ECAI 2024
U. Endriss et al. (Eds.)
© 2024 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA241000

A Lazy Approach to Neural Numerical Planning with Control Parameters

René Heesch[a,*], Alessandro Cimatti[b], Jonas Ehrhardt[a], Alexander Diedrich[a] and Oliver Niggemann[a]

[a]Helmut Schmidt University, Hamburg, Germany
[b]Fondazione Bruno Kessler, Trento, Italy

Abstract. In this paper, we tackle the problem of planning in complex numerical domains, where actions are indexed by control parameters, and their effects may be described by neural networks. We propose a lazy, hierarchical approach based on two ingredients. First, a Satisfiability Modulo Theory solver looks for an abstract plan where the neural networks in the model are abstracted into uninterpreted functions. Then, we attempt to concretize the abstract plan by querying the neural network to determine the control parameters. If the concretization fails and no valid control parameters could be found, suitable information to refine the abstraction is lifted to the Satisfiability Modulo Theory model. We contrast our work against the state of the art in NN-enriched numerical planning, where the neural network is eagerly and exactly represented as terms in Satisfiability Modulo Theories over nonlinear real arithmetic. Our systematic evaluation on four different planning domains shows that avoiding symbolic reasoning about the neural network not only leads to substantial efficiency improvements, but also enables their integration as black-box models.

1 Introduction

Planning is a fundamental step in the flexible operation of real-world domains, where complex physical processes must be sequentially controlled in order to achieve different states or products. For example, planning for factory automation requires the ability to streamline the proper tasks (first blending, then cooking) and to define the respective control parameters (e.g., temperature, humidity) in a recipe-based production environment (e.g., electroplating). In this context, some features, e.g., the evolution of physical processes, can hardly be modeled in a rule-based, symbolic representation formalism. However, in many cases, a representation based on neural networks (NNs) can be learned by analyzing the actual simulation traces [8].

We refer to this general setting first described in Heesch et al. [14] for the specialized field of production planning as NN-enriched Numerical Planning with Control Parameters (N3PCP). In [14], the authors addressed the N3PCP problem with an *eager* and *exact* encoding into Satisfiability Modulo Theories (SMT), with the NN being logically represented in the form of exact SMT terms. Unfortunately, this approach suffers from several problems. First, the NN may come as a black box, where the network's architecture, weights, and biases are unknown, so that it is impossible to convert it into logical terms. Second, even if the NN is a full-access white box model, reasoning symbolically about it may be challenging due to the NNs complexity. NNs with transcendental activation functions, e.g., sigmoid activation functions, require the non-standard SMT theory of nonlinear real arithmetic, for which only a limited number of solvers exist [4, 9]. Even with piecewise linear activation functions, the size of the NN may cause inefficiencies, despite the recent developments of dedicated verification approaches [16, 11].

In this paper, we propose the **L**azy **N**eural **P**laner (LNP). The LNP is a hierarchical, lazy approach to N3PCP, where the NNs are not symbolically modeled as part of the SMT encoding of the (bounded) planning problem, but are rather abstracted as (partly axiomatized) uninterpreted functions. The analysis of the NNs is delayed until the SMT solver finds a valid assignment, i.e., an abstract plan. When an abstract plan is found, the particular values to the NN's inputs/outputs are extracted in order to concretize the plan. This concretization checks if the values guessed in the abstract planning phase can actually be realized for some control parameter valuation. If not, they are blocked by way of conflict clauses, and the abstract search is resumed.

Our approach can be seen as a variant of the "incremental linearization" approach [4] in the SMT paradigm [1], where the concretization checks and the subsequent lifting are carried out by a dedicated theory solver. In our approach, the NN can be *executed* instead of being analyzed with formal reasoning.

We differentiate three types of NN models: *(i) full-access* white box models, whose architecture, weights, and biases are known, *(ii) function-access* black box models, where the architecture, weights, and biases are unknown, but specific functionalities, such as back-propagation, can be accessed via dedicated functions, *(iii) no-access* black box models, where the architecture, weights, and biases are unknown and no such functional interfaces are available, reducing the model to an executable with a pure input-output interface.

For the first two scenarios, given a pair of concrete states, gradient-based optimization techniques are employed in order to find suitable values for the control parameters. In the third scenario, an uninformed search algorithm is used to determine these values, enabling the integration of arbitrary network types and architectures.

In order to reduce the production of spurious counterexamples in the abstract state, we limit the search space by applying a variant of static learning that aims at conjecturing which variables are affected by a given NN and which ranges are possible for the variations of the various fluents. This information can be derived during the training of the NN or from running the NN on a given data set (either the one used for the training of the network, if available, or a sampling of the data space). Within this paper, we will focus on the former approach.

* Corresponding Author. Email: rene.heesch@hsu-hh.de

The proposed approach is correct, i.e., if it finds a plan it is indeed a solution plan, but incomplete, i.e., it may be unable to find a solution plan when one exists. We trade off the completeness of the eager, symbolic NN encoding into SMT [14] for generality (since checking via execution is basically agnostic with respect to the nature of the activation functions) and efficiency (symbolic reasoning about NNs is still very computationally expensive).

The planning approach has been instantiated in a framework where the domain is based on a vector representation, i.e., the state is constituted by a set of discrete and real-valued variables, similar to an infinite state symbolic transition system. In principle, N3PCP problems could be concretized in a fluent-based representation with PDDL-style modeling. However, from a representational standpoint, modeling the influence of NNs in PDDL is very hard.

We carried out an experimental evaluation based on multiple planning problems from four different numerical planning domains with control parameters, including both piecewise linear and nonlinear transcendental activation functions.

For those with nonlinear activations, as well as for those with piecewise linear activation functions, the latter can be encoded into the SMT theory QF-LRA, we show that the proposed approach is significantly faster and more effective than [14].

As the limitation of the search space by application of the static learning of NN constraints leads to incompleteness, we investigated the effectiveness of different styles of integration into the abstracted problem. Furthermore, we demonstrate that our approach can handle not only full-access white box, but also no-access black box models or executables that are only accessible via a pure input-output interface. Finally, we analyzed the runtime behavior with increasing complexity of the NNs, showing that the runtime increases linear at most.

