UCLouvain
Institute for Information
and Communication Technologies,
Electronics and Applied Mathematics

Computational limitations in Artificial Intelligence algorithms

Brieuc Pinon

Thesis submitted in partial fulfillment
of the requirements for the degree of
Docteur en sciences de l'ingénieur

Dissertation committee:

Prof. Jean-Charles Delvenne (UCLouvain, advisor)
Prof. Raphaël Jungers (UCLouvain, advisor)
Prof. Vincent François-Lavet (VU Amsterdam, Netherlands)
Prof. Marco Saerens (UCLouvain)
Prof. Johan Suykens (KU Leuven, Belgium)
Prof. Pierre-Antoine Absil (UCLouvain, chair)

Version of September, 2025 .

Abstract

This thesis investigates fundamental limitations and potential opportunities of learning-based Artificial Intelligence algorithms.

A contribution is the comparison of sample efficiency between expressive Deep Learning architectures close to Turing-complete systems of computation, to simpler architectures stacking linear operators and non-linear activation functions. We conclude that a detailed characterization of potential gains is tightly linked to open problems in Computational Complexity, but that an assumption from derandomization research entails a clear advantage to expressive architectures.

Additionally, we study empirically the relevance of the Transformer architecture and online planning in meta-Reinforcement Learning. Our numerical results suggest that an architecture like the Transformer is useful to model partially observable problem dynamics arising in meta-Reinforcement Learning. Moreover, extensive planning is crucial to handle the exploration and exploitation of the unknown dynamics.

This work also analyzes the interplay between learning and algorithms for problem-solving. We formalize computational limitations through counterexamples for multiple classical algorithms in Reinforcement Learning, such as algorithms treating the model of the dynamics as a black-box and universal value function estimating state-to-state reachability. These fundamental limitations open the possibility of designing more efficient algorithms leveraging Machine Learning for problem-solving.

Acknowledgements

I want to thank my promoters, Jean-Charles and Raphaël, for making this thesis possible. First, through their trust and support in launching it, then through their curiosity, patience, and advice when reading and listening to my sometimes imperfectly expressed ideas.

I am thankful as well to the members of my jury for their time and expertise in evaluating this work: Vincent, Marco, Johan, and to Pierre-Antoine who served as chair.

I am also grateful to the organizations in which I had the chance to evolve, and to the people who make these organizations work: the INMA department, the UCL, and the EPL. More broadly, I would like to thank the Belgian teaching and research system, which allowed me to conduct this thesis without financial pressure.

My journey in this thesis was also accompanied by many valuable new and old relationships built throughout my studies, in my kot, with colocs, and among colleagues; these make it all worth it. In particular, I want to thank Maxine for her (critical) advice on my presentations and her daily support both in my thesis and life.

Empirical evidence suggests that this thesis would not exist if not for my parents, Sophie and Christian. They cultivated my curiosity since as early as I can remember, and cared all along the way.

Contents

List of acronyms

AI	**A**rtificial **I**ntelligence
ANN	**A**rtificial **N**eural **N**etwork
ATP	**A**utomated **T**heorem **P**roving
CDCL	**C**onflict-**D**riven **C**lause **L**earning
CNN	**C**onvolutional **N**eural **N**etwork
DL	**D**eep **L**earning
GNN	**G**raph **N**eural **N**etwork
GRU	**G**ated **R**ecurrent **U**nits
HER	**H**indsight **E**xperience **R**eplay
LSTM	**L**ong **S**hort-**T**erm **M**emory
MDL	**M**inimum **D**escription **L**ength
MDP	**M**arkov **D**ecision **P**rocess
ML	**M**achine **L**earning
MPC	**M**odel **P**redictive **C**ontrol
POMDP	**P**artially **O**bservable **M**arkov **D**ecision **P**rocess
PPO	**P**roximal **P**olicy **O**ptimization
RL	**R**einforcement **L**earning

RNN **R**ecurrent **N**eural **N**etworks

SAT Boolean **Sat**isfiability

TRPO **T**rust **R**egion **P**olicy **O**ptimization

UVF **U**niversal **V**alue **F**unctions

List of symbols

$1(.)$	Indicator function.
$[n]$	The set $\{1, 2, \ldots, n\}$.
$\mathcal{A}$	Set of actions in an MDP/POMDP.
$\lvert.\rvert_0$	L0 norm (number of non-zero elements).
α	Learning rate or step-size parameter; parameter for goal-conditioned algorithm (number of active features); constant in ANN/Circuit proposition.
$\bar{Q}$	Target Q-network or an old version of the Q-value function.
$\bar{s}$	State reference number (pointer) in RL interface.
β	Parameter, e.g., weight of entropy regularization in PPO; constant in ANN/Circuit proposition.
$\cap$	Set intersection.
$\cup$	Set union.
δ	Confidence parameter in PAC-learning; small positive constant.
$\Delta(X)$	The set of probability measures over a set X.
ϵ	Error parameter in PAC-learning; exploration parameter (e.g., in ϵ-greedy).
η	Learning rate or scalar factor for gradient descent.

$\exists$	Existential quantifier (there exists).
$\forall$	Universal quantifier (for all).
γ	Discount factor in RL; small positive constant/exponent.
$\hat{f}$	A hypothesis function, an approximation of f.
$\in$	Set membership (is an element of).
$\lceil . \rceil$	Ceiling function.
$\leq, \geq, <, >$	Less than (or equal to), greater than (or equal to).
$\leftarrow$	Assignment operator.
$\lfloor . \rfloor$	Floor function.
$\mathbb{N}$	The set of natural numbers.
$\mathbb{R}$	The set of real numbers.
$\mathcal{B}$	The Boolean set $\{0, 1\}$.
$\mathcal{B}^*$	The set of all finite length binary strings.
$\mathcal{B}^n$	The set of n-bit binary strings.
$\mathcal{M}$	A family of RL problems.
$\mathcal{U}$	Universal Turing Machine interpreter.
$\mathcal{U}^c$	Polynomial-time Universal Turing Machine interpreter.
∇	Gradient operator.
Ω	Big Omega notation (asymptotic lower bound); Set of possible observations in a POMDP.
$\perp$	Symbol representing non-halting execution (TMs) or invalid/terminal output (RL interface).
$\phi_G(x)$	The G-profile of state x.
π, π_θ	A policy, possibly parameterized by θ.
π^U	Uniform random policy.

π^V	Sampling policy derived from a set of value functions V.
$\prod$	Product.
$\rightarrow$	Maps to (function notation).
$\mathcal{S}$	Set of states in an MDP/POMDP.
$\langle .,. \rangle$	Tuple or inner product; encoding of two strings.
Σ_2^p	A complexity class in the polynomial hierarchy.
$\sim$	Sampled from (distribution).
$\subseteq, \subset$	Subset / proper subset.
$\sum$	Summation.
τ	Small positive constant (exponent); Temperature parameter.
θ	Parameters of a model (e.g., neural network).
$\tilde{G}$	Auxiliary sample efficiency gain (provable from PAC guarantee).
φ	Interpreters (e.g., Turing machines, circuits).
a, a_t	An action, or action at step t.
$Check(x; p)$	Function checking if binary vector x satisfies CNF-SAT instance p.
D	A dataset of transitions or experiences.
D^p	Dynamics operator for CNF-SAT instance p.
E	Encoding of a Turing Machine; Set of languages (e.g., decidable in exponential time); Encoding function (e.g., one-hot).
F	Set of final states in a Turing Machine; A set of functions (e.g., linear functions with a threshold, set of functions satisfying a constraint in Definition 19).
f	A (Boolean) function, often the target function to be learned.
F^D	Set of functions derived from an interpreter with description length up to D.

G	Sample efficiency gain; A (finite) function class in RL interface definition; A Q-value estimate in tree search backup.
g	A goal-state in HER; a specific function in a function class.
H	Horizon of an MDP/POMDP.
H^n	A hypothesis class of functions from B^n to B.
I	An index list; Number of samples/iterations.
K	A constant; Number of iterations/samples/sub-problems/branches.
L	A language (set of strings); A learning algorithm for neural networks.
m	Number of samples in a dataset.
$m_\phi^{\epsilon,\delta}(f,P)$	Minimal number of samples for algorithm MDL^ϕ to PAC-learn (f,P) with parameters ϵ,δ.
MDL^ϕ	Learning algorithm based on the Minimum Description Length principle with interpreter ϕ.
N	A natural number; Number of samples; Number of nodes in a tree search.
n	Input size (number of Boolean variables or dimension of state space).
$N(.)$	Visit count for a node/action in tree search (sometimes $N^{node}(a)$ or $N^{root}(a)$).
O	Observation function in a POMDP.
$O(.)$	Big O notation (asymptotic upper bound).
o_t	Observation at step t.
$OP(M)$	Set of optimal Q-functions and policies for a family of RL problems M.
P	Probability measure; MDP transition operator (defining P_0 and P_{dyn}); Complexity class.
p	A CNF-SAT instance (problem); permutation of coordinates.

$P_0(s_0)$	Initial state distribution in an MDP.
$P_{\text{dyn}}(r, s'\|s, a)$	State transition dynamics in an MDP, yielding reward r and next state s'.
Q, Q_θ	Q-value function, possibly parameterized by θ.
$Q^{node}(a)$	Q-value estimate for action a at a specific node in tree search.
R	Cumulative future rewards in a trajectory; Set of rules of a Turing Machine.
r, r_t	A reward, or reward at step t.
s, s_t	A state, or state at step t.
T	A Turing machine; Temperature parameter in tree search action selection.
$v(s_1, s_2; p)$	Universal value function evaluated for reaching state s_2 from s_1 in instance p.
$v(x_{\leq i}; p)$	Value function evaluated for CNF-SAT instance p and partial assignment $x_{\leq i}$.
V, V_θ	Value function, possibly parameterized by θ; Set of (hypothesis) value functions.
V^t	Set of hypothesis value functions at iteration t.
$VC(.)$	VC-dimension of a hypothesis class.
X	A domain or set.
x_I	Vector x with coordinates selected by index list I.
$x_{\leq i}$	Vector x restricted to its first i coordinates.

1

Learning for problem-solving

Learning is a fundamental tool in Artificial Intelligence. Despite its centrality, there is no consensus on how best to leverage learning to solve problems. In this chapter, we present a variety of algorithmic ideas that integrate learning for problem-solving. These algorithms address a diverse set of challenges ranging from optimizing robot policies and strategies for the game of GO, to creating novel synthetic molecules and Automated Theorem Proving (ATP). Although the target problems differ in their properties, our discussion abstracts these differences to focus on the role of the learning component: what knowledge is acquired, by which means, and how it is exploited.

Our discussion is structured around several of the dimensions that emerge in the literature and are most relevant to this manuscript.

Levels of learning Learning can operate at two distinct levels. At the instance level, learning is used during the process of solving one specific problem instance. At the algorithm level, learning is used a priori to develop an algorithm (or parts thereof), which is then applied to new problem instances.

We define informally this distinction similarly to Thrun and Pratt [1998], an algorithm learns (part of) an algorithm if it leverages an auxiliary source of tasks or data to solve the problem instance of interest. We refer the in-

terested reader to Hospedales et al. [2021] for a review of the field of meta-learning with a discussion on possible formalizations.

Deductive vs inductive learning Another axis is the type of learning, which can be deductive or inductive. In the deductive case, a set of facts is iteratively grown using logical inference to derive provably entailed new facts. In contrast, inductive learning derives new knowledge from data by generalization using a Machine Learning algorithm.

This discrimination between logics is usual in Computer Science [Russell and Norvig, 2016] and also exists in Philosophy [Russell, 1946, Hawthorne, 2025].

Problem model vs solution search Learning can be used to build an understanding of the problem itself, for example, by building a world model that allows predicting the consequences of actions and evaluating a potential plan. But even with a perfect world model (obtained through learning or prior knowledge), finding an optimal solution might remain computationally hard [Mundhenk et al., 2000, Arora and Barak, 2009]. Additionally, learning can also address this issue by guiding the search process.

Building a model of the dynamics or not is an usually underlined characteristic of Reinforcement Learning algorithms [Sutton and Barto, 2018] or Adaptive Control under the name of indirect control or system identification [Ioannou and Sun, 1996].

Representation The type of representation manipulated by the algorithm is also an important property. Discrete/symbolic representations (like logical formulas, graphs, or binary vectors) can be used to enforce a strong inductive bias, hoping to improve generalization, but are also necessary to leverage deductive symbolic reasoning algorithms. Continuous/neural representations (like real vectors in high-dimensional spaces manipulated by neural networks) have the advantage of being easier to handle and optimized with Deep Learning using gradient-based methods. At the intersection, neuro-symbolic algorithms attempt to interface and reconcile these two representations, trying to extract the best of both worlds.

Our dimension aims to capture the differentiation in the literature between symbolic and neural, connectionist, sub-symbolic approaches [Bader and Hitzler, 2005, Kautz, 2022, Garcez and Lamb, 2023].

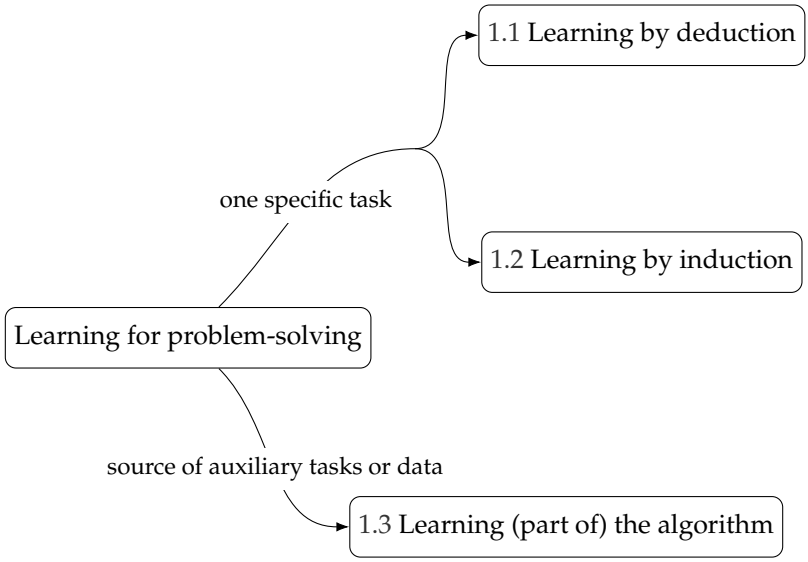

Fig. 1.1 Classification of our sections along our first two dimensions. Does the algorithm learn to solve the task of interest using auxiliary tasks or data? Is the algorithm learning by deduction or induction (Machine Learning)?

Learned quantities and feedback A central distinction in our presentation is also simply: what is learned? What are the input-output of the learned function? And with what feedback is it learned?

Exploitation How is the acquired knowledge exploited? Does the algorithm directly follow the output of a Machine Learning model (e.g., policy network)? Or does it use a planning procedure on top of a learned component (like a model of the dynamics)?

The decomposition of this chapter into sections and subsections is organized primarily along the dimensions of learning level and deductive vs inductive. In the following sections, we first discuss symbolic algorithms that leverage deduction during search, followed by methods that employ inductive learning to guide the search process, such as Reinforcement Learning. Both of these sections present algorithms operating at the instance level, learning from and for a specific task. Then, we examine meta-algorithms where a significant part (if not all) of the problem-solving procedure is learned in advance. This structure is pictured in Figure 1.1.

Finally, we provide at the end of this chapter an outline of this Thesis that analyzes and contrasts the different methods we present in this chapter.

1.1 Learning by deduction

We begin with a discussion of symbolic algorithms that leverage deduction for problem-solving. We focus on the archetype Boolean Satisfiability (SAT) problem, to which many problems reduce and which has historically motivated the development of many algorithmic techniques. In particular, the conflict-driven clause learning algorithm (CDCL) forms the backbone of modern SAT solvers [Silva and Sakallah, 1996, Biere et al., 2009, Knuth, 2015].

A CDCL algorithm starts with a set of constraints that define the SAT instance. The algorithm iteratively attempts to generate a complete solution guided by its current set of constraints. When its attempt fails, it leverages a logical inference rule, resolution, to deduce a new constraint preventing a similar failure in future attempts. This process refines the search space and guides the algorithm toward promising regions. The high-level steps are as follows:

1. Begin with a set of constraints defining the SAT instance.
2. Incrementally, assign truth values to the variables guided by the constraints.
3. Upon failure, apply resolution to deduce a new constraint ruling out a similar failure in the generation process.
4. Go back to 2. until a solution is found.

This scheme fits the representation in Figure 1.2 where a guide iteratively generates and learns from data. In that representation, the guide is defined by the set of learned constraints since those are used to direct the generation process. The data represents the failed generations with the constraints leading to each failure, this data can then be used by the deductive procedure to learn new constraints for the guide.

In the context of our dimensions, a CDCL algorithm learns at the instance level. Also, an instance is completely defined by its constraints, which are assumed to be given a priori at step 1. So learning is not required to determine the problem, but rather to inform the search process (solution search). Step 3. creates new knowledge through deduction using

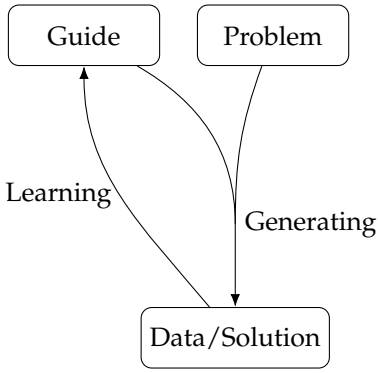

Fig. 1.2 A typical architecture for integrating learning in an algorithm for problem-solving: learning a guide iteratively generating data to learn from and improve itself.

a symbolic (discrete) representation. This knowledge is then exploited to guide future attempts at step 2.

1.2 Learning by induction

Shifting from logical deduction, inductive approaches leverage data-driven methods to acquire knowledge that supports the search for solutions. This section primarily focuses on Reinforcement Learning (RL) methods that learn from interactions with the problem.

For a presentation of RL, we refer the reader to Sutton et al. [1998], François-Lavet et al. [2018], Murphy [2024]. We differentiate beween model-free RL and model-based RL [Mordatch and Hamrick, 2020, Moerland et al., 2023].

Inductive learning here is typically applied at the instance level, guiding the search within a specific problem or environment instance. However, the learned policies or value functions could be used to generalize across similar problems. In that case, the algorithm becomes part of the next section, where (part of) the algorithm is learned from an auxiliary source of tasks.

In RL, the problem takes the form of taking a sequence of actions in an environment. Starting in a sampled initial state according to some distribution, the algorithm can then choose an action given the state; this action leads to a new state according to the dynamics of the problem, and a scalar

reward is received; this is repeated until an end-state is reached. This sequence of state, action, reward, state, etc., forms a trajectory. A *policy* is a map from the current state to an action to take.

The goal for the RL algorithm is to optimize its policy to maximize the expected sum of the rewards in trajectories. We focus on *online RL* where the algorithm can freely sample new trajectories to optimize decision-making. Some presented algorithms also assume access to a generative model where transitions from the dynamics can be sampled from any chosen state. Other algorithms further assume access to a description of the function making up the dynamics.

We note that depending on the problem, RL can take different roles. If the end goal is to get a good policy, RL directly outputs that object (like learning a policy for a robot). If the goal is to sample a good trajectory (like generating a good molecule description or a proof of a theorem), RL is used to construct a good sampler to orient the search, much like the learned clauses in CDCL guide the generation of solutions.

1.2.1 Model-free Reinforcement Learning

Model-free RL methods learn control *policies* or *value functions* directly from experienced trajectories without explicitly building a model of the environment dynamics.

Policy gradient

One way to address RL is to directly represent and optimize a policy. The policy can be represented by a neural network mapping states to action probabilities. Under this Deep Learning form, it can be directly optimized using stochastic gradient ascent on the expected cumulative reward objective.

By the Policy Gradient Theorem this gradient can be expressed using an expectation of trajectories data. The gradient of the expected cumulative rewards with respect to the parameters θ of the policy π_θ can be written:

$$\mathbb{E}_{s,a,R}\left[\nabla_\theta \log(\pi_\theta(a|s))R\right], \tag{1.1}$$

where s is the state, a is the action, and R is the future cumulative rewards in a step inside a trajectory, all of which are sampled according to the policy and environment dynamics. This expression can thus be approximated with an empirical average by sampling trajectories according to the current policy. This observation leads to practical algorithms.

Key algorithms leveraging policy gradients include REINFORCE, Trust Region Policy Optimization (TRPO), and Proximal Policy Optimization (PPO) [Williams, 1987, Schulman et al., 2015, 2017].

In its simplest form, like the REINFORCE algorithm, the feedback used to construct the approximate gradient and learn is sampled future cumulative rewards. A challenge for this approach is the imprecision between the stochastic approximation of the gradient and the true gradient. Methods like TRPO and PPO tame this challenge by using, in addition, value functions (presented after) to improve the estimation of expected future cumulative rewards for the gradient approximation.

This approach fits Figure 1.2 where the guide is the policy learned from previously generated trajectories. If the problem is to find a good policy then the guide is itself the solution.

Value-based methods (Bellman equation)

Alternatively, or complementarily, RL algorithms can learn a (Q-)value function estimating (maximum) expected future task-rewards given a state (and an action). These functions can be turned into a policy by taking the action leading to the highest estimated value.

The (Q-)value function can be a neural network and optimized with gradient-based methods [Mnih et al., 2013]. These functions can be learned using Dynamic Programming in the form of the Bellman equation. This equation constrains the evaluations of the (Q-)value to be consistent with the dynamics and itself.

The Bellman equation for a Q-value function reads

$$Q(s,a) = \mathbb{E}_{r,s'}\left[r + \max_{a'} Q(s',a')\right], \tag{1.2}$$

where s is a state, a an action, r and s' the next reward and state sampled according to the problem's dynamics, a' is a potential action to be taken in the next state. Enforcing this equation at all pairs of state and action produces a Q-value function predicting the best achievable expected cumulative rewards at any state, and can then be converted into an optimal policy.

Enforcing this equation at all pairs of state and action is not practicable in large state spaces. Thus, RL algorithms usually iteratively enforce the equation on previously sampled states. Moreover, using the current Q-value function itself for learning leads to instabilities in the learning pro-

cess. One possibility is to use an old version of the Q-value function and update it regularly. For example, this expression can be iteratively minimized in practice to learn a Q-value function

$$\sum_{s,a,r,s' \in D} \left[Q(s,a) - (r + \max_{a'} \bar{Q}(s',a')) \right]^2, \tag{1.3}$$

where D is a previously collected dataset of transition and $\bar{Q}$ is an old version of the Q-value function from the previous iteration.

Like policy gradient methods, this scheme also fits the representation in Figure 1.2 where the guide is instead a Q-value function and the generated data additionally includes information on old evaluations of the Q-value function.

1.2.2 Model-based Reinforcement Learning

In contrast to model-free methods, Model-based RL explicitly incorporates a model of the dynamics. This model can be known a priori or learned from experienced trajectories. Here, learning can take two distinct (complementary) roles: learning the model of the dynamics or directly learning a policy/(Q-)value function. These two roles are on different ends of our *problem model vs solution search* dimension. Moreover, having access to a model opens new possibilities to learn and define a policy efficiently. This leads to a diverse set of methods.

Simulator of transitions

A straightforward use of a dynamics model is to generate simulated experience. This synthetic data can then augment real experience to train model-free methods more efficiently, as seen in algorithms like Dyna-Q [Sutton, 1990] and its Deep Learning extension [Janner et al., 2019]. The model acts as a cheap source of "imagined" trajectories.

In our Figure 1.2, these algorithms replace the problem with a model to supply additional generated data.

Planning with differentiable or linear models

If the dynamics model is differentiable, gradient-based optimization techniques can be used for planning. Model Predictive control (MPC) methods can optimize a sequence of decisions by backpropagating gradients through the unrolled model predictions [Jacobson and Mayne, 1970, Kwakernaak and Sivan, 1972, Todorov and Li, 2005].

This applies whether the model is given or learned, and policy or value functions can be distilled from such planning procedures [Nguyen and Widrow, 1990, Schmidhuber, 1990, Jordan and Rumelhart, 2013, Deisenroth and Rasmussen, 2011, Grondman, 2015, Heess et al., 2015, Lowrey et al., 2018]. Assuming local linearity also enables a specific control procedure [Levine and Abbeel, 2014].

In that case, the guide in Figure 1.2 is composed of a differentiable model of the dynamics, a planning procedure leveraging that differentiability, and optionally a policy/value function that can be learned and included in the planning procedure.

Still assuming differentiability of the model, some algorithms learn to fit the gradient of the value function with respect to the state space, instead of only learning the value function output [Werbos, 1992, Miller et al., 1995, Fairbank and Alonso, 2012]. This approach leverages a stronger learning signal than algorithms purely learning the output of the value function.

Planning with sample-based and tree search methods

Planning can also be performed without assuming differentiability. Random shooting methods sample random action sequences and evaluate them using the model. Other techniques involve constructing a search tree over possible future states and actions, using the model to simulate outcomes.

AlphaGo and AlphaZero leverage policy and value functions to guide the tree search, and improve the policy and value functions by distilling the result of the search in a virtuous cycle [Silver et al., 2016, Anthony et al., 2017, Silver et al., 2018]. Similarly to Section 1.2.2, the guide in Figure 1.2 is composed of the model, a planning procedure, and a policy/value function learned from previous outputs.

Learning abstract models

Learning a full, high-fidelity model of complex environments can be difficult and potentially unnecessary. An alternative is to learn an abstract model that operates on an optimized representation of the states, to capture only the part of the dynamics relevant for the task. This offers several advantages: focusing the modeling effort, fast sampling of trajectories, potentially improving generalization by ignoring distractors, and enabling planning in a simpler (e.g., discrete or low-dimensional) abstract space.

To learn such abstract state spaces, several types of losses have been proposed and combined. Common losses are:

- **Reconstruction:** Forces the abstract state space to contain sufficient

information to reconstruct the original state or at least key parts of it (usually used with images).

- **Reward/value prediction:** The abstract state suffices to predict future rewards of the task or values.
- **Policy prediction:** The abstract state allows predicting the optimal action.
- **Transition consistency:** Predicted transitions in the abstract space should match the encoded representation of the actual next state.
- **Temporal contrastive :** Future abstract states should be distinguishable from unrelated states.

Planning can then occur within this learned abstract space using various methods, reflecting the ones already presented.

We provide here a sample list of diverse works representing the research performed in the direction of learning and planning with abstract models.

- Tabular RL in a discrete abstract state-space learned with a reconstruction loss and transition consistency [Corneil et al., 2018].
- Differentiable MPC or linear control in a continuous state-space learned with a reconstruction loss and transition consistency [Lenz et al., 2015, Watter et al., 2015].
- Architectures mixing continuous and discrete representation of the states to improve optimization, used with gradient-through-dynamics planning and as a simulator for model-free algorithms [Hafner et al., 2019a,b, 2020, 2023].
- Graph search in a learned discrete abstract state-space using a temporal contrastive and reconstruction loss [Kurutach et al., 2018].
- Learn a symbolic factorized abstract model using a reconstruction loss and transition consistency, allowing to plan with a symbolic method [Asai et al., 2022].
- Lookahead tree search on continuous abstract states learned with value and reward prediction [Oh et al., 2017], and with added transition consistency and entropy regularization of the representation [François-Lavet et al., 2019].

- Similarly, MuZero [Schrittwieser et al., 2020], uses a lookahead tree search and, like AlphaZero, distills the planning result in an improved policy and value function. Efficiency can be improved by using a temporal contrastive objective [Ye et al., 2021].

This diversity highlights the choices in designing abstract model algorithms based on the planning method leveraged and the requirements on the information contained in the abstract state representation. The feedback signals for learning vary accordingly to these requirements.

Dynamics prediction as an auxiliary loss

Even if a learned model is not directly used for planning, learning to predict future states can serve as an auxiliary self-supervised loss. This encourages the representation to capture meaningful temporal structure, potentially benefiting a primary model-free RL algorithm [Jaderberg et al., 2016].

Taking inspiration from contrastive self-supervised methods developed for vision, Self Predictive Representation forces the neural representation to be able to discriminate between possible future states, with greatly improved efficiency observed in some cases [Laskin et al., 2020, Schwarzer et al., 2020, 2023].

1.2.3 Goal-conditioned algorithms

Some algorithms learn to accomplish various goals, instead of only focusing on the task at hand. *Goal-conditioned policy* or *universal value functions* (UVFs) adapt classical RL approaches to this case.

In contrast to classical value functions focusing on predicting the task cumulative reward, UVFs predict state-to-state reachability [Sutton et al., 2011, Schaul et al., 2015]. An interesting advantage of UVFs is their ability to leverage more feedback for learning from sampled data than classical value functions. Even if the reward is sparse, UVFs can still get a rich learning signal from the observed state-to-state transitions [Andrychowicz et al., 2017].

UVFs can be learned with standard RL techniques using the Bellman equation, augmented with goal relabeling strategies like Hindsight Experience Replay (HER), which treats achieved states as goals to create successful examples even from failed trajectories [Andrychowicz et al., 2017].

Contrastive methods that learn to classify future states from random ones can also be used to learn UVFs [Eysenbach et al., 2022].

Learned UVFs or policies to reach arbitrary states can then be turned into a policy for the task of interest. These predictions can be turned into a policy by following actions that maximize the value to reach an a priori known goal-state.

Another way to turn them into a policy is to use them in a hierarchical framework where a master, higher-level policy iteratively calls the low-level policy with goal-states to reach [Levy et al., 2017, Nachum et al., 2018]. Letting the high-level policy learn in an environment with a lower temporal resolution.

An alternative line of work to UVFs leverages generative models, particularly diffusion models, to map the current state and a future desired goal-state to a complete plan (sequence of actions and states) reaching the goal-state [Janner et al., 2022, Ajay et al., 2022]. These algorithms represent another approach that can also leverage feedback from state-to-state transitions to learn a policy reaching a desired state.

1.2.4 Learning with a divide-and-conquer strategy in ATP

In the specific case of Automated Theorem Proving another Dynamic Programming approach than the Bellman equation on the task value function has been proposed.

In tactic-based theorem proving, a theorem is broken down into subgoals according to a chosen tactic, these subgoals are then further decomposed by tactics until elementary true propositions are reached, as in the Lean theorem prover [de Moura et al., 2015]. Several algorithms leveraging this divide-and-conquer decomposition have been proposed in the literature [Polu and Sutskever, 2020, Lample et al., 2022]. The algorithms here learn a value function that estimates the provability of each subgoal. The value function is also learned using Dynamic Programming.

This approach allows for leveraging the feedback from each of the constructed subgoals rather than only focusing on the main theorem. This represents inductive learning guided by a specific divide-and-conquer structure of the problem domain.

1.3 Learning (part of) the algorithm

In the essay *The bitter lesson* [Sutton, 2019], Sutton observed a historical trend in AI: approaches leveraging computation and general learning methods eventually outperform those relying heavily on hand-engineered knowl-

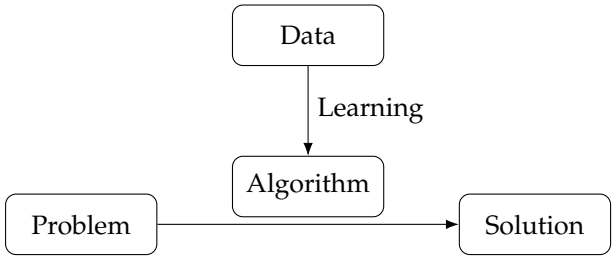

Fig. 1.3 Learning (part of) the algorithm that is then applied to solving a problem.

edge specific to the problem domain. Taken to an extreme, this observation suggests a direction in our search for general algorithms for problem-solving: instead of designing problem-solving algorithms ourselves, could we directly learn these algorithms?

In opposition to the previous sections, this section explores approaches where a significant part, if not the entirety, of the function mapping a problem to its solution is learned using a source of data or tasks. This approach is represented in Figure 1.3.

Learning the algorithm can be done from a dataset where the data might consist of (problem, answer) pairs, or richer (problem, reasoning/trace, answer) triplets, where intermediate steps provide stronger supervision. Alternatively, the data might also come from a set of interactive environments where solutions can be verified or optimized.

Interestingly, logical deductive procedures have been proposed to learn algorithms [Hutter, 2002, Schmidhuber, 2007]. However, this idea remains theoretical, as there is no practical implementation of it. This document does not present approaches based on logical deduction further and primarily focuses on Deep Learning approaches optimized via gradient descent, though other methods like evolutionary algorithms are also used.

We start our discussion by looking at how to represent such a learnable algorithm.

1.3.1 Expressive architectures

A prerequisite for learning algorithms is having model architectures capable of representing complex, potentially recursive or compositional, computations that go beyond the standard multilayer Perceptron. Researchers have proposed various architectures aimed at enhancing expressivity, to-

ward closing the distance to Turing equivalent computational models.

A first idea is to augment neural networks with pointers and/or an external memory accessible via differentiable read/write operations:

- Differentiable Neural Computers [Graves et al., 2014].
- Neural Turing Machines [Graves et al., 2016].
- Neural random-access machines [Kurach et al., 2015].
- Pointer Networks [Vinyals et al., 2015].
- Neural Programmer Interpreter [Reed and De Freitas, 2015].

Another approach to augment the expressivity is to perform parallel computation on all the elements in the memory using convolution-type computations.

- Neural GPU [Kaiser and Sutskever, 2015].
- Grid long short-term memory [Kalchbrenner et al., 2015].

A linked approach is to adapt Logic Programming in a differentiable Deep Learning architecture, e.g., Neural Logic Machines [Dong et al., 2019].

A particularly successful architecture is the Transformer that both performs computations in parallel and accesses information using a differentiable attention mechanism [Vaswani et al., 2017]. A recurrent version has also been proposed with the Universal Transformer [Dehghani et al., 2018]. Recently, Wang et al. [2025] propose improvements over the architecture and training to handle recursion depth.

These architectures have been trained on tasks like arithmetic, list sorting, sudoku, graph problems (e.g., shortest paths), and program execution, often using supervised learning on input-output pairs or intermediate execution traces. Reinforcement Learning has also been used to optimise some discrete computations within these architectures.

In parallel, Graph Neural Networks (GNNs) proved to be an efficient architecture for tasks where an underlying graph symmetry is known a priori [Battaglia et al., 2018]. GNNs apply neural message-passing operations iteratively over the nodes and edges of a graph, allowing them to learn functions that respect graph symmetries. They have been applied, for instance, to the SAT problem, attempting to learn a neural network that directly predicts satisfiability without explicit search [Selsam et al., 2018].

1.3.2 Foundation models

The architectures described above provide the capacity to represent algorithms. However, training them typically requires curated datasets specific to the target task (e.g., sorted lists, known satisfiable and unsatisfiable SAT instances). Creating such data for each conceivable problem class is impractical, and it fails to leverage potential synergies where similar reasoning or computational skills might be applicable.

An alternative paradigm is the development of *Foundation Models*. These are large-scale models, typically based on expressive architectures like the Transformer, trained on vast and diverse datasets encompassing text, code, images, and other modalities, often using self-supervised learning objectives [Radford et al., 2019]. These models can then be used to solve a wide array of downstream tasks, including complex problem-solving.

Direct problem-to-answer mapping

In the simplest application, a foundation model can be prompted with a problem description and asked to directly generate the answer. The model effectively acts as the learned algorithm mapping the problem to the answer.

Generating reasoning steps

Directly generating the answer can limit the model. With the Transformer architecture, it bounds to a constant the number of computational steps that the model can perform before its output. Thus, even if the problem is hard and might need more computations, the model is forced to directly output an answer.

We can alleviate this issue by letting or forcing the model to first write reasoning steps [Nye et al., 2021, Wei et al., 2022, Kojima et al., 2022].

Also, writing reasoning steps allows the model to learn from reasoning data to compute the answer. Reasoning data provides a richer learning signal than pure (problem, answer) pairs that force the learning algorithm to directly map end-to-end the problem and the answer. Data informing the intermediate steps leading to an answer breaks down the problem for the learning algorithm [Kim and Suzuki, 2024].

There exist multiple refinements over this approach. For example, the model can be used to judge the produced reasoning, allowing to sample several reasonings and choosing the best-valued [Shen et al., 2021, Weng et al., 2022]. More simply, another method is sampling several reasoning-answer pairs and choosing the answer that appears the most often [Wang

et al., 2022].

A key bottleneck to making this approach of generating reasonings work is the need for high-quality reasoning traces to tune the model. RL offers a way to partially bypass this problem by reinforcing generation strategies that lead to correct final outcomes (using outcome-based rewards) or valid intermediate steps (using process-based rewards) [Cobbe et al., 2021, Uesato et al., 2022, Lightman et al., 2023, Wang et al., 2023, Guo et al., 2025]. However, the learning algorithm must have a mechanism to compute the outcome-based or process-based rewards. Outcome-based rewards can be computed if the final answer is verifiable, and process-based rewards are typically computed from another "judging" model. This approach allows to learn better reasoning strategies, provided the final answer or reasoning can be reliably checked.

1.3.3 Learning to map problems to solvers and tools

The ambition to learn the entire problem-solving algorithm from scratch faces significant issues concerning robustness and generalization. A hybrid approach involves learning how to leverage existing, reliable, human-designed algorithms or tools. In this paradigm, the learned component maps the problem to a hard-coded solver/tool, and then interprets the result. This approach attempts to combine the strength of learning the algorithm with the guarantees of established solvers.

The hard-coded solver or tool can vary: code interpreter, symbolic solver, logical reasoning engine, planner, and continuous optimizers.

Mapping to a differentiable solver

Mathematical Optimization and logical deduction algorithms offer general, powerful solvers to a variety of problems. Differentiable versions of these solvers have been integrated as modules within neural networks. By ensuring that gradients can flow through the solver, the entire system can be trained end-to-end to map input problems to solver parameters and interpret the returned solution.

- OptNet introduces a differentiable quadratic programming solver layer [Amos and Kolter, 2017].
- SATNet developed a differentiable SAT solver approximation [Wang et al., 2019].
- Gradient descent itself can be posed as a differentiable Deep Learning

layer using gradient-through-gradient (second-order differentiation) [Du et al., 2022].

- Adapting logical deduction to make it differentiable to allow end-to-end differentiable proving [Rocktäschel and Riedel, 2017].
- Integrating a probabilistic logic programming language with Deep Learning using a differentiable deduction procedure [Manhaeve et al., 2018].

All these works require and make possible differentiation through the optimizer/deduction engine to train the whole procedure end-to-end.

Mapping to a symbolic interpreter or solver

Another approach is to directly use a symbolic interpreter or symbolic solver in the Deep Learning architecture. However, those are not differentiable, and other methods have to be employed to train the model.

One can use an RL method to train non-differentiable components. Several architectures are trained with REINFORCE to write a program in a Domain Specific Language [Liang et al., 2016, Mao et al., 2019].

Foundation models can be trained to call tools through text. Toolformer synthetically creates a dataset to learn to call tools [Schick et al., 2023].

Tensors treated as logical objects in a solver

Logic Tensor Networks encode knowledge into a fuzzy logic, then use an approximate procedure for deduction [Serafini and Garcez, 2016].

1.3.4 Meta-Reinforcement Learning

When the problem class itself involves sequential decision-making in an environment (i.e., finding an optimal policy for a new robot), the concept of "learning the algorithm" can take the form of meta-Reinforcement Learning (meta-RL).

If the environment is initially partially unknown, the goal of meta-RL is to learn an algorithm that can quickly adapt when placed in new, previously unseen (but related) environments. The learned algorithm must implement a strategy for exploring the new environment in a limited number of interactions and then exploiting the gathered information efficiently.

Even when the environment dynamics is known in advance, finding an optimal policy can still require planning procedures that can be optimized with meta-RL.

The meta-problem is a partially observable RL problem

The generated algorithm in Figure 1.3 takes a sequence of trajectories and a current state as input and outputs an action maximizing long-term cumulative rewards. The meta-problem of learning this algorithm can thus be approached with RL where the policy is the algorithm itself. One peculiarity of this RL problem is that it is partially observable since the dynamics is unknown and can only be indirectly inferred from observations.

RL algorithms can handle this partial observability. A direct approach is to represent the algorithm/policy in a trained neural network where the input is a sequence of all past trajectories composed of a sequence of state, action, reward, next state, ..., up to the current state and the output is an action. Such neural networks can be trained using model-free RL across a set of tasks [Wang et al., 2016, Duan et al., 2016].

Here, the meta-RL algorithm learns a neural network representing a complete RL algorithm. Exploration and exploitation are both optimized during the meta-learning stage.

While this approach lets a lot of freedom to learn the RL algorithm, it can also compromise robustness. Another approach is to start with a classical (robust) RL algorithm, then meta-learn only a part of it, where meta-learning is often performed using either a zeroth-order optimizer or second-order differentiation.

This approach of meta-learning only a part of the algorithm is similar to the above literature integrating human-designed solvers in Deep Learning architectures. This allows to enforce useful inductive bias in the learned algorithm.

For instance, the initialization parameters of the policy or value function can be learned using gradient-through-gradient [Finn et al., 2017] or with a black-box optimization algorithm [Fernando et al., 2018]. The rest of the hand-designed RL algorithm stays the same.

Another line of work meta-learns the loss minimized by the RL algorithm, using evolutionary optimization [Houthooft et al., 2018] or gradient-through-gradient [Kirsch et al., 2019, Bechtle et al., 2021]. A substitute to the Bellman equation can also be meta-learned [Oh et al., 2020].

Meta-learning other parts of the RL algorithm has been explored, such as hyperparameters, we refer to Hospedales et al. [2021] for a review.

Learning to plan

Meta-RL can be used to learn efficient planning strategies when the dynamics is fixed. Researchers have developed architectures that incorporate

planning-like primitives and learn how to use them effectively:

- Value Iteration Networks includes a differentiable approximation of a Dynamic Programming algorithm that can be embedded within a larger network [Tamar et al., 2016].
- Including lookahead search methods over a tree of possibilities [Farquhar et al., 2017, Guez et al., 2018].
- Universal Planning Networks learn a differentiable model of the dynamics leveraged by a differentiable gradient-based planner [Srinivas et al., 2018].

Alternatively, expressive architectures trained with model-free RL on tasks requiring planning might implicitly learn planning computations without explicit planning modules being hardcoded [Guez et al., 2019, Bush et al., 2025].

1.4 Outline

Learning is a flexible tool applicable in widely different areas of a search procedure. Giving a rich and diverse set of algorithmic ideas presented in this chapter.

This thesis analyzes some of the advantages and disadvantages of these approaches compared to one another.

Chapter 2 analyzes the theoretical advantage that expressive Deep Learning architectures of Section 1.3.1 can have over the classical stacking of linear operators and non-linear activation functions. From a statistical point of view, is it more efficient to learn with these architectures? The results of this Chapter are a reproduction of Pinon et al. [2023b].

Chapter 3 studies empirically the advantage of expressive architectures and model-based RL methods with local planning for meta-RL in contrast to the model-free RL methods of Section 1.3.4. This Chapter is an extended version of the publication Pinon et al. [2023a].

Chapter 4 and 5 unroll theoretical analyses contrasting the limitations and advantages of various algorithms leveraging learning in a loop to solve a problem like pictured in Figure 1.2: model-free RL methods of Section 1.2.1; model-based RL methods for simulation or using local planning with a tree search procedure in Section 1.2.2; universal value functions in Section 1.2.3; and resolution-based SAT solvers of Section 1.1.

More precisely, in these Chapters we construct example problems that are efficiently solvable by some algorithmic approaches but not others. Thus implying a separation in efficiency between methods. In the pre-existing literature, Sun et al. [2019] proves an exponential advantage for model-free RL methods in contrast to a model-based RL algorithm leveraging an a priori known goal-state.

In Chapter 4, we build on their result to extend the limitation on some model-based methods that we term *black-box dynamics RL methods*. Moreover, this limitation is in contrast with other model-based methods that do not leverage an a priori known goal-state.

In Chapter 5, we build example problems proving a limitation on some RL algorithms using value function and universal value function learning. In contrast, the problems are efficiently solved by a resolution-based SAT-solver.

Chapter 4 is a reproduction of the published results Pinon et al. [2025] and Chapter 5 iterates on the presented work Pinon et al. [2024].

The analyses of Chapters 4 and 5 reveal strong limitations on some approaches, and to conclude in Chapter 6, we return to the variety of ideas presented in this chapter to discuss potential solutions and point to open problems.

2

PAC-learning gains of Turing machines over circuits and neural networks

There is an interest in the literature for the development of methods learning using computationally universal programming languages. Section 1.3.1 presents several works creating Deep Learning (DL) architectures able to express recursion and compositionality. These architectures narrow the gap between Deep Learning and Turing machines in their ability to describe computable functions. This chapter studies the gain in sample efficiency that can be attained in principle from these methods in comparison to classical Artificial Neural Networks (ANNs) under simplifying algorithmic assumptions.

We use Turing machines in the analysis as a representative for the expressive DL architectures, and Boolean circuits as an equivalent to vanilla DL (ANNs).

This theoretical investigation is similar to comparisons that have been made between ANNs and other classical Machine Learning algorithms to explain the experimental success of the former.

We present those two branches of Machine Learning research. The branch that develops methods to make inductive inferences in computa-

tionally universal programming languages; and the branch that compares encodings for hypotheses based on the induced minimal description sizes to express functions.

Inductive inference in computationally universal programming languages An important aspect of DL research is the development of new architectures. Some of these architectures can efficiently address particular problems, like Convolutional Neural Network (CNN) for Computer Vision, Recurrent Neural Networks (RNN) for natural language processing, and Graph Neural Networks (GNN) that can be adapted to a wide variety of applications [Battaglia et al., 2018].

A common critical point in the development of these architectures is the exploitation of prior knowledge about the task at hand. This exploitation is done by imposing on the model some factorized representation. The structure of the representation enforces the a priori known symmetries in the underlying function to learn. This leads to improvement in the sample efficiency, see Xu et al. [2020] for a paper taking this perspective for GNN with associated PAC-learning results.

