

Reinforcement Learning (AI644)

Prepared by Dr. Ashraf Elsayed

Lecture 9



Reinforcement Learning

Lecture 9

Prepared by
Dr. Ashraf Elsayed

Expected SARSA & Planning

The objectives of this lecture is to:

- Describe the Expected SARSA
- Describe the Double Q-learning algorithm
- Understand and describe n-step TD Learning
- Describe what a model is and how they can be used
- Classify models as distribution models or sample models
- Explain how planning is used to improve policies

Expected Sarsa

- **Expected SARSA**, estimates the Q-value by taking a weighted average of the Q-values of all the possible actions in the next state.

The Bellman equation for action-values

$$q_{\pi}(s, a) = \sum_{s', r} p(s', r | s, a) \left(r + \gamma \sum_{a'} \pi(a' | s') q_{\pi}(s', a') \right)$$


Sarsa: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$

$$\sum_{a'} \pi(a' | S_{t+1}) Q(S_{t+1}, a')$$


Expected Sarsa

The Expected Sarsa algorithm

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \sum_{a'} \pi(a' | S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

$$Q(S_{t+1}, a') =$$

0.0	-1.0	2.0	1.0
-----	------	-----	-----

$$\pi(a' | S_{t+1}) =$$

0.1	0.1	0.7	0.1
-----	-----	-----	-----

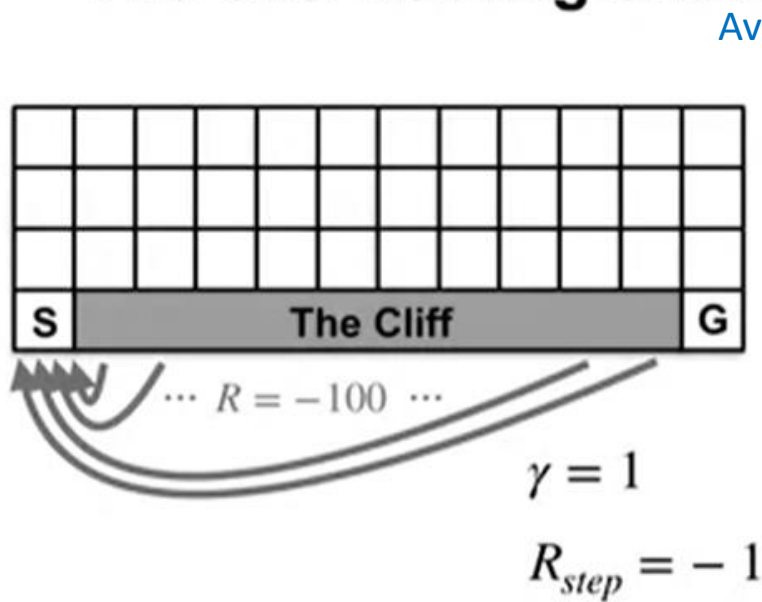
$$\sum_{a'} \pi(a' | S_{t+1}) Q(S_{t+1}, a') =$$

0.0	-	0.1	+	1.4	+	0.1	=	1.4
-----	---	-----	---	-----	---	-----	---	-----

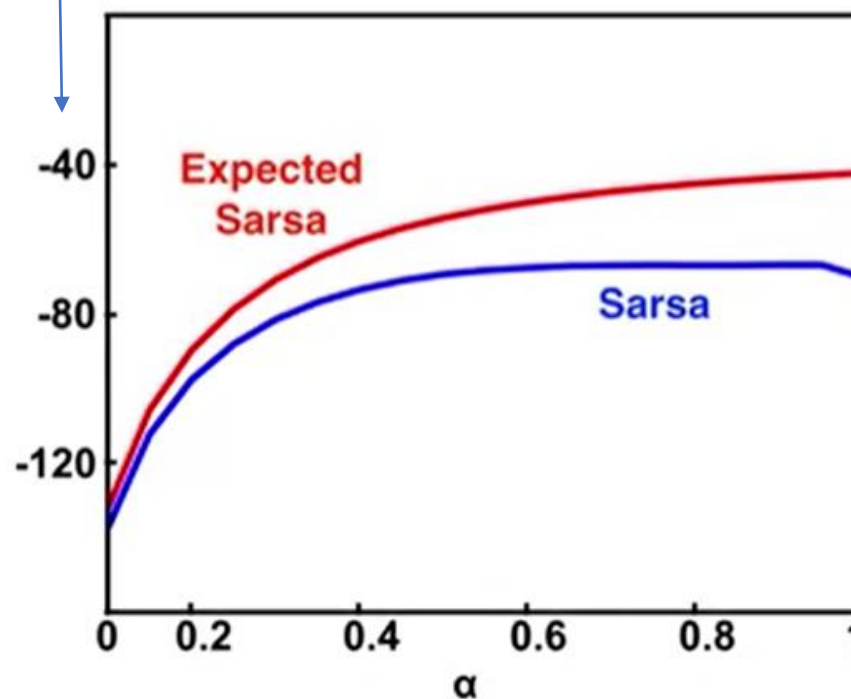


Example: Expected Sarsa in Cliff World

The cliff walking environment



Average return per episode



There exists a modification of the SARSA algorithm called **Expected SARSA**.

Assume π is the current update of the SARSA approach.

Then, instead of updating Q_t by sampling a new action A_{t+1} and consider $Q_t(A_{t+1}, S_{t+1})$ in the target, we could consider a weighed average across all possible actions at time $t + 1$.

The update in expected SARSA is given by

$$\begin{aligned} Q_{t+1}(S_t, A_t) &= Q_t(S_t, A_t) + \alpha [R_{t+1} + \gamma \mathbb{E}_{\pi_t} [Q_t(S_{t+1}, A_{t+1}) | S_{t+1}] - Q_t(S_t, A_t)] \\ &= Q_t(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi_t(a | S_{t+1}) Q_t(S_{t+1}, a) - Q_t(S_t, A_t) \right] \end{aligned}$$

when $S_t \neq s^*$, where π_t is the current estimate of the policy that is being optimized.

Thus, instead of sampling an action A_{t+1} as in SARSA, we take the expected value of $Q(S_{t+1}, A_{t+1})$ across all time $t + 1$ -actions using policy π_t .

This could also be seen as a modification of Q-learning where the maximum across actions a is replaced by a probability weighted averaged.

Pros and cons of expected SARSA

Expected SARSA involves more computations than SARSA to perform a single update;

- It requires summing over the set of all actions instead of considering a single action.
- This works best the action space is small.

However, expected SARSA removes a portion of the variance due to remove randomness around the choice of A_{t+1} .

- For this reason, empirical tests have revealed Expected SARSA converges quicker than SARSA in many applications, as stated in the textbook.

Expected SARSA in both on-policy or off-policy contexts

Expected Sarsa can used on-policy, by using the same π_t to generate action selection (episode generation) and the probability weighted summation.

However, Expected Sarsa can also be used as an off-policy method by

- using an exploratory policy to select actions to generate episodes,
- using the estimated greedy policy for performing the sample average.

Off-Policy Expected Sarsa

Expected Sarsa:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \sum_{a'} \pi(a' | S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

 $A_{t+1} \sim \pi(a' | S_{t+1}) \neq b(a' | S_{t+1})$

Q-learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

Expected Sarsa is Off-Policy

Greedy Expected Sarsa

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \sum_{a'} \pi(a' | S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

$\pi = \pi^*$?

