Online Learning of Numeric Action Models for Planning

Argaman Mordoch
Ben Gurion University
Beer Sheva, Israel
mordocha@post.bgu.ac.il

Yarin Benyamin
Ben Gurion University
Beer Sheva, Israel
bnyamin@post.bgu.ac.il

Shahaf S. Shperberg
Ben Gurion University
Beer Sheva, Israel
shperbsh@post.bgu.ac.il

Brendan Juba
Washington University in St. Louis
St. Louis, Missouri
bjuba@wustl.edu

Roni Stern
Ben Gurion University
Beer Sheva, Israel
roni.stern@gmail.com

ABSTRACT

Numeric planning addresses sequential decision-making in domains involving both discrete and continuous state variables. While effective, it requires a model of actions' preconditions and effects, which might be unavailable or hard to model manually. Therefore, prior work developed algorithms that automatically learn numeric action models from execution traces. However, these approaches assumed that such traces are available, which may not hold in realistic settings where past interactions are unavailable. In this work, we introduce NOAM, the first online action model learning algorithm for numeric planning. NOAM can learn from both successful and failed observations. It iteratively refines two types of action models: a safe, risk-averse model and a more optimistic, exploratory action model. It performs goal-oriented exploration and prioritizes actions that are expected to be informative. We evaluated NOAM on several classical numeric benchmark domains and found that the models it learns enable solving most problems within those domains.

KEYWORDS

Planning; Online Learning; Action Model Learning

ACM Reference Format:
Argaman Mordoch, Yarin Benyamin, Shahaf S. Shperberg, Brendan Juba, and Roni Stern. 2026. Online Learning of Numeric Action Models for Planning. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 9 pages. https://doi.org/10.65109/DJKY6536

1 INTRODUCTION

Domain-independent planning is a long-standing goal in Artificial Intelligence (AI). Most planning algorithms rely on a predefined *domain model*, typically expressed in a language such as the Planning Domain Definition Language (PDDL) [13, 14]. A domain model includes an *action model* specifying the available actions and their corresponding preconditions and effects. Manually crafting action models for real-world domains is challenging and error-prone, motivating research on automated action model learning.


This work is licensed under a Creative Commons Attribution International 4.0 License.

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus. © 2026 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). https://doi.org/10.65109/DJKY6536

Action model learning approaches are broadly categorized as *offline* or *online*. Offline methods assume that execution traces are available prior to learning and have been extensively studied [2, 5, 10, 34, 42]. In contrast, online methods do not assume access to prior traces; instead, the agent interacts with the environment to gather trace data while learning the model [8, 22, 30, 35]. Existing online action model learning approaches are limited to domains without numeric components, which severely restricts their applicability, as many real-world problems include numeric aspects.

To close this gap, we present Numeric Online Action Model Learner (NOAM), the first online action model learning algorithm that returns a numeric action model. NOAM maintains two action models: a *safe* action model, representing knowledge about the agent's actions that is certain, and an *optimistic* action model, which is riskier but facilitates exploration and enables solving more problems. In each episode, NOAM attempts to find and execute a plan using these models. If it fails, NOAM invokes a novel information-gathering heuristic that guides exploration toward performing actions that refine the safe and optimistic models. For the safe action model, NOAM employs Numeric Safe Action Models Learning (N-SAM) [27], a previously proposed method that learns safe numeric action models from successful traces. For the optimistic model, we propose Support Vector and Regression Action Model learner (SVRAM), a novel action model learning algorithm capable of learning from both positive (successful) and negative (unsuccessful) transitions. To utilize the negative transitions, SVRAM relies on the safe action model learned by N-SAM to isolate the cause of failed transitions. It learns Boolean preconditions and effects by starting with an overly permissive model and iteratively refining it, similar to [3, 35]. To learn numeric preconditions, SVRAM iteratively uses Support Vector Machines (SVMs) [15] to identify hyperplanes that separate positive and negative examples, resulting in a polytope representation of the preconditions.

We empirically evaluated NOAM on nine standard numeric planning domains, measuring both the correctness of the learned models and the agent's ability to solve problems using them. The results show that after less than 100 episodes, NOAM is able to learn highly accurate models that enable the agent to solve most test problems in all the evaluated domains.

2 PRELIMINARIES

We focus on planning problems in domains with deterministic action outcomes and fully observable states, represented using a mix of Boolean and continuous state variables. Such problems can be

modeled using the common fragment of PDDL2.1 [13]. A *domain* is a tuple $D = \langle F, X, A, M \rangle$, where F is a finite set of Boolean variables (fluents); X is a set of numeric variables (functions); A is a set of actions; and M is an *action model* for these actions. A state s is an assignment of values to all variables in $F \cup X$. $s|_F$ and $s|_X$ denote the projection of s onto its Boolean and numeric variables, respectively. For a state variable $v \in F \cup X$, we denote by $s(v)$ the value assigned to v in state s. Every action $a \in A$ is defined by a tuple $\langle name(a), params(a) \rangle$, representing its name and parameters. An action model M is a pair of functions, pre_M and eff_M, that map actions in A to their preconditions and effects, respectively. The preconditions of action a, i.e., $pre_M(a)$, consist of assignments over the Boolean fluents and a set of conditions over the functions, specifying the states in which a can be applied. These conditions are of the form (ξ, Rel, k), where ξ is an arithmetic expression over X, $Rel \in \leq, <, =, >, \geq$, and k is a number. An action a is applicable in a state s under action model M, denoted $app_M(a, s)$, if s satisfies $pre_M(a)$. The effects of action a, i.e., $eff_M(a)$, are assignments over F and X representing how the state changes after applying a. An assignment over a function $x \in X$ is a tuple of the form $\langle x, op, \xi \rangle$, where ξ is a numeric expression over X, and op one of the following operations: increase ("+="), decrease ("-="), or assign (":="). Applying a in s according to the action model M results in a state, denoted by $a_M(s)$, that differs from s only according to the assignments in $eff_M(a)$. The subscript M is omitted when it is clear from the context. A planning *problem* is a tuple $\langle D, s_0, G \rangle$, where D is the domain, s_0 the initial state, and G the goal—an assignment over a subset of the Boolean fluents and a set of conditions over the numeric functions. A solution to a planning problem is a *plan*, i.e., a sequence of actions $a_0, a_1, \ldots, a_n$ such that a_0 is applicable in s_0 and $a_n(a_{n-1}(\ldots a_0(s_0)\ldots))$ yields a state satisfying G.

Planning domains and problems are often defined in a *lifted* manner, where actions, fluents, and functions are parameterized and may have *types*. Grounded actions, fluents, and functions are pairs of the form $\langle v, b_v \rangle$, where v is an action, fluent, or function, and b_v maps the parameters of v to concrete objects of compatible types. A *literal* is a fluent or its negation. The notions of binding, lifting, and grounding for fluents extend naturally to literals. A state is an assignment of values to all grounded literals and functions. A plan is a sequence of *grounded actions*. The preconditions and effects of an action in a lifted domain contain *parameter-bound* literals (pb-literals) and functions (pb-functions). A pb-literal for a lifted action α is a pair $\langle \ell, b_{\ell,\alpha} \rangle$, where ℓ is a lifted literal and $b_{\ell,\alpha}$ maps each parameter of ℓ to a parameter of α. pb-functions are similarly defined. For a pb-literal $pb = \langle \ell, b_{\ell,\alpha} \rangle$ and an action $a = \langle \alpha, b_\alpha \rangle$, we denote by $pb(a)$ the grounded literal obtained by matching the parameters of pb and a, i.e., $pb(a) = \langle \ell, b_{\ell,\alpha} \circ b_\alpha \rangle$. We similarly denote $pf(a)$ for numeric functions. We denote by $L(\alpha)$ and $\Sigma(\alpha)$ the sets of pb-literals and pb-functions, respectively, that can be bound to α, i.e., all pb-literals and pb-functions whose parameters match those of α.

