International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-022-00685-9

EXPLANATION PARADIGMS LEVERAGING ALGEBRAIC INTUITION

Special Section: Introducing Explanation Paradigms Leveraging Algebraic Intuition

Analyzing neural network behavior through deep statistical model checking

Timo P. Gros[1] · Holger Hermanns[1] · Jörg Hoffmann[1] · Michaela Klauck[1] · Marcel Steinmetz[1]

Accepted: 18 October 2022

Abstract
Neural networks (NN) are taking over ever more decisions thus far taken by humans, even though verifiable system-level guarantees are far out of reach. Neither is the verification technology available, nor is it even understood what a formal, meaningful, extensible, and scalable testbed might look like for such a technology. The present paper is an attempt to improve on both the above aspects. We present a family of formal models that contain basic features of automated decision-making contexts and which can be extended with further orthogonal features, ultimately encompassing the scope of autonomous driving. Due to the possibility to model random noise in the decision actuation, each model instance induces a Markov decision process (MDP) as verification object. The NN in this context has the duty to actuate (near-optimal) decisions. From the verification perspective, the externally learnt NN serves as a determinizer of the MDP, the result being a Markov chain which as such is amenable to statistical model checking. The combination of an MDP and an NN encoding the action policy is central to what we call "deep statistical model checking" (DSMC). While being a straightforward extension of statistical model checking, it enables to gain deep insight into questions like "how high is the NN-induced safety risk?", "how good is the NN compared to the optimal policy?" (obtained by model checking the MDP), or "does further training improve the NN?". We report on an implementation of DSMC inside the Modest Toolset in combination with externally learnt NNs, demonstrating the potential of DSMC on various instances of the model family, and illustrating its scalability as a function of instance size as well as other factors like the degree of NN training.

Keywords Statistical model checking · Neural networks · Learning · Verification · Scalability

1 Introduction

Neural networks (NN), in particular deep neural networks, promise astounding advances across a manifold of computing applications in domains as diverse as image classification [51], natural language processing [43], and game playing [67]. NNs are the technical core of ever more *intelligent systems*, created to assist or replace humans in decision-making.

This development comes with the urgent need to devise methods to analyze, and ideally verify, desirable behavioral properties of such systems. Unlike for traditional programming methods, this endeavor is hampered by the nature of

Authors are listed alphabetically. This work was partially supported by the ERC Advanced Investigators Grant 695614 (POWVER), by the German Research Foundation (DFG) under Grant No. 389792660, as part of TRR 248 – CPEC, see https://perspicuous-computing.science, by the Key-Area Research and Development Program Grant 2018B010107004 of Guangdong Province, and by the European Regional Development Fund (ERDF).
A preliminary version of this work was published at FORTE [30]. Here, we report full technical details as well as new experiments exploring scalability as a function of instance size and the degree of NN training.

✉ Michaela Klauck
klauck@cs.uni-saarland.de

Timo P. Gros
timopgros@cs.uni-saarland.de

Holger Hermanns
hermanns@cs.uni-saarland.de

Jörg Hoffmann
hoffmann@cs.uni-saarland.de

Marcel Steinmetz
steinmetz@cs.uni-saarland.de

[1] Saarland Informatics Campus, Saarland University, Saarbrücken, Germany

Published online: 13 December 2022

neural networks, whose complex function representation is not suited to human inspection and is highly resistant to mechanical analysis of important properties.

Verification Challenge. As a matter of fact, remarkable progress is being made toward automated NN analysis, be it through specialized reasoning methods of the SAT-modulo-theories family [22,45,49], or through suitable variants of abstract interpretation [21,57] or quantitative analysis [17,70]. All these works thus far focus on the verification of individual NN decision episodes, i.e., the behavior of a single input/output function call. In contrast, the verification of NNs being the decisive (in the literal sense of the word) authorities inside larger systems placed in possibly uncertain contexts is wide-open scientific territory.

Very many real-world examples, where NNs are expected to become central decision entities—from autonomous driving to medical care robotics—involve discrete decision-making in the presence of random phenomena. The former are to be taken in the best possible manner, and it is the NN that decides which decisions to take when and where. A very natural formal model for studying the principles, requirements, efficacy and robustness of such an NN, is the model family of Markov decision processes [64] (MDP). MDPs are a very widely studied class of models in the AI community, as well as in the verification community, where MDPs are the main semantic object of probabilistic model checking [53].

Assume now we are facing a problem for which a NN decision entity has been developed by a different party. If the problem statement can be formally cast as a certain MDP, we may use this MDP as a context to study properties of the NN delivered to us. Concretely, the NN will be put to use as a determinizer of the otherwise nondeterministic choices in the MDP, so that altogether a Markov chain results, which in turn can be evaluated by standard probabilistic model checking techniques. The idea can be further extended by making the technology available to a certification authority responsible for NN system approval, or to the party designing the NN, as a valuable feedback mechanism in the design process.

Deep statistical model checking. However, this style of verification is challenged by the complexity of analyzing the participating NN *and* that of analyzing the induced system behaviors and interactions. Already the latter is a notorious practical impediment to successful verification rooted in state space explosion problems. Indeed, standard probabilistic model checking will suffer quickly from this. However, for Markov chains, there is a scalable alternative to standard model checking at hand, nowadays referred to as *statistical model checking* [42,54,71]. The latter method employs efficient sampling techniques to statistically check the validity of a certain formal property. If applicable, it does not suffer from the state space explosion problem, in contrast to standard probabilistic model checking.

The scalable verification method we proposed in *DSMC20* [30] is called *deep statistical model checking* (DSMC) by us. At its core is a straightforward variation of statistical model checking, applied to an MDP, together with an NN that has to take the decisions. For this, DSMC expects an NN that can be queried as a black-box oracle to resolve the nondeterminism in the MDP given: The NN receives the state descriptor as input, and it returns as output a decision determining the next step. The DSMC method integrates the pair of NN and MDP, and analyzes the resulting Markov chain statistically. In this way, it is possible to statistically verify properties of the NN itself, as we will discuss.

Racetrack. To study the potential of DSMC, we perform practical experiments with a case study family that remotely resembles the autonomous driving challenge, albeit with some drastic restrictions relative to the grand vision. These restrictions are as follows: (i) We consider a single vehicle, there is no traffic otherwise. (ii) No object or position sensing is in use, instead the vehicle is aware of its exact position and speed. (iii) No speed limits or other traffic regulations are in place. (iv) Fuel consumption is not optimized for. (vi) Weather and road conditions are constant. (vii) The entire problem is discretized in a coarse manner. What remains after all these restrictions (apart from inducing a roadmap of further works beyond what we study) is the problem of navigating a vehicle from start to goal on a discrete map, with actions allowing to accelerate/decelerate in discrete directions, subject to a probabilistic risk of action failing to take effect in each step. The objective is to reach the goal in a minimal number of steps without bumping into a boundary wall. This problem is known as the Racetrack, a benchmark originating in AI autonomous decision-making [9,63]. Recently, the benchmark has also been used in multiple model checking and verification contexts [7], where some of the restrictions from above have been relaxed and more features have been added. In formal terms, each map and parameter combination induces an MDP.

Racetrack is a simple problem, simple enough to put a neural network in the driver seat: This NN is then the central authority in the vehicle control loop. It needs to take action decisions with the objective to navigate the vehicle safely toward the goal. There are a good number of scientific proposals on how to construct and train an NN for mastering such tasks, and the present paper is not trying at all to innovate in this respect. Instead, *the central contribution of this paper is a scalable method to verify the effectiveness of an NN trained externally for its task.* This technique, DSMC, is by no means bound to the Racetrack problem domain, instead it is generally applicable. We evaluate it in the context of Racetrack because we do think that this is a crisp formal model family, which is of value in ongoing activities to systematize our understanding of NNs that are supposed to take over important decisions from humans.

Our concrete modelling context is MDPs represented in JANI [14], a language interfacing with the leading probabilistic model checkers out there. For the sake of experimentation and for use by third parties, we have implemented a connection between NNs and the state-of-the-art statistical model checker MODES [10,13], part of the MODEST TOOLSET [38]. This extension gives the possibility to use an NN oracle and to analyze the resulting Markov chain by SMC. We thus establish an initial DSMC tool infrastructure, which we apply on Racetrack benchmarks.

It will become evident by our empirical evaluation that there are a variety of use cases for DSMC, pertaining to end users and domain engineers alike:

– *Quality assurance*. DSMC can be a tool for end users, or engineers, in system approval or certification, regarding safety, robustness, absence of deadlocks, or performance metrics. The generic connection to model checking furthermore enables the comparison of NN oracles to provably optimal choices, on moderate-size models: taking out the NN, the original MDP results, and can be submitted to standard probabilistic model checking. In our implementation, we use MCSTA [38] for this purpose, the probabilistic model checker of the MODEST TOOLSET, based on value iteration.
– *Learning pipeline assessment*. DSMC can serve as a tool for the NN engineers designing the NN learning pipeline in the first place. This is because the DSMC analysis can reveal specific deficiencies in that pipeline. For example, we show that simple heat maps can highlight *where* the oracles are unsafe. And we exhibit cases where NN oracles turn out highly unsafe despite this phenomenon not being derivable from standard measures of learning performance. Such problems would likely have remained undetected without DSMC.

There are already works building up on DSMC giving evidence for the potential impact of the approach. The information delivered by DSMC has already been used to improve reinforcement learning strategies [32] and for the design of policy-analysis tools in synergy with interactive visualization techniques [26,28]. The most important work based on DSMC is MoGym [29], the integrated toolbox enabling the training and verification of machine-learned decision-making agents based on formal models, which bridges the reinforcement learning community to formal methods.

In summary, our contributions are as follows:

1. We present deep statistical model checking, which statistically evaluates the connection of an NN oracle and an MDP formalizing the problem context.
2. We establish tool infrastructure for DSMC within MODES to connect to NN oracles.
3. We establish infrastructure for Racetrack benchmarking, including parsing, simulation, JANI model export, comparison with optimal behavior, and also for NN learning.
4. We illustrate the use and feasibility of DSMC in Racetrack case studies.
5. We demonstrate the scalability of DSMC, depending on multiple dimensions, e.g., model size and number of training episodes, i.e., NN quality, in a huge, time consuming study on scaled Racetrack benchmarks.

The benchmark and all infrastructure including our modification of MODES as well as our JANI model is archived and publicly available at DOI https://doi.org/10.5281/zenodo.3760098 [31] as presented in *DSMC20*. The infrastructure for the scalability study is available at https://doi.org/10.5281/zenodo.7071405 on Zenodo.
The paper is organized as follows: Section 3 briefly covers the necessary background in model checking, neural networks, and the Racetrack benchmark. Section 4 introduces the DSMC connection and discusses our implementation. Section 5 introduces our Racetrack infrastructure, specifically the JANI model and the NN learning machinery. Section 6 describes the case studies and shows how DSMC can be applied. Section 7 evaluates the performance and scalability of the DSMC approach, and Sect. 8 closes the paper with a discussion of the approach and ideas for future developments.

2 Related work

As mentioned above, the need to analyze and verify NNs is becoming more and more important. Thus, several quite different methods have been invented for automated NN analysis, e.g., special methods based on SAT-modulo-theories [22,45,49], abstract interpretation [21,57] or quantitative analysis [17,70] have been developed. But all these techniques have in common that they try to verify individual NN decision episodes, i.e., the behavior of a single input/output function call. The field of analyzing NNs, taking the decisions in the context of a larger system with uncertainty, we enter with our work here, is quite new and unexplored.

Verification of NN control systems by integrating Taylor models and zonotopes [66] has recently been implemented. In addition, UPPAAL Stratego [19] combines formal methods with reinforcement learning and uses, e.g., decision trees for policy presentation and verification [5].

Other works combining formal methods with NNs, for example, study strategy synthesis for partially observable MDPs (POMDPs) to find strategies that fulfill certain probabilistic timed properties. In this approach, a recurrent neural network (RNN) is trained which encodes POMDP strategies. The RNN is then used to construct a Markov chain for which the temporal property can be checked using standard verification techniques [15]. The key difference to our work is that the Markov chain induced by a strategy given by the RNN is fully built and not simulated to check if a given property holds. If it does not hold, a counterexample is generated which helps to locally improve the strategy.

