DDA4230 Reinforcement learning Monte-Carlo tree search

Lecture 13

Lecturer: Guiliang Liu *Scribe: Baoxiang Wang*

1 Goal of this lecture

In this lecture we discuss imitation learning, where the reward signal is not available and instead a set of experiences generated by some experts is given.
Suggested reading: Chapter 8 and 16 of *Reinforcement learning: An introduction*; References in the lecture notes.

2 Introduction

In this lecture we will learn about model-based RL and simulation-based tree search methods. So far we have seen methods which attempt to learn either a value function or a policy from experiences. In contrast model-based approaches first learn a model of the world from experiences and then use this for planning and acting. Model-based approaches have been shown to have better sample efficiency and faster convergence in certain settings.

We will also have a look at Monte-Carlo tree search (MCTS) and its variants which can be used for planning given a model. MCTS was one of the main ideas behind the success of AlphaGo.

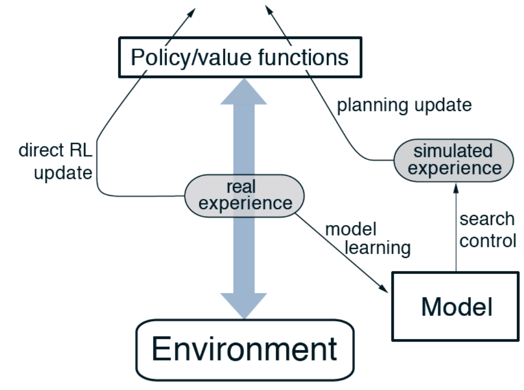


Figure 1: The general Dyna architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

3 Model learning

By model we mean a representation of an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \gamma)$ parametrized by η. In the model learning regime we assume that the state and action spaces $\mathcal{S}, \mathcal{A}$ are known and typically we also assume conditional independence between state transitions and rewards, i.e.,

$$\mathbb{P}(s_{t+1}, r_t \mid s_t, a_t) = \mathbb{P}(s_{t+1} \mid s_t, a_t)\mathbb{P}(r_t \mid s_t, a_t)\,.$$

Hence learning a model consists of two main parts of learning the reward function $R(\cdot \mid s, a)$ and the transition distribution $\mathbb{P}(\cdot \mid s, a)$, where $R(\cdot \mid s, a)$ is assumed to be a deterministic realization of $\mathcal{R}(\cdot)$.

Given a set of real trajectories $\{s_t^k, a_t^k, r_t^k, \dots, s_T^k\}_{k=1}^K$, the model learning can be posed as a supervised learning problem. Learning the reward function $R(s, a)$ is a regression problem whereas learning the transition function $\mathbb{P}(s' \mid s, a)$ is a density estimation problem. First we pick a suitable family of parametrization models. These may include table lookup models, linear expectation, linear Gaussian, Gaussian process, deep belief network, etc. Subsequently we choose an appropriate loss function, for example mean squared error, KL-divergence, etc., to optimize the choice of parameters that minimize this loss.

4 Model-based planning and learning

By a model of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward. If the model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities; these we call distribution models. Other models produce just one of the possibilities, sampled according to the probabilities; these we call sample models. In either case, models can be used to simulate the environment and produce simulated experience.

The word planning is used in several different ways in different fields. We use the term to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment. State-space planning, the canonical planning approach, is viewed primarily as a search through the state space for an optimal policy or an optimal path to a goal. Actions cause transitions from state to state, and value functions are computed over states. All state-space planning methods share a common structure, a structure that is also present learning methods. There are two basic ideas: (1) all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy, and (2) they compute value functions by updates or backup operations applied to simulated experience. This common structure can be diagrammed as follows:

$$\begin{tikzcd}\text{model} \arrow[r] & \begin{array}{c}\text{Simulated}\\\text{Experience}\end{array} \arrow[r, "\text{backups}"] & \text{values} \arrow[r] & \text{policy}\,.\end{tikzcd}$$

For example, dynamic programming methods (Algorithm 2, LN11) clearly fit this structure: they make sweeps through the space of states, generating for each state the distribution of possible transitions. Each distribution is then used to compute a backed-up value (update target) and update the state's estimated value

The heart of both learning and planning methods is the estimation of value functions by backing-up update operations. The difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment (for which, we will be discussing for the second half of the semester). Learning methods require only experience as input, and in many cases they can be applied to simulated experience just as well as to real experience.

