity in order to get a good reward), we can ignore the status of the food pellets, *i.e.* which food has already been eaten, reducing the state-space from a size of 10^{16} to 10^7. Overall, whenever exact methods become too expensive we will rely on heuristic approaches based on a combination of fixed-horizon and sampling-based state-space exploration techniques.

3.2 Monte Carlo tree search

We consider online procedures where the controller, upon visiting a new state s, computes what action a it thinks is best, and plays it. Then, the state evolves stochastically to a new state s' according to the distribution $P(s, a)$. This is known as *decision-time planning* [28, Chapter 8.8]. Specifically, we rely on the *receding horizon control* approach, where the controller fixes a small horizon H and finds an action that optimizes the expected total reward of horizon H. This approach is meant to select decisions with good short-term consequences, while a well-chosen terminal reward function can be used to predict long-term behaviors from there.

Given an initial state s, *Monte Carlo tree search* or MCTS algorithm [6] is a popular policy that incrementally constructs a search tree rooted at s describing paths of the MDP. This process goes on until a specified budget (of number of iterations or time) is exhausted. An iteration constructs a path by following a decision policy to *select* a sequence of nodes in the search tree. When a node that is not part of the current search tree is reached, the tree is expanded with this new node, whose expected reward is approximated by *simulation*. This value is then used to update the knowledge of all selected nodes in *backpropagation*. Thus, we get a value estimation $\mathsf{approxValue}(s, a)$ for all actions a from the state s. Then the controller takes the action maximizing $\mathsf{approxValue}(s, a)$.

3.3 Monte Carlo tree search with advice

MCTS can be augmented with *symbolic advice* [8] which prune a part of the search tree according to formal specifications meant to differentiate the "good" and "bad" parts of the tree.

A qualitative approach considers a logical formula that the "good" paths need to satisfy. For example, consider the set of states labelled

[2] Note that in some situations, randomization can help a memoryless policy emulate the behaviour of a non-memoryless policy [11].

with *loss* where Pac-Man gets eaten by a ghost. Since reaching such a state is heavily penalized, a simple advice would be to avoid such states. Given a horizon H and a state $s \in S$, the search would be restricted in order to satisfy the path formula $\varphi^H = \Box^{\leq H}(\neg loss)$ that encodes that safety constraint.

A more quantitative approach would compute for each action a and over all policies the best probability $\eta_H(s,a)$ to satisfy φ^H when the action a is taken from s: $\eta_H(s,a) = \sup_{\sigma:\sigma(s)=a} \mathbb{P}^\sigma_s(s \models \varphi^H)$. Then, the advice restricts Pac-Man to almost-optimal actions, *i.e.* decisions a where $\eta_H(s,a) \geq t \times \max_{a'} \eta_H(s,a')$, where t is a threshold in $[0,1]$. Probabilistic model-checkers such as Storm can accept logical specifications in PCTL and compute the probability of path formulæ φ. This approach is similar to probabilistic shielding [21] where bad actions are pre-calculated and used to safely explore the search space during reinforcement learning. A notable difference is that building such a shield requires one to construct the entire state-space of the MDP, whereas our approach performs its computations on-the-fly based on the current position alone. Note that these computations are frequently performed on smaller models, for example in Pac-Man we only need to consider a safety-relevant variant of the MDP where food pellets are ignored and that is restricted to states at distance at most H from the current state. The practical interest of such advice for MCTS is detailed in [8].

4 IMITATING EXPERT POLICIES

In order to reach on-the-fly computing times low enough for real-time control, we train policies, encoded as neural networks, to imitate an expert policy. This can take different forms depending on the expert policy σ. In general, we define a function $f_\sigma : S \times A \to \mathbb{R}$ encoding the policy σ so that from state s, the decision made by σ is equivalent to choosing an action from $\arg\max_{a \in A}(f_\sigma(s,a))$ uniformly at random. Intuitively, f_σ is a scoring function that rates how good every action is from the current state. To learn a memoryless policy σ, this function can output the expected total reward under σ, or a heuristic score approximating it as returned by MCTS for example. This framework can also be used to learn quantitative advice, e.g. by using $\eta_H(s,a)$ as a scoring function. In this case, the advice is seen as a (non-deterministic) expert policy to be imitated. This way, we suggest that a symbolic advice can also be imitated by a neural network that can then be used as a *neural advice* in MCTS.

