Deep Reinforcement Learning

Deep Learning (ENGG*6600*07)

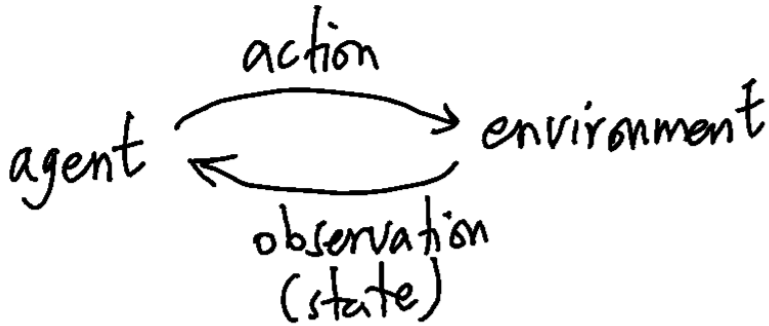
School of Engineering,
University of Guelph, ON, Canada

Course Instructor: Benyamin Ghojogh
Fall 2023

Introduction

Introduction

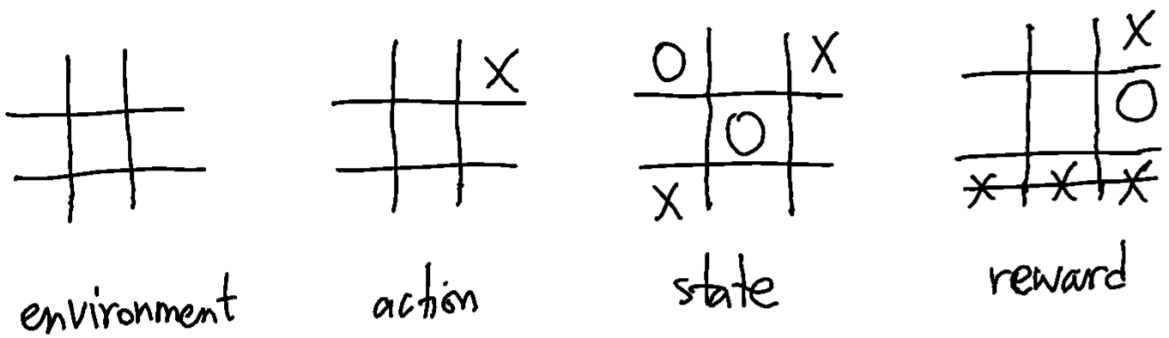
- **Reinforcement Learning (RL)** [1] is an area in machine learning where one or several **agents** take **actions** in an **environment** to maximize their **cumulative reward**.
- In other words, RL is learning to make decisions from interactions.



- If there are multiple agents cooperating with each other, i.e., learning the environment together, reinforcement learning is called **Multi-Agent Reinforcement Learning (MARL)** [2].
- It is like how **animals**, such as dogs, are trained. Suppose we want to train a dig. Food and hunger can be used as positive and negative rewards, respectively, to teach it to perform some particular actions in specific situations. As a result, the dog is trained using reinforcement. In reinforcement learning, the machine is like a dog which is trained to perform suitable actions in the situations of the environment.

Elements of Reinforcement Learning

- Reinforcement learning contains some elements. In the following, some examples of these elements are introduced in the game of tic-tac-toe.
  - **environment**: the environment in which the agents play
  - **action**: the action which the agents play
  - **state**: an state or realization of the environment. The environment has a state at every particular time.
    - ★ The state s can be a vector or matrix representing the state of environment. For example, in tic-tac-toe, it is a nine-dimensional vector whose every element can get one of three values (one value for cross, one value for circle, and one value for empty). In inverted pendulum, the state is four-dimensional vector having the values for position, velocity, angle, and angular velocity. In Atari games, it can be the frame (RGB image) of the game at the current time. In the game of Go or chess, it can be a matrix or reshaped vector having values for the cells of the board game.
  - **reward**: the reward that the agents get when they play some action. Better actions have more rewards.



Markov Decision Process

Markov Process

- Consider a times series of random variables $s_1, s_2, \ldots, s_n$. In general, the joint probability of these random variables can be written as:

$$\mathbb{P}(s_1, s_2, \ldots, s_n) = \mathbb{P}(s_1)\,\mathbb{P}(s_2 \mid s_1)\,\mathbb{P}(s_3 \mid s_2, s_1)\ldots\mathbb{P}(s_n \mid s_{n-1}, \ldots, s_2, s_1), \tag{1}$$

  according to chain (or multiplication) rule in probability.
- (The first order) Markov property is an assumption which states that in a time series of random variables $s_1, s_2, \ldots, s_n$, every random variable is merely dependent on the latest previous random variable and not the others [3]. In other words:

$$\mathbb{P}(s_i \mid s_{i-1}, s_{i-2}, \ldots, s_2, s_1) = \mathbb{P}(s_i \mid s_{i-1}). \tag{2}$$

- Hence, with Markov property, the chain rule is simplified to:

$$\mathbb{P}(s_1, s_2, \ldots, s_n) = \mathbb{P}(s_1)\,\mathbb{P}(s_2 \mid s_1)\,\mathbb{P}(s_3 \mid s_2)\ldots\mathbb{P}(s_n \mid s_{n-1}). \tag{3}$$

- The Markov property can be of any order. For example, in a second order Markov property, a random variable is dependent on the latest and one-to-latest variables. Usually, the default Markov property is of order one. A stochastic process which has the Markov property is called a Markovian process or a Markov process.

Markov Decision Process

- **Markov Decision Process (MDP)** is a decision process which is Markov process, too. In other words, every state is conditioned on only the previous state in MDP where all the past information is assumed to eb encapsulated in the previous state.
- An MDP is defined by a tuple $\{S, A, R, P, \gamma\}$:
  - States: $s \in S$
  - Actions: $a \in A$
  - Rewards: $r \in R$, reward model: $\mathbb{P}(r_t|s_t, a_t)$
  - Transition model: $\mathbb{P}(s_t|s_{t-1}, a_{t-1})$
  - Discount factor: $\gamma \in [0, 1]$
    - ★ discounted: $\gamma < 1$, undiscounted: $\gamma = 1$
    - ★ discounted rewards: $r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$
- Note that the total reward is important and not merely the short-term reward. sometimes, it is better to sacrifice immediate reward to gain more long-term reward. An example is playing chess where a player may sacrifice their queen to win the chess in ten later moves.