$Q(S_{t+1}, a') =$	0.0	-0.8	2.1	1.2
$\pi(a' S_{t+1}) =$	0.0	0.0	1.0	0.0

$$\sum_{a'} \pi(a' | S_{t+1}) Q(S_{t+1}, a') = \max_{a'} Q(S_{t+1}, a')$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

= Q-learning

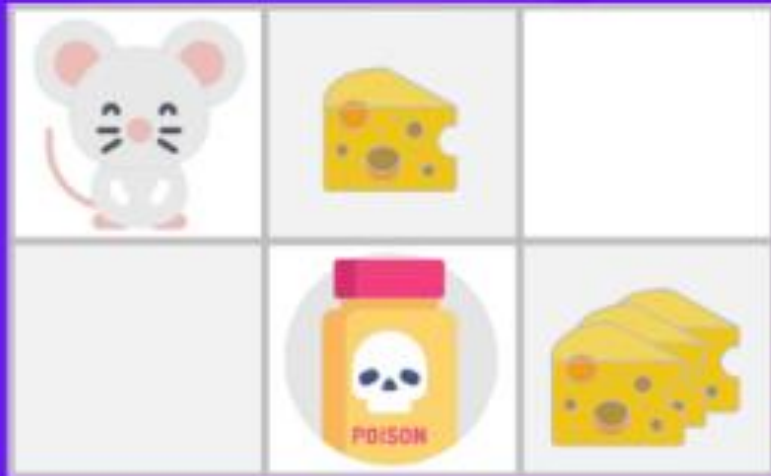
Q-learning is a special case of the Expected Sarsa

A Q-Learning example



- You're a mouse in this tiny maze. You always start at the **same starting point**.
- The goal is to eat the **big pile of cheese** at the **bottom right-hand corner** and avoid the poison. After all, who doesn't like cheese?
- The episode ends if we eat the poison, eat the **big pile of cheese** or if we spent **more than five steps**.
- The learning rate is 0.1
- The gamma (discount rate) is 0.99

Example



- You always start at the **same starting point**.
- The goal: eat the **big pile of cheese** (at the bottom right-hand corner) and **avoid the poison**.
- The episode ends if we eat the poison, eat the big pile of cheese or if we spent more than 5 steps.
- Learning rate = 0.1
- Gamma = 0.99

Example

- The reward function:
 - 0: Going to a state with no cheese in it.
 - +1: Going to a state with a small cheese in it.
 - +10: Going to the state with the big pile of cheese.
 - -10: Going to the state with the poison and thus die.



Example, Step 1

Initialize Q arbitrarily (e.g., $Q(s, a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\text{terminal-state}, \cdot) = 0$)

				
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

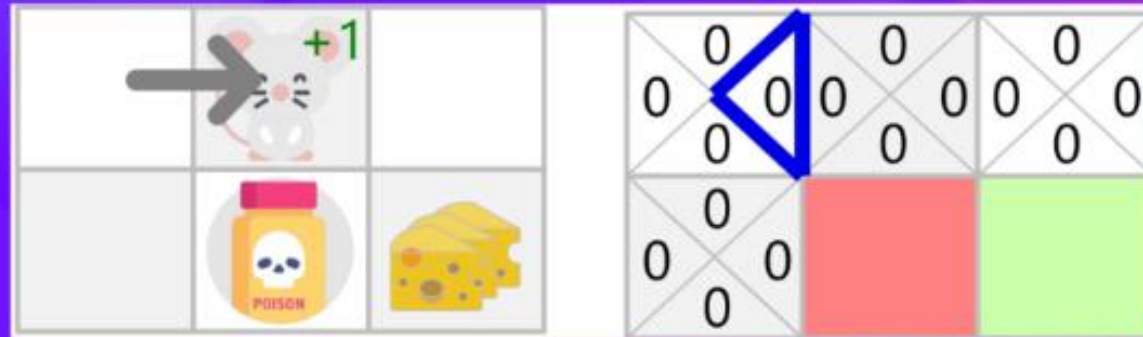
We initialize the Q-Table

Step 2: Choose action using Epsilon Greedy Strategy

Because epsilon is big = 1.0, I take a random action, in this case, I go right.

Example, Step 2

Choose action A_t using policy derived from Q (e.g., ϵ -greedy)



We took a random action (exploration)

Step 3: Perform action A_t , gets R_{t+1} and S_{t+1}

By going right, I've got a small cheese, so $R_{t+1} = 1$, and I'm in a new state.

Example, Step 3

Take action A_t and observe R_{t+1}, S_{t+1}



Step 4: Update $Q(S_t, A_t)$

We can now update $Q(S_t, A_t)$ using our formula.

Example, Step 4

$$\underline{Q(S_t, A_t)} \leftarrow \underline{Q(S_t, A_t)} + \underline{\alpha} [\underline{R_{t+1}} + \underline{\gamma \max_a Q(S_{t+1}, a)} - \underline{Q(S_t, A_t)}]$$

New
Q-value
estimation

Former
Q-value
estimation

Learning
Rate

Immediate
Reward

Discounted Estimate
optimal Q-value
of next state

Former
Q-value
estimation

TD Target

TD Error

Update our Q-value estimation

Example, Step 4

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

$$Q(\text{Initial state, Right}) = 0 + 0.1 * [1 + 0.99 * 0 - 0]$$

$$Q(\text{Initial state, Right}) = 0.1$$

	←	→	↑	↓
	0	0.1	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

Training timestep 2:

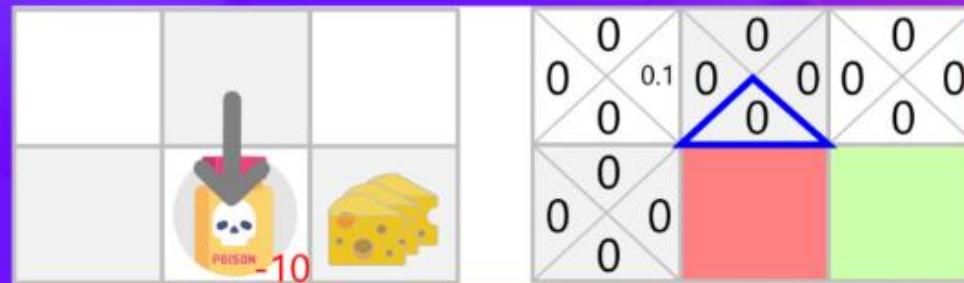
Step 2: Choose action using Epsilon Greedy Strategy

I take a **random** action again, since **epsilon** is big 0.99 (since we decay it a little bit because as the training progress, we want less and less exploration).

I took action down. Not a good action since it leads me to the poison.

Example, Step 2

Choose action A_t using policy derived from Q (e.g., ϵ -greedy)



We took a random action (exploration)

Step 3: Perform action A_t , gets R_{t+1} and S_{t+1}

Because I go to the poison state, I get $R_{t+1} = -10$, and I die.