The plan is to teach a neural network the function f_σ as an offline step, and use it to speed up the computations of decision-time planning. Depending on the set of actions, we can either train a neural network that takes a state-action pair (s,a) and outputs a single value $f_\sigma(s,a)$ or a neural network that takes a state s and outputs a vector in $\mathbb{R}^{|A|}$ with values for each available actions.

We address the following challenges: encoding a state s and its corresponding values $(f_\sigma(s,a))_{a \in A}$ so that it is easily processable by the neural network, generating data for f_σ representative of the state-space, choosing an architecture for the neural network, comparing the learnt policy and the expert policy σ.

4.1 Training a neural network

We divide our datasets in $5:2:3$ ratios to create distinct datasets for training, validation and testing of the neural networks. We propose the use of convolutional neural networks which would take a state in the MDP as a *tensor* with each channel of the tensor representing different features extracted from the state. In Pac-Man, each tensor representing a state has 7 channels to denote respectively the distribution of walls, food pills, position of Pac-Man, and for each direction, positions of the ghosts who are moving towards that direction. For example the channel representing the distribution of walls would be a matrix w_{ij} of the size of the grid where $w_{ij} = 1$ if there is a wall at the co-ordinate (i,j), and otherwise $w_{ij} = 0$.

We considered different approaches for normalization, either by globally scaling the values between 0 and 1 so that $\min_{s,a} f(s,a)$ becomes 0 and $\max_{s,a} f(s,a)$ becomes 1 after normalization, or scaling locally so that for all state s, $\min_a f(s,a)$ becomes 0 and $\max_a f(s,a)$ becomes 1. We argue that this local normalization is sufficient to learn the policy as it captures the ordering of the actions. Experimentally, local normalization performed better than global normalization. We also experimented with non-linear transformations [5, 29] but they did not improve learning performances in our settings. Our neural networks contain a 2D convolution layer with 3×3 filters, a flattening layer, a few dense layers with the ReLU activation function and a final dense layer with the sigmoid activation function. Training is performed using ADAM optimizer [22] with mean squared error as loss function. To choose the optimal hyperparameters, *e.g.* the exact number of layers and their size or the number of filters, we use *hyperparameter tuning* [4] in each setting. In particular, we relied on the Python library KerasTuner [24].

4.2 Formally sharp DAgger

Let us detail how to construct a set of data of the shape (x,y), where $x \in S$ is the input of the neural network and $y \in \mathbb{R}^{|A|}$ is its output encoding $(f_\sigma(x,a))_{a \in A}$. We argue for the use of formal methods in order to answer: how to get a representative set of input values x, and how to get good y values for this set of input.

Perfect data. Note that an expert policy generated by an exact method is ensured an expected payoff higher than any expert policy generated from a heuristic approach. In a sense, if one sees a heuristic approach as an approximation of the optimal policy, the data obtained from heuristic policies can be seen as a noisy version of data that would otherwise be "perfect", *i.e.* pairs (x,y) where y is a vector encoding the decisions of a policy σ that is optimal.

Representative set of inputs. In order to generate a dataset to train on, a classical method is to pick states and actions uniformly at random within the state-space and to evaluate f_σ on these inputs. For example, one can consider Frozen Lake states obtained by placing the walls, the holes, the target and the robot at random empty positions. However, a neural network trained from such a dataset may perform poorly for states that play a key role in the expected payoff of a policy (*i.e.* states that represent crucial decisions), as such states may not be likely to be selected at random within the state-space. The DAgger (Dataset Aggregation) algorithm, in contrast, offers a dataset generation method based on running simulations in order to get a more realistic view of the states frequently encountered in real plays. While this approach can be part of the answer, it may not provide sufficiently many datapoints on the crucial decisions mentioned before, that may be few and far-between.

