Neural Network Action Policy Verification via Predicate Abstraction

Marcel Vinzent,[1] Marcel Steinmetz,[1] Jörg Hoffmann[1,2]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
{vinzent, steinmetz, hoffmann}@cs.uni-saarland.de

Abstract

Neural networks (NN) are an increasingly important representation of action policies. Verifying that such policies are safe is potentially very hard as it compounds the state space explosion with the difficulty of analyzing even single NN decision episodes. Here we address that challenge through abstract reachability analysis. We show how to compute predicate abstractions of the policy state space subgraph induced by fixing an NN action policy. A key sub-problem here is the computation of abstract state transitions that may be taken by the policy, which as we show can be tackled by connecting to off-the-shelf SMT solvers. We devise a range of algorithmic enhancements, leveraging relaxed tests to avoid costly calls to SMT. We empirically evaluate the resulting machinery on a collection of benchmarks. The results show that our enhancements are required for practicality, and that our approach can outperform two competing approaches based on explicit enumeration and bounded-length verification.

1 Introduction

Neural networks (NN) are an increasingly important representation of action policies, in particular in planning (Issakkimuthu, Fern, and Tadepalli 2018; Groshev et al. 2018; Garg, Bajpai, and Mausam 2019; Toyer et al. 2020). But how to verify that such a policy is safe?

While there has been remarkable progress on analyzing individual NN decision episodes (Katz et al. 2017, 2019; Huang et al. 2017; Gehr et al. 2018; Li et al. 2019), the verification of NN decision sequences is still in its early stages. The most prominent line of works addresses neural controllers of dynamical systems, where the NN outputs a vector u of reals forming input to a continuous state-evolution function f. This has been investigated for linear f (Sun, Khedr, and Shoukry 2019; Tran et al. 2019) as well as for Lipschitz continuous f (Huang et al. 2019; Dutta, Chen, and Sankaranarayanan 2019). Recent work extends this thread to hybrid systems, addressing smooth (tanh/sigmoid) activation functions by compilation into such systems (Ivanov et al. 2021). In a context closer to AI sequential decision making, but still considering NN controllers influencing a linear state-evolution function, the use of MIP encodings for safety verification has been explored (Akintunde et al. 2018, 2019). Here we explore a context and method complementary to all those, namely NN policies π with ReLU activation functions taking discrete action choices in sequential decision making, and the extension of **predicate abstraction (PA)** (Graf and Saïdi 1997; Ball et al. 2001; Henzinger et al. 2004) for verifying the safety of such π.

We tackle non-deterministic state spaces over bounded-integer state variables. Given a policy π, a **start condition** ϕ_0, and an **unsafety condition** ϕ_U, we verify whether a state $s_U \models \phi_U$ is reachable from a state $s_0 \models \phi_0$ under π. We do so by building an abstraction defined through a set $\mathcal{P}$ of **predicates**, where each $p \in \mathcal{P}$ is a linear constraint over the state variables (e.g. $x = 7$ or $x \leq y$). Abstract states are characterized by truth value assignments to $\mathcal{P}$, grouping together all concrete states that result in the same truth values. Like in other abstraction methods (e.g. underlying heuristic functions (Edelkamp 2001; Helmert et al. 2014; Seipp and Helmert 2018)), transitions are over-approximated to preserve all possible behaviors. However, we abstract not the full state space Θ, but the **policy-restricted** state space Θ^π, i.e., the state-space subgraph containing only the transitions taken by π. We refer to the predicate abstraction of Θ^π as the **policy predicate abstraction (PPA)** $\Theta^\pi_\mathcal{P}$. We build the fragment of $\Theta^\pi_\mathcal{P}$ reachable from ϕ_0, and check whether ϕ_U is reached. If this is not the case then π is safe.

To compute the PA $\Theta_\mathcal{P}$, one frequently needs to solve the sub-problem of deciding whether there is a transition from abstract state A to abstract state A': *does there exist a state $s \in A$ and an action a s.t. executing a in s results in $s' \in A'$?* This satisfiability problem is routinely addressed using SMT solvers such as $Z3$ (de Moura and Bjørner 2008). To compute the PPA $\Theta^\pi_\mathcal{P}$ however, we additionally need to check whether $\pi(s) = a$, i.e., whether the policy actually selects a on s. This is still an SMT problem: one can encode the entire NN as a conjunction of constraints – one for every neuron – and add those to the SMT encoding. But scalability of course becomes an issue as these SMT encodings can get large.

We devise a range of algorithmic enhancements to address this, using relaxed tests to avoid costly calls to SMT. Most importantly, continuous relaxation of the state variables allows to leverage recent SMT solvers specialized to NN with ReLU activation functions (Katz et al. 2017, 2019). We devise a method that simplifies exact-SMT tests via information obtained on relaxed tests, and a method using branch-and-bound around relaxed tests to avoid exact tests altogether. While these enhancements are conceptually straightforward (and effective PA though not PPA has been intensively explored (Cimatti et al. 2009; Cavada et al. 2014)), our overall machinery constitutes a substantial engineering effort. We contribute that effort in terms of an implementation based on the automata language JANI (Budde et al. 2017). Our tool (and all experiments) are publicly available.[1]

We run experiments on a collection of benchmarks, consisting of Racetrack, Blocksworld, SlidingTiles, and a simple Transport domain. We adapted the latter three of these to include non-determinism and an unsafety condition. We do not automate the selection of abstraction predicates yet, instead providing these as input and scaling them as an important algorithm parameter in our experiments. As competing approaches, we implement a naïve approach explicitly enumerating all states the policy can reach, as well as a bounded-length verification approach following the ideas of Akintunde et al. (2018; 2019). Our results show that our algorithmic enhancements are required for practicality, and that our approach can outperform its competitors.[2]

2 State Space Representation

The particular language JANI we use in our implementation is not relevant to understanding our contribution. We hence abstract from the language to a generic representation of non-deterministic state spaces, as follows.

A state space is a tuple $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$ of **state variables** $\mathcal{V}$, **action labels** $\mathcal{L}$, and **operators** $\mathcal{O}$. For each variable $v \in \mathcal{V}$ the domain D_v is a non-empty bounded integer interval. We denote by Exp the set of **linear integer expressions** over $\mathcal{V}$, i.e., expressions of the form $d_1 \cdot v_1 + \cdots + d_r \cdot v_r + c$ with $d_1, \dots, d_r, c \in \mathbb{Z}$. C denotes the set of **linear integer constraints** over $\mathcal{V}$, i.e., constraints of the form $e_1 \bowtie e_2$ with $\bowtie \in \{\leq, =, \geq\}$ and $e_1, e_2 \in Exp$, and all Boolean combinations thereof. An **operator** $o \in \mathcal{O}$ is a tuple (g, l, u) with **label** $l \in \mathcal{L}$, **guard** $g \in C$, and **update** $u\colon \mathcal{V} \to Exp$.

