



Theoretical and Experimental Results for Planning with Learned Binarized Neural Network Transition Models

Buser Say¹(✉) , Jo Devriendt^{2,3} , Jakob Nordström^{3,2} ,
and Peter J. Stuckey¹ 

¹ Monash University, Melbourne, Australia
{buser.say,peter.stuckey}@monash.edu

² Lund University, Lund, Sweden
jo.devriendt@cs.lth.se

³ University of Copenhagen, Copenhagen, Denmark
jn@di.ku.dk

Abstract. We study planning problems where the transition function is described by a learned binarized neural network (BNN). Theoretically, we show that feasible planning with a learned BNN model is *NP*-complete, and present two new constraint programming models of this task as a mathematical optimization problem. Experimentally, we run solvers for constraint programming, weighted partial maximum satisfiability, 0–1 integer programming, and pseudo-Boolean optimization, and observe that the pseudo-Boolean solver outperforms previous approaches by one to two orders of magnitude. We also investigate symmetry handling for planning problems with learned BNNs over long horizons. While the results here are less clear-cut, we see that exploiting symmetries can sometimes reduce the running time of the pseudo-Boolean solver by up to three orders of magnitude.

Keywords: Automated planning · Binarized neural networks · Mathematical optimization · Pseudo-Boolean optimization · Cutting planes reasoning · Symmetry

1 Introduction

Automated planning is the reasoning side of acting in Artificial Intelligence [23]. Planning automates the selection and ordering of actions to reach desired states of the world. An automated planning problem represents the real-world dynamics using a model of the world, which can either be manually encoded [7, 13, 14, 20, 24], or learned from data [1, 2, 12, 29]. In this paper, we focus on the latter.

Automated planning with deep neural network (DNN) learned state transition models is a two stage data-driven framework for learning and solving planning problems with unknown state transition models [28]. The first stage of the framework learns the unknown state transition model from data as a DNN.

The second stage of the framework plans optimally with respect to the learned DNN model by solving an equivalent mathematical optimization problem (e.g., a mixed-integer programming (MIP) model [28], a 0–1 integer programming (IP) model [25, 26], or a weighted partial maximum satisfiability (WP-MaxSAT) model [25, 26]). In this paper, we focus on the theoretical, mathematical modelling and the experimental aspects of the second stage of the data-driven framework where the learned DNN is a binarized neural network (BNN) [16].

We study the complexity of feasible automated planning with learned BNN transition models under the common assumption that the learned BNN is fully connected, and show that this problem is *NP*-complete. In terms of mathematical modelling, we propose two new constraint programming (CP) models that are motivated by the work on learning BNNs with CP [33]. We then conduct two sets of experiments for the previous and our new mathematical optimization models for the learned automated problem. In our first set of experiments, we focus on solving the existing learned automated problem instances using off-the-shelf solvers for WP-MaxSAT [6], MIP [17], pseudo-Boolean optimization (PBO) [10] and CP [17]. Our results show that the PBO solver RoundingSat [10] outperforms the existing baselines by one to two orders of magnitude. In our second set of experiments, we focus on the challenging task of solving learned automated planning problems over long planning horizons. Here, we study and test the effect of specialized symmetric reasoning over different time steps of the learned planning problem. Our preliminary results demonstrate that exploiting this symmetry can significantly reduce the overall runtime of the underlying solver (i.e., RoundingSat) by upto three orders of magnitude. Overall, with this paper we make both theoretical and practical contributions to the field of data-driven automated planning with learned BNN transition models.

In the next section we formally define the planning problem using binarized neural network (BNN) transitions functions. In Sect. 3 we define a 0–1 integer programming (IP) model that will solve the planning problem given a learned BNN. In Sect. 4 we show that the feasibility problem is *NP*-complete. In Sect. 5 we give two constraint programming models for the solving the planning problem. In Sect. 6 we discuss a particular symmetry property of the model, and discuss how to take advantage of it. In Sect. 7 we give experimental results. Finally, in Sect. 8 we conclude and discuss future work.

2 Planning with Learned BNN Transition Models

We begin by presenting the definition of the learned automated planning problem and the BNN architecture used for learning the transition model from data.

2.1 Problem Definition

A *fixed-horizon learned deterministic automated planning problem* [25, 28] is a tuple $\tilde{\Pi} = \langle S, A, C, \tilde{T}, V, G, R, H \rangle$, where $S = \{s_1, \dots, s_n\}$ and $A = \{a_1, \dots, a_m\}$ are sets of state and action variables for positive integers n, m with domains

D_{s_1}, \dots, D_{s_n} and D_{a_1}, \dots, D_{a_m} respectively, $C : D_{s_1} \times \dots \times D_{s_n} \times D_{a_1} \times \dots \times D_{a_m} \rightarrow \{true, false\}$ is the global function, $\tilde{T} : D_{s_1} \times \dots \times D_{s_n} \times D_{a_1} \times \dots \times D_{a_m} \rightarrow D_{s_1} \times \dots \times D_{s_n}$ denotes the learned state transition function, and $R : D_{s_1} \times \dots \times D_{s_n} \times D_{a_1} \times \dots \times D_{a_m} \rightarrow \mathbb{R}$ is the reward function. Further, V is a tuple of constants $\langle V_1, \dots, V_n \rangle \in D_{s_1} \times \dots \times D_{s_n}$ that denotes the initial values of all state variables, $G : D_{s_1} \times \dots \times D_{s_n} \rightarrow \{true, false\}$ is the goal state function, and $H \in \mathbb{Z}^+$ is the planning horizon.

A *solution* to (i.e., a *plan* for) $\tilde{\Pi}$ is a tuple of values $\bar{A}^t = \langle \bar{a}_1^t, \dots, \bar{a}_m^t \rangle \in D_{a_1} \times \dots \times D_{a_m}$ for all action variables A over time steps $t \in \{1, \dots, H\}$ such that $\tilde{T}(\langle \bar{s}_1^t, \dots, \bar{s}_n^t, \bar{a}_1^t, \dots, \bar{a}_m^t \rangle) = \langle \bar{s}_1^{t+1}, \dots, \bar{s}_n^{t+1} \rangle$ and $C(\langle \bar{s}_1^t, \dots, \bar{s}_n^t, \bar{a}_1^t, \dots, \bar{a}_m^t \rangle) = true$ for time steps $t \in \{1, \dots, H\}$, $V_i = \bar{s}_i^1$ for all $s_i \in S$ and $G(\langle \bar{s}_1^{H+1}, \dots, \bar{s}_n^{H+1} \rangle) = true$. An *optimal* solution to $\tilde{\Pi}$ is a solution such that the total reward $\sum_{t=1}^H R(\langle \bar{s}_1^{t+1}, \dots, \bar{s}_n^{t+1}, \bar{a}_1^t, \dots, \bar{a}_m^t \rangle)$ is maximized.