Another work combining formal methods and machine learning reasons about the behavior of NN structures by extracting a decision-tree model of it over which reasoning is possible using model checking [4]. This model forms a correct-by-design controller with performances of usual NNs in reinforcement learning. This controller can be integrated in a bounded model checking procedure to find re-training opportunities.

To be able to add features to NNs acting as a controller without re-training and loosing too much performance, quantitative runtime shields have been invented [6]. The shields may alter the command given by the controller before passing it to the system under control. To generate these shields, reactive synthesis is used, i.e., a stochastic model of the system is built. The controller performance and shield interference is defined by quantitative measures given as weighted automata. The shield construction task can then be reduced to finding an optimal strategy in a stochastic 2-player game.

Furthermore, an iterative learning procedure consisting of SMT-solving and learning phases has been used to construct controllers for stochastic and partially unknown environments [48]. The problem is given as an MDP with an a-priori unknown cost function. Learning techniques can be used to get cost-optimal strategies but without safety guarantees. By first constructing a set of safe schedulers using an SMT-solver and then refining this set to an optimal scheduler, the problem can be solved.

In addition, a reinforcement learning algorithm has been invented to synthesize policies which fulfill a given linear time property on an MDP [40]. By expressing the property as a Limit Deterministic Büchi Automaton a reward function over the state-action pairs of the MDP can be defined such that the policy is only constructed by considering the part of the MDP which fulfills the property.

Another work on controller synthesis and verification uses policy refinement to construct strategies fulfilling temporal logic syntactically co-safe properties, which can be unbounded in time, on general MDPs (discrete-time stochastic models over uncountable state spaces) by using approximately similar abstract models [34].

Avoid reachability properties have been verified on neural agent-environment systems represented as a feed-forward ReLU NN by expressing the problem as a mixed-integer linear program [2]. This approach has been applied to arbitrary-step reachability properties and properties asking if an action will be applied. An extension of this work [3] also supports agents defined on recurrent NNs [41] using a simplified version of linear temporal logic on bounded executions.

3 Background

In this section, we introduce the theoretical background and all the concepts we need and build upon later when presenting and discussing our DSMC approach on the Racetrack benchmark.

3.1 Markov decision processes

The models we consider are discrete-state Markov Decision Processes (MDP). For any nonempty set S, we let $\mathcal{D}(S)$ denote the set of probability distribution over S. We write $\delta(s)$ for the *Dirac distribution* that assigns probability 1 to $s \in S$.

Definition 1 *(Markov Decision Process)* A Markov Decision Process (MDP) is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_0 \rangle$ consisting of a finite set of *states* $\mathcal{S}$, a finite set of *actions* $\mathcal{A}$, a partial *transition probability function* $\mathcal{T} : \mathcal{S} \times \mathcal{A} \hookrightarrow \mathcal{D}(\mathcal{S})$, and an *initial state* $s_0 \in \mathcal{S}$. We say that action $a \in \mathcal{A}$ is *applicable* in state $s \in \mathcal{S}$ if $\mathcal{T}(s, a)$ is defined. We denote by $\mathcal{A}(s) \subseteq \mathcal{A}$ the set of actions applicable in s. We assume that $\mathcal{A}(s)$ is nonempty for each s (which is no restriction because always a self-loop can be added).

MDPs are often associated with a *reward* structure, specifying numerical rewards to be accumulated when moving along state sequences, i.e., $r : \mathcal{S} \times A \times \mathcal{S} \to \mathbb{R}$. Here we are interested instead in the probability of property satisfaction. Rewards, however, appear in our case study as part of the NN training which aims at optimizing the return

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{1}$$

which is the accumulated discounted reward from time t on, where R_i is the random variable representing the reward obtained in step i, $\gamma \in [0, 1]$ is a discount factor, and T is the final time step [68].

The behavior of an MDP is usually considered together with an entity resolving the otherwise nondeterministic choices in a state. This is effectuated by an *action policy* (or scheduler, or adversary) that determines which applicable action to apply when and where. In full generality, this policy may use randomization (picking a distribution over applicable actions), and it may use the past history when picking. The former is of no importance for the setting considered here, while the latter is. Histories are represented as finite sequences of states (i.e., words over $\mathcal{S}$), thus they are drawn from $\mathcal{S}^+$. We use *last*(w) to denote the last state in $w \in \mathcal{S}^+$.

Definition 2 *(Action Policy)* A (deterministic, history-dependent) *action policy* is a function $\sigma : \mathcal{S}^+ \to \mathcal{A}$ such that $\forall w \in \mathcal{S}^+ : \sigma(w) \in \mathcal{A}(\mathit{last}(w))$. An action policy is *memoryless* if it satisfies $\sigma(w) = \sigma(w')$ whenever $\mathit{last}(w) = \mathit{last}(w')$.

Memoryless policies can equally be represented as $\sigma : \mathcal{S} \to \mathcal{A}$ such that $\forall s \in \mathcal{S} : \sigma(s) \in \mathcal{A}(s)$.

Definition 3 *(Markov Chain)* A Markov Chain is a tuple $\mathcal{C} = \langle \mathcal{S}, \mathcal{T}, s_0 \rangle$ consisting of a set of states $\mathcal{S}$, a transition probability function $\mathcal{T} : \mathcal{S} \to \mathcal{D}(\mathcal{S})$ and an initial state $s_0 \in \mathcal{S}$.

An MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_0 \rangle$ together with an action policy $\sigma : \mathcal{S}^+ \to \mathcal{A}$ induces a countable-state Markov chain $\langle \mathcal{S}^+, \mathcal{T}', s_0 \rangle$ over state histories in the obvious way: For any $w \in S^+$ with $\mathcal{T}(\mathit{last}(w), \sigma(w)) = \mu$, set $\mathcal{T}'(w) = d$ where $d(ws) = \mu(s)$. For memoryless σ, the original state space $\mathcal{S}$ can be recovered by setting $\mathcal{T}'(\mathit{last}(w)) = \mu$ in the above, since both are lumping equivalent [12].

3.2 Probabilistic and statistical model checking

Model checking of probabilistic models (such as MDPs) nowadays comes in two flavors. *Probabilistic model checking* (PMC) [53] is an algorithmic technique to determine the extremal (maximal or minimal) probability (or expectation) with which an MDP satisfies a certain (temporal logic) property when ranging over all imaginable action policies. For some types of properties (step-bounded reachability, expected number of steps to reach) it does not suffice to restrict to memoryless policies, while for others (inevitability, step-unbounded reachability) it does. At the core of PMC are numerical algorithms that require the full state space to be available upfront (in some way or another) [37,61].

If fixing a particular policy, the MDP turns into a Markov chain. In this setting, *statistical model checking* (SMC) [42,55,71] is a popular alternative to probabilistic model checking. This is because PMC, requiring the full state space, is limited by the state space explosion problem. SMC is not, even if the underlying model is infinite in size. Furthermore, SMC can extend to non-Markovian formalisms or complex continuous dynamics effectively. At its core, SMC harvests classical Monte Carlo simulation and hypothesis testing techniques. In a nutshell, n finite samples of model executions are generated and evaluated to determine the fraction of executions satisfying a property under study. This yields an estimate q' of the actual value q of the property, together with a statistical statement on the potential error. A typical guarantee is that $\mathbb{P}(|q' - q| < \epsilon) > \delta$, where $1 - \delta$ is the confidence that the result is ϵ-correct. To decrease ϵ and δ, n must be increased. SMC is attractive as it only requires constant memory independent of the size of the state space. When facing rare events, however, the number of samples needed to achieve sufficient confidence may explode.

In the MDP setting (or more complicated settings), SMC analysis is always bound to a particular action policy turning an otherwise nondeterministic model into a stochastic process. Nevertheless, many SMC tools support nondeterministic models, e.g., PRISM [52] and UPPAAL SMC [20]. They use an implicitly defined uniform random action policy to resolve choices. UPPAAL Stratego [19] is using Q-learning and SMC to iteratively learn a near-optimal policy. Reinforcement learning strategies to tackle continuous state spaces have also been integrated in the tool [46]. In addition, for probabilistic timed automata strategies to find near-optimal schedulers have been developed using abstraction and sampling techniques [18]. The statistical model checker MODES [13], which is part of the MODEST TOOLSET [38], lets the user choose out of a small set of predefined policies, or provides light-weight support for iterating over policies [13,56] to statistically approximate an optimal policy in addition to the uniform random scheduler. In any case, results obtained by SMC are to be interpreted relative to the implicitly or explicitly defined action policy.

In the following, we will use the statistical model checker MODES of the MODEST TOOLSET which contains simulation algorithms specifically tailored to MDPs and more advanced models. The tool is implemented in C#. It offers multiple statistical methods including confidence intervals, Okamoto bound [60], and SPRT [69]. As simulation is easily and efficiently parallelizable, MODES can exploit multi-core architectures.

3.3 Deep Q-learning

Neural networks (NN) have recently been applied with dramatic successes to the learning of action policies in large transition systems, from Atari games [59] over Go and Chess [67] to Rubik's cube [1]. This clearly suggests that NNs will play a key role in action decisions of autonomous systems in the future. In particular, this pertains to action decisions in environments formalizable as MDPs.

NNs consist of neurons: atomic computational units that typically apply a nonlinear function, their *activation function*, to a weighted sum of their inputs [65]. For example, *rectified linear units (ReLu)* use the activation function $f(x) = \max(0, x)$. Here, we consider feed-forward NNs, a classical architecture where neurons are arranged in a sequence of layers. Inputs are provided to the first (input) layer, and the computation results are propagated through the layers in sequence until reaching the final (output) layer. In every layer, every neuron receives as inputs the outputs of all neurons in the previous layer. For a given set of possible inputs $\mathcal{I}$ and (final layer) outputs $\mathcal{O}$, a neural network can be considered as an efficient-to-query total function $\pi : \mathcal{I} \to \mathcal{O}$.

So-called deep neural networks consist of many layers. In tasks such as image recognition, successful NN architectures have become quite sophisticated, involving, e.g., convolution and max-pooling layers [51]. Feed-forward NNs are comparatively simple, yet they are in widespread use [24], and are in principle able to approximate any function to any desired degree of accuracy [44].

Such NNs can be trained in a multitude of ways. Here we use *deep Q-learning* [59], a successful and nowadays widespread form of deep reinforcement learning (DRL). In DRL, the NN is trained by iteratively executing the policy and updating it. Each step executes the current NN from some state, and updates the NN weights using gradient descent.

The so-called q-values represent the expected return, i.e., the expected discounted accumulated reward, that is received when taking an action a and following the q-values-induced policy afterward. In classical Q-learning [68], these q-values are learned separately for each state-action pair by using a table for approximation. In contrast, deep Q-learning uses an NN to jointly approximate all the q-values, i.e., the q-values of all actions, for a given state. Such an NN is also called *deep Q-network* (DQN).

Deep Q-learning has been shown to learn high-quality NN action policies in a variety of challenging decision-making problems [59], and especially to perform better on the benchmark we used here, the Racetrack, then policies trained with supervised learning [33].

4 Neural networks as MDP action policies

4.1 Connecting MDP and action oracle

Racetrack is a simple instance of many further examples representing real-world phenomena that involve randomness and decision-making. This is the natural scenario where NNs are taking over ever more duties. In essence, their role is very close to that of an action policy: Decide in each situation what options to pick next. If we consider the "situations" (the inputs $\mathcal{I}$) as the states $\mathcal{S}$ of a given MDP, and the "options" (outputs $\mathcal{O}$) as actions $\mathcal{A}$, then the NN is a function $\pi : \mathcal{S} \rightarrow \mathcal{A}$. We call such a function an *action oracle*. Indeed this is what the reinforcement learning process in Q-learning and other approaches delivers naturally.

Observe that an action oracle can be cast into an action policy except for a subtle problem. Action policies only pick actions (from $\mathcal{A}(s)$, thus) applicable at the current state s, while action oracles may not. A better fitting definition would constrain oracles to always return an applicable action. Yet it is not clear how to guarantee this for NNs – it is easy to see that, even for linear multi-classification, the hard constraints required to guarantee action applicability lead to non-convex optimization problems. An easy fix would use the highest-ranked applicable action instead of the NN classifier output itself. For our purposes however, where we want to analyze the quality of the NN oracle, it makes sense to explicitly distinguish inapplicable actions as a form of low quality.