2.1 Action Model Learning Algorithms

A *labeled state transition* is a tuple $\langle \langle s, a, s' \rangle, label \rangle$, where s is the state before executing a, and s' is the result of applying a in s, i.e., $s' = a(s)$. The states s and s' are referred to as the *pre-state* and *post-state*, respectively. The *label* indicates whether the transition was **successful**, i.e., whether a is applicable in s. A *trajectory* is a sequence of labeled state transitions. *Offline action-model learning* algorithms receive a set of trajectories and output an action model.

Some offline action-model learning algorithms, such as LOCM [10] and LOCM2 [11], analyze observed plan sequences rather than state transitions. SLAF [5] learns action models from partially observable state transitions. FAMA [2] frames action-model learning as a planning problem, ensuring consistency with the provided observations. NOLAM [23] learns action models from noisy trajectories. LatPlan [6] and ROSAME-I [41] learn propositional action models from visual inputs. None of these algorithms guarantees execution soundness—that plans based on the learned model are applicable in the real environment. The Safe Action Model Learning (SAM) framework [19, 20, 24, 28, 36] addresses this by ensuring *safety*: plans generated with the learned model are guaranteed to execute successfully and produce the predicted states. The offline action-model learning algorithms discussed above assume the state variables are only Boolean fluents. To the best of our knowledge, PlanMiner [34] and N-SAM [27] are the only approaches that support learning numeric action models. PlanMiner uses symbolic regression and classification methods to learn preconditions and effects from partially known and noisy plan traces. N-SAM extends the SAM framework to learning numeric preconditions and effects while maintaining the aforementioned safety guarantee.

Several algorithms have been proposed for learning planning action models *online*—that is, without access to pre-collected trajectories. An online action-model learning algorithm must interact with the environment to gather transitions while simultaneously learning the action model [8, 9, 18, 21, 22, 30, 35, 40]. However, to the best of our knowledge, no existing online action-model learning algorithm supports numeric planning.

2.2 Problem Setting

We consider a standard online action-model learning setting. An agent operates in a planning domain; it knows the state representation and actions but not the domain's action model. The agent receives and handles a sequence of planning problems, possibly starting in different states, one at a time. For each problem, it starts in the initial state and performs actions until either a predefined action limit is reached or the problem is solved. The agent may attempt an inapplicable action, resulting in an *action failure*. For simplicity, we assume that if a is inapplicable in s, then $a(s) = s$. The agent has full observability of the state before and after each action. Also, it detects action failures—if action a in state s failed, then the agent observes the labeled transition $\langle \langle s, a, s \rangle, failure \rangle$.

In this work, the agent operates in a numeric planning domain. Action preconditions are conjunctions of Boolean literals and linear inequalities, and effects are conjunctions of Boolean literals and linear equations. The agent's objective is to learn an action model that enables it to solve problems in the domain, as in [19, 27, 35].

3 THE NOAM ALGORITHM

In this section, we present the Numeric Online Action Model (NOAM) learning algorithm. For each lifted action α, NOAM maintains two sets, $DB_+(\alpha)$ and $DB_-(\alpha)$, initially empty and populated over time

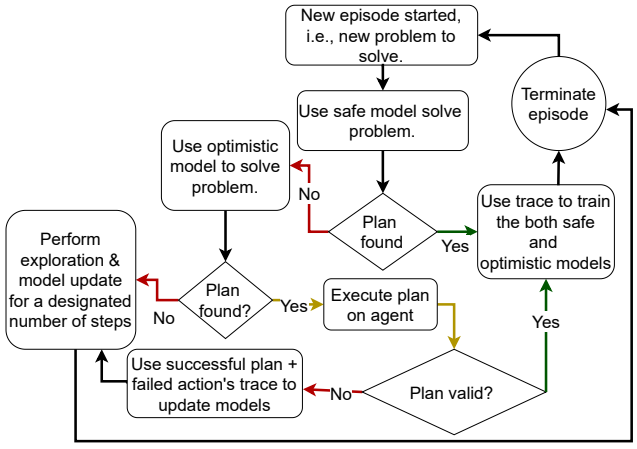

Figure 1: The pipeline of NOAM.

with successful and failed transitions, respectively. Based on these sets, it maintains two action models: M_{safe} and M_{op}. M_{safe} is a safe action model that allows the agent to execute only actions guaranteed to succeed. M_{op}, the *optimistic* action model, is less conservative, allowing actions that may fail, thereby encouraging exploration and enabling the solution of more problems.

Figure 1 provides a high-level view of NOAM. Each iteration of NOAM, referred to as an *episode*, begins with a new input problem π. NOAM first attempts to solve π using M_{safe}. If successful, the resulting plan is executed and the episode terminates. Otherwise, NOAM attempts to solve π using M_{op}. If a plan is found, the agent attempts to execute it. Since M_{op} is not safe, some planned actions may be inapplicable, and the plan may fail to reach the goal. If the agent reaches the goal, the episode ends. Otherwise, or if no plan is found, NOAM performs a fixed number of exploration steps, selecting actions expected to provide new information about the action model and updating the learned models accordingly. The trajectory executed in each episode is stored in $DB_+(\alpha)$ and $DB_-(\alpha)$ and used to update M_{safe} and M_{op}. Next, we describe the components of NOAM.

3.1 Learning the Safe Action Model

NOAM updates M_{safe} by running N-SAM [27], the only algorithm to date that returns a safe action model for numeric planning.[1] N-SAM learns Boolean preconditions and effects using SAM [19]. To learn numeric preconditions, N-SAM computes convex hulls over observed numeric variable values; for numeric effects, it performs linear regression. For details, see [27].

Note that N-SAM takes as input the successful transitions (DB_+) and ignores the failed transitions (DB_-), as failed transitions do not provide useful information for constructing a safe action model. Mordoch et al. [27] showed that, for action models with halfspace preconditions, the optimal safe preconditions are the convex hull of the pre-states, which correspond to positive examples in our setting. Theorem 1 extends this result and shows that, for conjunctions of linear inequalities, the optimal preconditions remain the convex hull of the positive examples, even when negative examples are available. Thus, negative examples do not contribute to learning safe numeric preconditions in this case.

[1]Technically, we use N-SAM *, a more advanced version of N-SAM [29].