A (partial) **variable assignment** s over $\mathcal{V}$ is a function with domain $dom(s) \subseteq \mathcal{V}$ and $s(v) \in D_v$ for $v \in dom(s)$. Given s_1, s_2, we denote by $s_1[s_2]$ the update of s_1 by s_2, i.e., $dom(s_1[s_2]) = dom(s_1) \cup dom(s_2)$ with $s_1[s_2](v) = s_2(v)$ if $v \in dom(s_2)$, else $s_1[s_2](v) = s_1(v)$. By $e(s)$ we denote the evaluation of $e \in Exp$ over s, and by $\phi(s)$ the evaluation of $\phi \in C$. If $\phi(s)$ evaluates to true, we write $s \models \phi$.

The **state space** of $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$ is a labeled transition system (LTS) $\Theta = \langle \mathcal{S}, \mathcal{L}, \mathcal{T} \rangle$. The set of **states** $\mathcal{S}$ is the (finite) set of all complete variable assignments over $\mathcal{V}$. The set of **transitions** $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ contains (s, l, s') iff there exists an operator $o = (g, l, u)$ such that $s \models g$ and $s' = s[u(s)]$, i.e., the guard is satisfied in the source state s, and the successor state s' results from applying the update to s. Here, $u(s)$ denotes the partial variable assignment induced by u evaluated over s, i.e., $u(s) = \{v \mapsto u(v)(s) \mid v \in dom(u)\}$.We also write $s \models o$ for $s \models g$, and abbreviate $s[\![o]\!]$ for $s[u(s)]$.

[1]https://fai.cs.uni-saarland.de/vinzent/downloads/icaps22.zip

[2]We skip many formal details here. These are available in an online TR at https://fai.cs.uni-saarland.de/vinzent/papers/icaps22-tr.pdf, which also discusses related work in more detail.

From an AI Planning perspective, the only unusual aspect here (reflecting JANI/automata languages) is the separation between action labels and operators. This is useful because it supports both, state-dependent effects (different operators with the same label l applicable in different states); as well as action outcome non-determinism (different operators with the same label l applicable in the same state).

3 NN Action Policies

An **action policy** π is a function $\mathcal{S} \to \mathcal{L}$. The **policy-restricted state space** Θ^π is the subgraph $\langle \mathcal{S}, \mathcal{L}, \mathcal{T}^\pi \rangle$ of Θ with $\mathcal{T}^\pi = \{(s, l, s') \in \mathcal{T} \mid \pi(s) = l\}$.

Note that we allow π to select inapplicable actions, i.e., there may be $s \in \mathcal{S}$ for which $\pi(s)$ does not label any outgoing transition. In this case, the policy execution stops. Two remarks are in order here: (1) the possibility of the policy getting stuck raises the issue of deadlock verification (as known from concurrent systems); (2) a popular practical trick is to super-impose applicability on π, letting it select only from the applicable actions. Our approach can in principle be adapted to perform deadlock verification or to super-impose applicability. Both require substantially more complex SMT encodings though, resulting in serious computational challenges that future work needs to address.

We consider action policies represented by **neural networks** (NN), specifically fully connected feed-forward NN. These consist of an input layer with an input for each state variable; arbitrarily many hidden layers; and an output layer with an output for each action. The policy π is obtained by applying argmax to the output layer. Our approach is, in principle, agnostic to the activation functions used. In our current implementation we leverage SMT solvers specialized to **rectified linear units (ReLU)**, $ReLU(x) = \max(x, 0)$, so our experiments focus on those exclusively.

4 Safety Properties & Predicate Abstraction

We next review safety of systems in general, not considering a policy. We give a corresponding definition of safety, and give the background on predicate abstraction in this context.

Definition 1 (Safety Property)**.** A **safety property** is a pair $\rho = (\phi_0, \phi_U)$ with $\phi_0, \phi_U \in C$. ρ is *violated in* Θ iff there exist states $s_0, s_U \in \mathcal{S}$ such that $s_0 \models \phi_0$, $s_U \models \phi_U$, and s_U is reachable from s_0 in Θ. Θ is **unsafe** with respect to ρ if ρ is violated in Θ, and **safe** otherwise.

The **unsafety condition** ϕ_U identifies the set of **unsafe states** that should be unreachable from the set of possible **start states** S_0 represented by ϕ_0.

Predicate abstraction (Graf and Saïdi 1997) verifies safety within an abstract state space, as follows. Assume a set of predicates $\mathcal{P} \subseteq C$. An **abstract state** $s_\mathcal{P}$ is a (complete) truth value assignment over $\mathcal{P}$, also referred to as a **predicate state**. The abstraction of a (concrete) state $s \in \mathcal{S}$ is the predicate state $s|_\mathcal{P}$ with $s|_\mathcal{P}(p) = p(s)$ for each $p \in \mathcal{P}$. Conversely, $[s_\mathcal{P}] = \{s' \in \mathcal{S} \mid s'|_\mathcal{P} = s_\mathcal{P}\}$ denotes the concretization of predicate state $s_\mathcal{P}$, i.e., the set of all concrete state represented by $s_\mathcal{P}$. The abstract state space now is defined in a transition-preserving manner:

Definition 2 (Predicate Abstraction)**.** The **predicate abstraction** of Θ over $\mathcal{P}$ is the LTS $\Theta_{\mathcal{P}} = \langle \mathcal{S}_{\mathcal{P}}, \mathcal{L}, \mathcal{T}_{\mathcal{P}} \rangle$, where $\mathcal{S}_{\mathcal{P}}$ is the set of all predicates states over $\mathcal{P}$, and $\mathcal{T}_{\mathcal{P}} = \{(s|_{\mathcal{P}}, l, s'|_{\mathcal{P}}) \mid (s, l, s') \in \mathcal{T}\}$.

We say that a predicate state $s_{\mathcal{P}}$ satisfies a constraint $\phi \in C$, written $s_{\mathcal{P}} \models \phi$, iff there exists $s \in [s_{\mathcal{P}}]$ such that $s \models \phi$. Similarly, a safety property $\rho = (\phi_0, \phi_U)$ is violated in $\Theta_{\mathcal{P}}$ iff there exist $s_{\mathcal{P}}, s'_{\mathcal{P}} \in \mathcal{S}_{\mathcal{P}}$ with $s_{\mathcal{P}} \models \phi_0$, $s'_{\mathcal{P}} \models \phi_U$ and $s'_{\mathcal{P}}$ is reachable from $s_{\mathcal{P}}$ in $\Theta_{\mathcal{P}}$. Due to the over-approximating nature of $\Theta_{\mathcal{P}}$, safety in Θ can be proven via safety in $\Theta_{\mathcal{P}}$:

Proposition 3 (Safety in $\Theta_{\mathcal{P}}$)**.** Let ρ be a safety property. If $\Theta_{\mathcal{P}}$ is safe with respect to ρ, then so is Θ.