Markov Decision Process

- At every time slot $t \in \{0, 1, 2, \dots\}$, the agent is at state s_t.
- It takes an action a_t.
- By this action, it moves to a new state s_{t+1}, according to $s_{t+1} \sim \mathbb{P}(s_{t+1}|s_t, a_t)$.
- It recieves some reward $r_t \sim \mathbb{P}(r_t|s_t, a_t)$.
- A **policy** π is a mapping from states to actions:

$$\pi : S \to A, \quad \pi : s \mapsto a. \tag{4}$$

  - In other words, it determines what action should be performed at every state of the environment.
  - We can have deterministic policy:

$$a_t = \pi(s_t), \tag{5}$$

    or a stochastic policy:

$$a_t = \pi(a_t|s_t). \tag{6}$$

- The goal of MDP is to find a policy π^* which maximizes the long-term reward:

$$\pi^* = \arg\max_{\pi} \sum_{t=0}^{h} \gamma^t \mathbb{E}_{\pi}[r_t], \tag{7}$$

where t is the time slot index, h is the horizon, and $\mathbb{E}_{\pi}[.]$ denotes expectation with respect to policy.

Bellman's Equation

Bellman's Equation

- Value is defined to be the maximum or best reward that the agent can get in some state. Therefore, value is in the essence of reward. The value at state s_t and time t is denoted by $v(s_t)$ or $v_t(s)$.
- Consider a time period or horizon h. The last time slot of this horizon is at time t and its value is:

$$v(s_t) = \max_{a_t} r(s_t, a_t).$$

- The value at one-to-last time slot of the horizon is:

$$v(s_{t-1}) = \max_{a_{t-1}} r(s_{t-1}, a_{t-1}) + \gamma \mathbb{P}(s_t | s_{t-1}, a_{t-1})\, v(s_t).$$

- The value at two-to-last time slot of the horizon is:

$$v(s_{t-2}) = \max_{a_{t-2}} r(s_{t-2}, a_{t-1}) + \gamma \mathbb{P}(s_{t-1} | s_{t-2}, a_{t-2})\, v(s_{t-1}),$$

and so on. This series can be generalized into the following equation, named the **Bellman's equation**:

$$v(s_t) = \max_{a_t} r(s_t, a_t) + \gamma \sum_{s_{t+1}} \mathbb{P}(s_{t+1} | s_t, a_t)\, v(s_{t+1}). \tag{8}$$

Bellman's Equation

- We found the **Bellman's equation**:

$$v(s_t) = \max_{a_t} r(s_t, a_t) + \gamma \sum_{s_{t+1}} \mathbb{P}(s_{t+1}|s_t, a_t)\, v(s_{t+1}).$$

- In the literature of reinforcement learning, the variable at the next time slot $(t+1)$ is also denoted by a prime at the variable while the variable at the current time slot t is denoted without prime. For example, the states s_t and s_{t+1} are also denoted by s and s', respectively.
- The value also may have the superscript π, i.e., $v^\pi(s)$, showing that its policy is π.
- Therefore, another notation of this equation is:

$$v(s) = \max_{a} r(s, a) + \gamma \sum_{s'} \mathbb{P}(s'|s, a)\, v(s'). \tag{9}$$

It is also sometimes denoted using expectation:

$$v(s) = \mathbb{E}[r|s, \pi(s)] + \gamma \sum_{s'} \mathbb{P}(s'|s, \pi(s))\, v(s'), \tag{10}$$

where $\pi(s) = a$.

Solving MDP

Value Iteration

- Recall that the goal of MDP is to find the optimal policy.
- There are two ways to solve the MDP problem, i.e., to find the optimal policy. These two approaches are **value iteration** and **policy iteration**.
- Let h denote the horizon and t denote the time slot in the horizon. The algorithm of **value iteration** calculates the optimal values backwards in the horizon:

Algorithm Value iteration

$v_0^* \leftarrow \max_a r(s,a), \quad \forall s$

for t *from* 1 *to* h **do**

$\quad v_t^*(s) \leftarrow \max_a r(s,a) + \gamma \sum_{s'} \mathbb{P}(s'|s,a)\, v_{t-1}^*(s'), \quad \forall s$

Return v_h^*

- After finding the optimal value by value iteration, the optimal policy can be found for the last time slot and other time slots in the horizon:

$$
\begin{aligned}
t = 0 &: \pi_0^*(s) \leftarrow \arg\max r(s,a), \quad \forall s, \\
t > 0 &: \pi_t^*(s) \leftarrow \max_a r(s,a) + \gamma \sum_{s'} \mathbb{P}(s'|s,a)\, v_{t-1}^*(s'), \quad \forall s.
\end{aligned} \tag{11}
$$

As it is obvious from the above equations, the optimal policy π^* is non-stationary, i.e., it is time-dependant.

Policy Iteration

- Another way to find the optimal policy in MDP is **policy iteration**.
- Policy iteration alternates between two steps iteratively until convergence. These two steps are **policy evaluation** and **policy improvement**. Policy evaluation evaluates how good the found policy is so far. Policy improvement improves the policy.
- The algorithm of **policy iteration** is as follows.

Algorithm Policy iteration

Initialize $\pi(s)$ with a random action for every state s.
while *not converged* **do**
$v^{\pi}(s) \leftarrow r(s, \pi(s)) + \gamma \sum_{s'} \mathbb{P}(s'|s, \pi(s))\, v^{\pi}(s'), \quad \forall s$
$\pi(s) \leftarrow \arg\max_{a} r(s, a) + \gamma \sum_{s'} \mathbb{P}(s'|s, a)\, v^{\pi}(s'), \quad \forall s$
Return $v^{\pi}(s)$ and $\pi(s)$

- Note that these equations are both the Bellman's equation where, in policy evaluation, the action a is replaced with the policy $\pi(s)$ which is in the essence of action (because policy is the mapping from state to action).

Value Iteration vs. Policy Iteration

- In value iteration, the policy evaluation step is repeated during the horizon to maximize the value $v^{\pi}(s)$ and then policy improvement step is performed only once to find the optimal policy. In policy iteration, however, policy evaluation and improvement are repeated alternatively until convergence.
- The time complexities of every iteration in value iteration and policy iteration are $\mathcal{O}(|S|^2|A|)$ and $\mathcal{O}(|S|^3 + |S|^2|A|)$, respectively, where $|S|$ is the number of states and $|A|$ is the number of actions. The number of iterations in value iteration and policy iteration are linear convergence and quadratic convergence, respectively. As a result, compared to policy iteration, value iteration has less computation at every iteration but it needs more number of iterations.