4.1 Dyna: Integrated planning, acting, and learning

When planning is done online, while interacting with the environment, a number of interesting issues arise. New information gained from the interaction may change the model and thereby interact with planning. It may be desirable to customize the planning process in some way to the states or decisions currently under consideration, or expected in the near future. If decision making and model learning are both computation-intensive processes, then the available computational resources may need to be divided between them. To begin exploring these issues, in this section we present Dyna-Q, a simple architecture integrating the major functions needed in an online planning agent.

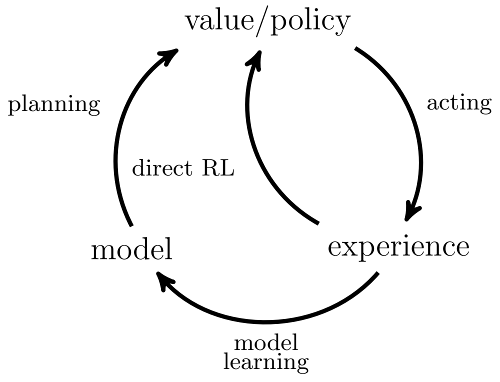


Figure 2: Relationships between experience, model, values, and policy

Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (to make it more accurately match the real environment) and it can be used to directly improve the value function and policy using the kinds of planning value/policy reinforcement learning methods. The former we call model learning, and the latter we call direct reinforcement learning (direct RL). The possible relationships between experience, model, values, and policy are summarized in Diagram 2. Note how experience can improve value functions and policies either directly or indirectly via the model. It is the latter, which is sometimes called indirect reinforcement learning, that is involved in planning.

Both direct and indirect methods have advantages and disadvantages. Indirect methods often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions. On the other hand, direct methods are much simpler and are not affected by biases in the design of the model.

Dyna-Q includes all of the processes shown in the diagram above - planning, acting, model learning, and direct RL - all occurring continually. The planning method is the random-sample one-step tabular Q-planning method. The direct RL method is one-step tabular Q-learning. The model-learning method is also table-based and assumes the environment is deterministic.

The overall architecture of Dyna agents, of which the Dyna-Q algorithm is one example, is shown in Figure 3. The central column represents the basic interaction between agent and environment, giving rise to a trajectory of real experience. The arrow on the left of the figure represents direct reinforcement learning operating on real experience to improve the value function and the policy. On the right are model-based processes. The model is learned from real experience and gives rise to simulated experience. We use the term search control to refer to the process that selects the starting states and actions for the simulated experiences generated by the model. Finally, planning is achieved by applying reinforcement learning methods to the simulated experiences just as if they had really happened. Typically, as in Dyna-Q, the same reinforcement learning method is used both for learning from real experience and for planning from simulated experience. The reinforcement learning method is thus the “final common path” for both learning and planning. Learning and planning are deeply integrated in the sense that they share almost all the same machinery, differing only in the source of their experience.

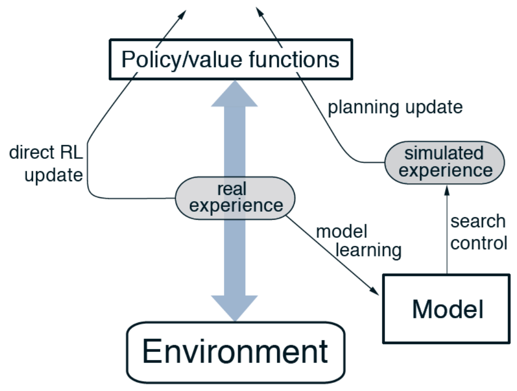


Figure 3: The general Dyna architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

5 Planning

Given a learned model of the environment, planning can be accomplished by any of the methods we have studied so far like value-based methods, policy search or tree search which we will describe later in this lecture.

A contrasting approach to planning uses the model to only generate sample trajectories and methods like Q-learning, Monte-Carlo control, SARSA, etc. for control. These sample-based planning methods are often more data efficient.