The computation of $\Theta_{\mathcal{P}}$ necessitates to solve a satisfiability problem for every possible abstract state transition: $(s_{\mathcal{P}}, l, s'_{\mathcal{P}}) \in \mathcal{T}_{\mathcal{P}}$ iff there exists an operator $o \in \mathcal{O}$ with label l and a concrete state $s \in [s_{\mathcal{P}}]$ such that $s \models o$ and $s[\![o]\!] \in s'_{\mathcal{P}}$. We denote this test by **TSat**$(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$.

TSat$(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ can be encoded into a satisfiability modulo theories (SMT) (Barrett et al. 1994) formula over the variables $\mathcal{V}$, including a primed and an unprimed form for each. The unprimed variables represent concrete states in $[s_{\mathcal{P}}]$, the primed ones $[s'_{\mathcal{P}}]$. Predicate state constraints enforce that the predicates over the (un)primed variables evaluate according to the truth values in $s_{\mathcal{P}}$ ($s'_{\mathcal{P}}$). Operator constraints ensure that the unprimed variables satisfy the guard of o, and the primed variables are consistent with the updates of o. For example, say we have state variables x, y each with range $[0, 5]$, $\mathcal{P} = \{p\}$ with $p = (x \geq y)$, $s_{\mathcal{P}} = (p \mapsto 1)$, $s'_{\mathcal{P}} = (p \mapsto 0)$, and o with guard $x \geq y$ and update $x := x - 1$. Then the encoding of TSat$(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ is the conjunction of $x \geq y$ [$s_{\mathcal{P}}$, guard]; $x' = x - 1$ [update]; $\neg(x' \geq y)$ [$s'_{\mathcal{P}}$]; and the bounding constraints $0 \leq x, y, x', y' \leq 5$. This is satisfiable (e.g. by $s(x) = s(y) = 1$), so there is an abstract state transition from $s_{\mathcal{P}}$ to $s'_{\mathcal{P}}$. A full specification of the SMT encoding is in the TR.

5 Policy Predicate Abstraction

We now extend the above concepts to policy verification. As we shall see, the definitions themselves transfer straightforwardly. What becomes substantially more complex is the satisfiability test needed to identify abstract state transitions.

Definition 4 (Policy Safety)**.** Let ρ be a safety property, and let π be a policy. π is safe with respect to ρ iff Θ^{π} is safe with respect to ρ.

In words, we apply the definition of safety (Definition 1) to the policy-restricted state space Θ^{π}. Predicate abstraction for policy verification is defined correspondingly:

Definition 5 (Policy Predicate Abstraction)**.** Let $\mathcal{P} \subseteq C$ be a predicate set, and let π be a policy. The **policy predicate abstraction** of Θ^{π} over $\mathcal{P}$ is the LTS $\Theta^{\pi}_{\mathcal{P}} = \langle \mathcal{S}_{\mathcal{P}}, \mathcal{L}, \mathcal{T}^{\pi}_{\mathcal{P}} \rangle$ where $\mathcal{T}^{\pi}_{\mathcal{P}} = \{(s|_{\mathcal{P}}, l, s'|_{\mathcal{P}}) \mid (s, l, s') \in \mathcal{T}, \pi(s) = l\}$.

Applying the same arguments as above, policy predicate abstraction yields a sufficient condition for policy safety:

Proposition 6 (Safety in $\Theta^{\pi}_{\mathcal{P}}$)**.** Let $\rho = (\phi_0, \phi_U)$ be a safety property. If $\Theta^{\pi}_{\mathcal{P}}$ is safe with respect to ρ, then so is π.

We compute the fragment of $\Theta^{\pi}_{\mathcal{P}}$ reachable from the **abstract start states** $S_0|_{\mathcal{P}} = \{s_{\mathcal{P}} \in \mathcal{S}_{\mathcal{P}} \mid s_{\mathcal{P}} \models \phi_0\}$. If that fragment does not contain any **abstract unsafe state** $s'_{\mathcal{P}} \models \phi_U$, then with Proposition 6 the policy is safe.[3]

The new source of complexity in computing abstract state transitions is that, in addition to the standard test TSat$(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$, we now need to check whether the policy π actually chooses l in the state $s \in [s_{\mathcal{P}}]$:

Definition 7 (Transition Test of $\Theta^{\pi}_{\mathcal{P}}$)**.** Let $s_{\mathcal{P}}, s'_{\mathcal{P}}$ be predicate states, and let $o = (g, l, u)$ be an operator. The **transition test of** $\Theta^{\pi}_{\mathcal{P}}$, denoted **TSat**$^{\pi}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$, is satisfied iff there exists $s \in [s_{\mathcal{P}}]$ s.t. $s \models o$, $s[\![o]\!] \in [s'_{\mathcal{P}}]$ and $\pi(s) = l$.

Whether and how this test can be conducted depends on the representation of π. Policy predicate abstraction is applicable in principle so long as any method for solving TSat$^{\pi}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ is available. Here, we focus on feed-forward NN with ReLU activation functions.

We encode these into SMT by extending the TSat$(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ encoding as follows. The inputs to the NN are the unprimed variables from TSat$(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$. We add an additional variable for every internal and output edge of the NN. Each neuron is a constraint relating its input and output edges, with the ReLU activation being encoded via an if-then-else construct. The NN output edges are constrained such that the maximal-valued edge is the one corresponding to the label l.

For illustration, say that in the example from Section 4 we have a single-layer NN with three neurons whose outputs are encoded by variables n_1, n_2, and n_3, of which n_2 corresponds to the label of the desired operator o. Then TSat$^{\pi}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ contains constraints relating x and y to each of n_1, n_2, n_3 according to the NN weights and ReLU cases, as well as the constraints $n_2 > n_1$ and $n_2 > n_3$ encoding that the correct label is chosen. A full specification of the SMT encoding is in the TR.

6 Enhancements through Relaxed Tests

An exact SMT solution of TSat$^{\pi}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ is computationally very expensive, due to the large number of disjunctions encoding every ReLU activation function – every neuron – in the NN policy representation. Indeed, as we shall see in our experiments, this computational expense makes policy predicate abstraction infeasible in practice.

To improve this, we next introduce a range of algorithmic enhancements, leveraging relaxed tests that over-approximate TSat$^{\pi}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$. If such a relaxed test is unsatisfiable, then TSat$^{\pi}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ is unsatisfiable and we don't need to call the exact SMT solver. We design such relaxed SMT tests in two ways, namely 1. through reduced conditions that are necessary for TSat$^{\pi}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ to be satisfiable, and 2. through continuous relaxation of the bounded-integer state variables. We now consider these two

[3]One could in principle build the entire graph $\Theta^{\pi}_{\mathcal{P}}$, not restricted to a start condition, based on which one could then answer arbitrary safety queries ρ. Yet this would forego the graph-size reduction resulting from the use of a fixed policy from a fixed start condition. As we will see, that reduction is crucial for practicability.

possibilities in turn. Then we introduce additional enhancements: 3. using results of the relaxed test as per 2. to simplify the exact SMT test; and 4. using branch-and-bound around relaxed test as per 2. to avoid the exact SMT test altogether.