It is assumed that the functions C, G, R and \tilde{T} are known, that C, G can be equivalently represented by a finite set of linear constraints, that R is a linear expression and that \tilde{T} is a learned binarized neural network [16]. Next, we give an example planning problem where these assumptions are demonstrated.

Example 1. A simple instance of a learned automated planning problem $\tilde{\Pi}$ is as follows.

- The set of state variables is defined as $S = \{s_1\}$ where $s_1 \in \{0, 1\}$.
- The set of action variables is defined as $A = \{a_1\}$ where $a_1 \in \{0, 1\}$.
- The global function C is defined as $C(\langle s_1, a_1 \rangle) = true$ when $s_1 + a_1 \leq 1$.
- The value of the state variable s_1 is $V_1 = 0$ at time step $t = 1$.
- The goal state function G is defined as $G(\langle s_1 \rangle) = true$ if and only if $s_1 = 1$.
- The reward function R is defined as $R(\langle s_1, a_1 \rangle) = -a_1$.
- The learned state transition function \tilde{T} is in the form of a BNN, which will be described below.
- A planning horizon of $H = 4$.

A plan (assuming the BNN described later in Fig. 1) is $\bar{a}_1^1 = 1, \bar{a}_1^2 = 1, \bar{a}_1^3 = 1, \bar{a}_1^4 = 0$ with corresponding states $\bar{s}_1^1 = 0, \bar{s}_1^2 = 0, \bar{s}_1^3 = 0, \bar{s}_1^4 = 0, \bar{s}_1^5 = 1$. The total reward for the plan is -3 . \square

2.2 Binarized Neural Networks

Binarized neural networks (BNNs) are neural networks with binary weights and activation functions [16]. As a result, BNNs can learn memory-efficient models by replacing most arithmetic operations with bit-wise operations. The fully-connected BNN that defines the learned state transition function \tilde{T} , given L layers with layer width W_l in layer $l \in \{1, \dots, L\}$, and a set of neurons $J(l) = \{u_{1,l}, \dots, u_{W_l,l}\}$, is stacked in the following order.

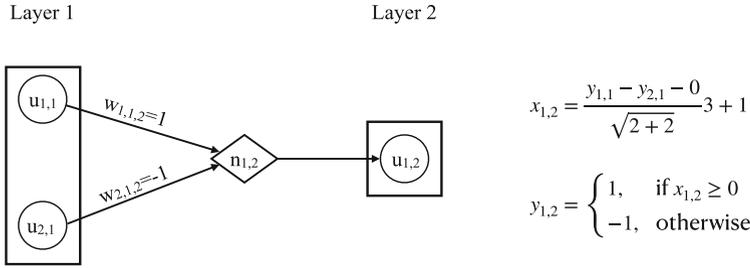


Fig. 1. Learned BNN with two layers $L = 2$ for the problem in Example 1. In this example learned BNN, the input layer $J(1)$ has neurons $u_{1,1}$ and $u_{2,1}$ representing s_1 and a_1 , respectively. The node $n_{2,1}$ represents batch normalization for neuron $u_{2,1}$. Given the parameter values $w_{1,1,l} = 1$, $w_{2,1,l} = -1$, $\mu_{1,2} = 0$, $\sigma_{1,2}^2 = 2$, $\epsilon_{1,2} = 2$, $\gamma_{1,2} = 3$ and $\beta_{j,l} = 1$, the input $x_{1,2}$ to neuron $u_{1,2}$ is calculated according to the formula specified in Sect. 2.2.

Input Layer. The first layer consists of neurons $u_{i,1} \in J(1)$ that represent the domain of the learned state transition function \tilde{T} . We will assume that the domains of action and state variables are binary, and let neurons $u_{1,1}, \dots, u_{n,1} \in J(1)$ represent the state variables S and neurons $u_{n+1,1}, \dots, u_{n+m,1} \in J(1)$ represent the action variables A . During the training of the BNN, binary values 0 and 1 of action and state variables are represented by -1 and 1 , respectively.

Batch Normalization Layers. For layers $l \in \{2, \dots, L\}$, Batch Normalization [18] transforms the weighted sum of outputs at layer $l - 1$ in $\Delta_{j,l} = \sum_{i \in J(l-1)} w_{i,j,l} y_{i,l-1}$ to inputs $x_{j,l}$ of neurons $u_{j,l} \in J(l)$ using the formula $x_{j,l} = \frac{\Delta_{j,l} - \mu_{j,l}}{\sqrt{\sigma_{j,l}^2 + \epsilon_{j,l}}} \gamma_{j,l} + \beta_{j,l}$, where $y_{i,l-1}$ denotes the output of neuron $u_{i,l-1} \in J(l - 1)$, and the parameters are the weight $w_{i,j,l}$, input mean $\mu_{j,l}$, input variance $\sigma_{j,l}^2$, numerical stability constant $\epsilon_{j,l}$, input scaling $\gamma_{j,l}$, and input bias $\beta_{j,l}$, all computed at training time.

Activation Layers. Given input $x_{j,l}$, the deterministic activation function $y_{j,l}$ computes the output of neuron $u_{j,l} \in J(l)$ at layer $l \in \{2, \dots, L\}$, which is 1 if $x_{j,l} \geq 0$ and -1 otherwise. The last activation layer consists of neurons $u_{i,L} \in J(L)$ that represent the codomain of the learned state transition function \tilde{T} . We assume neurons $u_{1,L}, \dots, u_{n,L} \in J(L)$ represent the state variables S .

The proposed BNN architecture is trained to learn the function \tilde{T} from data that consists of measurements on the domain and codomain of the *unknown* state transition function $T : D_{s_1} \times \dots \times D_{s_n} \times D_{a_1} \times \dots \times D_{a_m} \rightarrow D_{s_1} \times \dots \times D_{s_n}$. An example learned BNN for the problem of Example 1 is visualized in Fig. 1.

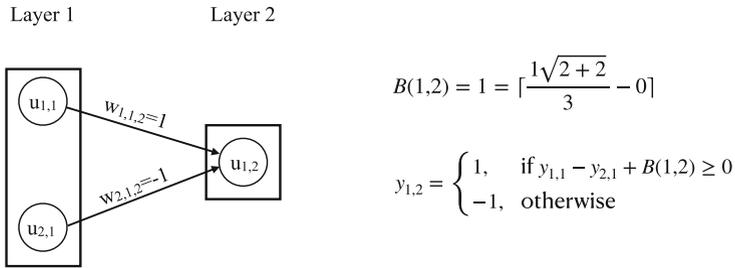


Fig. 2. The visualization of bias computation $B(1,2)$ for neuron $u_{1,2} \in J(2)$ in the example learned BNN presented in Fig. 1.