Structure of the paper: Section 2 presents some background on SMT and NNs. Section 3 defines NN-enriched numerical planning with control parameters. Section 4 presents our novel planning algorithm, which integrates an SMT-based planning approach with pre-trained, encapsulated machine learning models. Section 5 provides an overview of existing planning approaches capable of addressing numerical planning problems with control parameters. Section 6 delivers a comprehensive evaluation of our algorithm, including comparisons with other state-of-the-art approaches. Finally, we draw some conclusion and outline the directions for future work in Section 7.

2 Background

2.1 Logical Preliminaries

We work in the setting of first order logic. Terms are either constants or individual variables, or n-ary function symbols applied to n terms. Atoms are either propositional variables or n-ary predicates applied to n terms. Formulae are either atoms or the application of the standard boolean connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ to formulae, or the application of a quantifier $\exists, \forall$ to one or more variables and a formula. We adopt the standard semantic notions of interpretation, model, satisfiability, and validity [26].

We follow the Satisfiability Modulo Theories (SMT) paradigm, where the interpretation of a given set of symbols is restricted according to a given theory. In this paper, we consider the theory of nonlinear arithmetic with transcendental symbols (NTA). We have numerical constants, real-valued variables, the standard function symbols $+, -, \cdot, /$ and predicate symbols $<, \leq, >, \geq$. We also assume transcendental functions (e.g., $\tan$).

An SMT solver is a software tool that, given a formula, decides whether it is satisfiable or not (and may return a satisfying assignment or a proof of unsatisfiability). Most modern SMT solvers adopt a lazy paradigm: a SAT solver is used to enumerate satisfying assignments of the boolean abstraction of the formula, and theory solvers are used to check the satisfiability of the set of constraints corresponding to the assignment at hand. If the theory solver detects that the set of constraints is consistent, then a model is found. Otherwise, clauses are produced to block the generation of satisfying assignments failing for the same reason.

We use a symbolic representation for infinite state transition systems [5]. Given $\langle V, I(V), T(V, V')\rangle$, with the symbolic representation a state is an assignment to V. $I(V)$ is an SMT formula characterizing the set of initial states, i.e., a state s is initial if it satisfies I, written $s \models I$. T is the transition relation, with V and V' denoting the state before and after the transition, respectively. There is a transition from s to s' iff $s \cdot s' \models T$; we say that s' is a successor of s (and s a predecessor of s').

Given a state transition system Γ, a trace is a sequence of states $s_0, s_1, \ldots$ such that s_0 is initial and for all i s_{i+1} is a successor of s_i. Linear temporal logic [19] is used to express the properties of traces. The model checking problem $\Gamma \models \phi$, with ϕ a temporal formula, checks if all the paths of Γ satisfy ϕ. A number of verification problems exist, based on induction, interpolation and IC3, both for the finite-state and the infinite-state case. We focus on the bounded model checking [2] approach (BMC), that is based on the reduction of bounded reachability to satisfiability, by "unrolling" the transition relation on $k+1$ replicas of the state vector V. The formula

$$I(V_0) \wedge T(V_0, V_1) \wedge \ldots T(V_{k-1}, V_k) \wedge G(V_k) \tag{1}$$

is satisfiable iff there exists a path of k transitions from an initial state to a goal state in G. The path is the satisfying assignment to the $V_0, V_1, \ldots, V_k$ variables.

2.2 Neural Networks and Logical Paradigms

Neural Networks are the quintessential models for approximating complex functions, such as the transition relation $T(V, V')$, in Machine Learning [10]. The most basic variant, the Feedforward Neural Network, is defined as a mapping function $f(\mathbf{x}; \psi) = \mathbf{y}$ between an input $\mathbf{x}$ and an output $\mathbf{y}$, with learnable parameters ψ. $f(\mathbf{x})$ itself is a chain of functions that consist of a linear part $f^{(i)}(\mathbf{x}) = \Sigma \mathbf{w}^T \mathbf{x} + b$ with i describing the depth, $\mathbf{w}$ describing weights, and b describing biases, which is fed into a nonlinear part, the activation function, $g(\cdot)$. There is a wide variety of activation functions [18] ultimately allowing to approximate any continuous function, given the network has at least two layers [15]. NNs with at least two hidden layers are denoted as Deep Neural Networks [7]. In a supervised setting, training a NN uses gradient descent and backpropagation of a loss metric to adapt ψ in the NN.

The application of trained NNs together with logical paradigms primarily focuses on the verification of individual NNs with SMT. The NN verification problem refers to the challenge of certifying that a NN behaves in compliance with a specified property of interest, ρ [12]. Here, ρ is defined by the formula $\rho_{in} \Rightarrow \rho_{out}$, where ρ_{in} describes a property concerning the input of the NN, $\mathbf{x}$, and ρ_{out} relates to a property of the NN's output, $\mathbf{y}$. The mapping function $f(\mathbf{x}; \psi) = \mathbf{y}$, which includes architecture-specific characteristics along with activation functions, weights, and biases, is directly translated into an SMT formula F. By incorporating the property ρ, for which the NN is to be verified, into F an SMT solver can be utilized to prove the NN's compliance with this property or to show that there exists a counter-example for which F is falsified. Motivated by the verification domain, new and more efficient frameworks such as NeuralSAT, which is based on the same DPLL architecture as modern SMT solvers, have been developed [7]. However, these frameworks cannot handle nonlinear transcendental activation functions, and hence are limited to piecewise linear activation functions.

Approaches that extend NN verification into contexts like SMT-planning problems [14] also rely on the exact and eager encoding of the NN into SMT.

3 Problem Definition

An NN-enriched Numerical Planning with Control Parameters (N3PCP) problem is a tuple:

$$P = \langle B, N, A, \Theta, T, M, I, G \rangle \tag{2}$$

B and N denote the set of propositional and numerical state variables. These sets characterize the state-space S: each state $s_i \in S$ is uniquely defined by attributing specific values to the numerical variables within N, and by designating each variable in B as either true or false. Adopting a standard representation from symbolic model checking, a state is an assignment to $B \cup N$. Without loss of generality, we assume that B and N are ordered, and we denote with X their concatenation. The state-space S is the set of possible assignments to X. A denotes the set of actions, and Θ is the set of control parameters. An action describes the possible transitions from a state s_i to a state s'_i, given a valuation to the control parameters Θ. A characterizing feature of our framework is that actions may come with two possible representations: as symbolic transition function and/or executable transition function. The set T denotes the set of symbolic transition functions and M denotes the set of executable transition functions. The set of initial conditions is denoted as I. G represents the set of goal conditions, where the goal conditions and the preconditions of actions within NN-enriched planning problems span state spaces instead of explicit states.