Modified Policy Iteration

- Therefore, there is a method, named **modified policy iteration**, which is a middle case scenario between value iteration and policy iteration to have the best of two worlds. The modified policy iteration is more practical and it usually converges faster than value iteration and policy iteration.
- This algorithm is similar to the policy iteration algorithm except that it performs its policy iteration step for multiple times, inspired by value iteration.
- The algorithm of **modified policy iteration** is as follows:

Algorithm Modified policy iteration

Initialize $\pi(s)$ with a random action for every state s.
while *not converged* **do**
  for k *times* **do**
    $v^{\pi}(s) \leftarrow r(s, \pi(s)) + \gamma \sum_{s'} \mathbb{P}(s'|s, \pi(s))\, v^{\pi}(s'), \quad \forall s$
  $\pi(s) \leftarrow \arg\max_{a} r(s, a) + \gamma \sum_{s'} \mathbb{P}(s'|s, a)\, v^{\pi}(s'), \quad \forall s$
Return $v^{\pi}(s)$ and $\pi(s)$

- Note that these equations are both the Bellman's equation where, in policy evaluation, the action a is replaced with the policy $\pi(s)$ which is in the essence of action (because policy is the mapping from state to action).

Reinforcement Learning

Reinforcement Learning

- **Reinforcement Learning (RL)** is similar to MDP except that the **transition model is not known** in it. Not having the transition model is more realistic as it is usually not available in practice. As a result, RL is a harder task compared to MDP. It is defined by a tuple $\{S, A, R, P, \gamma\}$:
  - States: $s \in S$
  - Actions: $a \in A$
  - Rewards: $r \in R$, reward model: $\mathbb{P}(r_t|s_t, a_t)$
  - Discount factor: $\gamma \in [0, 1]$
    - ★ discounted: $\gamma < 1$, undiscounted: $\gamma = 1$
    - ★ discounted rewards: $r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$
- The goal of RL, so as in MDP, is also finding the optimal policy.

Reinforcement Learning

- There are two main approaches for finding the best policy in RL:
  - **Model-based RL** [4]: it finds the model first and then solves the problem. Specifically, it finds the transition model and the reward model and then solves the problem.
  - **Model-free RL** [5]: it finds the policy without the transition model and the reward model. In the model-free RL, the model is unknown explicitly and it is implicitly found to solve the problem.
- Here, we go over the model-free RL algorithms.

Temporal Difference Evaluation

Monte-Carlo Evaluation

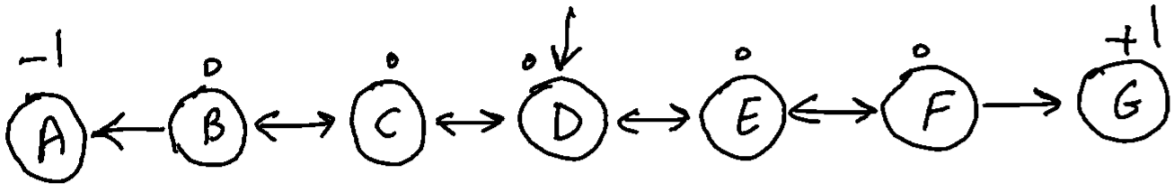
- In the model-free RL, the value $v^{\pi}(s)$ should be estimated, given a policy π, without any transition model.
- Recall that value is the expected total rewards with respect to the policy:

$$v^{\pi}(s) = \mathbb{E}_{\pi}\Big[\sum_{t=0}^{h}\gamma^{t} r_t\Big] \overset{(a)}{\approx} \frac{1}{n(s)}\sum_{k=1}^{n(s)}\Big[\sum_{t=0}^{h}\gamma^{t} r_t^{(k)}\Big], \tag{12}$$

  where (a) is because of the Monte-Carlo approximation (evaluation) of expectation [6], h is the horizon, $n(s)$ is the number of times that the state s happens, and $r_t^{(k)}$ is the reward at time t in the horizon and the k-th episode. the larger the $n(s)$, the more accurate the approximation is.
- For this Monte-Carlo evaluation of the value, the agent should play in the environment for k times. An *episode* is referred to each time that the agent plays in the environment once completely.

Monte-Carlo Evaluation

- For example, in the following game, the possible actions at every middle node is going left or right stochastically. Let the rewards of nodes A and G be -1 and $+1$, respectively, and the other nodes have reward zero. The game starts from node D. The policy is stochastic left or right movements with different probabilities on the nodes.
- For a particular policy (particular probabilities of actions on nodes), the game is played for k episodes. Every episode is represented as a trajectory ending on either node A or node G. The Monte-Carlo evaluation is the average of total rewards across the trajectories.



Temporal Difference Evaluation

- The problem with the Monte-Carlo evaluation is that it needs many iterations, i.e., a large $n(s)$, to be accurate. For addressing this problem, temporal difference, introduced in the following, is used.
- Let G_k be a one-trajectory Monte-Carlo evaluation of the value:

$$G_k := \sum_{t=0}^{h} \gamma^t r_t^{(k)}, \tag{13}$$

which is the total discounted reward.

- According to Eq. (12), the value is:

$$\begin{aligned} v^{\pi}_{n(s)}(s) &\approx \frac{1}{n(s)} \sum_{k=1}^{n(s)} G_k \qquad (14) \\ &= \frac{1}{n(s)} \left(G_{n(s)} + \sum_{k=1}^{n(s)-1} G_k \right) \overset{(14)}{=} \frac{1}{n(s)} \left(G_{n(s)} + (n(s)-1) v^{\pi}_{n(s)-1}(s) \right) \\ &= v^{\pi}_{n(s)-1}(s) + \frac{1}{n(s)} \left(G_{n(s)} - v^{\pi}_{n(s)-1}(s) \right). \end{aligned}$$

Therefore, the Monte-Carlo evaluation can be stated as an **incremental update**:

$$v^{\pi}_n(s) = v^{\pi}_{n-1}(s) + \alpha_n \left(G_n - v^{\pi}_{n-1}(s) \right), \tag{15}$$

where n is the simplified notation of $n(s)$ and $\alpha_n := 1/n(s)$.

Temporal Difference Evaluation

- We found:

$$v_n^\pi(s) = v_{n-1}^\pi(s) + \alpha_n\big(G_n - v_{n-1}^\pi(s)\big).$$

- The **temporal difference** evaluation is similar to the incremental update except that it replaces G_n with $r + \gamma v_{n-1}^\pi(s')$ where s' is the next state at time $t+1$. This replacement makes sense because, according to Eq. (13), G_n is the value (or total rewards) of a trajectory. In other words, recall the Bellman's equation in Eq. (10). G_n is the Monte-Carlo evaluation of the value using only one trajectory, which is not a very accurate approximation:

$$v(s) = \mathbb{E}[r|s, \pi(s)] + \gamma \sum_{s'} \mathbb{P}(s'|s, \pi(s))\, v(s') \approx r + \gamma v^\pi(s'), \tag{16}$$

  where that one trajectory has only one s' in it.