3 0–1 Integer Programming Model for the Learned Planning Problem

In this section, we present the 0–1 integer programming (IP) model from [25,26] previously used to solve learned automated planning problems. A 0–1 IP model can be solved optimally by a mixed-integer programming (MIP) solver (as was previously investigated [25,26]). Equivalently, this can be viewed as a pseudo-Boolean optimization (PBO) model to be solved using a PBO solver, since all the variables are 0–1 or equivalently Boolean.

Decision Variables. The 0–1 IP model uses the following decision variables:

- $X_{i,t}$ encodes whether action $a_i \in A$ is executed at time step $t \in \{1, \dots, H\}$ or not.
- $Y_{i,t}$ encodes whether we are in state $s_i \in S$ at time step $t \in \{1, \dots, H + 1\}$ or not.
- $Z_{i,l,t}$ encodes whether neuron $u_{i,l} \in J(l)$ in layer $l \in \{1, \dots, L\}$ is activated at time step $t \in \{1, \dots, H\}$ or not.

Parameters. The 0–1 IP model uses the following parameters:

- $\bar{w}_{i,j,l}$ is the value of the learned BNN weight between neurons $u_{i,l-1} \in J(l-1)$ and $u_{j,l} \in J(l)$ in layer $l \in \{2, \dots, L\}$.
- $B(j,l)$ is the bias of neuron $u_{j,l} \in J(l)$ in layer $l \in \{2, \dots, L\}$. Given the values of normalization parameters $\bar{\mu}_{j,l}$, $\bar{\sigma}_{j,l}^2$, $\bar{\epsilon}_{j,l}$, $\bar{\gamma}_{j,l}$ and $\bar{\beta}_{j,l}$, the bias is computed as $B(j,l) = \left\lfloor \frac{\bar{\beta}_{j,l}\sqrt{\bar{\sigma}_{j,l}^2 + \bar{\epsilon}_{j,l}}}{\bar{\gamma}_{j,l}} - \bar{\mu}_{j,l} \right\rfloor$. The visualization of the calculation of the bias $B(j,l)$ is presented in Fig. 2.

Constraints. The 0–1 IP model has the following constraints:

$$Y_{i,1} = V_i \quad \forall s_i \in S \quad (1)$$

$$G(\langle Y_{1,H+1}, \dots, Y_{n,H+1} \rangle) = true \quad (2)$$

$$C(\langle Y_{1,t}, \dots, Y_{n,t}, X_{1,t}, \dots, X_{m,t} \rangle) = true \quad \forall t \in \{1, \dots, H\} \quad (3)$$

$$Y_{i,t} = Z_{i,1,t} \quad \forall s_i \in S, t \in \{1, \dots, H\} \quad (4)$$

$$X_{i,t} = Z_{i+n,1,t} \quad \forall a_i \in A, t \in \{1, \dots, H\} \quad (5)$$

$$Y_{i,t+1} = Z_{i,L,t} \quad \forall s_i \in S, t \in \{1, \dots, H\} \quad (6)$$

$$(B(j,l) - |J(l-1)|)(1 - Z_{j,l,t}) \leq In(j,l,t) \quad \forall u_{j,l} \in J(l), l \in \{2, \dots, L\}, t \in \{1, \dots, H\} \quad (7)$$

$$(B(j,l) + |J(l-1)| + 1)Z_{j,l,t} - 1 \geq In(j,l,t) \quad \forall u_{j,l} \in J(l), l \in \{2, \dots, L\}, t \in \{1, \dots, H\} \quad (8)$$

where the input expression $In(j,l,t)$ for neuron $u_{j,l} \in J(l)$ in layer $l \in \{2, \dots, L\}$ at time step $t \in \{1, \dots, H\}$ is equal to $\sum_{u_{i,l-1} \in J(l-1)} \bar{w}_{i,j,l}(2 \cdot Z_{i,l-1,t} - 1) + B(j,l)$. In the above model, constraints (1) set the initial value of every state variable. Constraints (2)–(3) enforce the global constraints (i.e., constraints representing C) and the goal constraints (i.e., constraints representing G). Constraints (4)–(6) map the input and output layers of the learned BNN to the corresponding state and action variables. Finally, constraints (7)–(8) model the activation of each neuron in the learned BNN, where the decision variable $Z_{j,l,t} \in \{0, 1\}$ represents the output of neuron $u_{j,l} \in J(l)$ at time step $t \in \{1, \dots, H\}$ using the expression $(2 \cdot Z_{j,l,t} - 1) \in \{-1, 1\}$.

Objective Function. The 0–1 IP model has the objective function

$$\max \sum_{t=1}^H R(\langle Y_{1,t+1}, \dots, Y_{n,t+1}, X_{1,t}, \dots, X_{m,t} \rangle), \quad (9)$$

which maximizes the total reward accumulated over time steps $t \in \{1, \dots, H\}$.

Example 2. The 0–1 IP (or the equivalent PBO) model that is presented in this section can be solved to find an optimal plan to the instance that is described in Example 1. The optimal plan is $\bar{a}_1^t = 0$ for all time steps $t \in \{1, 2, 3, 4\}$, and the total reward for the optimal plan is 0. \square

4 Theoretical Results

In this section, we establish the *NP*-completeness of finding feasible solutions to learned planning problems.

Theorem 1. *Finding a feasible solution to a learned planning problem $\tilde{\Pi}$ with a fully-connected batch normalized learned BNN \tilde{T} is an NP-complete problem.*

Proof. We begin by showing that $\tilde{\Pi}$ is in *NP*. Given the values \bar{A}^t of action variables A for all time steps $t \in \{1, \dots, H\}$ and the initial values V_i of state

