Seminar in AI: Mastering Atari, Go, chess and shogi by planning with a learned model

Hunor Karamán — k11919366

June 2021

1 Introduction

Lookahead based planning algorithms achieved great success in game playing and have already had positive impact in real-world applications such as logistics and chemical synthesis, but these methods rely on knowledge of the true environment dynamics. However, most often in real-world situations, the dynamics of the environment are complex or unknown. In their paper "Mastering Atari, Go, chess and shogi by planning with a learned model" [1], Schrittwieser et al. of DeepMind propose the MuZero architecture, the fourth iteration of their tree-search based reinforcement learning models: AlphaGo (2016) [2], AlphaGo Zero (2017) [3], AlphaZero (2018) [4], MuZero (2020).

The MuZero algorithm matches the performance of their existing models in games requiring exact planning (like AlphaGo and AlphaZero in the game of go; AlphaZero in the games chess and shogi), however, unlike said models, it also achieves a new state-of-the-art in visually complex domains (like the 57 Atari 2600 games) and it has no access to the rules of the game. The model instead tries to learn the rules, by estimating the dynamics of the environment with respect to the observed rewards returned by the simulator as described in Sections 3 and 4.

MuZero incorporates a learned model of the environment into the successful search algorithm of AlphaZero, while it also generalizes this to a broader set of environments, "including single-agent domains and non-zero rewards at intermediate steps."

2 Reinforcement learning: model-based and model-free methods

The goal in solving reinforcement learning problems is to find an agent that can maximize the cumulative reward returned by the environment simulator after choosing from a finite set of actions, in a finite set of states. This setting can formally be modelled as a Markov Decision Process (MDP) with the respective environment (and agent) states s, actions a, the probabilities of transitioning from state s_k to s_{k+1} by taking action a_k, and the corresponding immediate rewards r_k after taking action a_k in s_k to transition to s_{k+1}. A policy π describes the action-selection strategy of the agent. It maps action-state pairs (a, s) to the probability of taking action a while being in state s. A value function $v_\pi(s) = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s\right)$ is the expected return starting with state s and choosing successor states according to π. The policy maximizing the expected reward is called

optimal, π^*, and the expected value function using the optimal policy is the optimal value function $v^*(s) = \max_\pi v_\pi(s)$.

To solve the aforementioned cumulative reward maximization task, reinforcement learning methods can be split into model-based and model-free approaches (as seen in Figure 1). Model-based methods rely on a model of the environment in order to estimate how the real environment will react to their planned actions. This model is often provided by the system designer (e.g. coding the rules of the game into the model, as in the case of AlphaZero) or is first learned by the reinforcement learning algorithm itself. After a model has been constructed, applying an MDP planning algorithm (such as Monte-Carlo Tree-Search) is straight forward for predicting the optimal value function and policy. However, the split between first learning a model and later using this for planning is problematic as the agent cannot optimize its representation of the environment for the purpose of effective learning. Model-based methods showed great success in scenarios requiring precise lookeahead planning, but their performance in previous works remains far from model-free approaches in visually complex domains. These, model-free, methods directly try to estimate the optimal policy and value functions from their interactions with the environment, however, they suffer in problems requiring exact planning.

We will see that MuZero, in some sense, combines both model-free and model-based methods, building on the respective strengths of these to end up with a more general method thriving in most situations. Because it is not provided with the rules of the game during planning (although, let's not forget that a simulator is still required for training the model), it seems to be a method much more applicable to real-world scenarios and is potentially moving towards a solution for cases where a perfect simulator does not exist.