For CNN, RNN, and GNN, the factorization's structure is classically fixed a priori with respect to prior information about the symmetries in the function to learn. Alternatively, the structure could be learned from the data.

It is thus natural to invest efforts in the construction of learning algorithms with the flexibility to define and use potential abstract structures found in the data. In this line of work, we mention two approaches: *Differentiable programming* which consists in learning with DL architectures close to Turing-complete systems such as the papers mentioned in Section 1.3.1; and learning in non-differentiable programs written in universal languages for which combinatorial optimization methods such as genetic programming must be used, see Koza and Poli [2005]. This latter approach is usually referred to as *Inductive Programming* or *Program synthesis* from examples, see Kitzelmann [2009] and Gulwani et al. [2017].

Our work is a theoretical investigation of the potential sample efficiency gains that could be observed from Inductive Programming or learning with these new expressive DL architectures in comparison to classical Artificial Neural Networks.

Models: the necessary sizes to represent functions This last decade DL algorithms allowed to tackle problems that had been impossible to solve

until then. These successes opened the question: Why DL is more efficient on these complex problems than other classical Machine Learning algorithms, such as shallow neural networks?

A theoretical answer is that depth in Artificial Neural Networks (ANNs) allows us to efficiently encode some classes of functions. More precisely there exists a sequence of functions for which low-depth ANNs need an exponential number of neurons to approximate it and, in comparison, higher-depth ANNs only need a polynomial number of neurons to achieve the same approximation. Examples of references on the subject are Telgarsky [2015], Liang and Srikant [2016], Eldan and Shamir [2016], and, from the perspective of Boolean circuits, Rossman et al. [2015].

The expressive power of depth in ANNs to efficiently represent functions in terms of the sizes of the hypotheses has also been studied in comparison with Support Vector Machines in Bengio et al. [2007] and with Decision Trees in Bengio et al. [2010].

Our work is inspired by these comparisons between models. Similarly, we show a separation in terms of the sizes of the hypotheses that are necessary to fit functions. More precisely, we study the advantage that Turing machines have over circuits and neural networks. Pushing this observation to PAC-learning claims, we study potential gains on the number of samples that are necessary to learn Boolean functions by Turing machines.

Objectives and formal choices Given the motivations for learning in computationally universal programming languages, we investigate the sample efficiency gains that we can hope from this approach relative to classical ANNs.

We use Turing machines to represent models based on Turing-complete systems, such as the new expressive DL architectures for example. This choice of the computational model is not determinant since Turing machines can serve as a proxy to study other Turing-complete systems.

We compare Turing machines with Boolean circuits and classical ANNs. By ANNs we mean the simplest form of DL with non-linear activation functions composed on top of linear transformations and without any repetition of the weights such as in CNN or RNN. To represent ANNs we choose a computationally feasible model, the model is not based on real numbers but is discrete and finite.

The *minimum description length* (MDL) principle consists in choosing the hypothesis with the shortest description length while being consistent with the data. It is a classical formalization of Occam's razor principle. The

MDL principle will allow us to translate our choices of models —which are ways to express the hypotheses— into learning algorithms. It will give us a general and effective way to compare inductive biases posed by different representations such as ANNs and Turing machines for example. We refer to Grünwald [2007] for an introduction to the MDL principle and to Li et al. [2019] for an introduction to Kolmogorov complexity, applying the MDL with universally computable languages. The book Li et al. [2019] can also serve as an introduction to understand interpreters and how they can define a language to describe computable functions.

To assess the performance of the models/learning algorithms, we follow the classical PAC-learning framework. However, we do not explore the computational efficiency question of finding the shortest hypothesis; rather we focus on the sample efficiency. In other words, throughout the paper, we neglect the computational resources needed to find the minimum description length hypothesis consistent with the data; but set our attention on the size of the dataset that is necessary to find a hypothesis that is Probably Approximately Correct. We note that this choice creates a gap between our theory and what can be observed in practice when using imperfect but practical optimizers for DL models.

We refer the interested reader to Shalev-Shwartz and Ben-David [2014] for an introduction to ML theory including PAC-learning.

Outline In Section 2, we provide some background on PAC-learning, interpreters, and the MDL principle.

Then, from it, we introduce in Section 3 the critical metric at the heart of our objectives: the *sample efficiency gains* of a model over another.

In Section 4, we prove bounds on the sample efficiency gains that circuits have over (polynomial-time) Turing machines and conversely. In particular, we show that the sample efficiency gains of polynomial-time Turing machines over circuits are at least linear in the input-size of the function to learn. Whether they are superlinear or not is an open question. We connect this question to different open problems from Computational Complexity.

2.1 Background

2.1.1 Our learning problem

We want to learn an unknown function f belonging to a *hypothesis class*

$$H^n = \{f : \mathcal{B}^n \to \mathcal{B}\}, \tag{2.1}$$

where $\mathcal{B} = \{0, 1\}$, and thus H^n is finite.

For some fixed integer $n > 0$, a *learning problem* is determined by a boolean function $f \in H^n$ and a probability measure $\mathcal{P}$ on $\mathcal{B}^n$, $\mathcal{P} \in \Delta(\mathcal{B}^n)$.

Definition 1. *A* learning problem *m-sample dataset is a random variable defined as $\left[x_j, f(x_j)\right]_{j=1}^{m}$ where the x_j are sampled independently and according to the learning problem probability measure $\mathcal{P}$.*

To *solve* a learning problem is to find an approximation $\hat{f}$ of f from a realization of the learning problem m-sample dataset, for some natural m.

We formalize the notion of approximation with the classical accuracy.

Definition 2. *The* accuracy *of a function $\hat{f} \in H^n$ with respect to learning problem $f \in H^n$, $\mathcal{P} \in \Delta(\mathcal{B}^n)$ is*

$$acc_f^{\mathcal{P}}(\hat{f}) \stackrel{\text{def}}{=} \Pr_{x \sim \mathcal{P}}\left[\hat{f}(x) = f(x)\right]. \tag{2.2}$$

Definition 3. *A* learning algorithm *is a function that given the realization of a learning problem m-samples dataset, for some learning problem $f \in H^n, \mathcal{P} \in \Delta(\mathcal{B}^n)$, outputs a function $\hat{f} \in H^n$, for any $n, m \in \mathbb{N}^+$.*

We do not specify a representation for the output's function since, as said in the introduction, our work focuses on the sample efficiency of the learning algorithm and not its computational complexity.

The learning algorithm sample efficiency performance will be assessed by PAC-learning claims.

Definition 4. *For any $\epsilon \in (0, 1/2)$, $\delta \in (0, 1)$, an algorithm $\mathcal{A}$ has an (ϵ, δ)-PAC-learning performance with an m-sample dataset on learning problem $(f, \mathcal{P})$ if for all $m' \geq m$*

$$\Pr_{x \sim \mathcal{P}^{m'}}\left[acc_f^{\mathcal{P}}\left(\mathcal{A}(\left[x_j, f(x_j)\right]_{j=1}^{m'})\right) \geq 1 - \epsilon\right] \geq 1 - \delta. \tag{2.3}$$

2.1.2 Interpreter

We now introduce the concept of interpreters, which will allow us to define the notion of description-length of a hypothesis given a model.

Definition 5. *An* interpreter φ^T *is a Turing machine computing a two-arguments partial computable binary-valued function* $\varphi : \mathcal{B}^* \times \mathcal{B}^* \to \mathcal{B} \cup \{\bot\}$. *Where* $\mathcal{B}^*$ *is the set of finite length binary strings,* $\mathcal{B}^* = \cup_{i=0,1,...}\mathcal{B}^i$, *and* $\bot$ *is the symbol representing non-halting executions.*

In the rest of the paper, we will identify the Turing machine implementation with its computed function by dropping the T in φ^T with exceptions where the distinction is useful.

In our developments, the first argument will correspond to the code / program / hypothesis and the second argument will correspond to the input of the function to learn.

We make Definition 5 concrete by presenting some interpreters, see Section 1.3 in the appendices for complete descriptions:

- **Universal Turing Machine $\mathcal{U}$:**

$$\mathcal{U}([\text{binary encoding of a two-inputs Turing machine } \mathcal{T}, \text{first input}], \text{second input}) = \mathcal{T}(\text{first input}, \text{second input}).$$

  Note that we will define Turing machines with, only, binary-valued outputs. The encoding of Turing machines and the Universal Turing Machine are formally defined in the appendices, Definitions 45 and 46 respectively.

- **Polynomial-time Universal Turing Machine $\mathcal{U}^c$:** A Universal Turing machine with a limited computation time in $\mathcal{O}(n^c)$ steps, where n is the size of the input and $c \in \mathbb{N}^+$.

  This interpreter will allow us to make claims using hypotheses with reasonable running time.

- **Boolean circuit interpreter $\mathcal{C}$:** A Boolean circuit is a directed acyclic graph where each node is either an input node or a gate. Each input node is associated with an input value, and gates are associated with unary or binary logical operators (OR, AND, and NOT). There is one node with no child (sink), this node is the output node of the circuit.

The topology of the graph is consistent with the nodes' logical association: input nodes' are not the child of any other node, gates with a binary operator have two parent nodes, and gates with a unary logical operator have a unique parent node.

The output of a circuit on a binary input is the result of the output node after the application of the logical operations associated with the gates. In this computation, the input nodes naturally take their values from the input.

Boolean circuits are encoded as binary strings by first noting the input-size, then the number of nodes in the circuit (the circuit's size), and finally by describing the nodes one by one (associated input or logical operation, and the potential parents).

With this encoding, the description-length of a Boolean circuit of size S is in $O(S \log S)$.

Again the application of this interpreter is

$$\mathcal{C}(\text{binary representation of a Boolean circuit}, \text{input}) = \text{output of the circuit.}$$

- **ANNs interpreter:** We define an ANN as a directed acyclic graph whose nodes correspond to either an input or a floating-point operator. Similarly to Boolean circuits, each input node is associated with one variable of the binary input. The floating-point operators are taken from a predefined arbitrary set. This set can contain binary or unary operators such as $+, -, \times, /, \max(.,.), \exp(.)$; it also contains 0-ary/constant operators: the floating-point values themselves.

  Again similarly to Boolean circuits, there is a unique output node that has no children.

  The values of the nodes, given an input, are determined by the recursive application of the input nodes' association to input variables or of the corresponding floating-point operators until a value for the output node is obtained.

  There is a linear relationship between the description-length of a function with the Boolean circuits' or the ANNs' interpreter, see Proposition 57 in the appendices. We use this link to present all the results with the interpreter of Boolean circuits, $\mathcal{C}$, while exactly the same results will hold for ANNs.

- **Support Vector Machines, Binary Decision Trees, CNN, RNN, GNN :** An interpreter can be defined for each of these classes. Note that in some cases the length of the binary representation of some functions can be drastically reduced or increased by using these specialized interpreters.

  The existence of these interpreters is noted to point out that the formal background presented applies to more than just Turing machines, Boolean circuits, and ANNs. None of these interpreters will be discussed further in this work.

2.1.3 The Minimum description length (MDL) principle

We define how the application of the MDL principle with an interpreter gives a learning algorithm.

Notation. *Let $|h|$ be the length of a binary string h.*
We denote, for any interpeter φ, any n and any function $f \in H^n$:

$$|f|_\varphi \stackrel{\text{def}}{=} \min_{h \in \mathcal{B}^*} |h| \ \ s.t.\ \varphi(h, x) = f(x),\ \forall x \in \mathcal{B}^n.$$

By convention, if the problem is infeasible the value will be $+\infty$.

Definition 6. MDL principle with an interpreter φ: *MDL^φ. From an interpreter φ, one can create the following learning algorithm, MDL^φ.*
Input: the realization of a learning problem m-samples dataset.
Select the output function, $\hat{f}$, corresponding to a minimal-description-length program consistent with the dataset

$$\begin{aligned} h^* \in \arg\min_{h \in \mathcal{B}^*} \quad & |h| \\ \textit{subject to} \quad & \varphi(h, x_j) = f(x_j),\ j \in \{1, \dots m\}; \\ & \varphi(h, x) \neq \perp,\ \forall x \in \mathcal{B}^n. \end{aligned} \tag{2.4}$$

The output function is thus $\hat{f} = \varphi(h^*, .)$.

Thus, the choice of an interpreter completely determines the learning algorithm with the MDL principle. The inductive bias is fixed toward low complexity (short to express) hypotheses.

Let us remark that, any computable time-limit on the execution of an interpreter φ will ensure that MDL^φ is computable. In this paper, we will

always assume that such an arbitrary sufficiently large time-limit is imposed, say $2^{2^{2^n}}$ for example.

The following PAC-learning bound will be the main proposition used through the paper to go from descriptions' length constraints to PAC-learning affirmations. It is based on a classical argument in the PAC-learning literature to show uniform convergence of the accuracy for finite hypothesis classes, see Blumer et al. [1987]. The argument is simply adapted to our specific MDL framework.

Proposition 7 (Description-length PAC-guarantee)**.** *There exists constants $a_1, a_2 > 0$ such that the following holds.*

For any interpreter φ and associated learning algorithm MDL$^\varphi$, any $(f, \mathcal{P})$ learning problem and any PAC-learning parameters $\epsilon \in (0, 1/2)$, $\delta \in (0,1)$, MDL$^\varphi$ has an (ϵ, δ)-PAC-learning performance with an m-sample dataset on the learning problem, where

$$m = \frac{a_1}{\epsilon}\left[\log\frac{1}{\delta} + |f|_\varphi + a_2\right]. \tag{2.5}$$

For $\varphi = \mathcal{U}$ or $\varphi = \mathcal{C}$ under mild conditions on the size of the circuits considered, the guarantee of Proposition 7 is tight up to a fixed factor for any configurations (ϵ, δ), any input-size n and any sufficiently large maximal description-length $|f|_\varphi$ on some learning problems $(f \in H^n, \mathcal{P} \in \Delta(\mathcal{B}^n))$. This is shown using a VC-dimension based argument in the appendices, we refer to Proposition 63 and its associated section.

2.2 Sample efficiency gains

We propose here a PAC-learning criterion to compare the sample efficiency of two arbitrary MDL-based learning algorithms. This criterion will then be applied to MDL with circuits and MDL with Turing machines.

We want to analyze the largest gap, in terms of necessary samples to get some learning performance, that can exist between two learning algorithms. We quantify the number of samples needed by a learning algorithm in order to solve a learning problem as the following.

Definition 8. *The minimal number of samples needed to get an (ϵ, δ)-PAC-learning performance for a learning algorithm MDL$^\varphi$ on a learning problem $(f, \mathcal{P})$*

is

$$m_{\varphi}^{\epsilon,\delta}(f,\mathcal{P}) \stackrel{\text{def}}{=} \min\{\{m \in \mathbb{N}^+ \,|\, MDL^{\varphi} \text{ has an } (\epsilon,\delta)\text{-PAC-learning performance with an } m\text{-sample dataset on learning problem } (f,\mathcal{P})\} \cup \{+\infty\}\}. \quad (2.6)$$

The PAC-learning guarantee given in Proposition 7 provides an upper-bound on the quantity defined in Equation 2.6.

Our main goal in this work is to study variations of the sample efficiency according to n the input-size of the underlying function to learn. We will thus parametrize our criterion with this quantity.

Moreover, we impose a practical restriction on the description-length of the function to learn.

Definition 9. *Given two interpreters φ and ψ, any $\epsilon \in (0, 1/2)$, $\delta \in (0,1)$, and any $n, d \in \mathbb{N}^+$, the* sample efficiency gain *of MDL^{φ} over MDL^{ψ} is*

$$G_{\varphi\to\psi}^{d}(\epsilon,\delta,n) \stackrel{\text{def}}{=} \sup_{f\in H^n, \mathcal{P}\in\Delta(\mathcal{B}^n)} \frac{m_{\psi}^{\epsilon,\delta}(f,\mathcal{P})}{m_{\varphi}^{\epsilon,\delta}(f,\mathcal{P})} \quad \text{subject to} \quad |f|_{\varphi} \leq n^d. \quad (2.7)$$

The restriction $|f|_{\varphi} \leq n^d$ naturally applies on the interpreter for which we try to study a potential sample efficiency advantage, φ; the restriction ensures from the PAC-learning guarantee presented in Proposition 7 an (ϵ,δ)-PAC-learning performance with a number of samples polynomial in n.

We now define an auxiliary metric for two reasons. First, it is a lower bound on the sample efficiency gains and will be used as such in some theorems' proofs. Second, on some questions we only obtained partial results that are expressed through this metric, instead of sample efficiency gains.

Definition 10. *Given two interpreters φ and ψ, any $\epsilon \in (0, 1/2)$, $\delta \in (0,1)$, any $n, d \in \mathbb{N}^+$, and $a_1, a_2 > 0$ fixed in Proposition 7,*

$$\tilde{G}_{\varphi\to\psi}^{d}(\epsilon,\delta,n) \stackrel{\text{def}}{=} \sup_{f\in H^n, \mathcal{P}\in\Delta(\mathcal{B}^n)} \frac{m_{\psi}^{\epsilon,\delta}(f,\mathcal{P})}{\frac{a_1}{\epsilon}(\log\frac{1}{\delta} + |f|_{\varphi} + a_2)} \quad \text{subject to} \quad |f|_{\varphi} \leq n^d. \quad (2.8)$$

This value can be interpreted as the best sample efficiency gains that can be obtained from MDL^{φ} over MDL^{ψ} while being provable from the

guarantee given in Proposition 7.

Remark. *We defined G —the sample efficiency gain— as our metric of interest. However, in the literature —for example, the literature on depth-separation in ANNs— a usually assumed criterion of comparison is the necessary sizes of the hypotheses to fit functions, since these can usually be translated into PAC-learning guarantees.*

We refer the reader to Section 1.2 in the appendices for a complete development based on the gains in description-length, $\sup_{f \in H^n} \frac{|f|_\psi}{|f|_\varphi}$ *instead of G; and, to Section 1.4 for links between constraints on the description-length of an hypothesis class and its VC-dimension for Turing machines and circuits.*

2.3 Main results: comparison of Turing machines and circuits

Now that the formal background and the metric of interest are defined, we study the sample efficiency gains by learning with Turing machines or Boolean circuits interpreters under the minimum description length principle. All the results are given for Boolean circuits but also hold for ANNs. In other words, the Boolean circuits' interpreter $\mathcal{C}$ can be substituted for the ANNs' interpreter, given in Definition 56, in all the provided theorems.

2.3.1 Sample efficiency gains of circuits over Turing machines

Before studying the sample efficiency gains of Turing machines over Boolean circuits, we present a partial result in the direction of showing that the sample efficiency gains of circuits over Turing machines are below a constant.

The following theorem shows that on any particular learning problem a PAC-guarantee obtained through Proposition 7 for learning with circuits will imply a similar PAC-guarantee for learning with Turing machines.

Theorem 11. *There exists a constant* $q \in \mathbb{R}^+$ *such that for all* $\epsilon \in (0, 1/2)$, $\delta \in (0,1)$ *and* $n, d \in \mathbb{N}^+$,

$$\tilde{G}^d_{\mathcal{C} \to \mathcal{U}}(\epsilon, \delta, n) \leq q. \tag{2.9}$$

The rest of the paper analyzes the converse question: can we prove an advantage of Turing-complete systems over circuits for learning in terms of sample efficiency gains?

2.3.2 Sample efficiency gains of Turing machines over circuits

In the next theorem, we show that we can construct a sequence of learning problems such that learning with Turing-complete languages becomes more and more advantageous in terms of sample efficiency over learning with Boolean circuits. Moreover, the advantage grows exponentially in the input-size of the function to learn.

Theorem 12. *For all* $\epsilon \in (0, 1/2), \delta \in (0,1), d \in \mathbb{N}^+$ *we have*

$$G^d_{\mathcal{U}\to\mathcal{C}}(\epsilon, \delta, n) \in \Omega(2^n/n). \tag{2.10}$$

The theorem shows that the potential advantage of learning with Turing machines over circuits can quickly become significant.

In the proof of Theorem 12, an explicit sequence of learning problems with an advantage for Turing machines is given, the functions to learn are defined by a Turing machine that enumerates on the functions in H^n and the Boolean circuits. The functions to be learned are thus hard to compute.

This can be seen as a practical limitation on the scope of this theorem. We will now use polynomial computational limits on our interpreter. Our formalism will use the interpreter of polynomial-time Turing machines — $\mathcal{U}^c$— to this effect.

2.3.3 Limits on the sample efficiency gains of polynomial-time Turing machine over circuits

In this new computationaly constrained setting, we prove a bound on the sample efficiency gains that can be shown from the length-based PAC-guarantee of Proposition 7.

Theorem 13. *For all* $\epsilon \in (0, 1/2)$, $\delta \in (0,1)$, $c, d \in \mathbb{N}^+$ *we have*

$$\tilde{G}^d_{\mathcal{U}^c\to\mathcal{C}}(\epsilon, \delta, n) \in O(n^c \log^2 n). \tag{2.11}$$

The proof of Theorem 13 is based on a result of Pippenger and Fischer [1979], their result states that circuits can compute functions as fast as multi-tape Turing machines. More precisely, we use a refinement by Schnorr [1976] that takes into account the size of the involved Turing machine.

2.3.4 Sample efficiency gains of polynomial-time Turing machines over circuits are at least linear in the input-size

We show a positive result for learning with polynomial-time Turing machines, the sample efficiency gains grow at least (nearly) linearly in the input-size of the function to learn.

Theorem 14. *For all $\epsilon \in (0, 1/2)$, $\delta \in (0, 1)$, $1 < c, d \in \mathbb{N}^+$ and all $\gamma > 0$ we have*

$$G^d_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) \in \Omega(n^{1-\gamma}). \tag{2.12}$$

2.3.5 Are sample efficiency gains of polynomial-time Turing machines over circuits superlinear in the input-size?

The Theorems 13 and 14 open the question of whether the growth of the sample efficiency gains is actually a superlinear polynomial in the input-size of the function to learn.

As we will show, this question connects with open problems in Computational Complexity.

If gains are superlinear

The first open problem with which we make a connection is the existence of a problem in **P** for which superlinear sized circuits are necessary.

This problem is of importance in Computational Complexity. There are links between the collapse at the first and second level of the polynomial hierarchy and the computability of languages in **NP** by polynomial-sized families of Boolean circuits, see Karp and Lipton [1980].

However, despite years of efforts, the maximal size-lower-bound known on Boolean circuits for a language in **NP** is linear, see Iwama and Morizumi [2002], Arora and Barak [2009], Jukna [2012].

The Theorem 15 shows that proving that the sample efficiency gains are actually superlinear through the PAC-guarantee offered by Proposition 7 solves this frontier. It solves the frontier by proving the existence of a problem in **P**, and thus **NP**, with superlinear circuit complexity.

Theorem 15. *If there exists $\epsilon \in (0, 1/2)$, $\delta \in (0, 1)$, $c, d \in \mathbb{N}^+$ and some $\gamma > 0$ such that*

$$\tilde{G}^d_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) \notin O(n^{1+\gamma}) \tag{2.13}$$

*then there exists a language in **P** not computable by any sequence of Boolean circuits whose sizes are in $O(n^{1+\tau})$ for some $\tau > 0$.*

If gains are not superlinear

On the other hand, if the sample efficiency gains are not superlinear in the input-size of the function to learn for polynomial-time Turing machines over circuits then $\mathbf{P} \neq \mathbf{NP}$.

Theorem 16. *If for all* $\epsilon \in (0, 1/2)$, $\delta \in (0, 1)$, $c, d \in \mathbb{N}^+$ *and all* $\gamma > 0$

$$G^{d}_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) \in O(n^{1+\gamma}) \tag{2.14}$$

then $\boldsymbol{P} \neq \boldsymbol{NP}$.

Gains are superlinear under circuit lower bounds

The contraposite of the last result yields superlinear gains in the case $\mathbf{P} = \mathbf{NP}$. However, $\mathbf{P} = \mathbf{NP}$ is not a common assumption in computer science. We propose an alternative in the following theorem. Let $\mathbf{E}$ be the set a language decidable with time-complexity $O(2^{O(n)})$ by Turing machines.

Theorem 17. *If there exists* $f \in \boldsymbol{E}$ *and* $\iota > 0$ *such that the Boolean circuits computing* f_n *are at least of size* $2^{\iota n}$*; then for any* $\gamma > 0$ *there exists* $\epsilon \in (0, 1/2)$, $\delta \in (0, 1)$, $c, d \in \mathbb{N}^+$ *such that we have*

$$G^{d}_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) \in \Omega(n^{1+\gamma}). \tag{2.15}$$

We note that the circuit lower-bounds of the assumption are the same as the ones arising in derandomization research, and the theorem is based on a worst-case to average-case hardness result [Arora and Barak, 2009].

2.4 Conclusion

In this work, we analyzed the sample efficiency gains of Turing machines over Boolean circuits and classical neural networks under the minimum description length principle in the PAC-learning framework. The Turing machines served as a proxy in the analysis for other Turing-complete systems, such as the expressive Deep Learning architectures cited in the introduction, or programs (written in computationally universal languages) that can be learned from Inductive Programming techniques.

We showed that learning with expressive models such as Turing machines can yield sample efficiency gains that are exponential in the input-size of the function to learn. Learning with polynomial-time Turing machines can also yield PAC-learning profits in comparison to learning with

circuits. The sample efficiency gains grow at least linearly in the input-size of the function to learn.

Whether they are superlinear or not is an open problem. One of our main results showed that if these sample efficiency gains are not superlinear in the input-size then $\mathbf{P} \neq \mathbf{NP}$. Another of our main contributions demonstrated that if it is possible to prove superlinear gains using a classical PAC-learning uniform convergence argument, then there exists a problem in $\mathbf{P}$ with superlinear circuit complexity. Making progress on this question thus requires answering long-standing open problems in Computational Complexity.

Nevertheless, we proved that under a circuit lower bound assumption used in derandomization research, the gains are superlinear in the input-size. This result provides a theoretical argument for the use and development of expressive DL architectures presented in Section 1.3.1.

A parallel investigation of the gains in terms of description-length to express Boolean functions is also an output of this research, given in 1.2.

This paper also leaves some questions open. Some of our results offer bounds on $\tilde{G}$, thus providing information on the sample efficiency gains that are provable using the PAC-guarantee of Proposition 7. Improving these results by using G instead would render them independent of any particular PAC-guarantee.

An extension of the analysis to classically used DL architectures such as Convolutional Neural Networks and Recurrent Neural Networks would also broaden our insight on the potential advantage of using Turing-complete systems as models for learning.

3

A model-based approach to meta-Reinforcement Learning: Transformers and tree search

Deep Learning methods have been successfully applied to various problems such as in image processing, natural language processing, and games [Schmidhuber, 2015, Goodfellow et al., 2016, Li, 2018]. However, these solutions usually require many samples. The field of meta-learning addresses this issue. The meta-learning paradigm supposes access to data from other tasks in relation to the task of interest. A meta-learning method can then take advantage of this supplementary data to learn an efficient learning algorithm for the task of interest.

There are several areas of research close to meta-learning: transfer learning, domain adaptation/generalization, continual learning, multi-task learning, AutoML, ...These differ from meta-learning by having various restrictions on the type of algorithm used or the source data. We refer to Hospedales et al. [2021] for a review of the differences between these approaches, especially in contrast to meta-learning.

3 | A model-based approach to meta-Reinforcement Learning: Transformers and tree search

This chapter focuses on meta-Reinforcement Learning (meta-RL), where tasks involve sequential decision making in an environment.

Recently proposed in Wang et al. [2021], the Alchemy benchmark aims to test and understand proposed meta-RL algorithms. The benchmark poses a fixed distribution over a space of tasks. Each task in that space has a dynamics defined by a set of causal links between "potions" use and their effects on "stones". This dynamics is not directly observable by the agent. To maximize its reward, the agent must understand this dynamics through experimentation, then exploit that knowledge over an episode. Two versions of the benchmark have been proposed, a 3D visual version and a symbolic version. We focus on the symbolic version.

As presented in Section 1.3.4, one approach to meta-RL is to cast the problem as a partially observable Markov Decision Process (POMDP), where a latent space represents the hidden dynamics. Model-free RL methods that support partial observability can then be used for meta-RL [Wang et al., 2016, Duan et al., 2016]. The authors of Alchemy tested state-of-the-art model-free RL methods on it. Their experimentation revealed a failure of these model-free RL methods to learn a policy that efficiently explores for information and then exploits it, both on the 3D visual and the symbolic case [Wang et al., 2021].

We investigate the use of a model-based algorithm with online planning on the symbolic version of Alchemy and show significant improvements. This result shows both:

- The capability of Deep Learning methods, in particular the Transformer architecture, to fit complex dynamics in environments where model-free RL methods fail. These complex dynamics emerge naturally in meta-RL problems where the latent space is a critical part.

- The strength of online planning algorithms in challenging environments, such as those that arise in meta-RL where the reward is delayed between the gain in information from exploration and its exploitation.

3.1 Related work

This section expands on the literature already presented in Section 1.3.4 toward algorithms closer to this chapter. We survey model-free RL applied to meta-RL problems literature since it is the solution studied by Wang

et al. [2021]. We then describe model-based approaches applied in meta-RL.

Model-free RL for meta-RL

As said, one approach to meta-RL is to pose the construction of the efficient agent as an RL problem with partial observability. Then this POMDP can be solved with classical deep RL methods. A usual way in deep RL to support partial observability is to make the past observations input to the neural networks representing the policy or (Q-)value function. Several architectures that support these sequential inputs have been applied: long short-term memory (LSTM) and gated recurrent units (GRU) [Wang et al., 2016, Duan et al., 2016], memory-augmented neural networks by [Santoro et al., 2016], a mix of temporal convolution and attention [Mishra et al., 2017], and a variation of Transformers [Parisotto et al., 2020].

Several works use a model-free RL algorithm and complement it with auxiliary losses that force an inner representation of the studied system dynamics [Wayne et al., 2018, Gelada et al., 2019, Zintgraf et al., 2019]. They use modeling losses to have self-supervision and create potentially relevant abstractions, but do not use their ability to predict the dynamics of the system directly for planning. Similarly, a task description can be used as a target to construct useful abstractions [Humplik et al., 2019]. It has been proposed that model-free methods in meta-RL implicitly learn to do such task inference [Alver and Precup, 2021].

Model-based RL for meta-RL

In the context of meta-RL models have been used in Hiraoka et al. [2021] to train a policy from simulated trajectories while providing theoretical arguments for their method.

In Perez et al. [2020], they use probabilistic inference on latent variables representing the hidden dynamics, and then use the cross-entropy method for planning. The same idea is applied in Sæmundsson et al. [2018] using Gaussian Processes for learning and Model Predictive Control for planning.

Section 1.3.4 mentions that the initialization parameters of the neural network representing the policy or value function can be trained to maximize performance on a dataset of tasks [Finn et al., 2017]. This is also true for the initialization parameters of a neural network representing a model of the dynamics [Nagabandi et al., 2018a,b, Mendonca et al., 2020]. With this training, the model is trained to quickly adapt to the dynamics of the

current task with gradient steps. We note that the model is adapted on-the-fly to fit the task of interest.

Still another way to learn models in the meta-RL literature is to fix a component of the model. For example, a fixed state-to-state dynamics but with a non-fixed reward function. However, this can only work for problems where the set of tasks respects strong constraints on their dynamics. An example of such a work is done in Byravan et al. [2020].

Our solution

Our work presents a model-based algorithm that learns a model of the POMDP dynamics with a Transformer and then incorporates this model in a tree search planner.

3.2 Methods

We define the general problem we address 3.2.1, and describe the symbolic Alchemy benchmark 3.2.2. We then explain the construction of our model-based agent. How we learn a neural network model of the dynamics 3.2.3, and what online planner we use on top of this model 3.2.4.

3.2.1 Problem formulation

We define a partially observable Markov decision process (POMDP) as a tuple $(\mathcal{S}, \mathcal{A}, P, \Omega, O, H)$ where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, and P is an operator defining the initial probability distribution over states with $P_0(s_0)$ and the conditional transition probability distribution over states and rewards $P_{\text{dyn}}(s_{t+1}, r_{t+1}|s_t, a_t)$ where $s_{t+1}, s_t \in \mathcal{S}$, $a_t \in \mathcal{A}$ and $r_{t+1} \in \mathbb{R}$. The set Ω defines the possible observations, and O is a conditional probability distribution $O(o_t|s_t)$ describing the link between the state and the observations given to the agent. The horizon $H \in \mathbb{N}$ gives the number of steps in one episode.

In our meta-learning problem, at the start of each episode, latent variables are sampled. These hidden variables define the dynamics until the end of the episode. This fits the definition of a POMDP where the hidden variables are part of the states but not directly observed.

Our goal is to define a policy that acts to maximize the expected sum of rewards over an episode.

3.2.2 Symbolic Alchemy

Symbolic Alchemy, as defined in Wang et al. [2021], links an unknown "chemistry" to the dynamics of an environment which consists of a variety of stones and potions. Where potions can be used to change stones and render them more valuable. Here, the chemistry determines the effect of applying a potion on a stone, and a new chemistry is sampled each time an episode starts. For example, applying the red potion on a small purple stone might make it large in one episode and increase its value, but might change its color in another and decrease its value. The goal is to explore and use to our advantage acquired knowledge about this chemistry. We define next what constitutes a state and the unrolling of one episode.

In symbolic Alchemy, three types of objects determine a state: the chemistry; the stones, their respective 3 visual features and associated reward; and the potions with their respective colors. The features of the stones and potions are directly observable, but the chemistry is not.

The agent can do several actions. He can apply a potion on a stone, this can change its perceptual features and reward. He can also collect the reward associated with a stone (which can only be done once for each stone). The hidden chemistry determines the effects of potions on stones and the possible features of stones.

An episode unrolls as follows: a chemistry is sampled among 167,424 others at the start and is kept constant during the whole episode; then a sequence of 10 trials follows. Each trial is further decomposed into: sampling a set of stones and potions; then 20 time steps where actions can be taken to understand the chemistry through experiments, generating high-value stones by using potions, and collecting rewards associated with the stones. The decomposition of one episode is given in Figure 3.1.

We describe an example episode with a potential agent. At the start, a hidden chemistry is silently sampled. This single chemistry dictates the rules of transformation for the entire episode, which consists of 10 trials. The only information the agent can know at this point is that the chemistry is constrained to follow some rules, like Red and Green potions must have opposite effects on the same stone feature if they have an effect, but the agent does not know which feature.

In our case the following chemistry has been sampled:

Potion effects on the features of stones:

- Red and Green potions affect Size.

- Yellow and Orange potions affect Color.
- Pink and Turquoise potions affect Shape.

The stone with the combination of Large, Purple, and Triangular features is the most valuable, and is worth +15 points. All other combinations are worth less.

With the chemistry set, the first of ten trials begins.

The environment for trial 1 is generated, three stones and twelve potions appear:

- A Small, Blue, Triangular stone;
- A Large, Blue, Triangular stone ;
- A Small, Purple, Round stone;
- and 12 potions, with 2 of each color (Red, Green, Yellow, Orange, Pink, Turquoise).

The agent has 20 timesteps to act. Not knowing the chemistry, its initial actions are experiments to uncover the rules.

- Timestep 1: The agent chooses to act on the first stone. It applies a Red potion to the Small, Blue, Triangular stone. The potion disappears, and the stone transforms into a Large, Blue, Triangular stone. The agent now knows that the Red potion changes a stone from Small to Large. It can infer that the Green potion likely does the opposite.
- Timestep 4: The agent continues its experiments. It applies a Yellow potion to the same stone, which is now Large, Blue, and Triangular. The stone's color shifts, becoming a Large, Purple, Triangular stone. The agent learns that Yellow potions can change a stone's color from Blue to Purple. The stone has now a +15 reward associated to it.
- Timestep 8: The agent performs an exploitation action. The reward of the +15 value stone is collected, and this stone is removed from the trial.

At the start of trial 2, the board is reset with a new set of 3 stones and 12 potions. However, the chemistry remains the same. The agent retains the knowledge from the first trial 1. It can now plan its actions to reach high-reward stones efficiently with the knowledge acquired in the first trial.

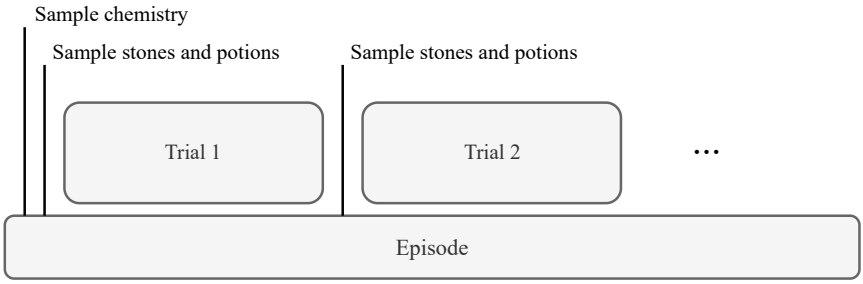

Fig. 3.1 Decomposition of one episode of symbolic Alchemy: a chemistry is sampled for the episode, then trials are interleaved with resamples of new potions and stones.

This shows the importance of doing the trials sequentially, one after another. The knowledge gained in the first trials can be used to be more efficient in the subsequent ones.

3.2.3 **Learning a model**

We describe how we train and leverage a model of the dynamics to compute actions given a history of environment observations.

Data

We sample a dataset of trajectories prior to any learning and planning. We simply use a uniform policy over the action space to generate the trajectories. We define a supervised learning problem from this data.

Since the actions are chosen randomly, the data is far from describing an optimal policy. Optimized actions will be determined by our planning procedure.

While this method is sufficient to learn an accurate model in the case of symbolic Alchemy, other environments might need interleaving data generation and model learning to get a sufficient coverage of the state and action spaces.

Model architecture

The architecture is presented in Figure 3.2. Our model is based on the Transformer using causal attention for its main block [Vaswani et al., 2017]. It receives as input a sequence of observations, rewards, and actions, and it outputs a prediction for the next observation and reward at each step.

We use the Transformer architecture to fit the dynamics for its capacity to learn complex autoregressive functions, as demonstrated in natural language processing.

In the prediction head for the next observation, a GRU allows our neural network to model probability distributions over the next observation that are non-factorizable over the coordinates. In other words, our model is not constrained to only represent distributions $p(o_{t+1})$ that must factorize in $\Pi_i p_i(o^i_{t+1})$ where i is the ith coordinate of the observation. A more detailed view of this part of the model is available in Figure B.1 in Appendix 2.2.

In addition to the GRU, our prediction head also includes a linear component making a direct link from the output of the Transformer to each coordinate of the next observation. We show in our results that both the linear and GRU parts are necessary to have an accurate model of the dynamics of symbolic Alchemy.

To train our model, we use teacher forcing: at training time to predict the ith coordinate of the observation at timestep $t+1$, we use the true observations and rewards up to the timestep t, and also, the ground truth up to the $i-1$th coordinate of the observation at timestep $t+1$. See Figure B.1 in Appendix 2.2 for a detailed representation of teacher forcing.

We encode the real-valued observations and rewards as categorical variables, both for the input and output. Discretizing the inputs allows us to have efficient cross-entropy loss and helps to get consistent trajectory simulation over longer time horizons. We note that discretization is not unusual in the RL literature, for example, MuZero discretizes the prediction of the reward [Schrittwieser et al., 2020]. The encoding, E, is a one-hot transformation that takes as input a d-dimensional real-valued vector and returns a $d \times n$-tensor where n is the number of categories.

A full description of the hyperparameters that we use for our architecture and its training for the Alchemy benchmark can be found in the Appendix 2.1.

3.2.4 Tree search with stochastic transitions

We define the model-based online planning algorithm that leverages, for its simulations, the learned model defined in the previous section.

We construct our online planner on top of the tree search planner presented in Silver et al. [2016], and more specifically, the version defined by Schrittwieser et al. [2020]. The tree search algorithm grows iteratively a

tree where each branch represents a potential sequence of events starting at the current state. The original algorithm supposes full observability, deterministic transitions, a policy and a value function. We adapt it to support partial observability and stochastic transitions.

We use a common transformation that casts POMDPs as equivalent MDPs where states correspond to the complete history of observations up to that point, we refer to Baxter and Bartlett [2000] for an example. The transformation ensures that a policy defined on the MDP corresponds to a behavior with the same performance on the original POMDP. We note that even if the original POMDP had a deterministic dynamics, the resulting MDP can be stochastic in general.

To support stochastic dynamics, we adapt the tree search algorithm by adding chance nodes as children of decision nodes. At chance nodes, we branch different paths representing different stochastic realizations of the system dynamics. For each chance node, we use a fixed number of samples from which stochastic branches are created. We merge identical samples to reduce the tree breadth. When we merge samples, we use weights to not introduce any bias.

To compensate for our lack of policy and value function, we use defaults: a uniform probability distribution over actions; and a zero-everywhere value function is taken.

The algorithm grows the tree by prioritizing more promising branches. At each node, it uses Q-value estimates and also uncertainty estimates on these values to quantify how interesting an action might be.

The algorithm cycles through 3 steps to grow the three:

- *Selection* Starting at the root node, the branches of decision nodes are taken depending on the Q-value estimates, the uncertainty on these estimates, and the size of the subtree at that node. At chance nodes, we sample a branch with a probability proportional to its weight. We stop when we have no child for the action to take.
- *Expansion* Where we block in the selection step, we create a chance node and sample its children.
- *Backup* With the new rewards obtained at the expansion step, we update the Q-value estimates on the decision nodes taken in the selection step.

At the end of this procedure, an action is sampled from a distribution proportional to the number of times each action has been selected in the

search.

See Algorithm 6 in the appendices for a more detailed description of the tree search and Figure 3.3 for a partial representation of a tree.

We note that since the number of samples by chance node is constant, a limited number of possible paths for the system dynamics are considered. Thus, in general, the Q-values estimates in the tree will not asymptotically converge to their true values with the size of the tree. However, in practice, the size of the tree is bounded and the estimates will have non-zero variance in any case. The number of samples by chance node is a hyperparameter that can be optimized to compromise between the tree width and the lack of precision of these estimates.

3.3 Results

We compare our method against the results obtained in Wang et al. [2021]. They tested V-MPO a state-of-the-art model-free RL method [Song et al., 2019], applied with a Transformer architecture improved for RL [Parisotto et al., 2020]. This method iteratively optimizes a policy and a value function using on-policy data. The optimization steps involve carefully designed losses to stabilize the complete process while retaining performance.

The authors of Wang et al. [2021] also implemented an ideal observer, which is a Bayes-optimal agent that has the best performance achievable on the benchmark. This agent is not directly comparable to the others, since its implementation relies on hand-coded human prior knowledge of the problem. For further insights into the agent performances, they also implemented a simple Random Heuristic which does not exploit information efficiently.

We sample $1e6$ episodes to construct our dataset of trajectories from which we learn our model. In the paper Wang et al. [2021], the authors report that $1e9$ episodes were used to train the V-MPO agent on symbolic Alchemy.

See Table 3.1 for the comparison, we test our method with several magnitudes of computational resources going into planning. We optimize the hyperparameters of the tree-search for the case of 1.250 expansions by step in the environment, see Appendix 2.1 for the hyperparameters.

The comparison reveals that our method surpasses the model-free RL method V-MPO if given sufficient computational resources for online planning. It also shows the importance of planning on this benchmark, the performance increases significantly with more allocation of resources into

planning. However, the marginal gains diminish too fast to reach the optimal performance in our tests.

We note that under two conditions the score is guaranteed to reach an optimal value with more and more expansions. First, the learned model of the dynamics is accurate. Second, the number of branches considered at chance nodes grows with the number of expansions.

Table 3.1 Comparison with the V-MPO RL method [Song et al., 2019] and the baselines given in Wang et al. [2021]. We provide the scores with the standard error. The number of expansions refers to the expansion operation defined in the description of the tree search.

Agent	Score on symbolic Alchemy
Ours (# expansions 100)	79.3 ± 1.6
Ours (# expansions 500)	161.8 ± 3.9
Ours (# expansions 1.250)	207.1 ± 3.5
Ours (# expansions 2.500)	220.5 ± 3.2
Ours (# expansions 5.000)	229.5 ± 3.9
Ours (# expansions 10.000)	251.5 ± 4.5
V-MPO	155.4 ± 1.6
Ideal Observer	284.4 ± 1.6
Random Heuristic	145.7 ± 1.5

3.3.1 Architecture tests

We conduct an ablation study on our model's architecture to identify the essential elements to its success. We do three tests: replacing the Transformer part with a recurrent architecture, a GRU; suppressing the GRU in the prediction head of the next observation; and suppressing the linear layer in the prediction of the next observation. We test the capacity of the architecture to fit the dynamics of the symbolic Alchemy environment. We use this metric instead of the direct score because it better measures the potential of the learned model to construct an optimal policy. An error in the learned model induces a limit on the performance of the behavior for any online or offline planning method that we would build with the learned model. Results are given in Table 3.2.

If we replace the Transformer part with a GRU, the prediction performance drops. In symbolic Alchemy, the entropy in the prediction has two

Table 3.2 All other hyperparameters being equal to the original architecture, see 2.1. With an exception when we remove the final linear layer, where to increase performance, we decrease the learning rate to $2e-5$ and we double the number of hidden units in the GRU to 64. When we replace the Transformer with a GRU, the GRU has 3 layers each of dimension 512. The entropy loss is computed on a held-out fraction of the generated dataset described in Section 3.2.3.

Model architecture	entropy loss on validation set
original architecture	1.31
replace Transformer with GRU	1.37
without the GRU head	1.61
without the final linear layer	2.25

sources: the uncertainty that comes from predicting the next state of a stone after applying a potion; and predicting the uncertainty at each trial start when the set of stones and potion is reinitialized. To further understand the difference in entropy loss, we measure the part of the entropy loss coming from the former which correspond to the uncertainty on the current chemistry. We obtain 0.066 with the original model and 0.11 with the GRU as a main component. So part of the structure in the symbolic Alchemy problem is harder to learn for the GRU architecture, where in comparison, the Transformer architecture successfully grasps the latent dynamics.