Theorem 1. *For the class of action models with numeric preconditions defined by conjunctions of linear inequalities, the indicator for the convex hull of the set of positive examples $DB_+(\alpha)$ is a safe precondition for action α, and moreover, if any safe precondition for α is satisfied on a state s, the convex hull of $DB_+(\alpha)$ contains s.*

Proof. We first observe that the optimal safe precondition for action α is given by $\bigcap_{h\in\mathcal{H}^*} h$ for the set $\mathcal{H}^* = \bigcup_{\mathcal{H}'\in\mathbb{H}} \bigcup_{h\in\mathcal{H}'} h$ where

$$\mathbb{H} = \left\{ \text{sets of halfspaces } \mathcal{H}' : \begin{array}{c} \forall h \in \mathcal{H}' DB_+(\alpha) \subseteq h \text{ and} \\ \forall \tilde{s} \in DB^N_-(\alpha) \exists h \in \mathcal{H}' \tilde{s} \notin h \end{array} \right\}$$

i.e., the intersection of all halfspaces contained in any conjunction of halfspaces that are consistent with the observations. First, observe that for any $\tilde{s} \notin \bigcap_{h\in\mathcal{H}^*} h$, there must exist some $\mathcal{H}'$ such that $\tilde{s} \notin \bigcap_{h\in\mathcal{H}'} h$. Then since $\bigcap_{h\in\mathcal{H}'} h$ is consistent with the observations, these observations could have been generated by an action model where $\bigcap_{h\in\mathcal{H}'} h$ is the precondition of α, and thus any action model that allows α in state $\tilde{s}$ is not safe. Hence, any safe action model must have a precondition that is a subset of $\bigcap_{h\in\mathcal{H}^*} h$. At the same time, $\bigcap_{h\in\mathcal{H}^*} h$ is safe since whatever the true set of inequalities $\mathcal{H}'$ is, $\mathcal{H}' \in \mathbb{H}$ and hence $\mathcal{H}' \subseteq \mathcal{H}^*$. So if $s \in \bigcap_{h\in\mathcal{H}^*} h$, $s \in \bigcap_{h\in\mathcal{H}'} h$ as well, i.e., α is allowed in s in the real environment.

We next observe that $\bigcap_{h\in\mathcal{H}^*} h$ is precisely the convex hull of $DB_+(\alpha)$. For any set P defined by a conjunction of linear inequalities, P is convex. For any convex P, if $DB_+(\alpha) \subseteq P$, then the convex hull of $DB_+(\alpha)$ is also contained in P. Therefore, $\bigcap_{h\in\mathcal{H}^*} h$ contains the convex hull of $DB_+(\alpha)$. In particular, also note that since we are given that the real environment has preconditions defined by a conjunction of linear inequalities, the convex hull of $DB_+(\alpha)$ cannot contain any $\tilde{s} \in DB^N_-(\alpha)$. The convex hull of $DB_+(\alpha)$ is itself defined by a set of halfspaces $\mathcal{H}'$, so $\bigcap_{h\in\mathcal{H}^*} h$ is also contained in the convex hull of $DB_+(\alpha)$. Hence, the sets are equal. □

3.2 Learning the Optimistic Action Model

To learn the optimistic action model (M_{op}), we introduce a novel algorithm, the Support Vector and Regression Action Model learner (SVRAM). SVRAM learns a numeric action model from both successful and failed transitions. The optimistic model it produces may include actions that are inapplicable in the environment. However, since exploration is essential for learning the action model, executing potentially unsafe actions is unavoidable.

As a prerequisite, SVRAM uses the safe model M_{safe} to classify failed transitions according to the root cause of failure.

Classifying failed transitions. A transition may fail due to a violated (1) Boolean precondition, (2) numeric precondition, or (3) both. To capture these cases, SVRAM maintains two sets of failed transitions for each lifted action α: $DB^B_-(\alpha)$, for failures attributed to Boolean preconditions, and $DB^N_-(\alpha)$, for failures attributed to numeric preconditions. SVRAM uses M_{safe} to classify each failed transition $\langle s, a, s' \rangle$ into one or both sets. If a transition fails due to both types of preconditions, it appears in both sets. The classification is performed as follows. If s does not violate any Boolean

Algorithm 1 Boolean Model Refinement Algorithm

1: **Input**: $D = \langle F, A \rangle, DB_+, DB^B_-, L$
2: **Output**: refined Boolean model for every action in A
3: **for** $\alpha \in A$ **do**
4: $pre_\perp(\alpha),\ pre_\top(\alpha),\ eff_\perp(\alpha) \leftarrow \emptyset$
5: **for** $\langle s, a, s' \rangle \in DB_+$ **do**
6: **for** $pb \in L(\alpha)$ **do**
7: Add pb to $pre_\perp(\alpha)$ if $pb(a) \notin s|_F$
8: Add pb to $eff_\perp(\alpha)$ if $pb(a) \notin s'|_F$
9: **for** $C \in pre_\top(\alpha)$ **do**
10: $C \leftarrow C \setminus pre_\perp(\alpha)$
11: **for** $\langle s, a, s' \rangle \in DB^B_-$ **do**
12: $C \leftarrow \{pb \mid pb \in L(\alpha), pb(a) \notin s|_F\}$
13: Add $\{C \setminus pre_\perp(\alpha)\}$ to $pre_\top(\alpha)$
14: ApplyUnitPropagation($pre_\top(\alpha)$)
15: **for** $\alpha \in A$ **do**
16: $pre_{bool}(\alpha) \leftarrow \bigwedge\{\bigvee\{pre_\top(\alpha)\}\}$
17: $eff_{bool}(\alpha) \leftarrow \bigwedge\{L(\alpha) \setminus eff_\perp(\alpha)\}$
18: **return:** pre_{bool}, eff_{bool}

precondition of a according to M_{safe}, we add $\langle s, a, s' \rangle$ to $DB^N_-(\alpha)$. If s does not violate any numeric precondition of a according to M_{safe}, we add $\langle s, a, s' \rangle$ to $DB^B_-(\alpha)$. Transitions that violate both Boolean and numeric preconditions of a according to M_{safe} are labeled *temporarily unknown* and excluded from learning until additional data allow reclassification into either DB^B_- or DB^N_-. Miss-classifying such transitions may compromise N-SAM's safety guarantees; therefore, SVRAM attempts to reclassify them at the end of each episode.

Next, we describe how SVRAM uses DB_+, DB^B_-, and DB^N_- to learn an action model. Similar to N-SAM, it learns the Boolean and numeric parts of the action model separately, as follows.

Learning the Boolean action model. To learn the Boolean action model, SVRAM follows an approach similar to [3, 35]. These methods maintain a set of candidate action models and iteratively prune those inconsistent with the observed trajectories. In contrast, SVRAM maintains a single action model that is iteratively refined using both successful and failed transitions. Specifically, it constructs a conservative approximation of what *cannot* be a precondition or an effect. Based on this approximation, preconditions are represented as disjunctions of literals that may be necessary for applicability, and effects as a conjunction of all literals that may result from applying the action. Algorithm 1 describes this process in detail.

SVRAM receives as input the dataset of successful state transitions DB_+, the dataset of transitions that failed due to Boolean preconditions DB^B_-, and background knowledge about the domain—the set of pb-literals relevant to the actions (L). For each $\alpha \in A$, SVRAM initializes three sets: $pre_\perp(\alpha)$: the set of pb-literals that cannot be preconditions of α, $eff_\perp(\alpha)$: the set of pb-literals that cannot be effects of α, $pre_\top(\alpha)$: a set of CNF clauses of pb-literals, where each clause includes at least one pb-literal that is a precondition of α (line 4).