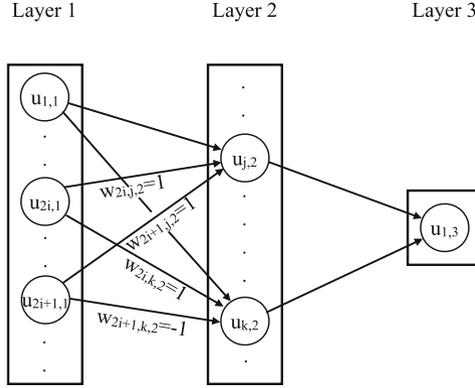


Fig. 3. Visualization of the *NP*-hardness proof by a reduction from a 3-CNF formula $\phi = \bigwedge_{j=1}^q c_j$ to the learned planning problem \tilde{I} . In the first layer, two neurons $u_{2i,1}, u_{2i+1,1} \in J(1)$ together represent the Boolean variable z_i from the formula ϕ . When variable z_i does not appear in clause c_k , the weights $\bar{w}_{2i,k,2}, \bar{w}_{2i+1,k,2}$ are set so the input to neuron $u_{k,2} \in J(2)$ is cancelled out (i.e., case (a) of step (7)). In the remaining cases, the weights $\bar{w}_{2i,j,2}, \bar{w}_{2i+1,j,2}$ are set to ensure the input to neuron $u_{j,2} \in J(2)$ is positive if and only if the respective literal that appears in clause c_j evaluates to true (e.g., case (c) of step (7) is visualized).

variables $s_i \in S$, the learned BNN \tilde{T} can predict the values $\bar{S}^t = \langle \bar{s}_1^t, \dots, \bar{s}_n^t \rangle \in D_{s_1} \times \dots \times D_{s_n}$ of all state variables S for all time steps $t \in \{2, \dots, H+1\}$ in linear time in the size of the BNN and the value of the planning horizon H .

We proceed by showing that \tilde{I} is in *NP*-hard by a reduction from 3-SAT. Let ϕ be a 3-CNF formula such that $\phi = \bigwedge_{j=1}^q c_j$ for some positive integer q . Further let z_1, \dots, z_r denote the (Boolean) variables that appear in the formula ϕ for some positive integer r . As visualized in Fig. 3, we define the learned planning problem \tilde{I} to represent any 3-CNF formula ϕ as follows:

1. Planning horizon $H = 1$.
2. State variable $S = \{s_1\}$.
3. Action variables $A = \{a_1, \dots, a_{2r}\}$.
4. The global function C is true if and only if $a_{2i-1} = a_{2i}$ for all $i \in \{1, \dots, r\}$.
5. Neurons $J(1) = \{u_{1,1}, \dots, u_{1+2r,1}\}$ in the first layer.
6. Neurons $J(2) = \{u_{1,2}, \dots, u_{q,2}\}$ in the second layer. Each neuron $u_{i,2} \in J(2)$ is normalized so that $B(i, 2) = 3$.
7. Set the learned weights between neurons $u_{2i,1}, u_{2i+1,1} \in J(1) \setminus u_{1,1}$ and $u_{j,2} \in J(2)$ according to the following rules. (a) If z_i does not appear in clause c_j , set $\bar{w}_{2i,j,2} = 1, \bar{w}_{2i+1,j,2} = -1$, (b) else if the negation of z_i appears in clause c_j (i.e., $\neg z_i$), set $\bar{w}_{2i,j,2} = \bar{w}_{2i+1,j,2} = -1$, (c) else, set $\bar{w}_{2i,j,2} = \bar{w}_{2i+1,j,2} = 1$.
8. Neuron $J(3) = \{u_{1,3}\}$ in the third layer. Neuron $u_{1,3}$ is normalized such that $B(1, 3) = -q$.
9. Set the learned weights $\bar{w}_{i,1,3} = 1$ between $u_{i,2} \in J(2)$ and $u_{1,3} \in J(3)$.
10. The goal state function G is defined as $G(\langle 1 \rangle) = \text{true}$ and $G(\langle 0 \rangle) = \text{false}$.

In the reduction presented above, step (1) sets the value of the planning horizon H to 1. Step (2) defines a single state variable s_1 to represent whether the formula ϕ is satisfied (i.e., $s_1 = 1$) or not (i.e., $s_1 = 0$). Step (3) defines action variables a_1, \dots, a_{2r} to represent the Boolean variables z_1, \dots, z_r in the formula ϕ . Step (4) ensures that the pairs of action variables a_{2i-1}, a_{2i} that represent the same Boolean variable z_i take the same value. Step (5) defines the neurons in the first layer (i.e., $l = 1$) of the BNN. Step (6) defines the neurons in the second layer (i.e., $l = 2$) of the BNN. Each neuron $u_{i,2} \in J(2)$ represents a clause c_i in the formula ϕ , and the input of each neuron $u_{i,2}$ is normalized so that $B(i, 2) = 3$. Step (7) defines the weights between the first and the second layers so that the output of neurons $u_{2i,1}, u_{2i+1,1} \in J(1)$ only affects the input of the neurons $u_{j,2} \in J(2)$ in the second layer if and only if the Boolean variable z_i appears in clause c_j . When this is not the case, the output of neurons $u_{2i,1}, u_{2i+1,1}$ are cancelled out due to the different values of their weights, so that $\bar{w}_{2i,j,2} + \bar{w}_{2i+1,j,2} = 0$. Steps (6) and (7) together ensure that for any values of V_1 and $\bar{w}_{1,j,2}$, neuron $u_{j,2} \in J(2)$ is activated if and only if at least one literal in clause c_j evaluates to true.¹ Step (8) defines the single neuron in the third layer (i.e., $l = 3$) of the BNN. Neuron $u_{1,3} \in J(3)$ predicts the value of state variable s_1 . Step (9) defines the weights between the second and the third layers so that the neuron $u_{1,3} \in J(3)$ activates if and only if all clauses in the formula ϕ are satisfied. Finally, step (10) ensures that the values of the actions constitute a solution to the learned planning problem \tilde{I} if and only if all clauses are satisfied. \square

5 Constraint Programming Models for the Learned Planning Problem

In this section, we present two new constraint programming (CP) models to solve the learned automated planning problem \tilde{I} . The models make use of reification rather than restricting themselves to linear constraints. This allows a more direct expression of the BNN constraints.

5.1 Constraint Programming Model 1

Decision Variables and Parameters. The CP model 1 uses the same set of decision variables and parameters as the 0–1 IP model previously described in Sect. 3.

¹ Each neuron $u_{j,2}$ that represents clause c_j receives seven non-zero inputs (i.e., one from state and six from action variables). The bias $B(j, 2)$ is set so that the activation condition holds when at least one literal in clause c_j evaluates to true. For example, the constraint $-2 + \bar{w}_{1,j,2}V_1 + B(j, 2) \geq 0$ represents the case when exactly one literal in clause c_j evaluates to true where the terms -2 and $\bar{w}_{1,j,2}V_1$ represent the inputs from the six action variables and the single state variable, respectively. Similarly, the constraint $-6 + \bar{w}_{1,j,2}V_1 + B(j, 2) < 0$ represents the case when all literals in clause c_j evaluate to false and the activation condition does not hold.