Algorithm 1: Sharp Dataset Aggregation (Sharp DAgger)

Input: A function $f_\sigma : S \to \mathbb{R}^{|A|}$ encoding an expert policy σ, $s_0 \in S$, a metric d, $\epsilon \in \mathbb{R}$, a horizon $h \in \mathbb{N}$.
Output: A policy σ_i that imitates the policy σ

```
DATASET = initial dataset;
NN_0 = neural network trained using DATASET;
σ_0 = policy extracted from NN_0;
for 0 ≤ i ≤ iters do
    Paths_i = paths sim'd following σ_i from s_0 for h steps;
    for state s in paths p ∈ Paths_i do
        if d(NN_i(s), f(s)) ≥ ε then
            Add (s, f(s)) to DATASET;
    NN_i = neural network trained using DATASET;
    σ_i = policy extracted from NN_i;
return σ_i;
```

We propose an algorithm named *sharp DAgger* that would detect these states, refine the training set and retrain the network. This is done by simulating the policy using the learnt neural network on the MDP and finding *counter-examples* where the neural network is performing poorly by comparing the value given by the network and the value $f_\sigma(s)$ associated with the exact method.

In Algorithm 1, we present a method to train the neural network by an iterative process that generates new data for the training set. In the first iteration, we train a neural network NN_0 from an initial training dataset DATASET and in later iterations, we add more interesting data-points in that set. Initially, one could either randomly generate a small amount of data or simulate the MDP by following a uniform policy. In iteration i, starting from an initial state s_0 in the MDP, we simulate a fixed number of paths until a given horizon H. We extract from these paths the states for which the current neural network NN_i trained from DATASET fails to predict the correct values. We add them to our dataset, then train the next iteration of the neural network. The decision on when to stop the sharp DAgger loop is taken based on evaluations of the quality of the neural network NN_i at each iteration i.

4.3 Evaluating a learnt policy

In order to evaluate the trained neural network, a traditional approach for machine learning can report on a loss function for a test dataset. Alternatively, one can measure the accuracy of the network by reporting how many times the resulting learnt policy has differed from the expert policy as a classifier. But this may not be sufficient to evaluate how the learnt policy is performing on the MDP. In Frozen Lake, consider a learned policy that returns the same action as the expert policy for all states in the MDP, except for one state where the learnt policy gives a bad action that leads to a hole. Even though the learnt policy has an almost perfect accuracy, it would perform badly compared to the expert policy in real plays, and could lead to much worse rewards on expectation.

As such, we argue for the use of *statistical model checking* to evaluate the expected reward of a (neural) policy. In particular, we can use the approximate probabilistic model checking method [17] where we simulate a set of paths following the expert policy on the one hand and the neural policy on the other, then compare their average rewards on these paths.

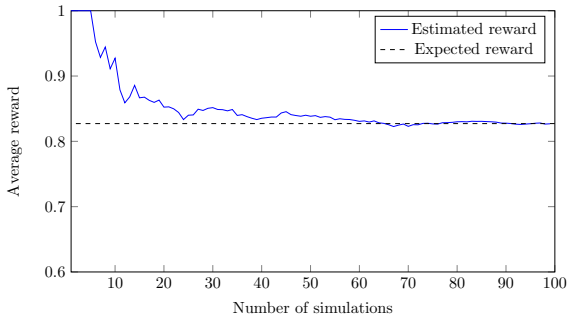

Figure 3: Statistical model checking for Frozen Lake

Theorem 4.1. *Suppose for MDP M, there exists $a < b$ such that $a \le \mathsf{Reward}^h_M(p) \le b$ for all paths p in M. Let $\delta \in (0,1]$ and $\epsilon \in (0, b-a]$. Then for a policy σ, suppose we sample $n \ge \frac{(b-a)^2}{2\epsilon^2} \ln(\frac{2}{\delta})$ paths $p_1, p_2 \dots p_n$ independently at random from a state s in the MDP M following the policy σ. Let $\overline{r} = \frac{1}{n}\sum_{i=1}^{n} \mathsf{Reward}^h_M(p_i)$. Then,*

$$\mathbb{P}^\sigma_s(|\overline{r} - \mathsf{Val}^h_M(s,\sigma)| \ge \epsilon) \le \delta\,.$$