- Therefore, the temporal difference evaluation is formulated as:

$$v_n^\pi(s) = v_{n-1}^\pi(s) + \alpha_n\big(r + \gamma v_{n-1}^\pi(s') - v_{n-1}^\pi(s)\big). \tag{17}$$

Temporal Difference Evaluation

Theorem

If α_n is appropriately decreased with the number of times a state is visited, then $v_n^\pi(s)$ converges to the correct value. Sufficient conditions for α_n are:

$$\lim_{k\to\infty} \sum_{n=1}^{k} \alpha_n = \infty, \tag{18}$$

$$\lim_{k\to\infty} \sum_{n=1}^{k} (\alpha_n)^2 < \infty. \tag{19}$$

- α_n is called the **learning rate** and it can be any function of n satisfying the above sufficient conditions. Often, $\alpha_n = 1/n(s)$ is used where $n(s)$ is the number of times the state s is visited.

Temporal Difference Evaluation

- Instead of trying to calculate total future reward, temporal difference simply tries to predict the combination of immediate reward and its own reward prediction at the next moment in time.
- When the next moment comes, the new prediction is compared against what it was expected to be. In other words, the temporal difference is calculated.
- This temporal difference is used to adjust the old prediction toward the new prediction.
- An example of using temporal difference is as follows. Consider weather prediction where the temperatures of all days in the coming week are predicted. When one of the days of the week arrives, the predicted temperature of that day can be compared with the actual temperature of that day. If the difference between them is small, it means that the prediction was accurate but if the difference is large, the predicted temperature of later days should be adjusted based on that calculated error. Assume the predicted temperature of Tuesday was 19 and the actual temperatures of Monday and Tuesday are 20 and 23. The temporal difference evaluation says that the predicted temperature of Tuesday should be adjusted to $20 + \alpha(23 - 19)$ where $\alpha > 0$ is some small multiplier.

Temporal Difference Evaluation

- The algorithm of **temporal difference evaluation** of value is as follows:

Algorithm Temporal difference evaluation

Input: policy π
while v^{π} *is not converged* **do**
    Execute $\pi(s)$
    Observe s and s'
    $n(s) \leftarrow n(s) + 1$
    $\alpha \leftarrow 1/n(s)$
    $v^{\pi}(s) \leftarrow v^{\pi}(s) + \alpha\big(r + \gamma v^{\pi}(s') - v^{\pi}(s)\big)$
    $s \leftarrow s'$
Return v^{π}

- As the temporal difference evaluation of value is making use of the incremental updates, it has much less computation compared to the Monte-Carlo evaluation of value. The Monte-Carlo evaluation requires to calculate the total trajectori(es) but temporal difference only calculates the differences of values of successive states along the trajectory.

Q-Function

Q-Function

- Instead of evaluating the state value $v^{\pi}(s)$, it is possible to evaluate the state-action value $Q^{\pi}(s, a)$. The state-action value is called the Q-function.
- The state value shows how good it is to be in a particular state s. The Q-function, however, determines how good it is to perform action a, following the policy π, in the state s.
- Recall the Bellman's equation for the state value $v(s)$, introduced in Eq. (10):

$$v(s) = \mathbb{E}[r|s, a] + \gamma \sum_{s'} \mathbb{P}(s'|s, a)\, v(s').$$

- Likewise, the Bellman's equation for the state-action value (Q-function) is:

$$Q(s, a) = \mathbb{E}[r|s, a] + \gamma \sum_{s'} \mathbb{P}(s'|s, a) \max_{a'} Q(s', a'), \tag{20}$$

  where a' is the next action in the next time slot.
- As a result, according to the Bellman's equation, the optimal state value and the optimal Q-function are:

$$v^*(s) = \mathbb{E}[r|s, a] + \gamma \sum_{s'} \mathbb{P}(s'|s, a)\, v^*(s'), \tag{21}$$

$$Q^*(s, a) = \mathbb{E}[r|s, a] + \gamma \sum_{s'} \mathbb{P}(s'|s, a) \max_{a'} Q^*(s', a'). \tag{22}$$

Q-Function

- If Q-function is used, then the optimal value $v^*(s)$ and the optimal policy π^* is then found by maximizing the Q-function:

$$v^*(s) = \max_a Q^*(s, a), \tag{23}$$

$$\pi^*(s) = \arg\max_a Q^*(s, a). \tag{24}$$

Maximizing the Q-function for finding the optimal policy is roughly in the form of Eq. (7):

$$\pi^* = \arg\max_\pi \sum_{t=0}^{h} \gamma^t \mathbb{E}_\pi[r_t],$$

which found the policy by maximizing the value or the total rewards.

Q-Function

- Recall Eq. (15) which was the incremental update for the state value:

$$v_n^\pi(s) = v_{n-1}^\pi(s) + \alpha_n\big(G_n - v_{n-1}^\pi(s)\big),$$

- Likewise, it is possible to have incremental update for the Q-function:

$$Q_n^\pi(s,a) = Q_{n-1}^\pi(s,a) + \alpha_n\big(G_n - Q_{n-1}^\pi(s,a)\big), \tag{25}$$

where G_n is a one-trajectory Monte-Carlo evaluation of the value, defined in Eq. (13):

$$G_n := \sum_{t=0}^{h} \gamma^t r_t^{(n)}.$$

- As a result, the policy iteration algorithm using incremental update of Q-function is as follows, where the algorithm alternates between policy evaluation and policy improvement steps:

Algorithm Policy iteration using Q-function

Initialize $Q^\pi(s,a)$ with a random value for every state s and action a.

while *not converged* **do**

$$Q^\pi(s,a) = Q^\pi(s,a) + \alpha_n\big(G_n - Q^\pi(s,a)\big), \quad \forall s$$
$$\pi(s) \leftarrow \arg\max_a Q^\pi(s,a), \quad \forall s$$

Return $Q^\pi(s,a)$ and $\pi(s)$

Q-Learning

Q-Learning

- Recall Eq. (16) which approximated the Monte-Carlo evaluation of the value using only one trajectory:

$$v(s) = \mathbb{E}[r|s, \pi(s)] + \gamma \sum_{s'} \mathbb{P}(s'|s, \pi(s))\, v(s') \approx r + \gamma v^{\pi}(s'),$$

- Likewise, this approximation for the Q-function is:

$$Q(s,a) \overset{(20)}{=} \mathbb{E}[r|s,a] + \gamma \sum_{s'} \mathbb{P}(s'|s,a) \max_{a'} Q(s',a') \approx r + \gamma \max_{a'} Q(s',a'). \tag{26}$$

- As a result, the temporal difference evaluation of the Q-function is:

$$Q_n^*(s,a) = Q_{n-1}^*(s,a) + \alpha_n \big(r + \gamma \max_{a'} Q_{n-1}^*(s',a') - Q_{n-1}^*(s,a)\big). \tag{27}$$

  which is similar to the temporal difference for state value in Eq. (17).
- It is noteworthy that the Q-function is updated merely by observing s, s', a, and r based on temporal difference. This leads to the Q-learning algorithm in the following.
- It is also worth comparing the updates in Q-learning and value iteration:

$$v_n^*(s) \leftarrow \max_a\ r(s,a) + \gamma \sum_{s'} \mathbb{P}(s'|s,a)\, v_{n-1}^*(s').$$

  Value iteration is for MDP but Q-learning is for RL. In MDP, the transition model is known but RL does not have the transition model; therefore, RL updates the Q-function which depends on both state and action.