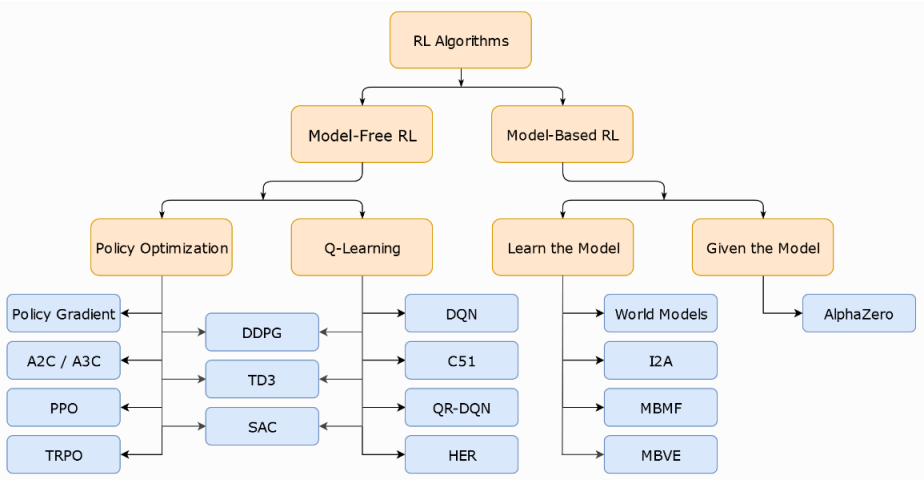

Figure 1: A non-exhaustive, but useful taxonomy of algorithms in modern RL. Source: [5]

3 Value Prediction Network

In previous model-based works, it was common to try to model the environment directly at the level of observations, e.g. the pixel values of the Atari screen, however this is computationally intractable

for large-scale problems. Furthermore, by using this representation, the algorithm must learn to model environment dynamics that are unnecessary for the planning and waste their capacity on irrelevant details.

In recent years, a new family of methods (value equivalent models) emerged focusing on predicting the value function by constructing "an abstract MDP model such that planning in the abstract MDP is equivalent to planning in the real environment. This is achieved by ensuring value equivalence, that is, that, starting from the same real state, the cumulative reward of a trajectory through the abstract MDP matches the cumulative reward of a trajectory in the real environment."

The most relevant of previous works is probably "Value Prediction Network" [6], a value equivalent model, in which the unrolled MDP is trained such that the cumulative sum of rewards, on a sequence of actions generated by a simple lookahead search matches the real environment. Future observations need not be predicted and the abstract MDP needs not resemble the real environment, as the only objective of the model is to match the cumulative sum of rewards. The sole purpose of the abstract MDP, learned by optimizing on the deviations of the predicted and real sum of rewards, therefore is to assist the equivalence in the reward prediction.

The value prediction network is parameterized by four independent modules that closely resemble the architecture of the MuZero model:

- The **encoding** module $f^{enc} : x \rightarrow s$ maps the observation x to the abstract state s using a neural network. Thus, "s is an abstract-state representation which will be learned by the network (and not an environment state or even an approximation to one)."[6]
- The **value** module $f^{val} : s \rightarrow V_\theta(s)$ estimates the value of the abstract state s.
- The **outcome** module $f^{out} : s, o \rightarrow r, \gamma$ predicts the reward r and the discount γ for executing option o in state s.
- The **transition** module $f^{trans} : s, o \rightarrow s'$ transforms the abstract-state s to the next abstract state s' by executing option o.

Compared to MuZero, there is no policy estimation in value prediction networks, and instead of Monte-Carlo Tree-Search it implements a simple planning algorithm utilizing only value predictions.

4 MuZero Architecture

The following two sections will focus on introducing the structural aspects of the MuZero model, as well as describing the formal representation of the algorithm. In Section 6, however, the implementation details of the model and its exact training procedure will be discussed, based on the pseudocode[1] which was published with the pre-print version of the paper and that was of great help during the process of writing this report. It's important to note that the paper does not contain detailed formal proofs and it only presents the specific formulas of its internal Monte-Carlo Tree-Search (MCTS) procedure (as described in Section 5). A slightly more formal formulation of a method with similar behavior can be found in the "Value Prediction Network" paper mentioned above.