The algorithm then iterates over the stored successful state transitions (line 5) and, for every ground action $a = \langle \alpha, b_\alpha \rangle$ it updates: (1) $pre_\perp(\alpha)$ to include all pb-literals whose grounding is *not present* in $s|_F$ (line 7), and (2) $eff_\perp(\alpha)$ to include all pb-literals whose

Algorithm 2 Numeric Preconditions Refinement Algorithm

1: **Input**: $D = \langle X, A \rangle, DB_+, DB^N_-, \Sigma$
2: **Output**: numeric preconditions for every action in A
3: **for** $\alpha \in A$ **do**
4: $\mathcal{H} \leftarrow \emptyset$
5: $Pos(\alpha) \leftarrow \{s|_X(pf(x)) \mid x \in \Sigma(\alpha), \langle s, a, s' \rangle \in DB_+(\alpha)\}$
6: $Neg(\alpha) \leftarrow \{s|_X(pf(x)) \mid x \in \Sigma(\alpha), \langle s, a, s' \rangle \in DB^N_-(\alpha)\}$
7: **while** $Neg(\alpha) \neq \emptyset$ **do**
8: $v^* \leftarrow \arg\min_{v \in Neg(\alpha)} \text{dist}(v, Pos(\alpha))$
9: $h \leftarrow SVM(Pos(\alpha), \{v^*\})$
10: $Valid_{neg}(\alpha) \leftarrow \{v \in Neg(\alpha) \mid h(v) < 0\}$
11: $Neg(\alpha) \leftarrow Neg(\alpha) \setminus Valid_{neg}(\alpha)$
12: $\mathcal{H} \leftarrow \mathcal{H} \cup \{h\}$
13: **for all** $h_i, h_j \in \mathcal{H}, i \neq j$ **do**
14: **if** h_j dominates h_i **then**
15: $\mathcal{H} \leftarrow \mathcal{H} \setminus \{h_i\}$
16: $pre_{numeric}(\alpha) \leftarrow \mathcal{H}$
17: **return:** $pre_{numeric}$

grounding is *not present* in $s'|_F$ (line 8). It then prunes each clause $C \in pre_\top(\alpha)$ by removing pb-literals that appear in $pre_\perp(\alpha)$, since they are known not to be preconditions (line 10). Next, SVRAM iterates over the failed state transitions in DB^B_- and creates a clause C containing all pb-literals whose grounding is not part of $s|_F$. The algorithm removes from C any pb-literals in $pre_\perp(\alpha)$ and adds the resulting clause to $pre_\top(\alpha)$ (line 13). It then applies unit propagation to $pre_\perp(\alpha)$, removing any clause subsumed by a unit clause (line 14).

Finally, the algorithm constructs and returns the Boolean action model for each lifted action: the preconditions $pre_{bool}(\alpha)$ are conjunction of the clauses in $pre_\top(\alpha)$, and the effects $eff_{bool}(\alpha)$ are all pb-literals in $L(\alpha)$ that are not in $eff_\perp(\alpha)$ (lines 16-17).

Learning the Numeric Action Model. The numeric learning component of SVRAM consists of two parts: *precondition learning* and *effects learning*. Numeric preconditions approximate the true numeric preconditions of the action, when they exist, whereas numeric effects correspond to the actual effects inferred from successful applications.

Numeric effects are learned using linear regression, as in N-SAM, since effects can be inferred only from successful transitions. In contrast, precondition learning benefits from both positive (DB_+) and negative (DB^N_-) examples. For each lifted action, SVRAM constructs a set of hyperplanes whose induced *convex polytope* separates the numeric pre-states of positive examples from those of negative examples. This is achieved by iteratively generating hyperplanes using the Support Vector Machine (SVM) algorithm [15], as shown in Algorithm 2.

The input to this procedure consists of the dataset of successful state transitions DB_+, failed numeric transitions DB^N_-, and the set of pb-functions (Σ). For each $\alpha \in A$, SVRAM initializes the set $Pos(\alpha)$ with the numeric vectors corresponding to the values of pb-functions with grounding in pre-states of all successful transitions in $DB_+(\alpha)$ (line 5). Similarly, it initializes $Neg(\alpha)$ with the numeric vectors to the values of pb-functions with grounding in pre-states of all failed transitions attributed to failed numeric preconditions $DB^N_-(\alpha)$ (line 6). SVRAM then iteratively selects the vector $v^* \in$

$Neg(\alpha)$ with minimal Euclidean distance to the vectors in $Pos(\alpha)$ (line 8), constructs a hyperplane h that separates it from $Pos(\alpha)$ (line 9), and removes all correctly classified negative vectors from $Neg(\alpha)$ (lines 10–11). To compute h, we use the *hard-margin SVM*, whose objective function ensures complete separation between classes. In our case, it finds a hyperplane that separates v^* from all points in $Pos(\alpha)$. This process repeats until $Neg(\alpha)$ is empty. As a post-processing step, SVRAM removes all *dominated* hyperplanes from $\mathcal{H}$. Specifically, it removes any hyperplane h_i for which there exists a hyperplane h_j that correctly classifies all negative examples classified by h_i (and possibly additional ones; line 13). The numeric preconditions are defined by the remaining set of hyperplanes.

Example 1. *Figure 2 illustrates the hyperplanes generated by the algorithm in a 2-D plane. In the example, the positive examples (circles) are vectors satisfying* $-10 \leq x \leq 10$ *and* $-10 \leq y \leq 10$. *The negative examples, marked with 'X', are the rest of the vectors. The SVRAM algorithm generates four hyperplanes that correctly separate the positive and negative examples.*

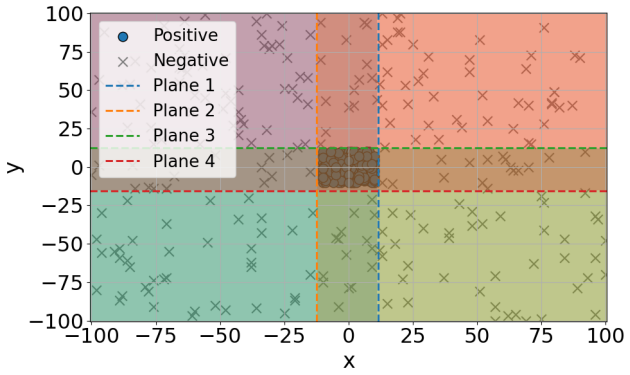

Figure 2: Visual example of the hyperplane separation using the SVM algorithm in a 2-D plane.

Theorem 2 (Runtime Complexity). *The runtime complexity of the numeric preconditions learning process is* $O(|A| \cdot (|DB_+(\alpha)| \cdot |DB^N_-(\alpha)| \cdot |\Sigma(\alpha)| + |DB^N_-(\alpha)|^2))$.

Proof. Initializing the datasets $Pos(\alpha)$ and $Neg(\alpha)$ requires time linear in $O(|DB_+(\alpha)|)$ and $O(|DB^N_-(\alpha)|)$, respectively. Identifying v^* has time complexity $O(|DB_+(\alpha)| \cdot |DB^N_-(\alpha)| \cdot |\Sigma(\alpha)|)$. Running the SVM algorithm is performed using the efficient LIBLINEAR implementation [12], with complexity $O(|DB_+(\alpha)| \cdot |\Sigma(\alpha)|)$. Classifying $Valid_{neg}(\alpha)$, filtering the set $Neg(\alpha)$, and adding the resulting hyperplane to the set $\mathcal{H}$ requires $O(|\Sigma(\alpha)| \cdot |DB^N_-(\alpha)|)$. The number of hyperplane generation iterations is bounded by $O(|DB^N_-(\alpha)|)$, since in the worst case, only one vector is correctly classified in each iteration. Dominance assertion between vectors requires comparing all correctly classified negative examples of the two hyperplanes, resulting in a worst-case complexity of $O(|DB^N_-(\alpha)|^2)$. The entire process is then repeated for every lifted action $\alpha \in A$. Therefore, the overall runtime complexity of the numeric precondition learning process is: $O\left(|A| \cdot \left(|DB_+(\alpha)| \cdot |DB^N_-(\alpha)| \cdot |\Sigma(\alpha)| + |DB^N_-(\alpha)|^2\right)\right)$. □

3.3 Informative Action Selection

NOAM is goal-oriented, in the sense that it first attempts to find a plan using M_{safe} or M_{op} and execute it. If no plan is found with either model, or if execution fails due to an inapplicable action, NOAM switches to exploration. Although random exploration is possible, it has drawbacks. In most states, many actions are inapplicable, so random exploration may waste time attempting actions already known to be inapplicable. Moreover, many applicable actions are already known to succeed and thus provide limited learning value. Therefore, NOAM prioritizes actions expected to yield new information and thus, improve the learned models. Below, we describe a heuristic for selecting such informative actions.