Constraints. The CP model 1 has the following constraints:

$$\begin{aligned} &\text{Constraints (1)–(6)} \\ &(In(j, l, t) \geq 0) = Z_{j,l,t} \quad \forall_{u_{j,l} \in J(l), l \in \{2, \dots, L\}, t \in \{1, \dots, H\}} \end{aligned} \quad (10)$$

where the input expression $In(j, l, t)$ for neuron $u_{j,l} \in J(l)$ in layer $l \in \{2, \dots, L\}$ at time step $t \in \{1, \dots, H\}$ is equal to $\sum_{u_{i,l-1} \in J(l-1)} \bar{w}_{i,j,l} (2 \cdot Z_{i,l-1,t} - 1) + B(j, l)$. In the above model, constraint (10) models the activation of each neuron in the learned BNN by replacing constraints (7)–(8).

Objective Function. The CP model 1 uses the same objective function as the 0–1 IP model previously described in Sect. 3.

5.2 Constraint Programming Model 2

Decision Variables. The CP model 2 uses the $X_{i,t}$ and $Y_{i,t}$ decision variables previously described in Sect. 3.

Parameters. The CP model 2 uses the same set of parameters as the 0–1 IP model previously described in Sect. 3.

Constraints. The CP model 2 has the following constraints:

$$\begin{aligned} &\text{Constraints (1)–(6)} \\ &(In(j, l, t) \geq 0) = Expr_{j,l,t} \quad \forall_{u_{j,l} \in J(l), l \in \{2, \dots, L\}, t \in \{1, \dots, H\}} \end{aligned} \quad (11)$$

where the input expression $In(j, l, t)$ for neuron $u_{j,l} \in J(l)$ in layer $l \in \{2, \dots, L\}$ at time step $t \in \{1, \dots, H\}$ is equal to $\sum_{u_{i,l-1} \in J(l-1)} \bar{w}_{i,j,l} (2 \cdot Expr_{i,l-1,t} - 1) + B(j, l)$, and output expression $Expr_{j,l,t}$ represents the binary output of neuron $u_{j,l} \in J(l)$ in layer $l \in \{2, \dots, L\}$ at time step $t \in \{1, \dots, H\}$. In the above model, constraint (11) models the activation of each neuron in the learned BNN by replacing the decision variable $Z_{j,l,t}$ in constraint (10) with the expression $Expr_{j,l,t}$. The difference between an integer variable and an expression is that during solving the solver does not store the domain (current set of possible values) for an expression. Expressions allow more scope for the presolve of CP Optimizer [17] to rewrite the constraints to a more suitable form, and allow the use of more specific propagation scheduling.

Objective Function. The CP model 2 uses the same objective function as the 0–1 IP model that is previously described in Sect. 3.

6 Model Symmetry

Examining the 0–1 IP (or equivalently the PBO) model, one can see the bulk of the model involves copies of the learned BNN constraints over all time steps.

These constraints model the activation of each neuron (i.e., constraints (4)–(8)) and constrain the input of the BNN (i.e., constraint (3)). The remainder of the model is constraints on the initial and goal states (i.e., constraints (1)–(2)). So if we ignore the initial and goal state constraints, the model is symmetric over the time steps. Note that this symmetry property is not a *global* one: the model is not symmetric as a whole. Rather, this *local* symmetry arises because subsets of constraints are isomorphic to each other. Because of this particular form of symmetry, the classic approach of adding symmetry breaking predicates [5] would not be sound, as this requires global symmetry.

Instead, we exploit this symmetry by deriving symmetric nogoods on-the-fly. If a nogood is derived purely from constraints (3)–(8), and if a sufficiently small subset of time steps was involved in its derivation, then we can shift the time steps of this nogood over the planning horizon, learning a valid symmetric nogood. To track which constraints a nogood is derived from, we use the SAT technique of *marker literals* lifted to PBO. Each constraint is extended with some marker literal, which, if true, enforces the constraint, and if false, trivially satisfies it. During the search, these marker literals are a priori assumed true, so we are solving essentially the same problem, but the nogood learning mechanism of the PBO solver ensures the marker literal of a constraint appears in a nogood if that constraint was required in the derivation of the nogood.

By introducing marker literals L_t for all time steps $t \in \{1, \dots, H\}$ for constraints (3)–(8) and an extra “asymmetric” marker literal L^* for constraint (2) and the constraints originating from bounding the objective function, and then treating all initial state constraints as markers, we can track if only constraints (3)–(8) were involved in the derivation of a nogood, and if so, for which time steps. When we find that the constraints involved in creating a nogood only refer to constraints from time steps l to u , then we know that symmetric copies of these nogoods are also valid for time steps $l + \Delta$ to $u + \Delta$ for all $-l < \Delta < 0$, $0 < \Delta \leq H - u$. Our approach to exploiting symmetry is similar to the ones proposed for bounded model checking in SAT [30,31].

Example 3. Marker literals are used to “turn on” the constraints and they are set to true throughout the search. For example constraint (8) becomes

$$L_t \rightarrow (B(j, l) + J(l - 1) + 1)Z_{j,l,t} - 1 \geq In(j, l, t) \forall_{u,j,l \in J(l), l \in \{2, \dots, L\}, t \in \{1, \dots, H\}}$$

or equivalently, the binary linear constraint

$$\mathbf{M}(1 - L_t) + (B(j, l) + J(l - 1) + 1)Z_{j,l,t} - 1 \geq In(j, l, t) \forall_{u,j,l \in J(l), l \in \{2, \dots, L\}, t \in \{1, \dots, H\}}$$

with \mathbf{M} chosen large enough so that the constraint trivially holds if $L_t = 0$. \square

We consider two ways of symmetric nogood derivation:

- **All:** whenever we discover a nogood that is a consequence of constraints from time steps l to u , we add a suitably renamed copy of the nogood to the variables for time steps $l + \Delta$ to $u + \Delta$ for all $-l < \Delta < 0$, $0 < \Delta \leq H - u$,

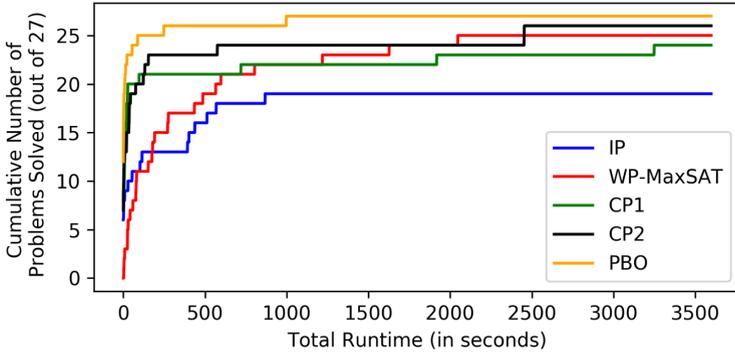


Fig. 4. Cumulative number of problems solved by IP (blue), WP-MaxSAT (red), CP1 (green), CP2 (black) and PBO (orange) models over 27 instances of the problem \tilde{I} within the time limit. (Color figure online)

- **Propagate:** we consider each possible Δ above, but only add the renamed nogood if it will immediately propagate or fail, similar to a SAT symmetric clause learning heuristic [8].