On a first glance, the most prominent development from AlphaZero to MuZero is that while the former incorporates a single neural network, MuZero is a combination of three models (not unlike

[1] https://arxiv.org/src/1911.08265v2/anc/pseudocode.py

the multiple models of the value prediction network). At each time step t, for each round $k = 0, ..., K$ of the MCTS procedure, these three modules combined predict the three future quantities that are used during planning: the policy p_t^k, the value function v_t^k and for $k > 0$ the immediate reward r_t^k.

Thus, at each time step t, the model is a combination of the following three functions (subscripts t are suppressed):

- **Representation function**: $s^0 = h_\theta(o_1, ..., o_t)$ Initializes the root hidden state s^0 by encoding past observations $o_1, ..., o_t$.
- **Dynamics function**: $r^k, s^k = g_\theta(s^{k-1}, a^k)$ From the previous hidden state s^{k-1} and the current action a^k, it predicts the next hidden state s^k and the immediate reward r^k.
- **Prediction function**: $p^k, v^k = f_\theta(s^k)$ Computes the policy p^k and value v^k functions from the hidden state s^k (resembling the joint policy and value network of AlphaZero).

To emphasise again, the hidden states s^k have "no special semantics beyond their support for future predictions" and they have no further conformity to the environment state attached to them, other than value equivalence within the trajectory. As the authors argue: "it is simply the hidden state of the overall model and its sole purpose is to accurately predict relevant, future quantities: policies, values and rewards." Furthermore, the dynamics function $g_\theta(s^{k-1}, a^k)$ is deterministic; the authors leave the extension of the method to stochastic transitions to future work.

The three functions are neural networks re-using most of the architecture from AlphaZero. The prediction function uses the exact same architecture as AlphaZero: "one or two convolutional layers that preserve the resolution but reduce the number of planes, followed by a fully connected layer to the size of the output". Both the representation and dynamics function use the same architecture, but with 16 instead of 20 residual blocks (using 3×3 kernels and 256 hidden planes for each convolution). As input, the representation function always gets a history of previous observation states (e.g. board states, RGB frames) and the actions that led to those, encoded as bias planes. Furthermore, the "input to the dynamics function is the hidden state produced by the representation function or previous application of the dynamics function, concatenated with a representation of the action for the transition."

5 Planning

Although basing the action selection only on the outputs of the three models (with a Q-learning-like approach) already matches the performance of the state-of-the-art model-free method R2D2 (as seen in Section 7), to further improve this, the authors incorporate the Monte-Carlo Tree-Search lookahead search used in AlphaZero on top of the outputs of the three networks.

Algorithms that use the Monte-Carlo method, solve, by using randomness, deterministic problems that would be difficult or impossible to solve with other methods.[7] Monte-Carlo Tree-Search (MCTS) uses this principle to solve decision problems, by running simulations in a search tree based on random selections of successor nodes. In game playing, this is usually done by performing multiple rounds of playing out the game from the current game-state until the end, by selecting random actions. After such a process, we get an estimate at the root node of which next step seems the most promising.

Each Monte-Carlo round is done in four phases:

1. **Selection**: Selecting a leaf node based on stored statistics by a pre-defined method that often contains some bias towards the most promising nodes, but enables random exploration as well.

2. **Expansion**: Expanding the selected node with its valid successor states and selecting one of them.

3. **Simulation**: Playing out the game until the very end, following some simple (e.g. random uniform selection) or an advanced policy (e.g. the policy, value function and reward estimates of the MuZero models).

4. **Backpropagation**: Updating the information of all nodes on the path from the root of the tree until the selected node.