When we remove the GRU head in the prediction of the next observation, we decrease the model capacity since we forbid the output to represent a probability distribution of independent variables (one variable by feature of the observation). The gap in predictive performance with the original architecture shows the importance of modeling the interdependence between the different variables in an observation.

We observe that without the final linear layer to help predict the next observation, optimizing the model is much harder. We suppose that this is not due to a lesser capacity of the model, since we increased the capacity of the GRU head to compensate for this effect. So we suspect that backtracking only through the GRU steps must be harder for the optimizer.

3.4 Discussion

We demonstrate that a model-based approach with online planning can substantially surpass the performance of a state-of-the-art model-free method on the symbolic Alchemy meta-RL benchmark. We determine that this success relies on several factors. First the architecture to train a model, a Transformer is more performant in comparison to a recurrent architecture. Additionaly, the architecture must be able to model non-factorizable probability distributions over the next observation coordinates. In our experiments, we used a GRU in the prediction head to achieve this property. We also added in parallel a direct linear layer to improve optimization. Besides the neural architecture, an online planning algorithm allowed to use many simulated steps is also necessary to achieve a non-trivial performance on the benchmark.

Our work shows that while most of the research effort has focused on model-free methods in meta-RL [Hospedales et al., 2020], learning to learn might need explicit modeling and then planning. Also, the use of performant auto-regressive architecture can be critical to model the complex relationship between histories of observations and beliefs upon the system transitions.

Improving the behavior The online planner that we use can take as input a learned policy and value function to improve its performance. Moreover, a performant policy and value function can be learned from the online planner. Due to limited computational resources, we did not investigate this solution. We believe that this method would significantly improve the behavior and decrease the computational burden when we apply the policy, see Hamrick et al. [2020] for an empirical investigation of this method. This improvement would not increase the number of ground-truth samples required, but would significantly increase the computational requirements to train the agent.

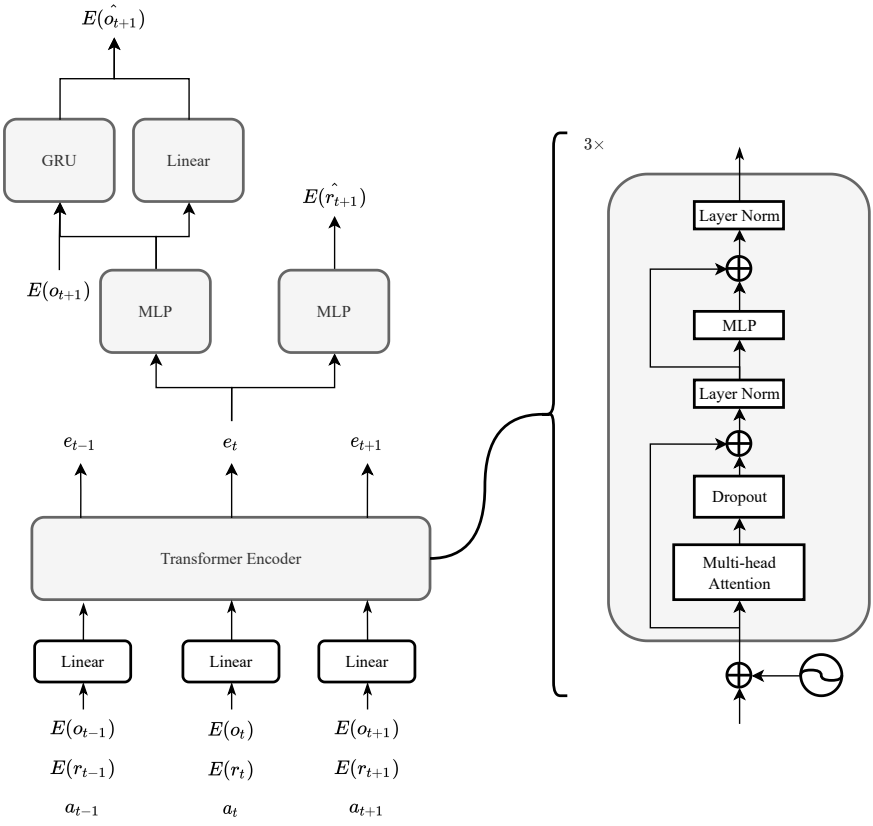

Fig. 3.2 A neural network architecture to fit trajectories. The sequence of observations and rewards is first transformed into categorical variables. Then the sequence is passed through a linear layer along with the sequence of actions. The resulting sequence is fed to a Transformer with three layers, causal attention, and a sinusoidal positional encoding. Encodings, e_t, from the Transformer are then used independently to predict the next elements in the sequences. For the rewards, an MLP is used. For the observations, first an MLP, then a GRU, plus a linear layer, are combined. We note that the next observation is also in input for the GRU since we use teacher forcing.

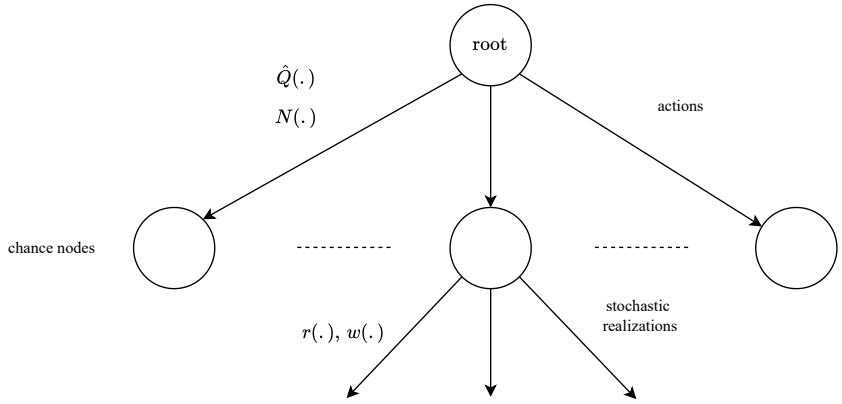

Fig. 3.3 Tree search algorithm with support for stochastic transitions.

4

A limitation on black-box dynamics approaches to Reinforcement Learning

We presented many algorithms leveraging learning for problem-solving in our introductory Chapter 1. Our goal in the next two chapters is to understand when and why some methods perform better than others from a computational point of view. Explaining such differences is central to guiding the design of new efficient algorithms.

A first step in understanding the differences between methods is to abstract them into classes. Two of the main classes of methods are model-based and model-free RL methods. Model-based methods are algorithms that leverage a known or learned model of the environment dynamics [Mordatch and Hamrick, 2020]. In contrast, model-free methods, such as Q-learning and Policy Optimization algorithms [Sutton and Barto, 2018], do not use such a model.

Although the distinction between model-free and model-based classes is commonly accepted [Sutton and Barto, 2018], there are no agreed-upon general formal definitions of these classes. Authors resort to proposing their own definitions [Sun et al., 2019] or to proving their results on classical algorithms representative of these classes [Tu and Recht, 2019].

Several works have studied the relative performance of these classes from a theoretical point of view, and it is cited as an open problem in the survey Levine et al. [2020]. For tabular Markov Decision Process, Tu and Recht [2019] review the existing literature that studied the model-free or model-based methods. They find no clear conclusion in favor of one class over another. In the specific problem family of the Linear Quadratic Regulator, Tu and Recht [2019] demonstrate a polynomial separation result in favor of a model-based method.

To our knowledge, Sun et al. [2019] provide the only result with a gap in the efficiency that is exponential in a relevant parameter of the problems. They offer a definition for the class of model-free methods and then present a family of problems intractable for model-free methods but easy for a specific model-based method.

In this work, we define the class of black-box dynamics RL methods containing model-free RL (Q-learning, Policy Optimization, ...) and several RL methods considered to be model-based such as AlphaZero [Silver et al., 2017]. These methods do not leverage (learned) knowledge of the dynamics or except possibly by generating simulations of transitions. Our paper unravels explicitly a limitation on this class of black-box dynamics methods and reviews potential ideas in the literature to circumvent this limitation.

We note that here by the term "black-box" we refer to the fact that the model is used only for its capacity to produce an output (next state) given an input (current state and action). Thus, any knowledge of the details of a potential computational model of the dynamics is not leveraged, even if this computational model has no physical interpretation. In other words, the state-to-state dynamics is opaque to the algorithm.

To establish our result, we formalize the class of black-box dynamics methods in Definition 19. The definition aims to achieve two objectives: (1) to capture a large class of methods encompassing model-free and some model-based RL methods, and (2) to allow the proof of a limitation on these methods. Our definition relies on the fact that black-box dynamics methods can be formulated to interact with problems through an *interface*, a novel concept we introduce (pictured in Figure 4.1). An RL method will fit our definition if the method can still be implemented when our interface is placed between the method and the problem. This interface allows us to formulate and check conditions on the information that flows from the problems to the black-box dynamics methods. In particular, states are not directly fully observable to the method through the interface but only

partial information that allows to get (Q-)value and policy functions evaluations.

Next, we introduce a family of problems provably intractable for black-box dynamics methods, one of these problems is represented in Figure 4.2. However, this family of problems is easy to solve and can be provably efficiently solved by a toy method that identifies a rewarding state and learns to reach it. Moreover, we perform numerical experiments and show that a planning algorithm leveraging a learned model of the dynamics also efficiently solves the problems in the family.

Our work differs from the result of Sun et al. [2019] in two ways. First, we broaden the class of methods with a limitation to some model-based methods such as AlphaZero. We obtain this result by leveraging our concept of interface to formalize this larger class of methods.

Our second contribution with respect to the result of Sun et al. [2019] is that our family of problems can be efficiently solved without strong specific priors about it. In contrast, in their result, the success of the model-based method crucially relies on an a priori known state to reach. They encode this knowledge into the model-based method to its advantage. In general, such knowledge cannot be assumed to be known in practice and impairs the practical applicability and generality of their claim.

In summary, our work identifies a failure-case for a broad range of general-purpose RL methods.

This chapter is structured as follows. Section 4.1 defines the notation and formalizes the problems we address. Section 4.2 characterizes the black-box dynamics RL methods with our definition and linked interface. Section 4.3 states our main theoretical result. Section 4.4 demonstrates numerically the performance of several RL methods. Finally, Section 4.5 provides a summary of our findings.

Appendix 3.2 presents an alternative formalization choice with its implications, and Appendix 3.3 discusses the differences with the work of Sun et al. [2019] in more detail.

4.1 Preliminaries

Notation We use $\Delta(\Omega)$ to denote the set of probability measures over a sample space Ω with an implicitly associated σ-algebra. We define the function $\mathbf{1}(.)$ to output 1 if the condition in its argument is respected, else 0. For a set A, we note $A^* = \cup_{i \in \mathbb{N}} A^i$ the set of all finite sequences of elements in A.

In this paper an *RL problem* is a finite horizon Markov Decision Process (MDP), which is defined by a horizon $H \in \mathbb{N}$, a state space $\mathcal{S}$, an action space $\mathcal{A}$ and an operator P, which determines an initial state distribution with $P_0(s_0) \in \Delta(\mathcal{S})$ and a dynamics with $P_{\text{dyn}}(r, s' | s, a) : \mathcal{S} \times \mathcal{A} \to \Delta(\mathbb{R} \times \mathcal{S})$, where r is the reward and s' is the next state.

Throughout the paper, the rewards are in the interval $[0, 1]$, the set of actions is binary $\mathcal{A} = \{0, 1\}$, and the sets of states $\mathcal{S}$ are of the form $\{(t, x) \in \{0, \dots, H\} \times \mathbb{R}^n\}$ for some $n \in \mathbb{N}$, where t is the time step and x a feature vector. Initial states have $t = 0$, and t is incremented at each transition by P_{dyn}. When the time step H is reached, we say that the state is final and the trajectory ends. Our formalization is a particular case of the general framework of Contextual Decision Processes [Jiang et al., 2017].

We note $(s_0, a_0, r_0, s_1, a_1, \dots, s_H) \sim P^\pi$ a trajectory sampled according to the operator P and a policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$. We use π^U to denote the policy which outputs a uniform distribution over actions.

The objective is to find a policy that maximizes the expected cumulative rewards $\mathbb{E}_{(s_0, a_0, r_0, \dots, s_H) \sim P^\pi}[\sum_{t=0}^{H-1} r_t]$. An *RL method* is an algorithm that outputs a policy given access to P. We note that the policy returned by the RL method might not be optimal. The method *solves* the MDP if it outputs an optimal policy.

4.2 Formalization of black-box dynamics RL methods

In this section, we define the class of black-box dynamics RL methods (Definition 19) and then state a limitation on their efficiency in the next section.

The formalism We defined in the last section an RL method as an algorithm that outputs a policy given P (the operator describing the dynamics of an RL problem), but we did not formally specify how P is accessed by the method. In the literature, it is common to assume that P can be used to sample trajectories from a chosen policy. Here instead, black-box dynamics RL methods access P through an *interface*. An algorithm for RL problems is considered black-box dynamics if it can be implemented in an RL method interacting only with the interface (Definition 19).

The interface, defined in Algorithm 2, is a set of algorithms (oracles) that can be called to get information about the RL problem. The interface is thus placed between the RL problem and the RL method as pictured in Figure 4.1. This setup allows us to formulate and check conditions on the

information that can flow from the RL problem to the black-box dynamics RL method.

To build intuition on the information that we impose the interface to filter, consider deep Q-learning methods as an example. In these methods, the Bellman equation might be enforced by minimizing an empirical average of $[Q(s,a) - (r + \max_{a'} \bar{Q}(s',a'))]^2$ for some sampled transitions (s,a,r,s') and where $\bar{Q}$ is an old estimate of the Q-values. We observe that in the Bellman equation, the state s' is not directly needed but only its evaluation through $\bar{Q}$.

More generally, across a broad range of RL methods relying on the Bellman equation, given a transition (s,a,r,s'), the state s' is only used through evaluations of learned ML models such as value or Q-value functions.

With the help of the interface, our Definition 19 formulates an abstract condition that these indirect accesses to s' satisfy in black-box dynamics methods. More precisely, when a transition s,a,r,s' is sampled from P, the interface does not directly return s' to the calling method, but $\mathcal{F}(s',D)$ where D is a dataset and $\mathcal{F}$ is a function chosen by the calling algorithm. The dataset D is a list of partial information (s,a,r,f) about past sampled transitions (s,a,r,s') where f is the evaluation of s' by $\mathcal{F}$ with an old version of the dataset.

The function $\mathcal{F}$ is chosen by the RL method and can be selected to be a Machine Learning (ML) algorithm to learn and use (Q-)value and/or policy functions. In that case, D can play the role of a replay buffer, with it, $\mathcal{F}(s,D)$ can learn a neural network according to the Bellman equation and evaluate the state s to extract Q-values. This allows the method to choose an action based on estimated Q-values.

We ask $\mathcal{F}$ to respect a constraint on its output without obstructing the implementation of our methods with the interface. Specifically, we demand that $\mathcal{F}$ is invariant to shufflings of the coordinates in the states. This condition is naturally satisfied by ML algorithms since, without a priori, a learning process is expected to treat the features symmetrically. This condition allows us to model the reliance of methods on rewards to extract information from states. This is leveraged in our Theorem 20 to prove that some states are indistinguishable from the point of view of a black-box dynamics RL method.

To implement an RL method with the interface, the RL method should also be able to ask new transitions to generate trajectories or to do local planning by testing several possible actions from a given state. To permit

these with the interface without revealing the state s', the interface returns a number $\bar{s'}$ that the RL method can use to ask for new transitions. In particular, we choose $\bar{s'}$ to be equal to the iteration number at which the state is sampled since the start. For example, at the start, the RL method can ask a new initial state; the interface numbers that state *0* and provide this number to the RL method; the RL method can ask for new transitions to the interface that leads to successive states numbered *1*, *2*, . . . ,*17*; at that point, the RL method could ask the interface to sample a new transition from the *12*th state with action $a = 1$; the interface then responds to the method with the obtained reward, *18*($= \bar{s'}$), and $\mathcal{F}(s', D)$ where the dataset D contains information about the *17*th first sampled transitions. Simultaneously, the dataset is augmented with the transition from the *12*th to the *18*th state for future (improved) state evaluations with $\mathcal{F}$.

In some algorithms, transitions are simulated with a learned model. It is not generally possible to learn a model of the dynamics with the interface under our Definition 19. However, it is possible to perfectly simulate the dynamics using calls to the interface that interacts directly with P (describing the true underlying dynamics).

Black-box dynamics RL methods Our Definition 19 below encompasses a large set of classical RL methods that includes model-free RL approaches such as Policy Optimization and Q-learning. The definition also includes some model-based RL methods that only use a model of the dynamics to generate new transitions such as AlphaZero [Silver et al., 2017]. These methods treat a model of the dynamics as an input-output black-box function and do not leverage its internal computations.

On the other hand, there are several examples of algorithmic schemes that our definition does not allow. We defer to Section 6.2 a discussion on RL methods from the literature that are excluded from our definition but describe shortly here some examples.

Learning a model of the dynamics is not allowed, except if it is only used to sample transitions since in that case these transitions can directly be generated with the interface. Thus, the definition does not allow taking a gradient through a learned model or looking inside a learned model of the dynamics.

Also, learning a model of the inverse dynamics is not allowed, such as learning to predict the action to take given the current state and a wanted future state.

Algorithm 1 Encoder and decoder of states.

last_idx ← -1
list ←[]
function ENCODE_STATE(s)
    list ←list.push(s)
    $\bar{s}$ ←last_idx+1
    output $\bar{s}$
function DECODE_POINTER($\bar{s}$)
    output s = list[$\bar{s}$]

Definitions As explained above, a black-box dynamics RL method interacts with an interface linked to an RL problem, Algorithm 2 pictured in Figure 4.1. The interface is a set of algorithms with an internal state maintained between calls made by the RL method. Through the interface, the RL method can ask for new transitions, the interface then returns two objects for any sampled state s'. The first object is a number $\bar{s'}$ computed by an encoder-decoder described below. The interface also returns a real vector f: the evaluation of a constrained function $\mathcal{F}$ on s' and an internal dataset D. This second mode of state access allows the RL methods to train and evaluate ML models on states, such as learning value functions using neural networks.

At each call for a new transition, the interface computes $\bar{s'}$ and $f = \mathcal{F}(s', D)$ then returns these values to the RL method. Simultaneously, the interface builds incrementally its internal dataset D with this information. Similarly to RL methods when ML algorithms are used, the ML models are trained from the data, and the data is generated with the help of the ML models. This cyclical dependency is represented by the curved arrows in Figure 4.1.

The state reference $\bar{s'}$ is computed by an encoder-decoder of states defined in Algorithm 1. The encoder and decoder translate the states to numbers, and numbers to states respectively. The algorithm numbers the encountered states in the order of their visit. The PUSH operation puts the state at the end of the list, and thus the encoder-decoder potentially associates several numbers with one state if it is visited several times. This numbering ensures that no information about the state is revealed to the calling method. We use $\bar{s}$ to denote a computed number corresponding to some state s.

Our Definition 19 requires that the function $\mathcal{F}$ respects a special sym-

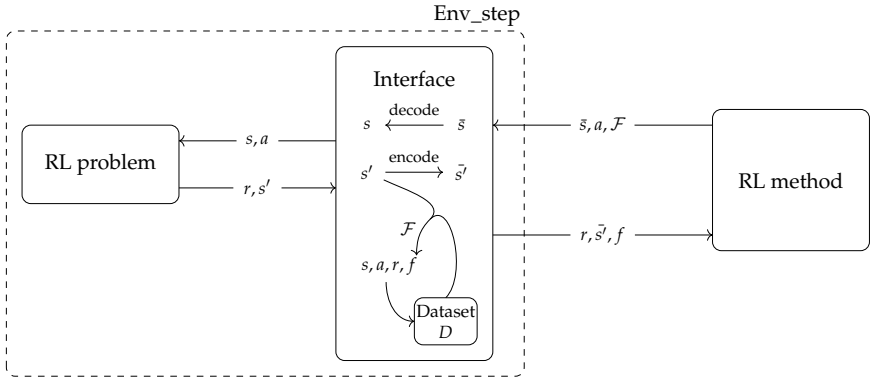

Fig. 4.1 Representation of an RL method that handles the interface linked to an RL problem. The interface allows filtering the information that can flow from the RL problem to the RL method by design. In this Figure, the Env_step algorithm of the interface is pictured. This algorithm allows the RL algorithm to draw a transition from the RL problem. The encountered state s' is passed to the RL algorithm through a reference number to the state, $\bar{s'}$, computed by an encoder-decoder defined in Algorithm 1. The state is also passed to the method through a function evaluation, $f = \mathcal{F}(s', D)$, defined by a dataset D, and a chosen function $\mathcal{F}$. This dataset is internal to the interface and incrementally constructed from previous transitions. The function $\mathcal{F}$ can be chosen to implement a learning algorithm that uses the dataset to train a Machine Learning model and evaluate it on the generated state s', the result can be used as a value function for example.

Algorithm 2 Env : an interface linked to an RL problem defined by the transition operator P. The interface is a set of algorithms with a common internal state. The interface has access to an initialized encoder-decoder of states as defined in Algorithm 1.

$D \leftarrow []$ ▷ internal dataset
function Env_init($\mathcal{F}$)
  $s_0 \sim P_0$
  $\bar{s_0} \leftarrow$ encode_state(s_0)
  output $\bar{s_0}$, $\mathcal{F}(s_0, D)$
function Env_step($\bar{s}$, a, $\mathcal{F}$, append_to_data = True)
  $s \leftarrow$ decode_pointer($\bar{s}$)
  if s is not final **then**
    $r, s' \sim P_{\text{dyn}}(r, s' | s, a)$
    $\bar{s'} \leftarrow$ encode_state(s') ▷ reference to the state
    $f \leftarrow \mathcal{F}(s', D)$ ▷ to use (Q-)value and policy functions
    if append_to_data **then**
      $D \leftarrow D$.append((s, a, r, f))
    output r, $\bar{s'}$, f
  else if s is terminal **then**
    output $\perp$
function Env_eval_state(s, $\mathcal{F}$)
  output $\mathcal{F}(s, D)$
function Env_encode(s)
  output encode_state(s)

metry condition under permutations of the input coordinates. We define such permutations here in Definiton 18.

Definition 18. *A* permutation of coordinates $p : \mathbb{R}^n \to \mathbb{R}^n$ *is a function such that for* $x \in \mathbb{R}^n$, $p(x)_{u(i)} = x_i$ *for some bijective function* u *from* $\{1 \dots n\}$ *to itself.*

We also use interchangeably $p' : \mathbb{N} \times \mathbb{R}^n \to \mathbb{N} \times \mathbb{R}^n$ *defined as* $p'((t, x)) = (t, p(x))$.

From these definitions, we can express Definition 19 of black-box dynamics RL methods.

Definition 19. *A* black-box dynamics RL method *interacts only with the interface (Algorithm 2) linked to the RL problem. The function* $\mathcal{F}$ *given in argument to the algorithms of the interface must have the following form* $\mathcal{F} : \mathcal{S} \times$

$D \to \mathbb{R}^N$ where N is a natural number and $D \in (\mathcal{S} \times \mathcal{A} \times \mathbb{R} \times \mathbb{R}^N)^*$ is a dataset. Moreover, for any permutation of the coordinates p, we must have $\mathcal{F}(s, ((s_0, a_0, r_0, f_0), \ldots)) = \mathcal{F}(p(s), ((p(s_0), a_0, r_0, f_0), \ldots))$ for any state $s \in \mathcal{S}$ and $(s_0, a_0, r_0, f_0), \ldots \in \mathcal{S} \times \mathcal{A} \times \mathbb{R} \times \mathbb{R}^N$.

Interpretation and examples This formulation of the Definition 19 reaches two objectives:

1. To allow a large class of common RL methods to fit the definition.
2. To constrain the information that can pass from the RL problem to the RL method, allowing us to prove the limitation in the next section.

The definition reaches the first objective by allowing to encode common patterns in RL methods. We provide several examples of this flexibility.

The first step in converting an RL method to fit the interface is to replace any generation of transition (s, a, r, s') by a call to the interface defined in Algorithm 2, in particular, the Env_step algorithm pictured in Figure 4.1.

The reward r is directly observable and thus the Definition 19 poses no limit on his manipulations by the calling method. The state s' is not directly observable but the RL method can use $\bar{s'}$ and $f = \mathcal{F}(s', D)$ (still defined in Algorithm 2).

The first element, $\bar{s'}$, can be used to draw new transitions with new calls to Env_step. With this ability, we can sample complete trajectories $(s_0, a_0, r_0, s_1, a_1, \ldots, s_H)$, something which is essential to RL methods. We can also leverage the generation of transitions to simulate common local planning procedures by calling the interface several times with $\bar{s'}$ and different actions (e.g. tree search).

Another step in converting an RL method using ML models to fit the interface is to code the manipulation of these ML models. To accomplish this, the function $\mathcal{F}(s', D)$ can: first, apply a learning algorithm applied on the internal dataset D (composed of old transitions and ML models evaluation) to produce a trained ML model; second, evaluate this model on the state s', giving f, to add information to D and be manipulated by the RL method, as a value function evaluation for example.

For instance, in the case of a deep Q-learning method: D has the role of a replay buffer of past experience; $\mathcal{F}$ is a learning algorithm that given the replay buffer D and a state s, learns a Q-value function then applies it on s. We show in Appendix 3.4 that this construction allows for modeling the training of deep (Q-)value and policy functions.

The function $\mathcal{F}$ is thus used as a learning algorithm in our translation of existing RL methods to use this interface. The symmetry restriction on $\mathcal{F}$ posed in Definition 19 can be intuitively understood as a restriction on the learning algorithm. Specifically, our restriction demands that the learning algorithm treats the coordinates of the input state vectors symmetrically. This restriction is natural since, without a specific prior about the task at hand, we do not wish to process the different coordinates in particular ways. However, we note that in a practical setting with neural networks, the weights are initialized randomly, which can create asymmetries. Nevertheless, we prove in Appendix 3.5 that the distribution of trained neural networks has the demanded symmetries under a natural assumption on their architecture.

We stress that the symmetry restriction imposed on $\mathcal{F}$ does not imply that the learned ML model is invariant to permutations of its input coordinates. Instead, the restriction imposes a symmetry between the learned ML model and the dataset used for its training. For example, when learning a linear model, if we permute the first two coordinates in the training data, then the resulting model will also have the values of its first two parameters permuted.

In the special case where there is known prior information about the RL problem, the practitioner might want to use a Machine Learning algorithm that does not fit this restriction. For example, in Image Vision tasks Convolutional Neural Networks (CNNs) are commonly used and those do not respect our symmetry restriction. However, in the case of CNNs, these architectures usually produce a latent space over which an agnostic neural network architecture is applied. Our results will apply to this latent space. More generally, we believe that our results can apply under any sufficiently broad prior.

In Appendix 3.4, as an example of our formalization, we show how classical RL methods can be translated to use the interface as constrained in Definition 19. We give four complete examples: a fitted Q-iteration method; a model-free policy gradient algorithm (Actor-Critic method); a local tree search model-based method; and a combination of tree-based planning and value function à la AlphaZero [Silver et al., 2017].

These examples are archetypes of the main categories of RL methods and cover Q-learning with the Bellman equation, Policy Optimization with policy gradients combined with a critic, and black-box model predictive control with tree-based search. The last example also covers a mix of model-based tree-based search and model-free RL. We refer to the documentation

of Achiam [2018] for a taxonomy of RL methods. These examples also illustrate how several commonly used exploration methods fit our framework: ϵ-greedy, on-policy, and optimism under uncertainty with UCB. Our proofs for these examples depict how other methods can be translated to fit our Definition 19.

4.3 Main theorem: limitation of black-box dynamics RL methods

We just characterized the class of black-box dynamics RL methods through the Definition 19. We prove in this section that these methods suffer from a computational efficiency limitation on a family of RL problems. In contrast, these problems are efficiently solved by a toy method. The toy method learns to map future states to actions and then applies that mapping to reach a detected rewarding state.

We define here this efficient toy method based on Hindsight Experience Replay [Andrychowicz et al., 2017].

The toy method, Algorithm 3, first samples a dataset of trajectories by drawing actions uniformly. From this dataset, it extracts a final state that gives maximal reward when entering it and poses this state as its goal. From the same dataset, a function is learned to predict the action taken from any state given the reached final state. Finally, the algorithm constructs a policy that follows the action predictor conditioned on the goal as the final state.

The learning algorithm for the action prediction minimizes the empirical rate of errors on the dataset. The space of functions in which we learn is a composition of a feature selection, a linear function, and a threshold (to output a binary prediction). The formal mathematical program is

$$\begin{aligned} \underset{f_w \in F}{\arg\min} \quad & \sum_{((s_t, s_H), a) \in D^t_{GC}} \mathbf{1}(f_w(\begin{bmatrix} s_t \\ s_H \end{bmatrix}) \neq a) \\ \text{s.t.} \quad & \|w\|_0 \leq \alpha, \end{aligned} \tag{4.1}$$

where $\begin{bmatrix} s_t \\ s_H \end{bmatrix}$ denotes the states concatenated in a vector in $\mathbb{R}^{2n}$, dataset D^t_{GC} is defined in Algorithm 3, F is the set of linear functions with a threshold, $F = \{x \in \mathbb{R}^{2n} \to \mathbf{1}(\langle w, x \rangle > 0) | \, w \in \mathbb{R}^{2n}\}$, and $w \in \mathbb{R}^{2n}$ is the parameter associated to f_w. The condition $\|w\|_0 \leq \alpha$ bounds the number of non-

Algorithm 3 A goal-conditioned algorithm.

Parameters: N: number of samples, α: parameter of the learning algorithm (number of active features)

Input: P: the transition operator corresponding to the RL problem

$D \leftarrow \{\}$

for $i \leftarrow 1, \dots N$ **do**

$D \leftarrow D \cup \{(s_0, a_0, r_0, s_1, a_1, \dots, s_H) \sim P^{\pi^U}\}$

$g \leftarrow \arg\max_{s_H} r \quad \text{s.t.} \quad (\dots, r, s_H) \in D$

for $t \in \{0, \dots, H-1\}$ **do**

$D^t_{GC} \leftarrow \{((s_t, s_H), a_t) | \, (\dots, s_t, a_t, \dots, s_H) \in D\}$

$f^t \leftarrow$ the result of the optimization program in Equation 4.1 with dataset D^t_{GC} and parameter α.

output $\pi(a|s_t) = f^t(a|s_t, g)$

zero weights in the linear function by $\alpha \in \mathbb{N}$. This space of hypotheses and regularization condition are chosen to facilitate the theoretical proof, our numerical experiments will show that the results also hold with more general ML algorithms.

This method is elementary and limited: exploration is performed with a uniform policy; F is a restricted set of functions; the method is only trying to reach a final rewarding state to maximize expected returns; and the scheme is not sound in stochastic dynamics. Thus, this method does not address the general problem of RL. Nevertheless, the method is sufficient to support the claims of this paper:

- The RL problems in Theorem 20 are easy to solve and not efficiently solving them is a limitation since any RL method that cannot efficiently solve these problems is beaten by Algorithm 3, a toy method that merely predicts actions to reach a detected rewarding state.
- A method that constructs a model of the inverse dynamics (such as Algorithm 3) can help to avoid the issue identified in this paper.

We have now defined everything needed to state our main result.

Theorem 20. *There exists a family of RL problems such that*

1. *For any RL problem in the family and $\delta \in (0, 1)$, with probability at least $1 - \delta$ (over the sampled trajectories), Algorithm 3 outputs an optimal policy with a number of samples and number of operations upper bounded by a polynomial in horizon H and $1/\delta$.*

2. *For any algorithm satisfying Definition 19 and using $o(2^H)$ calls to the interface (Algorithm 2), there exists a problem in the family for which it outputs a suboptimal policy with probability at least 1/3.*

The second affirmation bounds from below the computational complexity of black-box dynamics RL methods to solve the problems, thus for those methods the family of problems is intractable. Conversely, the first claim ensures that Algorithm 3 efficiently solves the family of problems. Thus, the Theorem proves a separation in efficiency to the disadvantage of black-box dynamics methods, implying a limitation on a large class of methods (which includes, as seen in Section 4.2, Q-learning, Actor-Critic methods, AlphaZero, ...). Another consequence of the Theorem is that the toy method based on Hindsight Experience Replay (Algorithm 3) cannot be formulated as a black-box dynamics method.

The complete proof is given in Appendix 3.1. It is partially inspired by some of the proofs in Sun et al. [2019]. We provide a sketch here with the main intuitions.

We explicitly construct a family of RL problems satisfying the requirements in Theorem 20. The problems in our family are defined by two parameters: a horizon $H \in \mathbb{N}$; a hidden binary word $b \in \{0,1\}^{H-2}$ that represents a sequence of actions that solves the task. A representation of one element of the family for a small horizon is given in Figure 4.2. We remark that the dynamics is deterministic with the exception of the start and a reward is obtained at one unique state. The problem can thus be seen as finding a path to a goal in a graph.

Each state is defined by $3(H-2)+1$ real variables. The action space is binary. An example in this family for $H = 4$ and $b = 01$ is represented in Figure 4.2.

We pose some notations to describe the states. Each state is decomposed into three parts a, b, and c, the part b is further decomposed into parts u and d. The time step to which the state belongs is kept implicit in our description. For i between 1 and $H-2$:

- $s^{a,i}$ the ith variable in the first part of the state vector;
- $s^{b,u/d,i}$ if u, the up ith variable in the second part of the state vector, if d, the down ith variable in the second part of the state vector;
- s^c the variable of the third part of the state vector.

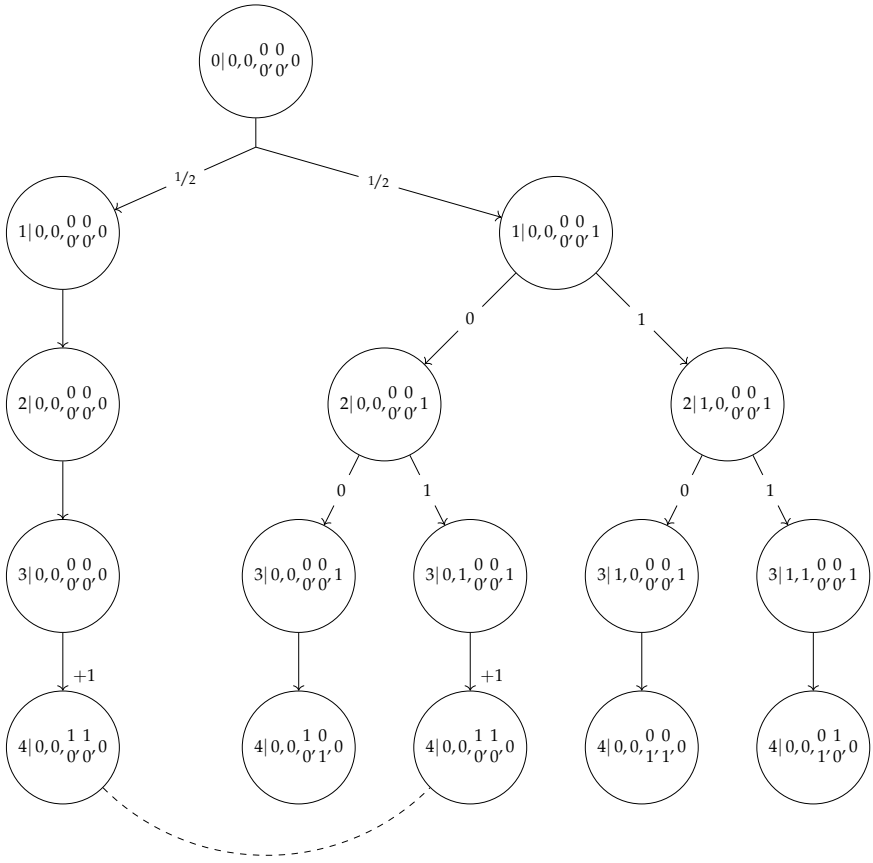

Fig. 4.2 Example of the family of RL problems for horizon $H = 4$ and hidden binary word $b = 01$. On the left-hand dynamics, the agent can easily discover the unique rewarding state. In comparison, a reward is hard to get in the right-hand dynamics using random exploration but by exploiting the dynamics and a discovered relevant goal, an algorithm can uncover an optimal path (such as toy method Algorithm 3).

Example of the notation for state vector s:

$$\left[s^{a,1}, s^{a,2}, \ldots, s^{a,i}, \ldots, s^{a,H-2}, \begin{matrix} s^{b,u,1} \\ s^{b,d,1} \end{matrix}, \begin{matrix} s^{b,u,2} \\ s^{b,d,2} \end{matrix}, \cdots, \begin{matrix} s^{b,u,i} \\ s^{b,d,i} \end{matrix}, \cdots, \begin{matrix} s^{b,u,H-2} \\ s^{b,d,H-2} \end{matrix}, s^{c} \right].$$

Now we describe the dynamics. At step 0, whatever action is taken,

with probability $1/2$, we reach either a state full of zeros, or a state full of zeros except for s^c which equals one.

If we have $s^c = 0$, then for the steps $t = 1$ to $t = H - 1$, the state vector stays zero. At the last step it gives a reward of 1 and the state with $s^a = 0^{H-2}$, $s^{b,u} = 1^{H-2}$, $s^{b,d} = 0^{H-2}$ and $s^c = 0$.

If $s^c = 1$, then from time step $t = 1$ to $t = H - 2$, the transition from t to $t + 1$ encodes the binary action taken into $s^{a,t}_{t+1}$ with zero reward (the other variables keep their values from the previous time step). At the last step, the dynamics fixes $s^c_H = 0$, all the variables that encode a past action to zero $s^a_H = 0^{H-2}$, and s^b_H to express if each action taken was correct $\begin{pmatrix}1\\0\end{pmatrix}$ or not $\begin{pmatrix}0\\1\end{pmatrix}$ with respect to a fixed optimal trajectory given by b the binary word parametrizing the problem. In other words, $s^{b,u,i}_H = \mathbf{1}(s^{a,i}_t = b_i)$ and $s^{b,d,i}_H = \mathbf{1}(s^{a,i}_t \neq b_i)$. At that step, if $s^{b,u}_H = 1^{H-2}$, then a reward of 1 is given. Such that, a reward is obtained only if all the actions taken follow the hidden binary word b.

At a high level, the dynamics starts at a unique initial state from which two possible states are drawn randomly. The first state starts the left-hand dynamics, a unique trajectory that ends up in a rewarding state. The second state starts the right-hand dynamics, in it the number of possible trajectories grows exponentially with the horizon. At the end of these trajectories, the dynamics transitions to a final state that represents the differences between the history of actions taken and the hidden binary word b. If there are no differences between the binary actions taken and b, we arrive at the same state as the left-hand dynamics, and a reward is obtained.

The right-hand dynamics has a number of possible final states that grows exponentially in the horizon, and the unique rewarding state is hard to reach by purely random actions independent of the hidden binary word b. Information about b is only present in the final states, and we show in the proof that the information in the final states of the right-hand dynamics is hidden under the Definition 19. More precisely, Definition 19 only allows information about these states to filter through evaluations of $\mathcal{F}(., D)$, and any $\mathcal{F}(., D)$ is provably constant on these states due to the symmetry restriction. Methods satisfying Definition 19 will thus not do better than enumerating all the possible behaviors since they do not have access to relevant information to orient the search in the right-hand dynamics.

We illustrate the reasoning with the example of a Q-learning algorithm

implemented with the interface applied to the problem in Figure 4.2. When sampling some final state s_H, the only information filtering through the interface to the method is the obtained reward, the encoding $\bar{s}_H$, and $Q_{s_H} = \mathcal{F}(s_H, D)$. The encoding $\bar{s}_H$ is simply the number of states encoded up to now and reveals no information about s_H and the problem. The output Q_{s_H} represents the Q-value estimates at that state built from the dataset D composed of tuples $(s, a, r, Q_{s'})$, where these tuples are based on previously sampled transitions (s, a, r, s') and an old estimation $Q_{s'}$ of the Q-values at s'. The dataset can only contain non-final states by construction of the interface. Also, by construction of the problem, the (non-final) states in the dataset and final states have different coordinates with non-zero elements. Given these elements, an analysis with the symmetry condition on $\mathcal{F}$ in Definition 19 reveals that $Q_{s_H} = \mathcal{F}(s_H, D)$ is constant with respect to s_H. Thus, in this case, only the obtained reward exposes information about the problem for a black-box dynamics method.

In this analysis, the symmetry condition asked on $\mathcal{F}$ is crucial. It allows us to prove that the outputs of $\mathcal{F}$ are equal for different states in our problems. These states are thus indistinguishable, and imply that the algorithm fails to leverage the crucial information in them.

For Algorithm 3, the left-hand dynamics allows the RL method to discover the rewarding state easily and set it as its goal. The right-hand dynamics is sufficiently simple for the learning procedure to reliably predict the necessary sequence of actions to reach this goal. With high probability, the returned policy will thus reach the rewarding state in the right-hand dynamics.

Generalization Our result defines an example family of problems but does not provide general conditions on problems upon which the limitation on black-box dynamics methods appears. Nevertheless, from this family, we can extract some key elements to our proof of the limitation:

- the dynamics have several actions to take and each is critical to the success;
- each action has a positive or negative effect depending on the initially unknown dynamics;
- the dynamics aggregate the effects of all the actions into one binary reward, thus hiding the effects of each action to a method relying on reward feedback.

At the same time, these problems are tractable for our toy methods because:

- a rewarding state reachable with the above actions can be inferred from random exploration;
- the effect of each action is revealed in the state-to-state transitions.

These core characteristics informally define sufficient conditions to show a limitation. An improvement over our result could be to formally define sufficient conditions as general as possible on problems for the occurrence of a limitation. This would require several generalizations over Theorem 20. More precisely, it would be necessary to generalize the bound on the performance of black-box dynamics methods to a broader range of problems. Additionally, a more universal algorithm should be proposed than our toy method (Algorithm 3), supported by an equally universal guarantee of its performance. These generalizations and their scope will determine the breadth of any obtained limitation.

Domination A corollary of Theorem 20 is that for any black-box dynamics RL method, another method exists that always performs at least as well and sometimes better on RL problems. This result can be obtained by constructing a new method that executes in parallel both the black-box dynamics method and Algorithm 3. Appendix 3.1 contains a formal statement and associated derivation of this fact.

4.4 Numerical experiments

We illustrate and confirm numerically how practical deep RL methods perform on the family of RL problems constructed in the proof of Theorem 20. We also test the goal-conditioned method from the last section. Moreover, we introduce another method that performs well on our set of problems. This method leverages a non-black-box model of the dynamics for planning.

We test model-free methods such as fitted Q-iteration [Riedmiller, 2005]. We implement and run a classical Actor-Critic method, Proximal Policy Optimization (PPO) [Schulman et al., 2017]. Moreover, we test AlphaZero, a model-based method leveraging learning and local planning [Silver et al., 2017]. We refer to Appendix 3.7 for details on the implementations.

We test Algorithm 3 but with neural networks trained by gradient descent as a learning procedure instead of the function space defined in Equation 4.1.

In addition, we propose and test a second algorithm that empirically performs well on the family of RL problems. A more detailed description of this algorithm is available in Appendix 3.6.

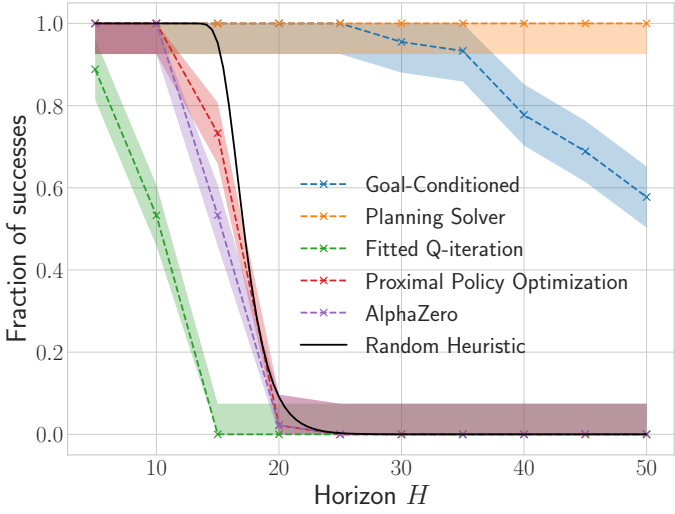

Fig. 4.3 Fraction of successes as a function of the horizon for each method on the RL problem family illustrated in Figure 4.2. We performed 45 runs for each point on the Figure. An upper bound on the standard deviation is displayed, it is computed using the maximal possible variance for a Bernoulli distribution, $1/4$. The *Goal-Conditioned* denotes Algorithm 3 with neural networks and *Planning Solver* is the planning method presented in this section. The *Random Heuristic* denotes the analytically computed performance of a method that samples random actions until reaching a rewarding state on the right-hand dynamics of the problems, then copies that behavior. This method assumes access to 50.000 sampled trajectories.

This algorithm iteratively alternates between two computations. One, learning a model of the dynamics from sampled trajectories. Two, sample trajectories by using a planning algorithm to determine the actions. The planning algorithm converts the current state and the learned model of the

dynamics into a mathematical program that maximizes the sum of future rewards. Then, an off-the-shelf solver solves the mathematical program and outputs a plan with the action to take. The algorithm iterates this procedure until a performance threshold is obtained, then the training stops and the method outputs the planning procedure as its policy.

We propose this algorithm to support two claims.

Firstly, we illustrate in a second way that our RL problems are tractable. This method relies on the fact that the dynamics of our MDPs are easy to learn and an off-the-shelf solver can plan efficiently in these dynamics. Those are general properties that an efficient general method should be able to take advantage of.