Finally, we denote **Base** as the version of RoundingSat that does not add the symmetric nogoods.

Example 4. Consider the problem in Example 1. Assume we generate a nogood $X_{1,1} \vee Y_{1,1} \vee \neg X_{1,2}$, which is a consequence only of constraints for the BNNs for time steps 1 and 2. The actual generated nogood is then $\neg L_1 \vee \neg L_2 \vee X_{1,1} \vee Y_{1,1} \vee \neg X_{1,2}$ which illustrates that it depends only on the constraints in time steps 1 and 2. We can then add a symmetric copy $\neg L_2 \vee \neg L_3 \vee X_{1,2} \vee Y_{1,2} \vee \neg X_{1,3}$ for time steps 2 and 3, as well as $\neg L_3 \vee \neg L_4 \vee X_{1,3} \vee Y_{1,3} \vee \neg X_{1,4}$ for steps 3 and 4. These new constraints must be correct, since the BNN constraints for time steps $t \in \{1, 2, 3, 4\}$ are all symmetric. The marker literals are added so that later nogoods making use of these nogoods also track which time steps were involved in their generation. Using **All** we add both these nogoods, using **Propagate** we only add those that are unit or false in the current state of the solver. \square

7 Experimental Results

In this section, we present results on two sets of computational experiments. In the first set of experiments, we compare different approaches to solving the learned planning problem \tilde{I} with mathematical optimization models. In the second set of experiments, we present preliminary results on the effect of deriving symmetric nogoods when solving \tilde{I} over long horizons H .

7.1 Experiments 1

We first experimentally test the runtime efficiency of solving the learned planning problem \tilde{I} with mathematical optimization models using off-the-shelf solvers.

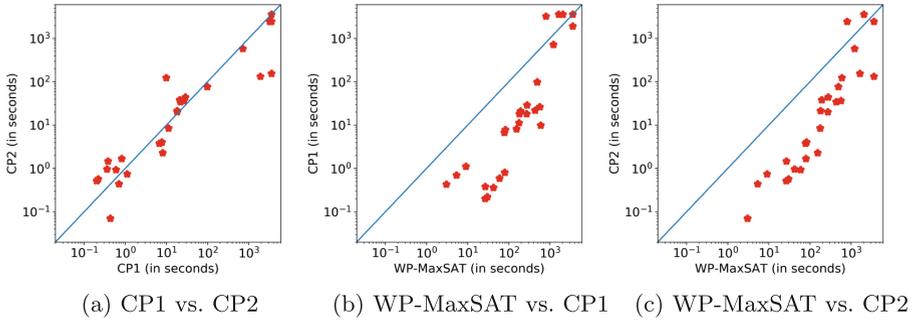


Fig. 5. Pairwise runtime comparison between WP-MaxSAT, CP1 and CP2 models over 27 instances of problem \tilde{I} within the time limit.

All the existing benchmark instances of the learned planning problem \tilde{I} (i.e., 27 in total) were used [26]. We ran the experiments on a MacBookPro with 2.8 GHz Intel Core i7 16GB memory, with one hour total time limit per instance. We used the MIP solver CPLEX 12.10 [17] to optimize the 0–1 IP model, MaxHS [6] with underlying CPLEX 12.10 linear programming solver to optimize the WP-MaxSAT model [26], and CP Optimizer 12.10 [17] to optimize the CP Model 1 (CP1) and the CP Model 2 (CP2). Finally, we optimized a pseudo-Boolean optimization (PBO) model, which simply replaces all binary variables in the 0–1 IP model with Boolean variables, using RoundingSat [10].

In Fig. 4, we visualize the cumulative number of problems solved by all five models, namely: IP (blue), WP-MaxSAT (red), CP1 (green), CP2 (black) and PBO (orange), over 27 instances of the learned planning problem \tilde{I} within one hour time limit. Figure 4 clearly highlights the experimental efficiency of solving the PBO model. We find that using the PBO model with RoundingSat solves all existing benchmarks under 1000s. In contrast, we observe that the 0–1 IP model performs poorly, with only 19 instances out of 27 solved within the one hour time limit. The remaining three models, WP-MaxSAT, CP1 and CP2, demonstrate relatively comparable runtime performance, which we explore in more detail next.

In Figs. 5a, 5b and 5c, we present scatter plots comparing the WP-MaxSAT, CP1 and CP2 models. In each figure, each dot (red) represents an instance of the learned planning problem \tilde{I} and each axis represents a model (i.e., WP-MaxSAT, CP1 or CP2). If a dot falls below the diagonal line (blue), it means the corresponding instance is solved faster by the model represented by the y-axis than the one represented by the x-axis. In Fig. 5a, we compare the two CP models CP1 and CP2. A detailed inspection of Fig. 5a shows a comparable runtime performance on the instances that take less than 1000s to solve (i.e., most dots fall closely to the diagonal line). In the remaining two instances that are solved by CP2 under 1000 s, CP1 runs out of the one hour time limit. These results suggest that using expressions instead of decision variables to model the neurons of the learned BNN allows the CP solver to solve harder instances (i.e., instances that

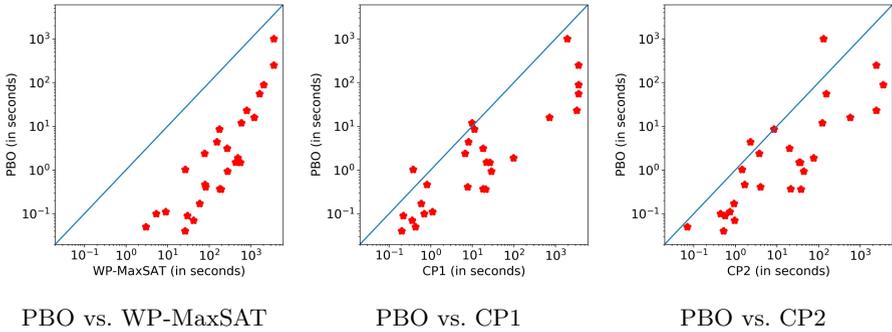


Fig. 6. Pairwise runtime comparison between PBO, and WP-MaxSAT, CP1 and CP2 models over 27 instances of problem \tilde{I} within the time limit.

take more than 1000 s to solve) more efficiently. In Figs. 5b and 5c, we compare CP1 and CP2 against the WP-MaxSAT model, respectively. Both figures show a similar trend on runtime performance; the CP models close instances that take less than 1000 s to solve by one to two orders of magnitude faster than the WP-MaxSAT model, and the WP-MaxSAT model performs comparably to the CP models on the harder instances. Overall, we find that WP-MaxSAT solves one more and one less instances compared to CP1 and CP2 within the one hour time limit, respectively. These results suggest that the WP-MaxSAT model pays a heavy price for the large size of its compilation when the instances take less than 1000 s to solve, and only benefits from its SAT-based encoding for harder instances.