If the selection phase uses an appropriate method (e.g. Upper Confidence Bound applied to Trees (UCT)), MCTS converges asymptotically to the optimal policy in single agent domains and to the minimax value function in zero sum games.[8]

The planning part of the MuZero algorithm resembles a Monte-Carlo Tree-Search procedure with upper confidence bounds, more precisely with maximizing over a probabilistic upper confidence tree (PUCT) bound seen in (1). The authors differentiate between three stages in the planning that are repeated for a number of rounds K. These do not exactly align with the four phases of standard MCTS, but they do share the same logic. As some of the terms in the paper get confusing, I will try to comment on these first. The authors use the word "simulation" both for the process of selecting a leaf node by walking down the tree and for the combined sequence of all of the phases (used as "round" in the description of MCTS). In order to avoid confusion, I will continue using the word "round" for the loop of the three stages, and leave the word "simulation" for the process of walking down the tree until stopping at a leaf node (which truly corresponds to simulating the game being played).

The MCTS process in MuZero is based on a set of statistics stored along the edges (s, a). An edge (s, a) exists if there is an available action a from internal hidden state s. Visit counts $N(s, a)$, policy $P(s, a)$, mean value $Q(s, a)$, reward $R(s, a)$ and state transition $S(s, a)$ are stored at each edge.

Selection. Every round starts with the selection stage, which consists of a simulation of the game on the existing nodes of the tree, played out by selecting an action according to (1) at each state. The simulation starts from the root hidden state s^0 and finishes when the play-out reaches a leaf node s^l. At each hypothetical time step $k = 1, ..., l$, an action a^k is selected by maximizing over the PUCT bound

$$a^k = \arg\max_a \left\{ Q(s,a) + P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \left(c_1 + \log \left(\frac{\sum_b N(s,b) + c_2 + 1}{c_2} \right) \right) \right\} \tag{1}$$

using the stored statistics, where a and b are possible actions. The constants c_1, c_2 "are used to control the influence of the policy $P(s, a)$ relative to the value $Q(s, a)$ as nodes are visited more often". In the experiments described in Section 7, these correspond to $c_1 = 1.25$ and $c_2 = 119,652$.

Expansion. After selecting an action at the final step l of the simulation, the tree is extended by new node s^l generated by the dynamics function $r^l, s^l = f_\theta(s^{l-1}, a^l)$. The reward r^l, along with the policy and value function computed by the prediction function $p^l, v^l = f_\theta(s^l)$ are stored.

Each edge from the newly expanded node is initialized with visit count $N(s^l, a) = 0$, mean value $Q(s^l, a) = 0$ and policy $P(s^l, a) = p^l$.

Backup. At the end of the simulation and expansion, the edge statistics are updated along the trajectory to count with the final outcome of the hypothetical game. Compared to AlphaZero, the backup is generalized to cases where "the environment can emit intermediate rewards, have a discount γ different from 1 and the value estimates are unbounded." In board games, where there is no intermediate reward, the discount is assumed to be constant 1. First, for $k = l, ..., 0$ an $l-k$ step estimate of the cumulative discounted reward is computed, bootstrapping from the value function v^l

$$G^k = \sum_{\tau=0}^{l-1-k} \gamma^\tau r_{k+1+\tau} + \gamma^{l-k} v^l. \tag{2}$$

Then for $k = l, ..., 1$, the statistics for each edge (s^{k-1}, a) in the trajectory are updated according to

$$Q(s^{k-1}, a^k) = \frac{N(s^{k-1}, a^k) Q(s^{k-1}, a^k) + G^k}{N(s^{k-1}, a^k) + 1}. \tag{3}$$

Visit counts $N(s^{k-1}, a^k)$ are increased as well. It is worth noting, that unlike in two-player zero sum games, the value functions can be unbounded. To avoid the need for game specific normalization and prior knowledge about the rules of the game, MuZero normalizes the Q-value estimates to $\bar{Q} \in [0, 1]$ "by using the minimum–maximum values observed in the search tree up to that point."

The lookahead search described above produces an improved policy π_t and value function v_t at the root of the search tree, by combining the policy, value function and reward estimates of the three models. The next action is chosen according to this improved policy $a_{t+1} \approx \pi_t$.