If an oracle returns an inapplicable action, then no valid behavior is prescribed and in that sense the system can be considered stalled.

Definition 4 *(Action Oracle Stalling)* Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_0 \rangle$ be an MDP, and $\pi : \mathcal{S} \rightarrow \mathcal{A}$ be an action oracle. We say that $s \in \mathcal{S}$ is *stalled* under π if $\pi(s) \notin \mathcal{A}(s)$.

To accommodate for stalling, we augment the MDP upfront with a fresh action † available at every state, this action is chosen upon stalling, leading to a fresh state ‡ with only that action to continue. So $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_0 \rangle$ is transformed into $\mathcal{M}^{\ddagger} = \langle \mathcal{S} \cup \{\ddagger\}, \mathcal{A} \cup \{\dagger\}, \mathcal{T}', s_0 \rangle$ where for each state s, $\mathcal{T}'(s, \dagger) = \delta(\ddagger)$ and otherwise $\mathcal{T}'(s, a) = \mathcal{T}(s, a)$ wherever the latter is defined.

Definition 5 *(Oracle induced Markov chain)* Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_0 \rangle$ be an MDP, and let π be an action oracle for $\mathcal{M}$. Then the Markov chain $\mathcal{C}^{\pi}$ induced by π is the one induced in $\mathcal{M}^{\ddagger}$ by the memoryless action policy σ defined by $\sigma(w) = \dagger$ whenever $last(w)$ is ‡ or stalled under π, and otherwise by $\sigma(w) = \pi(last(w))$.

In words, the oracle induced policy fixes the probability distribution over transitions in each state to that of the chosen action. If that action is inapplicable, then the chain transitions to the fresh state ‡ which represents stalled situations.

Deep Statistical Model Checking. Overall, $\mathcal{C}^{\pi}$ is a Markov chain that uses π as an oracle to determinize the MDP $\mathcal{M}$ whenever possible, and stalls otherwise. With π implemented by a neural network, we can use statistical model checking on $\mathcal{C}^{\pi}$ to analyze the NN behavior in the context of $\mathcal{M}$. This analysis has the potential to deliver deep insights into the effectiveness of the NN applied, allowing for comparisons with other policies and also with optimal policies, the latter obtained from exhaustive model checking. From a practical perspective, an important remark is that in the definitions above and in our implementation of DSMC described below, the inputs to the NN are assumed to be the MDP states $s \in \mathcal{S}$. This captures the scenario where the NN takes the role of a classical system controller, whose inputs are system state attributes, such as program variables. More generally, the connection from the MDP model to the NN input may require an intermediate function f mapping $\mathcal{S}$ to the input domain of the NN. This is in particular the case for NNs processing image sequences, like in vision systems in autonomous driving. In such a scenario, the MDP model states have to represent the relevant aspects of the NN input (e.g., objects and their properties in an image). This advanced form of connection remains a topic for future work. It lacks the crisp nature of the problem considered here.

4.2 DSMC implementation

Deep statistical model checking is based on a pair consisting of an NN and an MDP operating on the same state space. The NN is assumed to be trained externally prior to the analysis, in which it is combined with the MDP. To experiment with this concept in a real environment, we have developed a DSMC implementation inside the MODEST TOOLSET [38], which includes the explicit-state model checker MCSTA, and in particular the statistical model checker MODES [13]. MODES thus far offers the options `Uniform` and `Strict` to resolve nondeterminism. We implemented a novel option called `Oracle`, which calls an external procedure to resolve nondeterminism. With that option in place, every time the next action has to be chosen, MODES provides the current model state s to the `Oracle`, which then calls the external procedure and returns the chosen action to MODES. In this way, the `Oracle` can connect to an external NN serving as an action oracle from MODES's perspective.

At the implementation level, connecting to standard NN tools is non-trivial due to the programming languages used. The MODEST TOOLSET is implemented in C#, whereas standard NN tools are bound to languages like Python or Java. Our key observation to overcome this issue is that a seamless integration is not actually required. Standard NN tools are primarily required for NN *training*, which is computationally intensive and requires highly optimized code. In contrast, implementing our NN `Oracle` requires only NN *evaluation* (calling the NN on a given input) which is easy—it merely requires to propagate the input values through the network. We thus implemented NN evaluation directly in the MODEST TOOLSET's code base, as part of our extension. The NNs are learned using standard NN tools. From there, we export a file containing the NN weights and biases. Our extension of MODES reads that file, and uses it to reconstruct the same NN, for use with our evaluation procedure. When the `Oracle` is called, it connects to that procedure.

5 Racetrack

As previously outlined, we consider Racetrack as a simple and discrete, yet highly extensible approximation of real-world phenomena that involve randomness and decision-making. In this section, we spell out how these benchmarks are made concrete use of, how they are implemented and designed in detail.

5.1 Background on Racetrack

Originally, Racetrack is a pen and paper game [23]. A track is drawn with a start line and a goal line on a squared sheet of paper. A vehicle starts with velocity 0 from some position on the start line, with the objective to reach the goal as fast as possible without crashing into a wall. Nine possible actions modify the current velocity vector by one unit (up, down, left, right, four diagonals, keep current velocity). This simple game lends itself naturally as a benchmark for sequential decision-making in risky scenarios. In particular, when extending the problem with noise, we obtain MDPs that do not necessarily allow the vehicle to reach the goal with certainty. In a variety of such noisy forms, Racetrack was adopted as a benchmark for MDP algorithms in the AI community [9,11,58,62,63]. Because of its analogy to autonomous driving, Racetrack has recently also been used in multiple verification and model checking contexts [7].

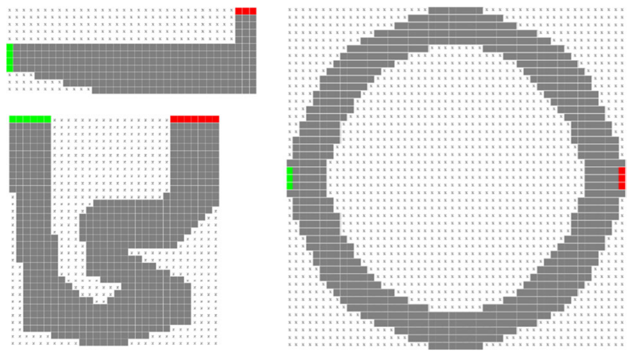
Fig. 1 The maps of our Racetrack benchmarks: Barto-small (left top), Barto-big (left bottom), Ring (right)

Like in previous work, we consider the single-agent version of the game. We use some of the benchmarks, i.e., track shapes, that are readily available. Specifically, we use the three Racetrack maps illustrated in Fig. 1, originally introduced by Barto et al. [9]. The track itself is defined as a two-dimensional grid, where each cell of the grid can represent a possible starting position “`s`” (indicated in green), a goal position “`g`” (red), or can contain a wall “`x`” (white, crossed). Like Barto et al. [9], we consider a noisy version of Racetrack that emulates slippery road conditions: actions may *fail* with a given probability, in which case the action does not change the velocity and the vehicle instead continues driving with unchanged velocity vector.

5.2 JANI framework

Central to our practical work is the JANI-*model format* [14,47]. It can express models of distributed and concurrent systems in the form of networks of automata, and supports property specification based on probabilistic computation tree logic (PCTL) [36]. In full generality, JANI models are networks of stochastic timed automata, but we concentrate on MDPs here. Automatic translations from and into other modeling languages are available, connecting among others to the planning language PPDDL [50] and to the PRISM language, and thus to the model checker Prism [52]. A large set of quantitative verification benchmarks (QVBS) [39] is available in Jani, and many tools offer direct support, among them ePMC, Storm and the Modest Toolset [25,35,38].

5.3 Racetrack model in Jani

In the following, we discuss the details of the Racetrack model representation and implementation in Jani as done in our online appendix of *DSMC20* [30,31].

The track itself is represented as a (constant) two-dimensional array whose size equals that of the grid. The Jani files of different Racetrack instances differ only in this array. Vehicle movements and collision checks are represented by separate automata that synchronize using shared actions.

The vehicle automaton keeps track of the current state of the vehicle via four bounded integer variables, position and directional velocity, described by two vectors: its current position (x, y) indexing a cell within the grid, and its current velocity $(d_x, d_y) \in \mathbb{Z}^2$ in x- and y-direction. The state of the vehicle is updated at discrete steps. At each step, the speed of the vehicle can be controlled via 9 different actions corresponding to the acceleration vectors $(a_x, a_y) \in (\{-1, 0, 1\})^2$. Acceleration is applied additively, i.e., the vehicle's new velocity vector (d'_x, d'_y) after applying acceleration (a_x, a_y) is given by $d'_x = d_x + a_x$ and $d'_y = d_y + a_y$. The position of the vehicle is updated according to the updated velocity vector, i.e., $x' = x + d'_x$ and $y' = y + d'_y$.

What we just specified is the deterministic variant of Racetrack. In the noisy variant, acceleration only succeeds with a probability of $p \in [0, 1)$, while with probability $(1 - p)$ the vehicle's velocity remains the same.

In addition, a state of the model contains two Boolean variables indicating whether the vehicle has crashed, or has reached a goal cell. We say that the vehicle has *crashed* if the vehicle either moved out of the grid (i.e., its position no longer constitutes a valid grid coordinate), or the vehicle's last movement trajectory crossed a wall cell.

As described, the vehicle automaton starts at a location with one edge for each one of the 9 different acceleration vectors. Each of the edges updates the velocity accordingly and sends the start and resulting end coordinates to the collision check automaton. The collision check can respond with three different answers: "valid", "crash", or reached "goal". If the trajectory was valid, the vehicle automaton transitions back to its initial location. Otherwise the vehicle automaton transitions into a terminal location where no further moves are possible.

The collision check automaton takes care of two things. It first checks whether the vehicle's destination lies within the grid. If so, it then iteratively computes the discretized trajectory T, and looks up for each referenced coordinate whether the corresponding entry in the grid array represents a wall or goal cell. If the trajectory leads out of the track, or when an intersection of the trajectory with either a wall or a goal cell is detected, the result is immediately sent to the vehicle automaton. If the trajectory was completely generated without detecting a collision, the vehicle automaton's request is answered with "valid", and the location is reset, waiting for the next trajectory to test.

Determining whether the vehicle has crashed or has passed a goal is done by discretizing the trajectory from the vehicle's former position $(x_0, y_0) := (x, y)$ to its new position $(x_n, y_n) := (x', y')$ into a sequence of coordinates $T = \langle (x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n) \rangle$. Then, the vehicle has touched a wall if and only if T references a coordinate of a wall or goal cell, respectively. Checking whether the vehicle traversed a goal cell is done in the same fashion. The trajectory discretization T is defined as displayed in Eq. 2, where $\sigma_x = sgn(d_x)$, $\sigma_y = sgn(d_y)$ and $m_x = \frac{d_x}{|d_y|}$, $m_y = \frac{d_y}{|d_x|}$. In words, if either the horizontal or vertical speed is 0 (cases 1 to 3), the trajectory contains exactly all grid coordinates on the straight line between (x, y) and (x', y'). Otherwise, we linearly interpolate n points between the two positions and then for each such point round to the closest position on the map. In our model, n is given by $\max\left(|d_x|, |d_y|\right)$, while the original discretization models always choose $n = d_x$. The latter is problematic when having a velocity which moves more into the y- (case 5) than into the x-direction (case 4), as then only few points will be contained in the trajectory and counterintuitive results are produced.

5.4 Scaling Racetrack

In the scalability study, which we will show later in this paper, we scale a Racetrack benchmark up by using finer discretizations, thereby effectively making the track larger to navigate. This scaling approach is simple and canonical, and facilitates a detailed direct comparison across different sizes. Specifically, we scale by the factor N where every cell in the original map is replaced by a square of N^2 cells. The map growth thus is quadratic in N; e.g., for the Barto-big map shape in Fig. 1, the original map has a size of 30×33 cells, while with $N = 2$ we get 60×66 cells and with $N = 3$ we get 90×99 cells and so on.