Proof. We have n independent identically distributed random variables $\mathsf{Reward}^h_M(p_i)$ with expected value $\mathsf{Val}^h_M(s,\sigma)$. Then, $\mathbb{E}^\sigma_s(\overline{r}) = \mathsf{Val}^h_M(s,\sigma)$. The Chernoff-Hoeffding inequality [19] then yields $\mathbb{P}^\sigma_s(|\overline{r} - \mathsf{Val}^h_M(s,\sigma)| \ge \epsilon) \le 2\exp\left(-\frac{2n\epsilon^2}{(b-a)^2}\right) \le \delta\,.$ □

The above theorem gives a theoretical bound on the number of simulations needed to get a *probably approximately correct* approximation of the real expected reward. In practice, we typically need fewer simulations to achieve a good approximation. For example, consider the Frozen Lake layout in Figure 1. Using exact methods we calculated the optimal expected reward to be 0.827. In Figure 3, for $n \in [1, 100]$, we independently simulated n paths using the optimal policy and plotted the estimated reward obtained from statistical model checking. We see that we get a good approximation of the real expected reward with under 100 simulations.

5 EXPERIMENTAL RESULTS

We ran experiments on the two MDPs previously introduced in Section 2.3. Frozen Lake is an MDP that can be fully handled by model-checkers (using exact methods), and as such we use it to report on the benefits of using perfect data to train the surrogate policy. Whereas, the Pac-Man game provides more challenging MDPs to handle. There, we report on the performance of MCTS equipped with perfect or neural advice and on the performance of a surrogate policy trained on data obtained from MCTS. The sharp DAgger algorithm (Algorithm 1) proves to be instrumental for learning efficiently in Pac-Man. The code is available at [9].

5.1 Frozen Lake

For the game described in Example 2.3, we randomly generated layouts of size 10x10 where we place walls at each cell in the border of the grid and with probability 0.1 at each of the other cells.

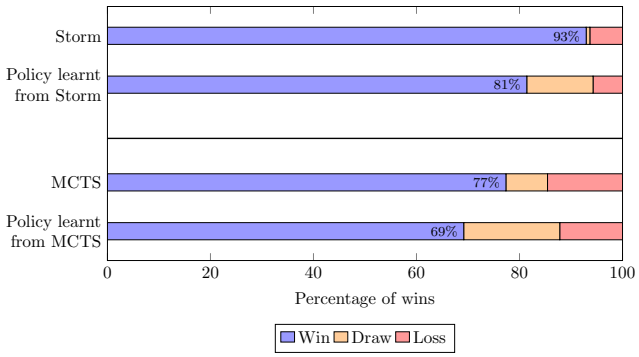

Figure 4: Perfect vs MCTS-based policies for Frozen Lake.

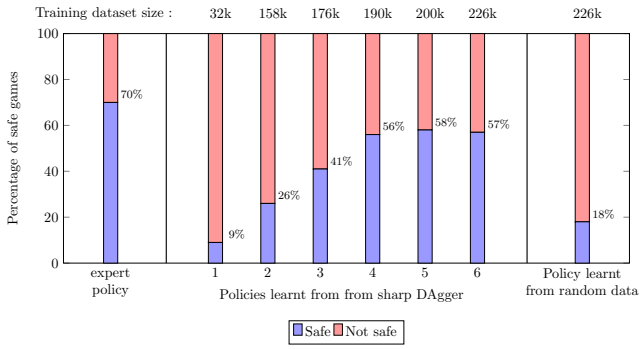

Figure 5: Sharp DAgger for Pac-Man neural advice

Then we place holes in remaining cells with probability 0.1. Finally, we randomly place a target and an initial position in two of the remaining empty cells. If the game is neither won nor lost within 1000 steps, the game is considered a draw.