Q-Learning

Algorithm Q-learning

while Q^* *is not converged* **do**
- Select and execute action a
- Observe s' and r
- $n(s,a) \leftarrow n(s,a)+1$
- $\alpha \leftarrow 1/n(s,a)$
- $Q_n^*(s,a) = Q_{n-1}^*(s,a) + \alpha_n\left(r + \gamma \max_{a'} Q^*(s',a') - Q^*(s,a)\right)$
- $s \leftarrow s'$

Return $Q^\pi(s,a)$

- The $n(s,a)$ is the number of times the state s and action a are observed together.
- The learning rate $\alpha > 0$ can be any function of $n(s,a)$ satisfying the sufficient conditions in the theorem. A common choice is $1/n(s,a)$.

Q-Learning

Algorithm Q-learning

while Q^* *is not converged* **do**
  Select and execute action a
  Observe s' and r
  $n(s,a) \leftarrow n(s,a)+1$
  $\alpha \leftarrow 1/n(s,a)$
  $Q_n^*(s,a) = Q_{n-1}^*(s,a) + \alpha_n\big(r + \gamma \max_{a'} Q^*(s',a') - Q^*(s,a)\big)$
  $s \leftarrow s'$
Return $Q^*(s,a)$

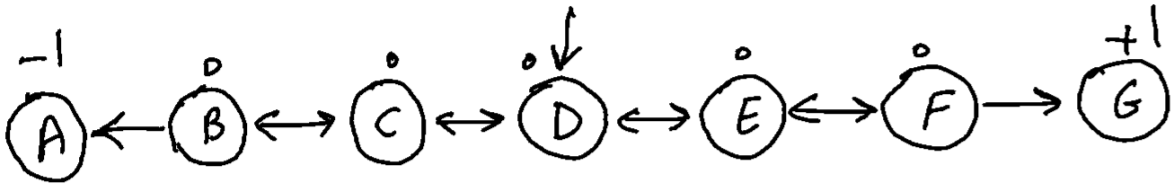
- There are two approaches for selecting the action a in Q-learning:
  - either choose action a at state s randomly,
  - or choose action a which maximizes the Q-function.
- The former approach is for **exploration** which takes time to converge.
- The latter approach is for **exploitation** which converges faster but it may get stuck in the local best solutions without finding the global best solution.
- Therefore, it is better to have a combination of the two approaches for selecting actions in order to use the benefits of both approaches. For example, it is possible to use the first approach for exploration in the initial iterations and then to use the second approach for exploitation in later iterations. This is because initially, the algorithm should explore the possibilities but later, it is supposed to be closer to the best solution and it should not oscillate much around the solution.

Q-Learning: Example

- We had:

$$Q_n^*(s,a) = Q_{n-1}^*(s,a) + \alpha_n\big(r + \gamma \max_{a'} Q^*(s',a') - Q^*(s,a)\big).$$

- Example for Q-learning:



Assume the current Q-function at this iteration for the states and actions are as follows:

$$Q(\text{D}, \text{right}) = 16, Q(\text{D}, \text{left}) = 10, Q(\text{E}, \text{right}) = 20, Q(\text{E}, \text{left}) = 15.$$

Assume, at the current iteration of Q-learning, we are at state D and we choose action "moving to right". So, the next state is E and the reward is 0 for state D:

$$s = \text{D},\ a = \text{right},\ s' = \text{E},\ r = 0.$$

Assume $n(\text{D}, \text{right}) = 2$ so far and let the discount factor is $\gamma = 0.9$. in this iteration. By performing this action at this state, the $Q(\text{D}, \text{right})$ is updated from 16 to 17 as:

$$\begin{aligned} Q_n^*(\text{D}, \text{right}) &= Q_{n-1}^*(\text{D}, \text{right}) + \alpha_n\big(r + \gamma \max_{a'} Q^*(\text{E}, a') - Q^*(\text{D}, \text{right})\big) \\ &= 16 + (1/2)(0 + (0.9 \times \max\{20, 15\}) - 16) = 17, \end{aligned}$$

which makes sense because it is better to move right to hopefully end up in state G eventually.

Q-function Approximation

- The Q-function can be approximated by any linear or nonlinear function.
- For example, it can be approximated by a neural network which is a nonlinear function. Let $Q_w(s, a)$ be the approximated Q-function with parameters $\boldsymbol{w}$; for example, it is a neural network with weights $\boldsymbol{w}$.
- Recall Eq. (27):

$$Q_n^*(s, a) = Q_{n-1}^*(s, a) + \alpha_n \big(r + \gamma \max_{a'} Q_{n-1}^*(s', a') - Q_{n-1}^*(s, a)\big),$$

in which the temporal difference is the difference between $r + \gamma \max_{a'} Q_{n-1}^*(s', a')$ and $Q_{n-1}^*(s, a)$. Therefore, $r + \gamma \max_{a'} Q_{\boldsymbol{w}}(s', a')$ can be the target where the Q-function estimate $Q_{\boldsymbol{w}}(s, a)$ needs to get close to.

- As a result, the loss function of the neural network can be the mean squared error between these two terms:

$$\ell(\boldsymbol{w}) = \frac{1}{2}\big(Q_{\boldsymbol{w}}(s, a) - r - \gamma \max_{a'} Q_{\boldsymbol{w}}(s', a')\big)^2. \tag{28}$$

This loss function has a problem. Both $Q_{\boldsymbol{w}}(s, a)$ and $r + \gamma \max_{a'} Q_{\boldsymbol{w}}(s', a')$ contain the weights $\boldsymbol{w}$; therefore, changing the weights $\boldsymbol{w}$ modifies both the function estimate and the target. In this sense, the target becomes a moving target. It has been empirically observed that if the Q-function is linear, minimizing this loss function converges but if it is nonlinear, it may not converge.