6.1 Necessary Conditions for $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$

We devise four different conditions that are necessary for $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$ to be satisfiable. The conditions essentially check different parts of $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$ in isolation. The resulting SMT encodings are smaller, and hence cheaper to reason about.

- **Transition test of** $\mathcal{T}_\mathcal{P}$: $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$ can only be satisfied if $\text{TSat}(s_\mathcal{P}, o, s'_\mathcal{P})$ is. The SMT encoding of $\text{TSat}(s_\mathcal{P}, o, s'_\mathcal{P})$ does not involve the NN.
- **Selection test IsSelect**$^\pi(s_\mathcal{P}, l)$: $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$ can only be satisfied if there exists a concrete state $s \in [s_\mathcal{P}]$ such that $\pi(s) = l$, where l is the label of o. While still involving the NN, $\text{IsSelect}^\pi(s_\mathcal{P}, l)$ does not include the operator-related constraints. Moreover, once $\text{IsSelect}^\pi(s_\mathcal{P}, l)$ is violated for some l, one can skip the transition tests $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$ for all l-labeled operators o and predicate states $s'_\mathcal{P}$ altogether.
- **Applicability test isApp**$(s_\mathcal{P}, o)$: $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$ can only be satisfied if o is applicable in some concrete state $s \in [s_\mathcal{P}]$, i.e., $s \models o$. The SMT encoding of this test is a subset of that of $\text{TSat}(s_\mathcal{P}, o, s'_\mathcal{P})$. If $\text{isApp}(s_\mathcal{P}, o)$ is violated, one can directly skip $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$ for all predicate states $s'_\mathcal{P}$.
- **Policy-restricted applicability test isApp**$^\pi(s_\mathcal{P}, o)$: $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$ can only be satisfied if the policy actually selects the label l of o for some state for which o is applicable. The SMT encoding of this test is given by the combination of $\text{IsSelect}^\pi(s_\mathcal{P}, l)$ and $\text{isApp}(s_\mathcal{P}, o)$.

6.2 Continuous Relaxation

Each test involving the NN can be relaxed by interpreting the integer state variables at the NN input as continuous variables (with domain $\mathbb{R}$). We notate such continuously-relaxed tests by an $\mathbb{R}$ subscript, e.g., $\text{TSat}^\pi_\mathbb{R}(s_\mathcal{P}, o, s'_\mathcal{P})$.

The decisive advantage of this relaxation is the applicability of existing SMT solvers dedicated to NN analysis. Specifically, this allows us to leverage *Marabou* (Katz et al. 2019), an SMT solver tailored to satisfiability queries over neural networks. *Marabou* assumes a neural network with ReLU activation functions, and conjunctions of linear constraints over the NN inputs and outputs. It decides whether there exists an input/output pair of tuples over $\mathbb{R}$ satisfying these constraints. All our continuously-relaxed tests match this profile and can thus be tackled by *Marabou*.

6.3 Fixing Activation Cases

If the continuously relaxed test $\text{TSat}^\pi_\mathbb{R}(s_\mathcal{P}, o, s'_\mathcal{P})$ is satisfiable, then we still need to run the exact test $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$. One can, however, even in this case leverage $\text{TSat}^\pi_\mathbb{R}(s_\mathcal{P}, o, s'_\mathcal{P})$ to improve the performance of the $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$, namely by fixing some of the "activation cases" in the neural network. This idea has been deployed in other contexts before (e.g. (Mohammadi et al. 2020; Katz et al. 2019)), and here we adopt it in our setting.

The idea for ReLU works as follows: if the activation-function input x is known to be ≤ 0, then the SMT constraints can fix the output x' to $x' = 0$; if x is known to be ≥ 0, the output can be fixed to $x' = x$. The required knowledge here can be derived from reasoning about the relaxed encoding (e.g. $\text{TSat}^\pi_\mathbb{R}(s_\mathcal{P}, o, s'_\mathcal{P})$). Marabou does so by identifying bounds implied by individual constraints, as well as reasoning about network topology through symbolic interval propagation (Wang et al. 2018). We use these bounds to simplify the exact SMT encoding (e.g. $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$).

6.4 Branch & Bound around Relaxation

Observe that, in the case where a relaxed test (like $\text{TSat}^\pi_\mathbb{R}(s_\mathcal{P}, o, s'_\mathcal{P})$) is satisfiable, the corresponding exact test (like $\text{TSat}^\pi(s_\mathcal{P}, o, s'_\mathcal{P})$) is not needed if the solution to the relaxed test found by *Marabou* happens to be integer. While this will typically not be the case, one can iterate calls to *Marabou* in a search for such a solution, instead of calling the exact SMT test.

We realize this approach in terms of a **branch & bound (B&B)** search around *Marabou*. In each iteration, if there exists a state variable v assigned to a non-integer value α in the solution returned by *Marabou*, we pick one such v and create two search branches, adding $v \leq \lfloor \alpha \rfloor$ respectively $v \geq \lceil \alpha \rceil$ to the relaxed-test encoding. A branch is terminated when the encoding is proved to be unsatisfiable, or when an integer solution is found.

7 Computing the Abstract State Space

Putting the pieces together, we are now ready to explain how the abstract state space is computed. Specifically, given an NN policy π, a set of predicates $\mathcal{P}$, and a safety property $\rho = (\phi_0, \phi_U)$, we build the fragment of $\Theta^\pi_\mathcal{P}$ reachable from the abstract start states $S_0|_\mathcal{P} = \{s_\mathcal{P} \in \mathcal{S}_\mathcal{P} \mid s_\mathcal{P} \models \phi_0\}$. We do so using a forward search in abstract state space. The main challenge here is how to effectively implement abstract state expansion. Algorithm 1 shows pseudo-code.

The main loop of the procedure iteratively processes each action label. For the ones selected by the policy (lines 2 and 3), it proceeds to the corresponding operators. If an operator is applicable (lines 5 to 7), the `enumerate_states` procedure generates the successor predicate states $s'_\mathcal{P}$.