5.1.1 Expert policies. Consider the state in the MDP where the robot is on the target position. We label this state with *target*. Using the model checker, we can compute the policies $\text{Opt}(s) = \arg\max_\sigma \mathbb{P}_s^\sigma(s \models \Diamond target)$ that maximize the probability to reach the target t starting from state s. The practical policy that we are interested in should not only maximize the probability to reach the target but also minimize the expected number of steps needed to reach the target (in order to reach it before the horizon H whenever possible). For a path ρ in MC M_σ, we define $\text{len}(\rho, target) = i$ if $\rho[i]$ is the target state and for all $j < i$, $\rho[j]$ is not the target set. Using formal methods techniques, we can calculate a policy in

$$\arg\min_{\sigma \in \text{Opt}(s)} \mathbb{E}_s^\sigma(\text{len}(\rho, target) \mid \rho \models \Diamond target)) .$$

This policy can be shown to be optimal for total reward of any large enough horizon H. We compared it with the policy generated from MCTS with horizon $H = 30$. From state s, a search tree is constructed for 40 iterations. Thus, the search tree constructed by the MCTS algorithm contains up to 40 nodes. In each iteration, when a new node is added to the search tree, 10 samples are obtained by using a uniform policy to estimate the value of the node.

5.1.2 Learnt policies. Our training dataset contained 760k data-points which we used to imitate the expert policies. Hyperparameter tuning resulted in neural networks containing a 2D convolution layer with 6 filters, a flattening layer and 2 dense layers. We randomly generated 1000 layouts and ran 100 games from each layout for 1000 steps using both expert policies and the learnt policies. The average outcomes are reported in Figure 4. Using Storm, we calculate the optimal expected win rate to be 93% on average in the generated layouts. This value denotes the probability to reach the target eventually, using the optimal policy. In practice, our statistical model checking approach requires fixing a finite horizon. Figure 4 confirms that horizon 1000 is sufficient as the expert policy from Storm still reaches a win rate of 93%. In comparison, our policy learnt from Storm had a win rate of 81%. The expert policy calculated using MCTS is suboptimal and showed a win rate of 77% while the policy learnt from it has a win rate of 69%. This highlights the benefits of using exact methods to get noise-free data.

5.2 Pac-Man

We performed our experiments on the game Pac-Man in a grid of size 9×21 described in Figure 2. In our experiments, the ghosts always choose an action uniformly at random from the legal actions available. As explained in Example 2.4, we can view this as an MDP. Moreover, if Pac-Man does not win (eats all food pills) or lose (makes contact with a ghost) within 300 steps, we consider it a draw.

5.2.1 Expert policies. The state-space of the MDP is too large to apply directly to find the optimal policy. As a consequence, we decided to use Monte Carlo tree search with a receding horizon of $H = 10$. From state s, a search tree is constructed with a maximum depth of H for 40 iterations.[3] We combined MCTS with the notion of advice as used in [8] in order to play Pac-Man. In each iteration of the MCTS algorithm, when a new node is added, 20 samples are obtained by using a uniform policy to estimate the value of the node among the paths that are safe i.e. where Pac-Man is not eaten by a ghost. This optimistic estimation matches the notion of *simulation advice* of [8]. During the exploration of the search tree, we also restrict ourselves to actions a that maximize the probability to stay safe for the next 8 steps, *i.e.*, actions a such that $\eta_8(s, a) = \max_{a' \in A} \eta_8(s, a')$ as defined in Section 3.3. Since the online computation of the η_8 function is too expensive to be done at every node of the search tree, we only restrict the root node of the tree so as to ensure the safety of the immediate decisions

We compare four different variants of MCTS in Figure 6: a version without this expert (safety) advice, one where it is used at the root node of the tree, one where a neural advice is trained to imitate the safety advice and is used at the root node, and finally one where the neural advice is used at every node in the tree. For reference, [8] reports that human players win 44% of the time on this grid.

5.2.2 Neural advice. To speed up the MCTS procedure we train a neural network to imitate the safety advice. We used Algorithm 1 to create a dataset. We use the L_∞ *metric* with precision value $\epsilon = 0.2$ to find new data-points during the aggregation. In other words, we

[3] 40 iterations was selected experimentally as a good compromise between achieving high expected rewards and minimising computation time.