Example, Step 3

Take action A_t and observe R_{t+1}, S_{t+1}



Step 4: Update $Q(S_t, A_t)$

Example, Step 4

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

$$Q(\text{State 2, Down}) = 0 + 0.1 * [-10 + 0.99 * 0 - 0]$$

$$Q(\text{State 2, Down}) = -1$$

	←	→	↑	↓
👤	0	0.1	0	0
💰	0	0	0	-1
□	0	0	0	0
■	0	0	0	0
👤	0	0	0	0
💰	0	0	0	0

Because we're dead, we start a new episode. But what we see here is that **with two explorations steps, my agent became smarter.**

As we continue exploring and exploiting the environment and updating Q-values using TD target, Q-Table will give us better and better approximations. And thus, at the end of the training, we'll get an estimate of the optimal Q-Function.

The previous temporal difference algorithms seen (SARSA, Q-learning, expected SARSA) **provide asymptotic convergence** of the action-value estimate Q to its optimal value q_* .



However, in practice the number of episodes considered to perform estimates is finite, and temporal difference methods are typically biased for finite samples.

As we will see, the **bias** is caused by the following inequality.

Assume Q_t is an unbiased estimator of q_* . Then,

$$\mathbb{E} \left[\max_a Q_t(a, s) \right] \geq \max_a \mathbb{E} [Q_t(a, s)] = \max_a q_*(a, s) = v_*(s)$$

which is true since $\max_{\tilde{a}} Q_t(\tilde{a}, s) \geq Q_t(a, s)$ for all a .

Maximization Bias in TD

Therefore, in Q-learning, the target

$$R_{t+1} + \gamma \max_a Q_t(S_{t+1}, a)$$

is a biased estimate of $v_*(S_t)$.

- Thus all updates of $Q_t(S_t, A_t)$ end up including a bias.

This bias comes from the fact that the same sample is used to both determine the optimal action and the associated value.

- Therefore a similar bias appears in the SARSA algorithm.

What is the problem with Q-learning?

The problem with Q-learning is that it overestimates the value of certain actions. This is due to the max operator, which uses the same values to select and evaluate an action. This can lead to overoptimistic value estimates, resulting in a sub-optimal policy.

For example, let's say we have two actions in a certain state, A and B. A has a Q-value of 10, and B has a Q-value of 9. According to Q-learning, A is the optimal action to take in that state. However, what if the true Q-values for A and B are 4 and 12, respectively? Q-learning would still select A as the optimal action, even though it is sub-optimal.

The problem is named **maximization bias** problem.

Double Q-learning

In Q-learning with double learning, at every time step, a “coin” toss would be made to determine which of $Q^{(1)}$ or $Q^{(2)}$ is updated.

If the coin gives head, then the update formula would be

$$Q_{t+1}^{(1)}(S_t, A_t) := Q_t^{(1)}(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_t^{(2)} \left(S_{t+1}, \arg \max_a Q_t^{(1)}(S_{t+1}, a) \right) - Q_t^{(1)}(S_t, A_t) \right],$$

$$Q_{t+1}^{(2)}(S_t, A_t) := Q_t^{(2)}(S_t, A_t).$$

If the coin is tail, then $Q^{(1)}$ or $Q^{(2)}$ would be interchanged in the previous formula.

Note that there also exists analogous versions of SARSA and expected SARSA using double learning.

- Train 2 action-value functions, Q_1 and Q_2
- Do Q-learning on both, but
 - never on the same time steps (Q_1 and Q_2 are indep.)
 - pick Q_1 or Q_2 at random to be updated on each step
- If updating Q_1 , use Q_2 for the value of the next state:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left(R_{t+1} + Q_2(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right)$$

- Action selections are (say) ϵ -greedy with respect to the sum of Q_1 and Q_2

Initialize $Q_1(s, a)$ and $Q_2(s, a)$, $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily

Initialize $Q_1(\text{terminal-state}, \cdot) = Q_2(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

Initialize S

Repeat (for each step of episode):

Choose A from S using policy derived from Q_1 and Q_2 (e.g., ϵ -greedy in $Q_1 + Q_2$)

Take action A , observe R, S'

With 0.5 probability:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha \left(R + \gamma Q_2(S', \arg \max_a Q_1(S', a)) - Q_1(S, A) \right)$$

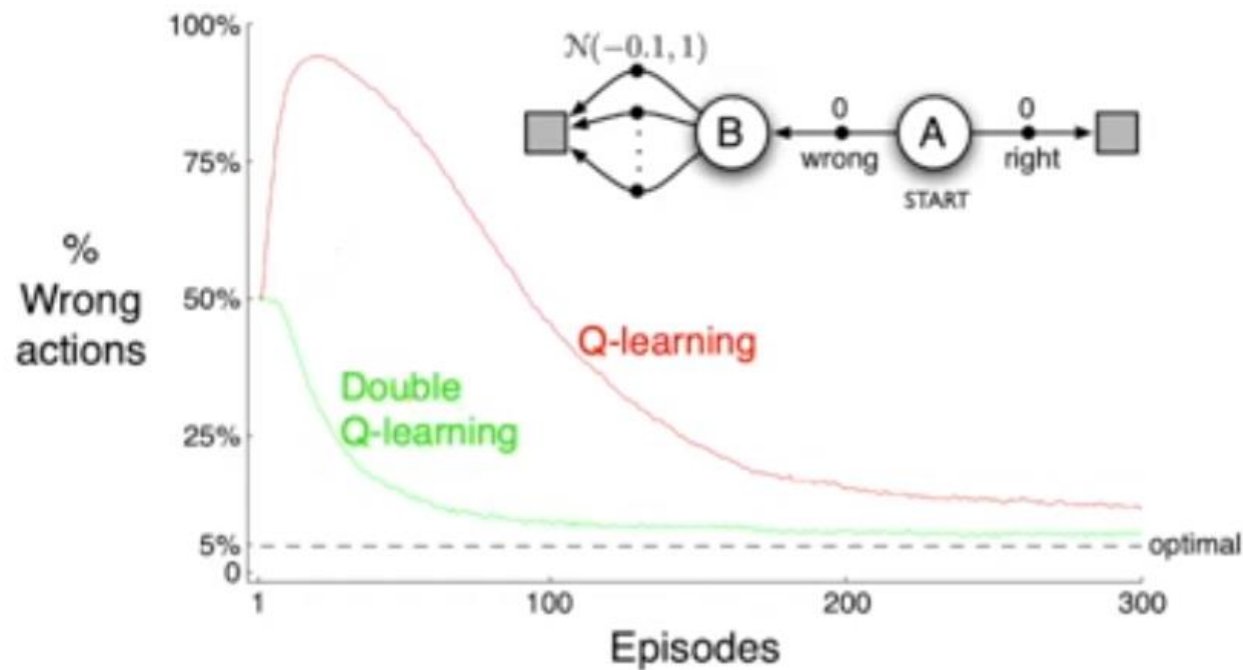
else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha \left(R + \gamma Q_1(S', \arg \max_a Q_2(S', a)) - Q_2(S, A) \right)$$

$S \leftarrow S'$;

until S is terminal

Example of Maximization Bias



Double Q-learning:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2(S_{t+1}, \arg \max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right]$$

Unbiasedness of double learning estimator

This approach corrects for the bias caused by using the same data to determine the optimal action and its value.