5.5 Learning neural networks for Racetrack

For the sake of realistic empirical studies, we have drawn on established NN learning techniques to obtain NN oracles for the Racetrack case studies. Here, we briefly summarize the main design decisions. Notably, DSMC is entirely indepen-

$$T = \begin{cases} \langle (x, y) \rangle & \text{if } d_x = 0 \text{ and } d_y = 0 \;(1) \\ \langle (x, y), (x+\sigma_x, y), (x+2\cdot\sigma_x, y)\ldots, (x', y') \rangle & \text{if } d_x \neq 0 \text{ and } d_y = 0 \;(2) \\ \langle (x, y), (x, y+\sigma_y), (x, y+2\cdot\sigma_y)\ldots, (x', y') \rangle & \text{if } d_x = 0 \text{ and } d_y \neq 0 \;(3) \\ \langle (x, y), (x+\sigma_x, \lfloor y+m_y \rceil), (x+2\cdot\sigma_x, \lfloor y+2\cdot m_y \rceil)\ldots, (x', y') \rangle & \begin{array}{l}\text{if } d_x \neq 0 \text{ and } d_y \neq 0 \\ \quad\text{and } |d_x| \geq |d_y|\end{array} \;(4) \\ \langle (x, y), (\lfloor x+m_x \rceil, y+\sigma_y), (\lfloor x+2\cdot m_x \rceil, y+2\cdot\sigma_y)\ldots, (x', y') \rangle & \begin{array}{l}\text{if } d_x \neq 0 \text{ and } d_y \neq 0 \\ \quad\text{and } |d_x| < |d_y|\end{array} \;(5) \end{cases} \qquad (2)$$

dent of the concrete learning process, depth, and shape of the NN employed.

NNs are learnt for a specific map (cf. Fig. 1), with the inputs being 15 integer values, encoding the two-dimensional position, the two-dimensional velocity, the distance to the nearest wall in eight directions, the x and y differences to the goal coordinates, and Manhattan goal distance (absolute x- and y-difference, summed up). Actions to accelerate in the 9 possible directions are encoded as classification outputs, i.e., the output layer consists of 9 neurons.

A crucial design decision is the learning objective, i.e., the rewards used in deep Q-learning. We set the reward for reaching the goal line to 100, and for crashing into a wall to values within $[-50, -20]$. We used a discount factor of 0.99 to encourage short trajectories to the goal. This arrangement was chosen because, empirically, it resulted in an effective learning process [27]. With higher negative rewards for crashing, the policies learn to prefer not to move or to move in circles.

Similarly, smaller negative rewards make the learnt policies prefer to crash quickly. Using a discount factor yieldsbetter learning performance, but does not match the overall Racetrack setup. This exemplifies that the choice of objectives for learning is governed by learning performance. Both meta-parameters and numeric parameters such as rewards typically require fine-tuning orthogonal to, or at least below the level of abstraction of, the qualities of interest in the application.

We experimented with a range of NN architectures and hyperparameter settings, the objective being to keep the NNs simple while still able to learn useful oracles in our Racetrack benchmarks. The NNs we settled on have the above described input and output layers, and two hidden layers each of size 64. All neurons use the ReLU activation function.

NNs are learnt in two variants: First, starting on the starting line (so called normal start (NS)) vs. second, starting from a random point anywhere on the map (so-called random start (RS)), each with initial velocity 0. Variant RS turned out to yield much more effective and robust learning.

Intuitively, RS seems a more challenging task as there is more that the policy needs to learn. Still, for NS, it takes the policy a long time to reach the goal at all, while with RS this happens more quickly yielding earlier and more robust learning also farther away from the goal. Consider Fig. 2, which depicts the training curve of two policies, one trained in the NS setting and the other in the RS setting. The trainingsplot depicts the sliding mean of the returns achieved during training. For the RS mode, the goal line is already reached shortly after the training starts (as indicated by the dotted orange line) and the policy increases steadily until a plateau, where the policy only improves slightly, is reached. In contrast, for the NS mode, the goal line is reached for the first time after about 17.000 episodes (blue dashed line) and therefore just then receives the first positive reward. Thus, the policy can only start to learn how to reach the goal after these 17.000 episodes, which explains the abrupt increase in achieved returns afterward.

Note, that the average value of achieved returns in the end of training cannot directly be compared to another. As the episodes trained with random start in average are shorter, as they regularly start closer to the goal line, the achieved returns are discounted less and therefore are higher (see Eq. 1).

6 NN quality analysis using DSMC

We now demonstrate the statistical model checking approach to NN policy verification through case studies in Racetrack. Section 6.1 illustrates the use of DSMC for quality assurance by human analysts (end users, engineers) in system approval. Section 6.2 illustrates the use of DSMC as a tool for the engineers designing the NN learning pipeline.

Throughout, we use MODES with an error bound $P(\text{error} > \epsilon) < \kappa$, where $\epsilon = 0.01$ and $\kappa = 0.05$, i.e., a confidence of 95%. We set the maximal run length to 10,000 steps. Unless otherwise stated, we set the slippery-noise level in Racetrack,

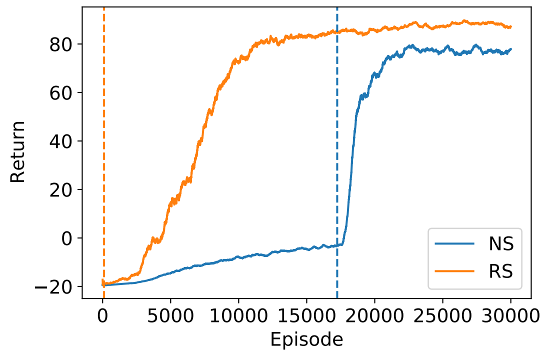

Fig. 2 Combined training plot for training two polices on the Ring map. The curves depict the sliding mean of the last 500 observed returns during training. The dashed lines indicate the first time the goal line was reached, i.e., the first time a positive return was observed

i.e. the probability of action failure, to 20%. The NN oracles are learnt by training runs starting anywhere on the map; we will illustrate how DSMC can highlight the deficiencies of the alternate approach (starting on the starting line only). The experiments in this section were run on an Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz (4 cores, 8 threads) with 32 GB RAM and a 450 GB HDD.

6.1 Quality assurance in system approval

The variety in abstract property specification gives versatility to the quality assurance process. This is important in particular because, as previously argued, the relevant quality properties will typically not be identical to the objectives used for NN learning. In the Racetrack example, NN learning optimizes expected return subject to fine-tuned reward and discount values. For the quality assurance, we consider crash probability and goal probability, expressed as CTL path formulas in JANI, namely $\Diamond crashed$ ("eventually crashed") for the former and $\neg crashed\ U\ goal$ ("not crashed until reaching goal") for the latter.[1]

We highlight that the DSMC analysis can not only point out *that* an NN oracle has deficiencies, but also *where*: in which regions of the MDP state space $\mathcal{S}$. Namely, in cyber-physical systems, it is natural to use the spatial dimension underlying $\mathcal{S}$ for systematizing the analysis and visualizing its result. This delivers not only a yes/no answer, but an actual quality report. We illustrate this here through the use of simple heat maps over the Racetrack road map. The heat maps visualize the value of the respective property for every cell when starting in it with velocity 0.

Figure 3 shows quality assurance results for crash probability in all the Racetrack benchmarks, using for each the best NN oracle from reinforcement learning (i.e. those yielding highest returns). The heat maps use a simple color scheme as an illustration how the analysis results can be visualized for the human analysts. Similar color schemes will be used in all plots below.

From the displayed DSMC results, quality assurance analysts can directly conclude that the NN oracles are fairly safe in Barto-small (left top), with crash probabilities mostly below 0.1; but not on Barto-large (left bottom) and Ring (right) where crash probabilities are above 0.5 on significant parts of the map. Generally, crash probability increases with distance to the goal line. Some interesting subtleties are also visible, for example that crash probabilities are relatively high in the left-turn before the goal in Barto-small.

Our next results, in Fig. 4, illustrate the quality assurance versatility afforded by DSMC, through an analysis quite different from the previous one. The human analysts here decide to evaluate goal probability (a quality stronger than not crashing because the latter may be achieved by idling). Apart from the original setting, they consider a stress-test scenario where the road is significantly more slippery than during NN training, namely 50% instead of 20%. They finally decide to compare with optimal goal probabilities, computable via the probabilistic model checker MCSTA, so that they can see whether any deficiencies are due to the NN, or are unavoidable given the high amount of noise.

The figure shows the outcome for Barto-large. One of the deficiencies is immediately apparent, the NN policy does not pass the stress test. Its goal probability matches the optimal values only near the goal line, and exhibits significant deficiencies elsewhere. Based on these insights, the quality analysts can now decide whether to relax the stress-test (after all, even optimal behavior here does not reach the goal with certainty), or whether to reject these NN polices and request re-training.

6.2 Learning pipeline analysis and revision

More generally, DSMC can yield important insights not only for quality assurance, but also for the engineers designing the NN learning pipeline in the first place. There are two distinct scenarios:

(i) The engineers run the same success tests as in quality assurance, and re-train if a test is not passed.
(ii) The engineers assess different properties of interest to the learning process itself (e.g. expected length of policy runs), or assess the impact of different hyperparameter settings.

In both scenarios, the DSMC analysis results point to specific state space regions that require improvement. This can be directly operationalized to revise the learning pipeline, by

[1] Further properties of interest could be, e.g., bounded goal probability (how likely is it that we will reach the goal within a given number of steps?), expected number of steps to goal, or risk of stalling.

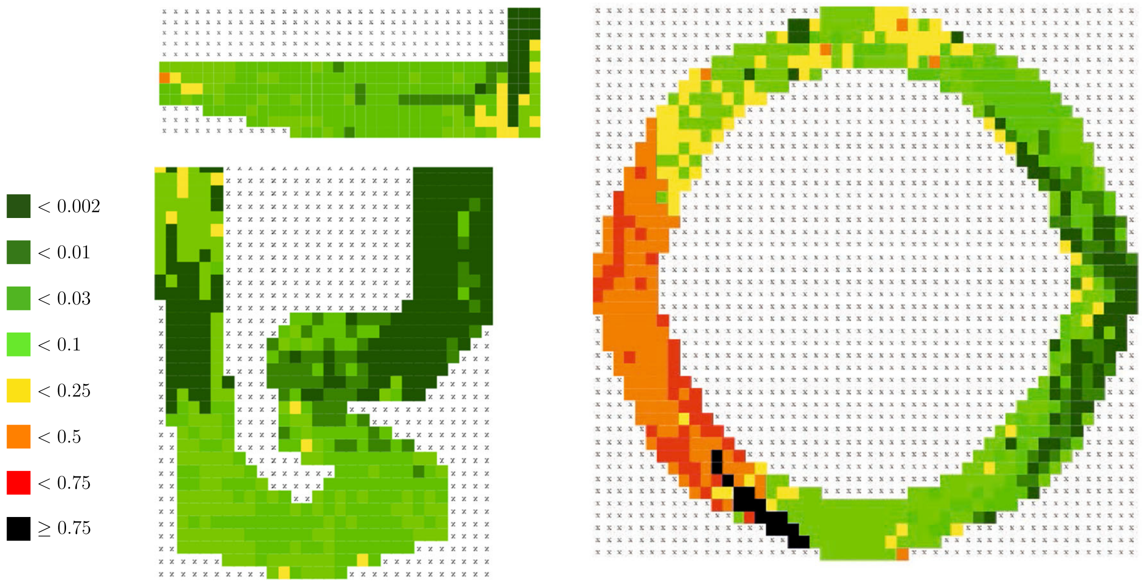

Fig. 3 Heat maps of NN-induced crash probabilities for all Racetrack benchmarks

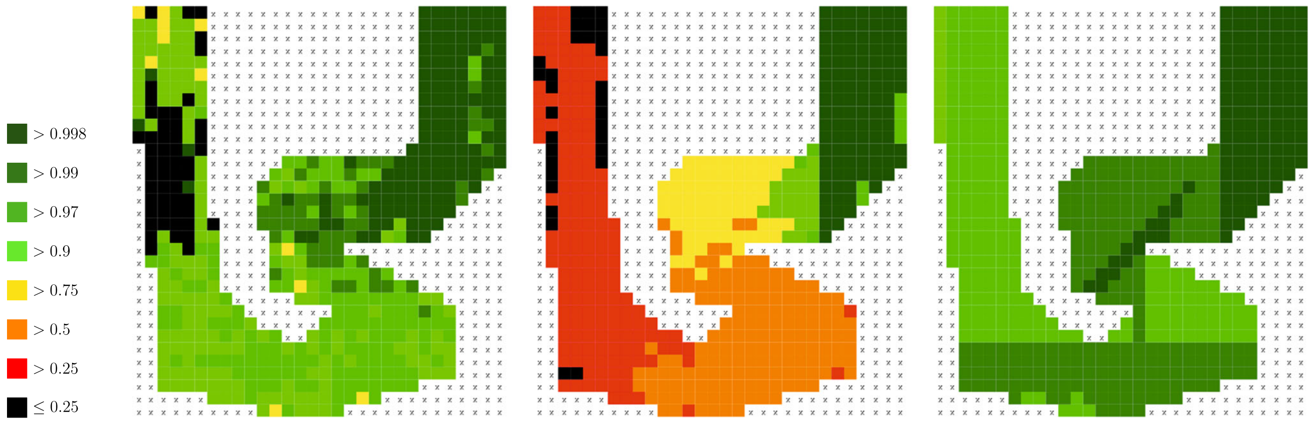

Fig. 4 Goal probability of NN oracle on the Barto-big benchmark trained and executed with 20% noise versus stress-test executed with 50% noise using the same NN (middle) versus optimal policies obtained by probabilistic model checking with 50% noise (right)

starting more training runs from states in the critical regions. DSMC has already been applied for analyses of this kind during evaluation stages [32].