A symbolic transition function describes – as standard in planning (e.g., PDDL, ANML) – the preconditions and effects in a logical language. An executable transition function returns, given a state and a control parameter valuation, a new state. Such executables could be neural networks, possibly in the form of black boxes, without the model being accessible.

Each action $a \in A$ is associated with a symbolic representation T_a, which is an SMT formula $T_a(X, \Theta, X')$, and/or with an executable representation M_a, which is an imperative-style, loop-free program mapping a vector X and a vector Θ to a next-state vector X'. We assume that there is a 1:1 correspondence between the symbolic variables and the corresponding vectors in the executable representation, and we use them interchangeably. In the following, we will refer to actions with an executable representation as *neural actions*.

From the symbolic representation of a transition T_a it is straightforward to generate a corresponding executable representation $toExec[T_a]$. The symbolic representation can be used, e.g., to generate artificial data. A NN trained on these data, can serve as executable representation. For each a we require that T_a and M_a agree on all the states, i.e., for every state s and control parameter valuation Θ $M_a(s, \Theta) = toExec[T_a](s, \Theta)$.

An action $a \in A$ may not always be applicable to all states in S, so that T_a and M_a are actually partial functions. We denote with ${S_{in}}^{(a)} \subseteq S$ the applicability set of a. The control parameters can be subject to similar restrictions. We denote with $LegalP_a(\Theta)$ the space of legal assignments to control parameters for action a. For example, each control parameter $\theta \in \Theta$ could be subject to an interval constraint of the form $lb \leq \theta \wedge \theta \leq ub$, where lb denotes the lower and ub denotes the upper bound.

4 A Hierarchical Approach to N3PCP

We propose a novel lazy planning approach to N3PCP, which incorporates pre-trained, encapsulated NNs without translating and integrating them directly into the SMT planning problem (cf. Figure 1). The basic idea is to delay the analysis of actions that are hard or impossible to deal with at the symbolic level, i.e., the ones that are represented by NNs, until the SMT solver finds a solution to the satisfiability problem on the abstract level. Therefore, on the abstract level, the satisfiability problem incorporates lifted, generalized information about the effects of neural actions. The integration of the lifted information allows clearly specifying the intermediate states between the actions. Based on these intermediate states, the encapsulated NNs are queried to concretize the abstract plan, determining the feasibility of the input-output evaluations, as well as the control parameters of the actions via optimization of the input.

The integration of the lifted information at the abstract level limits the solution space and increases the efficiency of the algorithm at the cost of completeness due to the potential incompleteness of this lifted information. To reduce this potential risk, we propose to extract the lifted information during the training process of the NNs, assuming that the NNs are trained on data, that represent the full capabilities of the neural action without any noise.

For modelling the N3PCP problem, we opted to modify the feature-vector-based state-space representation [14], rendering it applicable to NN3PCP problems more broadly. In this framework, each domain has an infinite n-dimensional state-space, where n is equivalent to the number of variables in X. Each state, $s_i \in S$, is represented by a feature-vector of length n, which assigns a value to each

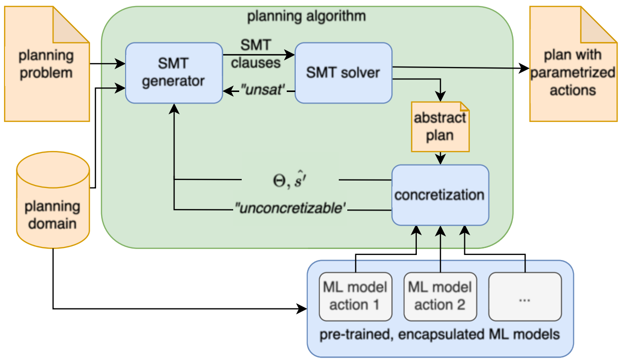

Figure 1. A high level view on our Lazy Neural Planner. The planning algorithm initially generates an abstracted satisfiability problem in SMT which is solved by the SMT solver, resulting in an abstract plan. Based on this abstract plan and the pre-trained, encapsulated ML models, the concretization either returns '*unconcretizable*' or the set of control parameters Θ and the updated state variables of the resulting state. These are either integrated into the satisfiability problem, checking for validity, or a blocking lemma is generated and added. If the set of control parameters is valid, the parameterized plan is returned.

of these variables.

In the following, we describe the process of encoding the N3PCP problem as a satisfiability problem, along with the integration of lifted information regarding neural actions and the interaction between the SMT solver and the encapsulated NNs (cf. 4.1). Subsequently, we introduce the concretization of the abstract plans via the trained NNs (cf. 4.2). Lastly, we describe the extraction of the lifted information from these NNs via an auxiliary function, which we call the *observer* function (cf. 4.3).

4.1 Planning Algorithm

To solve N3PCP problems, we propose the **L**azy **N**eural **P**lanner (LNP). The LNP initially transforms the planning problem P into a satisfiability problem in SMT denoted as F (cf. Alg. 1, line 1). For the set of propositional variables B, the set of numerical variables N and the initial I and goal G conditions, the corresponding entities within the satisfiability problem are generated directly. The actions A and the corresponding symbolic transition functions T and the trained NNs M are excluded at this point. The algorithm operates iteratively, starting with an assessment of whether the initial conditions, denoted as I, already meet the goal conditions, G and no action is required (cf. Alg. 1, line 2).

Algorithm 1 The **L**azy **N**eural **P**laner (LNP) takes an N3PCP problem as input and returns either a plan Γ or *no plan found*.

```
Require: N3PCP $P = \langle B, N, A, \Theta, T, M, I, G \rangle$
Ensure: Plan $\Gamma$ to $P$ or no plan found
 1: $F \leftarrow initialize(B, N, \Theta, I, G)$
 2: $solved = check\_satisfiability(I \wedge G)$
 3: while $\neg solved$ & $k \leq maxlength$ do
 4:     $A_n \leftarrow instantiate(A)$
 5:     $F \leftarrow add\_clauses(A_n)$
 6:     while $F == satisfiable$ do
 7:         $solved = True$
 8:         for all $M_a$ in $\Gamma$ do
 9:             $\Theta, \hat{s'} \leftarrow concretization(M_a, s, s')$
10:             if $check\_satisfiability(\Theta \wedge \hat{s'})! = sat$ then
11:                 $solved = False$
12:                 $F \leftarrow add\_clauses(blocking\_clauses(s\_model(F)))$
13:                 break
14:             end if
15:         end for
16:         if $solved == True$ then
17:             return Plan $\Gamma$ to $P$
18:         end if
19:     end while
20: end while
21: return no plan found
```

If not, it incrementally increases the number of actions. To enable multiple executions of actions, the algorithm utilizes instances of actions and the corresponding control parameters. For each action $a \in A$, a new instance is generated. All instances of the actions of this step i as well as the clauses, specifying their preconditions and their effects, are collected in the set A_i (cf. Alg. 1, line 4). This set of clauses is then added to the satisfiability problem F (cf. Alg. 1, line 5).