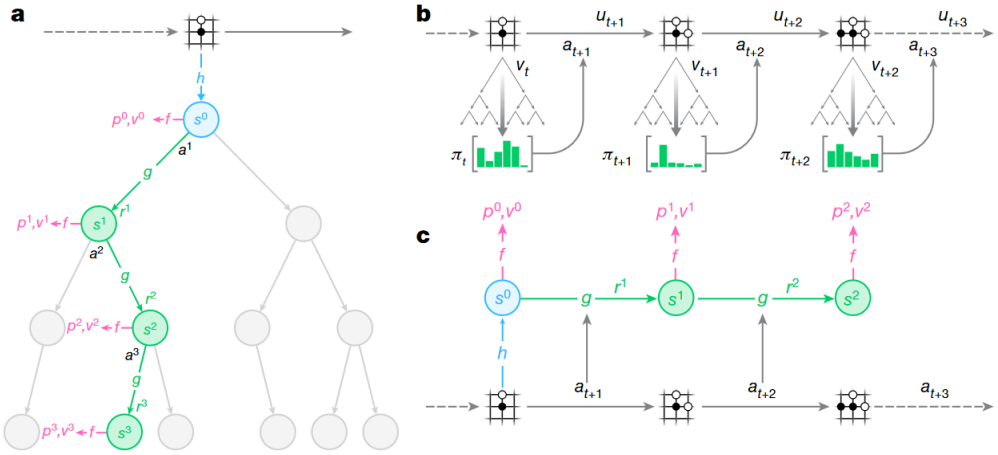

Figure 2: Visualization of the game-play using planning, including a) the simulation; b) the action selection based on the improved policy; c) the relation between the predicted and target values during training. Source: [1]

6 Training

As already announced, in this section, a brief description of the training procedure follows. All parameters of the three models are trained together to match the policy, value function and reward prediction for each hypothetical step k of the simulation, to three true targets observed after k actual steps in the game. Here we can observe, that the planning procedure and the model predictions improve each other bidirectionally: better predictions improve the search and better search improves the game-play, from which the true targets are derived.

The overall loss

$$l_t(\theta) = \sum_{k=0}^{K} l^p(\pi_{t+k}, p_t^k) + \sum_{k=0}^{K} l^v(z_{t+k}, v_t^k) + \sum_{k=0}^{K} l^r(u_{t+k}, r_t^k) + c||\theta||^2 \tag{4}$$

is formed by three objectives and additional L2 normalization. The first objective is to minimize the error between the actions predicted by the policy p_t^k and by the improved policy of the search π_{t+k}. The second objective is to minimize the error between the value function v_t^k and the n-step bootstrapped value target $z_{t+k} = u_{t+1} + \gamma u_{t+2} + ... + \gamma^{n-1} u_{t+n} + \gamma^n v_{t+n}$. The third objective is to minimize the error between the predicted immediate reward r_t^k and the observed immediate reward u_{t+k}. For chess, Go and shogi, the same squared error loss as AlphaZero is used for rewards l^r and values l^v. The authors found cross-entropy loss to be more stable when encountering rewards and values of variable scale in Atari. Cross-entropy was used for the policy loss l^p in both cases.

Here, again, we can observe how the hidden states are not directly present in the loss function and that they are trained indirectly through the three values. As mentioned before, this enables MuZero to optimize its hidden state representation to maximally improve the planning.

The training process is split into two independent parts that are repeated for a number of training steps: data generation through self-play and network training. These two parts only communicate by transferring the latest network checkpoint from the training to the self-play process, and the finished games, with their stored statistics, from the self-play to the network training procedure typical of neural networks.

Self-play. For a fixed amount of episodes, the game is played out by MCTS planning from a starting game-state until the end of the game. In the experiments, 800 rounds of MCTS planning were done for board-games at each game-step, while 50 rounds for Atari. Each board state along the sequence with their respective search policy, search value function and true environment reward is stored, together with the saved edge statistics along the search including the outputs of the three networks.