The model we learn can be inaccurate and as a result the policy learned by planning for it would also be suboptimal, i.e., model-based RL is dependent on the quality of the learned model. Techniques based on the exploration-exploitation trade-off can be used to explicitly reason about this uncertainty in the model while planning. Alternatively if we ascertain the model to be wrong in certain situations model-free RL methods can be used as a fall back.

6 Simulation-based search

Given access to a model of the world (either an approximate learned model or an accurate simulation in the case of games like Go) simulation-based search methods seek to identify the best action to take based on forward search and simulations. A search tree is built with the current state as the root and the other nodes as states generated using the model. Such methods can give big savings on computational cost as we do not need to solve the entire MDP but just the sub MDP starting from the current state. In general once we have gathered this set of simulated experience $\{s_t^k, a_t^k, r_t^k, \ldots, s_T^k\}_{k=1}^K$ we can apply model free methods for control like Monte-Carlo control, giving us a Monte-Carlo search algorithm, or SARSA, giving us a TD search algorithm.

More concretely, in a simple MC search algorithm given a model $\mathcal{M}$ and a simulation policy π, for each action $a \in \mathcal{A}$ we simulate for K episodes of the form $\{s_t^k, a_t^k, r_t^k, \ldots, s_T^k\}_{k=1}^K$ (which follows π after the first action and onward). The $Q(s_t, a)$ value is evaluated as the average reward of the above trajectories and we subsequently pick the action which maximizes this estimated $Q(s_t, a)$ value.

6.1 Monte-Carlo tree search

This family of algorithms is based on two principles. First, the true value of a state can be estimated using average returns of random simulations. Second, these values can be used to iteratively adjust the policy in a best first nature allowing us to focus on high value regions of the search space.

We progressively construct a partial search tree starting out with the current node set as the root. The tree consists of nodes corresponding to states s. Additionally, each node stores statistics such as the total visitation count N_s, a count for each state-action pair $N_{s,a}$ and the Monte-Carlo action value estimates $Q(s, a)$. A typical implementation builds this search tree until some preset computational budget is exhausted with the value estimates (particularly for the promising moves) becoming more and more accurate as the tree grows larger. Each iteration can be roughly divided into four phases.

1. **Selection.** Starting at the root node, we select child nodes recursively in the tree till a non-terminal leaf node is reached.

2. **Expansion.** The chosen leaf node is added to the search tree.

3. **Simulation.** Simulations are run from this node to produce an estimate of the outcomes.

4. **Backpropagation.** The values obtained in the simulations are backpropagated through the tree by following the path from the root to the chosen leaf in reverse and updating the statistics of the encountered nodes.

Variants of MCTS generally contain modifications to the two main policies involved.

- **Tree policy.** To chose actions for nodes in the tree based on the stored statistics. Variants include greedy, UCB.
- **Rollout policy.** For simulations from leaf nodes in the tree. Variants include random simulation, default policy network in the case of AlphaGo.

These steps are summarized in Algorithm 1, where we start out from the current state and iteratively grow the search tree, using the tree policy at each iteration to chose a leaf node to simulate from. Then we backpropagate the result of the simulation and finally output the action which has the maximum value estimate form the root node.

Algorithm 1: General MCTS algorithm

```
Input $s_0$
Create root node $s_0$
while within computational budget do
    $s_k \leftarrow TreePolicy(s_0)$
    $\Delta \leftarrow Simulation(s_k)$
    $Backprop(s_k, \Delta)$
Return $\arg\max_a Q(s_0, a)$
```

A simple variant of this algorithm chooses actions greedily amongst the tree nodes in the first stage, as implemented in Algorithm 2, which also generates rollouts using a random policy in the simulation stage.

Algorithm 2: Greedy MCTS algorithm

```
Input $s_0$
Create root node $s_0$
while within computational budget do
    $s_{next} \leftarrow s_0$
    while $|Children(s_{next})| \neq 0$ do
        $a \leftarrow \arg\max_{a \in \mathcal{A}} Q(s_{next}, a)$
        $s_{next} \leftarrow NextState(s_{next}, a)$
    $\Delta \leftarrow RandomSimulation(s_{next})$
    $Backprop(s_{next}, \Delta)$
Return $\arg\max_a Q(s_0, a)$
```

Various modifications of the above scheme exist to improve memory usage by adding a limited set of nodes to the tree. Smart pruning and tree policies based on using more complicate statistics stored in the nodes are also investigated.