For actions with logically described effects, the algorithm integrates the mathematical expressions that describe their effects on each variable $x \in X$. The lifted information about the action's effects is provided by the *observer* procedure (Alg. 2) and integrated into the satisfiability problem.

The *observer* procedure returns the three vectors $\mathit{eff}_{change}{}^{(a)}$, $\mathit{eff}_{low}{}^{(a)}$ and $\mathit{eff}_{up}{}^{(a)}$ for each of the learned actions (cf. Alg. 2). The LNP initiates the integration of this information by examining each position in the vector $\mathit{eff}_{change}{}^{(a)}$. If the value at the position l is zero, this indicates, that the action a does not affect x_l and the algorithm adds the clause $a \rightarrow x'_l = x_l$ to the satisfiability problem. Otherwise, a new constant $\triangle x_l$ is generated and the clause $a \rightarrow x'_l = x_l + \Delta_{x_l}$ with $\mathit{eff}_{low}{}^{(a)}_l \leq \Delta_{x_l} \leq \mathit{eff}_{up}{}^{(a)}_l$ is added.

After adding another step to the plan, the LNP uses the SMT solver to check, whether there is a valid assignment for the satisfiability problem on the abstract level (cf. Alg. 1, line 6). When the SMT solver identifies a valid assignment, it returns the satisfiability model s_model. For those actions, whose transition functions are modelled using mathematical expressions, these expressions already contain the dependencies of the control parameters. Thus, the control parameters for these actions are already determined by the SMT solver and no further proceeding is needed. To determine the remaining control parameters of the neural actions, the encapsulated, pre-trained NNs are used.

The integration of the lifted information of these actions restricts the solution space and permits the algorithm to explicitly determine the intermediate states between the different actions. For every NN $M_a \in M$, describing the effect of the neural action a, the planning algorithm captures both the state to which the action is applied, s, and the state the application leads to, s'. Subsequently, the trained NNs are utilized to determine the sets of control parameters using the *concretization*, described in section 4.2 (cf. Alg. 1, line 8). The *concretization* takes the trained NN M_a and the two states s and s' as input and returns either *unconcretizable* or the set of control parameters Θ and the actual state $\hat{s'}$ reached by applying the neural action with set Θ to the state s.

Subsequently, the planning algorithm checks the results of the concretization for validity (cf. Alg. 1, line 10). When the planning algorithm initially transforms the planning problem into a satisfiability problem (cf. Alg. 1, line 1), it also included the limits of the control parameters lb and ub. Within this step, for each neural action a and every control parameter $\theta \in \Theta$ returned by the *concretization*, it is checked whether it is a legal assignment $LegalP_a(\Theta)$, where $lb \leq \theta \wedge \theta \leq ub$. Additionally, for each neural action it has to be checked, whether the actual resulting state $\hat{s'}$ is violating any preconditions of the subsequent actions and the goal conditions can still be reached by the original plan, considering the deviation between s' and $\hat{s'}$. If all concretizations are valid, a valid plan to the N3PCP problem is found and the planning algorithm returns the detected plan Γ (cf. Alg. 1, line 17).

If either, the *concretization* returns *unconcretizable* or the results of the concretization are detected to be not valid (cf. Alg. 1, line 10), a new set of clauses is generated and added to the satisfiability problem, blocking the satisfiability model s_model previously found by the SMT solver (cf. Alg. 1, line 12). Subsequently, the planning algorithm searches again for a valid plan on the abstract level, using the SMT solver (cf. Alg. 1, line 6). If there is no other valid assignment for the satisfiability problem available with the considered number of actions, the algorithm takes one step backwards and extends the number of actions leading from I to G. This procedure is repeated until a valid plan with a valid set of control parameters for each action is found, or the algorithm has not found a solution within a maximum number of actions (cf. Alg. 1, line 3).

4.2 Concretization

Let s and s' be the input and output values for a neural action a found by the symbolic search. The *concretization* checks if $\exists\Theta.T_a(X,\Theta,X')[X/s,X'/s']$, i.e., if there exists a valuation to the control parameters such that the network computes the output s' given the input s.

Notice that, in principle, for full-access white box models, this step could be done with an SMT solver, provided the model is known and can be represented as an SMT formula. In practice, for full-access white box and function-access black box models, we adopt an optimization approach inspired by the works of [27] and [21]. For no-access black box models that are only available as pure input-output interfaces, a search algorithm, e.g., beam search, is required. The focus of this paper is on full-access and function-access models of neural networks; thus, only the former will be described in detail in this section.

For full-access and function-access models, we employ backpropagation combined with stochastic gradient descent to optimize the missing input values, specifically Θ in this context, rather than the weights of M_a. It is a well-founded assumption that certain control parameters do not affect the outcome of neural actions; these are systematically excluded from the gradient computation. Concurrently, we incorporate $\Theta^{(a)} \subseteq \Theta$, which delineates the subset of control parameters that do influence the neural actions, into the gradient tree.

We set the initial values of $\Theta^{(a)}$ to zero and infer $\hat{s'}$. The discrepancy between $\hat{s'}$ and the actual s' is propagated backward through the fixed weights of M_a, allowing for the iterative refinement of $\Theta^{(a)}$ towards convergence.

To mitigate the risk of converging to local optima, we employ an ensemble strategy alongside dropout in the hidden layers during inference. We additionally reduce the risk of generating erroneous plans, e.g., due to a local optimum, by establishing a threshold for the loss during the optimization phase. In detail, we focus on the distance between s' and $\hat{s'}$. Should the loss exceed the predefined threshold, the *concretization* returns *unconcretizable* and the planning algorithm blocks the current assignment. This allows us to omit network-input-output combinations that do not fit the observed action functionality. We determine the threshold value as validation loss from the training phase of M and a factor τ. If the loss not exceed the predefined threshold, the set of control parameters $\Theta^{(a)}$ as well as the actual state $\hat{s'}$ resulting from the application of a with $\Theta^{(a)}$ to s is returned.