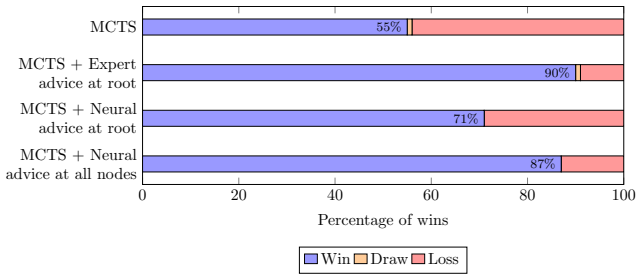

Figure 6: Different MCTS variants for Pac-Man.

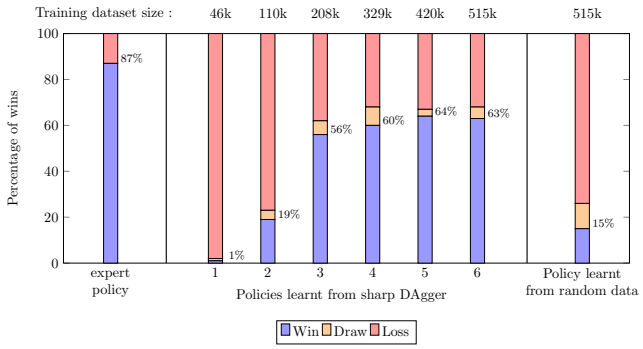

Figure 7: Sharp DAgger for Pac-Man surrogate policies.

add $(s, (\eta_H(s,a))_{a\in A})$ to the dataset at the i^{th} iteration of sharp DAgger if $\max_{a\in A}(|\eta_H(s,a) - \mathsf{NN}_i(s,a)|) > 0.2$.

In each iteration, we simulate 4000 games for 300 steps to generate Paths_i. We compare the safety status of the neural networks at each iteration of sharp DAgger in Figure 5. After 5 iterations, we observe that Pac-Man stays safe (for 300 steps) in 58% of games when using the learnt policy instead of staying safe in 70% of games with the policy calculated from model checking. Hyperparameter tuning stabilized on neural networks using a 2D convolution layer with 6 filters, a flattening layer and 4 dense layers. The entire training dataset generated from sharp DAgger contains $226k$ data-points. To check the effectiveness of our method of data aggregation, we compare our learnt policy with a policy trained on $226k$ randomly generated data-points. This learnt policy performs worse and stays safe in only 18% of games. Dataset generation and training of the neural networks was performed in 36 hours with a cluster of 250 CPU cores, for a total of 9000 hours of computing time (at 2.9 GHz).

5.2.3 Using the neural advice in MCTS. To accommodate for the inherent noise in the output of the neural network NN, we fix a threshold $t = 0.9$ and consider the advice that allows almost-optimal actions with respect to t, *i.e.* the neural advice that restricts to actions a such that $\mathsf{NN}(s,a) \geq 0.9 \times \max_{a'\in A} NN(s,a')$.

We compare in Figure 6 the performance of MCTS variants using expert or neural policies as advice. We ran each setup on 100 games. The Python implementation of MCTS that we rely on was not designed to optimize the performance in terms of computing time. In our case, the MCTS algorithm without any selection advice uses 9 seconds to decide on an action. Using the (formal methods based) expert advice at the root node of MCTS increases the time per decision by 8 extra seconds. While the 9 seconds spent in MCTS can be expected to be vastly lowered using code improvements,[4] the model checking done by Storm is already optimized. By replacing the expert advice with a neural advice, we can avoid this fixed cost of 8 seconds per decision, as the network can be consulted in 3 ms instead. While the neural advice is not as good as the expert advice (it ensures safety in 71% of games instead of 90% when used identically at the root node of the MCTS tree), we can afford to use it on every node of the search tree to dynamically prune the search. In this way, we can get an 87% win-rate that is the best of both worlds: we approach the win-rate of the expert advice with the computing time of the bare-bones version of MCTS. Since the neural advice requires much less computational power per call than the expert advice, using it would compensate the expensive computational cost of its training in the long run. In our case, we break even after 4 million calls (roughly 40k games of Pac-Man).