Gradient Q-learning

- There are two approaches to resolve the problem of moving target.
- In the first approach, the Q-function in the target can be fixed and gets updated every k iterations. The estimated Q-function, however, gets updated by optimization in every iteration. In practice, this approach can be implemented using two neural networks – Q-value estimate network and target network – are used rather than one network. Let the weights of the Q-value estimate network and the target network be denoted by $\boldsymbol{w}$ and $\bar{\boldsymbol{w}}$, respectively. The loss function becomes:

$$\ell(\boldsymbol{w}) = \frac{1}{2}\left(Q_{\boldsymbol{w}}(s,a) - r - \gamma \max_{a'} Q_{\bar{\boldsymbol{w}}}(s',a')\right)^2. \tag{29}$$

Backpropagation is gradient decent and chain rule. The gradient of this loss function is:

$$\frac{\partial \ell(\boldsymbol{w})}{\partial \boldsymbol{w}} = \left(Q_{\boldsymbol{w}}(s,a) - r - \gamma \max_{a'} Q_{\bar{\boldsymbol{w}}}(s',a')\right) \frac{\partial Q_{\boldsymbol{w}}(s,a)}{\partial \boldsymbol{w}}. \tag{30}$$

Gradient Q-learning

Algorithm Gradient Q-learning

Initialize $\boldsymbol{w}$ and $\bar{\boldsymbol{w}}$
Observe the current state s
while *not converged* **do**
  Select and execute action a
  Observe the new state s'
  $\frac{\partial \ell(\boldsymbol{w})}{\partial \boldsymbol{w}} = \left(Q_{\boldsymbol{w}}(s,a) - r - \gamma \max_{a'} Q_{\bar{\boldsymbol{w}}}(s',a')\right) \frac{\partial Q_{\boldsymbol{w}}(s,a)}{\partial \boldsymbol{w}}$
  $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \frac{\partial \ell(\boldsymbol{w})}{\partial \boldsymbol{w}}$
  $s \leftarrow s'$
  if *it is every k iterations* **then**
    Update $\bar{\boldsymbol{w}}$
Return $Q_{\boldsymbol{w}}(s,a)$

- In this algorithm, $\alpha > 0$ is the learning rate of optimization.
- Updating $\bar{\boldsymbol{w}}$ can be performed either by optimizing $\bar{\boldsymbol{w}}$ in the target neural network or it can be updated by coping the weights of the target network to the Q-value estimate network ($\bar{\boldsymbol{w}} \leftarrow \boldsymbol{w}$) in case the two networks have the same structure.

Experience Replay

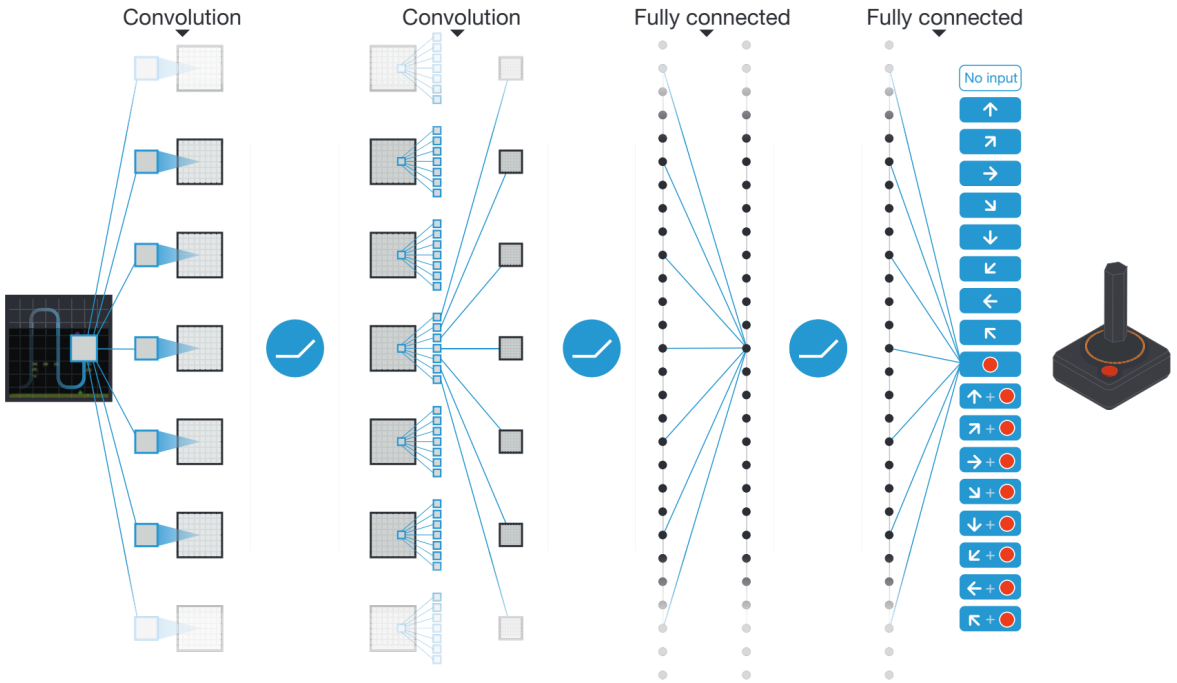
- Another approach for resolving the problem of moving target is experience replay. In this approach, the previous experiences $\{s, a, s', r\}$ are stored in a buffer and a mini-batch of previous experiences is sampled at every iteration of gradient Q-learning.
- This approach uses the same weights and one network for the Q-value estimate function and the target.
- By using the previous experiences, the target function which is optimized does not change dramatically. In other words, this reduces the effect of the moving target.
- Both approaches of using two networks and using experience replay usually work in practice but they do not have any theoretical guarantee for convergence.

Deep Q-Network

- The first paper which implemented gradient Q-learning by a deep neural network was Deep Q-Network (DQN) (2013) [7, 8].
- This paper is also considered as the first algorithm in Deep Reinforcement Learning (DRL).
- DRL is RL where the Q-function, or value-function, or policy function is estimated using a neural network as a estimator function; therefore, DRL is basically not different from RL except that its function estimator is a neural network.

Deep Q-Network

- DQN plays the Atari games in a human-level where the trained network can play even better than humans.
- The input of DQN is the frame of Atari game at the every time slot. DQN is a convolutional neural network approximating the Q-function. The output neurons of the network correspond to the possible actions performable on the joystick of Atari. Softmax activation function is used in the last layer to output the probability of actions and the action with largest probability determines the action at the current time slot.