4.3 Lifting the Neural Network

To obtain lifted information about the neural actions, that are integrated into the abstract level of the satisfiability problem, we introduce the procedure *observer* (cf. Algorithm 2). This procedure is applied during the training phase of the NNs. In each training iteration, the *observer* assesses the transformation, which an action a performs on a state s within the given data sample.

Initially, the procedure creates three n-dimensional zero-filled vectors $\mathit{eff}^{(a)}_{change}$, $\mathit{eff}^{(a)}_{low}$ and $\mathit{eff}^{(a)}_{up}$, where n is the number of variables in X. For each training step, it examines the input state vector s, and the resulting output state vector s' for each feature separately. If a value changes in the transition from s to s', the effect of the action a is updated in the corresponding $\mathit{eff}^{(a)}_{change}$ vector (cf. Alg. 2, line 5). Additionally, the procedure assesses the magnitude of the action's effect on the feature. If the observed difference is smaller than the previously recorded minimum value for a feature that was

Algorithm 2 The procedure *observer* extracts lifted information from a data sample.

```
Require: states before ($s$) and after ($s'$) the action
Ensure: $\mathit{eff}^{(a)}_{change}, \mathit{eff}^{(a)}_{low}, \mathit{eff}^{(a)}_{up}$
 1: procedure observer:
 2: $\mathit{eff}^{(a)}_{change}, \mathit{eff}^{(a)}_{low}, \mathit{eff}^{(a)}_{up} \leftarrow create(s)$
 3: for $l = 0, l < n$ do
 4:     if $s[l] \neq s'[l]$ then
 5:         $\mathit{eff}^{(a)}_{change}[i] = 1$
 6:         $\mathit{eff}^{(a)}_{low}[l] = min(\mathit{eff}^{(a)}_{low}[l], (s'[l] - s[l]))$
 7:         $\mathit{eff}^{(a)}_{up}[l] = max(\mathit{eff}^{(a)}_{up}[l], (s'[l] - s[l]))$
 8:     end if
 9: end for
10: return $\mathit{eff}_{change}{}^{(a)}, \mathit{eff}_{low}{}^{(a)}, \mathit{eff}_{up}{}^{(a)}$
```

already stored within $\mathit{eff}_{low}{}^{(a)}$, the procedure updates this minimum value accordingly (cf. Alg. 2, line 6). Maxima are equivalently updated in the vector $\mathit{eff}_{up}{}^{(a)}$ (cf. Alg. 2, line 7).

At the end of the training phase, the procedure returns the three vectors (cf. Alg. 2, line 10) containing lifted information on the transition, the trained NN M_a is describing. The first vector $\mathit{eff}_{change}{}^{(a)}$ indicates whether a feature is modified, represented by ones and zeros. The remaining two vectors $\mathit{eff}_{low}{}^{(a)}$ and $\mathit{eff}_{up}{}^{(a)}$ specify, the relative lower and upper bounds of the transition caused by the action a in the given training data.

5 Related Work

The concept of planning with control parameters was introduced by Savaş et al. [24], enhancing the applicability of planning approaches to real-world scenarios. In these domains, a planner tasked with addressing a planning problem involving control parameters must determine the values of these parameters to ensure that actions yield the intended outcomes [24]. To address this challenge, Savaş et al. [24] developed the POPCORN planner, which augments the capabilities of the partial-order planning framework of the POPF planner [6] by incorporating linear constraints and employing linear programming to maintain state consistency in order to handle control parameters. However, the application of POPCORN is restricted to discrete numeric modifications of state variables [24].

The SMTPlan planner, presented by Cashmore et al. [3], encodes a hybrid planning problem into a satisfiability problem in SMT which is then iteratively solved using the SMT solver Z3 ([20]). Their empirical evaluations demonstrated the efficacy of an SMT-based approach in managing planning problems with control parameters, indicating its suitability for such applications [3].

Sapena et al. [23] introduced the NextFLAP planner. The planner is based on the forward partial-order planner TFLAP [22] that is combined with a numeric constraint optimizer, the SMT solver z3, which is used to check for a valid assignment of values to the numeric variables and to select the optimal one [23].

All approaches discussed previously are capable of addressing numerical planning domains that involve control parameters. However, none of these approaches solves NN-enriched numerical planning problems with control parameters.

Introducing the N3PCP problem for the field of production planning, Heesch et al. [14] also proposed their planning algorithm RAINER, able to solve this kind of problems in cyber-physical production systems based on a new feature-vector-based state-space representation. RAINER is based on an eager and exact encoding of the

NNs, where the NNs are logically represented in exact SMT terms representing the mathematical relationships inherent in the NNs. To be able to directly integrate the mathematical relationships within the NN into the satisfiability problem, the model's architecture and the weights have to be known beforehand, excluding the possibility of integrating models that are not fully accessible as executable functions. Besides this, the main downside of this approach is the chosen integration method of the NNs into the satisfiability problem. Given the inherent mathematical relationships among the neurons, weights, and biases within NNs, direct integration into a satisfiability problem in SMT significantly increases the runtime of the SMT solver [14].

6 Evaluation

Benchmarks We evaluate our approach on planning problems from four different planning domains.

The **drone** domain, featured in the numerical track of the International Planning Competition (IPC) 2023 [25], initially lacked control parameters. To integrate such parameters, we modified the fixed numerical effect associated with the action *recharge*, substituting it with a mathematical expression that includes the control parameter 'energy'.

Introduced during the IPC in 2002, the **zenotravel** domain also did not include control parameters, initially.[1] To incorporate these, we modified the fixed numerical effect of the action *refuel* to a mathematical expression that encompasses the control parameter 'fuel'.

The **cashpoint** domain was introduced by Savaş et al. [24] to be a numerical domain with control parameters. Within this domain, the effect of the action *withdraw_cash* is described using the control parameter 'cash'. To pose a N3PCP problem, we omitted the temporal aspect from the domain.

The **FliPSi** domain, introduced by Krantz et al. [17], simulates a modular metal processing facility. Within this domain, the facility's modules are conceptualized as actions, while the manufactured products are depicted as states.

For each action with control parameters, we trained two types of NNs. The first type uses piecewise linear ReLU activation functions, the second type uses nonlinear sigmoid activations. All NNs have four hidden layers. Their input and output dimensions depend on the feature numbers from each domain.