Secondly, this algorithm illustrates another idea in the literature than the goal-conditioned Algorithm 3. Algorithm 3 learns a link from future states to present actions directly, while here we propose an algorithm that learns a model of the forward dynamics. This algorithm is also interesting to compare with other model-based RL methods on which the limitation applies (e.g. based on tree search). This new planning algorithm distinctively leverages an explicit model of the dynamics by decomposing this model into structured constraints, instead of only using the model as a black-box generator of samples.

We apply these methods to problems of the family constructed in the proof of Theorem 20 with an increasing horizon. For the fitted Q-iteration and PPO methods, we sample 50 times 1000 trajectories during one training run. The AlphaZero method accesses 3 times 1000 trajectories, where for each state in these trajectories, the planning procedure leverages 50 simulation of transitions. Thus, the method leverages an equivalent total of 150 000 sampled trajectories. We sample a dataset of 1000 trajectories for the goal-conditioned method. The new planning algorithm takes 1000 randomly sampled trajectories for its initialization, and then at most 1000 additional trajectories are sampled in the alternations between learning and planning for any run.

To measure the success of a method, we check if its returned policy reaches the rewarding state in the right-hand dynamics with 1000 sampled trajectories, this allows the goal-conditioned algorithm to do minimal exploration.

The results are presented in Figure 4.3. The model-free methods quickly fail to solve the task when the horizon is increased, their performance is bounded by a simple heuristic method that randomly explores the environment and then copies the best-performing behavior. This is in line with

our theoretical results.

Despite being allowed fewer samples and not more computing time, the goal-conditioned and planning methods successfully solve the tasks for much longer horizons.

4.5 Conclusion

We introduced the class of black-box dynamics RL methods with Definition 19 and its linked interface. This class encompasses model-free methods such as Q-learning and Policy Optimization, as well as several model-based methods such as AlphaZero. For this broad range of methods, we proved a computational efficiency limitation on a family of problems in Theorem 20. However, we described two toy methods that can efficiently solve these problems.

The problems used in the proof of Theorem 20 feature two different dynamics randomly chosen at the start of the trajectory. The first dynamics is simple and allows an algorithm to easily discover a rewarding state. In the second dynamics, finding a behavior that reaches a rewarding state is hard. However, these two dynamics have a common unique rewarding state. An efficient method is able to discover the rewarding state in the first dynamics and then plan to reach that state with learned knowledge of the second dynamics.

To summarize our findings intuitively, a large class of RL methods will unnecessarily struggle in environments where following rewards alone is not informative enough. Our results reveal an incapacity of black-box dynamics methods to leverage essential information from some RL problem dynamics.

Our work not only describes a limitation on a class of methods but also suggests algorithmic ideas that could be interesting to overcome the limitation. For example, our first toy method (Algorithm 3) leverages state-to-state reachability to avoid the limitation. This leads to a new question: does learning with state-to-state reachability is enough for efficient problem-solving? Or does another limitation apply to this approach, and other ideas are needed?

An additional dimension worth exploring is the generalization of our limitation to more cases. One important aspect is that our limitation relies on a family of problems with distinct dynamics. However, there are practical problems where the dynamics is well known in advance. Does it mean that there is no limitation when the dynamics is known a priori?

The next chapter answers these questions.

5

Inefficiencies in (universal) value functions learning

This chapter expands on the analyses made in the last chapter, studying limitations on algorithms blending search and learning for problem-solving. A typical paradigm for these algorithms is to leverage learning to accelerate the search process. These algorithms iteratively attempt to construct solutions, leveraging feedback from previous attempts to learn to guide the next constructions (as pictured in Figure 1.2).

Reinforcement Learning (RL) implements this paradigm to optimize objectives over sequences of decisions, where each decision-making point is associated with a state. This structure allows Dynamic Programming to be applied in the form of the Bellman equation to learn a value function. RL algorithms typically sample sequences of states guided by the value function while simultaneously improving this value function by enforcing the Bellman equation on the sampled states. This approach forms the foundation of many RL algorithms, with various adaptations [Sutton and Barto, 2018].

We note that this approach of constructing a (probabilistic) Machine Learning (ML) model of the value function is sound even in the case where the problem has a deterministic dynamics. The ML model allows learning, evaluating, and representing a value function over a potentially combina-

torially large state-space while leveraging (or relying on) generalization to tame this large space.

This chapter examines the theoretical limitations of this RL approach to enhance our understanding and support the development of more effective algorithms. We note that, in contrast to the last chapter, instead of defining an abstract class of methods and then proving a limitation on the whole class, here, we construct our result by describing and analyzing one representative algorithm. This approach ensures greater readability but at the cost of formal generality.

Intuitively, the results of this chapter formalize the incapacity of these algorithms to efficiently "learn from failure". For instance, in Automated Theorem Proving, the probability of proving a theorem with randomly sampled logical rules is often low, this leads to sparse rewards and might hinder the capacity of an algorithm to learn. While proving an incapacity to solve a problem with sparse rewards without any other information is trivial —since any algorithm would resort to exhaustive search— we analyze the non-trivial case: when the algorithm can leverage a priori information to help his search and learn from its failures.

To study this limitation, we analyze a concrete RL algorithm that applies the Bellman equation with Bayesian Learning. The algorithm starts with a set of hypothesis value functions and iteratively refines this prior through the Bellman equation. Bayesian Learning provides us with an idealized framework for incorporating prior knowledge while refining it with experience.

Under natural assumptions on the initial prior, we prove a key limitation. Despite the strengths of the RL approach and the learning procedure, the algorithm struggles with certain counterexample problems with an "easy" design. In contrast, we show that a classical symbolic algorithm is not subject to such a limitation. This suggests that the highlighted limitation is not a fatality but rather a shortcoming due to algorithmic choices.

The counterexamples are based on a similar principle than the last chapter: combining a set of problems into one aggregated problem that should be (with an appropriate strategy) nearly as easy to solve as the original sub-problems. However, by carefully designing the aggregation, we can obscure the information of each individual sub-problem, forcing the value-based RL algorithm to tackle all original sub-problems simultaneously rather than breaking them down into independent, simpler tasks. From each failure to generate a solution, the algorithm only learn to prune a tiny corner of the space of possibilities, thus ineffectively "learning from failure".

A potential remedy to this limitation is expanding the feedback leveraged by the algorithm. One such approach is Hindsight Experience Replay (HER) [Andrychowicz et al., 2017]. HER builds upon the Bellman equation to learn *universal value functions* presented in Section 1.2.3 [Sutton et al., 2011, Schaul et al., 2015]. Assuming that the objective is to reach some goal-state, a standard value function only estimates the value of a state in terms of its ability to reach this goal. In contrast, a universal value function predicts the reachability of any state from any other state. HER utilizes the states encountered during the learning process to learn this universal value function, leveraging richer feedback than a simple binary outcome of whether the goal was reached or not.

However, this new feedback also misses critical information to solve our counterexamples efficiently. We straightforwardly extend our analysis to HER with state-to-state universal value functions, proving that the same limitation applies.

Contributions Both Sun et al. [2019] and our previous chapter highlighted limitations in methods relying on value functions by embedding critical information within unknown dynamics. Instead of relying on uncertainty in the dynamics, our analysis builds on the computational limitations of evaluating (universal) value functions. This method enables our contributions:

- Demonstrating a limitation for an algorithm based on value functions on a family of problems with an a priori known common dynamics across the problems. This extends the applications in which an inefficiency for a value-based method can be expected.
- Formalizing a limitation for an algorithm relying on universal value functions learning state-to-state reachability, such as in Hindsight Experience Replay.

Our contributions answers the two questions opened in Section 4.5:

- Is there still a limitation on a value-based RL algorithm when the dynamics is known a priori? Yes.
- Does learning to perform state-to-state reachability and a known goal-state is enough for efficient problem-solving? No.

Related work The structure in our counterexamples is efficiently solved by a conflict-driven clause learning (CDCL) SAT solver. These modern SAT

solvers include many enhancements over the simple enumeration of solutions, such as unit propagation, (non-chronological) backtracking, clause learning, variable selection and assignment heuristics, and restart policies [Knuth, 2015]. From all these ideas, our positive result for this approach is demonstrated using the clause learning procedure, which automatically divides our counterexamples into their independent sub-problems.

The close problems of #SAT and MaxSAT have their own algorithms, which are usually based on Dynamic Programming to either count the number of solutions or the maximal number of clauses concurrently satisfied in one assignment [Sæther et al., 2015]. These algorithms decompose the problem according to the structure of its clauses, and thus can leverage the aggregation structure present in the counterexamples of our results.

Outline In the preliminaries, we provide convenient notations and definitions. In Section 5.2, we present an algorithm implementing the Bellman equation applied to learn a value function, and then derive a computational limitation for that algorithm. In Section 5.3, we extend that limitation to an HER algorithm. Finally, in Section 5.4, we discuss our results and conclude.

5.1 Preliminaries

We note $[n] = \{1, \dots, n\}$ the set of the n first natural numbers. For a vector $x \in X^n$, with some set X and $n \in \mathbb{N}$, we note $x_{\leq i}(/x_{<i})$ the vector restricted to the first $i(/i-1)$th coordinates. An index list I over $n \in \mathbb{N}$ is a sequence of numbers in $[n]$. For I an index list over n, x_I is the vector composed of the values of x at the coordinates in I. For x and y two vectors, we note $[x, y]$ their concatenation.

Our counterexamples are based upon the *Boolean satisfiability* problem, we define here a classical form of this problem. Given a vector of $n \in \mathbb{N}$ Boolean variables x_i ($i \in [n]$): a *literal* is one of these variables x_i or its negation $\neg x_i$; a *clause* is a set of literals joined by disjunctions $\vee$. Finally, a *conjunctive normal form satisfiability* (CNF-SAT) instance is a set of clauses over the n variables and a solution for the instance is a vector $x \in \{\text{False}, \text{True}\}^n$ such that all the clauses evaluate to True under the interpretation given by x. We denote this evaluation function checking a binary vector x against a CNF-SAT instance p $\text{Check}(x; p)$.

We take two liberties with respect to this formalism. One, we use the binary space $0/1$ in place of the Boolean space. Two, in all the formal parts

of the chapter, we keep the name "CNF-SAT instance" (following the computational complexity literature) but we also denote them by the generic term of "problem" in other parts of the manuscript.

5.2 Limitation on the Bellman Equation with Value Functions

We identify a limitation of a value-based algorithm when applied to solving CNF-SAT instances. Specifically, we show that this algorithm can fail to exploit structure in aggregated instances. We demonstrate this with Algorithm 4, which implements the Bellman equation to learn a value function and guide the search towards solutions for a CNF-SAT instance. Theorem 26 establishes that the algorithm does not effectively decompose some counterexample aggregated instances, resulting in an exponential runtime with respect to the number of instances.

5.2.1 The algorithm

To solve a CNF-SAT instance using RL, we formulate the problem as a sequence of binary decisions. An instance with n binary variables is modeled as a sequence of n binary decisions that form a candidate solution. For each decision, the state is defined as the SAT instance and the first i fixed variables, while the next action involves assigning a value (0 or 1) to the $i+1$th variable. The RL algorithm seeks to optimize these decisions under the objective of maximizing 1 for a solution and 0 otherwise.

To guide the generation toward a solution, our RL algorithm leverages value functions learned by enforcing the Bellman equation. We now define these value functions, their optimality, and the Bellman equation.

Definition 21. *A* value function $v(x_{\leq i}; p)$ *maps a CNF-SAT instance p over n variables and a partial assignment $x_{\leq i} \in \{0,1\}^i$ with $i \in [n]$ to $\{0,1\}$.*

Definition 22. *A value function v is* optimal *if for any CNF-SAT instance p with n variables and any partial assignment $x_{\leq i}$, $v(x_{\leq i}; p) = 1$ iff there exists an extension $y \in \{0,1\}^{n-i}$ such that* Check$([x_{\leq i}, y]; p)$ *is True.*

Adapting the Bellman equation 1.2 of Section 1.2.1 gives for some value function v at some partial assignment $x_{\leq i}$:

$$v(x_{\leq i}; p) = \max\{v([x_{\leq i}, 0]; p),\ v([x_{\leq i}, 1]; p)\}, \tag{5.1}$$

Algorithm 4 Learning value functions with the Bellman equation for CNF-SAT.
The algorithm iteratively samples candidate solutions guided by a hypothesis set of value functions. In parallel, the Bellman equation is iteratively enforced on the set. The sampling procedure follows $\pi^{V^t}(p)$ defined in Definition 23.

Inputs: V^0 : an initial set of value functions; p : the CNF-SAT instance to solve with n variables

for $t \leftarrow 0, 1, \ldots$ **do**
  $x^t \sim \pi^{V^t}(p)$
  if Check$(x^t; p)$ **then**
    output x^t
  $V^{t+1} \leftarrow \{v \in V^t \,|\, v(x^t; p) = 0\}$
  for $i \leftarrow 1, \ldots, n-1$ **do**
    $V^{t+1} \leftarrow$
      $\{v \in V^{t+1} \,|\, v(x^t_{\leq i}; p) = \max\{v([x^t_{\leq i}, 0]; p),\, v([x^t_{\leq i}, 1]; p)\}\}$

where $i \in [n-1]$ and p is the CNF-SAT instance to solve. Given that our setup involves deterministic transitions, this expression does not use expectations. Also, since the objective is binary, we use value functions with a binary output.

The Bellman equation must also enforce consistency with Check when evaluating a complete assignment. This additional constraint reads $v(x; p) =$ Check$(x; p)$ where x is any complete assignment.

We now define Algorithm 4, it iteratively generates candidate solutions guided by values' estimates while simultaneously improving these estimates through the Bellman equation.

The algorithm starts with an initial set of hypotheses for the value functions, denoted as V^0, and iteratively samples sequences of decisions to construct candidate solutions, guided by the current set V^t. At each step, the Bellman equation is applied to eliminate inconsistent value functions from V^t, continuing this process until a solution is found.

The initial hypothesis set V^0 encodes a prior over possible value functions. This prior can incorporate knowledge about solving SAT instances, value functions excluded from V^0 reduce the set of possibilities and potentially accelerate the search. This knowledge may originate from some human-coded prior or from training on other SAT instances.

Likewise, the set V^0 also encodes uncertainties. When V^0 contains

value functions providing different estimates, this reflects uncertainty and can force the algorithm to explore multiple possibilities. These possibilities are then progressively discarded with the Bellman equation.

Algorithm 4 thus implements a form of Bayesian Learning where a set of hypotheses is iteratively refined based on new evidence. Traditional Bayesian Learning employs a probabilistic distribution over the hypothesis space, but in this case, Equation 5.1 provides a deterministic condition for consistency. This eliminates the need for a probabilistic framework, allowing us to work directly with the set of hypotheses. However, we note that the results presented here are transposable to the case where probabilistic distributions are used.

The sampling of candidate solutions is guided by the current set of value functions V^t, and the sampling procedure is formalized as follows:

Definition 23. *For a given set of value functions V, we define π^V as a mapping from any CNF-SAT instance p with n variables to a probability distribution over the space of binary candidate solutions $x = [x_0, \ldots, x_n] \in \{0,1\}^n$.*

Let $V_{z,p}$ denote the number of value functions in V that evaluate to 1 for inputs $z \in \{0,1\}^{[n]}$ and p, i.e., $|\{v \in V | \, v(z;p) = 1\}|$.

The probability distribution is defined incrementally as

$$\pi^V(x_i = 0; p, x_{<i}) = \frac{V_{[x_{<i},0],p}}{V_{[x_{<i},0],p} + V_{[x_{<i},1],p}}, \tag{5.2}$$

with $\pi^V(x_i = 1; p, x_{<i}) = 1 - \pi^V(x_i = 0; p, x_{<i})$. If the denominator in Equation 5.2 equals zero, both actions are assigned equal probabilities.

This procedure converts a set of hypothesis value functions V into a sampling policy. The resulting samples are sequentially constructed candidate solutions, where each bit x_i is appended based on an estimated probability of the existence of a solution.

5.2.2 Counterexamples

With our algorithm defined, we now present a negative result by constructing examples of hard instances. To do so, we introduce an operation that aggregates multiple CNF-SAT instances. This operation is formally defined as follows:

Definition 24. *Let $p_1, \ldots, p_K$ be CNF-SAT instances over $n_1, \ldots, n_K$ variables, respectively. Additionaly, let $I_1, \ldots, I_K$ be index lists over $n = n_1 + \ldots + n_K$*

variables such that $|I_1| = n_1, \ldots, |I_K| = n_K$*, and all the elements in these lists are distinct. An* aggregation *of* $p_1, \ldots, p_K$ *with index lists* $I_1, \ldots, I_K$ *produces a new CNF-SAT instance* p *over* n *variables, where all the literals in the clauses are mapped according to the index lists. For example a clause* $x_i \vee \neg x_j$ *in* p_k *becomes* $x_{I_{k,i}} \vee \neg x_{I_{k,j}}$ *in* p*.*

To support our result, we also introduce a monotonic property for value functions.

Definition 25. *A value function* v *is* monotonic *if, for any CNF-SAT instance* p *with* n *variables, any* $x \in \{0,1\}^n$*, and any* $i, j \in [n]$ *with* $i < j$*, the function satisfies:* $v(x_{\leq i}; p) \geq v(x_{\leq j}; p)$*.*

This property implies that, from any partially constructed solution, each decision can only decrease the value function's output. It is a characteristic of any optimal value function.

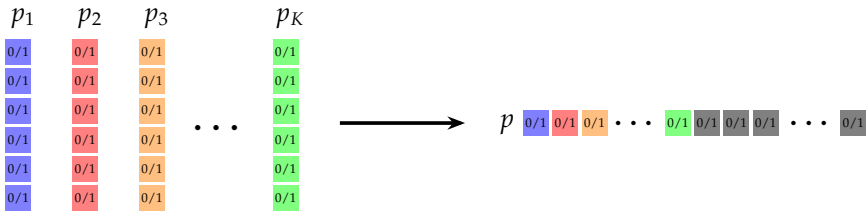

Fig. 5.1 Representation of a counterexample problem. Independent sub-problems $p_0, \ldots, p_K$ are aggregated into a single composite problem p, where the first variable of each sub-problem is mapped at the start of the new problem. Under appropriate assumptions, Theorem 26 states that this construction forces a value-based algorithm (Algorithm 4) to have an exponential runtime in the number of sub-problems K.

We now have all the necessary elements to state our result. Theorem 26 demonstrates that, under certain assumptions, an aggregated problem is computationally intractable for Algorithm 4. The proof leverages the fact that, while the Bellman equation is sufficient to eventually identify an optimal value function and produce a solution, it does so inefficiently in this context.

Importantly, the Bellman equation relies on the outputs of the value function to learn. When a solution attempt fails, the output 0 of the value function (indicating failure) provides minimal feedback. This lack of informative feedback prevents the algorithm from understanding the reasons behind the failure and learning to anticipate similar failures.

Theorem 26 constructs a problem with this difficulty by forcing Algorithm 4 to make several hard independent binary decisions at the start. If any one of these decisions is incorrect, the attempt fails (value 0) and the algorithm struggles to attribute the failure to any particular decision.

A representation of the aggregation operation used in our counterexamples is provided in Figure 5.1.

Theorem 26. *Let p be a CNF-SAT instance over n variables, constructed by an aggregation of CNF-SAT instances $p_1, \dots, p_K$ using index lists $I_1, \dots, I_k, \dots, I_K$, where the first element of each I_k is k.*

Let V and $V_1, \dots, V_K$ represent sets of value functions, and let v^* denote an optimal value function.

Assume the following:

1. *The set V factorizes into $V_1, \dots, V_K$; that is, $v \in V$ iff there exist $v_1 \in V_1, \dots, v_K \in V_K$ such that for all $x \in \{0,1\}^n$ and $i \in [n]$, $v(x_{\leq i}; p) = \prod_{k \in [K]} v_k(x_{I_k \cap [i]}; p)$.*
2. *For all $k \in [K]$, either $v^*(0; p_k) = 1$ or $v^*(1; p_k) = 1$.*
3. *For all $k \in [K]$ and $v \in V_k$, either $v(0; p_k) = 1$ or $v(1; p_k) = 1$.*
4. *For all $k \in [K]$, if $v^*(0; p_k) = 1$, then $\pi^{V_k}(x_0 = 0; p_k) \leq \pi^{V_k}(x_0 = 1; p_k)$; otherwise, $\pi^{V_k}(x_0 = 0; p_k) \geq \pi^{V_k}(x_0 = 1; p_k)$.*
5. *Any $v \in V$ is monotonic.*

Under these assumptions, Algorithm 4, initialized with $V^0 = V$ and $p = p$, runs for an expected time of at least 2^{K-1} steps.

The complete proof is deferred to Appendix 4.1. Here, we outline the key ideas.

By assumption (2), only one possible assignment for the Kth first variables leads to a solution. Due to assumptions (1) and (4), this assignment has a low prior probability under V^0.

The algorithm tries to iteratively improve its prior through the Bellman equation. However, due to assumptions (1), (3), and (5), only a small portion of the hypothesis space is inconsistent with the equation on the generated states. Consequently, the prior is not substantially improved at each iteration.

By quantifying these statements, it follows that the algorithm requires 2^{K-1} expected steps to solve the problem.

Discussion of Assumptions in Theorem 26 Assumption (1) requires that the prior over value functions V^0 reflects the structure of the aggregated problem. Specifically, the set of value functions must be factorizable into sets of value functions corresponding to each individual sub-problem. This assumption is natural when sub-problems are independent, meaning that learning something about one sub-problem does not aid in solving the others.

Assumption (2) posits that the first bit of each sub-problem is critical to solving that sub-problem. SAT instances exhibiting this property can be straightforwardly constructed. Assumption (3) asks that the value functions in the prior V^0 reflect that structure. Any optimal value function satisfies this condition.

Assumption (4) asks for uncertainty in the prior V^0. With this condition, the prior V^0 does not provide sufficient guidance to decide optimally the first bit of each sub-problem. This can be the case if deciding the first bits of the SAT instances is difficult due to a computational barrier in evaluating value estimates. This assumption ensures that each sub-problem necessitates some degree of search to be solved.

Assumption (5) stipulates that any value function in the prior is monotonic. Monotonicity is a reasonable property, as any optimal value function for SAT instances must be monotonic.

Interpretation of Theorem 26 Theorem 26 highlights a key limitation of relying on the Bellman equation to refine a prior over value functions: when faced with aggregated problems that are not directly solved by the prior, the resulting problem can become computationally intractable. For the algorithm, the expected running time grows at least exponentially with the number of aggregated sub-problems.

This issue lies in the inability of the Bellman equation to effectively decompose the aggregated problem, thereby hindering the learning of an optimal value function that could guide the search.

The relevance of our result, and its interpretation as a limitation, are based on the intuition that a "good" algorithm should not struggle from the aggregation of several sub-problems. Ideally, solving an aggregated problem should only incur a modest computational overhead compared to solving each problem independently.

In contrast, SAT solvers based on resolution techniques of Section 1.1 [Silva and Sakallah, 1996, Biere et al., 2009, Knuth, 2015], straightforwardly leverage the structure of an aggregated problem, and thus do not suffer

from the identified issue. We provide a formal statement for this observation in Appendix 4.2.

5.3 Limitation on the Bellman Equation with Universal Value Functions

Provided a goal to achieve, a common approach in RL is to sample trajectories using some initial exploration policy, then improve the policy by reinforcing behaviors that lead to the goal. However, there is a common pitfall with this approach: the goal can be hard to achieve with the initial policy, leading to a lack of feedback to improve the policy.

To address this challenge, Hindsight Experience Replay (HER) was introduced [Andrychowicz et al., 2017]. HER deviates from traditional RL by learning a *universal value function* rather than focusing solely on the task-specific value function. A typical use of universal value functions is to estimate the reachability between states, predicting whether any particular state b can be reached from any other state a. Then to reach a desired goal-state g, actions with high estimates of reaching g are taken.

However, despite HER's design to create feedback in challenging environments with sparse rewards, it can still overlook critical information in the problems. Moreover, under assumptions, the issue is independent of the prior used over universal value functions. Consequently, like classical value functions, improving the prior does not address the core limitation.

To prove this limitation, we construct a counterexample similar to the one presented in the previous section. In this counterexample, a search guided by classical value functions learned using the Bellman equation struggles to find a solution. Furthermore, learning a universal value function in this scenario leverages no more feedback than learning a classical value function, leading to the same struggles in finding a solution.

5.3.1 The Algorithm: Hindsight Experience Replay

As in the previous section, we work with CNF-SAT instances. We model the problem as taking a sequence of n binary decisions with associated states, that incrementally build a candidate solution. To provide a clear goal-state for HER, we introduce a final step with two possible outcomes: True or False. The state True indicates that the constructed candidate satisfies the problem, while False signifies failure. Thus, the target goal-state for HER is $g = \text{True}$.

The state space S for a problem p with n variables is defined as $\{0,1\}^{[n]} \cup \{\text{True}, \text{False}\}$, and the set of actions is binary: $\{0,1\}$.

Definition 27. *Given a CNF-SAT instance p with n variables, its* dynamics D^p *is as follows:*

- *Start in the initial state* $[]$.
- *At each step (up to n), append the chosen binary action to the current state, forming a sequence of actions.*
- *For a state of length n represented by the binary vector x, transition to* $\text{Check}(x; p)$.

The operator $D^p : S \times \{0,1\} \to S$ implements that dynamics for instance p, taking a state and a binary action as input and outputting the next state.

We now define universal value functions estimating state-to-state reachability.

Definition 28. *A* universal value function *takes as input a CNF-SAT instance p, two states in S, and outputs a binary value. Moreover, it satisfies $v(s, s; p) = 1$ for any state s.*

Definition 29. *A universal value function is* optimal *if for any CNF-SAT instance p with n variables and any $s_1, s_2 \in S$, $v(s_1, s_2; p) = 1$ iff there exists a sequence of actions from s_1 that leads to s_2 under the instance's dynamics (Definition 27).*

As with classical value functions, the Bellman equation can be enforced to learn optimal universal value functions. For a universal value function v, an instance p, distinct states a and b, and the dynamics operator D^p, the Bellman equation is:

$$v(a, b; p) = \max\{v(D^p(a, 0), b; p),\ v(D^p(a, 1), b; p)\}. \tag{5.3}$$

Algorithm 5 implements this equation. It is initialized with a set of universal value functions representing a prior for SAT-solving. The algorithm iteratively samples candidate solutions with this prior while refining it with the Bellman equation.

There are numerous pairs of states a, b with which Equation 5.3 can be enforced. Algorithm 5 enforces the equation for pairs of states sampled in

Algorithm 5 Learning universal value functions estimating state-to-state reachability with the Bellman equation (Hindsight Experience Replay) for CNF-SAT.
Candidate solutions are iteratively sampled guided by a hypothesis set V of universal value functions to reach a given goal-state g. Simultaneously, the Bellman equation is enforced on V for pairs of sampled states and g. The sampling process is performed according to Definition 30.

Inputs: V^0 : an initial set of value functions; p : a CNF-SAT instance to solve with n variables; g : a final goal-state to reach

for $t \leftarrow 0, 1, \ldots$ **do**
  $s_0^t, s_1^t, \ldots, s_n^t, s_{n+1}^t \sim \pi^{V^t}(p, g)$
  if $s_{n+1}^t = g$ **then**
    output s_n^t
  $V^{t+1} \leftarrow V^t$
  for $i \leftarrow 1, \ldots, n$ **do**
    $V^{t+1} \leftarrow$
   $\{v \in V^{t+1} | \, v(s_i^t, g; p) = \max\{v(D^p(s_i^t, 0), g; p),\, v(D^p(s_i^t, 1), g; p)\}\}$
    for $j \leftarrow i+1, \ldots, n+1$ **do**
      $V^{t+1} \leftarrow$
   $\{v \in V^{t+1} | \, v(s_i^t, s_j^t; p) = \max\{v(D^p(s_i^t, 0), s_j^t; p),\, v(D^p(s_i^t, 1), s_j^t; p)\}\}$

the same sequence and with the goal-state g. This is a key feature of HER that we replicate.

To guide the construction of candidate solutions, a policy is derived from the universal value function hypotheses. This process follows a rule similar to Definition 23.

Definition 30. *Given a set V of universal value functions. The policy π^V maps a CNF-SAT instance p with n variables and a goal g to a distribution over sequences of states $s_0, \ldots, s_{n+1}$ in S.*

Define $V_{s,g,p}$ as the number of universal value functions in V evaluating to 1 for inputs s, g, and p: $|\{v \in V | \, v(s, g; p) = 1\}|$.

The sequence of states is determined by the dynamics and the actions sampled at each state. The probability of taking action 0 at some state s is

$$\frac{V_{D^p(s,0),g,p}}{V_{D^p(s,0),g,p} + V_{D^p(s,1),g,p}}, \tag{5.4}$$

and the probability of action 1 is the complementary. If the denominator is zero,

both actions are taken with equal probabilities.

5.3.2 Counterexamples

We now define a monotonicity property for universal value functions, it parallels the one introduced earlier for classical value functions.

Definition 31. *A universal value function v is* monotonic *if, for any CNF-SAT instance p with n variables, any state $s \in S$, any $x \in \{0,1\}^n$, and any $i, j \in [n]$ with $i < j$, v satisfies $v(x_{\leq i}, s; p) \geq v(x_{\leq j}, s; p)$.*

Theorem 32. *Let p be a CNF-SAT instance with n variables, constructed as an aggregation of CNF-SAT instances $p_1, \ldots, p_K$ with index lists $I_1, \ldots, I_k, \ldots, I_K$, where the first element of each I_k is k.*

Let V and $V_1, \ldots, V_K$ represent sets of universal value functions and $g = \text{True}$ denote the goal-state.

Assume the following:

1. *The set V factorizes into $V_1, \ldots, V_K$; that is, $v \in V$ iff there exists $v_1 \in V_1, \ldots, v_K \in V_K$ such that, for all $x \in \{0,1\}^n$ and all $i \in [n]$, $v(x_{\leq i}, g; p) = \prod_{k \in [K]} v_k(x_{I_k \cap [i]}, g; p)$.*
2. *For an optimal value function v^*, and for all $k \in [K]$, either $v^*(0; p_k) = 1$ or $v^*(1; p_k) = 1$.*
3. *For all $k \in [K]$ and $v \in V_k$, either $v(0, g; p_k) = 1$ or $v(1, g; p_k) = 1$.*
4. *For an optimal value function v^*, and for all $k \in [K]$, if $v^*(0; p_k) = 1$, then $\pi^{V_k}(s_0 = 0; p_k) \leq \pi^{V_k}(s_0 = 1; p_k)$; otherwise, $\pi^{V_k}(s_0 = 0; p_k) \geq \pi^{V_k}(s_0 = 1; p_k)$.*
5. *Any $v \in V$ is monotonic.*
6. *For any $v \in V$, $i, j \in [n]$ with $i \leq j$, and any $x_1 \in \{0,1\}^i$, $x_2 \in \{0,1\}^j$, $v(x_1, x_2; p) = v^*(x_1, x_2; p)$ where v^* is an optimal universal value function.*
7. *For an optimal universal value function v^* and any state $s \in S$, if $v^*(s, \text{False}; p) = 1$, then, for any $v \in V$, $v(s, \text{False}; p) = 1$.*

Under these assumptions, Algorithm 5 initialized with $V^0 = V$, $p = p$, and $g = g$, runs for an expected time of at least 2^{K-1} steps.

The proof is similar to the one of Theorem 26 and is provided in Appendix 4.1.

Theorem 32 relies on similar assumptions to Theorem 26. Assumptions (1), (3), (4), and (5) are adapted to universal value functions, while assumption (2) remains unchanged. Assumptions (6) and (7) are newly introduced.

Assumption (6) ensures that every universal value function in the initial set V^0 accurately predicts which partial candidate solutions can be reached from given partial candidate solutions. This condition is satisfied by any optimal value function and is independent of the specific problem being solved.

Assumption (7) requires that any value function in V^0 can determine whether a partial candidate must lead to a full solution, regardless of subsequent actions. This condition is also satisfied by any optimal universal value function.

Theorem 32 demonstrates that an application of HER with universal value functions for state-to-state reachability does not circumvent the issue identified in the previous section regarding the Bellman equation with classical value functions.

5.4 Discussion

Summary In this chapter, we defined two algorithms for problem-solving with learning-based guidance in their search processes, Algorithms 4 and 5. Both of these algorithms represent an approach of the RL literature, they enforce the Bellman equation on sampled states, one to learn a value function, the other a universal value function predicting state-to-state reachability (with Hindsight Experience Replay).

To analyze these algorithms, we constructed counterexample problems that, by design, have a clear structure to exploit, yet remain provably challenging for the respective algorithms, Theorems 26 and 32. Our proofs demonstrate that these algorithms struggle because they do not leverage rich feedback from their failed attempts to improve their guidance.

Limitations One limitation of our theoretical analysis is its reliance on counterexamples, which, by nature, are specific problems. This raises the question: do these counterexamples reflect broader limitations on the studied algorithmic approaches with practical problems?

We argue that they do. Our counterexamples and analysis are based on aggregating sub-problems into a larger problem, an expected structure in practical applications. Also, although we use SAT instances to build

minimal clean counterexamples, our analysis is not tied to this problem, and we believe that the results generalize to other domains.

A second limitation of our theoretical analysis is its reliance on specific algorithmic implementations. While our algorithms are straightforward implementations of each approach, they may inadvertently introduce specific implementation choices not part of the original approaches and limit the scope of our findings.

For example, our algorithms use Bayesian Learning by iteratively updating a set of hypotheses according to the sampled data. While Bayesian Learning is an idealized framework to describe a priori information and learning, it is not a common choice for practical Deep RL algorithms.

Conclusion Despite the use of specific algorithms and counterexamples, we believe our findings highlight a broad practical limitation on algorithms that rely on the Bellman equation enforced at sampled states to learn (universal) value functions. These include model-free Deep RL methods like Deep Q-learning [Mnih et al., 2013] and actor-critic methods [Schulman et al., 2017], as well as Hindsight Experience Replay for state-to-state reachability [Andrychowicz et al., 2017] and some model-based methods like AlphaZero [Silver et al., 2018].

Our work formalizes inefficiencies in these classical algorithmic ideas of RL. These insights can inform the development of novel algorithms better leveraging learning to accelerate search, leading to more effective problem-solving techniques. In the next section, we further discuss these results and provide an interpretation on their meaning for the field of RL.

6
Discussion

Chapters 4 and 5 present strong limitations on some approaches: model-free RL methods; model-based RL methods leveraging the model only as a black-box generator of transitions; and universal value functions evaluating state-to-state reachability. We first propose an intuitive interpretation of our results. Then, in search of always better algorithms, we return to the literature presented in Chapter 1 to look at methods that avoid the presented limitations but argue that these methods inherit other issues. Finally, we conclude by discussing some open problems.

6.1 An interpretation

Many reinforcement learning (RL) algorithms share a common objective: learning an optimal value function. The Bellman equation, shown in Figure 6.1, provides a classical framework for generating input-output data from sampled trajectories to enable this learning process.

However, as demonstrated in Chapter 4, this method can be inefficient. Algorithms that incorporate insights from a (learned) system dynamics can find more quickly an optimal policy.

Further, Chapter 5 shows that even exploiting state-to-state dynamics may not suffice. Specifically, limitations persist even when the dynamics remains fixed across different tasks. Leading to the question: which additional information should RL algorithms leverage to overcome this barrier?

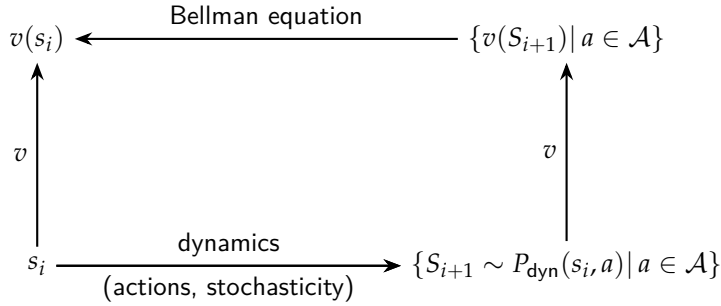

Fig. 6.1 The Bellman equation ensures that this diagram commutes: the value of a state s_i at time step i, denoted $v(s_i)$, can be computed based on the values of successor states reached through the dynamics (P_{dyn}) and available actions ($\mathcal{A}$). This formulation allows the value function v to be learned by fitting it to input–output pairs of the form $(s_i, v(s_i))$ that satisfy the Bellman equation. However, our work formalizes the intuition that relying solely on such input–output data overlooks valuable knowledge. Specifically, it ignores information embedded in the system dynamics and the computational model of the value function, both of which could be leveraged to improve learning.

Interestingly, the algorithm with superior performance in our analysis (a CDCL SAT solver) can itself be interpreted as learning a value function since the set of learned constraints act analogously to a value function. Unlike classical value learning algorithms, these constraints are derived through deductions based on the activation of other constraints in subsequent states. In other words, this solver exploits the computational structure underlying the value function.

Thus, while many RL algorithms leveraging Machine Learning treat the mapping from state s_i to value $v(s_i)$ as a black-box for generating training data, our findings suggest this approach may be suboptimal. Instead, algorithms that integrate models of both the dynamics and the value function can extract richer insights and significantly accelerate learning.

6.2 Algorithms avoiding the limitations

There exist several types of methods that are not limited by the results of Chapters 4 and 5, and are thus potentially free of the flaws highlighted in this work. We identify the following algorithmic approaches with that

property in the literature.

Classical/symbolic solvers and planning. Our limitation does not apply to symbolic methods for SAT-solving of Section 1.1, or other algorithms built on top of them, such as model-based approaches leveraging symbolic solvers for planning. These algorithms receive or construct a symbolic or semi-parametric model of the dynamics and then apply a classical planning algorithm [Russell and Norvig, 2010, Konidaris et al., 2018]. These methods circumvent the limitations, as the symbolic planner does not treat the model solely as a black-box generator of transitions, but rather leverages insights from it.

In Chapter 4, the method presented in the numerical experiments of Section 4.4 is an implementation of this idea since it decomposes its learned model of the dynamics into structured constraints for planning. In Chapter 5, we showed in Section 4.2 that a symbolic SAT solver does not have the limitation.

While these approaches circumvent the studied limitations, it is unclear if they can be a complete solution to the problem of leveraging Machine Learning (ML) for problem-solving. The symbolic algorithm itself relies on deductive reasoning and does not use inductive learning. ML can be used to learn a model of the dynamics, but is not used directly for planning.

We observe an exception in the literature to this rule: some algorithms can leverage these solvers in a learned abstract space, as proposed in [Kurutach et al., 2018, Asai et al., 2022]. In that case, ML can potentially simplify the initial problem of planning directly in the ground state space to planning in a potentially simpler abstract space.

Backpropagation through a learned model. The algorithms learn a smooth model of the dynamics, then use backpropagation through the learned model for planning or learning a policy/value function. We refer to the corresponding literature in Section 1.2.2. Similarly to the last example, these methods elude our limitations because the learned model is not treated as a black-box by the algorithm.

However, we note that these methods are mostly used in environments where the dynamics are approximately smooth, such as low-level control in robotics.

Goal-conditioned algorithms (universal value functions). Goal-conditioned algorithms of Section 1.2.3, such as universal value functions (UVFs), can

be used to evaluate and perform state-to-state reachability. These algorithms bypass our Definition 19 of Chapter 4 because the universal value functions take as input the states in the future of the trajectory, and the quantities they learn cannot be replaced trivially by a generator of transitions.

While they avoid the class and limitation defined in Chapter 4, they suffer from the limitation described in Chapter 5 in the case where state-to-state reachability is used. This does not pose an inherent limitation on UVFs since they can still be used in another space than the ground state space, but it opens an unanswered question: in which space should UVFs be used to avoid the limitations?

We also mention here the algorithms of Section 1.2.4 following a divide-and-conquer approach to Automated Theorem Proving (ATP). Similarly to goal-conditioned algorithms, they can leverage the decomposition of a problem into subgoals to enrich the learning signal. However, the approach relies on a human prior for the decomposition.

Learning (part of) the algorithm. Algorithms of Section 1.3 that learns algorithms using auxiliary tasks and data, can also evade our limitations. If the space of learnable algorithms is sufficiently expressive, the learning procedure could potentially learn an algorithm not subject to the limitations.

To take one example, Oh et al. [2020] proposes to learn from a set of tasks a new loss in place of the Bellman equation. This new loss could potentially augment the feedback received by the learning algorithm in the event of failure to generate a solution, thereby circumventing our limitations. A second example is the use of foundation models directly mapping problems to solutions with neural networks, where the learned algorithm is only constrained by the Deep Learning architecture, the data, and the learning algorithm used. In that case, the final algorithm could also evade the limitations.

Learning (part of) the algorithm is tempting. The approach can potentially learn a better solution than a human researcher directly coding the algorithm himself. However, there is a trade-off since the finally learned algorithm can be brittle outside of its training distribution.

Conclusion Our theoretical and numerical results suggest that some ideas present in these algorithms could help solve problems otherwise intractable

for a large class of classical methods. However, we believe that in their current form, these algorithms do not provide a compelling general solution to the problem of leveraging ML for problem-solving.

6.3 Open problems

To finish this manuscript, we wanted to discuss two open problems that we believe the answers to would be important intermediate steps toward efficient general-purpose algorithms leveraging learning to solve problems.

Open problem 1: What is the role of learning by induction in problem-solving? In other words, what is the plus-value of ML in problem-solving?

It seems widely accepted that inductive learning is useful for problem-solving. ML is at least useful when the problem is not completely defined a priori, and data has to be used instead. Additionally, ML can also be useful even when the problem is completely defined a priori, for example, by a model of the dynamics, such as in the game of GO or in ATP. In that case, ML can be used to build a policy, value function, universal value function, etc., to help find a solution.

However, for this last case, where the problem is already completely defined without the need for data, it seems unclear when and how ML should be leveraged to perform better than a symbolic solver/planner based on deduction.

One way to answer this question would be to propose minimal SAT instances that are hard to solve with a symbolic method of Section 1.1, but could be tractably solved by leveraging ML (using existing or new methods).

Ideally, answering this question would define a set of necessary properties that an algorithm leveraging ML for problem-solving must have.

Open problem 2: Designing new algorithms leveraging ML working efficiently on problems with sparse rewards but strong prior. As discussed in the introduction of Chapter 5, solving RL problems with sparse rewards and no specific prior is hard for any algorithm. However, if there is a strong prior, we should expect a good algorithm to be able to leverage it.

Our results used SAT solving to construct minimal examples where such sparse rewards but strong priors can be found. The prior allows

for decomposing the problem into constraints, this information can then be leveraged to learn during the search for a solution (like symbolic SAT solvers).

However, classical RL algorithms fail to fully leverage this information, and the algorithms of the literature discussed above also do not seem to properly address this issue. Letting the problem of designing such efficient algorithms an open question.

To create these new algorithms, we propose that SAT problems should be used as a north star that researchers have in mind. Can my algorithm make progress on a SAT instance despite the lack of rewards from random exploration? Can it leverage the decomposition of the problem into constraints?

Moreover, using the answer to the previous open problem: Does my algorithm enjoy the advantages that ML can bring to problem-solving?

Abstract SAT One possible answer to the first open problem is that ML can help solve constraint-satisfaction tasks defined over high-level, abstract objects. For example, one might wish to generate an image that meets specified requirements about the objects it depicts and their spatial relationships (e.g., "the fruit must rests on the table"). Attempting to encode such constraints directly over raw pixel values and then invoking a traditional SAT solver is hopeless. Instead, a learning component could help an algorithm to reason over the underlying abstract objects.

To illustrate this idea in a simpler, synthetic setting, consider the task of finding a binary vector of length n that satisfies a collection of CNF clauses expressed not on the individual bits, but on the parity of various subsets of those bits. Although the resulting CNF formula could be solved easily by a CDCL SAT solver, translating each parity constraint back into relationships between the original bits would create a difficult instance. Here, ML could assist by leveraging the structure, enabling efficient reasoning over the abstract parity variables rather than the raw bit assignments.

A

Appendix of Chapter 2

1.1 Main proofs

The following is a simple adaptation of a PAC-learning theorem over a finite hypothesis space [Shalev-Shwartz and Ben-David, 2014].

Proposition 7 (Description-length PAC-guarantee)**.** *There exists constants $a_1, a_2 > 0$ such that the following holds.*

For any interpreter φ and associated learning algorithm MDL^{φ}, any $(f, \mathcal{P})$ learning problem and any PAC-learning parameters $\epsilon \in (0, 1/2)$, $\delta \in (0, 1)$, MDL^{φ} has an (ϵ, δ)-PAC-learning performance with an m-sample dataset on the learning problem, where

$$m = \frac{a_1}{\epsilon}\left[\log\frac{1}{\delta} + |f|_{\varphi} + a_2\right]. \tag{2.5}$$

Proof. The algorithm MDL^{φ} never outputs a function with a description-length larger than $|f|_{\varphi}$. There are at most $I = \sum_{i=0}^{|f|_{\varphi}} 2^i = 2^{|f|_{\varphi}+1} - 1$ functions in this set.

Let's compute an upper-bound on the probability that there exists a function $\tilde{f}$ of description-length smaller than $|f|_{\varphi}$ such that $acc_f^{\mathcal{P}}(\tilde{f}) < 1 - \epsilon$ and which is consistent with a dataset composed of $m \geq \frac{1}{\epsilon}\left[\log\frac{I}{\delta}\right]$ samples.

For any function $\tilde{f}$ with $acc_f^{\mathcal{P}}(\tilde{f}) < 1 - \epsilon$ the probability to be consistent with the dataset is upper-bounded by $(1-\epsilon)^m$ since each sample is drawn independently according to $\mathcal{P}$.

By the union bound the probability of the existence of one low accuracy function consistent with the data is thus upper-bounded by $I(1-\epsilon)^m$.

Which develops in $I(1-\epsilon)^m \leq Ie^{-\epsilon m} \leq \delta$.