Figures 3 and 4 above have already demonstrated (i). Next we demonstrate (ii) through two case studies analyzing different hyperparameter settings.

Our first case study, in Fig. 5, analyzes the number n of training episodes, as a central hyperparameter of the learning pipeline. The only information available in deep Q-learning for the choice of this hyperparameter is the learning curve, i.e., the expected return as a function of n, depicted on the right. Yet, as our DSMC analysis here shows, this information is insufficient to obtain reliable policies. In Barto-big, the highest return is obtained after $n = 90{,}000$ episodes. From $n = 70{,}000$ to $n = 90{,}000$, the return slightly increases. Yet we see in Fig. 5 that the additional 20,000 training episodes, while increasing overall goal probability, lead to highly deficient behavior in an area near the start of the map, where goal probability drops below 0.25. If provided with that information, the engineers can focus additional training on that area, for instance.

In our next case study, we assume that the NN engineers decide to analyze the impact of starting training runs on (a) the starting line versus (b) random points anywhere on the map. Figure 6 shows the results for the Ring map, where they are most striking. In variant (a), the top part of the Racetrack was completely ignored by the learning process. Looking into this issue, one finds that, during training, the first solution happens to be found via the bottom route. From there on,

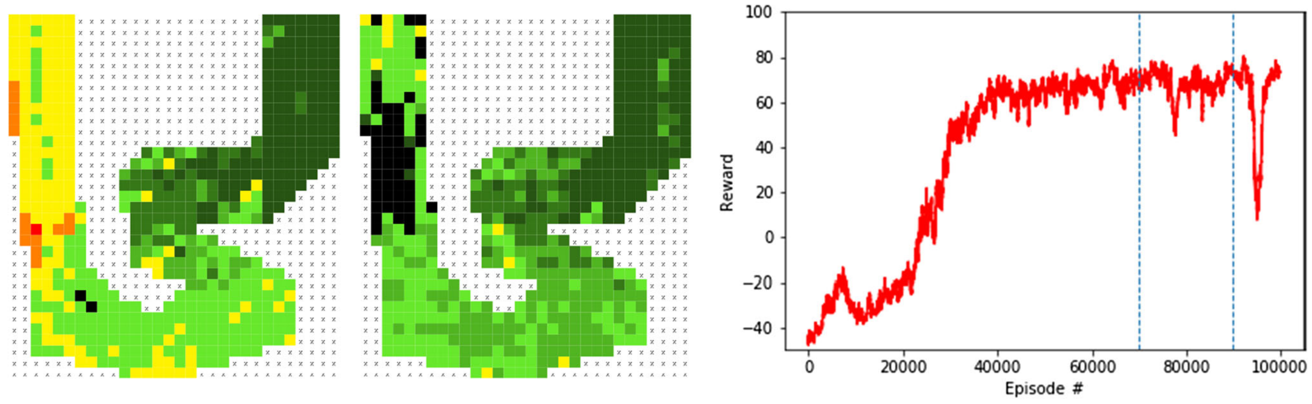

Fig. 5 Goal probabilities on the Barto-big benchmark (color coding as in Fig. 4), for NN oracles learnt over n = 70,000 (left) and n = 90,000 (middle) training episodes, together with Q-learning curve (right)

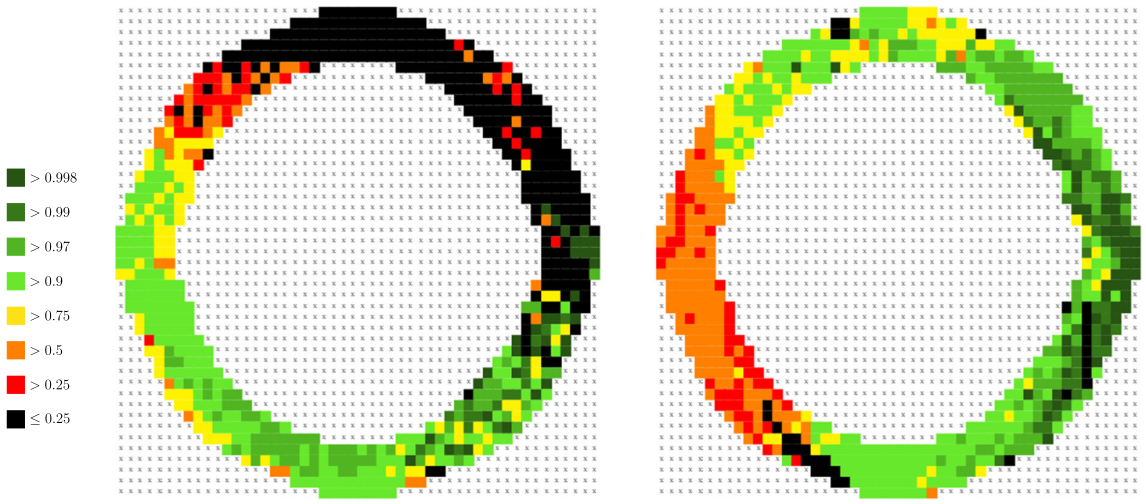

Fig. 6 Goal probabilities in Ring for NN oracles where training was carried out with reinforcing runs from the start line only (left) versus from anywhere on the map (right)

the reinforcement learning process has a strong bias to that route, preventing any further exploration of other routes.

Phenomena like this are highly detrimental if the learnt policy needs to be broadly robust, across most of the environment. The deficiency is obvious given the DSMC analysis results, and these results make it obvious how the problem can be fixed. But neither can be seen in the learning curves.

7 Computational performance of DSMC

After having demonstrated the strengths and usefulness of the DSMC approach, it remains to show its feasibility in a performance evaluation and scalability study. Section 7.1 evaluates the computational effort incurred by DSMC compared to a conventional SMC setting where the MDP policy is coded in the model itself. Afterward, we consider size scaling (see Sect. 7.2) of the benchmarks and evaluate scalability in different dimensions. Section 7.3 demonstrates scalability as a function of training episodes and Sect. 7.4 concentrates on scalability w.r.t. instance size

7.1 NN versus engineered policy

As discussed, it can be highly demanding or infeasible to verify the input/output behavior of even a single NN decision episode, and that complexity is potentially compounded by the state space explosion problem when endeavoring to verify the behavior induced by an NN oracle. Deep statistical model checking carries promise as a “light-weight” approach to this formidable problem, as no state space needs to be stored and on the NN side it merely requires to call the NN on sample inputs. In addition, it is efficiently parallelizable, just like SMC. Yet (1) the approach might suffer from an excessive number of sample runs needed to obtain sufficient confidence, and/or (2) the overhead of NN calls might severely hamper its runtime feasibility.

Figure 7 shows data regarding (1). We compare the effort for analyzing our NN policies to that required for analyzing a conventional engineered (hand-coded) policy that we incorporated into our JANI models.[2] As the heat maps show, the latter effort is higher. This is due to a tendency to more risky behavior in the hand-made policy, resulting in higher variance. Regarding (2), the runtime overhead for NN calls is actually negligible in our study. Each call takes between 1 and 4 ms. There is an added overhead for constructing the NN once at the beginning of the analysis, but that takes at most $6ms$.

These results should not be over-interpreted, as they pertain to the particular engineered policy experimented with. Nevertheless, they indicate that, as one would expect, the performance variance of NN polices (and therewith the DSMC analysis effort) is not necessarily higher than that of conventional policies.

As a side remark, please note, that for both of these aspects, we decided not to compare to SMC using a uniform random scheduler because first, driving randomly around is quite unrealistic, e.g., because it is quite unsafe. Second, we saw in our experiments with a uniform random scheduler that the goal probability calculated with SMC is 0 in most of the cases because it is so unsafe. Thus, SMC with a random scheduler and DSMC are not comparable because the results and runtimes are influenced by more factors than just by replacing the NN by a scheduler.

7.2 Scalability study: setup

In the remainder of this section, we consider size scaling, using the scaled Racetrack instances as per Sect. 5.4. We concentrate on the Barto-big track shape in Fig. 1. Fixing that shape, we scale up by using finer discretizations, thereby effectively making the track larger to navigate. This may impact the performance of DSMC (number of sample runs, runtime) in several ways:

(i) Analyzing policy behavior from every map cell (with initial velocity 0), the number of calls to DSMC equals the number of cells after scaling.
(ii) The MDP becomes larger and individual policy runs become longer, which may affect the number of sample runs required to obtain the desired statistical confidence in the analysis result.
(iii) The quality of an NN policy—its ability to successfully navigate the map—may affect the number of sample runs required in DSMC.

We now summarize the results of our study examining these effects. We consider (iii) first as it turns out to influence DSMC performance quite substantially, thus being important to understand as a prerequisite for our scalability study. We analyze (iii) as a function of training degree, which is of interest in itself if one is interested in analyzing the NN policy under training at different stages (which is a natural application of DSMC). Given our insights into (iii), we then turn to our study of (i) and (ii) using NN policies of comparable quality.

All experiments were run on 5 virtual machines having an AMD EPYC Processor at approximately 2.5 GHz using Ubuntu 18.04 with 8 vCPUs and 16 GB RAM. A total of 158,377 processing hours have been invested in this study, i.e., reproducing already a fraction of these results takes a lot of time. All our scripts and infrastructure we used are available online at https://doi.org/10.5281/zenodo.7071405.

Like in the experiments described above, we use MODES with an error bound $P(\text{error} > \epsilon) < \kappa$, where $\epsilon = 0.01$ and $\kappa = 0.05$, i.e., a confidence of 95% and a maximal run length of 10,000 steps.

We investigated if the performance when running a DSMC experiment with a specific NN multiple times is affected by perturbations caused by the probabilistic behavior of the model or the mode of operation of SMC. Thereby, we observed that the performance and quality differences are negligible and mostly caused by machine performance variations and thus will not look deeper into this in the following.

7.3 Scalability as a function of training episodes

To evaluate the impact of training strength on the runtime of DSMC, we extracted networks for the Barto-big map in Fig. 1 after 5k, 10k, 15k, 20k and 25k training episodes for $N = 1$, and for $N = 2$ after 30k, 35k, 40k and 45k training episodes (because here training takes longer). Figure 8 summarizes the results.