A run through of MCTS is visualized in Figure 4. We start out from the root node and simulate a trajectory which gives us a reward of 1 in this case. We then use the tree policy which greedily selects the node to add to the tree and subsequently simulate an episode from it using the simulation (default) policy. This episode gives us a reward of 0 and we update the statistics in the tree nodes. This process is then repeated, to check the understanding of MCTS we direct the reader to verify in the below example that the statistics have been updated correctly and that the tree policy chooses the correct node to expand.

The main advantages of MCTS include

1. Its tree structure makes the computation feasible in parallel.
2. States are evaluated dynamically, that is, MDP are only solved from the current state unlike dynamic programming.
3. The method is end-to-end without the need of domain-specific engineering.
4. It efficiently combines planning and sampling to break the curse of dimensionality in complicated games like Go.

6.2 Upper confidence tree search

Similar to the multi-armed bandit case, using a greedy policy as the tree policy can often be suboptimal, making us avoid actions after even one bad outcome despite the significant uncertainty about its true value. As an example, consider the rightmost node in Figure 4 from which we do a single rollout, receive a 0 reward and never visit it again even though this reward might have just been due to randomness. To fix this, we can apply the principle of optimism under uncertainty to MCTS using the UCB algorithm. More specifically the tree policy instead of picking the action greedily picks the action which maximizes the upper confidence bound on the value of the action, which is

$$Q(s,a) + \sqrt{\frac{2 \log N_s}{N_{s,a}}}\,.$$

The pseudocode of the tree policy used in UCT is illustrated in Algorithm 5. It can be plugged into the general MCTS algorithm described earlier. Note that the $NextState(s,a)$ function uses the model of the MDP to sample a next state when picking action a from state s.

7 Case study: Go

Go is a very famous game board game that have been continuously played in the world. Solving it had been a long standing challenge for AI and before AlphaGO traditional game tree search algorithms had failed to achieve professional human-level performance on it. Go is a two player game (Black/White). It is played on a 19×19 board (while smaller variants exist). Black and White alternatively place down stones on the board. The main goal of the game is to surround and capture territories. Additionally, stones surrounded by the opponent are removed.

The simplest reward function gives a reward of $+1$ if Black wins and -1 if White wins on terminal states and 0 elsewhere. As a result the goal of the Black player is to maximize