Next, we compare the runtime performance of WP-MaxSAT, CP1 and CP2 against the best performing model (i.e., PBO) in more detail in Figs. 6a, 6b and 6c. These plots show that the PBO model significantly outperforms the WP-MaxSAT, CP1 and CP2 models across all instances. Specifically, Fig. 6a shows that the PBO model is better than the previous state-of-the-art WP-MaxSAT model across all instances by one to two orders of magnitude in terms of runtime performance. Similarly, Figs. 6b and 6c show that the PBO model outperforms both CP models across all instances, except in one and two instances, respectively, by an order of magnitude.

It is interesting that the 0–1 IP model works so poorly for the MIP solver, while the equivalent PBO model is solved efficiently using a PBO solver. It seems that the linear relaxations used by the MIP solver are too weak to generate useful information, and it ends up having to fix activation variables in order to reason meaningfully. In contrast, it appears that the PBO solver is able to determine some useful information from the neuron constraints without necessarily fixing the activation variables—probably since it uses integer-based cutting planes reasoning [4] rather than continuous linear programming reasoning for the linear expressions—and the nogood learning helps it avoid repeated work.

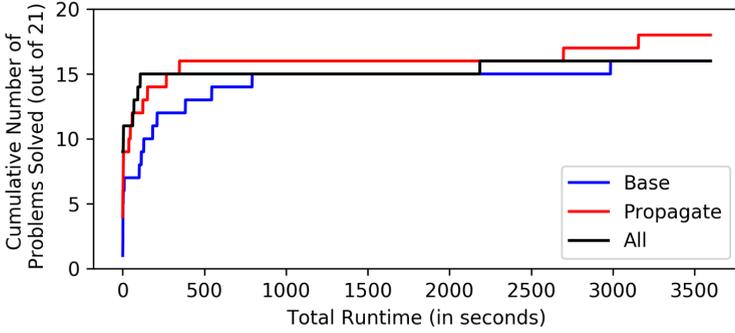


Fig. 7. Cumulative number of problems solved for Base (blue), Propagate (red) and All (black) models over 21 instances of the problem \tilde{I} within the time limit. (Color figure online)

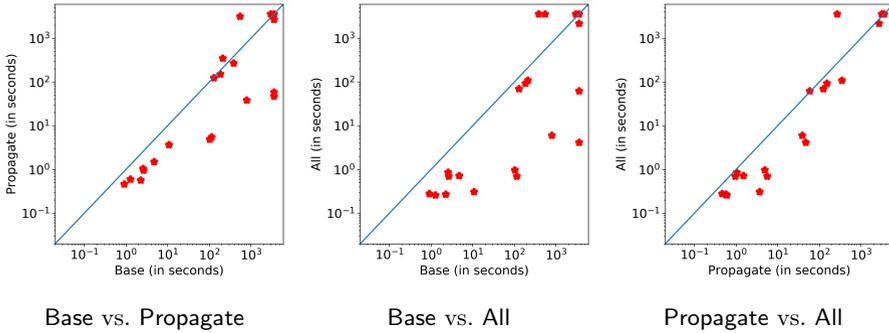


Fig. 8. Pairwise runtime comparison between Base, Propagate and All models over 21 instances of problem \tilde{I} within the time limit.

7.2 Experiments 2

We next evaluate the effect of symmetric nogood derivation on solving the learned planning problem \tilde{I} over long horizons H . For these experiments, we generated instances by incrementing the value of the planning horizon H in the benchmark instances in Sect. 7.1, and used the same hardware and time limit settings. We modified the best performing solver RoundingSat [10] to include symmetry reasoning as discussed in Sect. 6.

In Fig. 7, we visualize the cumulative number of problems solved for all three versions of RoundingSat, namely Base (blue), Propagate (red), and All (black), over 21 instances of the learned planning problem \tilde{I} over long horizons H within one hour time limit. Figure 7 demonstrates that symmetric nogood derivation can improve the efficiency of solving the underlying PBO model. We find that Propagate solves the most instances within the time limit. A more detailed inspection of the results further suggests that between the remaining two version of RoundingSat, All solves more instances faster compared to Base.

Next, in Figs. 8a, 8b and 8c, we explore the pairwise runtime comparisons of the three versions of RoundingSat in more detail. In Figs. 8a and 8b, we compare `Propagate` and `All` against `Base`, respectively. It is clear from these scatter plots that `Propagate` and `All` outperform `Base` in terms of runtime performance. Specifically, in Fig. 8b, we find that `All` outperforms `Base` by up to three orders of magnitude in terms of runtime performance. Finally, in Fig. 8c, we compare the two versions of RoundingSat that are enhanced with symmetric nogood derivation. A detailed inspection of Fig. 8c reveals that `All` is slightly faster than `Propagate` in general.

8 Related Work, Conclusions and Future Work

In this paper, we studied the important problem of automated planning with learned BNNs, and made four important contributions. First, we showed that the feasibility problem is *NP*-complete. Unlike the proof presented for the task of verifying learned BNNs [3], our proof does not rely on setting weights to zero (i.e., sparsification). Instead, our proof achieves the same expressivity for fully connected BNN architectures, without adding additional layers or increasing the width of the layers, by representing each input with two copies of action variables. Second, we introduced two new CP models for the problem. Third, we presented detailed computational results for solving the existing instances of the problem. Lastly, we studied the effect of deriving symmetric nogoods on solving new instances of the problem with long horizons.

It appears that BNN models provide a perfect class of problems for pseudo-Boolean solvers, since each neuron is modelled by pseudo-Boolean constraints, but the continuous relaxation is too weak for MIP solvers to take advantage of, while propagation-based approaches suffer since they are unable to reason about linear expressions directly. PBO solvers directly reason about integer (0–1) linear expressions, making them very strong on this class of problems.