Experimental Setup We implemented our LNP approach in Python 3.12.[2] As SMT backend, we used the Z3 solver [20]. All experiments were carried out on an Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz with the default settings of the Z3 solver. Each variable was represented numerically, with propositional variables expressed as ones (true) and zeros (false). We neglected the accuracy of the trained NNs, representing the effects of the neural actions, as the focus of this paper is not on the training but on the combination with a logical paradigm. Because not all network architectures that are used within the evaluation are equally suited to map the effects of the neural actions, we relaxed the threshold regarding the concretization as well as the limits of the control parameters in order to obtain comparable results. We set the runtime limit for simulations to 648,000 seconds.

[1] https://ipc02.icaps-conference.org/

[2] The code for replicating the experiments can be found under https://github.com/RHeesch/LNP

6.1 Results

Our evaluation of LNP is fourfold: *(1)* We compare our LNP's performance with the only other planner for NN-enriched planning problems [14], *(2)* we conduct an ablation study on the influence of the *observer* procedure on our LNP, *(3)* we compare the runtime with different types of model accesses and *(4)* we analyze the scalability of our LNP with complete *observer* in the dimensions of planning problem complexity, neural network architecture complexity, and number of neural actions.

6.1.1 Lazy Planning vs. Eager Planning

To evaluate our LNP algorithm's performance, we benchmarked it against the only other planner for NN-enriched planning problems, RAINER [14], resembling an eager planner. For this purpose, we selected for each domain three planning problems, each with an increasing plan length required to reach the optimal solution. We solved each planning problem four times, using both types of NNs – with ReLU and sigmoid activation functions – and problem variants where the NNs are available as full-access and function-access models. We defined a plan as optimal, if there is no plan with fewer actions and the set of control parameters is valid and not violating the thresholds. The results for all problems, domains and configurations are available online, along with the code for reproducing the experiments.

Table 1. The table shows the computation time in seconds of the eager and the lazy approach for finding an optimal solution in seconds. (TO) denotes that the algorithm was not able to find a plan within seven days (648,000 seconds). The NNs have **ReLU** activation functions and are full-access models.

domain	plan length	RAINER	$LNP_{full-access}$
FliPSi	2 actions	TO	5.60
	3 actions	TO	5.37
	5 actions	TO	5.55
Drone	2 actions	TO	5.46
	3 actions	TO	5.65
	15 actions	TO	15.69
Zeno	2 actions	TO	5.78
	3 actions	TO	6.06
	9 actions	TO	8.03
Cashpoint	2 actions	TO	5.73
	3 actions	TO	5.88
	6 actions	TO	6.49

Table 2. The table shows the computation time in seconds of the eager and the lazy approach for finding an optimal solution in seconds. (TO) denotes that the algorithm was not able to find a plan within seven days (648,000 seconds). The NNs have **sigmoid** activation functions and are full-access models.

domain	plan length	RAINER	$LNP_{full-access}$
FliPSi	2 actions	TO	5.44
	3 actions	TO	5.46
	5 actions	TO	5.51
Drone	2 actions	MO	5.59
	3 actions	MO	5.57
	15 actions	MO	15.82
Zeno	2 actions	TO	5.73
	3 actions	TO	5.95
	9 actions	TO	8.03
Cashpoint	2 actions	TO	5.75
	3 actions	TO	5.90
	6 actions	TO	6.72

The eager approach, RAINER, can only handle full-access NNs with known characteristics, while our proposed LNP can also handle NNs with function-access (cf. subsection 6.1.3). Table 1 summarizes the results for the domains with ReLU activation functions and fully known, accessible models. The results for the domains with sigmoid activation functions and fully known, and accessible models are summed up in Table 2.

Even with known network characteristics, RAINER can never find a solution within the runtime limit, or can only reach a depth of two to three steps. Within the *drone* domain, the SMT solver stops due to a memory overflow (MO) when sigmoid activation functions are used in the NNs. The proposed LNP approach, in contrast, successfully solves all problems across all domains in short runtimes, handling both ReLU and sigmoid activation functions.

6.1.2 Observer Ablation Study

We conducted an ablation study on the observer procedure based on the previously introduced planning problems and domains. Table 3 provides an overview of the results for models with ReLU activation functions and backpropagation capabilities. The results for all problems and domains, along with the code for reproducing the experiments, are available online.

We considered three scenarios in which we reduce the integration of information provided by the observer from *(i)* $LNP_{complete}$ integrating all information provided by the observer, i.e., $\mathit{eff}^{(a)}_{change}$, $\mathit{eff}^{(a)}_{low}$, $\mathit{eff}^{(a)}_{up}$, *(ii)* $LNP_{partial}$ only integrating the information whether a feature is influenced or not, i.e., $\mathit{eff}^{(a)}_{change}$, and *(iii)* LNP_{no} integrating no information provided by the observer.

Table 3. The table summarizes the computation time of the ablation study on the observer procedure, in seconds. (TO) denotes that the algorithm was not able to find a plan within seven days (648,000 seconds).

domain	plan length	$LNP_{complete}$	$LNP_{partial}$	LNP_{no}
FliPSi	2 actions	5.60	5.12	15.55
	3 actions	5.37	5.67	20.46
	5 actions	5.55	5.56	4753.75
Drone	2 actions	5.46	5.50	5.59
	3 actions	5.65	5.51	25.43
	15 actions	15.69	16.06	TO
Zeno	2 actions	5.78	5.66	6.08
	3 actions	6.06	5.77	26.29
	9 actions	8.03	8.62	TO
Cashpoint	2 actions	5.73	5.84	5.91
	3 actions	5.88	5.80	26.50
	6 actions	6.49	6.43	13701.58

The runtime of the LNP, when fully or partially integrating the observer's information, is similar; on average, the runtime with partial integration is only milliseconds slower. When no information from the observer is integrated, the runtime remains similar for the smallest planning problems, which involve only two steps. However, if the optimal plan consists of more than two steps, the runtime significantly increases, leading to runtimes above the timeout for the largest planning problems.

6.1.3 Model Accessibilty

We evaluated the performance of our LNP approach under three distinct types of model access: (i) the model is fully known and accessible (*full-access*), (ii) the model is unknown, but offers dedicated functionalities such as backpropagation (*function-access*), and (iii) the model is unknown and only available as a pure input-output interface (*no-access*).

For *full-access* or *function-access* models, we use backpropagation during the concretization process to determine the control parameters. However, for models available only as executables with a pure input-output interface, an alternative algorithm is needed.