Definition 1 (Informative Action). *Given a planning state* s *and an action* $a = \langle \alpha, b_\alpha \rangle$, *we say that an action is non-informative if one of the following conditions holds: (1)* a *is applicable in* s *according to* M_{safe}, *(2)* a *is not applicable in* $s|_F$ *according to* M_{op}, *and (3) the convex hull induced by augmenting the vector* $v_s = \{s|_X(pf(x)) \mid x \in \Sigma(\alpha)\}$ *includes a known negative example from* $DB^N_-(\alpha)$.

An action satisfying condition (1) is guaranteed to be applicable in s, due to the safety of M_{safe}. Similarly, an action satisfying condition (2) is guaranteed to be inapplicable in s. The third condition in Def. 1 also implies that a is inapplicable in s, since numeric preconditions in the real domain are convex. Thus, if a is *informative*, its applicability in s is unknown without executing it. Therefore, executing it provides new relevant information. NOAM prioritizes informative actions during exploration. If it reaches a state where all actions are non-informative, it selects a random action. Importantly, NOAM updates both M_{safe} and M_{op} after each transition, as incorporating new data improves the identification of informative actions in subsequent steps.

Table 1: General statistics of the domains.

Domain	$\lvert A\rvert$	$\lvert F\rvert$	$\lvert X\rvert$	AVG($\lvert a\rvert$)	MAX($\lvert a\rvert$)
Counters	4	0	3	30	80
Depot	5	6	4	1,278	2,718
Driverlog	6	6	4	11,964	89,144
Farmland	2	1	2	280	800
Pogo	7	5	5	1,442	1,442
Sword	5	4	3	1,370	1,370
Satellite	5	8	6	505	1,734
Sailing	8	1	3	229	375
Rovers	10	26	2	3,168	10,800

4 EXPERIMENTAL RESULTS

We empirically evaluated the performance of NOAM on numeric planning domains satisfying two conditions: (1) the domains include only actions with linear preconditions and effects, and (2) a problem generator exists for each domain. Specifically, we used the domains Depot, Driverlog, Rovers, and Satellite from the 3rd International Planning Competition (IPC3) [25]; Farmland and Sailing from [31]; the extended version of Counters from IPC 2023 [32, 38]; and two Minecraft crafting tasks, Wooden-Sword and Pogo-Stick (referred to later as Sword and Pogo, respectively),

from [7]. Table 1 details general statistics about these domains. The columns "$|A|$", "$|F|$", and "$|X|$" denote the number of lifted actions, lifted fluents, and numeric functions, respectively, and the columns "AVG($|a|$)" and "MAX($|a|$)" report the average and maximum number of grounded actions per problem, respectively. Train and test problem instances were generated for each domain using [1]. To ensure efficient evaluation, we filtered out problem instances unsolvable by both ENHSP [31] and Metric-FF [16] planners using the ground truth action model in each domain, denoted M^*.

As our focus is on solving the problems successfully rather than optimally, both planners were configured for satisfying planning. ENHSP was run with Greedy Best-First Search, the MRP heuristic, helpful actions, and helpful transitions ("sat-hmrphj" configuration). Metric-FF used its default configuration: Enforced Hill Climbing followed by Breadth-First Search. Plan applicability was validated using VAL [17]. All tools, including VAL, used a numeric threshold of 0.1. To construct the trajectory dataset, we used ENHSP and Metric-FF to solve the generated training problems with a five-minute time limit per problem, retaining only solved instances. The dataset comprises 100 problems per domain, split 80:20 between online model learning and test validation. This procedure was repeated using five-fold cross-validation, and results are averaged across folds. If NOAM fails to solve a planning problem using both planners, the agent is switched to an exploration mode until it completes 100 successful steps or reaches 50,000 failed attempts. Experiments were conducted on a 92-machine cluster, each equipped with an AMD EPYC 7763 CPU (256 hyperthreads) and 1000GB of RAM. Each domain ran on a single thread with a 50GB memory limit. We set the random seed of all experiments to be 42 for reproducibility purposes.

As this is the first work on online learning of numeric action models, no natural baseline exists. We therefore focus on comparing the safe and optimistic models returned by NOAM and on evaluating its overall behavior. Comparison with offline learning algorithms is not appropriate, as they provide no mechanism for action selection.[2] The code of the NOAM algorithm and the dataset used in the experiments are available in the Github repository: https://github.com/SPL-BGU/numeric-online-action-model-learning.

4.1 Model Correctness Results

To evaluate the correctness of the learned models, we created two types of evaluation trajectories. First, we solved the test-set problems using the ground-truth domain and generated trajectories from the resulting plans. Since these trajectories may not cover all actions, we also generated additional trajectories by performing 500 random actions for each test problem. To assess correctness in inapplicable states, 40% of the random actions were inapplicable under the ground-truth model (M^*). This increased proportion of inapplicable actions creates states in which some actions in M_{op} may appear applicable, although they are not under M^*, as most actions are inapplicable under both models. We denote the combined set of trajectories by $\hat{T}$.

Let $\hat{M}$ be the learned action model, i.e., M_{safe} or M_{op}. For every lifted action α and given a state transition $\langle s, a, s' \rangle$, if s is applicable under $\hat{M}$, it is added to $app_{\hat{M}}(\alpha, s)$; if applicable under M^*, it is added to $app_{M^*}(\alpha, s)$. We evaluate the correctness of the learned action model using three metrics, following the action model learning evaluation framework defined in [37] and [29]: (i) the preconditions' precision and recall, (ii) the Boolean effects' precision and recall and (iii) the numeric effects' Mean Squared Error (MSE). We measure the preconditions precision and recall using the True Positive (TP), False Positive (FP), and False Negative (FN) values as follows:

- $\text{TP}_{pre}(\alpha) := \sum_{\langle s,a,s' \rangle} app_{M^*}(a, s) \wedge app_{\hat{M}}(a, s)$
- $\text{FP}_{pre}(\alpha) := \sum_{\langle s,a,s' \rangle} \neg app_{M^*}(a, s) \wedge app_{\hat{M}}(a, s)$
- $\text{FN}_{pre}(\alpha) := \sum_{\langle s,a,s' \rangle} app_{M^*}(a, s) \wedge \neg app_{\hat{M}}(a, s)$