For instance, in the previous $N(0, 1)$ k -armed bandit problem, due to the independence of estimates $Q_t^{(1)}$ and $Q_t^{(2)}$,

$$\begin{aligned}
 \mathbb{E}[Q_t(A_t^*)] &= \sum_a \mathbb{P}[A_t^* = a] \mathbb{E}[Q_t^{(2)}(A_t^*) | A_t^* = a] \\
 &= \sum_a \mathbb{P}\left[\arg \max_{\tilde{a}} Q_t^{(1)}(\tilde{a}) = a\right] \mathbb{E}[Q_t^{(2)}(a) | A_t^* = a] \\
 &= \sum_{j=1, \dots, k} \frac{1}{k} \mathbb{E}[Q_t^{(2)}(a)] \quad (\text{from independence}) \\
 &= \sum_{j=1, \dots, k} \frac{1}{k} 0 = 0. \quad \Rightarrow \text{unbiased estimator}
 \end{aligned}$$

Afterstates

We will now present a trick to accelerate some computations for some specific problems in practice.

- Relies on quantities called **afterstates** of **post-decision state variables**.

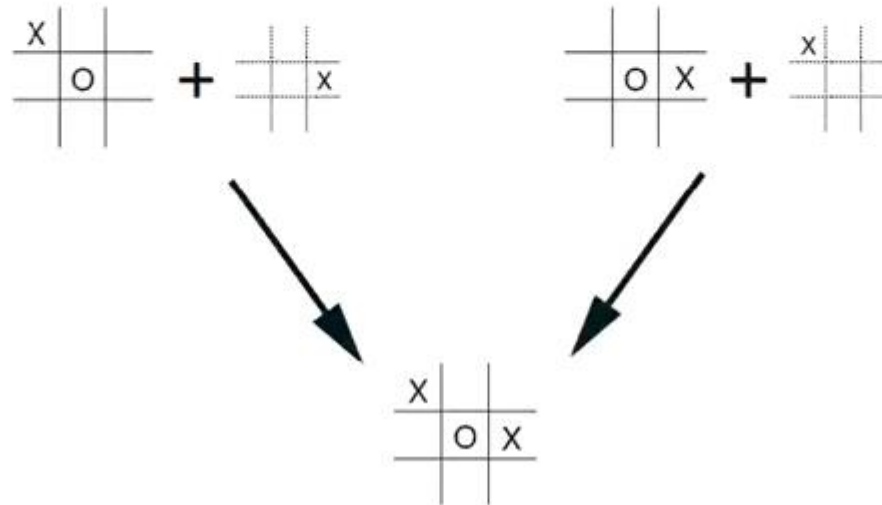
The idea is that for some problems many state-action pairs lead to identical situations.

Thus the action-value function for each of such pairs should be identical and not be evaluated separately.

Tic-tac-toe afterstates

For instance, consider the following state-action pair in tic-tac-toe, where

- the state (position of x's and o's) is shown on the left,
- the action (next x placed) is shown on the right.
- The resulting positioning (called afterstate) is shown on the bottom



Tic-tac-toe afterstates

We saw two pairs of state-action, denoted by (s_1, a_1) and (s_2, a_2) , which lead to the same positioning after the action.

- This implies $q_*(s_1, a_1) = q_*(s_2, a_2)$.⁴

In this case, we can define the afterstate X_t as the location of x's and o's after state S_t and action A_t . It can be obtained through some mapping ϕ

$$X_t = \phi(S_t, A_t).$$

Furthermore, there must exist an alternative function $\tilde{q}_*$ such that

$$q_*(S_t, A_t) = \tilde{q}_*(\phi(S_t, A_t)) = \tilde{q}_*(X_t)$$

i.e. the value of a state-action is the value of its corresponding afterstate.

⁴assuming the opponent follows a Markov playing strategy.

Afterstate value function

In this case, if the number of afterstates is smaller than the number of state-action combinations, learning is faster because there are less values to learn.

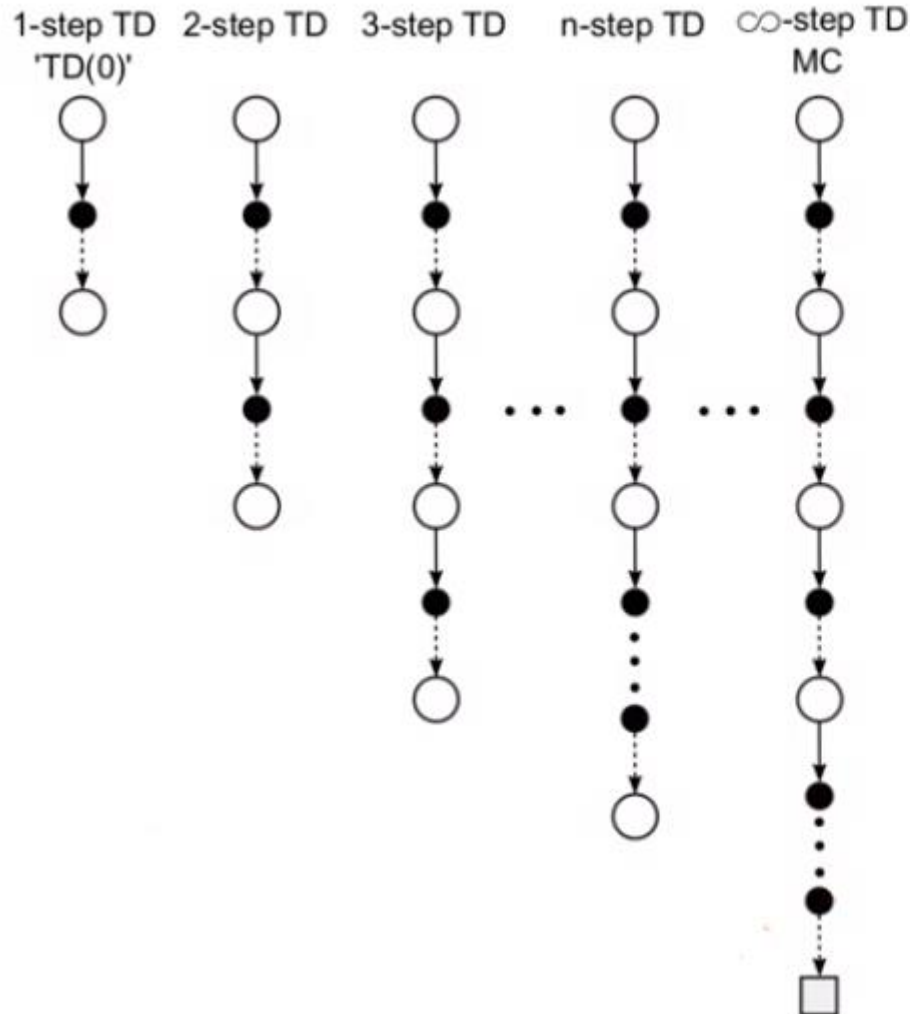
In this case, the DP or reinforcement learning algorithm used should instead provide an estimator $\tilde{Q}_t(x)$ the value of an afterstate $\tilde{q}_*(x)$.

- This can be done through methods analogous to these seen previously.

Then, the greedy action in a state s can be identified through

$$A_t^*(s) := \arg \max_a \tilde{Q}_t(\phi(s, a)).$$

N- Step TD



- ▶ n -step update: consider n rewards plus estimated value n -steps later (bootstrapping).
- ▶ Consequence: Estimate update is available only after an n -step delay.
- ▶ TD(0) and MC are special cases included in n -step prediction.