Network training. For training the three neural networks, first a batch of games is sampled (using batch size 2048 for board games and 1024 for Atari), then for each one, a random position along the true game-steps is selected. An item in the batch consists of the board state o_k of this step, and the policy, value function and reward of the next K states. For every item in the batch, first the representation and prediction functions compute the root hidden state s^0 and from this the policy p^0 and value function v^0. The loss is initialized on the latter two values. Then, a recurrent process of K steps follows, using the dynamics and prediction functions, starting from the previously computed hidden state, where the dynamics function transforms the previous hidden state s^{k-1} to a new one s^k. The losses along the recurrent process are summed up, together with the initial loss from the root hidden state. Lastly, the loss is normalized to account for a consistency across different unroll steps K.

7 Results

In the experiments described in the paper, MuZero was applied to go, shogi and chess and to all 57 games of the Atari 2600 environment. We can see that the performance of MuZero matches the already super-human performance of AlphaZero in both go, shogi and chess, without having direct access to the rules of the game. It also outperforms the previous state-of-the-art in Atari games, R2D2, in 42 our of 57 games, also outperforming by far the previous model-based method SimPLe.

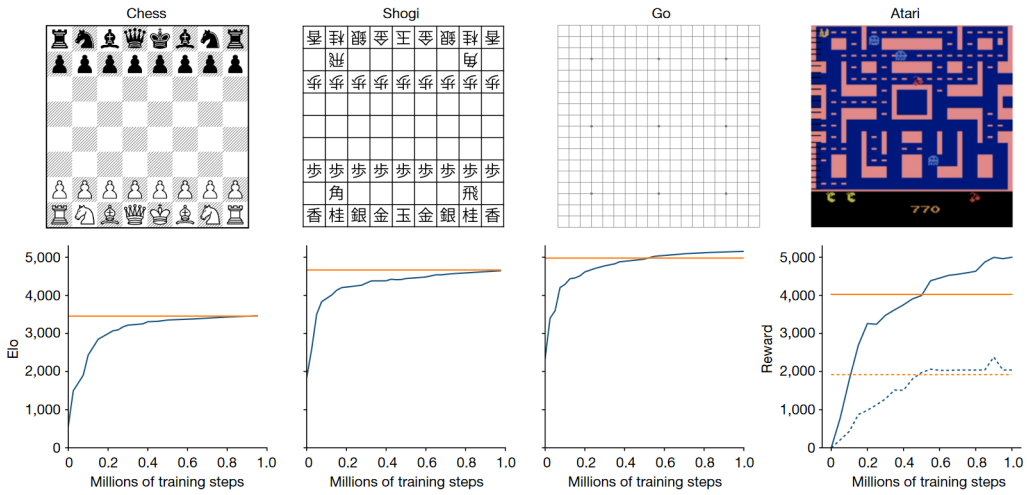

Figure 3: Performance of MuZero, indicated with blue lines, compared to baseline models AlphaZero (chess, shogi and go) and R2D2 (Atari) visualized by the orange lines. Source: [1]

To further understand the strength of the method, the authors conducted a set of scalability experiments to see how a fully trained MuZero performs in different planning and resource availability situations.

First, they tested the "scalability of planning, in the canonical planning problem of Go". They compared how different thinking times (different number of MCTS rounds per step) change the performance of MuZero compared to AlphaZero's perfect model performing with the same thinking time (Figure 4/a). Interestingly, they found that MuZero matched the performance of the perfect model, even when doing much larger searches compared to those at training, which suggests that the environment model learned by MuZero got very close to the perfect model of AlphaZero. The same experiment in the Atari games showed much less marked improvements with increased search time (Figure 4/b).

They also validated the importance of the MCTS-based training, by replacing the MCTS planning by a Q-learning approach (similar to the one in R2D2). The Q-learning approach reaches the performance of the state-of-the-art Q-learning method, R2D2, but it improves slower and reaches a far worse result than the original version (Figure 4/c).