DSMC exhibits an easy-hard-easy pattern as the training degree grows. This is characteristic: for other scaling factors N the same pattern emerges. Indeed the pattern is easily explained and makes sense. Little-trained NN policies tend to crash quickly and thus are easy to analyze. Strongly trained policies tend to reach the goal reliably with little variance, again resulting in high statistical confidence after relatively few sample runs. The hard cases lie in the middle where the NN policy exhibits high variance between runs that crash and ones that reach the goal, necessitating more analysis effort.

To corroborate these findings, let us have a closer look at the dependency between policy quality and DSMC runtime.

[2] The policy implements a simple reactive controller that brakes if a wall is near and otherwise accelerates toward the goal. Its goal probability is moderately worse than that of the best NN policies.

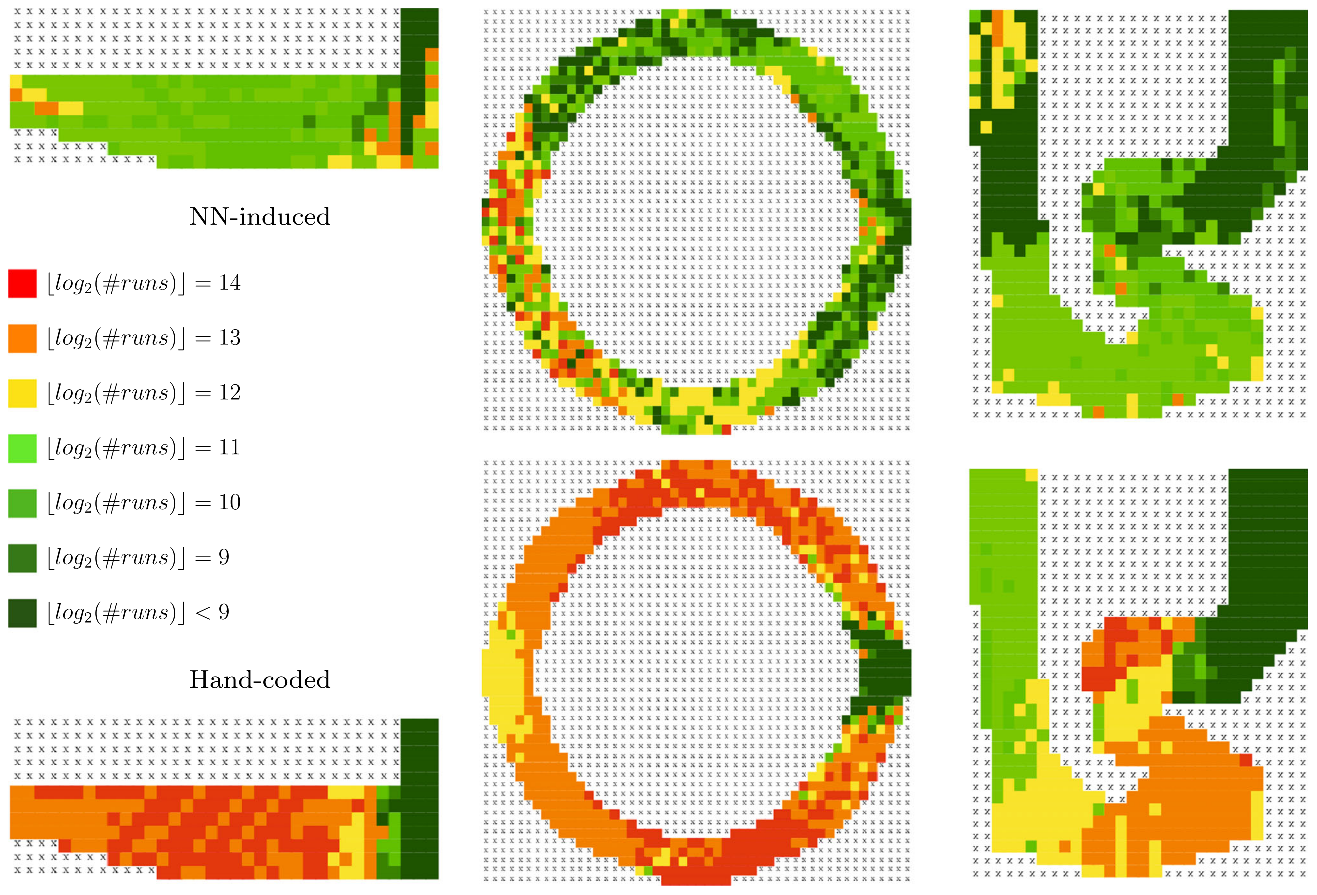

Fig. 7 Heat maps showing computational effort needed by DSMC, measured by the number of sample runs performed by MODES to analyze goal probability for each map location. Results shown for the policies induced by our learnt NN in the top row, versus a simple hand-coded policy (see text) at the bottom. Each point on the map shows $\lfloor \log_2(\#runs) \rfloor$

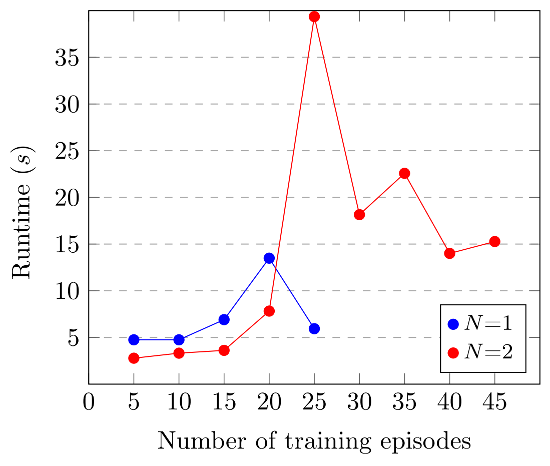

Fig. 8 Average runtime of DSMC per map cell, over training episodes

Fixing $N = 3$, we examine two NN policies σ_{bad} and σ_{good} of different quality, analyzing their goal probability and DSMC runtime locally, specific to different regions of the map in difference to the global analysis afforded by Fig. 8. Figure 9 shows the data.

In Fig. 9a, b we depict, for two different policies σ_{bad} and σ_{good}, for each map cell the goal probability when starting the policy from that cell with an initial velocity of 0. This goal probability was determined by running DSMC on the respective MDP state. In Fig. 9c, we depict the difference in runtime between (a) and (b), namely the quotient of DSMC runtime for σ_{bad} over DSMC runtime for σ_{good} on a cell-by-cell basis. Briefly put, dark green to yellow colors mean that DSMC on σ_{bad} takes less time than DSMC on σ_{good}, orange to light red means that both are analyzed in similar runtime, darker red to blue means that σ_{bad} takes more time to analyze up to a factor of > 10. The exact color-coding legend is given as part of Fig. 12.

The heat maps clearly show the effect of local policy quality on DSMC runtime. Near the starting line, where σ_{bad} typically does not reach the goal, σ_{bad} is much easier to analyze than σ_{good}. This changes drastically in the first curve of the track, where σ_{bad} exhibits high variance and becomes

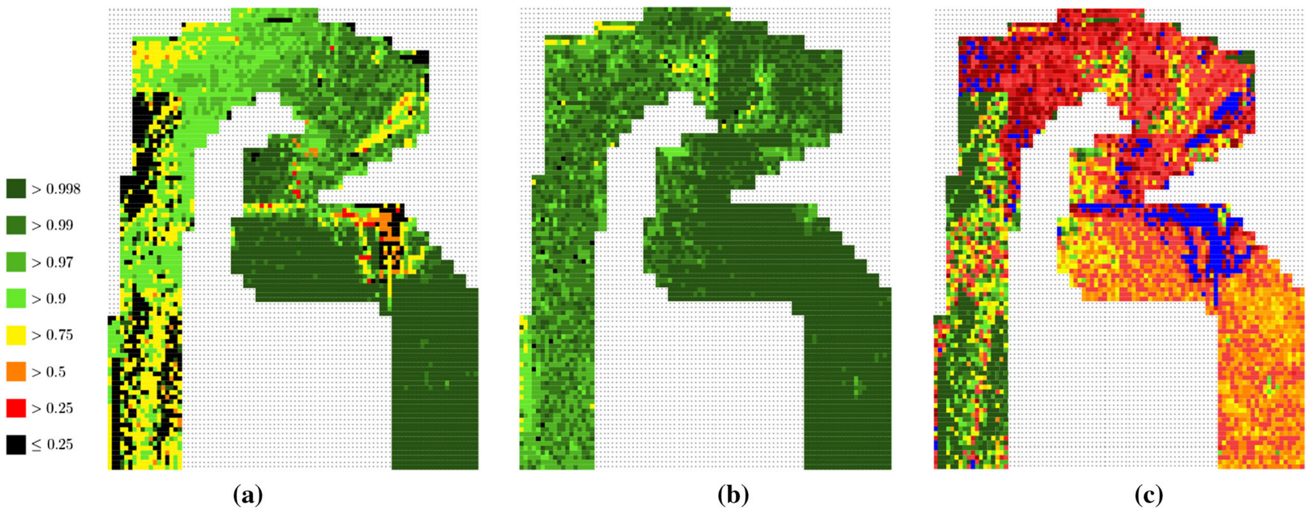

Fig. 9 **a**, **b** Goal probability per cell for $N = 3$ with a bad-quality NN policy σ_{bad} (**a**) versus a good-quality NN policy σ_{good} (**b**). **c** DSMC runtime difference quotient $\frac{\sigma_{\text{bad}}}{\sigma_{\text{good}}}$ per cell; color coding same as in Fig. 12, please see the legend there

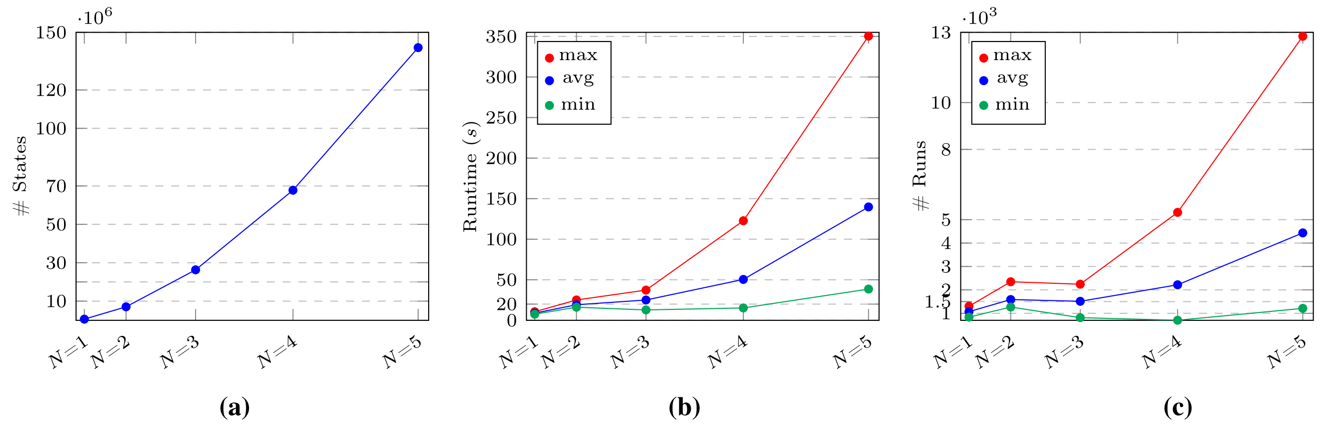

Fig. 10 **a** Total number of states in the MDP; **b** runtime of DSMC per map cell; **c** number of runs in DSMC per map cell. Each shown as a function of map size. **b** and **c** show min/average/max over 5 policies

much harder to analyze than σ_{good}. As we move closer to the goal, this latter phenomenon gradually diminishes, except for the last curve which σ_{bad} frequently fails to navigate successfully resulting in higher DSMC runtimes.

7.4 Scalability as a function of instance size

We now examine DSMC scalability as a function of instance size. Given the above insights, in this study, we only compare NN policies of similar global quality, as measured by the training return they achieve. We mainly focus on strongly trained policies, where DSMC serves for quality assurance. To account for variance in local policy quality (which is impossible to avoid), we train and analyze 5 different NN policies for each value of N.