Using TP, FP, and FN we compute the precision and recall:

$$P_{pre}(\alpha) = \frac{TP_{pre}(\alpha)}{TP_{pre}(\alpha) + FP_{pre}(\alpha)} \tag{1}$$

$$R_{pre}(\alpha) = \frac{TP_{pre}(\alpha)}{TP_{pre}(\alpha) + FN_{pre}(\alpha)} \tag{2}$$

Note that P_{pre} and R_{pre} are computed for the action as a whole, considering both Boolean and numeric preconditions. To evaluate the Boolean effects, we compared the predicted post-state $a_{\hat{M}}(s)|_F$ with the observed post-state $s'|_F$ in each successful transition in $\hat{T}$, as follows:

- $\text{TP}_{eff}(\alpha, s) := (a_{\hat{M}}(s)|_F \setminus s|_F) \cap (s'|_F \setminus s|_F)$
- $\text{FP}_{eff}(\alpha, s) := (a_{\hat{M}}(s)|_F \setminus s|_F) \setminus (s'|_F \setminus s|_F)$
- $\text{FN}_{eff}(\alpha, s) := (s'|_F \setminus s|_F) \setminus \neg(a_{\hat{M}}(s)|_F \setminus s|_F)$

Based on these values, we computed the effect precision ($P_{eff}(\alpha)$) and recall ($R_{eff}(\alpha)$). To quantify the correctness of the numeric effects, we also measured the Mean Squared Error (MSE) between the expected and observed numeric projection of the post-states. The P_{eff}, R_{eff} and MSE_{eff} values were only measured on transitions $\langle s, a, s' \rangle$ where a is applicable in s according to both M^* and $\hat{M}$.

Table 2 presents the results for each domain for the safe (M_{safe}) and optimistic (M_{op}) models generated by NOAM. P^{eff}, R^{eff} were always one. Thus, we omitted these results from Table 2. All results are reported for models obtained after completing all 80 episodes and are averaged across 5 folds. Since M_{safe} is guaranteed to be safe, P^{pre} for M_{safe} is consistently 1. In contrast, M_{op} is not safe, resulting in $P^{pre} < 1$ in some domains. This occurs primarily due to inaccurate numeric preconditions. For instance, in the COUNTERS domain, the action *decrement* has the true precondition `(>= (- (value ?c) (rate_value ?c)) 0)`. However, M_{op} learned an approximate condition: `(>= (+(*(value ?c) 0.52) (+(*(rate_value ?c)-0.32) (*(max_int) 0.01))) -0.29)`, which is imprecise and may result in the action being applied in invalid states.

In general, increasing the number of observations is expected to improve the precision of M_{op}. However, there is no fixed number of observations after which perfect P^{pre} is guaranteed, since numeric state variables have an uncountable set of possible values. It may be possible to derive PAC-style guarantees, but this requires further research.

In terms of R^{pre}, M_{op} has significantly higher than M_{safe} with a maximal difference of 90% in the Satellite domain. This is expected, as the preconditions learned by N-SAM are conservative and prevent actions from being applied in many states where they could be. Regarding MSE, we observed that the MSE_{eff} was zero in all domains except SWORD. In general, both M_{safe} and M_{op} learn

[2] In the ablation study in Table 3, we compare NOAM with N-SAM, an offline algorithm, using trajectories generated by random walks.

numeric effects using the same method (regression). Thus, the difference in the Sword domain is intriguing. A deeper investigation revealed that this difference occurred in the actions *craft_stick*, *craft_wooden_sword*, and *break*. For *craft_stick* action, its effects include `(increase (count_stick_in_inventory) 4)`. In our training episodes, this action was always executed when the number of sticks was zero. Therefore, the learned model instead inferred the effect `(assign (count_stick_in_inventory) 4)`—replacing the value rather than incrementing it. This was mitigated by M_{safe} by including the precondition `(= (count_stick_in_inventory) 0)`, preventing the learned assignment from producing incorrect values. M_{op} does not include this constraint, thus the learned model applies the assignment in states where it should not. Similar behavior was observed for the other actions, yielding inaccurate values and a high MSE_{eff} of 3.77.

Table 2: Semantic evaluation and problems solved.

	P^{pre}		R^{pre}		% Problems Solved (S/N/T/I)	
Domain	M_{safe}	M_{op}	M_{safe}	M_{op}	M_{safe}	M_{op}
Counters	**1.00**	0.97	0.79	**0.98**	**97**/3/0/0	**97**/0/0/3
Depots	1.00	1.00	0.24	**0.99**	71/2/27/0	**94**/0/5/1
Driverlog	1.00	1.00	0.52	**1.00**	53/0/47/0	**100**/0/0/0
Farmland	**1.00**	0.99	0.41	**0.83**	**95**/1/0/0	91/0/0/5
Pogo	1.00	1.00	0.36	**0.93**	**81**/17/2/0	24/0/0/76
Rovers	1.00	1.00	0.68	**0.99**	50/46/4/0	**88**/0/12/0
Sailing	1.00	1.00	0.73	**0.95**	**71**/0/29/0	63/0/20/17
Satellite	**1.00**	0.99	0.07	**0.97**	24/15/61/0	**71**/0/27/2
Sword	**1.00**	0.99	0.22	**0.96**	**96**/4/0/0	93/0/0/7

4.2 Problem-Solving Results

The column "% Problems Solved (S/N/T/I)" in Table 2 reports the results of solving the test problems with our planner portfolio using M_{safe} and M_{op}. For each domain and action model, the table shows the average percentage of test problems that were solved and validated (S), reported unsolvable (N), timed out (T), or yielded plans inapplicable in the real domain (I). In five out of the nine experimented domains,[3] M_{op} had the same or significantly higher solving rates than M_{safe}. For example, in Driverlog , using M_{op} allowed solving all problems while only 53% were solved when using M_{safe}. This was not always the case, however, and in some domains using M_{safe} was better. A prominent example is the Pogo domain, where 76% of the problems had inapplicable solutions when using M_{op} while using M_{safe} allowed solving 81%. In general, since M_{safe} is safe, it never returns an inapplicable solution. Thus, while the results are not conclusive, we can say that in most cases we observed a tradeoff between the higher ability of M_{op} to solve problems and the lower reliability of the plan it outputs compared to those created with M_{safe}. Note that NOAM can be seen as an effective hybrid that enjoys both worlds: it first tries to plan with M_{safe} and then switches to M_{op} if needed.

To gain a deeper insight into the behavior of NOAM, we measured its ability to solve problems during training. We distinguished between the following outcomes NOAM may reach in an episode: *solved safe*, where the goal was reached by following a plan generated using M_{safe}; *solved*, where the goal was reached either by following the plan generated using M_{op} or during exploration; *inapplicable*, when a plan was found but its execution failed due to inapplicable actions, and *not solved*, encompassing instances where the planner times out and instances where no valid plan could be generated using the learned model.

Figure 3 shows rolling average results (window size of 5 episodes) across the evaluated domains as a function of training episodes. The red, blue, yellow, and green regions indicate the proportion of episodes *not solved*, *inapplicable*, *solved*, and *solved safe*, respectively. As can be seen, in all domains, the average rate of problems that were either solved safely or solved increases with the number of episodes. Notably, Driverlog and Farmland reach perfect performance within 20 episodes. In most domains, the average rate of problems solved by M_{safe} increases over time. However, in some cases, such as Driverlog, the success rate starts to deteriorate at some point. This may be due to the increasing complexity of the safe models, which leads to planner timeouts. The results also show that NOAM is less effective in the Rovers, Sailing, and Satellite domains. Indeed, these domains are more difficult to learn than the other domains. Rovers is more difficult to learn than other domains because it has significantly more lifted actions (10) and lifted fluents (26), as shown in Table 1. Sailing is difficult to learn compared to other domains, mainly due to its large number of lifted actions (8) and specifically the `save_person` action, which has relatively more complex preconditions that require four inequalities. Lastly, the Satellite domain is harder to learn compared to the other domains because it has relatively many lifted fluents (8) and functions (6), which results in learning complex conditions that limit the planner's ability to solve planning problems in this domain with the learned model.