Observe that any predicate state may in principle qualify for $s'_\mathcal{P}$ – in contrast to explicit-state search, we do not have a declarative model from which we could read off directly which states $s'_\mathcal{P}$ may be reached in a single step from $s_\mathcal{P}$. Hence `enumerate_states` performs backtracking search in the space of possible $s'_\mathcal{P}$. Branches in that search are cut based on entailment information gleaned from simple (small) SMT tests. Namely, first, we check in line 8 for each predicate individually whether a truth value is entailed by $s_\mathcal{P}$ along with the operator o. We initialize $s'_\mathcal{P}$ accordingly. Second, truth value commitments for one predicate may entail truth values for other predicates. We pre-compute such relations for each predicate individually. During backtracking, we use this information to propagate truth values, akin

Algorithm 1: Abstract state expansion.

Input: $s_{\mathcal{P}} \in \mathcal{S}_{\mathcal{P}}$

1 **for** *each* $l \in \mathcal{L}$ **do**
  `// selection tests:`
2   **if** $\neg \mathit{IsSelect}^{\pi}_{\mathbb{R}}(s_{\mathcal{P}}, l)$ **then continue**
3   **if** $\neg \mathit{IsSelect}^{\pi}(s_{\mathcal{P}}, l)$ **then continue**
4   **for** *each* $o \in \mathcal{O}$ *with* $o = (g, l, u)$ **do**
    `// applicability tests:`
5   **if** $\neg \mathit{isApp}(s_{\mathcal{P}}, o)$ **then continue**
6   **if** $\neg \mathit{isApp}^{\pi}_{\mathbb{R}}(s_{\mathcal{P}}, o)$ **then continue**
7   **if** $\neg \mathit{isApp}^{\pi}(s_{\mathcal{P}}, o)$ **then continue**
8     $s'_{\mathcal{P}} \leftarrow$ `entailment_by` $(s_{\mathcal{P}}, o)$
9   `enumerate_states` $(s'_{\mathcal{P}})$

10 **Procedure** `enumerate_states` $(s'_{\mathcal{P}} : \mathcal{P} \rightarrow \{0, 1\})$ **:**
11   **if** $dom(s'_{\mathcal{P}}) = \mathcal{P}$ **then**
    `// transition tests:`
12   **if** $\neg \mathit{TSat}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ **then return**
13   **if** $\neg \mathit{TSat}^{\pi}_{\mathbb{R}}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ **then return**
14   **if** $\neg \mathit{TSat}^{\pi}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ **then return**
15   add $(s_{\mathcal{P}}, l, s'_{\mathcal{P}})$ to $T^{\pi}_{\mathcal{P}}$
16   **else**
17   pick some $p \in \mathcal{P} \setminus dom(s'_{\mathcal{P}})$
18   **let** $s'_{\mathcal{P}}(p) = 1$ **in**
19       $s'_{\mathcal{P}} \leftarrow s'_{\mathcal{P}}[$`entailment_by` $(p, 1)]$
20   `enumerate_states` $(s'_{\mathcal{P}})$
21   **let** $s'_{\mathcal{P}}(p) = 0$ **in**
22       $s'_{\mathcal{P}} \leftarrow s'_{\mathcal{P}}[$`entailment_by` $(p, 0)]$
23   `enumerate_states` $(s'_{\mathcal{P}})$

to unit propagation, prior to the recursion (lines 19 and 22). In the leaves of the search, we run satisfiability tests to check whether or not a transition is possible (lines 12 to 14).

Throughout Algorithm 1, we apply the various tests from Section 6 to reduce computational effort in SMT. All tests except $\mathrm{TSat}^{\pi}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$ are optional, yet may reduce work. The algorithm is modular with respect to how the tests are performed, e.g., whether an off-the-shelf SMT solver or our branch & bound method is used for $\mathrm{TSat}^{\pi}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$.

The set of abstract start states $S_0|_{\mathcal{P}} = \{s_{\mathcal{P}} \in \mathcal{S}_{\mathcal{P}} \mid s_{\mathcal{P}} \models \phi_0\}$ at the beginning of forward search is computed in a manner analogous to the `enumerate_states` procedure. We always build the entire $\Theta^{\pi}_{\mathcal{P}}$ reachable from those states, continuing even if we already reached an abstract unsafe state. This is because reaching an unsafe state just means that *at least one* start state $s_{\mathcal{P}} \in S_0|_{\mathcal{P}}$ is unsafe. We can still prove other start states safe by continuing the construction.

8 Experiments Design

The setup of our experiments is complex. Our algorithm has a large number of possible configurations; due to the recency of research into neural action policy verification, there is no established competition we can compare against; and there is no established set of benchmarks. We now address these points, before reporting our results in the next section.

Configuration	$\mathbb{R}$-test (Alg. 1 line 13)	Exact test (Alg. 1 line 14)
Base	×	Z3
Mar+Z3	✓	Z3
Mar+Z3(Mar)	✓	*Marabou* →Z3
BnB(Mar)	✓	B&B
Mar	✓	×

Table 1: Algorithm configurations evaluated. Base serves as a baseline, not using any algorithmic enhancements.

8.1 Algorithm Configurations

We evaluate five variants of our $\Theta^{\pi}_{\mathcal{P}}$ construction method, shown in Table 1. Base is a baseline version, constructing the reachable fragment of $\Theta^{\pi}_{\mathcal{P}}$ in the most straightforward fashion based on SMT tests. Mar+Z3 extends this by the observation that continuous relaxation with *Marabou* can be used to avoid costly SMT tests. Mar+Z3(Mar) in addition leverages the *Marabou* outcome to fix activation cases in the Z3 queries. BnB(Mar) instead modifies Mar+Z3 by using our branch-and-bound on top of Marabou. Finally, Mar just drops the exact tests altogether, relying completely on the continuous relaxation and thus computing an over-approximation of abstract reachability.

The predicate abstraction base tests (Algorithm 1 lines 5 and 12) are enabled throughout as they never hurt. The selection tests (lines 2 and 3) and applicability tests (lines 6 and 7) are disabled throughout. Our evaluation shows that they can improve performance, and can also deteriorate it when the benefit of the additional tests does not outweigh the gain. For space reasons, these results are not discussed in what follows. They are available in the TR.

8.2 Competing Approaches

To provide a comparison to alternative verification ideas, we implemented two competing methods:[4]

- *Explicit enumeration (EE)*. This constructs the concrete start states $s_0 \models \phi_0$ by querying $Z3$ in a binary search over the state variable domains (we experimented with several methods, and this one worked best). It then runs the policy from every s_0 in turn, enumerating non-deterministic transition outcomes. We employ duplicate checking across all these runs to avoid repeated work.
- *Bounded model checking (BMC)*. This encodes bounded-length unsafety into satisfiability queries, in a straightforward manner loosely inspired by Akintunde et al. (2018; 2019). We incrementally build SMT queries asking whether Θ^{π} contains a path of length L from ϕ_0 to ϕ_U. If the answer is negative, the SMT query is extended by unrolling the transition function one step further. This is repeated until either an unsafe path is found, or L exceeds a fixed upper bound L_{max}.