Figure 10a displays the size of the MDP state space (number of states) to be considered by the analysis. The plots in (b) and (c) present our main scalability result as functions of the map size, in terms of (b) average DSMC runtime per map cell (initialized with velocity zero) and (c) average number of sample runs per map cell. We detail these results for the most demanding policy (max) and for the easiest policy (min) at each scale, together with the average (avg). Averaging over all cells factors out complexity source (i) from above which is a trivial phenomenon here due to our complete coverage of cells on the track.

The model size shown in (a) indicates that the MDPs analyzed are quite non-trivial, with millions of states already for $N = 1$ and $N = 2$, and going up to almost 150 million states for $N = 5$. Against this background, (b) clearly shows that the effort needed by DSMC increases linearly as a function of map size. This is corroborated by (c) which shows that the required number of sample runs barely has any tendency to increase with increasing map size at all; the scaling curve is

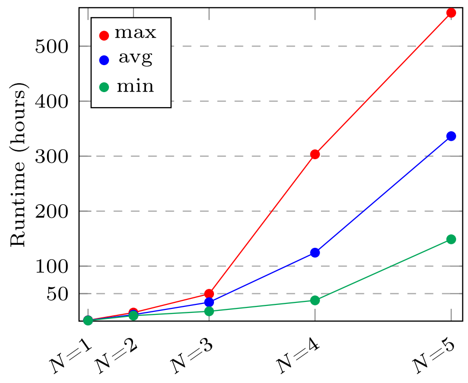

Fig. 11 Accumulated runtime of DSMC over whole map as function of map size; max, min, avg over 5 policies

dominated instead by the amount of variance across different policies.

We also ran these scalability experiments with lesser training, choosing low/middle quality policies following [16] as ones that deliver 20% (50%) of the maximal achieved return. The results are similar to the above in terms of the scaling behavior over N, so we do not repeat Figs. 10 and 11 for those settings. In terms of scaling over training degree as discussed in the previous section, low-quality policies are much easier to analyze, as expected. For middle-quality policies, the results are less conclusive, with DSMC effort roughly similar to high-quality policies but with more variance; we conclude from this that the hard region as displayed in Fig. 8 tends to be narrow, and correlate only loosely with policy return.

Together, these findings indicate that DSMC can be scalable in non-trivial application scenarios. The data confirm the expected result that, all other circumstances being equal, run length is the determining factor for DSMC performance, and thus the advantages of statistical model checking carry over to DSMC.

The accumulated effort for DSMC across all map cells grows substantially as a function of N, see Fig. 11, simply due to map size. This illustrates that an exhaustive analysis of the state space is highly demanding in these benchmarks. Note though that this task is trivial to parallelize, so that it can still be feasible to check large fractions of the state space. Indeed this was exploited in our experimental setup, running on a cluster of multicores.

Figure 12 provides a fine-grained view of differences in DSMC performance as a function of scaling size, comparing $N = 1$ versus $N = 2$ (left) and $N = 2$ versus $N = 5$ (right). Each cell in the heat maps shows the quotient of DSMC runtime of the smaller map over the larger map. Map cells are aligned across different map sizes according to their positions in the respective discretization.

In both heat maps, "strong" colors are rare, i.e., there is only little dark green and dark red/blue. The runtime differences hence are mostly not extreme, corroborating our observations from Fig. 10. There is however a certain degree of variation, which turns out to again be mostly caused by policy quality differences.

To understand this, consider first the left-hand side heat map. Near the start line and goal of the track, orange and yellow dominate—indicating similar runtimes—because DSMC analysis for both values of N tends to be quick. This is different in the remaining middle part of the track, where there is more policy-success variance, and hence more sample runs are needed, for both values of N. The smaller map size for $N = 1$ then results in significantly smaller runtimes.

In the right-hand side heat map, the picture is not as clear. Differences are again small close to the goal (light green this time as the size gap from $N = 2$ to $N = 3$ is larger), but elsewhere the picture is very mixed. The latter is due to local policy-quality variation, which is more pronounced in the larger maps. All the areas with distinctly large performance differences (e.g., the dark green stripe in the last curve) are due to poor quality of one of the two policies.

8 Conclusion

NNs are an increasingly widespread decision-making component in intelligent systems. Verifying the overall behavior of systems incorporating such components remains a grand challenge. When such a network is integrated into a control loop, the verification needs to intertwine controller and network verification [16].

Deep statistical model checking is a promising approach to address this challenge, leveraging the strength of statistical model checking as a light-weight approach for the purpose of checking the behavior of systems incorporating neural networks treated as black-box functions that merely need to be called not analyzed.

The most important aspects of the DSMC approach are its (i) genericity—in that it provides a generic and scalable basis for analyzing learnt action policies; its (ii) openness—since the approach is put into practice using the JANI format, supported by many tools for probabilistic or statistical model checking; and its (iii) focus—on an abstract fragment of the "autonomous driving" challenge. We consider these contributions as a conceptual nucleus of broader activities to foster the scientific understanding of neural network efficacy, by providing the formal and technological framework for precise, yet scalable problem analysis.

From a general perspective, DSMC provides a refined form of SMC for MDPs where thus far only implicitly defined random action policies have been available. If those were applied to Racetrack, goal probabilities < 0.1 would result—except directly at the goal line. DSMC instead can harvest available data for a far better suited action policy, in the form of an NN oracle trained on the data at hand. Of course, other

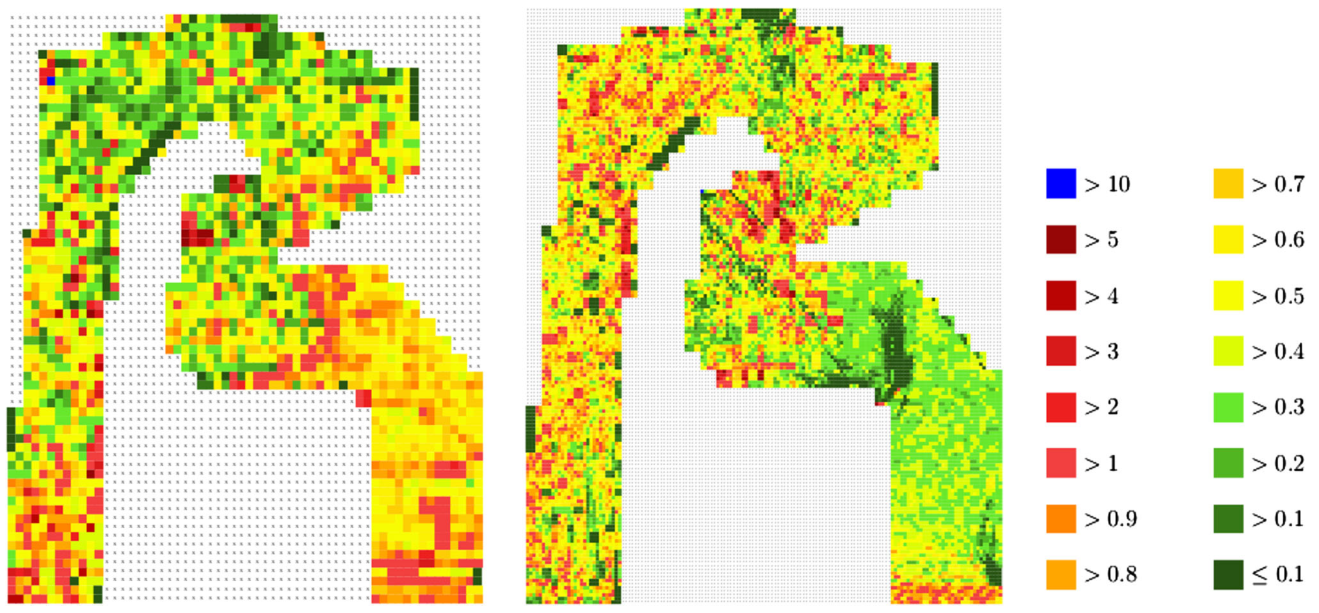

Fig. 12 DSMC runtime difference quotient $\frac{\text{small-map}}{\text{large-map}}$ per cell for $N = 1$ versus $N = 2$ (left) and $N = 2$ versus $N = 5$ (right)

forms of oracles (based on, e.g., random forests) can be considered with DSMC right away, too.

In addition to the initial case study of *DSMC20* suggesting that the approach may indeed be useful and feasible, we have contributed new evidence that DSMC can be scalable. The advantages of statistical model checking are inherited in our study, exhibiting a linear runtime increase per state as a function of instance size. We have furthermore shown that there are significant interactions between policy quality and analysis performance, which become important when using DSMC during the training process (e.g., to identify weak-quality regions for re-training) [32].

Note also that the DSMC approach is highly parallelizable in terms of all its major activities, (i) statistical model checking (independent sample runs), (ii) neural network evaluation (GPU/TPU hardware), and (iii) sweeping a state space partition (trivial). So, by effectively leveraging large amounts of hardware, there is some hope that large scalability challenges can be tackled.

We hope that the study provides a compelling basis for further research on deep statistical model checking.

Racetrack forms a viable starting point for this endeavor in that it can be made more realistic in a manifold of dimensions: car configurations regarding speed and acceleration limits, fuel efficiency, different surface conditions [8], appearing/disappearing obstacles, other traffic participants, speed limits and other traffic regulations, different probabilistic perturbances, change from map perspective to ego-perspective of an autonomous vehicle, mediated by vision and other sensor systems. We are actually embarking on an exploration of these dimensions, focussing first on speed limits and random obstacles.

Our Racetrack case study makes it easy to produce "heat maps", as a meaningful way to represent a partitioned perspective on the state space and sampling one member state from each set as a representative. With the TraceVis tool, we also showed how visualization techniques in 3D can help to get even more insights from the DSMC results and to display more information than in the simple heat maps [26,28]. We believe that such a representative analysis makes sense (e.g., to provide an overview for human users) in many application scenarios. An open question is how to partition states, or how to support users in doing so; physical location might work in many cases.

Apart from the extension of our study to more general Racetrack maps and to examples with larger state spaces, an important scaling dimension yet to be evaluated is NN complexity. In particular, convolutional networks from computer vision are of interest, in a context where the policy inputs are images. Such an architecture is possible in principle, but would require an extension of DSMC to incorporate a model-to-NN adapter producing (or approximating) the image based on the MDP state.

In the MDPs considered so far, we always assumed scenarios with perfect knowledge and full observability. It would be worth investigating how DSMC can be applied to POMDP scenarios.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was partially supported by the ERC Advanced Investigators Grant 695614 (POWVER), by the German Research Foundation (DFG) under Grant No. 389792660, as part of TRR 248 – CPEC, see https://perspicuous-computing.science, by the Key-Area Research and Development Program Grant 2018B010107004 of Guangdong Province, and by the European Regional Development Fund (ERDF).