Recap the **update targets** for the incremental prediction methods

- ▶ Monte Carlo: Builds on the complete sampled return series

$$g_{k:T} = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \cdots + \gamma^{T-k-1} r_T.$$

- ▶ $g_{k:T}$ denotes that all steps until termination at T are considered to derive an estimate target addressing step k .
- ▶ TD(0): Utilizes a one-step bootstrapped return

$$g_{k:k+1} = r_{k+1} + \gamma \hat{v}_k(x_{k+1}).$$

- ▶ For TD(0), $g_{k:k+1}$ highlights that only one future sampled reward step is considered before bootstrapping.
- ▶ $\hat{v}_k$ is an estimate of v_π at time step k .

n -step state-value prediction target

Now, the target is generalized to an arbitrary n -step target:

$$g_{k:k+n} = r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{v}_{k+n-1}(x_{k+n}). \quad (6.3)$$

- ▶ Approximation of full return series truncated after n -steps.
- ▶ If $k + n \geq T$ (i.e. n -step prediction exceeds termination lookahead), then all missing terms are considered zero.

n -step TD

The state-value estimate using the n -step return approximation is

$$\hat{v}_{k+n}(x_k) = \hat{v}_{k+n-1}(x_k) + \alpha [g_{k:k+n} - \hat{v}_{k+n-1}(x_k)], \quad 0 \leq k < T. \quad (6.4)$$

- ▶ Delay of n -steps before $\hat{v}(x)$ is updated.
- ▶ Additional auxiliary update steps required at the end of each episode.

Algorithmic Implementation: n -step TD Prediction

```

input: a policy  $\pi$  to be evaluated
parameter: step size  $\alpha \in (0, 1]$ , prediction steps  $n \in \mathbb{Z}^+$ 
init:  $\hat{v}(x) \forall x \in \mathcal{X}$  arbitrary except  $v_0(x) = 0$  if  $x$  is terminal
for  $j = 1, \dots, J$  episodes do
    initialize and store  $x_0$ ;
     $T \leftarrow \infty$ ;
    repeat  $k = 0, 1, 2, \dots$ 
        if  $k < T$  then
            take action from  $\pi(x_k)$ , observe and store  $x_{k+1}$  and  $r_{k+1}$ ;
            if  $x_{k+1}$  is terminal:  $T \leftarrow k + 1$ ;
         $\tau \leftarrow k - n + 1$  ( $\tau$  time index for estimate update);
        if  $\tau \geq 0$  then
             $g \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} r_i$ ;
            if  $\tau + n < T$ :  $g \leftarrow g + \gamma^n \hat{v}(x_{\tau+n})$ ;
             $\hat{v}(x_\tau) \leftarrow \hat{v}(x_\tau) + \alpha [g - \hat{v}(x_\tau)]$ ;
    until  $\tau = T - 1$ ;
  
```

TD(λ)

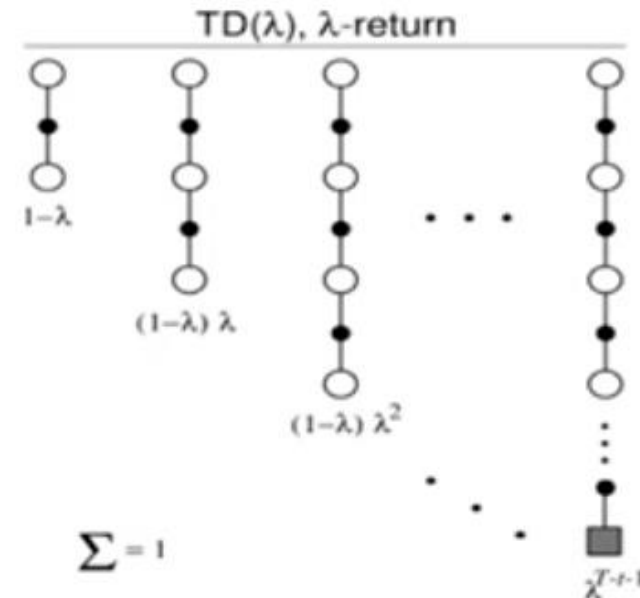
Instead of using n -step backups, we can consider an average of n -step returns.

Example: $G_t^{avg} = \frac{1}{2} G_t^{(2)} + \frac{1}{2} G_t^{(4)}$

In TD(λ), the average contains all the n -step backups each weighted proportional to λ^{n-1} , where $0 \leq \lambda \leq 1$.

λ -return:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$



$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t,$$

From the formula, we can see that when $\lambda = 1$, only the last term is kept and this is essentially Monte Carlo method, as the state, action process goes all the way to the end, when $\lambda = 0$, the term reduces to $G_{t:t+1}$, which is 1-step TD method, and for $0 < \lambda < 1$, the method becomes a mixed of weighted TD(n) method.

- When $\lambda = 0$, only current state is updated

$$E_t(s) = \mathbf{1}(S_t = s)$$

$$V(s) \leftarrow V(s) + \alpha \delta_t E_t(s)$$

- This is exactly equivalent to TD(0) update

$$V(S_t) \leftarrow V(S_t) + \alpha \delta_t$$

TD(λ) and MC:

- When $\lambda = 1$, credit is deferred until end of episode
- Consider episodic environments with offline updates
- Over the course of an episode, total update for TD(1) is the same as total update for MC

Semi-gradient TD(λ) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated

Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$

Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$

Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:

 Initialize S

$\mathbf{z} \leftarrow \mathbf{0}$

(a d -dimensional vector)

 Loop for each step of episode:

 | Choose $A \sim \pi(\cdot | S)$

 | Take action A , observe R, S'

 | $\mathbf{z} \leftarrow \gamma \lambda \mathbf{z} + \nabla \hat{v}(S, \mathbf{w})$

 | $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$

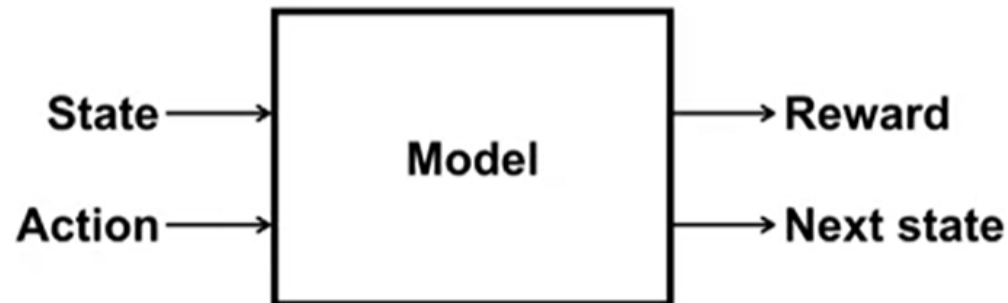
 | $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \mathbf{z}$

 | $S \leftarrow S'$

 until S' is terminal

What is a Model?

- **Models** are used to store knowledge about the dynamics.
- From a particular state in action, the model should produce a possible next state and reward.
- This allows us to see an outcome of an action without having to actually take it.
- Models store knowledge about the transition and reward dynamics.