BMC can only prove safety up to length bound L_{max}, and our method is the only one parameterized by abstraction

[4]Other approaches, such as reachability analysis using star sets (Tran et al. 2019) or encoding abstract reachability into SMT (Cavada et al. 2014), would be interesting to try as well but are challenging to realize in our setting and thus beyond scope.

predicates, so the comparison across approaches needs to be handled with care. Nevertheless though, EE cannot handle the state explosion, and BMC cannot handle large L as the SMT query contains one copy of the NN for every step.

8.3 Benchmarks

The benchmarks required for policy verification include not only planning tasks, but also trained policies for those. We trained policies for a collection of domains from the literature, adapted to include unsafety conditions and non-deterministic actions. Details are available in the TR. In what follows, we give a short summary.

Planning domains. We experimented with variants of the Racetrack, Blocksworld, and SlidingTiles domains, as well as a simple transportation domain we will refer to as Transport. We encoded all these domains in the JANI format.

In Racetrack, we use the Barto-small map (Barto, Bradtke, and Singh 1995). Our safety property has 1000 randomly chosen start states, and a state is unsafe if the car has crashed into a wall. We use deterministic actions (no "slippery road") because otherwise an unsafe state is always reachable (namely when all actions fail).

In Blocksworld, actions moving a block b may non-deterministically fail, and when this happens the cost of moving b (represented by an additional state variable) is incremented. The start condition imposes a partial order on the blocks in the initial stacks. A state is unsafe if the number of blocks on the table exceeds a fixed limit. We consider instances with 6 and 8 blocks.

For SlidingTiles, we use an 8-puzzle instance. Like in Blocksworld, actions may fail, and if they do then the cost of moving the respective tile is incremented. The start condition imposes a partial order on the tile positions, and unsafe states are specified in terms of a set of unsafe tile positions.

In Transport, a truck must deliver packages on a straight-line road to the other side of a bridge, and an unsafe state occurs if the truck is too heavily loaded while crossing the bridge. The start condition restricts the truck and packages to be on the "non-goal" side of the bridge.

Policy training often had trouble dealing with large numbers of actions (inapplicable actions were often selected). Hence, in all domains, we leveraged the possibility (mentioned in Section 2 and available in JANI) to express state-dependent effects in terms of sets of operators sharing the same action label. For example, our actions in SlidingTiles are simply "left, right, up, down", avoiding the enumeration of tile/position combinations at the level of actions.

Trained policies. For every considered domain instance, we used deep Q-learning (Mnih et al. 2015) to train three feed-forward NN policies of different sizes. The number of hidden layers is fixed to 2 for each policy, the number of neurons per layer is 16, 32, and 64 respectively. The rewards for the training are positive on goal states and negative on unsafe ones. In some cases, we used mild reward shaping (giving positive rewards already for achieving individual goal facts) to achieve more effective training.

In Blocksworld and SlidingTiles, we distinguish policies that do vs. do not take move costs into account. While the former is more natural, it sometimes makes verification infeasible. Hence we show results for cost-aware policies where feasible, and for cost-ignoring policies elsewhere.

The policies are mostly safe (as our verification results show). The policies mostly select applicable actions, so that the number of reachable states under non-determinism is too large to enumerate (as our results for EE show).

Abstraction predicates used in our experiments. We do not automate the selection of abstraction predicates yet, instead providing these as input and scaling them as an important algorithm parameter in our experiments.

We consider predicates of the form $v \geq c$, comparing a state variable $v \in \mathcal{V}$ to a threshold value $c \in D_v$. We scale $\mathcal{P}$ by gradually adding predicates until, in the maximal predicate set, all variable values can be distinguished (and hence the abstraction $\Theta^{\pi}_{\mathcal{P}}$ equals the policy-restricted state space Θ^{π}). We mildly adapt this scheme to each domain. In Racetrack, we refine all state variables simultaneously, adding more predicates for every v in each step. In Blocksworld and SlidingTiles, we refine the move-cost variables last, which makes sense as these are least important to safety. In Transport, we first completely refine the truck location, then add predicates for the other variables (package locations, truck load) individually, as verification becomes very hard when adding the latter. For each v, the sequence of predicates follows a binary search pattern, iteratively cutting intervals between neighboring threshold values in half.

9 Experiments Results

We have implemented our approach on top of a C++ code base for automata networks modeled in JANI (Budde et al. 2017). We use *Marabou* to solve the continuously-relaxed NN-SAT tests and we use $Z3$ for all other SMT queries. All experiments were run on machines with Intel Xenon E5-2650 processors with a clock rate of 2.2 GHz, with time and memory limits of 12 h and 4 GB respectively.

Our evaluation in what follows addresses four questions:

1. What are the sources of complexity in policy predicate abstraction (PPA), compared to standard predicate abstraction (PA) ignoring the policy?
2. How do the PPA algorithm variants from Table 1 compare? In particular, to what extent do our enhancements improve performance?
3. Which safety properties does PPA manage to prove in our benchmark collection?
4. How does PPA fare compared to the competing policy verification approaches?

Figure 1 shows the data for all these discussions (data for competing approaches is given below in Section 9.4).

9.1 Sources of Complexity

In PA, the dominating source of complexity is the state-space explosion, which leads to **(1) exponential growth of the abstract state space as a function of** $|\mathcal{P}|$. When computing reachability from a start condition ϕ_0 as we do here (as opposed to building the entire abstract state space), this can be counter-balanced by **(2) the gain in precision as** $|\mathcal{P}|$