Data availability The benchmark and all infrastructure including our modification of MODES as well as our JANI model are archived and publicly available at DOI 10.5281/zenodo.3760098 [31]. The infrastructure for the scalability study is available at DOI 10.5281/zenodo.7071405.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Agostinelli, F., McAleer, S., Shmakov, A., Baldi, P.: Solving the Rubik's cube with deep reinforcement learning and search. Nat. Mach. Intell. **1**(8), 356–363 (2019)
2. Akintunde, M., Lomuscio, A., Maganti, L., Pirovano, E.: Reachability analysis for neural agent-environment systems. In: Thielscher, M., Toni, F., Wolter, F. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October–2 November 2018, pp. 184–193. AAAI Press (2018). https://aaai.org/ocs/index.php/KR/KR18/paper/view/17991
3. Akintunde, M.E., Kevorchian, A., Lomuscio, A., Pirovano, E.: Verification of RNN-based neural agent-environment systems. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27–February 1, 2019, pp. 6006–6013. AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33016006
4. Alamdari, P.A., Avni, G., Henzinger, T.A., Lukina, A.: Formal methods with a touch of magic. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21–24, 2020, pp. 138–147. IEEE (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_21
5. Ashok, P., Kretínský, J., Larsen, K.G., Coënt, A.L., Taankvist, J.H., Weininger, M.: SOS: safe, optimal and small strategies for hybrid Markov decision processes. In: Parker, D., Wolf, V. (eds.) Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10–12, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11785, pp. 147–164. Springer (2019). https://doi.org/10.1007/978-3-030-30281-8_9
6. Avni, G., Bloem, R., Chatterjee, K., Henzinger, T.A., Könighofer, B., Pranger, S.: Run-time optimization for learned controllers through quantitative games. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification—31st International Conference, CAV 2019, New York City, NY, USA, July 15–18, 2019, Proceedings, Part I, Lecture Notes in Computer Science, vol. 11561, pp. 630–649. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_36
7. Baier, C., Christakis, M., Gros, T.P., Groß, D., Gumhold, S., Hermanns, H., Hoffmann, J., Klauck, M.: Lab conditions for research on explainable automated decisions. In: Proceedings of the 1st TAILOR Workshop—Foundations of Trustworthy AI—Integrating Learning, Optimization and Reasoning Co-Located with 24th European Conference on Artificial Intelligence, TAILOR 2020, Santiago de Compostela, Spain (2020)
8. Baier, C., Dubslaff, C., Hermanns, H., Klauck, M., Klüppelholz, S., Köhl, M.A.: Components in probabilistic systems: Suitable by construction. In: Margaria,T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles—9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part I, Lecture Notes in Computer Science, vol. 12476, pp. 240–261. Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_13
9. Barto, A.G., Bradtke, S.J., Singh, S.P.: Learning to act using real-time dynamic programming. Artif. Intell. **72**(1–2), 81–138 (1995)
10. Bogdoll, J., Fioriti, L.M.F., Hartmanns, A., Hermanns, H.: Partial order methods for statistical model checking and simulation. In: FMOODS-FORTE, LNCS 6722, pp. 59–74 (2011)
11. Bonet, B., Geffner, H.: Labeled RTDP: improving the convergence of real-time dynamic programming. In: ICAPS, pp. 12–21 (2003)
12. Buchholz, P.: Exact and ordinary lumpability in finite Markov chains. J. Appl. Probab. **31**(1), 59–75 (1994)
13. Budde, C.E., D'Argenio, P.R., Hartmanns, A., Sedwards, S.: A statistical model checker for nondeterminism and rare events. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems—24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings, Part II, Lecture Notes in Computer Science, vol. 10806, pp. 340–358. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_20
14. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: Quantitative model and tool interaction. In: TACAS (2), LNCS 10206, pp. 151–168 (2017)
15. Carr, S., Jansen, N., Wimmer, R., Serban, A.C., Becker, B., Topcu, U.: Counterexample-guided strategy improvement for pomdps using recurrent neural networks. In: Kraus, S. (ed.) Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10–16, 2019, pp. 5532–5539. ijcai.org (2019). https://doi.org/10.24963/ijcai.2019/768
16. Christakis, M., Eniser, H.F., Hermanns, H., Hoffmann, J., Kothari, Y., Li, J., Navas, J.A., Wüstholz, V.: Automated safety verification of programs invoking neural networks. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification—33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I, Lecture Notes in Computer Science, vol. 12759, pp. 201–224. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_9
17. Croce, F., Andriushchenko, M., Hein, M.: Provable robustness of ReLU networks via maximization of linear regions. In: AISTATS, PMLR 89, pp. 2057–2066 (2019)
18. D'Argenio, P.R., Hartmanns, A., Legay, A., Sedwards, S.: Statistical approximation of optimal schedulers for probabilistic timed automata. In: IFM, LNCS 9681, pp. 99–114 (2016)
19. David, A., Jensen, P.G., Larsen, K.G., Mikucionis, M., Taankvist, J.H.: Uppaal stratego. In: Baier, C., Tinelli, C. (eds.) Tools and

Algorithms for the Construction and Analysis of Systems—21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11–18, 2015. Proceedings, Lecture Notes in Computer Science, vol. 9035, pp. 206–211. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_16
20. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Wang, Z.: Time for statistical model checking of real-time systems. In: CAV, LNCS 6806, pp. 349–355 (2011)
21. Dehnert, C., Junges, S., Katoen, J., Volk, M.: A storm is coming: A modern probabilistic model checker. In: CAV, LNCS 10427, pp. 592–600 (2017)
22. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: ATVA, LNCS 10482, pp. 269–286 (2017)
23. Gardner, M.: Mathematical games. Sci. Am. **229**, 118–121 (1973)
24. Gardner, M., Dorling, S.: Artificial neural networks (the multilayer perceptron)-a review of applications in the atmospheric sciences. Atmos. Environ. **32**(14), 2627–2636 (1998)
25. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: Safety and robustness certification of neural networks with abstract interpretation. In: IEEE Symposium on Security and Privacy 2018, pp. 3–18 (2018)
26. Groß, D., Klauck, M., Gros, T.P., Steinmetz, M., Hoffmann, J., Gumhold, S.: Glyph-based visual analysis of q-learning based action policy ensembles on racetrack. In: 26th International Conference on Information Visualisation (IV) (2022)
27. Gros, T.P.: Tracking the race: Analyzing racetrack agents trained with imitation learning and deep reinforcement learning. Master's thesis, Saarland University (2021)
28. Gros, T.P., Groß, D., Gumhold, S., Hoffmann, J., Klauck, M., Steinmetz, M.: TraceVis: Towards Visualization for Deep Statistical Model Checking. In: Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. From Verification to Explanation. (2020)
29. Gros, T.P., Hermanns, H., Hoffmann, J., Klauck, M., Köhl, M.A., Wolf, V.: Mogym: Using formal models for training and verifying decision-making agents. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification—34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part II, Lecture Notes in Computer Science, vol. 13372, pp. 430–443. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_21
30. Gros, T.P., Hermanns, H., Hoffmann, J., Klauck, M., Steinmetz, M.: Deep Statistical Model Checking In: Proceedings of the 40th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'20) (2020). Available at https://doi.org/10.1007/978-3-030-50086-3_6
31. Gros, T.P., Hermanns, H., Hoffmann, J., Klauck, M., Steinmetz, M.: Models and Infrastructure used in Deep Statistical Model Checking (2020). Available at https://doi.org/10.5281/zenodo.3760098
32. Gros, T.P., Höller, D., Hoffmann, J., Klauck, M., Meerkamp, H., Wolf, V.: DSMC evaluation stages: Fostering robust and safe behavior in deep reinforcement learning. In: Abate, A., Marin, A. (eds.) Quantitative Evaluation of Systems—18th International Conference, QEST 2021, Paris, France, August 23–27, 2021, Proceedings, Lecture Notes in Computer Science, vol. 12846, pp. 197–216. Springer (2021). https://doi.org/10.1007/978-3-030-85172-9_11
33. Gros, T.P., Höller, D., Hoffmann, J., Wolf, V.: Tracking the race between deep reinforcement learning and imitation learning. In: International Conference on Quantitative Evaluation of Systems, pp. 11–17. Springer (2020)
34. Haesaert, S., Soudjani, S., Abate, A.: Temporal logic control of general markov decision processes by approximate policy refinement. In: Abate, A., Girard, A., Heemels, M. (eds.) 6th IFAC Conference on Analysis and Design of Hybrid Systems, ADHS 2018, Oxford, UK, July 11–13, 2018, IFAC-PapersOnLine, vol. 51, pp. 73–78. Elsevier (2018). https://doi.org/10.1016/j.ifacol.2018.08.013
35. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasMc: A web-based probabilistic model checker. In: FM 2014, LNCS 8442, pp. 312–317 (2014)
36. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Form. Asp. Comput. **6**(5), 512–535 (1994)
37. Hartmanns, A.: On the analysis of stochastic timed systems. Ph.D. thesis, Saarland University, Germany (2015)
38. Hartmanns, A., Hermanns, H.: The Modest toolset: An integrated environment for quantitative modelling and verification. In: TACAS, LNCS 8413, pp. 593–598 (2014)
39. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: TACAS (1), LNCS 11427, pp. 344–350 (2019)
40. Hasanbeig, M., Abate, A., Kroening, D.: Logically-correct reinforcement learning. CoRR (2018). arxiv:1801.08099
41. Hausknecht, M.J., Stone, P.: Deep recurrent q-learning for partially observable MDPs. In: 2015 AAAI Fall Symposia, Arlington, Virginia, USA, November 12–14, 2015, pp. 29–37. AAAI Press (2015). http://www.aaai.org/ocs/index.php/FSS/FSS15/paper/view/11673
42. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: VMCAI, LNCS 2937, pp. 73–84 (2004)
43. Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T.N., Kingsbury, B.: Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. IEEE Signal Process. Mag. **29**(6), 82–97 (2012)
44. Hornik, K., Stinchcombe, M.B., White, H.: Multilayer feedforward networks are universal approximators. Neural Netw. **2**, 359–366 (1989)
45. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: CAV (1), LNCS 10426, pp. 3–29 (2017)
46. Jaeger, M., Jensen, P.G., Larsen, K.G., Legay, A., Sedwards, S., Taankvist, J.H.: Teaching stratego to play ball: Optimal synthesis for continuous space MDPs. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis—17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28–31, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11781, pp. 81–97. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_5
47. The JANI specification. http://www.jani-spec.org/. Accessed on 28/02/2020
48. Junges, S., Jansen, N., Dehnert, C., Topcu, U., Katoen, J.: Safety-constrained reinforcement learning for MDPs. In: Chechik, M., Raskin, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems—22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, Lecture Notes in Computer Science, vol. 9636, pp. 130–146. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_8
49. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: CAV (1), LNCS 10426, pp. 97–117 (2017)
50. Klauck, M., Steinmetz, M., Hoffmann, J., Hermanns, H.: Compiling probabilistic model checking into probabilistic planning. In: ICAPS, pp. 150–154 (2018)
51. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: NIPS, pp. 1097–1105 (2012)
52. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV, LNCS 6806, pp. 585–591 (2011)

53. Kwiatkowska, M.Z., Norman, G., Parker, D.: Stochastic model checking. In: SFM 2007, Advanced Lectures, LNCS 4486, pp. 220–270 (2007)
54. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. **94**(1), 1–28 (1991). https://doi.org/10.1016/0890-5401(91)90030-6
55. Legay, A., Lukina, A., Traonouez, L., Yang, J., Smolka, S.A., Grosu, R.: Statistical model checking. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science—State of the Art and Perspectives, Lecture Notes in Computer Science, vol. 10000, pp. 478–504. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_23
56. Legay, A., Sedwards, S., Traonouez, L.: Scalable verification of Markov decision processes. In: SEFM Workshops, LNCS 8938, pp. 350–362 (2014)
57. Li, J., Liu, J., Yang, P., Chen, L., Huang, X., Zhang, L.: Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In: SAS, LNCS 11822, pp. 296–319 (2019)
58. McMahan, H.B., Gordon, G.J.: Fast exact planning in Markov decision processes. In: ICAPS, pp. 151–160 (2005)
59. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M.A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**, 529–533 (2015)
60. Okamoto, M.: Some inequalities relating to the partial sum of binomial probabilities. Ann. Inst. Stat. Math. **10**(1), 29–35 (1959)
61. Parker, D.A.: Implementation of symbolic model checking for probabilistic systems. Ph.D. thesis, University of Birmingham, UK (2003)
62. Pineda, L.E., Lu, Y., Zilberstein, S., Goldman, C.V.: Fault-tolerant planning under uncertainty. In: IJCAI, pp. 2350–2356 (2013)
63. Pineda, L.E., Zilberstein, S.: Planning under uncertainty using reduced models: Revisiting determinization. In: ICAPS (2014)
64. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, Hoboken (1994)
65. Sarle, W.S.: Neural networks and statistical models (1994)
66. Schilling, C., Forets, M., Guadalupe, S.: Verification of neural-network control systems by integrating Taylor models and zonotopes. In: Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22–March 1, 2022, pp. 8169–8177. AAAI Press (2022). https://ojs.aaai.org/index.php/AAAI/article/view/20790
67. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)
68. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction In: Adaptive Computation and Machine Learning, 2nd edn. The MIT Press (2018)
69. Wald, A.: Sequential tests of statistical hypotheses. Ann. Math. Stat. **16**(2), 117–186 (1945)
70. Wicker, M., Huang, X., Kwiatkowska, M.: Feature-guided black-box safety testing of deep neural networks. In: TACAS (1), LNCS 10805, pp. 408–426 (2018)
71. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: CAV, LNCS 2404, pp. 223–235 (2002)