Moreover, there exists constants $a_1, a_2 > 0$ s.t

$$\frac{1}{\epsilon}\left[\log\frac{I}{\delta}\right] \leq \frac{a_1}{\epsilon}\left[\log\frac{1}{\delta} + |f|_\varphi + a_2\right] \tag{A.1}$$

independently of the interpreter, learning problem, and PAC-learning parameters. □

Theorem 11. *There exists a constant $q \in \mathbb{R}^+$ such that for all $\epsilon \in (0, 1/2)$, $\delta \in (0,1)$ and $n, d \in \mathbb{N}^+$,*

$$\tilde{G}^d_{\mathcal{C}\to\mathcal{U}}(\epsilon, \delta, n) \leq q. \tag{2.9}$$

Proof. From Definition 10 of $\tilde{G}$

$$\begin{aligned} \tilde{G}^d_{\mathcal{C}\to\mathcal{U}}(\epsilon,\delta,n) = \sup_{f\in H^n, \mathcal{P}\in\Delta(\mathcal{B}^n)} & \frac{m^{\epsilon,\delta}_{\mathcal{U}}(f,\mathcal{P})}{\frac{a_1}{\epsilon}(\log\frac{1}{\delta} + |f|_{\mathcal{C}} + a_2)} \\ \text{subject to} & \quad |f|_{\mathcal{C}} \leq n^d. \end{aligned} \tag{A.2}$$

For any n, consider any learning problem $(f \in H^n, \mathcal{P} \in \Delta(\mathcal{B}^n))$.

By the Definition 8 of $m^{\epsilon,\delta}_\varphi$ and the PAC-guarantee given in Proposition 7, for any $\epsilon \in (0, 1/2)$ and $\delta \in (0,1)$, we have

$$m^{\epsilon,\delta}_{\mathcal{U}}(f,\mathcal{P}) \leq \frac{a_1}{\epsilon}(\log\frac{1}{\delta} + |f|_{\mathcal{U}} + a_2). \tag{A.3}$$

By the main theorem of Kolomogorov complexity, Proposition 64, we have $|f|_{\mathcal{U}} \leq |f|_{\mathcal{C}} + K$ for some constant K independent of f. The PAC-learning guarantee for $MDL^{\mathcal{C}}$ can thus be transformed in a guarantee for $MDL^{\mathcal{U}}$ since

$$\frac{a_1}{\epsilon}(\log\frac{1}{\delta} + |f|_{\mathcal{U}} + a_2) \leq \frac{a_1}{\epsilon}(\log\frac{1}{\delta} + |f|_{\mathcal{C}} + K + a_2). \tag{A.4}$$

Following these inequalities, we get

$$\tilde{G}^d_{\mathcal{C}\to\mathcal{U}}(\epsilon,\delta,n) \leq \frac{\frac{a_1}{\epsilon}(\log\frac{1}{\delta}+|f|_{\mathcal{C}}+K+a_2)}{\frac{a_1}{\epsilon}(\log\frac{1}{\delta}+|f|_{\mathcal{C}}+a_2)} \leq \frac{K}{a_2}+1. \tag{A.5}$$

□

Theorem 12. *For all $\epsilon \in (0,1/2), \delta \in (0,1), d \in \mathbb{N}^+$ we have*

$$G^d_{\mathcal{U}\to\mathcal{C}}(\epsilon,\delta,n) \in \Omega(2^n/n). \tag{2.10}$$

Proof. By Definition 9 of the sample efficiency gains, we have

$$\begin{aligned} G^d_{\mathcal{U}\to\mathcal{C}}(\epsilon,\delta,n) = \sup_{f\in H^n,\mathcal{P}\in\Delta(\mathcal{B}^n)} \quad & \frac{m^{\epsilon,\delta}_{\mathcal{C}}(f,\mathcal{P})}{m^{\epsilon,\delta}_{\mathcal{U}}(f,\mathcal{P})} \\ \text{subject to} \quad & |f|_{\mathcal{U}} \leq n^d. \end{aligned} \tag{A.6}$$

For any $\epsilon \in (0,1/2), \delta \in (0,1), d \in \mathbb{N}^+$, we define a sequence of learning problems which prove the statement. For any n, we define a learning problem to solve. For any n, the learning problem is to learn under the uniform distribution, U, the binary function computed by the interpretation of the following program.

For input x of size n:

1. *Compute the input size n.*
2. *Enumerate all the functions in H^n in some fixed lexicographic order. For each of these functions, f:*
   (a) *Compute the hypothesis of minimal-description-length according to the interpreter $\mathcal{C}$ to represent a function $\hat{f}$ such that $acc^U_f(\hat{f}) \geq 1-\epsilon$. Rewritten*
   $$\begin{aligned} \min_{h\in\mathcal{B}^*} \quad & |h| \\ \textit{subject to} \quad & acc^U_f(\mathcal{C}(h,.)) \geq 1-\epsilon. \end{aligned} \tag{A.7}$$
   (b) *With $H(.)$ the binary entropy function, see Definition 69, if the optimal description-length is bigger than $2^n(1-H(\epsilon))-2$ then return $f(x)$.*

We first show that this is a well-defined computable function in the sense that the step *(b)* will always be satisfied for some function in the enumeration for all n sufficiently large.

The next development shows by a counting argument, that all the binary functions cannot be approximated within $1-\epsilon$ by Boolean circuits of binary description-length smaller than $2^n(1-H(\epsilon))-2$.

The number of functions that can approximated with accuracy higher than $1-\epsilon$ by Boolean circuits of description-length smaller than $2^n(1-H(\epsilon))-2$ is

$$\left| \bigcup_{l\in 0,\ldots,\lfloor 2^n(1-H(\epsilon))-2\rfloor} \bigcup_{h\in\mathcal{B}^l} \left\{ f\in H^n \middle| \, acc_f^U(\mathcal{C}(h,.)) \geq 1-\epsilon \right\} \right|. \tag{A.8}$$

Using Proposition 70 and the fact that there are 2^{2^n} functions in H^n,

$$\begin{aligned}
&\left| \bigcup_{l\in 0,\ldots,\lfloor 2^n(1-H(\epsilon))-2\rfloor} \bigcup_{h\in\mathcal{B}^l} \left\{ f\in H^n \middle| \, acc_f^U(\mathcal{C}(h,.)) \geq 1-\epsilon \right\} \right| \\
&\quad \leq \sum_{l\in 0,\ldots,\lfloor 2^n(1-H(\epsilon))-2\rfloor} \sum_{h\in\mathcal{B}^l} \left| \left\{ f\in H^n \middle| \, acc_f^U(\mathcal{C}(h,.)) \geq 1-\epsilon \right\} \right| \\
&\quad\quad = \sum_{l\in 0,\ldots,\lfloor 2^n(1-H(\epsilon))-2\rfloor} \sum_{h\in\mathcal{B}^l} \sum_{i\in 0,\ldots,\lfloor \epsilon 2^n\rfloor} \binom{2^n}{i} \\
&\quad\quad\quad \leq \sum_{l\in 0,\ldots,\lfloor 2^n(1-H(\epsilon))-2\rfloor} \sum_{h\in\mathcal{B}^l} 2^{2^n H(\epsilon)} \\
&\quad\quad\quad\quad \leq 2^{2^n(1-H(\epsilon))-1} 2^{2^n H(\epsilon)} \\
&\quad\quad\quad\quad\quad = 2^{2^n-1} < 2^{2^n} = |H^n|.
\end{aligned}$$

Thus the condition in step *(b)* will always be satisfied for some function.

Also, the condition $|f|_{\mathcal{U}} \leq n^d$ will always be satisfied for all n large enough since the learning problem's function corresponds to a program/Turing-machine of fixed description-length.

Moreover, using the guarantee of Proposition 7, $m_{\mathcal{U}}^{\epsilon,\delta}(f,\mathcal{P}) \leq \frac{a_1}{\epsilon}(\log\frac{1}{\delta} + |f|_{\mathcal{U}} + a_2)$, we deduce that $m_{\mathcal{U}}^{\epsilon,\delta}(f,\mathcal{P})$ is upper-bounded by a constant independent of n.

Also by construction, $MDL^{\mathcal{C}}$ has to select circuits of description-length growing at least as fast as $2^n(1-H(\epsilon))-2$ to be able to approximate the function to learn with an average error at most ϵ. We show that this condition has implications on the size of the minimal circuit that has to be selected and then on the minimal number of samples needed.

By Definition 52, for all n sufficiently large, all circuits of size lower than

$\alpha 2^n/n$, for any $\alpha > 0$, can be described with the Boolean circuit interpreter $\mathcal{C}$ such that their description-length is lower than

$$9\alpha \frac{2^n}{n} \log(\alpha 2^n/n). \tag{A.9}$$

Thus, for all n sufficiently large, the description-lengths of these circuits are lower than

$$9\alpha \left[\log(2) 2^n + \log(\alpha) \frac{2^n}{n} - \frac{2^n}{n} \log n \right] \leq 9\alpha \log(2\alpha) 2^n. \tag{A.10}$$

Thus there exists a α sufficiently small such that, for all n sufficiently large, the description-lengths of these circuits are smaller than $2^n(1 - H(\epsilon)) - 2$.

Consequently, all these circuits with size at most $\alpha 2^n/n$ must be eliminated by $MDL^{\mathcal{C}}$. This requires at least $\frac{C}{b} 2^n/n$ samples, for some fixed $b > 0$, by Proposition 68. □

Theorem 13. *For all $\epsilon \in (0, 1/2)$, $\delta \in (0,1)$, $c, d \in \mathbb{N}^+$ we have*

$$\tilde{G}^d_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) \in O(n^c \log^2 n). \tag{2.11}$$

Proposition 33. *Pippenger and Fischer [1979], Schnorr [1976]. If a multi-tape Turing machine M computes a function on inputs of size n within t steps then there exists a Boolean circuit of size at most $\alpha(\text{number of rules of } M) t \log t$ that computes the same function, where α depends only on the number of tapes and the alphabet size of the Turing machine.*

Proof of Theorem 13. By Definition 10 of $\tilde{G}$

$$\tilde{G}^d_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) = \sup_{f \in H^n, \mathcal{P} \in \Delta(\mathcal{B}^n)} \frac{m^{\epsilon,\delta}_{\mathcal{C}}(f, \mathcal{P})}{\frac{a_1}{\epsilon}(\log \frac{1}{\delta} + |f|_{\mathcal{U}^c} + a_2)} \quad \text{subject to} \quad |f|_{\mathcal{U}^c} \leq n^d. \tag{A.11}$$

We will upper-bound the ratio for any learning problem ($f \in H^n, \mathcal{P} \in \Delta(\mathcal{B}^n)$). Note that only functions f corresponding to finite $|f|_{\mathcal{U}^c}$ have to be considered.

We use Proposition 49 on the Turing machine of the interpreter $\mathcal{U}^c$ and the input h of length $|f|_{\mathcal{U}^c}$ such that $\mathcal{U}^c(h, .) = f$. From the application of the proposition, we deduce that there exists a Turing machine with at most

$\rho\,|f|_{\mathcal{U}^c}$ rules that compute f, where $\rho > 0$ is a parameter independent of f and n. Moreover, the proposition tells us that the resulting Turing machine has the same computational time-limit as $\mathcal{U}^c$, and is thus also bounded by βn^c, for some $\beta > 0$.

Applying now Proposition 33 on these facts, we deduce that there exists a Boolean circuit computing f of size at most, with $t = \beta n^c$,

$$\alpha\rho\,|f|_{\mathcal{U}^c}\, t \log t, \tag{A.12}$$

where $\alpha > 0$ is independent of f and n.

By Definition 52, the description-length of this circuit with the Boolean circuit interpreter $\mathcal{C}$, and thus $|f|_{\mathcal{C}}$, will be upper bounded by

$$2\lceil \log_2(n)\rceil + 2 + \underbrace{\alpha\rho\,|f|_{\mathcal{U}^c}\, t \log t(3 + \max\{2\lceil \log_2 \alpha\rho\,|f|_{\mathcal{U}^c}\, t \log t\rceil), \lceil \log_2 n\rceil\}}_{\stackrel{\text{def}}{=} B}. \tag{A.13}$$

We use the PAC-guarantee of Proposition 7 to upper-bound $m_{\mathcal{C}}^{\epsilon,\delta}(f, \mathcal{P})$ with the upper-bound on $|f|_{\mathcal{C}}$ of Equation A.13.

Then the quantity of Equation A.11 is also upper-bounded

$$\begin{aligned}
\tilde{G}_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) &\leq \sup_{f \in H^n} \frac{\frac{a_1}{\epsilon}(\log \frac{1}{\delta} + 2\lceil \log_2(n)\rceil + 2 + B + a_2)}{\frac{a_1}{\epsilon}(\log \frac{1}{\delta} + |f|_{\mathcal{U}^c} + a_2)} \text{ s.t. } |f|_{\mathcal{U}^c} \leq n^d, \\
&\leq \sup_{f \in H^n} \frac{B}{|f|_{\mathcal{U}^c}} + \frac{2}{a_2}\lceil \log_2(n)\rceil + \frac{2}{a_2} + 1 \text{ s.t. } |f|_{\mathcal{U}^c} \leq n^d.
\end{aligned} \tag{A.14}$$

Where $\frac{B}{|f|_{\mathcal{U}^c}}$ equals

$$\alpha\rho t \log t(3 + \max\{2\lceil \log_2 \alpha\rho\,|f|_{\mathcal{U}^c}\, t \log t\rceil), \lceil \log_2 n\rceil\}. \tag{A.15}$$

Using $|f|_{\mathcal{U}^c} \leq n^d$ and $t = \beta n^c$, the value in Equation A.15 is in

$$O(n^c \log^2 n). \tag{A.16}$$

Returning this result to the inequalities of Equation A.14, we obtain

$$\tilde{G}_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) \in O(n^c \log^2 n). \tag{A.17}$$

□

We note that the Theorem 13 can be improved upon by using another definition for the polynomial-time universal Turing machine, $\mathcal{U}^c$. The Definition 47 use the construction of Hennie and Stearns [1966]. Another possibility is to use the construction of Pippenger and Fischer [1979] related to Proposition 33. This construction would make $\mathcal{U}^c$ oblivious and directly transformable into a Boolean circuit without the additional $\log n$ factor. The final bound would thus be in $O(n^c \log n)$ instead of $O(n^c \log^2 n)$.

Theorem 14. *For all $\epsilon \in (0, 1/2)$, $\delta \in (0,1)$, $1 < c, d \in \mathbb{N}^+$ and all $\gamma > 0$ we have*

$$G^d_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) \in \Omega(n^{1-\gamma}). \tag{2.12}$$

Proof. By Definition 9, the sample efficiency gain is

$$G^d_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) = \sup_{f \in H^n, \mathcal{P} \in \Delta(\mathcal{B}^n)} \frac{m^{\epsilon,\delta}_{\mathcal{C}}(f, \mathcal{P})}{m^{\epsilon,\delta}_{\mathcal{U}^c}(f, \mathcal{P})} \quad \text{subject to} \quad |f|_{\mathcal{U}^c} \leq n^d. \tag{A.18}$$

Fix any combination of $\epsilon \in (0, 1/2), \delta \in (0,1), 1 < c, d \in \mathbb{N}^+$.

For any input size n, we contruct the following learning problem, $(f \in H^n, \mathcal{P} \in \Delta(\mathcal{B}^n))$. The function to learn f is the parity function on $\mathcal{B}^n$, noted $\oplus$ (it is the number of 1 in the input modulo 2). The probability measure on the domain $\mathcal{B}^n$, $\mathcal{P}$, is the uniform distribution U^n. Let $\oplus|_n$ denote the function $\oplus$ restricted to size n inputs.

There exists a fixed Turing machine computing the parity function in linear time for all n. By Definition 47, for $c > 1$ there exists an $h \in \mathcal{B}^*$ such that $\mathcal{U}^c(h, .)$ computes $\oplus$ for all n large enough. Thus $|\oplus|_n|_{\mathcal{U}^c}$ is at most some constant.

A first consequence is that, for n large enough, $|\oplus|_n|_{\mathcal{U}^c} \leq n^d$ will be satisfied for the form of learning problem we defined.

A second consequence, using the PAC-guarantee given in Proposition 7, is that the denominator $m^{\epsilon,\delta}_{\mathcal{U}^c}(f, \mathcal{P}) \leq \frac{a_1}{\epsilon}(\log \frac{1}{\delta} + |\oplus|_n|_{\mathcal{U}^c} + a_2)$ is less than a constant.

Now for the numerator, for any $\gamma > 0$ take any sampling of size less than $n^{1-\gamma}$. By Proposition 68 and Definition 52, we know that a Boolean circuit of size less than $bn^{1-\gamma}$ will be selected by $MDL^{\mathcal{C}}$ for some fixed $b > 0$.

For n sufficiently large $bn^{1-\gamma} < n$, and thus the circuit selected by $MDL^{\mathcal{C}}$ will not depend on all the inputs' variables. Suppose without loss

of generality that x_n is one of these variables to which the circuit is not sensible. The accuracy of a function C computed by such a selected circuit on our learning problem is

$$\begin{aligned} acc^{U^n}_{\oplus}(C) &= \frac{1}{2^n}\sum_{x\in\mathcal{B}^n}\oplus(x) = C(x_1,\ldots,x_{n-1},x_n) \\ &= \frac{1}{2^n}\sum_{x^-\in\mathcal{B}^{n-1}}\sum_{x_n\in\mathcal{B}}\oplus(x^-,x_n) = C(x_1^-,\ldots,x_{n-1}^-) \\ &= \frac{1}{2^n}\sum_{x^-\in\mathcal{B}^{n-1}} 1 \\ &= \frac{1}{2}. \end{aligned} \tag{A.19}$$

Since $\epsilon < 1/2$, it is impossible for $MDL^{\mathcal{C}}$ to get a sufficient accuracy with only $n^{1-\gamma}$ samples for n sufficiently large, and so $m_{\mathcal{C}}^{\epsilon,\delta}(f,\mathcal{P}) \in \Omega(n^{1-\gamma})$. □

Theorem 15. *If there exists $\epsilon \in (0,1/2)$, $\delta \in (0,1)$, $c,d \in \mathbb{N}^+$ and some $\gamma > 0$ such that*

$$\tilde{G}^d_{\mathcal{U}^c\to\mathcal{C}}(\epsilon,\delta,n) \notin O(n^{1+\gamma}) \tag{2.13}$$

then there exists a language in $\mathbf{P}$ not computable by any sequence of Boolean circuits whose sizes are in $O(n^{1+\tau})$ for some $\tau > 0$.

Proof. Let's suppose superlinear gains $\tilde{G}^d_{\mathcal{U}^c\to\mathcal{C}}(\epsilon,\delta,n) \notin O(n^{1+\gamma})$ for some $\epsilon \in (0,1/2)$, $\delta \in (0,1)$, $c,d \in \mathbb{N}^+$ and $\gamma > 0$.

Recall that by Definition 10, for some constants a_1, $a_2 > 0$ defined in Proposition 7,

$$\begin{aligned} \tilde{G}^d_{\mathcal{U}^c\to\mathcal{C}}(\epsilon,\delta,n) = \sup_{f\in H^n,\mathcal{P}\in\Delta(\mathcal{B}^n)} \quad & \frac{m_{\mathcal{C}}^{\epsilon,\delta}(f,\mathcal{P})}{\frac{a_1}{\epsilon}(\log\frac{1}{\delta} + |f|_{\mathcal{U}^c} + a_2)} \\ \text{subject to} \quad & |f|_{\mathcal{U}^c} \leq n^d. \end{aligned} \tag{A.20}$$

We give the proof outline:

1. First, we prove that the premise of the theorem's statement implies the existence of an infinite sub-sequence of functions with description-length gains superlinear in the input-size. This result appears in Equation A.25.

2. Second, we present a fixed program of polynomial computational complexity.

3. Third, we show lower-bounds on some input-sizes for this program with the interpreter for circuits. These lower-bounds come from the developments in the first point.

4. Fourth, we show from these lower-bounds that the description-length of the program with the circuit interpreter is at least superlinear in the input-size.

5. Fifth, we conclude by showing superlinear circuit complexity for the presented program.

For any n, the sets H_n are finite and thus, for any n, the supremum can be attained for some $f_n \in H_n$. Then there exists a sequence of functions $(f_n) = f_1 \in H_1, \ldots, f_n \in H_n, \ldots$ such that the sequence in n

$$\sup_{\mathcal{P} \in \Delta(\mathcal{B}^n)} \frac{m_{\mathcal{C}}^{\epsilon,\delta}(f_n, \mathcal{P})}{\frac{a_1}{\epsilon}(\log \frac{1}{\delta} + |f_n|_{\mathcal{U}^c} + a_2)} \tag{A.21}$$

is not in $O(n^{1+\gamma})$, and, morevover, with $f_n \leq n^d$ for all n.

By contraposition of the PAC-guarantee offered by Proposition 7, for all n and $\mathcal{P}$, $\frac{a_1}{\epsilon}(\log \frac{1}{\delta} + |f_n|_{\mathcal{C}} + a_2) \geq m_{\mathcal{C}}^{\epsilon,\delta}(f_n, \mathcal{P})$, and thus

$$\frac{\frac{a_1}{\epsilon}(\log \frac{1}{\delta} + |f_n|_{\mathcal{C}} + a_2)}{\frac{a_1}{\epsilon}(\log \frac{1}{\delta} + |f_n|_{\mathcal{U}^c} + a_2)} \notin O(n^{1+\gamma}). \tag{A.22}$$

Since ϵ and δ are fixed, the sequence of function (f_n) satisfies

$$\frac{|f_n|_{\mathcal{C}}}{|f_n|_{\mathcal{U}^c}} \notin O(n^{1+\gamma}). \tag{A.23}$$

We now restrict n to the indices that form a sub-sequence of (f_n) such that

$$\frac{|f_n|_{\mathcal{C}}}{|f_n|_{\mathcal{U}^c}} \in \Omega(n^{1+\gamma/2}). \tag{A.24}$$

Let $N \subseteq \mathbb{N}$ denote the set of such indices. Notice that such a restriction remove any function f such that $|f|_{\mathcal{U}^c} = +\infty$ from the sequence.

By definition of the big-Ω notation, there exists some $b > 0$ such that

for all n sufficiently large

$$|f_n|_{\mathcal{C}} \geq b\,|f_n|_{\mathcal{U}^c}\,n^{1+\gamma/2}. \tag{A.25}$$

We will now define a Boolean function computable in polynomial-time and, using Equation A.25, prove a superlinear circuit complexity for it.

This Boolean function will be noted I and is defined by

$$I(\langle x_1, x_2\rangle) = \mathcal{U}^c(x_1, x_2), \tag{A.26}$$

where $\langle .,.\rangle : \mathcal{B}^* \times \mathcal{B}^* \to \mathcal{B}|\,\langle x_1, x_2\rangle = \underbrace{0\ldots0}_{|x_2|}1x_1x_2$. The encoding $\langle .,.\rangle$ is bijective and invertible in polynomial-time. Thus I is computable in polynomial-time since $\mathcal{U}^c$ is also computable in polynomial-time.

We prove a lower-bound on the description-length of I with the Boolean circuit interpreter on some inputs' sizes as we show next by contradiction. We denote $I|_{2n+1+|f_n|_{\mathcal{U}^c}}$ the Boolean function I restricted to $2n+1+|f_n|_{\mathcal{U}^c}$ sized-inputs. We will show $\left|I|_{2n+1+|f_n|_{\mathcal{U}^c}}\right|_{\mathcal{C}} \geq |f_n|_{\mathcal{C}}$.

Suppose there exists some Boolean circuit that computes $I|_{2n+1+|f_n|_{\mathcal{U}^c}}$, and that its description-length is strictly lower than $|f_n|_{\mathcal{C}}$, i.e. $\left|I|_{2n+1+|f_n|_{\mathcal{U}^c}}\right|_{\mathcal{C}} < |f_n|_{\mathcal{C}}$.

For all n, let $p_n \in \mathcal{B}^{|f|_{\mathcal{U}^c}}$ be the Boolean string such that $\mathcal{U}^c(p_n, .) = f_n$. By definition of I (Equation A.26) the function $I(\langle p_n, .\rangle)$ computes f_n, i.e. for all $x \in \mathcal{B}^n$ we have $I(\langle p_n, x\rangle) = f_n(x)$.

For the supposed circuit, we hardwire the $n+2$ to the $n+2+|f_n|_{\mathcal{U}^c}$ inputs' variables to p_n. This force the circuit to compute f_n as shown.

When we hardwire some of the inputs' variables, the Boolean circuit size can only diminish. By Definition 52 of the Boolean circuit interpreter, the description-length of a circuit is an increasing monotone function of its size. So, when we hardwire p_n in the input, the circuit can only diminish in description-length. This implies that $|f_n|_{\mathcal{C}} \leq \left|I|_{2n+1+|f_n|_{\mathcal{U}^c}}\right|_{\mathcal{C}} < |f_n|_{\mathcal{C}}$, this inequality is a contradiction. Consequently, we must have

$$\left|I|_{2n+1+|f_n|_{\mathcal{U}^c}}\right|_{\mathcal{C}} \geq |f_n|_{\mathcal{C}}\,. \tag{A.27}$$

With our lower-bounds on circuits' description-lengths of Equation A.25,

it gives, for all n sufficiently large,

$$\left| I|_{2n+1+|f_n|_{\mathcal{U}^c}} \right|_{\mathcal{C}} \geq bn^{1+\gamma/2} |f_n|_{\mathcal{U}^c} . \tag{A.28}$$

We cannot conclude the Theorem directly from this, the fact that $|f_n|_{\mathcal{U}^c}$ can vary with n complexifies the analysis. The rest of the proof address this issue by identifying an infinite subsequence of (f_n) for which the evolution of $|f_n|_{\mathcal{U}^c}$ has a tight characterization.

We distribute the sequence of learning problems' functions, f_n, in different sets. For some precision parameter $\nu > 0$, we define $i^{\max} = \lceil \frac{1}{\nu} \rceil d$, the sequence of indices $i = 1, \ldots, i^{\max}$, the following sets

$$S_i = \left\{ n \in N | \frac{i-1}{i^{\max}} n^{\frac{i-1}{i^{\max}} d} < |f_n|_{\mathcal{U}^c} \leq \frac{i}{i^{\max}} n^{\frac{i}{i^{\max}} d} \right\}, \tag{A.29}$$

and S_0 containing the unique possible description-length of 0 function.

We have the upper-bound on the description-length of Equation A.20, for all n, $|f_n|_{\mathcal{U}^c} \leq n^d$. Thus, the functions in the infinite sequence (f_n) are well partitioned in the defined sets. By the pigeon-hole principle there exists an index i^* such that S_{i^*} is of infinite size.

We now restrict all n to be in S_{i^*}. We prove that the description-length of the program I is superlinear in the input-size with the interpreter for circuits for this sub-sequence of indices.

We distingish three cases that cover all the possibilities for i^*:

1. Zero-length $i^* = 0$.

   By Definition 44 of Turing machines, a Turing machine has at least two states. By Definition 45 of the encoder for Turing machines, a Turing machine with at least two states has at least a description-length of two bits. From these two facts and by Definition 47 of $\mathcal{U}^c$, a zero-length description interpreted by $\mathcal{U}^c$ outputs $\perp$. Thus, for all n and any function $f \in H^n$, $|f|_{\mathcal{U}^c} > 0$. Consequently, the set S_0 is empty and this case is not possible.

2. Sub-linear $i^* \leq \lceil 1/\nu \rceil = i^{\max}/d \to \frac{i^*}{i^{\max}} d \leq 1$.

   We provide a lower-bound on the power linking the input-size to the circuits' description-lengths lower-bounds given in Equation A.25.

   We note that we have, $0 < |f_n|_{\mathcal{U}^c} \leq \alpha n$ for $\alpha = i^*/i^{\max}$ by Equation A.29.

Using properties of the logarithm, for all $\kappa_1, \kappa_2 > 0$, and all n sufficiently large,

$$\begin{aligned}\log_{2n+1+|f_n|_{\mathcal{U}^c}} bn^{1+\gamma/2}\,|f_n|_{\mathcal{U}^c} &\geq \log_{(2+\alpha)n+1} n^{1+\gamma/2} \\ &\quad + \log_{2n+1+|f_n|_{\mathcal{U}^c}} b + \log_{2n+1+|f_n|_{\mathcal{U}^c}} |f_n|_{\mathcal{U}^c} \\ &\geq \frac{\log_n n^{1+\gamma/2}}{\log_n((2+\alpha)n+1)} - \kappa_1 + 0 \\ &\geq \frac{1+\gamma/2}{1+\kappa_2} - \kappa_1.\end{aligned} \tag{A.30}$$

Since $\gamma > 0$, there exists κ_1, κ_2 small enough such that this lower bound is strictly greater than one.

3. Superlinear $i^* > \lceil 1/\nu \rceil \to i^* - 1 \geq \lceil 1/\nu \rceil = i^{\max}/d \to \frac{i^*-1}{i^{\max}} d \geq 1$.

   We pose $q = \frac{i^*}{i^{\max}} d$, $\nu' = \frac{1}{\lceil 1/\nu \rceil} \leq \nu$, and $\alpha_1 = \frac{i^*-1}{i^{\max}}, \alpha_2 = \frac{i^*}{i^{\max}}$.

   The following holds, by definition of S_{i^*} in Equation A.29, $\alpha_1 n^{q-\nu'} \leq |f_n|_{\mathcal{U}^c} \leq \alpha_2 n^q$. Also, we have $q \geq 1$ and $q - \nu' = \frac{i^*-1}{i^{\max}} d \geq 1$.

   In this case the power linking the input-size to the circuits' description-lengths lower-bounds is, for all $\kappa_1, \kappa_2, \kappa_3 > 0$ and all sufficiently large n,

$$\begin{aligned}\log_{2n+1+|f_n|_{\mathcal{U}^c}} bn^{1+\gamma/2}\,|f_n|_{\mathcal{U}^c} &\geq \log_{2n+1+|f_n|_{\mathcal{U}^c}} \alpha_1 bn^{1+\gamma/2+q-\nu'} \\ &\geq \log_{(2+\alpha_2)n^q+1} n^{1+\gamma/2+q-\nu'} \\ &\quad + \log_{2n+1+|f_n|_{\mathcal{U}^c}} \alpha_1 b \\ &\geq \frac{\log_{n^q} n^{1+\gamma/2+q-\nu'}}{\log_{n^q}((2+\alpha_2)n^q+1)} - \kappa_1 \\ &\geq \frac{1+\frac{1+\gamma/2-\nu'}{q}}{1+\log_{n^q}(2+\alpha_2)+\kappa_2} - \kappa_1 \\ &\geq \frac{1+\frac{1+\gamma/2}{d}}{1+\kappa_2+\kappa_3} - \frac{\nu}{1+\kappa_2+\kappa_3} - \kappa_1.\end{aligned} \tag{A.31}$$

Our reasoning can be taken with arbitrarily small ν and $\kappa_1, \kappa_2, \kappa_3$, such that the lower bound can be made strictly greater than one.

All the possible cases have been treated.

The proved bound on the Boolean circuits' description-length extends to their sizes. More precisely, any superlinear lower-bound of the type $n^{1+\iota}$, for some $\iota > 0$, on the description-length of the Boolean circuits with interpreter $\mathcal{C}$ implies, for all n sufficiently large, a similar lower-bound, $n^{1+\tau}$, for some $0 < \tau < \iota$, for the Boolean circuits' sizes by Definition 52.

This finishes the proof. □

Theorem 16. *If for all $\epsilon \in (0, 1/2)$, $\delta \in (0,1)$, $c, d \in \mathbb{N}^+$ and all $\gamma > 0$*

$$G^d_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) \in O(n^{1+\gamma}) \tag{2.14}$$

then $\boldsymbol{P} \neq \boldsymbol{NP}$.

Proposition 34. *Pavan et al. [2006]. For any $k_1, k_2 \in \mathbb{N}^+$, there exists a language $L \in \boldsymbol{P}^{\Sigma_2^p}$ such that for every circuit sequence $(C_1, \ldots, C_n, \ldots)$ whose circuits' sizes are at most n^{k_1}, the following holds*

$$\Pr_{x \in U(\mathcal{B}^n)} [L(x) = C_n(x)] \leq 1/2 + 1/n^{k_2}, \tag{A.32}$$

where $U(\mathcal{B}^n)$ denotes the uniform distribution on $\mathcal{B}^n$.

Proof of Theorem 16. By contraposition, we suppose $\mathbf{P} = \mathbf{NP}$, then the polynomial hierarchy collapses, $\mathbf{P} = \mathbf{PH}$, and thus in particular $\mathbf{P} = \mathbf{P}^{\Sigma_2^p}$.

Implying with Proposition 34 that for all k_1 and k_2 there exists a language in $\mathbf{P}$ such that for all n there does not exist a Boolean circuit of size smaller than n^{k_1} which approximate the language with accuracy at least $\frac{1}{2} + \frac{1}{n^{k_2}}$ under the uniform distribution.

Fix $\epsilon = 1/4$, any $\delta \in (0,1)$, any $d \in \mathbb{N}^+$, and $\gamma = 1$.

Take $k_1 = 2$ and $k_2 = 1$. For any $n \geq 4$, to select a function of error rate at most $1/4$ all circuits of size lower than n^2 must be eliminated. By Definition 52 of the Boolean circuits' interpreter and Proposition 68, the link between circuits' size and description-length is an increasing monotonic function, and thus, at least n^2/b samples will be necessary to eliminate all these circuits for the learning algorithm $MDL^{\mathcal{C}}$, for some fixed constant $b > 0$.

Moreover, by Definition 47 of $\mathcal{U}^c$, since the language is in $\mathbf{P}$ there exists some $c \in \mathbb{N}^+$ and some $s \in \mathcal{B}^*$ such that the language is computed by $\mathcal{U}^c(s, .)$. For any $d \in \mathbb{N}^+$ and for all sufficiently large n, we have $|s| \leq n^d$.

Thus we have some combination of parameters for which, for some constant $|s|$,

$$G^d_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) \geq \frac{n^2/b}{\frac{a_1}{\epsilon}(\log \frac{1}{\delta} + |s| + a_2)} \in \Omega(n^2). \tag{A.33}$$

□

Theorem 17. *If there exists $f \in \boldsymbol{E}$ and $\iota > 0$ such that the Boolean circuits computing f_n are at least of size $2^{\iota n}$; then for any $\gamma > 0$ there exists $\epsilon \in (0, 1/2)$, $\delta \in (0, 1)$, $c, d \in \mathbb{N}^+$ such that we have*

$$G^d_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) \in \Omega(n^{1+\gamma}). \tag{2.15}$$

Proposition 35. *Arora and Barak [2009]. Let $S : \mathbb{N} \to \mathbb{N}$ and $f \in \boldsymbol{E}$ such that Boolean circuits that decide $f|_n$ are at least of sizes $S(n)$ for every n. Then there exists a function $g \in \boldsymbol{E}$ and a constant $b > 0$ such that approximating $g|_n$ under the uniform distribution with accuracy at least 0.99 requires Boolean circuits of sizes at least $S(n/b)/n^b$ for every sufficiently large n.*

Proof of Theorem 17. By Definition 9, the sample efficiency gain is

$$G^d_{\mathcal{U}^c \to \mathcal{C}}(\epsilon, \delta, n) = \sup_{f \in H^n, \mathcal{P} \in \Delta(\mathcal{B}^n)} \frac{m^{\epsilon,\delta}_{\mathcal{C}}(f, \mathcal{P})}{m^{\epsilon,\delta}_{\mathcal{U}^c}(f, \mathcal{P})} \quad \text{subject to} \quad |f|_{\mathcal{U}^c} \leq n^d. \tag{A.34}$$

Let's define a sequence of learning problems entailing our theorem.

By Proposition 35 and the assumption there exists a language $g \in \mathbf{E}$ such that $g|_n$ can only be approximated with accuracy 0.99 under the uniform distribution by Boolean circuits of size at least in $\Omega(2^{\Omega(n)}/n^b)$ for some constant $b > 0$.

We define g' to be the application of g on the $a \log n$ first variables of the input, for some constant $a > 0$. For every n, we define D_n a probability distribution on $\mathcal{B}^n$, where x_1 denotes the first $a \log n$ variables of the input and x_2 the others, $U(\mathcal{B}^N)$ is the uniform distribution on $\mathcal{B}^N$,

$$D_n[x_1, x_2] = \begin{cases} U(\mathcal{B}^{a \log n})[x_1] & \text{if } x_2 = \mathbf{0}^{n - a \log n} \\ 0 & \text{else.} \end{cases} \tag{A.35}$$

Finally, for every n, we define the learning problem $(g'|_n, D_n)$.

There exists a polynomial-time Turing machine that decides g'. Also the condition $|g'|_n|_{\mathcal{U}^c} \leq n^d$ will always be satisfied for all n large enough since the learning problem's function corresponds to a program/Turing-machine of fixed description-length.

Using the guarantee of Proposition 7, $m^{\epsilon,\delta}_{\mathcal{U}^c}(g'|_n, \mathcal{P}) \leq \frac{a_1}{\epsilon}(\log \frac{1}{\delta} + |g'|_n|_{\mathcal{U}^c} + a_2)$, we deduce that $m^{\epsilon,\delta}_{\mathcal{U}^c}(f, \mathcal{P})$ is upper-bounded by a constant independent of n.

For all n sufficiently large, all the Boolean circuits that approximates $g'|_n$ with accuracy at least 0.99 under D_n have sizes at least in $\tilde{\Omega}(n^{a\Omega(1)})$.

Thus, by Proposition 68, at least $\Omega(n^{a\Omega(1)})$ samples are necessary to solve the learning problem $(g'|_n, D_n)$ by $MDL^{\mathcal{C}}$ for all n sufficiently large.

Consequently, for any $\gamma > 0$ there exists a fixed and large enough, such that there exists $\epsilon \in (0, 1/2)$, $\delta \in (0, 1)$ and $c > 0$ satisfying $G^d_{\mathcal{U}^c}(\epsilon, \delta, n) \in \Omega(n^{1+\gamma})$. □

1.2 Description-length gains of Turing machines over circuits and neural networks

This section highlights the results focusing on description-length instead of PAC-learning gains. The proofs are similar to the proofs of the last section, and sometimes a reference to the proofs of the last section will be made.

The structure of the results' presentation is the same as for the study of PAC-learning gains.

Theorem 36. *There exists a constant $q \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}^+$,*

$$\sup_{f \in H^n} \frac{|f|_{\mathcal{U}}}{|f|_{\mathcal{C}}} \leq q. \tag{A.36}$$

Proof. A technical detail is that, by Definition 52, for any n and function $f \in H^n$, $|f|_{\mathcal{C}} > 0$. So, the denominator is always non-zero.

The Proposition 64, affirms that there exists a constant K such that for any n and any function $f \in H^n$, the following holds $|f|_{\mathcal{U}} \leq |f|_{\mathcal{C}} + K$.

This fact, with the fact that the denominator is never null, implies

$$\sup_{f \in H^n} \frac{|f|_{\mathcal{U}}}{|f|_{\mathcal{C}}} \leq \frac{|f|_{\mathcal{C}} + K}{|f|_{\mathcal{C}}} \leq K + 1. \tag{A.37}$$

□

Theorem 37. *We have*

$$\sup_{f \in H^n} \frac{|f|_{\mathcal{C}}}{|f|_{\mathcal{U}}} \in \Omega(2^n). \tag{A.38}$$

Proof. Let be the Turing machine p^T, computing function p, and let $p|_n$ denote function p restricted to inputs of size n.

For input x of size n:

1. *Compute the input size n.*
2. *Enumerate all binary function in H^n in some pre-defined fixed lexicographic order. For each function, f:*
   (a) *Compute the minimal description-length necessary to compute the function f for a circuit according to interpreter $\mathcal{C}$:*

$$\begin{aligned} &\min_{h \in \mathcal{B}^*} && |h| \\ &\textit{subject to} && \mathcal{C}(h, y) = f(y) \quad \forall y \in \mathcal{B}^n. \end{aligned} \tag{A.39}$$

   (b) *If $|h| \geq 2^n$ then return $f(x)$.*

For any input-size n, there always exists a function that will satisfy the description-length condition that appears in step *(b)*. We prove it by a counting argument, there are $2^{(2^n)}$ functions in H^n and at most $2^{(2^n-1)}$ functions can be represented by a binary representation of length at most $2^n - 1$.

By construction and Definition 46 of the universal Turing machine, the computed function is computable by the Turing machine p^T and, thus, for any n, $|p|_n|_{\mathcal{U}} \leq \alpha$ for some constant α.

Also by construction, for any n, the computed function is only computed by circuits of description-length at least 2^n, $|p|_n|_{\mathcal{C}} \geq 2^n$. □

Theorem 38. *For all $c \in \mathbb{N}^+$, we have*

$$\sup_{f \in H^n} \frac{|f|_{\mathcal{C}}}{|f|_{\mathcal{U}^c}} \in O(n^c \log^2 n). \tag{A.40}$$

Proof. The proof is the same as the proof of Theorem 13 pruned of the PAC-learning related terms. □

Theorem 39. *For all* $1 < c \in \mathbb{N}^+$*, we have*

$$\sup_{f \in H^n} \frac{|f|_{\mathcal{C}}}{|f|_{\mathcal{U}^c}} \in \Omega(n \log n). \tag{A.41}$$

Proof. Consider the parity function $\oplus(x) = \sum_i x_i \mod 2$. For any n, let $\oplus|_n$ denote the function $\oplus$ restricted to inputs of size n.

The parity function can be computed by a Turing machine in linear time, and thus, for some constant α and all n sufficiently large, $|\oplus|_n|_{\mathcal{U}^c} \leq \alpha$ according to Definition 47.

Moreover, for any n, if a circuit compute $\oplus|_n$ then it must have at least n nodes to depend on all the input's variables. Then, by Definition 52 linking the circuit's size to its description-length, $|\oplus|_n|_{\mathcal{C}} \in \Omega(n \log n)$. □

Theorem 40. *If there exists* $c, d \in \mathbb{N}^+$*, and* $\gamma > 0$ *such that*

$$\left[\sup_{f \in H^n} \frac{|f|_{\mathcal{C}}}{|f|_{\mathcal{U}^c}} \quad \textit{such that} \quad |f|_{\mathcal{U}^c} \leq n^d\right] \notin O(n^{1+\gamma}) \tag{A.42}$$

then there exists a language in $\boldsymbol{P}$ *not computable by a sequence of Boolean circuits whose sizes are in* $O(n^{1+\tau})$ *for some* $\tau > 0$*.*

Proof. Take the proof of Theorem 15 beginning in Equation A.23. □

We note that the upper-bound n^d on the description-length $|f|_{\mathcal{U}^c}$ is not necessary for Theorem 40 to hold. Using Proposition 50, for any function relevant in the proof of the theorem, an upper-bound on the function description-length with $\mathcal{U}^c$ polynomial in the input-size holds. This fact can replace the bound in n^d in the proof of Theorem 15.

Theorem 41. *If for all* $c \in \mathbb{N}^+$*, and all* $\gamma > 0$*,*

$$\sup_{f \in H^n} \frac{|f|_{\mathcal{C}}}{|f|_{\mathcal{U}^c}} \in O(n^{1+\gamma}) \tag{A.43}$$

then $\boldsymbol{P} \neq \boldsymbol{NP}$*.*

Proposition 42. *Kannan [1982]. For any nonnegative integer k, there exists a language* $L \in \sum_2^p$ *such that L is not computable by a sequence of circuits whose sizes are in* $O(n^k)$*, where n is the input-size.*

Proof of Theorem 41. By contraposition, suppose $\mathbf{P} = \mathbf{NP}$; then the polynomial hierarchy collapses and $\mathbf{P} = \mathbf{PH} = \sum_2^p$.

By Proposition 42, with $k = 2$, $\mathbf{P}$ has a language not computable by any circuit sequence whose sizes are in $O(n^2)$. Denote the function representing this language by f, and $f|_n$ its restriction to size n inputs.

Then, by Definition 52 of $\mathcal{C}$, a sequence of circuits that computes the function does not have their description-length in $O(n^2)$.

Moreover, by Definition 47 of $\mathcal{U}^c$, there exists constants c and α such that, for all n sufficiently large, $|f|_n|_{\mathcal{U}^c} \leq \alpha$ since there is a fixed Turing machine able to compute the function for all n. □

Theorem 43. *If there exist a language $g \in \boldsymbol{E}$ such that $g|_n$ can only be computed by circuits of sizes at least $2^{\epsilon n}$ for some $\epsilon > 0$; then for all $\gamma > 0$ there exists $c \in \mathbb{N}^+$ such that*

$$\sup_{f \in H^n} \frac{|f|_{\mathcal{C}}}{|f|_{\mathcal{U}^c}} \in \Omega(n^{1+\gamma}). \tag{A.44}$$

Proof. For any n pose f_n to be g applied to the first $a \log n$ variables of the input, for some $a > 0$. By construction $|f_n|_{\mathcal{C}} \in \Omega(n^{a\Omega(1)})$, and since there exists a polynomial-time Turing machine deciding (f_n), $|f_n|_{\mathcal{U}^c} \in O(1)$, for c large enough.

Fix a large enough to complete the proof. □

1.3 Interpreters

1.3.1 Universal Turing Machine

We restrict our Turing machines to binary-valued outputs in the whole work.

The definitions of this sub-section are given with the number of working-tape let as a variable in some cases. The results of this research are correct for any value of this parameter.