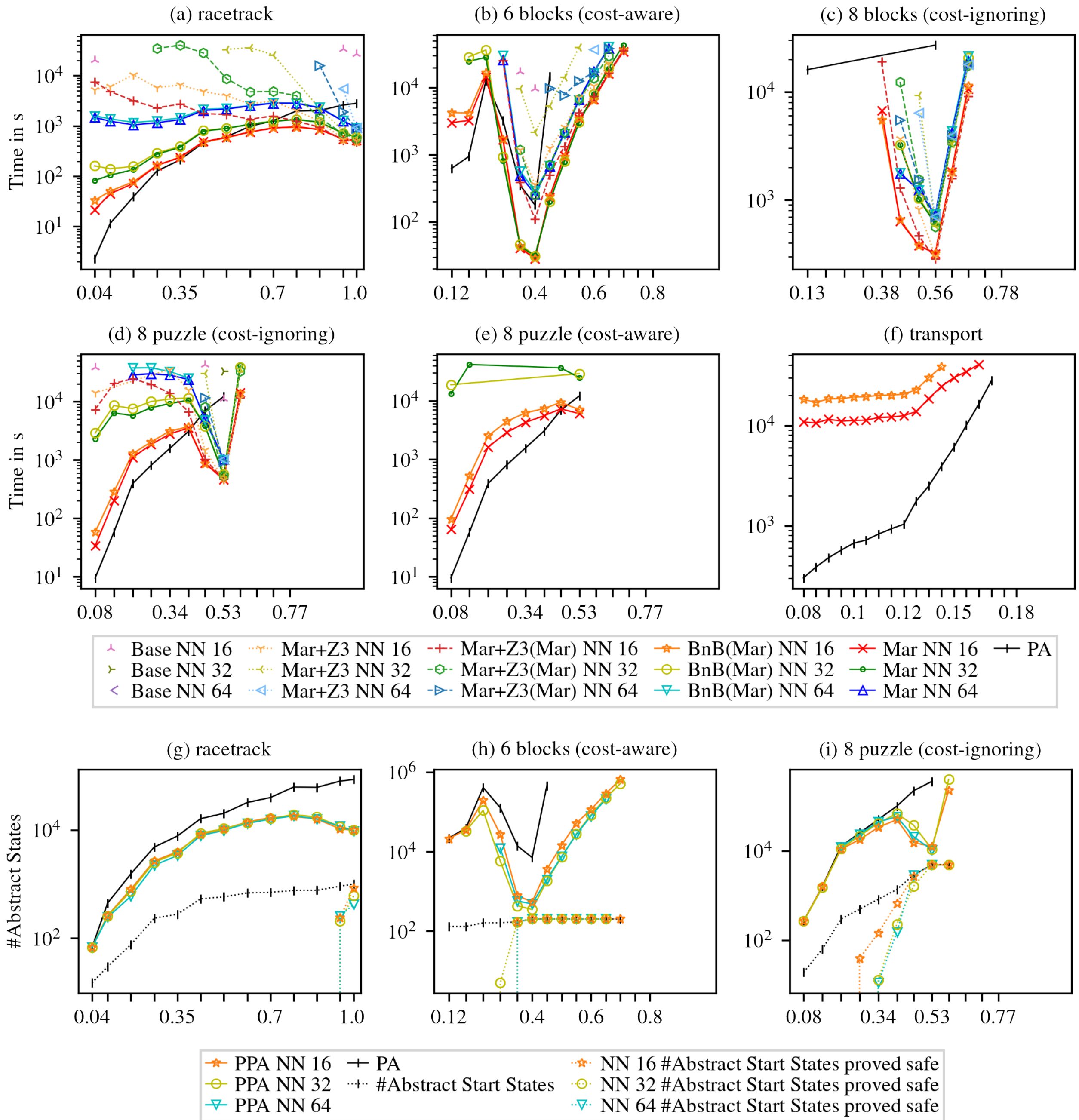

Figure 1: (a) – (f): runtime for policy predicate abstraction variants (cf. Table 1) and standard predicate abstraction (PA). (g) – (i): abstract state space size, number of abstract start states, and number of abstract start states proved safe (using BnB(Mar)). x-axes range over abstraction predicate sets $\mathcal{P}$ and show $\%$ of maximal $|\mathcal{P}|$. Timed-out runs are omitted from the plots.

grows, pruning spurious reachability. In PPA, we additionally have **(3) the new source of complexity in NN analysis**, i.e., complex SMT calls; and **(4) the new gain in reachability reduction from fixing the policy together with ϕ_0.**

Figure 1 nicely shows the interplay between these aspects. Consider first Racetrack, plots (a) and (g). In (g), we see the impact of (1) in the growth of the PA curve (note the logarithmic y-scale), and we see the impact of (4) in the reduction of abstract state space size for large predicate sets. In (a), focusing only on the most effective PPA variants Mar and BnB(Mar) for now, we additionally see the effect of (3), causing PPA to be more costly than PA up to mid-size predicate sets; and then the effect of (4), causing PPA to be *less* costly than PA for larger predicate sets. Note here that (4) can outweigh (3) – the verification of neural action policies can be more effective than classical verification!

Of course this observation is specific to our context, in particular the small size of the neural networks involved. But similar phenomena occur across our benchmarks. In Blocksworld, plots (b) (c) (h), the observations are exactly as above, except that now the reduction in abstract state space size happens near the middle of the predicate-set-size scale already; and that (2) kicks in for PA, leading to a temporary improvement in PA runtime. The sweetspot in abstraction complexity (0.4 in (b), 0.56 in (c)) is exactly that where all non-cost-predicates have been added, i.e., where costs are abstracted away but everything else is captured precisely. The observations in SlidingTiles, plots (d) (e) (i), are identical except that there is no benefit of kind (2) for PA. In Transport (f), PPA is exceedingly costly on small predicate sets due to (3). Indeed, we found (3) to typically be more problematic for small $\mathcal{P}$, as the SMT queries are then done for larger NN input regions. For larger $\mathcal{P}$, this effect gradually diminishes, and the gap between PA and PPA closes as, thanks to (4), PPA suffers much less from (1) than PA does.

9.2 PPA Algorithm Enhancements

Consider now the comparison across PPA variants as per Table 1. The baseline Base clearly is hopeless. There are only a few points in our benchmark space where it manages to construct the abstract state space. Adding continuous relaxation and *Marabou* in Mar+Z3 much improves this, but still is quite ineffective. The activation-case fixing in Mar+Z3(Mar) can yield substantial improvements (see e.g. Racetrack in Figure 1 (a)). But the key to scalability is to get rid of the generic SMT solver $Z3$ for queries involving the NN, and instead rely on *Marabou* completely, which still allows to compute the policy predicate abstraction exactly thanks to our branch-and-bound approach in BnB(Mar). The latter is only mildly less effective than the over-approximating variant Mar which uses continuous relaxation without branch-and-bound.

9.3 Safety Proved

Figure 1 (g) – (i) shows data on the number of abstract start states proved safe. In Racetrack, nothing is proved safe until all predicates are added and hence the abstraction is not abstract anymore. In Blocksworld however, all abstract start states – and hence the overall policy behavior – are proved safe once all non-cost-predicates are in. In SlidingTiles, this is the case for 4839 of 4900 abstract start states, many of which are proved safe already with smaller predicate sets.

Benchmark \ NN	$L_U^{\min}$ 16|32|64	$t_U^{\min}$ 16|32|64	$L_{checked}^{\max}$ 16|32|64
Racetrack	3|3|3	36.9|40.1|316.5	12|11|7
6 Blocks (cost-awa)	-	-	6|5|4
8 Blocks (cost-ign)	-	-	5|5|4
8-puzzle (cost-ign)	2| - | -	72.8| - | -	7|3|0
8-puzzle (cost-awa)	-	-	3|3|0
Transport	1|1| -	57.0|20548.0| -	2|1|0

Table 2: Results for BMC: length $L_U^{\min}$ of shortest unsafe path is one is found; runtime $t_U^{\min}$ to find that path in seconds; maximal path length $L_{checked}^{\max}$ checked at timeout; distinguishing cost-aware policies (cost-awa) and cost-ignoring policies (cost-ign) where applicable.