5.2.4 Learning a surrogate policy. We trained a surrogate neural network to imitate the expert policy defined previously as MCTS with a neural advice at every node, that reached an 87% win-rate while keeping computing times as low as possible. To generate the dataset, we use our sharp DAgger algorithm and simulate 4000 games with horizon 300 in each iteration. To evaluate how well our policies are performing, we compare the average number of wins obtained by following them in Figure 7. After 5 iterations, we reach a policy with a win-rate of 64%, which is higher than the 55% of the "standard" version of MCTS, while having almost no need for online computing time as it is using a pre-trained neural network. Hyperparameter tuning stabilized on neural networks using a 2D convolution layer with 5 filters, a flattening layer, 5 dense layers. Finally, the training dataset generated with sharp DAgger contains $515k$ data-points. In comparison, a policy learned from a randomly generated dataset of size $515k$ is only able to win in 15% of games, which confirms the importance of sharp DAgger in this setting.

6 CONCLUSION

In this work, we have proposed a framework to combine formal methods with MCTS and deep learning to obtain a scalable way of synthesising policies with both good performance and low latency.

From our experiments, we conclude that formal methods can provide good policies and useful advice for MCTS, albeit at a high computational cost. Training a neural network to play the role of the advice allows one to obtain the best of both worlds: the performance boost of the advice but without its computational cost. Particularly, neural advice compensates for its expensive computational training cost in the long run since it requires less computational power per call than expert advice. Using a sharp dataset-aggregation procedure is instrumental in reaching satisfactory rewards in practice because of the reliance of deep-learning techniques on the accumulation of huge amounts of data. Finally, while the best policy that we obtained for Pac-Man is based on MCTS, its surrogate neural-network policy is able to play relatively well while making near instantaneous decisions.

[4] MCTS and other simulation-based techniques are highly amenable to parallelism [10].

ACKNOWLEDGMENTS

Computational resources have been provided by the CÉCI, funded by the F.R.S.-FNRS under Grant No. 2.5020.11 and by the Walloon Region. This work was supported by the ARC "Non-Zero Sum Game Graphs" project (Fédération Wallonie-Bruxelles), the EOS "Verilearn" project (F.R.S.-FNRS & FWO), and the FWO "SAILor" project (G030020N).

REFERENCES

[1] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. 2018. Safe Reinforcement Learning via Shielding. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, (AAAI 2018)*. AAAI Press, 2669–2678.

[2] Robert B. Ash and Catherine A. Doleans-Dade. 1999. *Probability and Measure Theory* (2nd edition ed.). Harcourt Academic Press.

[3] Pranav Ashok, Tomás Brázdil, Jan Kretínský, and Ondrej Slámecka. 2018. Monte Carlo Tree Search for Verifying Reachability in Markov Decision Processes. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018) (Lecture Notes in Computer Science, Vol. 11245)*. Springer, 322–335. https://doi.org/10.1007/978-3-030-03421-4_21

[4] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, 2 (2012).

[5] George EP Box and David R Cox. 1964. An analysis of transformations. *Journal of the Royal Statistical Society: Series B (Methodological)* 26, 2 (1964), 211–243.

[6] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (2012), 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810

[7] Carlos E. Budde, Arnd Hartmanns, Michaela Klauck, Jan Kretínský, David Parker, Tim Quatmann, Andrea Turrini, and Zhen Zhang. 2020. On Correctness, Precision, and Performance in Quantitative Verification - QComp 2020 Competition Report. In *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 12479)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 216–241. https://doi.org/10.1007/978-3-030-83723-5_15

[8] Damien Busatto-Gaston, Debraj Chakraborty, and Jean-François Raskin. 2020. Monte Carlo Tree Search Guided by Symbolic Advice for MDPs. In *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference) (LIPIcs, Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 40:1–40:24. https://doi.org/10.4230/LIPIcs.CONCUR.2020.40