Definition 44. Turing machines, *Li et al. [2019]. We first define one-tape Turing machines before generalizing the definition to multi-tape Turing machines.*

A binary-valued one-tape Turing machine is defined by

- *Q a finite set of states;*
- *$A = \{0, 1, b\}$ the Turing machine's alphabet;*
- *$q_0 \in Q$ the initial state in which the Turing machine begins;*

- $\{accept, reject\} = F \subset Q$ *the set of the two final states which determine the output of the Turing machine: if the Turing machine stops in the accept state then the output is* 1*, else if it stops in the final state reject then the output is* 0*, else it is* $\perp$*;*

- *a mapping from* $((Q \backslash F) \times A)$ *to* $(Q \times S)$ *defining the transition of the Turing machine, where* $S = A \cup \{L, R\}$ *which correspond to either writing the element of* A *to the current head place or move the head to the L:left or R:right.*

The partial computable function implemented by the Turing machine is defined by setting the binary input on the unique tape in contiguous cells (the b symbol being assigned to the other cells); positioning the head on the first cell; set the Turing machine in the q_0 *state; then to apply recursively the mapping of the Turing that determines the change of states, writing on the tape, and head movements; finally the output is obtained either when a final state in* F *is obtained, or else by* $\perp$*.*

For $k_1, k_2 \in \mathbb{N}^+$*, we define* k_1*-input-tape,* k_2*-working-tape Turing machines. These machines have* k_1 *binary-inputs that are placed on the* k_1 *inputs' tapes. There is one head by tape and these tapes are read-only. There are also* k_2*-working-tapes with one head by tape, they are blank at the start, and are read and write.*

These machines are determined in a similar way to one-tape Turing machine: the elements Q*,* A*,* q_0*,* F*, and* S *are defined in the same way. The mapping is adapted, the mapping goes from* $((Q \backslash F) \times A^{k_1+k_2})$ *to* $(Q \times \{L, R\}^{k_1} \times S^{k_2})$*, with the natural interpretation.*

The partial computable function implemented by such Turing machines follows from a natural generalization of one-tape Turing machines.

Definition 45. Turing machines encoding $E(.)$*, Li et al. [2019]. Following Definition 44, any Turing machine,* T*, can be fully described by a set of states* Q*, the initial state* q_0*, and a mapping from* $((Q \backslash F) \times A)$ *to* $(Q \times S)$*.*

The mapping and T *can be described by a list of quadruples* $[(p_i, t_i, q_i, s_i)]_{i=1}^{r}$ *where* r *is the number of rules and for all* i*,* $p_i, q_i \in Q$*,* $t_i \in A$*,* $s_i \in S$*. Each element can be identified with* $s = \lceil \log(|Q| + 5) \rceil$ *bits. Be* $e(.) : Q \cup S \to \mathcal{B}^s$ *this encoding. By convention this encoding will satisfy the following constraint, the states* q_0*, accept, and reject will be encoded to predefined arbitrary values (the three first elements in the Boolean lexicographic order of the output for example).*

We define the encoding of T *to be*

$$E(T) = \underbrace{0 \dots 0}_{s} 1 \underbrace{0 \dots 0}_{r} 1 [e(p_i)e(t_i)e(s_i)e(q_i)]_{i=1}^{r}. \tag{A.45}$$

This encoding completely defines the Turing machine T.

The encoding is prefix-free: no encoding is the prefix of another.

To define an encoding for multiple-tape Turing machines, generalize the encoding to their mappings defined in Definition 44.

Definition 46. Universal Turing machine $\mathcal{U}$. *For any $k \in \mathbb{N}^+$, the interpreter $\mathcal{U} : \mathcal{B}^* \times \mathcal{B}^* \to \mathcal{B} \cup \{\perp\}$ computes $T(u, x)$ on input $([E(T), u], x)$, where E follows Definition 45 for two-input-tape and $k-$working-tape Turing machines. If the input has not a form that encodes a Turing machine then $\perp$ is the output.*

Note that the decomposition of the first argument in $E(T)$ and u is well defined since the encoding E is prefix-free.

Definition 47. Polynomial-time universal Turing machines $\mathcal{U}^c$.

For any $k, c \in \mathbb{N}^+$, we define an interpreter $\mathcal{U}^c : \mathcal{B}^* \times \mathcal{B}^* \to \mathcal{B} \cup \{\perp\}$. It is a 2-input-tape, 3-working-tape Turing machine.

On input $([E(T), u], x \in \mathcal{B}^*)$, for T a 2-input-tape k-working tape Turing machine, E the encoding in Definition 45, and $u \in \mathcal{B}^*$; the following operations are performed:

1. *On the third working tape, the interpreter computes the input-size, n, of the second input, x.*
2. *Still on the third work-tape, it computes n^c.*
3. *The interpreter computes in at most n^c steps that the form of the first input corresponds to the encoding of a Turing machine. If it does not correspond to a Turing machine or if the number of steps limit is reached, it outputs $\perp$.*
4. *Then, it computes a simulation of the behavior of the Turing machine T on input (u, x) with the two first work-tapes using the construction of Hennie and Stearns [1966], whose result is given in Proposition 48. Simultaneously, the interpreter computes the number of steps dedicated to the simulation on the third work-tape. (Note that it is well the number of steps dedicated to the simulation and not the number of simulated steps that are counted.)*
5. *In the computed simulation if a final state in F is reached then enters this state for the universal Turing machine. If the limit of computation for the simulation, n^c, is attained without entering a final state of F in the computed simulation then enter the state reject.*

For all these operations, the total number of steps for the first input, $[E(T), u]$, fixed can be made in βn^c, for some fixed $\beta > 0$.

Using Proposition 48, for any $\delta > 0$, any Turing machine with computational complexity in $O(n^{c-\gamma})$ can be simulated, for some $h \in \mathcal{B}^*$ and all n sufficiently large, by $\mathcal{U}^c(h,.)$.

Proposition 48. *Efficient universal Turing machines, Hennie and Stearns [1966], Arora and Barak [2009]. There exists a universal Turing machine which, for any Turing machine T, on inputs $E(T)$ and x computes $T(x)$.*

Furthermore, for some $\alpha > 0$, if the Turing machine T on input x stops in t steps then the universal Turing machine stops in $\alpha t \log t$.

Proposition 49. Hardwiring. *For any 2-input-tape Turing machine, T, and input $h \in \mathcal{B}^*$ there exists a Turing machine T^h such that T^h compute the function $T(h,.)$.*

Moreover, T^h has $\rho\,|h|$ rules, for some ρ independent of h; and T^h computes the function $T(h,.)$ in the same number of steps.

Proof. Let Q be the states and R be the set of rules of T that defines its mapping, construct the new states $Q \times \{1,\ldots,|h|\}$ and the new rules $R \times \{1,\ldots,|h|\}$. The new rules are made such that any operation of the Turing machine T on the first input-tape is translated into an equivalent change in the state of the Turing machine T^h. Allowing the simulation of the tape's head corresponding to input h in the states of T^h.

In our construction there are thus $R \cdot |h|$ rules in T^h, fix $\rho = R$ in the theorem statement. □

Proposition 50. *There exists a constant $\beta > 0$ such that for any function f in H_n, if $|f|_{\mathcal{U}^c} < +\infty$ then $|f|_{\mathcal{U}^c} \leq \beta n^c$.*

Proof. Any solution h of length larger than βn^c can be made smaller by cropping all binary symbols after index βn^c on the tape, since, by Definition 47, $\mathcal{U}^c$ cannot read them in βn^c steps. □

1.3.2 Boolean circuit

Definition 51. Boolean circuit, *Arora and Barak [2009]. A Boolean circuit C is a directed acyclic graph with $n \in \mathbb{N}$ potential sources and one sink. The source vertices have an associated input variable whose index is between 1 and n. The non-source vertices are called gates and have an associated logical operation OR, AND, or NOT ($\wedge$, $\vee$ or $\neg$) called label.*

The OR and AND vertices have two input edges, the NOT vertices have one input edge.

The number of vertices will be denoted $|C|$ and called the size of the circuit.

The output of the circuit on an input $x \in \mathcal{B}^n$ is the value associated to the sink vertice applying recursively the following assignment for each vertex v: if v is a source corresponding to input variable i then its value is x_i; else v is a gate, apply the logical operator corresponding to its label on the input values (values from its parent vertices).

Definition 52. Boolean circuit interpreter, $\mathcal{C}$. *We define $\mathcal{C} : \mathcal{B}^* \times \mathcal{B}^* \to \mathcal{B} \cup \{\perp\}$ to compute circuit $C(x)$ on input (h, x) with h of the form*

$$\underbrace{0\ldots0}_{\lceil \log_2 n \rceil} 1[\textit{binary description of } n] \underbrace{0\ldots0}_{|C|} 1[\textit{label and inputs' vertices of vertice } i]_{i=1}^{|C|}, \tag{A.46}$$

where label and inputs' vertices of any vertice correspond to

$$2 + \max\{2\lceil \log_2 |C| \rceil, \lceil \log_2 n \rceil\}$$

bits; 2 bits to denote its logical label, and $2\lceil \log_2 |C| \rceil$ bits for the input vertices.

It outputs $\perp$ if h has not an acceptable form.

The description-length of h is thus of

$$2\lceil \log_2(n) \rceil + 2 + |C|\,(3 + \max\{2\lceil \log_2 |C| \rceil), \lceil \log_2 n \rceil\}$$

bits.

Which can be bounded by $9\,|C| \log |C|$ bits for circuits of sizes $|C| \geq n \geq 3$.

Proposition 53. *Frandsen and Miltersen [2005].*

Boolean circuits of size at most

$$\frac{2^n}{n}(1 + 3\frac{\log(n)}{n} + O(\frac{1}{n})) \tag{A.47}$$

compute all functions in H^n.

1.3.3 Artificial Neural Network

Similarly to Boolean circuits, we define an interpreter for Artificial Neural Networks (ANNs) and provide propositions linking the two definitions in terms of description-length.

Definition 54. Floating-point operators. *For any $d \in \mathbb{N}^+$, a floating-point operator is:*

- *a 0-ary/constant operator, which is a float-number —an element in $\mathcal{B}^d$;*

- *an unary operator from $\mathcal{B}^d$ to $\mathcal{B}^d$, such as negation $-(.)$, inverse $1/(.)$, or the exponential $\exp(.)$;*
- *a binary operator from $\mathcal{B}^d \times \mathcal{B}^d$ to $\mathcal{B}^d$, such as addition $(. + .)$, or product $(. \cdot .)$.*

Definition 55. Artificial Neural Network. *Given $d \in \mathbb{N}^+$ and a fixed predefined finite set of floating-point operators $\mathcal{O}$ on $\mathcal{B}^d$ with at least two 0-ary operators identified as 0^F and 1^F (thus both in $\mathcal{B}^d$).*

For any $n \in \mathbb{N}^+$, an ANN is a directed acyclic graph with one sink and at most n input-variable sources. The sources vertices have each an associated input variable whose index is between 1 and n. The non-source vertices have an associated operator taken from the set $\mathcal{O}$. If the operator is 0-ary then they have no parent vertice, if it is unary then they have one parent vertice, else if the operator is binary then they have two parent vertices.

We denote $|A|$ the number of vertices of ANN A.

To compute the output of the ANN on input $x \in \mathcal{B}^n$, the following operations are performed:

1. *for all the input-variable sources: the floating-point operator $0^F \in \mathcal{B}^d$ or $1^F \in \mathcal{B}^d$ is assigned accordingly to the associated input variable value in $\{0, 1\} = \mathcal{B}$.*
2. *for all the other vertices assign the value in $\mathcal{B}^d$ corresponding to the associated floating-point operator in $\mathcal{O}$ and the values assigned to the potential parents.*
3. *when the output sink vertex has been assigned a value: if it is 0^F then output $0 \in \mathcal{B}$, if it is 1^F then output $1 \in \mathcal{B}$, else output $\perp$.*

Definition 56. Artificial Neural Network interpreter, $\mathcal{ANN}^{\mathcal{O}}$. *We define $\mathcal{ANN}^{\mathcal{O}} : \mathcal{B}^* \times \mathcal{B}^* \to \mathcal{B} \cup \{\perp\}$ the interpreter for ANN. On input (g, x) it computes the result of applying ANN A on x, $A(x)$, where g is of the form*

$$\underbrace{0\ldots0}_{\lceil \log_2 n \rceil} 1 [\textit{binary description of } n] \underbrace{0\ldots0}_{|A|} 1 \\ [\textit{input variable/operator and inputs' vertices of vertice } i]_{i=1}^{|A|}, \tag{A.48}$$

the operator in $\mathcal{O}$ and the parent(s) or the input variable will be described in $\lceil \log_2(|\mathcal{O}| + 1) \rceil + \max\{2\lceil \log_2 |\mathcal{O}| \rceil, \lceil \log_2 n \rceil\}$ bits for each vertex.

If g does not have a correct form then $\perp$ is returned.

The length of g encoding an ANN A is $2\lceil \log_2 n \rceil + 2 + |A| \, (1 + \lceil \log_2(|\mathcal{O}| + 1) \rceil + \max\{2\lceil \log_2 |A| \rceil, \lceil \log_2 n \rceil\})$

Proposition 57. *For any fixed set of floats' operators $\mathcal{O}$ for which the AND, OR, and NOT Boolean functions can each be computed by an ANN using the operators in $\mathcal{O}$, there exists constant $\alpha, \beta > 0$ such that the following holds.*

For any $n \in \mathbb{N}^+$ and any $h \in \mathcal{B}^*$ there exists $g \in \mathcal{B}^*$ such that for all $x \in \mathcal{B}^n$

$$\mathcal{C}(h, x) = \mathcal{ANN}^{\mathcal{O}}(g, x) \tag{A.49}$$

and

$$|g| \leq \alpha\,|h|\,. \tag{A.50}$$

Conversely, for any $n \in \mathbb{N}^+$ and any $g \in \mathcal{B}^*$ there exists $h \in \mathcal{B}^*$ such that for all $x \in \mathcal{B}^n$

$$\mathcal{ANN}^{\mathcal{O}}(g, x) = \mathcal{C}(h, x) \tag{A.51}$$

and

$$|h| \leq \beta\,|g|\,. \tag{A.52}$$

Proof. For the first part, if h has not a correct form to represent a circuit then $\mathcal{C}(h, .)$ always output $\perp$ which can also be done by a g that do not represent an ANN.

Otherwise, there is a circuit, C, that corresponds to $\mathcal{C}(h, .)$, the logical operation of each node of this circuit can be simulated by a part of an ANN of fixed maximal size. A combination of these parts gives an ANN, A, which computes the same function as the Boolean circuit and whose size is at most a fixed multiple of the size of the circuit.

Let's pose $\gamma > 0$ the factor of the sizes, $|A| \leq \gamma\,|C|$. The description-lengths have the following relationship

$$\begin{aligned}
&2\lceil \log_2 n \rceil + 2 + |A|\,(1 + \lceil \log_2(|\mathcal{O}| + 1) \rceil + \max\{2\lceil \log_2 |A| \rceil, \lceil \log_2 n \rceil\}) \\
&\qquad \leq 2\lceil \log_2 n \rceil + 2 + \gamma\,|C|\,(1 + \lceil \log_2(|\mathcal{O}| + 1) \rceil \\
&\qquad\quad + \max\{2\lceil \log_2 |C| \rceil + 2\lceil \log_2 \gamma \rceil, \lceil \log_2 n \rceil\}) \\
&\qquad \leq \alpha\,(2\lceil \log_2(n) \rceil + 2 + |C|\,(3 + \max\{2\lceil \log_2 |C| \rceil), \lceil \log_2 n \rceil\})\,,
\end{aligned} \tag{A.53}$$

with $\alpha = \max\{1, \gamma, (1 + \lceil \log_2(|\mathcal{O}| + 1) \rceil + 2\lceil \log_2 \gamma \rceil)/3\}$.

A similar argument holds for the second part by noting that any floating-point operator can be computed by a finite size Boolean circuit by Proposition 53. □

1.4 VC-dimension analysis and tightness of Proposition 7

For an introduction to the *shatter* concept and the *VC-dimension*, we refer to [Shalev-Shwartz and Ben-David, 2014].

Definition 58. Shatter. *Let be a set C.*
A set $A \subset 2^C$ shatters a set $B \subset C$ iff

$$\{a \cap B | \, a \in A\} = 2^{|B|}. \tag{A.54}$$

Definition 59. VC-Dimension, *Vapnik and Chervonenkis [2015]. The VC-dimension of a set $F \subset H^n$, $VC(F)$, is the size of the largest set $B \in \mathcal{B}^n$ such that F shatters B, where each binary-valued function in F is translated as a set in $\mathcal{B}^n$.*

Proposition 60. *VC-Dimension of Turing machines. There exists a constant $a > 0$ such that the following holds.*
For any n and D in $\mathbb{N}^+$, with D larger than some constant, we define the sets of functions $F^D = \{\mathcal{U}(h, .) \in H^n | \, h \in \mathcal{B}^* \ |h| \leq D\}$.
These sets satisfy $VC(F^D) \geq aD$.

Proof. Consider the two inputs Turing machine, $T(u, x)$, that output u_x if $x \leq |u|$ and 0 else, where the binary input string x is understood as the binary representation of a number.

From this Turing machine create for each D the set of functions

$$\{\mathcal{U}([E(T), u], .) | \, u \in \mathcal{B}^{\lfloor D/2 \rfloor}\},$$

this set shatters the set of the aD strings corresponding to the aD first natural numbers in binary representation, for some $a > 0$. Moreover, if D is bigger than $2 \cdot |E(T)|$, the construction satisfies the upper-bound of D on the description-lengths. □

Proposition 61. *VC-Dimension of Boolean circuits. There exists constants a, b, N such that the following holds for all $n \geq N$.*
Be the sets of functions $F^D = \{\mathcal{C}(h, .) \in H^n | \, h \in \mathcal{B}^* \ |h| \leq D\}$ for some constant $D \in \mathbb{N}$, with $D \geq bn^{1.01}$.
Then $VC(F^D) \geq aD$.

Proof. We pose $b = b_1 b_2$, for two positive constants b_1 and b_2.

We have $1.01 \log n \leq \log D - \log b$. Select $q = \lfloor \log D - \log b_1 \rfloor$ an integer and $b_2 \geq e$ for $1.01 \log n \leq q$ to hold.

Consider the inputs $\{[z\underbrace{0\dots0}_{n-q}] \in \mathcal{B}^n | z \in \mathcal{B}^q\}$. This set is shattered by a set of circuits of size lower than $2\frac{2^q}{q}$ for n (and thus q) sufficiently large by Proposition 53.

For all n sufficiently large, we have $2\frac{2^q}{q} \geq 2\frac{n^{1.01}}{1.01 \log n} \geq n$.

Thus, by Definition 52 these circuits can be expressed with

$$18\frac{2^q}{q}\log\frac{2^q}{q} = \frac{18}{\log_2 e}2^q(1-\log_2^{-1}(e)\frac{\log q}{q}+\frac{\log 2}{\log_2(e)}\frac{1}{q}) \tag{A.55}$$

bits.

There exists a constant b_1 sufficiently large such that, for all n sufficiently large (forcing q to be sufficiently large), $q \leq \log D - \log b_1 \equiv 2^{\log b_1}2^q \leq D$ implies that the description-length of the circuits, as bounded by Equation A.55, is smaller than D.

Finally, take $a = 2^{-\log b_1 - 1}$, we have $VC(F^D) \geq 2^q \geq aD$ since $q \geq \log D - \log b_1 - 1$. □

Proposition 62. *PAC-learning lower-bound, Shalev-Shwartz and Ben-David [2014]. There exists a constant α such that the following holds for any $\epsilon \in (0, 1/2), \delta \in (0,1)$, and any $n \in \mathbb{N}^+$.*

Consider the set of learning problems $(f, \mathcal{P})$ where $f \in F \subset H^n$ and $\mathcal{P} \in \Delta(\mathcal{B}^n)$ a probability measure.

For any learning algorithm, there exists a learning problem in the set such that a m-sample dataset of the considered learning problem with

$$m \geq \frac{\alpha}{\epsilon}\left[VC(F) + \log(\frac{1}{\delta})\right] \tag{A.56}$$

is necessary to get an (ϵ, δ)-PAC-learning performance.

Proposition 63. *Tightness of Proposition 7. There exists a constant α such that for any $\epsilon \in (0, 1/2), \delta \in (0,1), n \in \mathbb{N}^+$ and any interpreter φ, associated learning algorithm MDL$^\varphi$, and bound $D \in \mathbb{N}^+$ on the description-length of an underlying function to learn; there exists a learning problem such that a m-sample learning dataset with*

$$m \geq \frac{\alpha}{\epsilon}\left[VC(\{f \in H^n | \ |f|_\varphi \leq D\}) + \log(\frac{1}{\delta})\right] \tag{A.57}$$

is necessary for MDL$^{\varphi}$ to have an (ϵ, δ)-PAC-learning performance on the learning problem.

Proof. The statement is a direct consequence of Proposition 62. □

1.5 Kolmogorov complexity

The following proposition comes from Kolmogorov complexity theory, it is adapted to the context and notation of this paper. See Li et al. [2019] for a reference on the subject. The origins of the theorem can be found in Solomonoff [1960, 1962, 1964a,b] and Kolmogorov [1965].

Proposition 64. *Invariance Theorem. For all interpreters φ there exists a constant K such that for all $n \in \mathbb{N}^+$ and all functions $f \in H^n$, the following holds $|f|_{\mathcal{U}} \leq |f|_{\varphi} + K$.*

Proof. Be f^{φ} the string of length $|f|_{\varphi}$ such that $\varphi(f^{\varphi}, .) = f(.)$.

Take $E(\varphi^T)$ the encoding of the Turing machine corresponding to the interpreter φ, the encoding follows Definition 45.

The function $\mathcal{U}([E(\varphi^T), f^{\varphi}], .)$ is equal to f by Definition 46, and the string $[E(\varphi^T), f^{\varphi}]$ has length $|f|_{\varphi} + K$ where $K = \left|E(\varphi^T)\right|$ is independent of f. □

1.6 Technical propositions

1.6.1 Number of necessary samples for Boolean Circuits

Definition 65. *For any n, a* binary decision tree *is determined by*

- *a tree where all non-leaf nodes have exactly 3 neighbors: a first child, a second child, and one parent with the exception of one node which has no parent and is called the root;*
- *to each non-leaf node is associated an input-Boolean-variable;*
- *to each leaf is associated a Boolean value.*

The binary-valued function computed by the binary decision tree on an input $x \in \mathcal{B}^n$ is the result of the following computation:

1. *The current node is set to be the root.*

2. *If the current node is not a leaf, then if the associated input-Boolean-variable in x is* 1 *then set the current node as the first child, else set the second child as the current node. Else continue to the next step.*
3. *The current node is thus a leaf, return as output the Boolean value associated to the current node.*

Proposition 66. *Given m samples of a binary function there exists a binary decison tree with at most $2m-1$ nodes consistent with the samples.*

Proof. In an optimal-size binary decision tree, there are at most m leaves, one for each sample. By induction, we can show that in any binary decision tree there is at most the number of leaves minus one internal node. □

Proposition 67. *There exists $a > 0$ such that if there exists a binary decision tree of size S computing a boolean function then there exists a Boolean circuit of size at most aS computing that function.*

Proof. Create $S-1$ nodes, one for each node of the tree except the root. Each node has its value defined by the AND of the parent node and of either the input-Boolean-variable associated with the parent node if it is the first child or of its negation if the second child. To achieve this each time an input-variable is needed, create a node associated with the input-variable in the circuit. In the case of the second child, one supplementary node associated with the unary logical operator NOT is used to compute the negation of the corresponding input-variable.

For an input x, the values obtained at the nodes that correspond to the leaves —let's denote them b^t for leaf t– are all 0 except for the leaf at which the procedure described in Definition 65 terminates.

A network of logical operators can aggregate the output associated with the leaves —let's denote them o^t for leaf t— and output the answer associated with the only leaf to which 1 has been associated.

To do so consider the following two Boolean variables of $x_1, x_2 \in \mathcal{B}^n$, with $x_1^0, x_2^0 = 0$, and with the following update for leaf number $t \in \mathbb{N}$ with associated output Boolean value o_t and Boolean node value at runtime b_t,

$$\begin{bmatrix} x_1^{t+1} \\ x_2^{t+1} \end{bmatrix} = \begin{bmatrix} \text{if } x_2^t \text{ then } x_1^t; \text{else } o^t \\ \text{if } x_2^t \text{ then } x_2^t; \text{else } b^t \end{bmatrix}. \tag{A.58}$$

This system iterated for all the leaves computes the output of the binary decision tree in x_1, and this iteration can be computed with a number of nodes that scale linearly with the number of leaves.

The final circuit size is in $O(S)$. □

Proposition 68. *There exists a constant $b > 0$ such that given m samples of a binary function there exists a Boolean circuit of size at most bm consistent with the samples.*

Proof. Merge the last two results, Propositions 66 and 67, to first produce a binary decision tree, then to convert it into a Boolean circuit of suitable size. □

1.6.2 A combinatoral proposition

An useful Definition and Proposition, it comes from Lint [1999].

Definition 69. *The* binary entropy function *H is defined by*

$$H(x) = \begin{cases} 0 & \text{if } x = 0; \\ -x\log_2 x - (1-x)\log_2(1-x),\ 0 < x \leq \frac{1}{2} & \text{else if } 0 < x \leq 1/2. \end{cases} \tag{A.59}$$

Proposition 70. *Let $0 \leq \epsilon \leq \frac{1}{2}$ and $M \in \mathbb{N}^+$, we have*

$$\sum_{0 \leq i \leq \lfloor \epsilon M \rfloor} \binom{M}{i} \leq 2^{MH(\epsilon)}. \tag{A.60}$$

B

Appendix of Chapter 3

2.1 Hyperparameters and computational resources

We refer to the reader to Figure 3.2 for the model architecture. We describe in Table B.1 the hyperparameters used with this architecture. The size of the dataset of trajectories is chosen to be sufficiently large to avoid overfitting.

We train the model with 2 days of compute on a `NVIDIA Quadro RTX 6000` with `amp` on. Generating the dataset offline takes 60 cpu hours. Running the tree search with 1250 expansions for 250 episodes (each of 200 steps) takes 3 hours without `amp`.

All the results using tree-search presented in the paper uses the hyperparameters in Table B.2. These hyperparameters are found using a Bayesian Optimization library implemented by Head et al. [2020] for the case with 1.250 expansions in the tree.

2.2 Model architecture

2.3 Tree search stochastic planning

Algorithm 6 Online tree search planning adapted from Schrittwieser et al. [2020] for stochastic dynamics.

Input: a learned model; a discount factor γ; the number of sampled stochastic branches by chance nodes K; exploration parameters for the selection C_1, C_2; temperature parameter for action probabilities T; a number of expansions to run.

Initialize a root node with the current state.
for all 1…# expansions **do**

Select
decision node ← root node
loop
$a^* \leftarrow$
$\arg\max_{a \in A} Q^{\text{node}}(a) + \frac{1}{|A|} \frac{\sqrt{\sum_a N^{\text{node}}(a)}}{1+N^{\text{node}}(a)} (C_1 + \log(\frac{C_2+1+\sum_a N^{\text{node}}(a)}{C_2}))$
chance node ← take child of decision node following a^* **if** it does not exist **break**
decision node ← sample child of chance node

Expand
for all 1…K **do**
Sample $s, r \leftarrow$ learned model(decision node state, a^*).
Create chance node and its children decision nodes with sampled states. Compare samples, merge duplicates by increasing the sampling weight w initialized to 1. Assign the new chance node as a child to the decision node at the end of the select stage following a^*.

Backup
$G \leftarrow 0$.
$r \leftarrow \sum_{(s,r,w) \in \text{branches of chance node}} w \cdot r / \sum_{(s',r',w') \in \text{branches of chance node}} w'$
$a \leftarrow a^*$
repeat
$G \leftarrow r + \gamma G$
$Q^{\text{decision node}}(a) \leftarrow \frac{Q^{\text{decision node}}(a) \cdot N^{\text{decision node}}(a) + G}{N^{\text{node}}(a)+1}$
$N^{\text{decision node}}(a) \leftarrow N^{\text{decision node}}(a) + 1$
chance node, $r \leftarrow$ parent of decision node and associated reward
decision node, $a \leftarrow$ parent of chance node and associated action
until root node is updated

Actions probabilities
return action probability of $a \in A \leftarrow N^{\text{root}}(a)^{1/T} / \sum_{a' \in A} N^{\text{root}}(a')^{1/T}$

Table B.1 List of hyperparameters for the model's architecture and its training.

Parameter	Value/method
#dimensions at the input/out of a transformer layer	256
#heads for attention	4
#transformer layers	3
#dimensions of the layer in the mlp of transformer layer	256
dropout in transformer	0.1
activations	relu
#layers in mlps at transformer output	1
#dimensions of the layers in mlps at transformer output	512
#dimensions of the hidden state in GRU	32
optimizer	Adam
learning rate	1e-4
batch size	250
#trajectories in dataset (each of 200 steps)	$1e6$
# training steps	$750 \times 1e3$

Table B.2 List of hyperparameters for the tree search algorithm.

Parameter	Value
Temperature (T)	0.55
C_1	0.57
C_2	16.15
Stochastic branching factor	3
γ	0.99

Table B.3 From Wang et al. [2021], list of publicly available hyperparameters for the V-MPO agent. The architecture is defined in Parisotto et al. [2020] and the training algorithm in Song et al. [2019].

Parameter	Value/method
agent discount	0.99
TrXL MLP size	256
TrXL number of layers	6
TrXL number of heads	8
TrXL Key/Value size	32
ϵ_η	0.5
ϵ_α	0.001
T_{target}	100
β_π	1.0
β_V	1.0
#episodes used	$1e9$

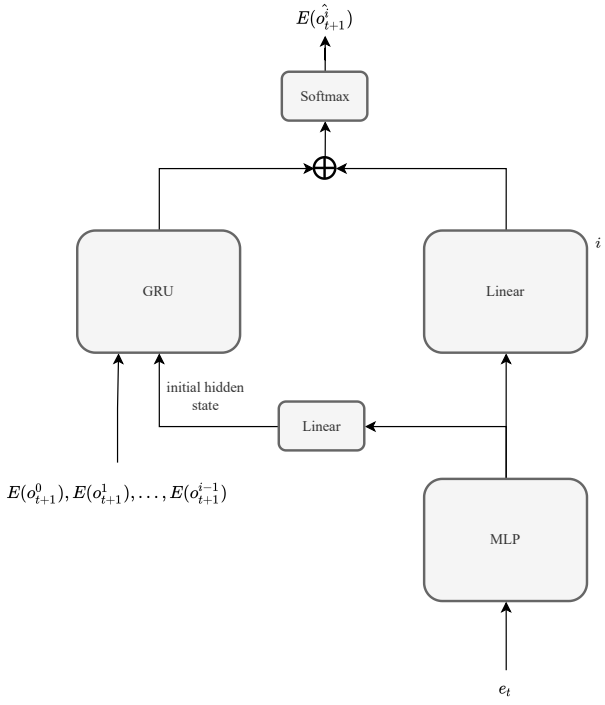

Fig. B.1 A more detailed view of the prediction head at training time using teacher forcing when predicting dimension i of the observation at timestep t. Each dimension of the observation space as its own linear head (on the right in the Figure), so we take the corresponding ith component.

C

Appendix of Chapter 4

3.1 Main Theorem proof

In this section, we provide the main Theorem proof, then compare it with the proof in Sun et al. [2019]. Finally, we state and prove a domination result inferred from our main Theorem.

Theorem 20. *There exists a family of RL problems such that*

1. *For any RL problem in the family and $\delta \in (0, 1)$, with probability at least $1 - \delta$ (over the sampled trajectories), Algorithm 3 outputs an optimal policy with a number of samples and number of operations upper bounded by a polynomial in horizon H and $1/\delta$.*
2. *For any algorithm satisfying Definition 19 and using $o(2^H)$ calls to the interface (Algorithm 2), there exists a problem in the family for which it outputs a suboptimal policy with probability at least $1/3$.*

Proof. We define the family of RL problems. The problems in the family are parametrized by a horizon H and by a hidden binary word b of length $H - 2$. Each state is defined by $3(H - 2) + 1$ real variables. The action space is binary. An example in this family for $H = 4$ and $b = 01$ is represented in Figure 4.2.

We pose some notations to describe the states. Each state is decomposed into three parts a, b, and c, the part b is further decomposed into

parts u and d. The time step to which the state belongs is kept implicit in our description. For i between 1 and $H-2$:

- $s^{a,i}$ the ith variable in the first part of the state vector;
- $s^{b,u/d,i}$ if u, the up ith variable in the second part of the state vector, if d, the down ith variable in the second part of the state vector;
- s^c the variable of the third part of the state vector.

Example of the notation for state vector s:

$$\left[\begin{matrix} s^{a,1}, s^{a,2}, \dots, s^{a,i}, \dots, s^{a,H-2}, \\ s^{b,d,1}, s^{b,d,2}, \dots, s^{b,d,i}, \dots, s^{b,d,H-2}, \end{matrix} \begin{matrix} s^{b,u,1} & s^{b,u,2} & & s^{b,u,i} & & s^{b,u,H-2} \\ & & \dots & & \dots & \end{matrix} s^c \right].$$

Now we describe the dynamics. At step 0, whatever action is taken, with probability $1/2$, we reach either a state full of zeros, or a state full of zeros except for s^c which equals one.

If we have $s^c = 0$, then for the steps $t = 1$ to $t = H-1$, the state vector stays zero. At the last step it gives a reward of 1 and the state with $s^a = 0^{H-2}$, $s^{b,u} = 1^{H-2}$, $s^{b,d} = 0^{H-2}$ and $s^c = 0$.

If $s^c = 1$, then from time step $t = 1$ to $t = H-2$, the transition from t to $t+1$ encodes the binary action taken into $s^{a,t}_{t+1}$ with zero reward (the other variables keep their values from the previous time step). At the last step, the dynamics fixes $s^c_H = 0$, all the variables that encode a past action to zero $s^a_H = 0^{H-2}$, and s^b_H to express if each action taken was correct $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ or not $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ with respect to a fixed optimal trajectory given by b the binary word parametrizing the problem. In other words, $s^{b,u,i}_H = \mathbf{1}(s^{a,i}_t = b_i)$ and $s^{b,d,i}_H = \mathbf{1}(s^{a,i}_t \neq b_i)$. At that step, if $s^{b,u}_H = 1^{H-2}$, then a reward of 1 is given. Such that, a reward is obtained only if all the actions taken follow the hidden binary word b.

We prove the first claim of the Theorem.

Take Algorithm 3 and fix any $\delta \in (0,1)$.

There exists a lower-bound polynomial in $\log 1/\delta$ on the number of sampled trajectories, K, such that for any horizon, H, with probability at least $1 - \delta/2$ the rewarding state is discovered with the left-hand dynamics ($s^c = 0$).

Moreover, we demonstrate in Lemma 76, that Algorithm 3 obtains, for $\alpha = 1$ and any K larger than some polynomial in H and $1/\delta$, with prob-

ability at least $1 - \delta/2$ over the sampled dataset, a goal-conditioned function which predicts correctly the action to take on the right-hand dynamics (when $s^c = 1$).

By choosing K sufficiently large and applying the union bound, the probability that one of these events fails is bounded by δ. With high probability, the policy $\pi(a|s_t) = f^t(a|s_t, g)$ will reach the rewarding state (and thus act optimally) when $s_t^c = 1$. When $s_t^c = 0$, the policy does not influence the expected return. The policy is thus optimal.

Moreover, this method can be efficiently implemented from a computational complexity point of view.

Now we prove the second claim. We will prove that the part of the problem where s^c equals one is equivalent to an armed-bandit problem family (Definition 77) with 2^{H-2} arms where only one arm gives a reward. Following Proposition 78, any algorithm using $o(2^H)$ calls will fail to identify the best arm with probability at least $1/3$ on some problem in the armed-bandit problems family. This entails the Theorem.

In the rest of the proof, we assume that we are on the branch where $s^c = 1$, the branch $s^c = 0$ is constant across the family for a fixed H and thus provides no information on b.

We prove that all the states at the last steps which have zero-reward are indistinguishable. Due to Definition 19, information on these states can only be obtained through evaluations of a function $\mathcal{F}$ with one of these states as the first argument (since these states are final, they cannot be added as direct input in the dataset D).

We prove that these evaluations are identical for all those states. These states only contain non-zero elements in coordinates which are always zero in the states present in the dataset. Moreover, they have exactly the same number of ones in those coordinates. Thus, $\mathcal{F}$, which is symmetric on permutations of the coordinates, cannot distinguish them. Formally, let be two final states on the right-hand side w, z, there exists a permutation of the coordinates p such that $w = p(z)$ and $s = p(s)$ for all states s present in the dataset D. This implies, for any $\mathcal{F}$, by the symmetry constraint on $\mathcal{F}$,

$$\begin{aligned}\mathcal{F}(z, ((s_0, a_0, r_0, f_0), \dots,)) &= \mathcal{F}(p(z), ((p(s_0), a_0, r_0, f_0), \dots)) \\ &= \mathcal{F}(w, ((s_0, a_0, r_0, f_0), \dots)).\end{aligned}$$

Thus, non-rewarding final states on the right-hand dynamics are indistinguishable for RL methods satisfying Definition 19. The family of right-hand dynamics is as hard as the family of armed-bandit problems for those

methods.

□

Similarly to Sun et al. [2019], we prove that for the studied class of methods, the problem family is indistinguishable from a hard family of black-box problems, implying the limitation. We adapt the proof to our class of black-box dynamics methods, by changing how information is hidden to prove the indistinguishability. Instead of hiding some states behind their evaluations with optimal value functions as in Sun et al. [2019], we hide them behind function evaluations corresponding to learned ML model outputs. We leverage the symmetries these models satisfy with respect to their constrained dataset to prove that some states are treated equivalently. Our interface also outputs pointers to states $\bar{s}$ but those do not disclose any information on the states.

Another point of comparison with the proof of Sun et al. [2019] is the addition of a trajectory allowing a method to discover a rewarding state. This is critical to get a method that can efficiently solve the presented family without an a priori goal-state. In parallel to this addition, the efficient method and the guarantee of its performance are also new to this proof. We refer to the introduction and Appendix 3.3 for a summary of what our results gain from these differences.

The following definitions and affirmations are classical results from VC dimension theory and can be found in the reference Shalev-Shwartz and Ben-David [2014].

Definition 71. *Hypothesis class.*
A set $\mathcal{H}$ is a hypothesis class if $\mathcal{H}$ is a set composed of functions from some domain $\mathcal{X}$ to $\{0,1\}$.

Definition 72. *Shattering.*
A finite set $S \subseteq \mathcal{X}$ is shattered by a hypothesis class $\mathcal{H}$ if for any function f from S to $\{0,1\}$, there exists $h \in \mathcal{H}$ such that for all $x \in S$ we have $f(x) = h(x)$.

Definition 73. *VC-dimension.*
Given a set $\mathcal{X}$, the VC-dimension of a hypothesis class defined on domain $\mathcal{X}$ is the maximal integer d such that there exists a subset of $\mathcal{X}$ of size d which is shattered by $\mathcal{H}$.

Proposition 74. *VC-dimension of the union.*
Given r hypothesis classes of VC-dimension at most d sharing the same domain, the VC-dimension of the union of these classes is at most $4d\log(2d) + 2\log(r)$.

Proposition 75. *Agnostic PAC-learnability from bounded VC-dimension.*
There exists a constant $C_1 \in \mathbb{R}$ such that for any $\delta, \epsilon \in (0,1)$, given a hypothesis class $\mathcal{H}$ of VC-dimension d, and a dataset $S \subset \mathcal{X} \times \{0,1\}$ of size at least $C_1 \frac{d+\log(1/\delta)}{\epsilon^2}$ composed of identically and independently drawn samples (x,y) according to a probability measure P on $\mathcal{X} \times \{0,1\}$, we have, with probability at least $1-\delta$ (over the sampled dataset),

$$L_P(\arg\min_{h \in \mathcal{H}} L_S(h)) \leq \min_{h' \in \mathcal{H}} L_P(h') + \epsilon,$$

where $L_P(h)$ is the error rate of classification according to measure P, $\mathbb{E}_{(x,y)\sim P}[\mathbf{1}(h(x) \neq y)]$, and $L_S(h)$ is the error rate on the dataset, $\frac{1}{|S|}\sum_{(x,y)\in S} \mathbf{1}(h(x) \neq y)$.

Lemma 76. *For Algorithm 3 with parameter $\alpha = 1$ and any $\delta \in (0,1)$ on the family presented in the proof of Theorem 20, there exists m a polynomial in H and $1/\delta$, such that with a dataset of trajectories larger than $m(H, 1/\delta)$ obtained by π^U, with probability at least $1-\delta$ over the sampled dataset, for all time steps $1 \leq t \leq H-2$ and with $s_t^c = 1$, the learned function $f^t(a|\, s_t, g)$ perfectly predicts the action to reach the goal g, if the state g is reachable from s_t.*

Proof. The proof can be decomposed in 4 main affirmations:

1. The classes of learning hypotheses have a VC-dimension upper-bounded by a polynomial in H.
2. For all the problems in the family, there exists a hypothesis with a low rate of errors.
3. A sufficiently low error rate implies a perfect accuracy on the predicted action to reach g when $s_t^c = 1$.
4. Set the last three points together to entail the Lemma.

We pose s_t as the current state, s_H as the state at the end of the trajectory after observing s_t. Thus, the learned function takes as input the vector $\begin{bmatrix} s_t \\ s_H \end{bmatrix}$ (concatenation of the real parts of the states without the time steps).

1. For $\alpha = 1$, at any time step, the class of hypotheses is the union of $O(H)$ sets of linear functions over 1 real-valued variable composed with a threshold. Those linear functions with a threshold have thus VC-dimension 2. Using Proposition 74, we infer that, for any time step, the VC-dimension of our hypothesis class is in $O(\log H)$.

2. Take the hypothesis that selects the variable $s_H^{b,u,t}$ and uses the linear identity function composed with a threshold $\mathbf{1}(s_H^{b,u,t} > 0)$.

Both sides of the environment ($s^c = 0$ or 1) have $1/2$ probability. Since all the samples on the right-hand dynamics are correctly predicted by this hypothesis and the samples on the left-hand side have $1/2$ probability to have either action, this hypothesis has an average error rate of $1/4$ on the distribution induced by π^U.

3. For all the possible feature selections, there is a finite set of possible inputs, for example, $\{0,1\}$ for feature $s_H^{b,u,i}$, or $\{0\}$ for feature $s_t^{b,d,i}$. Conditioned on $s_t^c = 1$, these inputs all have a constant probability of being sampled by π^U, thus the probability is independent of H and b the family parameters. We note the minimum on these probabilities $p_{\min}^{\text{input}}$, which is thus also independent of the family parameters H and b.

Similarly, we note $p_{\min}^{\text{output}}$ the minimal non-zero probability under π^U of any of the two actions conditioned on the selected feature from a state in the right-hand dynamics.

Suppose that the error rate under the distribution induced by π^U of hypothesis h for the goal-conditioned prediction problem is $1/4 + \epsilon$. If $\epsilon < 1/2 \cdot p_{\min}^{\text{input}} \cdot p_{\min}^{\text{output}}$ then the hypothesis does not make any errors in the right-hand dynamics. By contradiction, suppose that it does make a mistake on a pair of input-output, then take the value of the input of the variable selected by the hypothesis. There is at least $1/2 \cdot p_{\min}^{\text{input}}$ samples with that input and a $p_{\min}^{\text{output}}$ proportion of them with the same output. The hypothesis will thus make a mistake on a set of inputs of probability at least $1/2 \cdot p_{\min}^{\text{input}} \cdot p_{\min}^{\text{output}}$ under π^U on the right-hand dynamics. To which we have to add a $1/2$ error rate on the trajectories on the left-hand dynamics that have probability $1/2$.

4. To generalize the result to all the time steps at the same time, we use a union bound over the probability of failure of each time step.

Using Proposition 75 and the affirmations just proven, with any constant $\epsilon < p_{\min}^{\text{input}} \cdot p_{\min}^{\text{output}}/2$ and $\delta = \delta/H$, there exists $m(H,\delta) \in O(\log \frac{H}{\delta})$ such that the algorithm will output a hypothesis which makes no mistake on the right-hand dynamics. □

Definition 77. *A* deterministic armed-bandit problem *is composed of a finite set of arms $\mathcal{A}$ and a function $f : \mathcal{A} \to [0,1]$ which associates an arm to a*

deterministic reward.

Proposition 78. *Let be a family of deterministic armed-bandit problems defined by $A \in \mathbb{N}$ and $1 \leq i^* \leq A$. The deterministic armed-bandit problem defined by A and i^* has A arms, the arm i^* gives a reward of 1, and the others give a reward of 0.*

For any randomized algorithm that tries to identify the arm with the maximal associated reward with $o(A)$ calls to the reward associating function, there exists an A large enough and a i^* such that the algorithm fails with probability at least $1/3$ on the problem defined by A and i^*.

Proof. By contradiction, let's suppose that there exists a randomized algorithm Alg that solves any problem in the family with probability at least $1 - \delta$ for some $\delta \in (0, 1)$ using $K \in o(A)$ samples. Let us note $r \sim R$ the random variable following some probability distribution R upon which the algorithm depends, p the problem in input that it solves, and $\text{Alg}(p, r)$ the output of the algorithm with p and r in input. For any A, let $U(A)$ be the uniform probability distribution upon $\{1, \dots, A\}$ and $p^A_{i^*}$ the bandit problem defined by A and i^*. We have for any A

$$\mathbb{E}_{i^* \sim U(A)} \mathbb{E}_{r \sim R}[\mathbf{1}(\text{Alg}(p^A_{i^*}, r) = i^*)] \geq 1 - \delta. \tag{C.1}$$

Which implies

$$\mathbb{E}_{r \sim R} \mathbb{E}_{i^* \sim U(A)}[\mathbf{1}(\text{Alg}(p^A_{i^*}, r) = i^*)] \geq 1 - \delta. \tag{C.2}$$

There must exist some r which performs at least as well as the mean. Thus there exists a deterministic algorithm Alg_d such that $\mathbb{E}_{i^* \sim U(A)}[\text{Alg}_d(p^A_{i^*})] \geq 1 - \delta$.