It was also observed, that "networks trained with more simulations per move improve faster [...]. Surprisingly, MuZero can learn effectively even when training with less simulations per move than are enough to cover all eight possible actions in Ms. Pac-Man." (Figure 4/d)

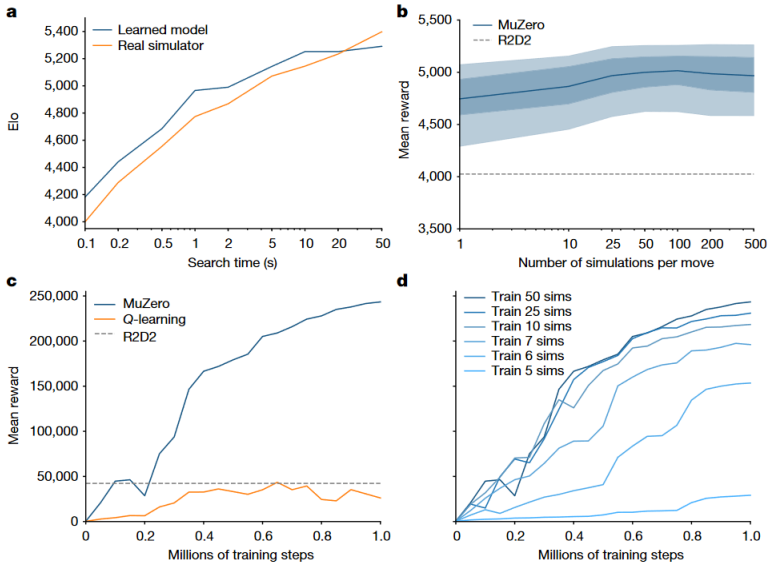

Figure 4: Visualization of the scalability experiments. Source: [1]

8 Conclusion

In the fourth generation of their planning based game-playing methods, Schrittweiser et al. introduce a method that incorporates both the high-performance planning capabilities of their previous methods and the benefits of model-free reinforcement learning techniques. Their results show, that the model performs equally well both in domains favoured by exact planning algorithms ("logically complex board games such as chess and Go") and in domains where model-free methods showed much better performance until now ("visually complex Atari games"), without relying on any built-in knowledge of the environment dynamics. They hope that this method potentially paves "the way towards the application of powerful learning and planning methods to a host of real-world domains for which there exists no perfect simulator."

References

[1] Schrittwieser, J., Antonoglou, I., Hubert, T. et al. *Mastering Atari, Go, chess and shogi by planning with a learned model*. Nature 588, 604–609 (2020). https://doi.org/10.1038/s41586-020-03051-4

[2] Silver, D., Huang, A., Maddison, C. et al. *Mastering the game of Go with deep neural networks and tree search*. Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961

[3] Silver, D., Schrittwieser, J., Simonyan, K. et al. *Mastering the game of Go without human knowledge*. Nature 550, 354–359 (2017). https://doi.org/10.1038/nature24270

[4] Silver D, Hubert T, Schrittwieser J, Antonoglou I, Lai M, Guez A, Lanctot M, Sifre L, Kumaran D, Graepel T, Lillicrap T, Simonyan K, Hassabis D. *A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play*. Science. 2018 Dec 7;362(6419):1140-1144. https://doi.org/10.1126/science.aar6404

[5] Achiam, J. (OpenAI) *Spinning Up in Deep RL*. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

[6] Oh, J., Singh, S. & Lee, H. *Value Prediction Network*. Adv. Neural Inf. Process. Syst. 30, 6118–6128 (2017). https://arxiv.org/abs/1707.03497

[7] Wikipedia *Monte Carlo tree search*. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

[8] Kocsis, L. & Szepesvári, C. *Bandit based Monte-Carlo planning*. European Conference on Machine Learning 282–293 (Springer, 2006).