4.3 Ablation Study

Next, we performed an ablation study comparing (i) solving with N-SAM and exploring randomly on failure; (ii) solving with SVRAM and exploring randomly on failure; (iii) solving with SVRAM and using the informative action selection on failure; (iv) a pipeline of N-SAM and SVRAM with random exploration on failure; and (v) the full NOAM algorithm. We denote these five variants as N-SAM, SVRAM, SVRAM +Info., N-SAM +SVRAM, and NOAM. Table 3 presents the average rate of problems solved across all training episodes and five folds cross-validation, for each domain and ablation variant of NOAM listed above. As expected, NOAM achieves the best results in most of the experimented domains, suggesting that all its components provide value. Using N-SAM and then performing random exploration yields the worst results in all the experimented domains. Using SVRAM +Info. resulted in a high problem-solving rate in domains with challenging goal actions, e.g., Satellite, Rovers, Pogo , and Sword. Finally, NOAM outperforms all other versions except in Rovers and Satellite. In Rovers domain, using a random walk resulted in better exploration of the environment, yielding an 11% increase compared to using the informative action selection. In both the Rovers and Satellite domains, the problem-solving rate using SVRAM +Info. outperforms that of NOAM. Upon investigation, we found that this

[3] In Farmland, 4% of the problems resulted in plans where the numeric goal was not achieved due to numeric imprecision, but as this only occurred in this domain, thus, these results were omitted.

Table 3: Ablation study: avg. rate of the problems solved.

Domain	N-SAM	SVRAM	SVRAM + Info.	N-SAM + SVRAM	NOAM
Counters	0.16	0.91	0.91	0.93	**0.94**
Depots	0.59	0.93	0.93	0.93	**0.94**
Driverlog	0.50	0.97	0.97	0.97	**0.98**
Farmland	0.92	0.96	0.96	**0.98**	**0.98**
Pogo	0.03	0.06	0.58	0.24	**0.75**
Rovers	0.43	0.63	0.66	**0.67**	0.56
Sailing	0.00	0.15	0.34	0.30	**0.67**
Satellite	0.03	0.43	**0.44**	0.28	0.40
Sword	0.78	0.91	0.98	0.89	**0.99**

difference arises in cases where the same problems were solved using M_{safe} in NOAM and using M_{op} in SVRAM +Info. While actions in plans generated by M_{safe} are non-informative by definition, those in plans produced by M_{op} can be informative and can thus be used to further improve the models. This observation suggests that solving problems using M_{op} may be beneficial even when M_{safe} is sufficient, as it trades the risk of failed execution for the potential to gain additional informative examples that can improve the learned models. Overall, these results emphasize the significance of each component in NOAM, namely, using SVRAM, combining it with N-SAM, and using informative actions to guide exploration.

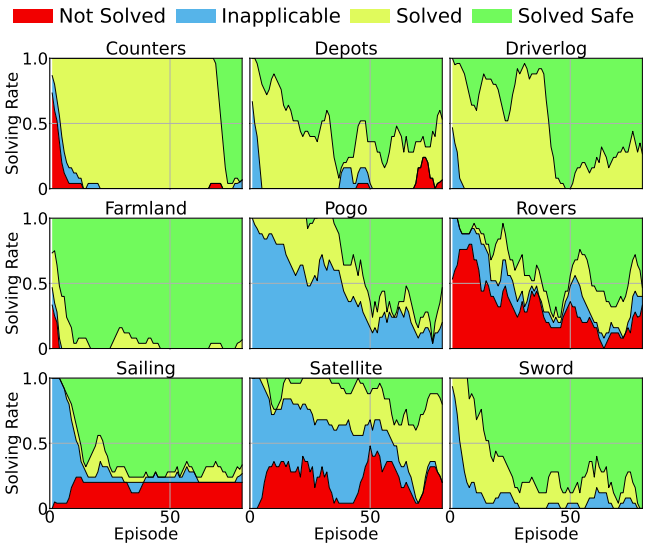

Figure 3: Rolling average of the problems solved by NOAM.

5 DISCUSSION

Deep Reinforcement Learning (DRL) algorithms, such as PPO [33] and DQN [26], learn policies that maximize expected cumulative reward through repeated interaction with the environment. These methods are broadly applicable: they do not assume deterministic action models and can handle stochastic effects (MDPs) and partial observability (POMDPs). Thus, DRL addresses a fundamentally different problem than NOAM, which aims to learn a numeric action model rather than a policy.

Learning action models offers several benefits. First, an action model enables solving a domain using any numeric planner, an area of ongoing research. Second, action models allow validation of plans generated by external sources, including humans, domain-specific planners, and RL algorithms. Third, they support solving multiple problems within the same domain by changing only the planner's goal. Finally, the action models learned by NOAM are *lifted*, allowing direct application to problems with different numbers of objects and enabling generalization from smaller to larger instances. In fact, our experimental results include problems of varying sizes (see Table 1).

Nevertheless, an interesting question is how NOAM compares to RL in terms of problem-solving performance as a function of training set size. To explore this question, we performed the following experiment comparing NOAM with the PPO [33] algorithm. PPO, however, requires a fixed network input size. Therefore, we restricted this evaluation to two domains—Sword and Pogo—where the training and test sets contain the same number of objects.[4]

Specifically, we used PPO with action masking [39] from `stable_baselines3`,[5] tuned with Optuna [4] on 10 validation maps per domain over 30 trials. For fair comparison with NOAM, PPO was trained under the same settings using five-fold cross-validation, evaluated on the 20 hold-out problems, with each episode limited to 100 successful steps or 50,000 failed attempts. The results show that PPO achieves success rates of 61% on Sword and 1% on Pogo. In comparison, the two models generated by NOAM outperform PPO: $M_{\textbf{op}}$ achieves 93% on Sword and 24% on Pogo, while $M_{\textbf{safe}}$ reaches 96% and 81% on the same domains. Effectively applying RL to these problems, as well as the other domains in the evaluation benchmark, remains an open challenge for future work.

6 CONCLUSION AND FUTURE WORK

In this work, we proposed NOAM, the first online algorithm for learning numeric action models. We introduced SVRAM, a novel optimistic numeric action-model learner that uses both positive and negative examples, showed how it integrates with N-SAM, a safe numeric action-model learner, and presented a new information-guided exploration scheme. We evaluated our approach on nine numeric benchmark domains and demonstrated that NOAM quickly learns effective action models in most domains. Our results also show that the model learned by SVRAM is often more effective than that learned by N-SAM. As future work, we intend to explore more advanced methods for exploration and to improve the expressiveness by adding numeric functions, e.g., sine and cosine.