We suppose w.l.o.g. that this algorithm uses its $K \in o(A)$ tries in the set of the K first arms. Then with probability $1 - \frac{K}{A}$ it only receives 0 rewards and must guess the best arm with no information for the $A - K$ left untested arms. Thus the deterministic algorithm has a probability of failure lower bounded by $1 - \frac{K}{A} - \frac{1}{A-K}$.

Since $K \in o(A)$, there exists A large enough such that this quantity is larger than δ. Thus we have a contradiction. □

3.1.1 Domination result

Here we derive from Theorem 20 a result on the domination of black-box dynamics RL methods.

We define $u(\pi, m) = \mathbb{E}_{(s_0,a_0,r_0,\ldots,s_H)\sim P^\pi}[\sum_{t=0}^{H-1} r_t]$ the expected cumulative reward of some policy π on RL problem m with associated operator P. Additionally, we define $U(A, m)$ the average expected cumulative reward of (randomized) RL method A on RL problem m. More formally, with π the (random) output of A on problem m, $U(A, m) = \mathbb{E}_\pi [u(\pi, m)]$.

Corollary 79. *For any RL method A taking at most polynomial time in the horizon of the problem and satisfying Definition 19 there exists an RL method B that takes at most polynomial time in the horizon and $1/\epsilon$ with $\epsilon \in (0, 1)$ such that*

1. *For any RL problem m, $U(A, m) \leq U(B, m) + \epsilon$;*
2. *There exists an RL problem m such that $U(A, m) \leq U(B, m) - 1/12$.*

Conversely, there exists some $\epsilon \in (0, 1)$ and an RL method B taking at most polynomial time in the horizon of the problem such that there does not exist a method A satisfying Definition 19 and that for any RL problem m satisfies $U(B, m) \leq U(A, m) + \epsilon$.

Proof. For any RL method A satisfying the conditions of the statement, construct the RL method B performing the following computations for an RL problem of horizon H:

1. Find policy π_{GC} by executing Algorithm 3 with parameters $\alpha = 1$ and N sufficiently large to have a maximal failure probability of $\delta_0 = 1/24$ for problems in the family in the proof of Theorem 20 with horizon H.
2. Evaluate the performance of π_{GC} with an accuracy of $\min\{\epsilon/4, 1/20\}$ with a success probability of at least $1 - \delta/2$ with $\delta = \min\{\epsilon/2H, 1/24\}$.
3. Execute method A on the RL problem, take output policy π_A.
4. Evaluate the performance of π_A with an accuracy of $\min\{\epsilon/4, 1/20\}$ with a success probability of at least $1 - \delta/2$ with $\delta = \min\{\epsilon/2H, 1/24\}$.
5. Return the policy with the largest estimated performance, call it π_B.

Step 1 takes polynomial time in the relevant quantities, we refer to the proof of Theorem 20. Step 3 is also efficient by the condition on method A. Steps 2 and 4 can be performed in polynomial time in the relevant quantities using Hoeffding's inequality. Thus the whole method is efficient (polynomial in the horizon and $1/\epsilon$).

For the first claim, consider m an input problem. By construction the policy returned by method B has expected cumulative rewards at least of

$\max\{u(\pi_{GC}, m), u(\pi_A, m)\} - \epsilon/2$ with probability $1 - \delta$. Thus we have the following lower-bound on the performance

$$\begin{aligned} u(\pi_B, m) &\geq (1-\delta)(\max\{u(\pi_{GC}, m), u(\pi_A, m)\} - \epsilon/2) + \delta \cdot 0 \\ &\geq (1-\delta)(u(\pi_A, m) - \epsilon/2) \\ &\geq u(\pi_A, m) - \delta u(\pi_A, m) + \delta\epsilon/2 - \epsilon/2 \\ &\geq u(\pi_A, m) - \delta H - \epsilon/2 \\ &\geq u(\pi_A, m) - \epsilon. \end{aligned} \tag{C.3}$$

We used the fact that the rewards are constrained in the interval $[0, 1]$ and thus the expected cumulative rewards are upper-bounded by H and lower-bounded by 0.

Since the probability distribution of π_A produced in method B is the same as the output of method A, we have

$$U(B, m) \geq E_{\pi_B}[u(\pi_B, m)] \geq E_{\pi_A}[u(\pi_A, m)] - \epsilon = U(A, m) - \epsilon. \tag{C.4}$$

This concludes the proof of the first claim.

For the second claim, method A satisfies Definition 19 and has computational complexity in $o(2^H)$. Thus method A satisfies the conditions of the second claim in Theorem 20. From the proof of Theorem 20, we can extract the fact that $U(A, m)$ is at most $5/6$ for some RL problem m in the family we defined[1].

While, by construction π_{GC} outputs an optimal policy with probability $1 - \delta_0$. Moreover, if π_{GC} is optimal, it is chosen as output π_B with probability at least $1 - \delta$ since the asked accuracy $1/20$ is below half the difference in expected returns for π_{GC} and π_A (lower-bounded by $1/12$).

Thus the expected returns of π_B satisfies, taking into account that for the problems in the family 1 is the optimal expected return achievable,

$$\begin{aligned} u(\pi_B, m) &\geq (1 - \delta - \delta_0) \cdot 1 + (\delta + \delta_0) \cdot 0 \\ &= 1 - 1/12. \end{aligned} \tag{C.5}$$

Thus,

$$U(B, m) - 1/12 \geq E_{\pi_B}[u(\pi_B, m)] - 1/12 \geq 1 - 1/12 - 1/12 \geq U(A, m). \tag{C.6}$$

[1]The method A has less than a $2/3$ probability to produce an optimal sequence of actions on the right-hand dynamics. Taking into account the two sides of the dynamics, the average expected cumulative rewards is thus at most $1/2 \cdot 1 + 1/2 \cdot 2/3 = 5/6$

The converse and last statement is a direct consequence of Theorem 20 proof by taking Algorithm 3 as B. □

3.2 Parallel result with a different assumption

This section presents another formalism than the main text, this formalism is closer to the paper of Sun et al. [2019]. We derive a parallel version of our theoretical result in this new formalism. Then, we discuss the advantages and inconveniences of this formalism in contrast to the one used in our main text.

This section aims to clarify the characteristics of different formalization choices. A second goal is to show that the ideas presented in this paper hold in the two formalisms; thus, their consequences are broader than what the main text formally presents.

3.2.1 The formalism and result of Sun et al. [2019]

We provide here briefly the result of Sun et al. [2019].

The following definition of a model-free method is given by Sun et al. [2019].

Definition 80. *Model-free algorithm, [Sun et al., 2019]. Given a (finite) function class $\mathcal{G} : (\mathcal{S} \times \mathcal{A}) \to \mathbb{R}$, define the $\mathcal{G}$-profile $\phi_{\mathcal{G}} : \mathcal{S} \to \mathbb{R}^{|\mathcal{G}|\times|\mathcal{A}|}$ by $\phi_{\mathcal{G}}(x) \stackrel{\text{def}}{=} [g(x,a)]_{g\in\mathcal{G},a\in\mathcal{A}}$. An RL method is model-free using $\mathcal{G}$ if it accesses x exclusively through $\phi_{\mathcal{G}}$ for all $x \in \mathcal{S}$ during its entire execution.*

The definition of a model-free method depends on a set of functions $\mathcal{G}$. For their theorem, they assume $\mathcal{G} = \mathrm{OP}(\mathcal{M})$, where OP stands for optimal planning, it is defined as the set of optimal Q-functions and policy functions for a given family of RL problems $\mathcal{M}$. In other words, the set $\mathrm{OP}(\mathcal{M})$ contains any policy and Q-function that is optimal for some problem in $\mathcal{M}$.

Depending on the family $\mathcal{M}$, this assumption implies a barrier to the information that can go from the encountered states to the model-free method. This loss of information allows them to prove their theorem on the gap in statistical complexity between a model-based method and model-free RL methods.

The work of Sun et al. [2019] proves that there exists a family of RL problems $\mathcal{M}$ such that:

- These problems are intractable in the horizon for RL methods that are model-free using $\mathcal{G} = \mathrm{OP}(\mathcal{M})$ following Definition 80.

Algorithm 7 Env : an interface linked to a RL problem defined by the transition operator P and a $\mathcal{G}$-profile $\phi_{\mathcal{G}}$ (Definition 80). The interface is a set of functions with an internal state. The interface has access to an initialized encoder/decoder of pointers and states as defined in Algorithm 1.

function Env_init($\mathcal{F}$)
  $s_0 \sim P_0$
  $\bar{s_0} \leftarrow$ encode_state(s_0)
  output $\bar{s_0}$, $\phi_{\mathcal{G}}(s_0)$
function Env_step($\bar{s}$, a)
  $s \leftarrow$ decode_pointer($\bar{s}$)
  if s is not terminal **then**
    $r, s' \sim P_{\text{dyn}}(r, s' | s, a)$
    $\bar{s'} \leftarrow$ encode_state(s')
    output r, $\bar{s'}$, $\phi_{\mathcal{G}}(s')$
  else if s is terminal **then**
    output $\perp$
function Env_eval_state(s, $\mathcal{F}$)
  output $\phi_{\mathcal{G}}(s)$
function Env_encode(s)
  output encode_state(s)

- These problems are solved efficiently in the horizon by an RL method that takes the family $\mathcal{M}$ as input.

3.2.2 An extension of the formalism

We extend their definition of a model-free method with an interface. Similarly to the first interface we defined in Algorithm 2, this interface uses the encoder-decoder (Algorithm 1) to hide the content of states while still allowing their manipulation. Thus, as for the previously defined interface, the methods can generate transitions and do local planning, such as tree-search.

Definition 81. *A* black-box dynamics RL method *interacts only with the interface defined in Algorithm 7 linked with the RL problem to solve and* $\mathcal{G} = \text{OP}(\mathcal{M})$ *for* $\mathcal{M}$ *a given set of problems containing the problem to solve.*

3.2.3 New result

We derive a new version of our theorem in this formalism. The structure of the result is the same: we provide a family of RL problems which are both

Algorithm 8 A second goal-conditioned algorithm.

Parameters: N: number of samples, α: parameter of the learning algorithm (number of active features)

Input: P: the operator corresponding to the RL problem to solve

$D \leftarrow \{\}$

for $i \leftarrow 1, \dots N$ **do**

$D \leftarrow D \cup \{(s_0, a_0, r_0, s_1, a_1, \dots, s_H) \sim P^{\pi^U}\}$

$g_T \leftarrow \arg\max_{s_T \in \mathcal{S}} r_T \quad \text{s.t.} \quad (\dots, s_T, a_T, r_T, \dots) \in D$

for $t \in \{0, \dots, H-1\}$ **do**

$D^t_{GC} \leftarrow \{((s_t, s_T), a_t) | (\dots, s_t, a_t, \dots, s_T, \dots) \in D\}$

$f^t \leftarrow$ the result of the optimization program 4.1 with dataset D^t_{GC} and parameter α.

output $\pi(a|s_t) = f^t(a|s_t, g_T)$

hard to solve for the class of RL methods that satisfies our definition, but is efficiently solved by another algorithm.

First, we define Algorithm 8. This method is similar to Algorithm 3 with two differences. One, the goal-state can be anywhere in the trajectory, while Algorithm 3 required it to be one of the final states. Two, the reward that decides the goal-state is not gained when entering the state, but when leaving it.

These changes are made to adapt to the new family of problems that will be used in the proof. Notice that Algorithms 3 and 8 both simply assign a state that appeared next to a high reward as a goal to reach.

Theorem 82. *There exists a family $\mathcal{M}$ of RL problems such that*

1. *For any problem in the family and $\delta \in (0, 1)$, with probability at least $1 - \delta$ (over the sampled trajectories), Algorithm 8 outputs an optimal policy with a number of samples and number of operations upper bounded by a polynomial in horizon H and $1/\delta$.*
2. *For any algorithm satisfying Definition 80 with the family $\mathcal{M}$ and using $o(2^H)$ calls to the interface (Algorithm 7), there exists a problem in the family for which it outputs a suboptimal policy with probability at least $1/3$.*

Proof. The family of RL problems we construct is the same as for the proof of Theorem 20 except in the last steps of the dynamics. An example of a problem in the family is represented in Figure C.1.

We describe the changes in comparison with the previous family of RL problems.

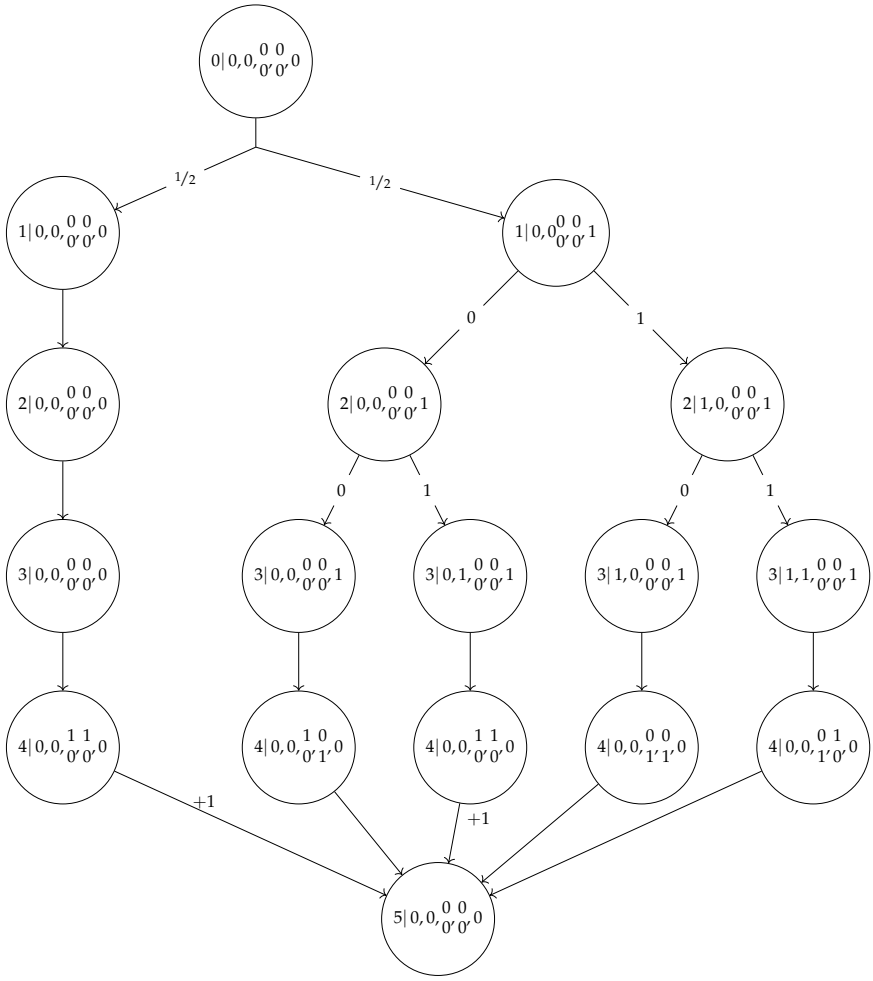

Fig. C.1 Example of the family of RL problems for horizon $H = 5$ and hidden binary word $b = 01$.

The rewards before entering the final states of the previous dynamics are all set to zero. A new step is added at the end of the dynamics where all states and actions lead to the same state full of zeros. The unique state that got a reward in the previous dynamics, leads now to a unitary reward in this last transition (the other states still lead to no reward).

For the first claim, the Algorithm 8 solves efficiently in the horizon the RL problems in the family, since the proof of Theorem 20 applies straight-

forwardly in this case. With high probability, the same goal state will be selected, and the same policy will then be learned.

We now prove the second claim. The left-hand dynamics is independent of b the hidden binary word defining the RL problem. Thus, the only information the method will receive is from the right-hand dynamics. This dynamics is the same as the problems in the work of Sun et al. [2019] and their proof applies.

We repeat the elements of the proof here for completeness.

For an algorithm satisfying Definition 81, the possible right-hand dynamics are equivalent to the family of problems defined in Proposition 78 with a set of 2^{H-3} actions, and the proposition implies exactly the claim to prove.

We prove the equivalence. First, we note that any states encountered before the $t = H - 2$ time step provide no information on the hidden binary word b. The same holds for the last state. Such that only the states at time step $t = H - 1$ are relevant to discover b.

Let us describe $\phi_{\mathcal{G}}$ with $\mathcal{G} = \mathrm{OP}(\mathcal{M})$, applied on these states. For a fixed horizon H, there exists as many different dynamics in $\mathcal{M}$, as the number of possible binary words b. Each of these dynamics entails both an optimal Q-value function and an optimal policy function. The outputs of policy functions contain no information since the dynamics are independent of the action at time step $t = H - 1$. The outputs of Q-value functions will evaluate to zero for all couples of states and actions at time step $t = H - 1$, except for the unique state leading to a reward, for which they will all evaluate to one.

Such that the output of $\phi_{\mathcal{G}}(s)$ will always be only zeros for all suboptimal states encountered at time step $t = H - 2$, and only be different for the unique rewarding state. All the suboptimal states will thus be indistinguishable for a method satisfying Definition 81. This is equivalent to the armed bandit problem described in Proposition 78. □

3.2.4 Discussion

The result obtained in this appendix is very similar to the one presented in the main text. This new parallel version of our theoretical result offers a new trade-off between the assumption made and the conclusion. We explain this trade-off.

We compare Definition 81 and linked Theorem 82 of this appendix section with Definition 19 and linked Theorem 20 presented in the main text.

As we explain and show in the main text and Appendix 3.4, the Definition 19 covers many common RL methods.

In contrast, the original assumption of Sun et al. [2019] (Definition 80 and the discussion below it) and our extension in Definition 81, pose conditions that do not seem to fit straightforwardly classical RL methods. These assumptions rely on looking states through a set of functions defined by $\mathrm{OP}(\mathcal{M})$, the set of optimal Q-functions and policy functions for a family $\mathcal{M}$ of RL problems. However, classical algorithms will use and evaluate encountered states using functions that cannot be linked to any arbitrary set of problems $\mathcal{M}$. For example, an algorithm could evaluate states with linear value functions but $\mathrm{OP}(\mathcal{M})$ might not contain these depending on the family $\mathcal{M}$ chosen.

Despite this problem, the assumption of Sun et al. [2019] and of this appendix still seems relevant. It seems counter-intuitive that a Q-learning methods needs to rely on sub-optimal Q-value functions to solve a problem.

If one accepts this argument, there is a gain on what can be proved. We believe that the family of RL problems used in the proof of Theorem 82 are easier to generalize than the ones in the proof of Theorem 20. The family of problems defined in this appendix sets the rewards after the states of interest not only just before, as pictured in Figure C.1. This allows a possible generalization, which is to extend the dynamics beyond the current last states and delay the gain in rewards. Thus, the goal-state used in our proof could become an intermediary sub-goal in a larger dynamics.

3.3 Comparison with the work of Sun et al. [2019]

The result in the main text and its parallel version in the previous section differ in two main ways from the result of Sun et al. [2019].

First, we redefine and broaden the class of methods with a limitation. Instead of proving a limitation on model-free methods only, we broaden the result of intractability to some methods that are usually considered model-based (such as AlphaZero). We obtain this by analyzing the computational complexity instead of the statistical complexity of the methods and by using an interface with an encoder-decoder of states. The interface allows more flexibility in the interactions between the method and the problem at hand, allowing us to abstract a larger class of methods. In particular, the interface allows to generate different potential future trajectories from a given state, something which is essential for several planning

algorithms with a known or learned model of the dynamics.

Our second contribution with respect to the result of Sun et al. [2019] is that our family of problems can be efficiently solved without strong specific priors about it. In contrast, their result crucially relies on an a priori known goal-state to reach. They encode this knowledge into a model-based method to its advantage and show that their model-free methods cannot leverage that prior information. In general, such knowledge cannot be assumed to be known in practice and impairs the practical applicability and generality of their claim.

Whereas in our result, no goal-state needs to be known a priori but it is discovered autonomously from obtained rewards. Thus, our result applies without needing to know and engineer strong a priori knowledge into an algorithm to get an advantage.

This leads to a new claim specific to our work: numerous classical RL methods of the literature have a fundamental limitation on problems that should be solvable by an efficient *general-purpose* RL method.

3.4 Classical RL methods implemented with the interface

In this Appendix, we reformulate common RL methods to use the interface defined in 2 instead of directly interacting with the RL problem dynamics. We thus demonstrate that these methods can be cast in the conditions of Definition 19 and suffer an upper bound on their performance provided by Theorem 20.

To clarify the presentation, we use methods as simplified as possible. For example, we only work with one-step RL methods, which use only one-step transition, (s, a, r, s'), instead of longer sequences to make updates. The presented formalism can however be extended to include these multi-step methods. Our results are robust to such changes.

Neural network initialization refers to the initialization of a multilayer perceptron (MLP), a classical neural network architecture that composes iteratively linear operators and non-linear point-wise activation functions. We keep implicit the input dimension, output dimension, and number of layers with their respective numbers of hidden units.

In Algorithm 9, we define the fitted Q-iteration algorithm with Deep Learning, a common variant of model-free deep Q-learning methods [Riedmiller, 2005]. The algorithm trains at each iteration a new Q-function based on the Bellman equation on a new dataset and the previously trained Q-function.

In Algorithm 10, we translate the original algorithm to use the interface defined in Algorithm 2. As explained in the main text, calls to draw trajectories from the RL problem are replaced by calls to Env_step. Instead of directly manipulating states and updating the neural networks, the call to Env_step uses function $\mathcal{F}$ to do these computations. The function $\mathcal{F}$ is used to learn and evaluate the neural networks (Q-value functions).

Following the argument formalized in Appendix 3.5 about the symmetries existing in neural networks training by gradient descent, the functions defined in $\mathcal{F}$ practically satisfy the symmetry condition of Definition 19.

We also implement an Actor-Critic method, defined in Algorithm 11, with the interface in Algorithm 12. This demonstrates the use of value and policy functions.

In Algorithm 13, we implement a local planning tree search procedure relying on the capacity of the interface to generate transitions at any given state.

It is possible to combine tree search and model-free methods to fit methods that leverage feedback from results of tree search procedures to train neural networks [Feinberg et al., 2018], then possibly improve the tree search procedure with these neural networks in a virtuous cycle [Anthony et al., 2017, Silver et al., 2017].

We fit the AlphaZero algorithm to our Definition 19 in Algorithms 16 and 17, the initial algorithm is described in Algorithms 14 and 15.

The original algorithm was described for 2 players zero-sum deterministic games with the use of self-play, but is straightforwardly appliable to deterministic RL problems. In addition, we note the following minor modifications with respect to the original AlphaZero described in Silver et al. [2016, 2017]:

- We evaluate the neural networks at each new node in the search-tree that is created, and not only at each leaf-node expanded. This incurs only a fixed factor more computations.

- Instead of using the final return of a full trajectory to learn the value function, we use one-step temporal differences. We do this because our Definition 19 does not support multi-steps temporal difference in the presented form. We note that multi-steps methods could be added to the formalism but at the cost of readability.

- To simplify the algorithm, we describe an online training version. We

Algorithm 9 Fitted Q-iteration (with ϵ-greedy exploration)

Parameters: H: horizon of the MDP, $\mathcal{S}$: state space, $\mathcal{A}$: action space, K: number of iterations, $\eta > 0$: scalar factor for gradient descent, I: number of samples by iteration (divisible by H), ϵ exploration parameter

Input: P: the operator corresponding to the MDP to solve

Initialize a neural network $Q_\theta : \mathcal{S} \to \mathbb{R}^{|A|}$

$\pi_\epsilon(a|\, s,\, Q) \leftarrow$
$(1-\epsilon)\mathbf{1}(a = \arg\max_{a \in A} Q(s,a)) + \epsilon/2\mathbf{1}(a = 0) + \epsilon/2\mathbf{1}(a = 1)$

for $k \leftarrow 1, \ldots, K$ **do**

  $\bar{Q} \leftarrow Q_\theta$

  Initialize a neural network Q_θ

  for $i \in 1, \ldots, I/H$ **do**

    $s \leftarrow P_0$

    for $t \leftarrow 0 \ldots H-1$ **do**

      $a \sim \pi_\epsilon(a|\, s,\, \bar{Q})$

      $r, s' \leftarrow P_{\text{dyn}}(r,\, s'|\, s,\, a)$

      $\theta \leftarrow \theta - \eta \nabla_\theta \left[Q_\theta(s,a) - r - \max_{a \in \mathcal{A}} \bar{Q}(s', a)\right]^2$

      $s \leftarrow s'$

$\pi(a|\, s) \leftarrow \pi_0(a|\, s,\, Q_\theta)$

output π

apply stochastic gradient descent directly with each new data sample, and do not construct datasets to leverage experience replay.

3.5 Symmetries in neural networks learning

We provide here a formal argument that classical neural network training procedures satisfy the condition in Definition 19 put on function $\mathcal{F}$.

We prove that the distribution of outputted functions by a gradient descent procedure respects such a symmetry.

In Definition 83, we construct a symmetry condition that will help us to state and to prove our result. In Theorem 84, we state the main result of this section on the symmetry of the learning procedure. Corollary 85 restate the Theorem, but in a form which can directly be compared with the condition on $\mathcal{F}$ stated in Definition 19.

For the clarity of the exposition, we will abusively refer to the probability of a function instead of the probability of events in the related implicit σ-algebra.

Algorithm 10 Fitted Q-iteration implemented with the interface defined in Algorithm 2. The function chunk(D, I) partitions the data sequence D into contiguous subsequences of length I until there is less than I elements left which are put in the last subsequence.

Parameters: H: horizon of the MDP, $\mathcal{S}$: state space, $\mathcal{A}$: action space, K: number of iterations, $\eta > 0$: scalar factor for gradient descent, I: number of samples by iteration, ϵ exploration parameter

Input: Env: the MDP interface with its methods defined in Algorithm 2

$\pi_\epsilon(a|\, Q_s) \leftarrow (1-\epsilon)\mathbf{1}(a = \arg\max_{a \in A} Q_s(a)) + \epsilon/2\mathbf{1}(a = 0) + \epsilon/2\mathbf{1}(a = 1)$

function learnQ(s, D)

Initialize a neural network $Q_\theta : \mathcal{S} \to \mathbb{R}^{|\mathcal{A}|}$

$D_1, \ldots, D_{n-1}, D_n \leftarrow \text{chunk}(D, I)$

for $(s_d, a_d, r_d, Q_{s'_d}) \leftarrow D_{n-1}$ **do**

$\theta \leftarrow \theta - \eta \nabla_\theta \left[Q_\theta(s_d, a_d) - r_d - \max_{a \in \mathcal{A}} Q_{s'_d}(a) \right]^2$

output $[Q_\theta(s, a)]_{a \in \mathcal{A}}$

$\mathcal{F} \leftarrow$ learnQ

for $k \leftarrow 1, \ldots, K \cdot I / H$ **do**

$\bar{s}, Q_s \leftarrow \text{Env\_init}(\mathcal{F})$

for $t \leftarrow 0, \ldots, H-1$ **do**

$a \sim \pi_\epsilon(a|\, Q_s)$

$r, \bar{s}, Q_s \leftarrow \text{Env\_step}(\bar{s}, a, \mathcal{F})$

$\pi(a|\, s) \leftarrow \mathbf{1}(a = \arg\max_{a \in A} \text{Env\_eval\_state}(s, \text{learnQ}))$

output π

Definition 83. *Symmetry condition.*

Let L be a function from a sequence of data points in $(\mathbb{R}^n \times \mathbb{R})^N$ for some natural number N, to a set of distributions over functions $\Delta(\{\mathbb{R}^n \to \mathbb{R}\})$.

For any sequence $d = ((x_0, y_0), \ldots)$ of data points and for any permutation of the coordinates p over $\mathbb{R}^n$ (Definition 18), we define $d_p = ((p(x_0), y_0), \ldots)$.

The function L satisfies the symmetry condition if, for any permutation of the coordinates p, we have $L(d) = L(d_p) \circ p^{-1}$. In other words, for any function $h : \mathbb{R}^n \to \mathbb{R}$, the output with d must give the same probability on h as the output with d_p on $h \circ p$.

Theorem 84. *The output of the randomized Algorithm 18, which produces a learned function given a dataset, follows a distribution defined by the dataset. The function between the dataset and the distribution of learned functions respects the*

Algorithm 11 Deep Policy Gradient with Value function learning (Actor-Critic).

Parameters: H: the horizon of the MDP, $\mathcal{S}$: state space, $\mathcal{A}$: action space, K: number of sampled trajectories, $\eta > 0$: scalar factor for gradient descent, I: number of iterations by initialization of the value function (should be divisible by H)

Input: P: the operator corresponding to the MDP to solve

Initialize a neural network $\pi_{\theta_1} : \mathcal{S} \to \Delta(\mathcal{A})$
Initialize a neural network $V_{\theta_2} : \mathcal{S} \to \mathbb{R}$
for $k \leftarrow 1 \dots K$ **do**
  if $k \cdot H \mod I = 0$ **then**
    $\bar{V} \leftarrow V_{\theta_2}$
    Initialize a neural network $V_{\theta_2} : \mathcal{S} \to \mathbb{R}$
  $s \leftarrow P_0$
  for $t \leftarrow 0, \dots, H-1$ **do**
    $a \sim \pi_{\theta_1}(a|s)$
    $r, s' \leftarrow P_{\text{dyn}}(r, s'|s, a)$
    $\theta_1 \leftarrow \theta_1 + \eta(r + \bar{V}(s'))\nabla_{\theta_1} \log \pi_{\theta_1}(a|s)$
    $\theta_2 \leftarrow \theta_2 - \eta\nabla_{\theta_2}\left[V_{\theta_2}(s) - r - \bar{V}(s')\right]^2$
    $s \leftarrow s'$
output π_{θ_1}

symmetry condition.

Proof. We prove by induction that for any permutation of the coordinates p both the process that constructs the linear operator represented by W and the processes that produce all the other functions of the input in the neural networks (output, gradients) satisfy the symmetry condition.

Let P be the permutation matrix corresponding to p.

At initialization, the symmetry condition is satisfied everywhere. The matrix W is as probable as WP^{-1} and is independent of the dataset. All the other functions computed in the neural network depend on the input only through the linear operator that satisfies the condition, such that they also satisfy the symmetry condition.

Now we look at the updates. Let g be the gradient at the output of the linear operator for some input x. Then the update of W is

$$W \leftarrow W - \eta g x^T. \tag{C.7}$$

On the transformed dataset, with WP^{-1} the matrix of the linear operator,

Algorithm 12 Deep Policy Gradient with Value function learning (Actor-Critic) implemented with the interface defined in Algorithm 2. The function chunk(D, I) partitions the data sequence D into contiguous subsequences of length I until there is less than I elements left which are put in the last subsequence. We simplify the learning of V, the value function, following the discussion in Section 3.5.

Parameters: H: horizon of the MDP, $\mathcal{S}$: state space, $\mathcal{A}$: action space, K: number of sampled trajectories, $\eta > 0$: scalar factor for gradient descent, I: number of iterations by initialization of the value function

Input: Env: the MDP interface with its methods defined in Algorithm 2

function learningVand$\pi(s,\ D)$
  Initialize a neural network $V_{\theta_2} : \mathcal{S} \to \mathbb{R}$
  $D_1, \ldots, D_{n-1}, D_n \leftarrow \text{chunk}(D,\ I)$
  for $(s, a, r, (V_{s'}, \pi_{s'})) \leftarrow D_{n-1}$ **do**
    $\theta_2 \leftarrow \theta_2 - \eta \nabla_{\theta_2} \left[V_{\theta_2}(s) - r - V_{s'}\right]^2$
  Initialize a neural network $\pi_{\theta_1} : \mathcal{S} \to \mathbb{R}^{|A|} (= \Delta(\mathcal{A}))$
  for $(s, (a, r, (V_{s'}, \pi_{s'}))) \leftarrow D$ **do**
    $\theta_1 \leftarrow \theta_1 + \eta(r + V_{s'}) \nabla_{\theta_1} \log \pi_{\theta_1}(a|\, s)$
  output $(V_{\theta_2}(s), \pi_{\theta_1}(s))$
$\mathcal{F} \leftarrow$ learningVandπ
for $k \leftarrow 1 \ldots K$ **do**
  $\bar{s}, (V_s, \pi_s) \leftarrow \text{Env\_init}(\mathcal{F})$
  for $t \leftarrow 0 \ldots H-1$ **do**
    $a \sim \pi_s(a)$
    $r, \bar{s}, (V_s, \pi_s) \leftarrow \text{Env\_step}(\bar{s},\ a,\ \mathcal{F})$
$\pi(a|\, s) \leftarrow [(V_s, \pi_s) \leftarrow \text{Env\_eval\_state}(s, \mathcal{F}); \quad \textbf{output}\ \pi_s(a)]$
output π

Px the new input, and h the new gradient, the update becomes

$$WP^{-1} \leftarrow WP^{-1} - \eta h(Px)^T, \tag{C.8}$$

which reduces to $(W - \eta h x^T)P^{-1}$. By the inductive hypothesis, we know that the processes that produce h respect the symmetry condition, thus the gradient h is sampled from the same distribution as g. Also, by the inductive hypothesis, the previous WP^{-1} is as probable as the previous W under their respective distributions. Consequently, the updated W is as probable as the updated WP^{-1}.

Similarly to the initialization, the processes that produce the rest of the

Algorithm 13 Tree-search implementation using the interface defined in Algorithm 2. This algorithm can be combined with the other methods presented, to use value function estimates instead of rewards for example.

Parameters: H: horizon of the MDP, $\mathcal{A}$: action space, $H_S > 0$: search horizon of the tree

Input: Env: the MDP interface with its methods defined in Algorithm 2

```
function TREE_SEARCH(s̄, h)
    if h = 0 then
        output _, 0
    for a ∈ 𝒜 do
        r, s̄', _ ← Env_step(s̄, a, _)
        if r = 1 then
            output a, 1
        else
            a', R ← Tree_search(s̄', h − 1)
            if R = 1 then
                output a, 1
    output _, 0
function TREE_SEARCH(s_t)
    a, r ← Tree_search(Env_encode(s_t), min{H_S, H − t})
    output a
π(a| s_t) ← Tree_search(s_t)
output π
```

network functions of the input respect the symmetry condition because they depend on the input only through the output of the linear operator since the start of the algorithm.

□

Corollary 85. *The expectation of the distribution of the output of neural networks applied on some input $z \in \mathbb{R}^n$ produced by randomised Algorithm 18 for some sequence of points d can be written as $f(z, ((x_0, y_0), \ldots))$ for some function f that is invariant to a permutation of the coordinates applied to z and $x_0, \ldots$*

Proof. By construction, f exists and is simply the function representing the process producing the distribution of learned functions composed with the expectation of their evaluations. Since the distribution of learned functions respects the symmetry condition by Theorem 84, we know that for any

permutation of the coordinates p, we have

$$\mathbb{E}_{\mathbf{nn}\sim\mathbf{Alg}(d)}[\mathbf{nn}(z)] = \mathbb{E}_{\mathbf{nn}\sim\mathbf{Alg}(d_p)}[\mathbf{nn}(p(z))]$$

where **nn** is the neural network trained by the algorithm **Alg**. □

3.6 A planning method

In this appendix, we describe the planning method introduced in the numerical experiments section of the main text.

The method follows these steps:

1. Sample an initial dataset of trajectories using a uniform policy.
2. Learn a model of the forward dynamics by fitting the dataset with Decisions Trees auto-regressively assuming a deterministic transition dynamics. If a learned model of the dynamics already existed from previous iterations combine the previously learned and new Decision Trees predicting rewards to produce a pessimistic prediction of obtained rewards. In other words, the method combines additively the constraints predicting zero rewards in the trees.
3. Sample trajectories following a policy that plans each step using the learned model of the dynamics. The planning is done by solving a Mixed-Integer Linear Programming (MILP) optimization problem. The main variables in this program represent the future trajectory (states, actions, rewards), there are also auxiliary variables that help encode the constraints coming from the learned dynamics. The constraints of the MILP are given by: the current state and the model of the dynamics defined by Decision Trees. The objective is to maximize the total return of the trajectory. To solve the MILP, we use the off-the-shelf solver Gurobi [Gurobi Optimization, LLC, 2023].
4. If the sampled trajectories reach reach a threshold in average returns, the method outputs the policy that plans each action by solving a MILP encoding the learned dynamics (as in the last step). Else, if the performance is not sufficient, add the trajectories to the dataset and the method goes back to step 2.

An algorithmic description of the method that represents the loops and branching statements more clearly is provided in Algorithm 19.

The method requires a sufficiently deterministic dynamics. The dynamics should also be simple enough for Decision Trees to be able to efficiently model it with a dataset of trajectories produced with a uniform policy, followed by a low exploration with several trajectories that are promising according to the dynamics model. This low exploration comes from the compromise of choosing a pessimistic prediction of the rewards.

As for Algorithm 3 this method makes several assumptions about the simplicity of the problems it solves. Similarly, those assumptions are not specific to the problems in this paper, but prevent the method from being a general-purpose RL method. The method only illustrates an algorithmic approach to RL on our problems and shows that these problems are easy to solve.

3.7 Numerical experiments details

We describe the implementation of the different methods we test: a goal-conditioned method in Algorithm 20, fitted Q-iteration [Riedmiller, 2005] in Algorithm 21, Proximal Policy Optimization (PPO) [Schulman et al., 2017], and AlphaZero [Silver et al., 2017] in Algorithm 15. We refer to the previous section for details on the planning method introduced in the numerical experiments section of the main text.

We test all these algorithms with MLP neural networks. Any state in input to a neural network $s = (t, x) \in \mathcal{S} = \{(t, x) \in \{0, \dots, H\} \times \mathbb{R}^n\}$ will be represented with a one-hot encoding of the time step t concatenated to the real part x.

Some notations, the function $\mathrm{CE} : \Delta(X) \times X \to \mathbb{R}$ (Cross-Entropy) computes $\mathrm{CE}(q, x) = -\log q(x)$ for some set X. The function $\mathrm{clip}(x \in \mathbb{R}, a \in \mathbb{R}, b \in \mathbb{R})$ computes $\min\{\max\{x, a\}, b\}$. The method copy copies the object, it allows to fix its state.

The optimizations are performed with mini-batches estimation of the gradient, and AdamW, a regularized version of Adam [Loshchilov and Hutter, 2017]. Hyper-parameter optimization was performed with Bayesian Optimization to maximize success probability on problems with horizon $H = 45$ for the goal-conditioned algorithm, $H = 12$ for the fitted Q-iteration algorithm, and $H = 17$ for the PPO algorithm. For AlphaZero, we performed coordinate ascent optimization to tune the local search parameters for $H = 15$.

The goal-conditioned algorithm was run with 2 hidden layers of 256 units each, parameter $I = 1000$, 30000 steps of AdamW with learning rate

$2 \cdot 10^{-2}$, weight decay 10^{-3}, and 100 batches.

The fitted Q-iteration algorithm was run with 2 hidden layers of 512 units each, for $K = 50$ iterations, with $I = 1000$ sampled trajectories by iteration, $\epsilon = 0.2$, 1000 steps of AdamW by iteration, learning rate $1 \cdot 10^{-3}$, weigh decay 10^{-2}, each 1000 trajectories samples were divided into 20 batches.

The PPO algorithm was run with 2 hidden layers of 512 units each, for $K = 50$ iterations, with $I = 1000$ trajectories sampled by iterations, clipping parameter $\epsilon = 0.1$, $\beta = 10^{-3}$, 500 steps of AdamW by iteration, learning rate of $2 \cdot 10^{-4}$, weight decay 10^{-8} and 10 batches of the sampled trajectories.

The AlphaZero algorithm was run with 2 layers of 256 neurons each, AdamW with learning rate $2 \cdot 10^{-3}$ and weight decay $1 \cdot 10^{-3}$. The algorithm performed the following operation three times: sampling a dataset of 1000 trajectories, then optimize an actor-critic for 30.000 steps with AdamW. For the local search procedure, we fixed a budget of 50 search node per decision and tuned the c and τ parameters (see Algorithm 15).

We measured that it takes less than two days with a single GPU to reproduce the results provided in the numerical section with only one test by experiment (pair of horizon and method).

Algorithm 14 AlphaZero planning functions.

Parameters: H: the horizon of the MDP, $\mathcal{S}$: state space, $\mathcal{A}$: action space

Input: P: the operator corresponding to the deterministic MDP to solve

Node: N, W, Q, P, v, s, children, parent, parent_action, leaf, h

function PLANNING($(s, h, \pi_{\theta_1}, V_{\theta_2}, K, c, \tau)$)

nodes $\leftarrow$ []

nodes.append(
Node($0^{\mathcal{A}}$,$0^{\mathcal{A}}$,$0^{\mathcal{A}}$,null,null,s,null,null,null,True,h))

for $k \leftarrow 1 \dots K$ **do**

last_node$\leftarrow$Select(nodes[0], c)

nodes.concatenate(Expand(last_node, π_{θ_1}, V_{θ_2}))

Backup(last_node.parent, last_node.v)

output root_node.N[a]$^{1/\tau}$ / $\sum_{a \in \mathcal{A}}$root_node.N[a]$^{1/\tau}$

function SELECT(node, c)

if node.leaf or node.h equal H **then**

output node

$a \leftarrow \arg\max_{a \in \mathcal{A}}$curr_node.Q[a] + c node.P[a] $\frac{\sqrt{\sum_{b \in \mathcal{A}} \text{node.N}[b]}}{1+\text{node.N}[a]}$

output Select(node.children[a])

function EXPAND(node, π_{θ_1}, V_{θ_2})

node.leaf$\leftarrow$ False

if node.h equal H **then**

stop

node.children $\leftarrow$ []

for $a \in \mathcal{A}$ **do**

r,s'$\leftarrow$ $P_{\text{dyn}}(r, s' | \text{node}.s, a)$

v'$\leftarrow$ V_{θ_2}(s')

node.children.append(
Node($0^{\mathcal{A}}$,$0^{\mathcal{A}}$,$0^{\mathcal{A}}$,null,v',s',null,node,a,True,node.h+1))

node.P$\leftarrow$ π_{θ_1}(node.s)

output node.children

function BACKUP(node, a, v)

if node is null **then**

stop

node.N[a] $\leftarrow$ node.N[a]+1

node.W[a] $\leftarrow$ node.W[a] + v

node.Q[a] $\leftarrow$ node.W[a]/node.N[a]

Backup(node.parent, node.parent_action, v)

Algorithm 15 AlphaZero.

Parameters: H: the horizon of the MDP, $\mathcal{S}$: state space, $\mathcal{A}$: action space, I: number of iterations of the procedure, K: number of expansions in the search, $\eta > 0$: scalar factor for gradient descent, τ temperature for action selection, c exploration parameter of the search

Input: P: the operator corresponding to the deterministic MDP to solve

Initialize a neural network $\pi_{\theta_1} : \mathcal{S} \to \Delta(\mathcal{A})$

Initialize a neural network $V_{\theta_2} : \mathcal{S} \to \mathbb{R}$

for $i \leftarrow 1 \dots I$ **do**

  $s \leftarrow P_0$

  for $h \leftarrow 0, \dots, H-1$ **do**

    $p \leftarrow \text{planning}(s, h, \pi_{\theta_1}, V_{\theta_2}, K, c, \tau)$

    $\theta_1 \leftarrow \theta_1 + \eta \nabla_{\theta_1} \sum_{a \in \mathcal{A}} p(a) \log \pi_{\theta_1}(a|s)$

    $a \sim p$

    $r, s' \leftarrow P_{\text{dyn}}(r, s'|s, a)$

    $\theta_2 \leftarrow \theta_2 - \eta \nabla_{\theta_2} \left[V_{\theta_2}(s) - r - \bar{V}(s') \right]^2$

    $s \leftarrow s'$

$\pi(a|s) \leftarrow \left[p \leftarrow \text{planning}(s, s.h, \pi_{\theta_1}, V_{\theta_2}, K, c, \tau); \quad \textbf{output } p(a) \right]$

output π

Algorithm 16 AlphaZero planning functions implemented with the interface defined in Algorithm 2, fitting Definition 19.

Parameters: H: the horizon of the MDP, $\mathcal{S}$: state space, $\mathcal{A}$: action space

Input: Env: the MDP interface defined in Algorithm 2

Node: N, W, Q, P, v, $\bar{s}$, children, parent, parent_action, leaf, h

function PLANNING($\bar{s}, h, \mathcal{F}, K, c, \tau$)

nodes $\leftarrow$ []

$(v_s, \pi_s, p) \leftarrow$ Env_eval_state$(s, \mathcal{F})$

nodes.append(Node($0^{\mathcal{A}}$,$0^{\mathcal{A}}$,$0^{\mathcal{A}}$,π_s,null,$\bar{s}$,null,null,null,True,h))

for $k \leftarrow 1 \dots K$ **do**

last_node$\leftarrow$Select(nodes[0], c)

nodes.concatenate(Expand(last_node, $\mathcal{F}$))

Backup(last_node.parent, last_node.v)

output root_node.N[a]$^{1/\tau}$ / $\sum_{a \in \mathcal{A}}$root_node.N[a]$^{1/\tau}$

function SELECT(node, c)

if node.leaf or node.h equal H **then**

output node

$a \leftarrow \arg\max_{a \in \mathcal{A}}$curr_node.Q[a] + c node.P[a]$\frac{\sqrt{\sum_{b \in \mathcal{A}} \text{node.N}[b]}}{1+\text{node.N}[a]}$

output Select(node.children[a])

function EXPAND(node, $\mathcal{F}$)

node.leaf$\leftarrow$ False

if node.h equal H **then**

stop

node.children $\leftarrow$ []

for $a \in \mathcal{A}$ **do**

$r, \bar{s}', (v_{s'}, \pi_{s'}, p_s) \leftarrow$

Env_step(node.$\bar{s}$, a, $\mathcal{F}$, append_to_data $=$ False)

node.children.append(

Node($0^{\mathcal{A}}$,$0^{\mathcal{A}}$,$0^{\mathcal{A}}$,$\pi_{s'}$,$v_{s'}$,$\bar{s}'$,null,node,a,True,node.h+1))

output node.children

function BACKUP(node, a, v)

if node is null **then**

stop

node.N[a] $\leftarrow$ node.N[a]+1

node.W[a] $\leftarrow$ node.W[a] + v

node.Q[a] $\leftarrow$ node.W[a]/node.N[a]

Backup(node.parent, node.parent_action, v)

Algorithm 17 AlphaZero implemented with the interface defined in Algorithm 2, fitting Definition 19.