Our results have the potential to improve other important tasks with learned BNNs (and other DNNs), such as automated planning in real-valued action and state spaces [27, 35, 36], decision making in discrete action and state spaces [21], goal recognition [11], training [33], verification [9, 15, 19, 22], robustness evaluation [32] and defenses to adversarial attacks [34], which rely on efficiently solving similar problems that we solve in this paper. The derivation of symmetric nogoods is a promising avenue for future work, in particular, if a sufficient number of symmetric nogoods can be generated. Relating the number of derived symmetric nogoods to the wall-clock speed-up of the solver or the reduction of the search tree might shed further light on the efficacy of this approach.

Acknowledgements. Some of our preliminary computational experiments used resources provided by the Swedish National Infrastructure for Computing (SNIC) at the High Performance Computing Center North (HPC2N) at Umeå University. Jo Devriendt and Jakob Nordström were supported by the Swedish Research Council grant 2016-00782, and Jakob Nordström also received funding from the Independent Research Fund Denmark grant 9040-00389B.

References

1. Bennett, S.W., DeJong, G.F.: Real-world robotics: learning to plan for robust execution. *Mach. Learn.* **23**, 121–161 (1996)
2. Benson, S.S.: Learning action models for reactive autonomous agents. Ph.D. thesis, Stanford University, Stanford, CA, USA (1997)
3. Cheng, C.-H., Nührenberg, G., Huang, C.-H., Ruess, H.: Verification of binarized neural networks via inter-neuron factoring. In: Piskac, R., Rümmer, P. (eds.) *VSTTE 2018*. LNCS, vol. 11294, pp. 279–290. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_16
4. Cook, W., Coullard, C.R., Turán, G.: On the complexity of cutting-plane proofs. *Discret. Appl. Math.* **18**(1), 25–38 (1987)
5. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. In: *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pp. 148–159. Morgan Kaufmann (1996)
6. Davies, J., Bacchus, F.: Solving MAXSAT by solving a sequence of simpler SAT instances. In: Lee, J. (ed.) *CP 2011*. LNCS, vol. 6876, pp. 225–239. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_19
7. Davies, T.O., Pearce, A.R., Stuckey, P.J., Lipovetzky, N.: Sequencing operator counts. In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling*, pp. 61–69. AAAI Press (2015)
8. Devriendt, J., Bogaerts, B., Bruynooghe, M.: Symmetric explanation learning: effective dynamic symmetry handling for SAT. In: Gaspers, S., Walsh, T. (eds.) *SAT 2017*. LNCS, vol. 10491, pp. 83–100. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_6
9. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D’Souza, D., Narayan Kumar, K. (eds.) *ATVA 2017*. LNCS, vol. 10482, pp. 269–286. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_19
10. Ellfers, J., Nordström, J.: Divide and conquer: towards faster pseudo-boolean solving. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018*, pp. 1291–1299 (2018)
11. Fraga Pereira, R., Vered, M., Meneguzzi, F., Ramírez, M.: Online probabilistic goal recognition over nominal models. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pp. 5547–5553. International Joint Conferences on Artificial Intelligence Organization (2019)
12. Gil, Y.: Acquiring domain knowledge for planning by experimentation. Ph.D. thesis, Carnegie Mellon University, USA (1992)
13. Helmert, M.: The fast downward planning system. *J. Artif. Intell. Res.* **26**, 191–246 (2006)
14. Hoffmann, J., Nebel, B.: The FF planning system: fast plan generation through heuristic search. *J. Artif. Intell. Res.* **14**, 253–302 (2001)
15. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčák, V. (eds.) *CAV 2017*. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1
16. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks. In: *Proceedings of the Thirtieth International Conference on Neural Information Processing Systems, NIPS 2016*, pp. 4114–4122. Curran Associates Inc., USA (2016)
17. IBM: IBM ILOG CPLEX Optimization Studio CPLEX User’s Manual (2020)

18. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. In: Proceedings of the Thirty-Second International Conference on International Conference on Machine Learning, ICML, pp. 448–456. JMLR.org (2015)
19. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
20. Kautz, H., Selman, B.: Planning as satisfiability. In: Proceedings of the Tenth European Conference on Artificial Intelligence, ECAI 1992, pp. 359–363 (1992)
21. Lombardi, M., Gualandi, S.: A Lagrangian propagator for artificial neural networks in constraint programming. *Constraints* **21**, 435–462 (2016)
22. Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, pp. 6615–6624 (2018)
23. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco (2004)
24. Pommerening, F., Röger, G., Helmert, M., Bonet, B.: LP-based heuristics for cost-optimal planning. In: Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, pp. 226–234. AAAI Press (2014)
25. Say, B., Sanner, S.: Planning in factored state and action spaces with learned binarized neural network transition models. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, pp. 4815–4821 (2018)
26. Say, B., Sanner, S.: Compact and efficient encodings for planning in factored state and action spaces with learned binarized neural network transition models. *Artif. Intell.* **285**, 103291 (2020)
27. Say, B., Sanner, S., Thiébaux, S.: Reward potentials for planning with learned neural network transition models. In: Schiex, T., de Givry, S. (eds.) CP 2019. LNCS, vol. 11802, pp. 674–689. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30048-7_39
28. Say, B., Wu, G., Zhou, Y.Q., Sanner, S.: Nonlinear hybrid planning with deep net learned transition models and mixed-integer linear programming. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, pp. 750–756 (2017)
29. Shen, W.M., Simon, H.A.: Rule creation and rule learning through environmental exploration. In: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI 1989, pp. 675–680. Morgan Kaufmann Publishers Inc., San Francisco (1989)
30. Shtrichman, O.: Tuning SAT checkers for bounded model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 480–494. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_36
31. Shtrichman, O.: Pruning techniques for the SAT-based bounded model checking problem. In: Margaria, T., Melham, T. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 58–70. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44798-9_4
32. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: Proceedings of the Seventh International Conference on Learning Representations. ICLR (2019)

33. Toro Icarte, R., Illanes, L., Castro, M.P., Cire, A.A., McIlraith, S.A., Beck, J.C.: Training binarized neural networks using MIP and CP. In: Schiex, T., de Givry, S. (eds.) CP 2019. LNCS, vol. 11802, pp. 401–417. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30048-7_24
34. Wong, E., Kolter, Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: Proceedings of the Thirty-Fifth International Conference on Machine Learning. ICML (2018)
35. Wu, G., Say, B., Sanner, S.: Scalable planning with tensorflow for hybrid nonlinear domains. In: Proceedings of the Thirty First Annual Conference on Advances in Neural Information Processing Systems, Long Beach, CA (2017)
36. Wu, G., Say, B., Sanner, S.: Scalable planning with deep neural network learned transition models. *J. Artif. Intell. Res.* **68**, 571–606 (2020)