ACKNOWLEDGMENTS

This research was funded by ISF grants No. 1238/23 to Roni Stern. Additional support was provided by Israel's Ministry of Innovation, Science and Technology (MOST) under Grant No. 1001706842, in

[4]Some DRL approaches, such as those based on Graph Neural Networks, aim to handle varying input sizes. However, these methods offer limited generalization to significantly different problem sizes and are not expected to outperform training on fixed-size inputs. Restricting our evaluation to problems with the same number of objects even favors DRL, making the comparison conservative.
[5]https://stable-baselines3.readthedocs.io

collaboration with the Israel National Road Safety Authority and Netivei Israel, awarded to Shahaf Shperberg. Brendan Juba was supported by NSF award IIS-1942336.

REFERENCES

[1] [n.d.]. GitHub - SPL-BGU/pddl-problem-generator: Repository that holds PDDL problem generators that are not available in the classical problem generator — github.com. https://github.com/SPL-BGU/pddl-problem-generator. [Accessed 18-07-2025].

[2] Diego Aineto, Sergio Jiménez Celorrio, and Eva Onaindia. 2019. Learning action models with minimal observability. *Artificial Intelligence* 275 (2019), 104–137.

[3] Diego Aineto and Enrico Scala. 2024. Action Model Learning with Guarantees. In *International Conference on Principles of Knowledge Representation and Reasoning*. 801–811.

[4] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2623–2631.

[5] Eyal Amir and Allen Chang. 2008. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research* 33 (2008), 349–402.

[6] Masataro Asai and Alex Fukunaga. 2018. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *AAAI conference on artificial intelligence*.

[7] Yarin Benyamin, Argaman Mordoch, Shahaf Shperberg, Wiktor Piotrowski, and Roni Stern. 2024. Crafting a pogo stick in minecraft with heuristic search. In *Proceedings of the International Symposium on Combinatorial Search*, Vol. 17. 261–262.

[8] Michal Čertickỳ. 2014. Real-time action model learning with online algorithm 3 sg. *Applied Artificial Intelligence* 28, 7 (2014), 690–711.

[9] Rohan Chitnis, Tom Silver, Joshua B Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. 2021. Glib: Efficient exploration for relational model-based reinforcement learning via goal-literal babbling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 11782–11791.

[10] Stephen Cresswell and Peter Gregory. 2011. Generalised domain model acquisition from action traces. In *International Conference on Automated Planning and Scheduling (ICAPS)*. 42–49.

[11] Stephen Cresswell, Thomas McCluskey, and Margaret West. 2013. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review* 28, 2 (2013), 195–213.

[12] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *the Journal of machine Learning research* 9 (2008), 1871–1874.

[13] Maria Fox and Derek Long. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20 (2003), 61–124.

[14] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. 1998. *Pddl — the planning domain definition language*. Technical Report. Yale Center for Computational Vision and Control.

[15] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. 1998. Support vector machines. *IEEE Intelligent Systems and their applications* 13, 4 (1998), 18–28.

[16] Jörg Hoffmann. 2003. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *Journal of Artificial Intelligence Research* 20 (2003), 291–341.

[17] Richard Howey, Derek Long, and Maria Fox. 2004. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 294–301.

[18] Mu Jin, Zhihao Ma, Kebing Jin, Hankz Hankui Zhuo, Chen Chen, and Chao Yu. 2022. Creativity of AI: Automatic symbolic option discovery for facilitating deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 7042–7050.

[19] Brendan Juba, Hai S. Le, and Roni Stern. 2021. Safe Learning of Lifted Action Models. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 379–389.

[20] Brendan Juba and Roni Stern. 2022. Learning Probably Approximately Complete and Safe Action Models for Stochastic Worlds. In *AAAI Conference on Artificial Intelligence*.

[21] Rushang Karia, Pulkit Verma, Gaurav Vipat, and Siddharth Srivastava. 2023. Epistemic Exploration for Generalizable Planning and Learning in Non-Stationary Stochastic Settings. In *NeurIPS 2023 Workshop on Generalization in Planning*.

[22] Leonardo Lamanna, Alessandro Saetti, Luciano Serafini, Alfonso Gerevini, and Paolo Traverso. 2021. Online Learning of Action Models for PDDL Planning. In *IJCAI*. 4112–4118.

[23] Leonardo Lamanna and Luciano Serafini. 2024. Action Model Learning from Noisy Traces: a Probabilistic Approach. In *ICAPS*. AAAI Press, 342–350.

[24] Hai S Le, Brendan Juba, and Roni Stern. 2024. Learning Safe Action Models with Partial Observability. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 20159–20167.

[25] Derek Long and Maria Fox. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20 (2003), 1–59.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[27] Argaman Mordoch, Brendan Juba, and Roni Stern. 2023. Learning Safe Numeric Action Models. In *AAAI*. AAAI Press, 12079–12086.

[28] Argaman Mordoch, Enrico Scala, Roni Stern, and Brendan Juba. 2024. Safe learning of pddl domains with conditional effects. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 34. 387–395.

[29] Argaman Mordoch, Shahaf S. Shperberg, Roni Stern, and Brendan Juba. 2024. Enhancing Numeric-SAM for Learning with Few Observations. In *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

[30] Jun Hao Alvin Ng and Ronald PA Petrick. 2019. Incremental Learning of Planning Actions in Model-Based Reinforcement Learning. In *IJCAI*. 3195–3201.

[31] Enrico Scala, Patrik Haslum, and Sylvie Thiébaux. 2016. Heuristics for Numeric Planning via Subgoaling. In *IJCAI*. 3228–3234.

[32] Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramirez. 2020. Subgoaling techniques for satisficing and optimal numeric planning. *Journal of Artificial Intelligence Research* 68 (2020), 691–752.

[33] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[34] José Á Segura-Muros, Raúl Pérez, and Juan Fernández-Olivares. 2021. Discovering relational and numerical expressions from plan traces for learning action models. *Applied Intelligence* (2021), 1–17.

[35] Sarath Sreedharan and Michael Katz. 2023. Optimistic exploration in reinforcement learning using symbolic model estimates. *Advances in Neural Information Processing Systems* 36 (2023), 34519–34535.

[36] Roni Stern and Brendan Juba. 2017. Efficient, Safe, and Probably Approximately Complete Learning of Action Models. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 4405–4411.

[37] Roni Stern, Leonardo Lamanna, Argaman Mordoch, Yarin Benyamin, Pascal Lauer, Brendan Juba, Gregor Behnke, Christian Muise, Pascal Bercher, Mauro Vallati, Kai Xi, Omar Wattad, and Omer Eliyahu. 2025. Evaluating Planning Model Learning Algorithms. In *Workshop on Knowledge Engineering for Planning and Scheduling (KEPS) at ICAPS*.

[38] Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fišer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, et al. 2024. The 2023 International Planning Competition.

[39] Cheng-Yen Tang, Chien-Hung Liu, Woei-Kae Chen, and Shingchern D. You. 2020. Implementing action mask in proximal policy optimization (PPO) algorithm. *ICT Express* 6, 3 (2020), 200–203.

[40] Pulkit Verma, Rushang Karia, and Siddharth Srivastava. 2023. Autonomous capability assessment of sequential decision-making systems in stochastic settings. *Advances in Neural Information Processing Systems* 36 (2023), 54727–54739.

[41] Kai Xi, Stephen Gould, and Sylvie Thiébaux. 2024. Neuro-Symbolic Learning of Lifted Action Models from Visual Traces. In *International Conference on Automated Planning and Scheduling*. 653–662.

[42] Qiang Yang, Kangheng Wu, and Yunfei Jiang. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence* 171, 2-3 (2007), 107–143.