[9] Debraj Chakraborty, Damien Busatto-Gaston, Jean-François Raskin, and Guillermo A. Pérez. 2023. Formally-Sharp DAgger for MCTS: Lower-Latency Monte Carlo Tree Search using Data Aggregation with Formal Methods. https://doi.org/10.5281/zenodo.7655528

[10] Guillaume M. J. B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. 2008. Parallel Monte-Carlo Tree Search. In *Proceedings of the 6th International Conference on Computers and Games (CG 2008) (Lecture Notes in Computer Science, Vol. 5131)*, H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands (Eds.). Springer, 60–71. https://doi.org/10.1007/978-3-540-87608-3_6

[11] Krishnendu Chatterjee, Luca De Alfaro, and Thomas A Henzinger. 2004. Trading memory for randomness. In *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.* IEEE, 206–217.

[12] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[13] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking*. Springer. https://doi.org/10.1007/978-3-319-10575-8

[14] Arthur Clavière, Souradeep Dutta, and Sriram Sankaranarayanan. 2019. Trajectory Tracking Control for Robotic Vehicles Using Counterexample Guided Training of Neural Networks. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*, J. Benton, Nir Lipovetzky, Eva Onaindia, David E. Smith, and Siddharth Srivastava (Eds.). AAAI Press, 680–688. https://ojs.aaai.org/index.php/ICAPS/article/view/3555

[15] OpenAI Gym. [n.d.]. *Frozen Lake*. https://www.gymlibrary.dev/environments/toy_text/frozen_lake/

[16] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The probabilistic model checker Storm. *Int. J. Softw. Tools Technol. Transf.* 24, 4 (2022), 589–610. https://doi.org/10.1007/s10009-021-00633-z

[17] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. 2004. Approximate probabilistic model checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 73–84.

[18] Michael Hertneck, Johannes Köhler, Sebastian Trimpe, and Frank Allgöwer. 2018. Learning an Approximate Model Predictive Controller With Guarantees. *IEEE Control. Syst. Lett.* 2, 3 (2018), 543–548. https://doi.org/10.1109/LCSYS.2018.2843682

[19] Wassily Hoeffding. 1963. Probability Inequalities for Sums of Bounded Random Variables. *J. Amer. Statist. Assoc.* 58, 301 (1963), 13–30. https://doi.org/10.1080/01621459.1963.10500830

[20] Radoslav Ivanov, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. 2019. Verisig: verifying safety properties of hybrid systems with neural network controllers. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. 169–178.

[21] Nils Jansen, Bettina Könighofer, Sebastian Junges, Alex Serban, and Roderick Bloem. 2020. Safe Reinforcement Learning Using Probabilistic Shields (Invited Paper). In *31st International Conference on Concurrency Theory, CONCUR 2020*, Vol. 171. 3:1–3:16. https://doi.org/10.4230/LIPIcs.CONCUR.2020.3

[22] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

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

[24] Tom O'Malley, Elie Bursztein, James Long, François Chollet, Haifeng Jin, Luca Invernizzi, et al. 2019. KerasTuner. https://github.com/keras-team/keras-tuner.

[25] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley. https://doi.org/10.1002/9780470316887

[26] Stephane Ross and Drew Bagnell. 2010. Efficient Reductions for Imitation Learning. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 9)*, Yee Whye Teh and Mike Titterington (Eds.). PMLR, Chia Laguna Resort, Sardinia, Italy, 661–668. https://proceedings.mlr.press/v9/ross10a.html

[27] Stephane Ross, Geoffrey Gordon, and Drew Bagnell. 2011. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 15)*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.). PMLR, Fort Lauderdale, FL, USA, 627–635.

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

[29] In-Kwon Yeo and Richard A Johnson. 2000. A new family of power transformations to improve normality or symmetry. *Biometrika* 87, 4 (2000), 954–959.

[30] Håkan LS Younes and Reid G Simmons. 2002. Probabilistic verification of discrete event systems using acceptance sampling. In *International Conference on Computer Aided Verification*. Springer, 223–235.