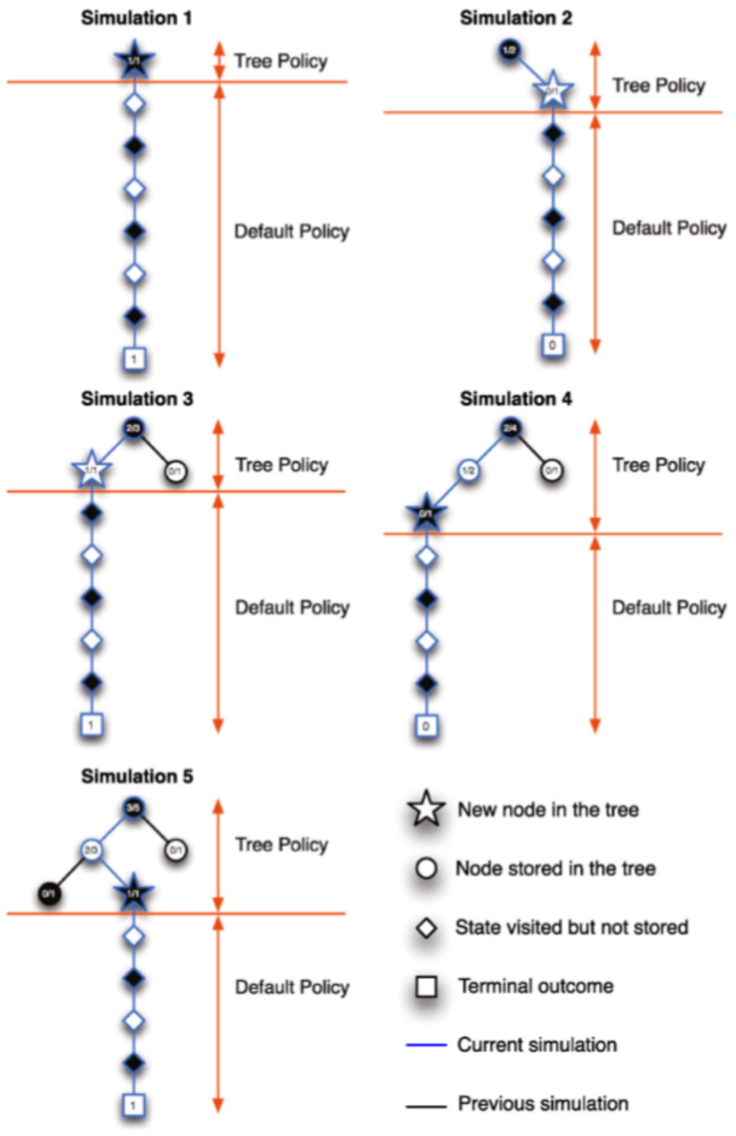


Figure 4: Demonstration of a simple MCTS. Each state has two possible actions (left/right) and each simulation has a reward of 1 or 0. At each iteration a new node (star) is added into the search tree. The value of each node in the search tree (circles and star) and the total number of visits is then updated.

Algorithm 3: Upper Confidence MCTS algorithm

```
Input $s_0$
Create root node $s_0$
while within computational budget do
    $s_{next} \leftarrow s_0$
    while $|Children(s_{next})| \neq 0$ do
        $a \leftarrow \arg\max_{a \in \mathcal{A}} Q(s,a) + \sqrt{\frac{2 \log \sum_b N_{s,b}}{N_{s,a}}}$
        $s_{next} \leftarrow NextState(s_{next}, a)$
    $\Delta \leftarrow RandomSimulation(s_{next})$
    $Backprop(s_{next}, \Delta)$
Return $\arg\max_a Q(s_0, a)$
```

Algorithm 4: Predictor Upper Confidence MCTS algorithm

```
Input $s_0$, $\pi_0$
Create root node $s_0$
while within computational budget do
    $s_{next} \leftarrow s_0$
    while $|Children(s_{next})| \neq 0$ do
        $a \leftarrow \arg\max_{a \in \mathcal{A}} Q(s,a) + c_{puct}\pi_0(a|s)\frac{\sqrt{\sum_b N_{s,b}}}{N_{s,a}+1}$
        $s_{next} \leftarrow NextState(s_{next}, a)$
    $\Delta \leftarrow StrategicSimulation(s_{next}, \pi_0)$
    $Backprop(s_{next}, \Delta)$
Return $\arg\max_a Q(s_0, a)$
```

Algorithm 5: Predictor Upper Confidence MCTS algorithm with Priors

```
Input $s_0$, $\pi_0$, $V_0$
Create root node $s_0$
while within computational budget do
    $s_{next} \leftarrow s_0$
    while $|Children(s_{next})| \neq 0$ do
        $a \leftarrow \arg\max_{a \in \mathcal{A}} Q(s,a) + c_{puct}\pi_0(a|s)\frac{\sqrt{\sum_b N_{s,b}}}{N_{s,a}+1}$
        $s_{next} \leftarrow NextState(s_{next}, a)$
    $\Delta \leftarrow StrategicSimulation(s_{next}, \pi_0)$
    $V(s_{next}) = (1-\lambda)V_0(s_{next}) + \lambda\Delta$
    $Backprop(s_{next}, V(s_{next}))$
Return $\arg\max_a Q(s_0, a)$
```