Maze Example

3	r	r	r	+1
2	u		u	-1
1	u			
	1	2	3	4

$$\gamma = 1$$

Reward is -0.04 for non-terminal states

$(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (4,3)_{,1}$
 $(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (3,2) \rightarrow (3,3) \rightarrow (4,3)_{,1}$
 $(1,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (3,2) \rightarrow (4,2)_{,1}$

$$P((2,3) | (1,3), r) = 2/3$$

$$P((1,2) | (1,3), r) = 1/3$$

Use this information in

$$V^*(s) = \max_a R(s, a) + \gamma \sum_{s'} \Pr(s' | s, a) V^*(s')$$

Maze Example

3	r	r	r	+1
2	u		u	-1
1	u			
	1	2	3	4

$$\gamma = 1$$

Reward is -0.04 for non-terminal states

We need to learn all the transition probabilities!

$(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (4,3)_{+1}$
 $(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (3,2) \rightarrow (3,3) \rightarrow (4,3)_{+1}$
 $(1,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (3,2) \rightarrow (4,2)_{+1}$

$$P((2,3)|(1,3),r) = 2/3$$

$$P((1,2)|(1,3),r) = 1/3$$

Use this information in

$$V^*(s) = \max_a R(s, a) + \gamma \sum_{s'} \Pr(s'|s, a) V^*(s')$$

Model-based RL

- Idea: at each step
 - Execute action
 - Observe resulting state and reward
 - Update transition and/or reward model
 - Update policy and/or value function

Model-based RL (with Value Iteration)

ModelBasedRL(s)

Repeat

Select and execute a

Observe s' and r

Update counts: $n(s, a) \leftarrow n(s, a) + 1$,
 $n(s, a, s') \leftarrow n(s, a, s') + 1$

Update transition: $\Pr(s'|s, a) \leftarrow \frac{n(s, a, s')}{n(s, a)} \forall s'$

Update reward: $R(s, a) \leftarrow \frac{r + (n(s, a) - 1)R(s, a)}{n(s, a)}$

Solve: $V^*(s) = \max_a R(s, a) + \gamma \sum_{s'} \Pr(s'|s, a) V^*(s') \forall s$

$s \leftarrow s'$

Until convergence of V^*

Return V^*

Model-based RL (with Q-learning)

ModelBasedRL(s)

Repeat

Select and execute a , observe s' and r

Update transition: $w_T \leftarrow w_T - \alpha_T (T(s, a) - s') \nabla_{w_T} T(s, a)$

Update reward: $w_R \leftarrow w_R - \alpha_R (R(s, a) - r) \nabla_{w_R} R(s, a)$

Repeat a few times:

sample $\hat{s}, \hat{a}$ arbitrarily

$\delta \leftarrow R(\hat{s}, \hat{a}) + \gamma \max_{\hat{a}'} Q(T(\hat{s}, \hat{a}), \hat{a}') - Q(\hat{s}, \hat{a})$

Update Q : $w_Q \leftarrow w_Q - \alpha_Q \delta \nabla_{w_Q} Q(\hat{s}, \hat{a})$

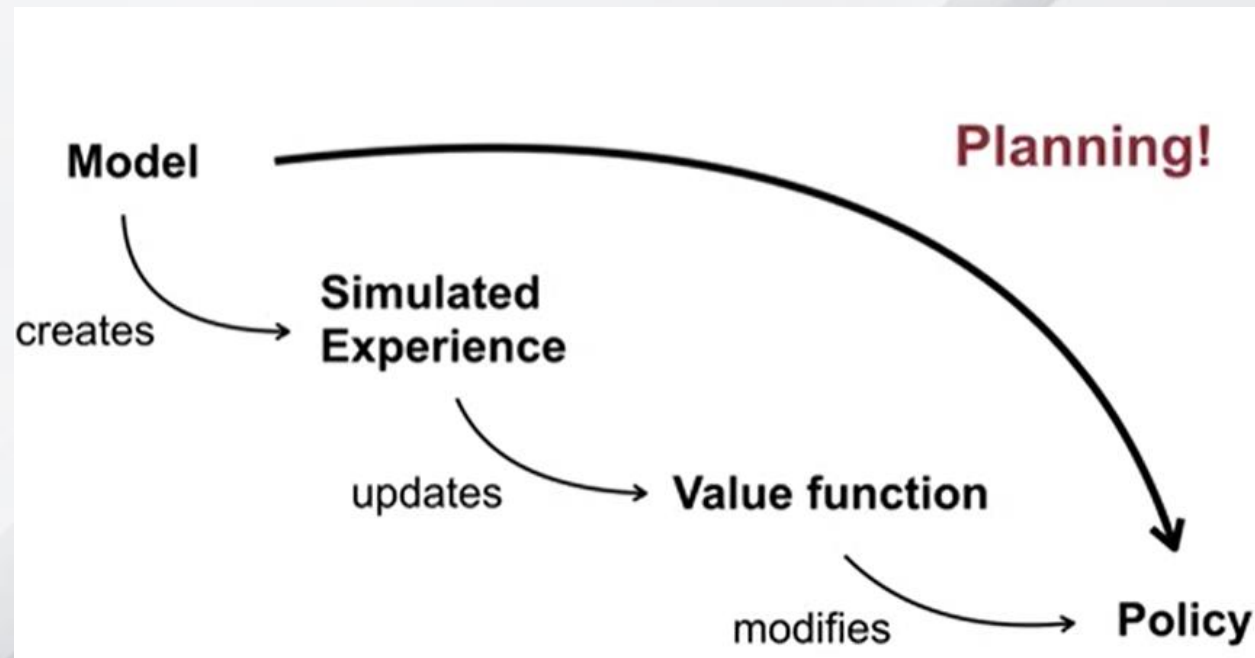
$s \leftarrow s'$

Until convergence of Q

Return Q

Models for Planning

- A model allows for *planning*.
- **Planning** refers to the process of using a model to improve a policy.
- One way to plan with a model is to use **simulated experience** and perform value function updates as if those experiences happened.

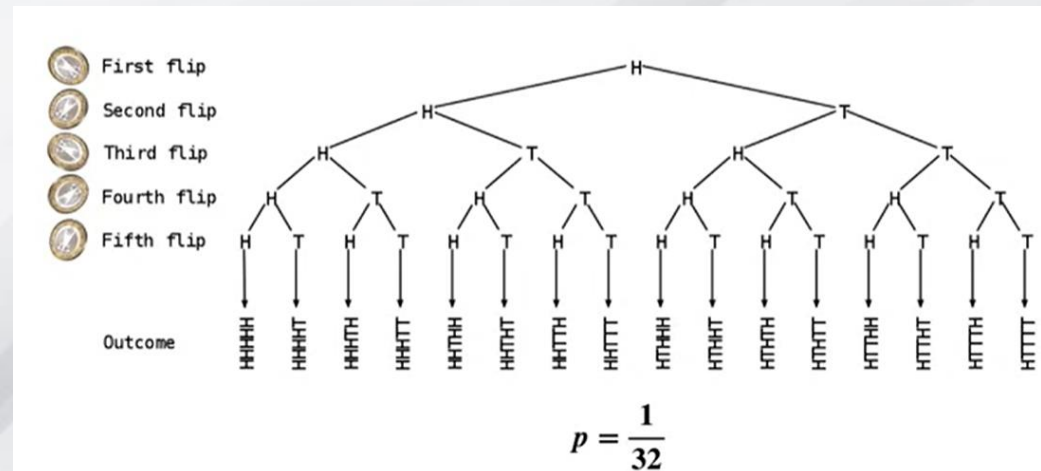


Types of Models

- **Sample models** produces an actual outcome drawn from some underlying probabilities. For example, a sample model for flipping a coin can generate a random sequence of heads and tails.