Credit of image: [8]

Policy Gradient

Policy Gradient

- Q-learning is a model-free value-based method. In Q-learning, the value function or Q-function is explicitly learned and the policy function is implicit.
- Policy gradient, however, is a model-free policy-based algorithm. In policy gradient, the policy function is explicitly learned and the value function or Q-function is implicit.
- Note that both Q-learning and policy gradient pursue the goal of RL which is finding the best map of policy $\pi : s \mapsto a$.
- The policy gradient attempts to learn the policy function explicitly. The policy function can be a deterministic (non-random) function $\pi(s) = a$ or a stochastic (random) function as a conditional probability $\pi(a|s)$.
- Each of these policy function have pros and cons. The deterministic function is usually a discrete function but the stochastic policy is usually a smooth and continuous probabilistic density function which is more suitable for learning. The stochastic policy function is also more useful when the action space is continuous.
- In general, the stochastic policy function can be used for both discrete and continuous actions. For the discrete actions and the continuous actions, softmax function and Gaussian distribution can be useful respectively:

$$\pi_{\boldsymbol{w}}(a|s) = \frac{\exp(f(s, a; \boldsymbol{w}))}{\sum_{\bar{a}} \exp(f(s, \bar{a}; \boldsymbol{w}))}, \tag{31}$$

$$\pi_{\boldsymbol{w}}(a|s) = \mathcal{N}\big(a \,|\, \mu(s; \boldsymbol{w}), \Sigma(s; \boldsymbol{w})\big), \tag{32}$$

where $f(s, a; \boldsymbol{w})$ is estimator function of policy, with parameter $\boldsymbol{w}$, before the softmax function and $\mu(s; \boldsymbol{w})$ and $\Sigma(s; \boldsymbol{w})$ are the Gaussian's mean and covariance functions of states with parameter $\boldsymbol{w}$.

Policy Gradient

- In supervised learning, the desire is to learn $\pi_w(a|s)$ where the training data are the state-action pairs, $\{(s_1, a_1^*), (s_2, a_2^*), \dots\}$, with the best actions as labels for the states. The cross-entropy of softmax function can be minimized or the log-likelihood of Gaussian distribution can be maximized. For example, maximization of log-likelihood in the forms of one-shot optimization and temporal difference are:

$$\begin{aligned} \boldsymbol{w}^* &= \arg\max_{\boldsymbol{w}} \sum_n \log \pi_{\boldsymbol{w}}(a_n^*|s_n), \\ \boldsymbol{w}_{n+1} &\leftarrow \boldsymbol{w}_n + \alpha_n \nabla_{\boldsymbol{w}} \log \pi_{\boldsymbol{w}}(a_n^*|s_n), \end{aligned} \tag{33}$$

  respectively, where $\nabla_{\boldsymbol{w}}(.)$ is the derivative operator with respect to $\boldsymbol{w}$. This equation updates the policy in supervised learning.

- In RL, the desire is to learn $\pi_w(a|s)$ where the training data are the state-action-reward triplets, $\{(s_1, a_1, r_1), (s_2, a_2, r_2), \dots\}$, where the rewards give a hint how good an action is in that state. Policy gradient in RL maximizes the discounted summation of rewards in the forms of one-shot optimization and temporal difference:

$$\begin{aligned} \boldsymbol{w}^* &= \arg\max_{\boldsymbol{w}} \sum_n \gamma^n \mathbb{E}[r_n|s_n, a_n], \\ \boldsymbol{w}_{n+1} &\leftarrow \boldsymbol{w}_n + \alpha_n \gamma^n G_n \nabla_{\boldsymbol{w}} \log \pi_{\boldsymbol{w}}(a_n|s_n), \end{aligned} \tag{34}$$

  respectively, where $\mathbb{E}[.]$ is the expectation operator and G_n is the total reward defined in Eq. (13), i.e., $G_n = \sum_{t=0}^{\infty} \gamma^t r_{n+t}$. This equation updates the policy in RL.

Policy Gradient

- We found:

$$\boldsymbol{w}_{n+1} \leftarrow \boldsymbol{w}_n + \alpha_n \nabla_{\boldsymbol{w}} \log \pi_{\boldsymbol{w}}(a_n^*|s_n),$$
$$\boldsymbol{w}_{n+1} \leftarrow \boldsymbol{w}_n + \alpha_n \gamma^n G_n \nabla_{\boldsymbol{w}} \log \pi_{\boldsymbol{w}}(a_n|s_n).$$

- Comparing Eqs. (33) and (34) shows that, compared to supervised learning, RL scales the gradient by the reward $\gamma^n G_n$. If the reward is large, that direction of gradient is amplified and the algorithm goes in that direction. However, if the reward is small, that direction of gradient is ignored and the algorithm does not go in that direction.

REINFORCE Algorithm

- REINFORCE algorithm is one of the very first algorithms in RL, proposed in 1992 [9].
- This algorithm is a policy gradient algorithm and it uses the temporal update in Eq. (34).
- It plays the game or interacts with the environment for multiple times. Every time that the game is completely played once is called an episode. Every episode has T steps where T might differ in different episodes.
- In this algorithm, $\alpha > 0$ is the learning rate.

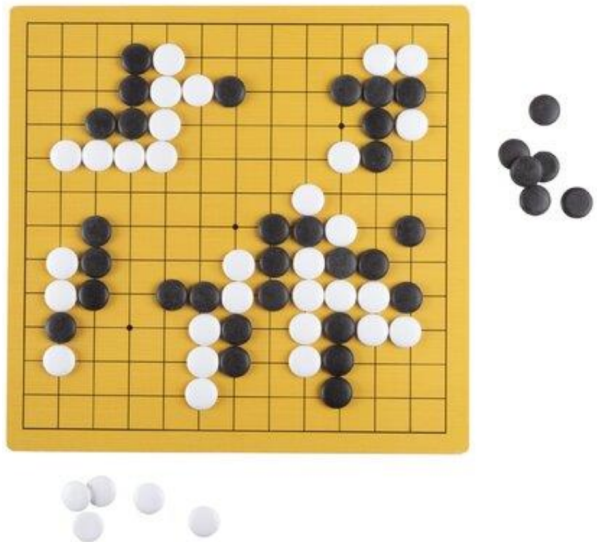
Algorithm REINFORCE algorithm

Initialize $\boldsymbol{w}$
Observe the current state s
for *each episode* **do**
    Generate (play) the episode
    . $\{(s_0, a_0, r_0), (s_1, a_1, r_1), \ldots, (s_T, a_T, r_T)\}$
    for n *from* 0 *to* T **do**
        $G_n \leftarrow \sum_{t=0}^{T-n} \gamma^t r_{n+t}$
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\gamma^n G_n \nabla_{\boldsymbol{w}} \log \pi_{\boldsymbol{w}}(a_n|s_n)$
Return $\pi_{\boldsymbol{w}}(a|s)$

AlphaGo: Game of Go by Reinforcement Learning

AlphaGo: Game of Go by Reinforcement Learning

- Game of Go is an Asian-originated game where two players alternate to place a stone on a vacant (empty) intersection (dot). One player has white stones and one has black stones.
- Connected stones which have no adjacent vacant intersection are removed.
- The player who controls the largest intersections at the end of the game becomes the winner.
- This game is difficult for the computer because the board of game is 19×19 (i.e., 361 intersections) where every intersection can either be vacant or have wight stone or black stone. Therefore, its state space is very huge and has 3^{361} possibilities.
- AlphaGo (2016) [10] uses deep reinforcement learning for playing the game of Go.