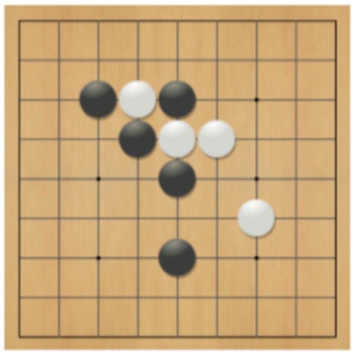
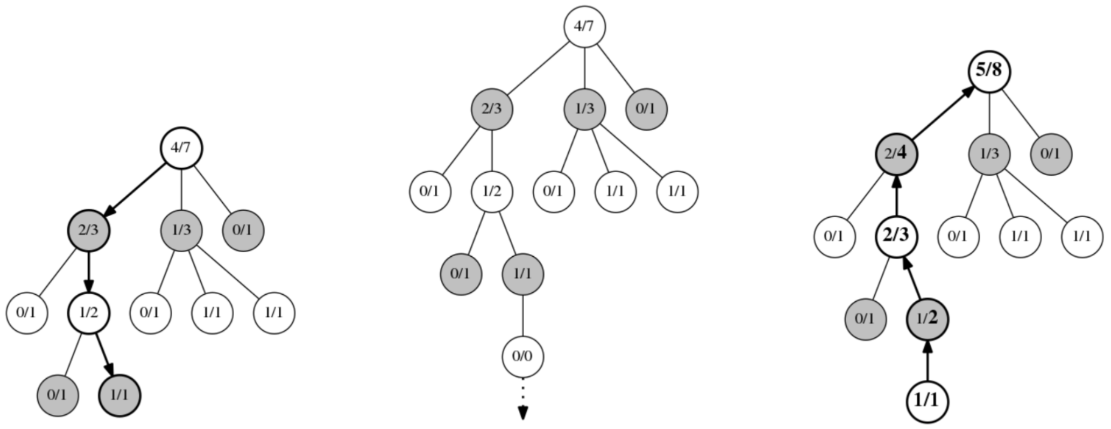


Figure 5: Go.



(a) Tree phase. (b) Rollout phase. (c) Update statistics.

Figure 6: UCT on Go. Colored nodes represent the players and each node contains the probability of that player winning the game. The figure is from [Bra].

the reward and White tries to minimize it. Given a policy $\pi = (\pi_B, \pi_W)$ for both players the value function is $V_\pi(s) = \mathbb{E}_\pi[R_T \mid s] = \mathbb{P}(\text{Black wins} \mid s)$ and the optimal value function is $V^*(s) = \max_{\pi_B} \min_{\pi_W} V_\pi(s)$.

7.1 MCTS for Go

Go, being a two-player game, warrants some fairly natural extensions to the previously discussed MCTS algorithm. We now build a minimax tree with nodes alternating players across levels. The White nodes seek to minimize while the black ones maximize the value. We use UCB as described above at the black nodes and LCB (Lower confidence bound, i.e., $\min_a Q(s,a) - \sqrt{\frac{2 \log N_s}{N_{s,a}}}$ at the white nodes as they seek to minimize the reward.

Consider the following state of the tree in Figure 6a with the statistics (number of wins/number of total games) recorded in the nodes and the colors representing the players.

The first phase of the algorithm (the tree policy) would use these statistics in the nodes, treating each of them as an independent MAB instance and would sequentially choose actions using UCB (LCB) from the root onward. These are highlighted with bold arrows

Once we reach a leaf node in this tree (Figure 6b) we use our rollout policy to simulate a game. The outcome of this game is then backpropagated through the tree (Figure 6c) and the statistics are updated.

This process continues until desired. The best action to take from the root are returned subsequently. For detailed pseudocode we direct the reader to [GS11] and [Bra] for a python implementation.

AlphaGo [SHM$^+$16] uses a deep policy network for the rollout phase. This allows the simulations to be much more realistic than just using random rollouts. Also in a complicated game like Go it is not feasible to simulate till completion. Thus early stopping is used along with a value network to get the required win probabilities. Following that, AlphaGo Zero [SSS$^+$17] is proposed, which uses a single network to output both the policy and the value function and is trained purely using self play with no expert knowledge input. This gets even more impressive performance than AlphaGo.

Acknowledgement

This lecture notes partially use material from *Reinforcement learning: An introduction* and *CS234: Reinforcement learning* from Stanford.

References

[Bra] Jeff Bradberry. Introduction to monte-Carlo tree search. `https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/`. Accessed: 2021-04.

[GS11] Sylvain Gelly and David Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[SHM$^+$16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[SSS$^+$17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.