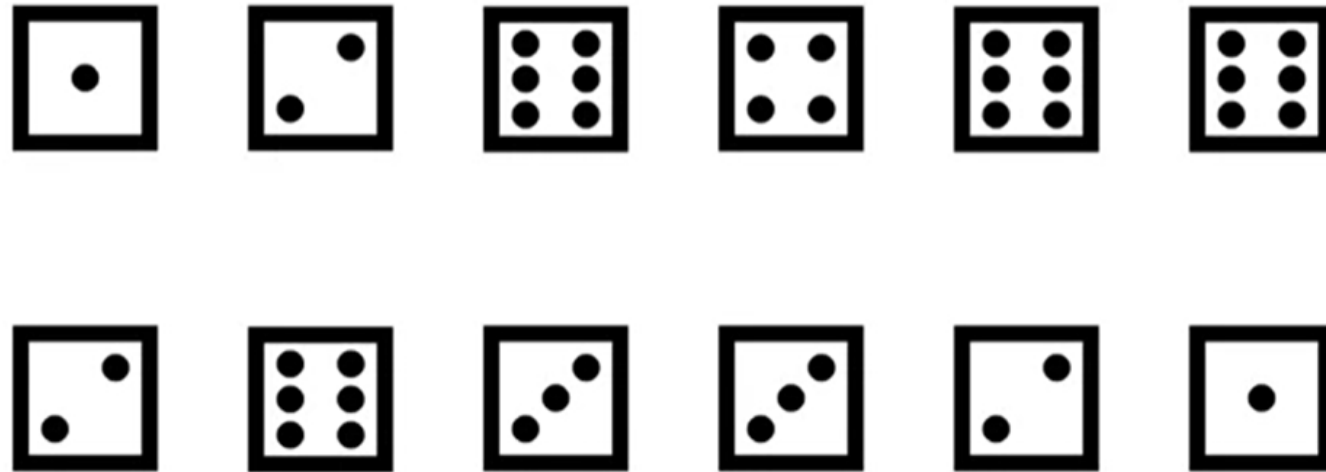


- **Distribution models** specifies the likelihood or probability of every outcome. In the coin flipping example, it would say that heads could occur 50 percent of the time and that tails could occur 50 percent of the time.



Example: Sample Models

It's easy to roll 12 dice



Example: Distribution Models

Modelling the distribution of 12 dice

	6 outcomes
	36 outcomes
	216 outcomes
	1 296 outcomes
	7 776 outcomes
	46 656 outcomes

With 12 dice...



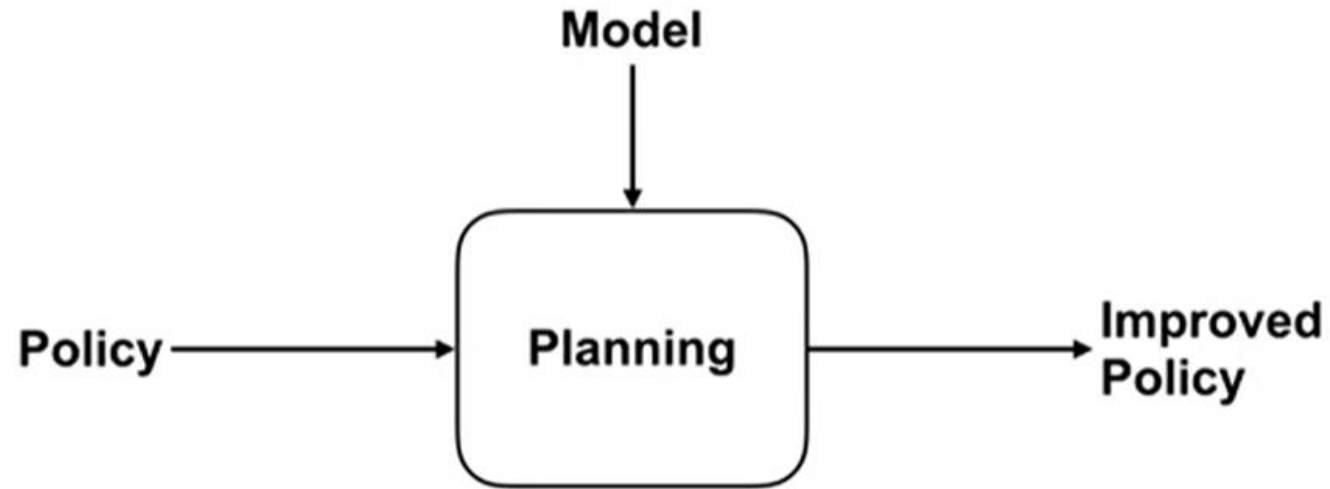
2 176 782 336
outcomes!

Sample Models vs Distribution Models

- **Sample models** can be computationally inexpensive because random outcomes can be produced according to a set of rules.
- **Sample models** require less memory.
- **Sample models** can only approximate this expected outcome by averaging many samples together.
- **Distribution models** contain more information, but can be difficult to specify and can become very large.
- **Distribution models** can be used to compute the exact expected outcome.
- **Distribution models** can be used to assess risk.

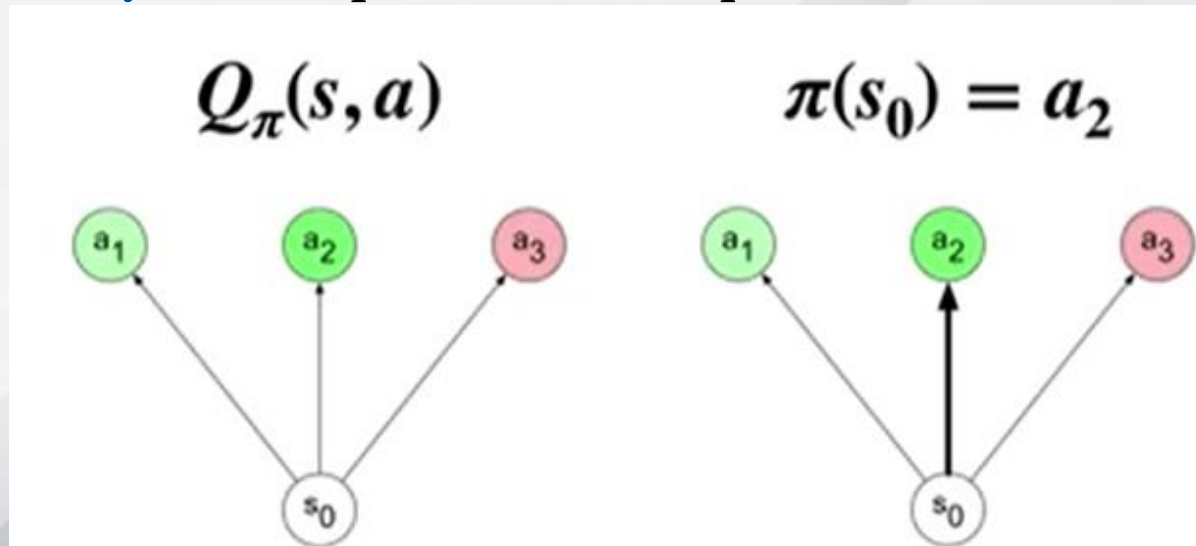
Planning with simulated experience

- One can leverage a model to better inform decision-making without having to interact with the world, we call this process **planning with model experience**.
- Planning is a process which **takes a model as input and produces an improved policy**.