9.4 Competing Approaches

Let us finally discuss the competing approaches, explicit enumeration (EE) and bounded model-checking (BMC). Data for these is not included in Figure 1 as they achieve very little on our benchmarks.

EE easily verifies the policies in the Racetrack task (less than a second for each policy), as the state space there is small. However, EE exhausts our 4GB memory limit on all other problem instances.

Table 2 shows the data for BMC. This approach is effective in finding short unsafe paths if these exist. Yet it is useless otherwise. As the $L_{checked}^{\max}$ data shows, except in Racetrack where the state space is small, BMC is unable to reach substantial path lengths.[5]

10 Conclusion

The verification of neural network behavior becomes more and more important. We have introduced policy predicate abstraction as a new method in the so-far scant arsenal to address such verification, and we have shown that it can be feasible and can outperform other methods. Interestingly, thanks to the reduced reachability when fixing both a start condition and a policy, it can even be more effective than standard predicate abstraction ignoring the policy.

A next step has to be the automatic derivation of abstraction predicates, canonically via counter example guided abstraction refinement (e.g. (Clarke et al. 2000)), which will need to be extended to distinguish states based on NN behavior. There are many opportunities to speed up our approach: the use of other NN analysis approaches; adversarial attacks to prove satisfiability of $\mathrm{TSat}^{\pi}(s_{\mathcal{P}}, o, s'_{\mathcal{P}})$; lazy abstraction refining the predicate set locally; and parallelization of SMT test variants and the entire abstract state-space construction. It may be interesting to look at possible connections to XAIP (see (Chakraborti et al. 2019) for an overview).

[5]To assess this in the cases where a short unsafe path exists, we keep running BMC after that happens.

Acknowledgments

This work was funded by DFG Grant 389792660 as part of TRR 248 – CPEC (https://perspicuous-computing.science).

References

Akintunde, M.; Lomuscio, A.; Maganti, L.; and Pirovano, E. 2018. Reachability Analysis for Neural Agent-Environment Systems. In *KR*.

Akintunde, M. E.; Kevorchian, A.; Lomuscio, A.; and Pirovano, E. 2019. Verification of RNN-Based Neural Agent-Environment Systems. In *AAAI*.

Ball, T.; Majumdar, R.; Millstein, T. D.; and Rajamani, S. K. 2001. Automatic Predicate Abstraction of C Programs. In *Prog. Lang. Design and Implementation (PLDI)*.

Barrett, C. W.; Sebastiani, R.; Seshia, S. A.; and Tinelli, C. 1994. Satisfiability modulo theories. *In Handbook of Satisfiability*, 825–885.

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to Act Using Real-Time Dynamic Programming. *AI*, 72(1-2): 81–138.

Budde, C. E.; Dehnert, C.; Hahn, E. M.; Hartmanns, A.; Junges, S.; and Turrini, A. 2017. JANI: Quantitative Model and Tool Interaction. In *TACAS*.

Cavada, R.; Cimatti, A.; Dorigatti, M.; Griggio, A.; Mariotti, A.; Micheli, A.; Mover, S.; Roveri, M.; and Tonetta, S. 2014. The nuXmv Symbolic Model Checker. In *CAV*.

Chakraborti, T.; Kulkarni, A.; Sreedharan, S.; Smith, D. E.; and Kambhampati, S. 2019. Explicability? Legibility? Predictability? Transparency? Privacy? Security? The Emerging Landscape of Interpretable Agent Behavior. In *ICAPS*.

Cimatti, A.; Dubrovin, J.; Junttila, T. A.; and Roveri, M. 2009. Structure-aware computation of predicate abstraction. In *Formal Methods in Computer-Aided Design (FMCAD)*.

Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2000. Counterexample-Guided Abstraction Refinement. In *CAV*.

de Moura, L.; and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

Dutta, S.; Chen, X.; and Sankaranarayanan, S. 2019. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *Intl Conf Hybrid Systems: Computation and Control (HSCC)*.

Edelkamp, S. 2001. Planning with Pattern Databases. In *ECP*.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *ICAPS*.

Gehr, T.; Mirman, M.; Drachsler-Cohen, D.; Tsankov, P.; Chaudhuri, S.; and Vechev, M. T. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Security and Privacy (SP)*.

Graf, S.; and Saïdi, H. 1997. Construction of Abstract State Graphs with PVS. In *CAV*.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *ICAPS*.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge & Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *JACM*, 61(3): 16:1–16:63.

Henzinger, T. A.; Jhala, R.; Majumdar, R.; and McMillan, K. L. 2004. Abstractions from proofs. In *Principles of Programming Languages (POPL)*.

Huang, S.; Fan, J.; Li, W.; Chen, X.; and Zhu, Q. 2019. ReachNN: Reachability analysis of neural-network controlled systems. *ACM Trans. Emb. Comp. Sys.*, 18: 1–22.

Huang, X.; Kwiatkowska, M.; Wang, S.; and Wu, M. 2017. Safety Verification of Deep Neural Networks. In *CAV*.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In *ICAPS*.

Ivanov, R.; Carpenter, T. J.; Weimer, J.; Alur, R.; Pappas, G. J.; and Lee, I. 2021. Verifying the Safety of Autonomous Systems with Neural Network Controllers. *ACM Trans. Emb. Comp. Sys.*, 20: 7:1–7:26.

Katz, G.; Barrett, C. W.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *CAV*.

Katz, G.; Huang, D. A.; Ibeling, D.; Julian, K.; Lazarus, C.; Lim, R.; Shah, P.; Thakoor, S.; Wu, H.; Zeljic, A.; Dill, D. L.; Kochenderfer, M.; and Barrett, C. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *CAV*.

Li, J.; Liu, J.; Yang, P.; Chen, L.; Huang, X.; and Zhang, L. 2019. Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In *Static Analysis (SAS)*.

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.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature*, 518: 529–533.

Mohammadi, K.; Karimi, A.; Barthe, G.; and Valera, I. 2020. Scaling Guarantees for Nearest Counterfactual Explanations. *CoRR*, abs/2010.04965.

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *JAIR*, 62: 535–577.

Sun, X.; Khedr, H.; and Shoukry, Y. 2019. Formal Verification of Neural Network Controlled Autonomous Systems. In *Intl Conf Hybrid Systems: Computation and Control (HSCC)*.

Toyer, S.; Thiébaux, S.; Trevizan, F. W.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *JAIR*, 68: 1–68.

Tran, H.; Cai, F.; Lopez, D. M.; Musau, P.; Johnson, T. T.; and Koutsoukos, X. D. 2019. Safety Verification of Cyber-Physical Systems with Reinforcement Learning Control. *ACM Trans. Embed. Comput. Syst.*, 18(5s): 105:1–105:22.

Wang, S.; Pei, K.; Whitehouse, J.; Yang, J.; and Jana, S. 2018. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *USENIX Security Symposium*.