Parameters: H: the horizon of the MDP, $\mathcal{S}$: state space, $\mathcal{A}$: action space, I: number of iterations of the procedure, K: number of expansions in the search, $\eta > 0$: scalar factor for gradient descent, τ temperature for action selection, c exploration parameter of the search

Input: Env: the MDP interface with its methods defined in Algorithm 2

function LEARNVAND$\pi(s, D|\, p)$
  Initialize a neural network $V_{\theta_2} : \mathcal{S} \to \mathbb{R}$
  for $(s, a, r, (V_{s'}, \pi_{s'}, p_{s'}) \leftarrow D$ **do**
    $\theta_2 \leftarrow \theta_2 - \eta \nabla_{\theta_2} \left[V_{\theta_2}(s) - r - V_{s'}\right]^2$
  Initialize a neural network $\pi_{\theta_1} : \mathcal{S} \to \mathbb{R}^{|\mathcal{A}|} (= \Delta(\mathcal{A}))$
  for $(s, a, r, (V_{s'}, \pi_{s'}, p_s)) \leftarrow D$ **do**
    $\theta_1 \leftarrow \theta_1 + \eta \nabla_{\theta_1} \sum_{a \in \mathcal{A}} p_s(a) \log \pi_{\theta_1}(a|\, s)$
  output $(V_{\theta_2}(s), \pi_{\theta_1}(s), p)$

$p \leftarrow 0$
$\mathcal{F} \leftarrow$ learnVand$\pi(.,.|\, p)$
for $i \leftarrow 1 \dots I$ **do**
  $\bar{s}, (V_s, \pi_s, p_s) \leftarrow$ Env_init$(\mathcal{F})$
  for $h \leftarrow 0, \dots, H-1$ **do**
    $p \leftarrow$ planning$(\bar{s}, h, \mathcal{F}, K, c, \tau)$
    $a \sim p$
    $\mathcal{F} \leftarrow$ learnVand$\pi(.,.|\, p)$
    $\_, \bar{s}, \_ \leftarrow$ Env_step$(\bar{s},\, a,\, \mathcal{F})$
$\pi(a|\, s) \leftarrow$
$[p \leftarrow$ planning(Env_encode$(s), s.h,$ learnVand$\pi(.,.|\, 0), K, c, \tau)$; **output** $p(a)]$
output π

Algorithm 18 Neural network training for a neural network with parameters $\theta = (W \in \mathbb{R}^{m \times n}, \theta' \in \mathbb{R}^k)$ for some $n, m, k \in \mathbb{N}$, which is interpreted in $\mathrm{nn}_\theta(x) = q_{\theta'}(Wx)$ for q that outputs a real and is smooth w.r.t. θ' and its input.

Parameters: $\eta > 0$: a scalar factor for gradient descent, loss: $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$: a smooth loss function w.r.t its inputs

Input: D: a sequence of couples $(x \in \mathbb{R}^n, y)$

Initialize θ' and matrix W. The elements of the matrix W are initialized i.i.d.

for $(x, y) \in D$ **do**

$\theta \leftarrow \theta - \eta \nabla_\theta \mathrm{loss}(q_{\theta'}(Wx), y)$

output $\mathrm{nn}_{\theta=(W,\theta')}$

Algorithm 19 A planning algorithm.

Parameters: N: number of initial samples.

Input: P: the transition operator corresponding to the RL problem to solve.

Create dataset D by sampling N trajectories from the uniform policy p^{π^U}.

function planning(s, DTs)

Build a Mixed-Integer Linear Program (MILP) that maximizes the future rewards given the current state s and the learned dynamics given by Decision Trees DTs.

Solve the mathematical program using an off-the-shelf MILP solver.

output a plan, consisting in a sequence of actions, states and rewards.

while True **do**

Fit autoregressively Decision Tree classifiers DTs on the trajectories in D. Accumulate previously trained Decision Trees for the reward predictions (pessimistic prediction).

Draw trajectories with P by computing at each step a plan with planning(s, DTs) and using the planned next action.

If the computed plans reach a sufficient performance threshold **break**.

Add the sampled trajectories to D.

output a policy that computes a plan at each step given the learned dynamics model: planning$(., \mathrm{DTs})$

Algorithm 20 Neural goal-conditioned algorithm.

Parameters: $\mathcal{S}$: state space, $\mathcal{A}$: action space, I: number of samples

Input: P: the operator corresponding to the MDP to solve

$D \leftarrow \{\}$

for $i \leftarrow 1, \dots I$ **do**

$D \leftarrow D \cup \{(s_0, a_0, r_0, s_1, a_1, \dots, s_H) \sim P^{\pi^U}\}$

$D \leftarrow \{(s_t, a_t, s_H) | \, (\dots, s_t, a_t, \dots, s_H) \in D\}$.

Initialize neural network $f_\theta : \mathcal{S} \times \mathcal{S} \to \Delta(\mathcal{A})$

$g \leftarrow \arg\max_{s_H} r \quad \text{s.t.} \quad (\dots, r, s_H) \in D$.

$\theta \leftarrow \arg\min_\theta \frac{1}{|D|} \sum_{(s,a,s_H) \in D} \mathrm{CE}(f_\theta([s, s_H]), a)$

output $\pi(a|s) = f_\theta(a|s, g)$

Algorithm 21 Fitted Q-iteration

Parameters: H: horizon of the MDP, $\mathcal{S}$: state-space, $\mathcal{A}$: action space, K: number of iterations, I: number of trajectory samples by iteration, ϵ: exploration parameter

Input: P: the operator corresponding to the MDP to solve

$\pi_\epsilon^Q(a|\, s) \leftarrow$

$\qquad (1-\epsilon)\mathbf{1}(a = \arg\max_{a \in A} Q(s,a)) + \epsilon/2\mathbf{1}(a = 0) + \epsilon/2\mathbf{1}(a = 1)$

Initialize a neural network $Q_\theta : \mathcal{S} \to \mathbb{R}^{|\mathcal{A}|}$

$D \leftarrow \{\}$

for $k \leftarrow 1, \dots, K$ **do**

$D' \leftarrow \{\}$

for $i \leftarrow 1, \dots, I$ **do**

$D' \leftarrow D' \cup \{(s_0, a_0, r_0, s_1, \dots, s_H) \sim P^{\pi_\epsilon^{Q_\theta}}\}$

$D \leftarrow D \cup D\{(s, a, r, s') | \, (\dots, s, a, r, s', \dots) \in D\}$

$\bar{Q} \leftarrow \mathrm{copy}(Q_\theta)$

$\theta \leftarrow \arg\min_\theta \frac{1}{|D|} \sum_{(s,a,r,s') \in D} [Q_\theta(s,a) - r - \max_{a \in \mathcal{A}} \bar{Q}(s', a)]^2$

output $\pi_0^{Q_\theta}(a|\, s)$

Algorithm 22 Proximal Policy Optimization

Parameters: H: horizon of the MDP, $\mathcal{S}$: state-space, $\mathcal{A}$: action space, K: number of iterations, I: number of trajectory samples by iteration, ϵ: clipping parameter, β: weight of the entropy regularization

Input: P: the operator corresponding to the MDP to solve

function PPO_loss(s, a, $R|\, \pi_{\theta_1}$, $\bar{\pi}$, $\bar{V}$)

$A \leftarrow R - \bar{V}(s)$

$r_\pi \leftarrow \frac{\pi_{\theta_1}(a|s)}{\bar{\pi}(a|s)}$

output $\min\{r_\pi A, \text{clip}(r_\pi, 1-\epsilon, 1+\epsilon)A\} + \beta \cdot \text{entropy}(\pi_{\theta_1}(.|s))$

Initialize a neural network $\pi_{\theta_1} : \mathcal{S} \to \Delta(\mathcal{A})$

Initialize a neural network $V_{\theta_2} : \mathcal{S} \to \mathbb{R}$

for $k \leftarrow 1, \ldots, K$ **do**

$D \leftarrow \{\}$

for $i \leftarrow 1, \ldots, I$ **do**

$D \leftarrow D \cup \{(s_0, a_0, r_0, s_1, \ldots, s_H) \sim P^{\pi_{\theta_1}}\}$

$D \leftarrow \{(s_t, a_t, \sum_{j=t}^{H-1} r_j) |\, (\ldots, s_t, a_t, r_t, \ldots) \in D\}$

$\bar{\pi} \leftarrow \text{copy}(\pi_{\theta_1})$

$\bar{V} \leftarrow \text{copy}(V_{\theta_2})$

$\theta_1 \leftarrow \arg\max_{\theta_1} \frac{1}{|D|} \sum_{(s,a,R)} \text{PPO\_loss}(s, a, R|\, \pi_{\theta_1}, \bar{\pi}, \bar{V})$

$\theta_2 \leftarrow \arg\min_{\theta_2} \frac{1}{|D|} \sum_{(s,a,R)\in D} (V_{\theta_2}(s) - R)^2$

output $\pi_{\theta_1}(a|\, s)$

D

Appendix of Chapter 5

4.1 Proofs of the Main Theorems

Theorem 26. *Let p be a CNF-SAT instance over n variables, constructed by an aggregation of CNF-SAT instances $p_1, \ldots, p_K$ using index lists $I_1, \ldots, I_k, \ldots, I_K$, where the first element of each I_k is k.*

Let V and $V_1, \ldots, V_K$ represent sets of value functions, and let v^* denote an optimal value function.

Assume the following:

1. *The set V factorizes into $V_1, \ldots, V_K$; that is, $v \in V$ iff there exist $v_1 \in V_1, \ldots, v_K \in V_K$ such that for all $x \in \{0,1\}^n$ and $i \in [n]$, $v(x_{\leq i}; p) = \prod_{k \in [K]} v_k(x_{I_k \cap [i]}; p)$.*
2. *For all $k \in [K]$, either $v^*(0; p_k) = 1$ or $v^*(1; p_k) = 1$.*
3. *For all $k \in [K]$ and $v \in V_k$, either $v(0; p_k) = 1$ or $v(1; p_k) = 1$.*
4. *For all $k \in [K]$, if $v^*(0; p_k) = 1$, then $\pi^{V_k}(x_0 = 0; p_k) \leq \pi^{V_k}(x_0 = 1; p_k)$; otherwise, $\pi^{V_k}(x_0 = 0; p_k) \geq \pi^{V_k}(x_0 = 1; p_k)$.*
5. *Any $v \in V$ is monotonic.*

Under these assumptions, Algorithm 4, initialized with $V^0 = V$ and $p = p$, runs for an expected time of at least 2^{K-1} steps.

Proof. By condition (2) each first variable of all instances $p_1, \ldots, p_K$ must be either 0 or 1 to be a solution. Thus, any solution to instance p must have its first K variables equal to some unique binary vector $y^* \in \{0,1\}^K$.

We derive an upper bound on the probability of generating this initial vector, then conclude from it the lower bound on the running time stated in the Theorem.

We prove by induction on t (the number of loops performed) that the probability of generating y^* is the smallest inside some set of $2^K - t$ binary sequences of length K as long as y^* has not been generated.

For $t = 0$, by (1) and (4), π^{V^0} assigns the lowest probability on y^* over the 2^K possibilities.

Assuming the inductive hypothesis holds at some loop t for some set S of sequence with size $2^K - t$ and the sequence $x^t_{\leq K}$ is not y^*. We look at the effect of the Bellman equations enforced on the set of value functions with two cases: $i < K$ and $i \geq K$ with the additional consistency equation for complete assignments.

If a Bellman equation is applied for $i < K$ then we show that no value functions are removed. Two possibilities: first, if $v(x^t_{\leq i}; p) = 0$ then, by the monotonicity assumption (5), $v([x^t_{\leq i}, 0]; p) = 0$ and $v([x^t_{\leq i}, 1]; p) = 0$; second, if $v(x^t_{\leq i}; p) = 1$ then, by (1) and (3), at least $v([x^t_{\leq i}, 0]; p)$ or $v([x^t_{\leq i}, 1]; p)$ must evaluate to 1. The equation is thus satisfied in both possibilities.

Now, in the second case, the Bellman equation is applied with $i \geq K$, including the condition ensuring consistency with Check. Let $v \in V^t$ be a value function that is removed by the Bellman equation for some $i \geq K$. We know $v(x^t_{\leq K}; p) = 1$ by the monotonicity assumption (5), otherwise the value function evaluates to 0 in all possible continuations and the Bellman equation is satisfied. Knowing this, by assumptions (1) and (3), the value function outputs 1 for $x^t_{\leq K}$ and 0 for all the other sequences of length K. Consequently, from Definition 23, at most one element of S decreases in probability, and all the others increase by a common normalizing factor. This positive factor preserves the order between the sequences' probabilities of being generated among this set, and thus the inductive hypothesis holds for $t \leftarrow t + 1$.

This bound leads to an expected number of loops of at least 2^{K-1} to generate y^* and thus a solution. □

Theorem 32. *Let p be a CNF-SAT instance with n variables, constructed as an aggregation of CNF-SAT instances $p_1, \ldots, p_K$ with index lists $I_1, \ldots, I_k, \ldots, I_K$,*

where the first element of each I_k is k.

Let V and $V_1, \ldots, V_K$ represent sets of universal value functions and $g = \text{True}$ denote the goal-state.

Assume the following:

1. *The set V factorizes into $V_1, \ldots, V_K$; that is, $v \in V$ iff there exists $v_1 \in V_1, \ldots, v_K \in V_K$ such that, for all $x \in \{0,1\}^n$ and all $i \in [n]$, $v(x_{\leq i}, g; p) = \prod_{k \in [K]} v_k(x_{I_k \cap [i]}, g; p)$.*
2. *For an optimal value function v^*, and for all $k \in [K]$, either $v^*(0; p_k) = 1$ or $v^*(1; p_k) = 1$.*
3. *For all $k \in [K]$ and $v \in V_k$, either $v(0, g; p_k) = 1$ or $v(1, g; p_k) = 1$.*
4. *For an optimal value function v^*, and for all $k \in [K]$, if $v^*(0; p_k) = 1$, then $\pi^{V_k}(s_0 = 0; p_k) \leq \pi^{V_k}(s_0 = 1; p_k)$; otherwise, $\pi^{V_k}(s_0 = 0; p_k) \geq \pi^{V_k}(s_0 = 1; p_k)$.*
5. *Any $v \in V$ is monotonic.*
6. *For any $v \in V$, $i, j \in [n]$ with $i \leq j$, and any $x_1 \in \{0,1\}^i$, $x_2 \in \{0,1\}^j$, $v(x_1, x_2; p) = v^*(x_1, x_2; p)$ where v^* is an optimal universal value function.*
7. *For an optimal universal value function v^* and any state $s \in S$, if $v^*(s, \text{False}; p) = 1$, then, for any $v \in V$, $v(s, \text{False}; p) = 1$.*

Under these assumptions, Algorithm 5 initialized with $V^0 = V$, $p = p$, and $g = g$, runs for an expected time of at least 2^{K-1} steps.

Proof. We reduce our claim to that of Theorem 26. The distribution used to sample candidate solutions follows the same constraints as in Theorem 26, with the value functions that guide the search, $v(.; p)$, being replaced by universal value functions evaluated with the goal g: $v(., g; p)$ for $v \in V$. The set V^t, when applied with g, follows the same constraints as in the previous theorem, with the exception that the Bellman equation is not only enforced with the final state g but also with the state False and states corresponding to partial solutions (innermost statement in the loops of Algorithm 5).

We study the situation where the Bellman equation is applied with the state $s^t_{n+1} = \text{False}$ as a second argument. The term $v(s^t_i, s^t_{n+1}; p)$ is either 0 or 1. If it evaluates to 0, by assumption (7), False is not reachable from s^t_i and thus $s^t_{n+1} = \text{False}$ is not possible, we have a contradiction, this case is not possible. For the case $v(s^t_i, s^t_{n+1}; p) = 1$, we remark that at least

one of $D^p(s_i^t, 0)$ or $D^p(s_i^t, 1)$ leads to the state False (since it was reached in this trajectory), thus, by assumption 7, one of $v(D^p(s_i^t, 0), s_{n+1}^t; p)$ or $v(D^p(s_i^t, 1), s_{n+1}^t)$ evaluates to 1. Consequently, all the universal value functions in V^t satisfy the Bellman equation with s_{n+1}^t/False as a second argument, and none are removed.

The Bellman equation procedure is also called with the second state corresponding to a partial solution. By assumption (6), in that case, any universal value function in V^0 (and thus V^t) already matches the output of an optimal universal value function and is consistent with the Bellman equation. Again, none of the universal value functions are removed by enforcing the equation.

Therefore, under the assumptions, the universal value functions in set V^t evaluated with $g =$ True in Algorithm 5, are updated identically to the set of value functions in Algorithm 4. The Theorem follows. □

4.2 Positive Result for a Resolution-Based SAT Solver

Our main text presented a design to produce counterexamples for algorithms based on the Bellman equation approach with classical value functions and universal value functions (HER). In this section, we prove that our design does not imply a limitation for SAT solvers based on the resolution operator. For these algorithms, the aggregation structure designed in our counterexamples does not incur an intractable computational cost. Enforcing the idea that being unable to decompose this structure is a limitation.

We note that this result applies to an algorithm constrained by a fixed variable assignment ordering, thus under the same context as the negative results for (universal) value-based RL.

We first define a resolution-based SAT solver, and then provide the positive result.

Our SAT solver is a minimalistic version of modern conflict-driven clause learning (CDCL) SAT solvers [Silva and Sakallah, 1996, Biere et al., 2009, Knuth, 2015]. At their core is a procedure that iteratively tries to generate a solution, and, simultaneously to these attempts, learns from its failures. This working principle is thus very similar to RL algorithms, the main difference is found where learning happens. Here, learning consists in deducting new logical formulas with the resolution operator.

We define this operator here, it allows us to learn a new clause by deduction.

Algorithm 23 Resolution-based SAT solver.
Iteratively attempt to generate a solution, each attempt incrementally assigns the variables until a complete solution is found or there is a conflict. In case of a conflict, the resolution operator is applied and the new clause is learned to avoid it in future attempts.

Inputs: C^0 : the set of clauses of a CNF-SAT instance with n variables

```
for t ← 1, 2, ... do
    for i ← 1, ..., n do
        if contradiction on x_i with clauses in C^{t-1} then
            C^t ←
                C^{t-1} ∪ {resolution(c_0, c_1) | c_0, c_1 ∈ C^{t-1}, c_0, c_1 in conflict}
            break
        else if x_i is forced to 1 then
            x_i ← 1
        else if x_i is forced to 0 then
            x_i ← 0
        else
            x_i ← 0
    if x is completely assigned then
        output x
    else
        reinitialize assignments
```

Definition 86. *For some CNF-SAT instance, let x_i be some variable and $l_1, \ldots, l_m, l'_1, \ldots l'_{m'}$ be some literals. Given two clauses of the form $x_i \vee l_1 \vee \ldots \vee l_m$ and $\neg x_i \vee l'_1 \vee \ldots, l'_{m'}$, the* resolution operator *produces the new clause $l_1 \vee \ldots \vee l_m \vee l'_1 \vee \ldots \vee l'_{m'}$.*

To describe the algorithm, we need two other concepts. As the algorithm incrementally builds a solution, some variables will be assigned and others not. We say that a variable x_i is *forced* to $1(/0)$ if a clause as the form $(\neg)x_i \vee l_1, \ldots \vee l_m$ where $l_1, \ldots, l_m$ are all literals that evaluate to False due to already assigned variables. A *conflict* is when the variable is forced to 0 and 1 by different clauses, in which case the current assignments cannot lead to a solution and these clauses can be resolved.

We now define Algorithm 23 iteratively performing generation loops where a candidate solution is incrementally generated.

Theorem 87. *Let p be an aggregation (Definition 24) of CNF-SAT instances $p_1, \ldots, p_K$ with index lists $I_1, \ldots, I_K$. Let $p'_1, \ldots, p'_K$ be the same CNF-SAT in-*

stances with their variables' indices permuted by their respective order in $I_1, \dots, I_K$, and let $T_1, \dots, T_K$ be the number of failed attempt loops of Algorithm 23 on those instances.

Then Algorithm 23 with input p performs at most $\sum_{k=1}^{K} T_k$ failed attempt loops.

Proof. We show that the problems are concurrently and independently solved by Algorithm 23. Each iteration of the algorithm on p simulates the iteration of the algorithm on one of the sub-problems.

The state of the algorithm evolving across loops is described by the set of clauses C^t. Let $C_k^{t_k}$ be the set of clauses of the algorithm on problem p'_k at some loop $t_k \in \mathbb{N}$. Inductively, at any loop $t \in \mathbb{N}$, C^{t-1} decomposes into the set of clauses $C_1^{t_1-1}, \dots, C_K^{t_K-1}$ for some $t_1, \dots, t_K \in \mathbb{N}$ up to a variable mapping (corresponding to the inverse of the index lists, I_k, composed with the permutation from p_k to p'_k).

This is true at initialization $t = 1$ with $t_1, \dots, t_K = 1$.

If the inductive hypothesis is true at some loop t for some $t_1, \dots, t_k$, then the generation process assigns identically the variables as the algorithm applied on the individual sub-problems at respective loops $t_1, \dots, t_k$ and has a conflict at the same variables with the same clauses up to the variable mapping. Let x_i with $i \in I_k$ be the first variable leading to a conflict in the assignment process. Up to the variable mapping, the same clauses lead to a conflict as in problem p'_k, thus the resolution operator computes the same resolved clauses that are added to $C_k^{t_k}$ at iteration t_k. The induction hypothesis is proved for the loop $t + 1$.

Moreover, at each attempt loop, one of the t_k is incremented while the others stay the same. After, $\sum_{k \in [K]} T_k$ loops the assignment process is successful on all the sub-problems, and a solution is found.

□

Theorem 87 states that for Algorithm 23 solving the aggregation of a set of problems or solving each problem in the set independently takes similar time (up to a permutation of the variables' indices). The proof leverages the fact that the resolution operator keeps the sub-problems independent.

Bibliography

J. Achiam. Spinning Up in Deep Reinforcement Learning, 2018. Accessed: 2023-11-31.

A. Ajay, Y. Du, A. Gupta, J. Tenenbaum, T. Jaakkola, and P. Agrawal. Is conditional generative modeling all you need for decision-making? *arXiv preprint arXiv:2211.15657*, 2022.

S. Alver and D. Precup. What is going on inside recurrent meta reinforcement learning agents? *arXiv preprint arXiv:2104.14644*, 2021.

B. Amos and J. Z. Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International conference on machine learning*, pages 136–145. PMLR, 2017.

M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. Pieter Abbeel, and W. Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

T. Anthony, Z. Tian, and D. Barber. Thinking fast and slow with deep learning and tree search. *Advances in neural information processing systems*, 30, 2017.

S. Arora and B. Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

M. Asai, H. Kajino, A. Fukunaga, and C. Muise. Classical planning in deep latent space. *Journal of Artificial Intelligence Research*, 74:1599–1686, 2022.

S. Bader and P. Hitzler. Dimensions of neural-symbolic integration-a structured survey. *arXiv preprint cs/0511042*, 2005.

P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

J. Baxter and P. L. Bartlett. Direct gradient-based reinforcement learning. In *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 271–274. IEEE, 2000.

S. Bechtle, A. Molchanov, Y. Chebotar, E. Grefenstette, L. Righetti, G. Sukhatme, and F. Meier. Meta learning via learned loss. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 4161–4168. IEEE, 2021.

Y. Bengio, Y. LeCun, et al. Scaling learning algorithms towards ai. *Large-scale kernel machines*, 34(5):1–41, 2007.

Y. Bengio, O. Delalleau, and C. Simard. Decision trees do not generalize to new variations. *Computational Intelligence*, 26(4):449–467, 2010.

A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information processing letters*, 24(6):377–380, 1987.

T. Bush, S. Chung, U. Anwar, A. Garriga-Alonso, and D. Krueger. Interpreting emergent planning in model-free reinforcement learning. *arXiv preprint arXiv:2504.01871*, 2025.

A. Byravan, J. T. Springenberg, A. Abdolmaleki, R. Hafner, M. Neunert, T. Lampe, N. Siegel, N. Heess, and M. Riedmiller. Imagined value gradients: Model-based policy optimization with tranferable latent dynamics models. In *Conference on Robot Learning*, pages 566–589. PMLR, 2020.

K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

D. Corneil, W. Gerstner, and J. Brea. Efficient model-based deep reinforcement learning with variational state tabulation. In *International Conference on Machine Learning*, pages 1049–1058. PMLR, 2018.

L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The lean theorem prover (system description). In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and Ł. Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

M. Deisenroth and C. E. Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472, 2011.

H. Dong, J. Mao, T. Lin, C. Wang, L. Li, and D. Zhou. Neural logic machines. In *International Conference on Learning Representations*, 2019.

Y. Du, S. Li, J. Tenenbaum, and I. Mordatch. Learning iterative reasoning through energy minimization. In *International Conference on Machine Learning*, pages 5570–5582. PMLR, 2022.

Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

R. Eldan and O. Shamir. The power of depth for feedforward neural networks. In *Conference on learning theory*, pages 907–940, 2016.

B. Eysenbach, T. Zhang, S. Levine, and R. R. Salakhutdinov. Contrastive learning as goal-conditioned reinforcement learning. *Advances in Neural Information Processing Systems*, 35:35603–35620, 2022.

M. Fairbank and E. Alonso. Value-gradient learning. In *The 2012 international joint conference on neural networks (ijcnn)*, pages 1–8. IEEE, 2012.

G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson. Treeqn and atreec: Differentiable tree-structured models for deep reinforcement learning. *arXiv preprint arXiv:1710.11417*, 2017.

V. Feinberg, A. Wan, I. Stoica, M. I. Jordan, J. E. Gonzalez, and S. Levine. Model-based value estimation for efficient model-free reinforcement learning. *arXiv preprint arXiv:1803.00101*, 2018.

C. Fernando, J. Sygnowski, S. Osindero, J. Wang, T. Schaul, D. Teplyashin, P. Sprechmann, A. Pritzel, and A. Rusu. Meta-learning by the baldwin

effect. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1313–1320, 2018.

C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, J. Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

V. François-Lavet, Y. Bengio, D. Precup, and J. Pineau. Combined reinforcement learning via abstract representations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3582–3589, 2019.

G. S. Frandsen and P. B. Miltersen. Reviewing bounds on the circuit size of the hardest functions. *Information processing letters*, 95(2):354–357, 2005.

A. d. Garcez and L. C. Lamb. Neurosymbolic ai: The 3 rd wave. *Artificial Intelligence Review*, 56(11):12387–12406, 2023.

C. Gelada, S. Kumar, J. Buckman, O. Nachum, and M. G. Bellemare. Deepmdp: Learning continuous latent space models for representation learning. In *International Conference on Machine Learning*, pages 2170–2179. PMLR, 2019.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

I. Grondman. *Online model learning algorithms for actor-critic control*. PhD thesis, Delft University of Technology, 2015.

P. D. Grünwald. *The minimum description length principle*. MIT press, 2007.

A. Guez, T. Weber, I. Antonoglou, K. Simonyan, O. Vinyals, D. Wierstra, R. Munos, and D. Silver. Learning to search with mctsnets. In *International conference on machine learning*, pages 1822–1831. PMLR, 2018.

A. Guez, M. Mirza, K. Gregor, R. Kabra, S. Racanière, T. Weber, D. Raposo, A. Santoro, L. Orseau, T. Eccles, et al. An investigation of model-free planning. In *International conference on machine learning*, pages 2464–2473. PMLR, 2019.

S. Gulwani, A. Polozov, and R. Singh. *Program Synthesis*, volume 4. NOW, August 2017.

D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL `https://www.gurobi.com`.

D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019a.

D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pages 2555–2565. PMLR, 2019b.

D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020.

D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

J. B. Hamrick, A. L. Friesen, F. Behbahani, A. Guez, F. Viola, S. Witherspoon, T. Anthony, L. Buesing, P. Veličković, and T. Weber. On the role of planning in model-based deep reinforcement learning. *arXiv preprint arXiv:2011.04021*, 2020.

J. Hawthorne. Inductive logic. `https://plato.stanford.edu/archives/sum2025/entries/logic-inductive/`, 2025. Accessed: 2025-05-29.

T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, and I. Shcherbatyi. scikit-optimize/scikit-optimize, Sept. 2020. URL `https://doi.org/10.5281/zenodo.4014775`.

N. Heess, G. Wayne, D. Silver, T. Lillicrap, T. Erez, and Y. Tassa. Learning continuous control policies by stochastic value gradients. *Advances in neural information processing systems*, 28, 2015.

F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape turing machines. *Journal of the ACM (JACM)*, 13(4):533–546, 1966.

T. Hiraoka, T. Imagawa, V. Tangkaratt, T. Osa, T. Onishi, and Y. Tsuruoka. Meta-model-based meta-policy optimization. In *Asian Conference on Machine Learning*, pages 129–144. PMLR, 2021.

T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey. Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*, 2020.

T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(9):5149–5169, 2021.

R. Houthooft, Y. Chen, P. Isola, B. Stadie, F. Wolski, O. Jonathan Ho, and P. Abbeel. Evolved policy gradients. *Advances in Neural Information Processing Systems*, 31, 2018.

J. Humplik, A. Galashov, L. Hasenclever, P. A. Ortega, Y. W. Teh, and N. Heess. Meta reinforcement learning as task inference. *arXiv preprint arXiv:1905.06424*, 2019.

M. Hutter. The fastest and shortest algorithm for all well-defined problems. *International Journal of Foundations of Computer Science*, 13(03):431–443, 2002.

P. A. Ioannou and J. Sun. *Robust adaptive control*, volume 1. PTR Prentice-Hall Upper Saddle River, NJ, 1996.

K. Iwama and H. Morizumi. An explicit lower bound of 5n- o (n) for boolean circuits. In *International Symposium on Mathematical Foundations of Computer Science*, pages 353–364. Springer, 2002.

D. H. Jacobson and D. Q. Mayne. Differential dynamic programming. *(No Title)*, 1970.

M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

M. Janner, J. Fu, M. Zhang, and S. Levine. When to trust your model: Model-based policy optimization. *Advances in neural information processing systems*, 32, 2019.

M. Janner, Y. Du, J. B. Tenenbaum, and S. Levine. Planning with diffusion for flexible behavior synthesis. *arXiv preprint arXiv:2205.09991*, 2022.

N. Jiang, A. Krishnamurthy, A. Agarwal, J. Langford, and R. E. Schapire. Contextual decision processes with low bellman rank are pac-learnable. In *International Conference on Machine Learning*, pages 1704–1713. PMLR, 2017.

M. I. Jordan and D. E. Rumelhart. Forward models: Supervised learning with a distal teacher. In *Backpropagation*, pages 189–236. Psychology Press, 2013.

S. Jukna. *Boolean function complexity: advances and frontiers*, volume 27. Springer Science & Business Media, 2012.

Ł. Kaiser and I. Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

N. Kalchbrenner, I. Danihelka, and A. Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

R. Kannan. Circuit-size lower bounds and non-reducibility to sparse sets. *Information and control*, 55(1-3):40–56, 1982.

R. M. Karp and R. J. Lipton. Some connections between nonuniform and uniform complexity classes. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 302–309. ACM, 1980.

H. Kautz. The third ai summer: Aaai robert s. engelmore memorial lecture. *Ai magazine*, 43(1):105–125, 2022.

J. Kim and T. Suzuki. Transformers provably solve parity efficiently with chain of thought. *arXiv preprint arXiv:2410.08633*, 2024.

L. Kirsch, S. van Steenkiste, and J. Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. *arXiv preprint arXiv:1910.04098*, 2019.

E. Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *International workshop on approaches and applications of inductive programming*, pages 50–73. Springer, 2009.

D. E. Knuth. *The art of computer programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 2015.

T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of information transmission*, 1(1):1–7, 1965.

G. Konidaris, L. P. Kaelbling, and T. Lozano-Perez. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61:215–289, 2018.

J. R. Koza and R. Poli. Genetic programming. In *Search Methodologies*, pages 127–164. Springer, 2005.

K. Kurach, M. Andrychowicz, and I. Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

T. Kurutach, A. Tamar, G. Yang, S. J. Russell, and P. Abbeel. Learning plannable representations with causal infogan. *Advances in Neural Information Processing Systems*, 31, 2018.

H. Kwakernaak and R. Sivan. *Linear optimal control systems*, volume 1. Wiley-interscience New York, 1972.

G. Lample, T. Lacroix, M.-A. Lachaux, A. Rodriguez, A. Hayat, T. Lavril, G. Ebner, and X. Martinet. Hypertree proof search for neural theorem proving. *Advances in Neural Information Processing Systems*, 35:26337–26349, 2022.

M. Laskin, A. Srinivas, and P. Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. In *International conference on machine learning*, pages 5639–5650. PMLR, 2020.

I. Lenz, R. A. Knepper, and A. Saxena. Deepmpc: Learning deep latent features for model predictive control. In *Robotics: Science and Systems*, volume 10, page 25. Rome, Italy, 2015.

S. Levine and P. Abbeel. Learning neural network policies with guided policy search under unknown dynamics. *Advances in neural information processing systems*, 27, 2014.

S. Levine, A. Kumar, G. Tucker, and J. Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.

A. Levy, G. Konidaris, R. Platt, and K. Saenko. Learning multi-level hierarchies with hindsight. *arXiv preprint arXiv:1712.00948*, 2017.

M. Li, P. Vitányi, et al. *An introduction to Kolmogorov complexity and its applications*, volume 4. Springer, 2019.

Y. Li. Deep reinforcement learning, 2018.

C. Liang, J. Berant, Q. Le, K. D. Forbus, and N. Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *arXiv preprint arXiv:1611.00020*, 2016.

S. Liang and R. Srikant. Why deep neural networks for function approximation? *arXiv preprint arXiv:1610.04161*, 2016.

H. Lightman, V. Kosaraju, Y. Burda, H. Edwards, B. Baker, T. Lee, J. Leike, J. Schulman, I. Sutskever, and K. Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2023.

J. v. Lint. *Introduction to coding theory*. Springer, 1999.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

K. Lowrey, A. Rajeswaran, S. Kakade, E. Todorov, and I. Mordatch. Plan online, learn offline: Efficient learning and exploration via model-based control. *arXiv preprint arXiv:1811.01848*, 2018.

R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018.

J. Mao, C. Gan, P. Kohli, J. B. Tenenbaum, and J. Wu. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In *International Conference on Learning Representations*, 2019.

R. Mendonca, X. Geng, C. Finn, and S. Levine. Meta-reinforcement learning robust to distributional shift via model identification and experience relabeling. *arXiv preprint arXiv:2006.07178*, 2020.

W. T. Miller, R. S. Sutton, and P. J. Werbos. *Neural networks for control*. MIT press, 1995.

N. Mishra, M. Rohaninejad, X. Chen, and P. Abbeel. A simple neural attentive meta-learner. *arXiv preprint arXiv:1707.03141*, 2017.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

T. M. Moerland, J. Broekens, A. Plaat, C. M. Jonker, et al. Model-based reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 16(1):1–118, 2023.

I. Mordatch and J. Hamrick. Tutorial on model-based methods in reinforcement learning. `https://sites.google.com/view/mbrl-tutorial`, 2020. Accessed: 2023-07-31.

M. Mundhenk, J. Goldsmith, C. Lusena, and E. Allender. Complexity of finite-horizon markov decision process problems. *Journal of the ACM (JACM)*, 47(4):681–720, 2000.

K. Murphy. Reinforcement learning: An overview. *arXiv preprint arXiv:2412.05265*, 2024.

O. Nachum, S. S. Gu, H. Lee, and S. Levine. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

A. Nagabandi, I. Clavera, S. Liu, R. S. Fearing, P. Abbeel, S. Levine, and C. Finn. Learning to adapt in dynamic, real-world environments through meta-reinforcement learning. *arXiv preprint arXiv:1803.11347*, 2018a.

A. Nagabandi, C. Finn, and S. Levine. Deep online learning via meta-learning: Continual adaptation for model-based rl. *arXiv preprint arXiv:1812.07671*, 2018b.

D. H. Nguyen and B. Widrow. Neural networks for self-learning control systems. *IEEE Control systems magazine*, 10(3):18–23, 1990.

M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, et al. Show your work: Scratchpads for intermediate computation with language models. 2021.

J. Oh, S. Singh, and H. Lee. Value prediction network. *Advances in neural information processing systems*, 30, 2017.

J. Oh, M. Hessel, W. M. Czarnecki, Z. Xu, H. P. van Hasselt, S. Singh, and D. Silver. Discovering reinforcement learning algorithms. *Advances in Neural Information Processing Systems*, 33:1060–1070, 2020.

E. Parisotto, F. Song, J. Rae, R. Pascanu, C. Gulcehre, S. Jayakumar, M. Jaderberg, R. L. Kaufman, A. Clark, S. Noury, et al. Stabilizing transformers for reinforcement learning. In *International Conference on Machine Learning*, pages 7487–7498. PMLR, 2020.

A. Pavan, R. Santhanam, and N. Vinodchandran. Some results on average-case hardness within the polynomial hierarchy. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 188–199. Springer, 2006.

C. Perez, F. P. Such, and T. Karaletsos. Generalized hidden parameter mdps: Transferable model-based rl in a handful of trials. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5403–5411, 2020.

B. Pinon, R. Jungers, and J.-C. Delvenne. A model-based approach to meta-reinforcement learning: Transformers and tree search. *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2023a.

B. Pinon, R. Jungers, and J.-C. Delvenne. Pac-learning gains of turing machines over circuits and neural networks. *Physica D: Nonlinear Phenomena*, 444:133585, 2023b.

B. Pinon, J.-C. Delvenne, and R. Jungers. Counterexamples to rl approaches blending search and learning for problem-solving. *BeNeLearn*, 2024.

B. Pinon, R. Jungers, and J.-C. Delvenne. A limitation on black-box dynamics approaches to reinforcement learning. *Transactions on Machine Learning Research*, 2025.

N. Pippenger and M. J. Fischer. Relations among complexity measures. *Journal of the ACM (JACM)*, 26(2):361–381, 1979.

S. Polu and I. Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

S. Reed and N. De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

M. Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005: 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005. Proceedings 16*, pages 317–328. Springer, 2005.

T. Rocktäschel and S. Riedel. End-to-end differentiable proving. *Advances in neural information processing systems*, 30, 2017.

B. Rossman, R. A. Servedio, and L.-Y. Tan. An average-case depth hierarchy theorem for boolean circuits. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 1030–1048. IEEE, 2015.

B. Russell. *History of western philosophy*. George Allen & Unwin Ltd, 1946.

S. J. Russell and P. Norvig. *Artificial intelligence a modern approach*. Prentice Hall, 2010.

S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Pearson, 2016.

S. Sæmundsson, K. Hofmann, and M. P. Deisenroth. Meta reinforcement learning with latent variable gaussian processes. *arXiv preprint arXiv:1803.07551*, 2018.

S. H. Sæther, J. A. Telle, and M. Vatshelle. Solving# sat and maxsat by dynamic programming. *Journal of Artificial Intelligence Research*, 54:59–82, 2015.

A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pages 1842–1850. PMLR, 2016.

T. Schaul, D. Horgan, K. Gregor, and D. Silver. Universal value function approximators. In *International conference on machine learning*, pages 1312–1320. PMLR, 2015.

T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

J. Schmidhuber. *Making the world differentiable: on using self supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments*, volume 126. Inst. für Informatik, 1990.

J. Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In *Artificial general intelligence*, pages 199–226. Springer, 2007.

J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

C.-P. Schnorr. The network complexity and the turing machine complexity of finite functions. *Acta Informatica*, 7(1):95–107, 1976.

J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839): 604–609, 2020.

J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

M. Schwarzer, A. Anand, R. Goel, R. D. Hjelm, A. Courville, and P. Bachman. Data-efficient reinforcement learning with self-predictive representations. *arXiv preprint arXiv:2007.05929*, 2020.

M. Schwarzer, J. S. O. Ceron, A. Courville, M. G. Bellemare, R. Agarwal, and P. S. Castro. Bigger, better, faster: Human-level atari with human-level efficiency. In *International Conference on Machine Learning*, pages 30365–30380. PMLR, 2023.

D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

L. Serafini and A. d. Garcez. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. *arXiv preprint arXiv:1606.04422*, 2016.

S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

J. Shen, Y. Yin, L. Li, L. Shang, X. Jiang, M. Zhang, and Q. Liu. Generate & rank: A multi-task framework for math word problems. *arXiv preprint arXiv:2109.03034*, 2021.

J. M. Silva and K. A. Sakallah. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227. IEEE, 1996.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

R. J. Solomonoff. A preliminary report on a general theory of inductive inference. Technical report, Zator Company, Cambridge, MA, November 1960.

R. J. Solomonoff. An inductive inference code employing definitions. Technical report, Rockford Research, Cambridge, MA, April 1962.

R. J. Solomonoff. A formal theory of inductive inference. part i. *Information and control*, 7(1):1–22, 1964a.

R. J. Solomonoff. A formal theory of inductive inference. part ii. *Information and control*, 7(2):224–254, 1964b.

H. F. Song, A. Abdolmaleki, J. T. Springenberg, A. Clark, H. Soyer, J. W. Rae, S. Noury, A. Ahuja, S. Liu, D. Tirumala, et al. V-mpo: On-policy maximum a posteriori policy optimization for discrete and continuous control. *arXiv preprint arXiv:1909.12238*, 2019.

A. Srinivas, A. Jabri, P. Abbeel, S. Levine, and C. Finn. Universal planning networks: Learning generalizable representations for visuomotor control. In *International conference on machine learning*, pages 4732–4741. PMLR, 2018.

W. Sun, N. Jiang, A. Krishnamurthy, A. Agarwal, and J. Langford. Model-based rl in contextual decision processes: Pac bounds and exponential improvements over model-free approaches. In *Conference on learning theory*, pages 2898–2933. PMLR, 2019.

R. Sutton. The bitter lesson. *Incomplete Ideas (blog)*, 13(1):38, 2019.

R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine learning proceedings 1990*, pages 216–224. Elsevier, 1990.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

R. S. Sutton, A. G. Barto, et al. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

R. S. Sutton, J. Modayil, M. Delp, T. Degris, P. M. Pilarski, A. White, and D. Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 761–768, 2011.

A. Tamar, Y. Wu, G. Thomas, S. Levine, and P. Abbeel. Value iteration networks. *Advances in neural information processing systems*, 29, 2016.

M. Telgarsky. Representation benefits of deep feedforward networks. *arXiv preprint arXiv:1509.08101*, 2015.

S. Thrun and L. Pratt. Learning to learn: Introduction and overview. In *Learning to learn*, pages 3–17. Springer, 1998.

E. Todorov and W. Li. A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 300–306. IEEE, 2005.

S. Tu and B. Recht. The gap between model-based and model-free methods on the linear quadratic regulator: An asymptotic viewpoint. In *Conference on Learning Theory*, pages 3036–3083. PMLR, 2019.

J. Uesato, N. Kushman, R. Kumar, F. Song, N. Siegel, L. Wang, A. Creswell, G. Irving, and I. Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. In *Measures of complexity*, pages 11–30. Springer, 2015.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

G. Wang, J. Li, Y. Sun, X. Chen, C. Liu, Y. Wu, M. Lu, S. Song, and Y. A. Yadkori. Hierarchical reasoning model. *arXiv preprint arXiv:2506.21734*, 2025.

J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

J. X. Wang, M. King, N. P. M. Porcel, Z. Kurth-Nelson, T. Zhu, C. Deck, P. Choy, M. Cassin, M. Reynolds, H. F. Song, et al. Alchemy: A benchmark and analysis toolkit for meta-reinforcement learning agents. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

P. Wang, L. Li, Z. Shao, R. Xu, D. Dai, Y. Li, D. Chen, Y. Wu, and Z. Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. *arXiv preprint arXiv:2312.08935*, 2023.

P.-W. Wang, P. Donti, B. Wilder, and Z. Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning*, pages 6545–6554. PMLR, 2019.

X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

M. Watter, J. Springenberg, J. Boedecker, and M. Riedmiller. Embed to control: A locally linear latent dynamics model for control from raw images. *Advances in neural information processing systems*, 28, 2015.

G. Wayne, C.-C. Hung, D. Amos, M. Mirza, A. Ahuja, A. Grabska-Barwinska, J. Rae, P. Mirowski, J. Z. Leibo, A. Santoro, et al. Unsupervised predictive memory in a goal-directed agent. *arXiv preprint arXiv:1803.10760*, 2018.

J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Y. Weng, M. Zhu, F. Xia, B. Li, S. He, S. Liu, B. Sun, K. Liu, and J. Zhao. Large language models are better reasoners with self-verification. *arXiv preprint arXiv:2212.09561*, 2022.

P. Werbos. Approximate dynamic programming for real-time control and neural modeling. *Handbook of intelligent control*, 1992.

R. Williams. A class of gradient-estimation algorithms for reinforcement learning in neural networks. In *Proceedings of the International Conference on Neural Networks*, pages II–601, 1987.

K. Xu, J. Li, S. S. Du, K. ichi Kawarabayashi, and S. Jegelka. What can neural networks reason about? In *Proceedings of the International Conference on Learning Representations*, pages –, 2020.

W. Ye, S. Liu, T. Kurutach, P. Abbeel, and Y. Gao. Mastering atari games with limited data. *Advances in neural information processing systems*, 34: 25476–25488, 2021.

L. Zintgraf, K. Shiarlis, M. Igl, S. Schulze, Y. Gal, K. Hofmann, and S. Whiteson. Varibad: A very good method for bayes-adaptive deep rl via meta-learning. *arXiv preprint arXiv:1910.08348*, 2019.