To evaluate the runtime for these input-output interfaces, we employed the same functions used to generate the training data for the NNs. These functions take a concatenation of a state vector s and the values of control parameters Θ as input and return a new state vector s'. To search for the required control parameters, we implemented a beam search algorithm with a perturbation of 0.01 and a beam width of 10. The algorithm was configured to search for the control parameters until the distance between s' and $\hat{s'}$ is smaller than 0.1 within the valid limits for the control parameters. The runtimes for different problems and configurations with complete observer information are shown in Table 4.

Table 4. The table displays the runtime behavior of the *LNP* with full-access, function-access and no-access models, in seconds. (TO) denotes that the algorithm was not able to find a plan within seven days (648,000 seconds).

domain	plan length	LNP_{full}	$LNP_{function}$	LNP_{no}
FliPSi	2 actions	5.60	5.60	TO
	3 actions	5.37	5.37	TO
	5 actions	5.55	5.55	TO
Drone	2 actions	5.46	5.46	3.05
	3 actions	5.65	5.65	2.31
	15 actions	15.69	15.69	3.27
Zeno	2 actions	5.78	5.78	5.58
	3 actions	6.06	6.06	3.87
	9 actions	8.03	8.03	7.94
Cashpoint	2 actions	5.73	5.73	0.38
	3 actions	5.88	5.88	0.62
	6 actions	6.49	6.49	1.97

Since the concretization for both the full and the function access model is based on backpropagation, the runtimes are identical. The evaluation demonstrates that our proposed LNP approach can also handle executables that are only available as input-output interfaces. Depending on the range of valid values for the control parameters, it can perform faster than backpropagation with *full-access* or *function-access* models. However, the runtime significantly increases when the range of valid values for the control parameters expands, e.g., in the FliPSi domain, which leads to time-outs.

6.1.4 Scalability

We evaluated the performance of our LNP approach under increasing planning problem complexity, neural network complexity, and number of neural actions, on the *drone* domain.

Therefore, we systematically increased the number of hidden layers in the NN representing the *charge* action from two to twelve. Additionally, we increased the number of neurons per layer from 14 to 100. We tested NNs with four different activation functions: ReLU, Sigmoid, Hyperbolic Tangent, and Leaky ReLU. Furthermore, we replaced the fixed effects of the actions *increase-x*, *increase-y*, and *increase-z* with control parameters that could take both positive and negative values, thereby rendering the opposing actions *decrease-x*, *decrease-y*, and *decrease-z* redundant. We also replaced the fixed effects of the actions *visit_location_1* and *visit_location_2* with control parameters.

The results show that modifications of the NN characteristics only affect the runtime of the concretization process (cf. Figure 2). The

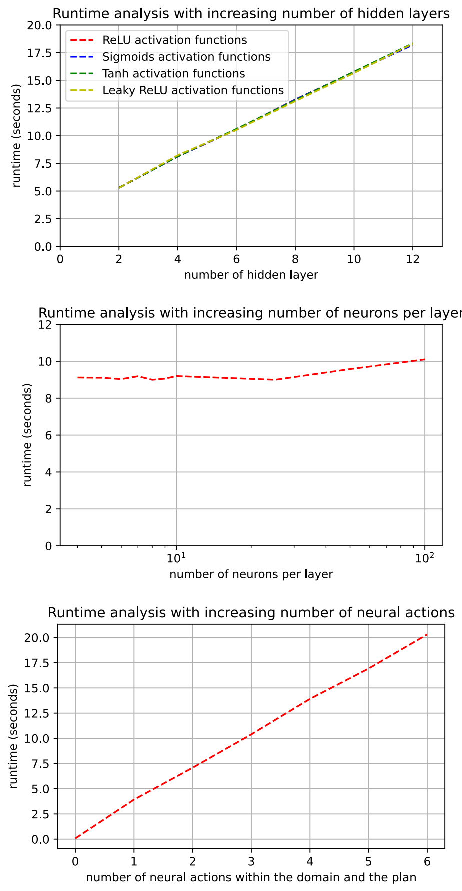

Figure 2. Runtime behavior of the LNP with increasing numbers of hidden layers (*top*), neurons per layer (*middle*) and neural actions per domain and plan (*bottom*).

runtime increases linearly with the number of hidden layers. Using different activation functions does not have an effect on the runtime of the algorithm. Increasing the number of neurons per layer results in a slightly higher runtime. Doubling the number of neurons per layer leads to a runtime increase of approximately ten percent. The runtime also increases linearly with the number of neural actions in the domain and within the plan.

6.2 *Discussion*

Our evaluation shows that our proposed lazy approach is significantly faster than the comparable eager algorithm, RAINER. RAINER always reaches the timeout, supporting previous findings that SMT solvers struggle to handle nonlinear activation functions effectively [13, 12]. As the LNP can manage NNs as not fully-accessible executable functions without requiring knowledge of their network characteristics, the type of activation function also has a negligible effect. In general, the LNP can handle all NN architectures that allow optimization of their input.

Within the regarded problems, the LNP with fully and partial integrated information provided by the observer performs similar. Without the integration of any observer information, the LNP is much less efficient in finding solutions, which leads to significantly longer runtimes and even timeouts.

From a theoretical perspective, the observer's information, specifically whether and within what range a feature can be modified by an action, may reduce the search space in a way that excludes the optimal solution, i.e., the shortest plan. However, if we integrate only partial information from the observer, such as which features can be modified by an action, this limitation of completeness can be reduced, provided the observer is trained on the same data as the NN. In cases where the data do not reflect a change in a feature that could be influenced by an action, the NN will also struggle to represent this feature. As a result, the planning approach with only partial information from the observer maintains completeness relative to the model of the planning domain.

On the other hand, lacking information about the range may lead to avoidable and time-consuming iterations between the SMT solver at the top level and the concretization. When the constraints of the control parameters restrict the transformation an action performs on a state, and the action must therefore be performed multiple times, this will result in at least one redundant and unsuccessful concretization. Starting with only the partial information provided by the observer and dynamically extending this information by narrowing the range when a concretization fails appears to be a promising solution to this issue.

To manage black box models, i.e., executables that are accessible only as input-output interfaces, an alternative approach to backpropagation is required for searching the control parameters within the concretization. But, by selecting an appropriate search algorithm, the LNP can manage these black-boxes, providing the opportunity to hide complex mathematical formulas that could not effectively be handled directly by an SMT solver inside a function. However, employing an uninformed search algorithm may considerably increase the runtime of the concretization, particularly if there is a wide range of valid assignments for the control parameters.