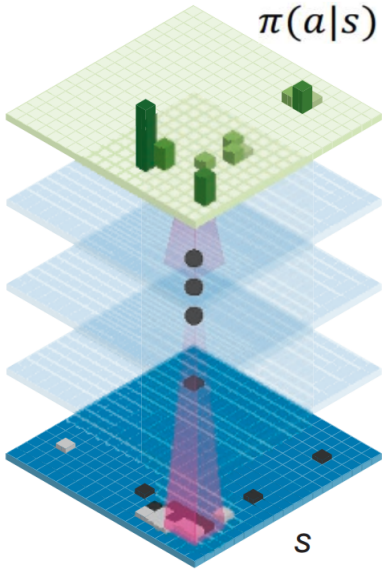


AlphaGo: Game of Go by Reinforcement Learning

- AlphaGo has a policy network for modeling $\pi(a|s)$ where the input of the network is the state s which is the configuration of the board.
- The output of network is the distribution of actions a. The intersection with maximum output probability determines the location on which the stone should be placed.
- They have used supervised learning for the policy network using 30 million board configurations played by the professionals.
- The network maximizes the log-likelihood of policy:

$$\underset{\boldsymbol{w}}{\text{maximize}} \quad \log \pi_{\boldsymbol{w}}(a|s),$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \frac{\partial \log \pi_{\boldsymbol{w}}(a|s)}{\partial \boldsymbol{w}}.$$



AlphaGo: Game of Go by Reinforcement Learning

- After the supervised learning, some policy function, implemented as the policy network, is found. This policy function can become better by policy gradient. In the policy gradient for the policy network, the network plays the game of Go in episodes against its previous version fine-tuned in the previous episode. In every episode, the policy network is trained and fine-tuned further.
- For every episode, the total reward is defined as:

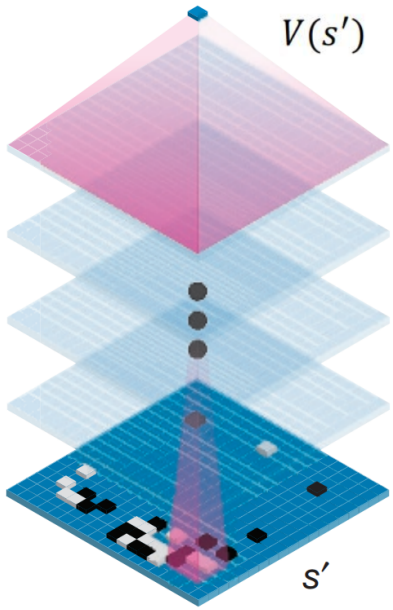
$$G_n = \begin{cases} 1 & \text{if it wins} \\ -1 & \text{if it loses.} \end{cases} \tag{35}$$

- Eq. (34) is used for fine-tuning the policy network in episodes by policy gradient:

$$\boldsymbol{w}_{n+1} \leftarrow \boldsymbol{w}_n + \alpha_n \gamma^n G_n \nabla_{\boldsymbol{w}} \log \pi_{\boldsymbol{w}}(a_n|s_n).$$

AlphaGo: Game of Go by Reinforcement Learning

- AlphaGo has also a value network to predict the value $v(s)$, i.e., to predict who will win the game eventually.
- The input of the value network is the state s which is the configuration of the board.
- The output of the value network is the expected discounted summation of rewards, i.e., $v(s)$.



AlphaGo: Game of Go by Reinforcement Learning

- The value network is trained by gradient value learning. The data for training are the (s, G) pairs where G is defined as in Eq. (35):

$$G = \begin{cases} 1 & \text{if it wins} \\ -1 & \text{if it loses.} \end{cases}$$

- The cost function to minimize by the network is:

$$\underset{\boldsymbol{w}}{\text{minimize}} \quad \frac{1}{2}(v_{\boldsymbol{w}}(s) - G)^2,$$

and the optimization in backpropagation is:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha(v_{\boldsymbol{w}}(s) - G)\frac{\partial v_{\boldsymbol{w}}(s)}{\partial \boldsymbol{w}}.$$

AlphaGo: Game of Go by Reinforcement Learning

- AlphaGo combines the policy network and the value network into a Monte Carlo Tree Search (MCTS) algorithm [11]. In this MCTS, every node of the tree is a state s and every edge is an action.
- Every possible episode of game is one of the trajectories in the tree with its own reward. As this tree becomes very large, it will cause memory error and also searching in the tree becomes hard and time consuming. Therefore, MCTS grows the tree only in the trajectories of most promising states.
- Note that many artificial intelligence algorithms for solving board games, such as AlphaGo for the game of Go [10] and AlphaZero for chess and shogi [12, 13, 14], use MCTS.

Acknowledgment

- Some slides of this slide deck are inspired by teachings of Prof. Ali Ghodsi at the University of Waterloo, Department of Statistics.
- Also watch teachings of Prof. Pascal Poupart at the University of Waterloo, Department of Statistics (available on YouTube).
- A good textbook in reinforcement learning by Richard Sutton: [1]

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] K. Zhang, Z. Yang, and T. Başar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms," *Handbook of reinforcement learning and control*, pp. 321–384, 2021.

[3] B. Ghojogh, F. Karray, and M. Crowley, "Hidden markov model: Tutorial," 2019.

[4] T. M. Moerland, J. Broekens, A. Plaat, C. M. Jonker, *et al.*, "Model-based reinforcement learning: A survey," *Foundations and Trends® in Machine Learning*, vol. 16, no. 1, pp. 1–118, 2023.

[5] S. Çalışır and M. K. Pehlivanoğlu, "Model-free reinforcement learning algorithms: A survey," in *2019 27th signal processing and communications applications conference (SIU)*, pp. 1–4, IEEE, 2019.

[6] B. Ghojogh, H. Nekoei, A. Ghojogh, F. Karray, and M. Crowley, "Sampling algorithms, from survey sampling to Monte Carlo methods: Tutorial and literature review," *arXiv preprint arXiv:2011.00901*, 2020.

[7] 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.

References (cont.)

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[9] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, pp. 229–256, 1992.

[10] 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*, vol. 529, no. 7587, pp. 484–489, 2016.

[11] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[12] 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.

[13] 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*, vol. 362, no. 6419, pp. 1140–1144, 2018.

References (cont.)

[14] 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*, vol. 588, no. 7839, pp. 604–609, 2020.