The scalability analysis indicates that our proposed LNP can effectively handle even large network architectures. Nevertheless, planning problems involving neural actions represented by wider NNs could be solved faster than those with neural actions represented by deeper NNs.

7 Conclusion

In this paper, we tackled the problem of numerical planning with control parameters for domains where actions can also be described with NNs. We proposed a planning approach based on the following insights: First, the search is hierarchical, and delays the analysis of the NN's until an abstract plan is found; Second, the discovery of spurious counterexamples is limited by the introduction of a form of underapproximating static learning; Third, the refinement is completely numerical, based on numerical optimization, and does not depend on complex symbolic reasoning.

With respect to previous approaches to N3PCP [14], our approach supports a more expressive model, and trades off completeness for generality and efficiency. The experiments show that our approach can solve a comprehensive set of numerical planning benchmarks, including networks with nonlinear activation functions, which cannot be represented by state-of-the-art solvers.

In the future, we plan to extend this work along the following directions. First, we want to integrate the reasoning about NN's directly in the SMT solver in a tighter way, following an online integration paradigm [1], and to exploit the inherent incrementality of the approach. We will investigate the integration of some symbolic reasoning to limit incompleteness, and finally, we will explore the case of temporally-extended N3PCP.

Acknowledgements

This research as part of the projects EKI and LaiLa is funded by dtec.bw – Digitalization and Technology Research Center of the Bundeswehr which we gratefully acknowledge. dtec.bw is funded by the European Union – NextGenerationEU. This work has also benefitted from Dagstuhl Seminar 24031 "Fusing Causality, Reasoning, and Learning for Fault Management and Diagnosis."

References

[1] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003.

[3] M. Cashmore, D. Magazzeni, and P. Zehtabi. Planning for hybrid systems via satisfiability modulo theories. *Journal of Artificial Intelligence Research*, 67:235–283, 2020.

[4] A. Cimatti, A. Griggio, A. Irfan, M. Roveri, and R. Sebastiani. Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. *ACM Trans. Comput. Log.*, 19 (3):19:1–19:52, 2018.

[5] A. Cimatti, A. Griggio, S. Mover, M. Roveri, and S. Tonetta. Verification modulo theories. *Formal Methods Syst. Des.*, 60(3):452–481, 2022.

[6] A. Coles, A. Coles, M. Fox, and D. Long. Forward-chaining partial-order planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 20, pages 42–49, 2010.

[7] H. Duong, L. Li, T. Nguyen, and M. Dwyer. A dpll (t) framework for verifying deep neural networks. *arXiv preprint arXiv:2307.10266*, 2023.

[8] J. Ehrhardt, R. Heesch, and O. Niggemann. Learning process steps as dynamical systems for a sub-symbolic approach of process planning in cyber-physical production systems. In *Artificial Intelligence. ECAI 2023 International Workshops*, pages 332–345, Cham, 2024. Springer Nature Switzerland.

[9] S. Gao, J. Avigad, and E. M. Clarke. δ-complete decision procedures for satisfiability over the reals. In *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 2012.

[10] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] D. Guidotti, F. Leofante, L. Pulina, and A. Tacchella. Verification of neural networks: Enhancing scalability through pruning. In *ECAI*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2505–2512. IOS Press, 2020.

[12] D. Guidotti, L. Pandolfo, and L. Pulina. Verifying neural networks with non-linear SMT solvers: a short status report. In *2023 IEEE 35th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 423–428. IEEE, 2023.

[13] D. Guidotti, L. Pandolfo, and L. Pulina. Leveraging satisfiability modulo theory solvers for verification of neural networks in predictive maintenance applications. *Information*, 14(7), 2023. ISSN 2078-2489. doi: 10.3390/info14070397. URL https://www.mdpi.com/2078-2489/14/7/397.

[14] R. Heesch, J. Ehrhardt, and O. Niggemann. Integrating machine learning into an SMT-based planning approach for production planning inc yber-physical production systems. In *Artificial Intelligence. ECAI 2023 International Workshops*, pages 318–331, Cham, 2024. Springer Nature Switzerland.

[15] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, Jan. 1989. ISSN 0893-6080. doi: 10.1016/0893-6080(89)90020-8. URL http://dx.doi.org/10.1016/0893-6080(89)90020-8.

[16] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *CAV (1)*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017.

[17] M. Krantz, N. Widulle, A. Nordhausen, A. Liebert, J. Ehrhardt, S. Eilermann, and O. Niggemann. Flipsi: Generating data for the training of machine learning algorithms for cpps. In *Annual Conference of the PHM Society*, volume 14, 2022.

[18] J. Lederer. Activation functions in artificial neural networks: A systematic overview, 2021. URL https://arxiv.org/abs/2101.09957.

[19] Z. Manna and A. Pnueli. The modal logic of programs. In *ICALP*, volume 71 of *Lecture Notes in Computer Science*, pages 385–409. Springer, 1979.

[20] L. d. Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[21] J.-P. Roche, O. Niggemann, and J. Friebe. Using autoencoders and autodiff to reconstruct missing variables in a set of time series, 2023. URL https://arxiv.org/abs/2308.10496.

[22] O. Sapena, E. Marzal, and E. Onaindia. TFLAP: a temporal forward partial-order planner. *URL https://ipc2018-temporal. bitbucket. io/planner-abstracts/team2. pdf*, 2018.

[23] O. Sapena, E. Onaindia, and E. Marzal. A hybrid approach for expressive numeric and temporal planning with control parameters. *Expert Systems with Applications*, 242:122820, 2024. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2023.122820. URL https://www.sciencedirect.com/science/article/pii/S0957417423033225.

[24] E. Savaş, M. Fox, D. Long, and D. Magazzeni. Planning using actions with control parameters. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, pages 1185–1193, 2016.

[25] A. Taitler, R. Alford, J. Espasa, G. Behnke, D. Fišer, M. Gimelfarb, F. Pommerening, S. Sanner, E. Scala, D. Schreiber, et al. The 2023 international planning competition, 2024.

[26] D. van Dalen. *Logic and structure (3. ed.)*. Universitext. Springer, 1994.

[27] G. Wu, B. Say, and S. Sanner. Scalable planning with tensorflow for hybrid nonlinear domains. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/98b17f068d5d9b7668e19fb8ae470841-Paper.pdf.