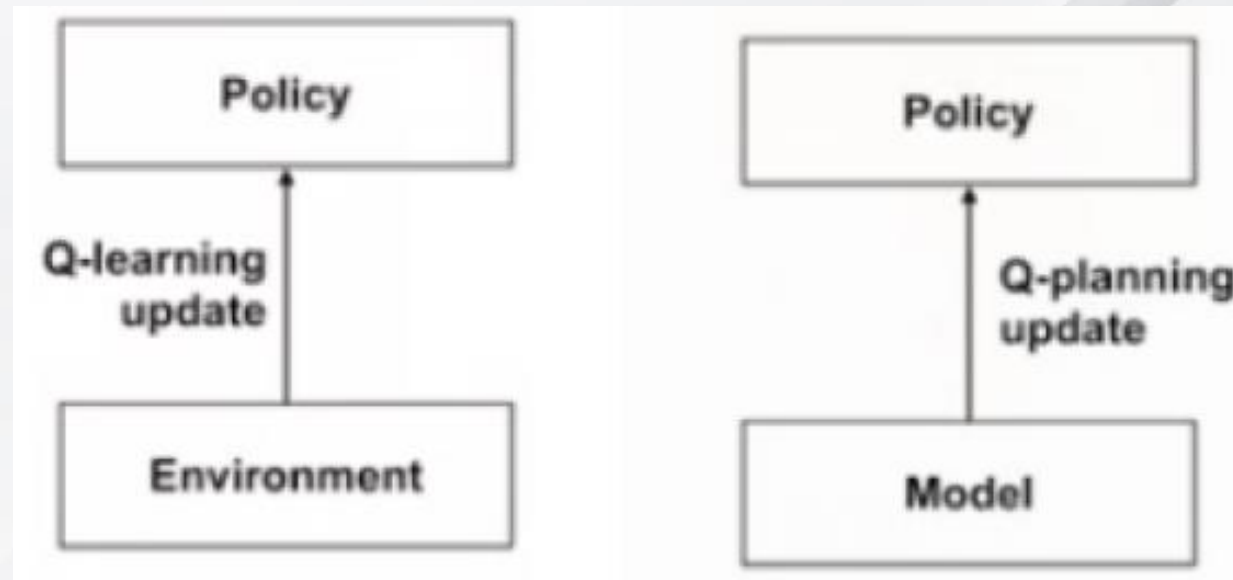
Planning improves policies

- One possible approach to planning is to first **sample experience from the model**.
- This is like **imagining possible scenarios** in the world based on your understanding of how the world works.
- This **generated experience** can then be **used to perform updates to the value function** as if these interactions actually occurred in the world.
- **Behaving greedily** with respect to these improved values results in **improved policy**.



Q-planning vs Q-learning

- Q-learning uses **experienced from the environment**, they performs an update to improve a policy.
- In Q-planning, we use **experience from the model** and perform a similar update to improve a policy.



Random Tabular Q-planning

1. Sampling  One possible option is to sample states and actions uniformly.
- $\mathcal{S} \longrightarrow S \longrightarrow \boxed{\text{Model}} \longrightarrow S'$
 $\mathcal{A} \longrightarrow A \longrightarrow \boxed{\text{Model}} \longrightarrow R$
2. Q-learning update
- $$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A) \right]$$
3. Greedy policy improvement
- $$\pi(s) = \operatorname{argmax}_a Q(s, a)$$

Random Tabular Q-planning Pseudo-code

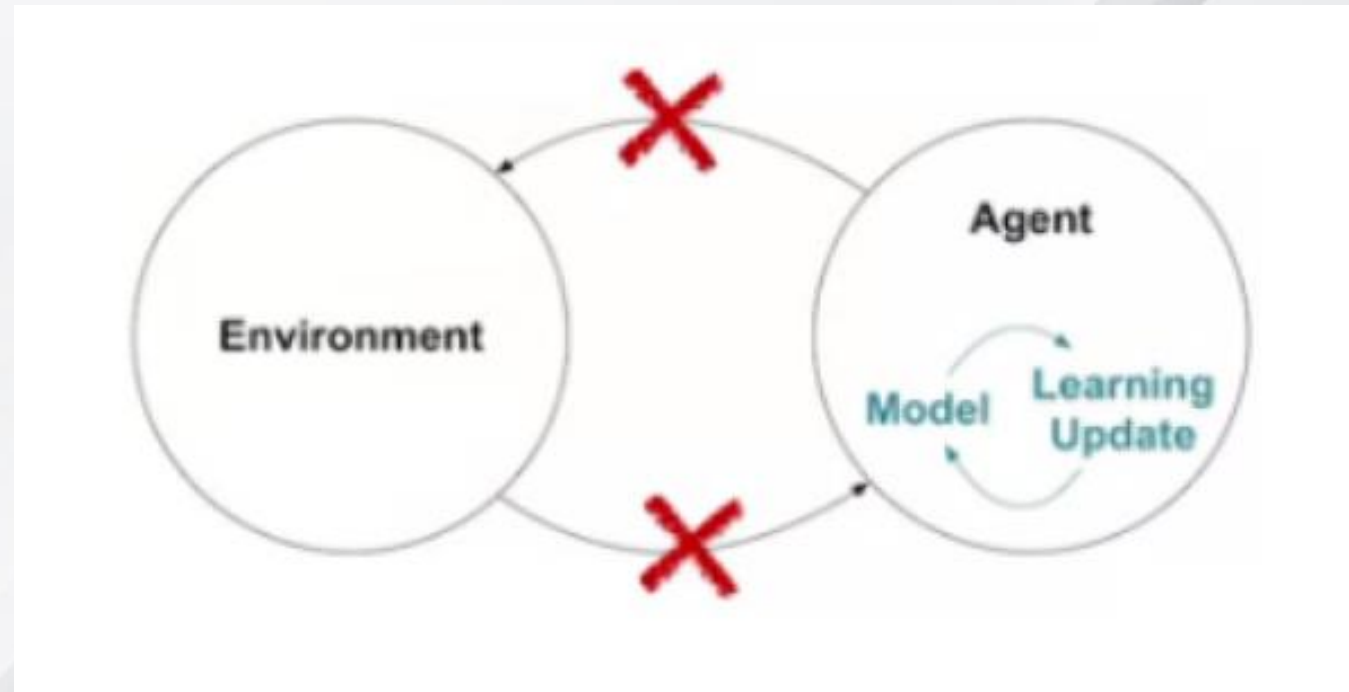
Random-sample one-step tabular Q-planning

Loop forever:

1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
2. Send S, A to a sample model, and obtain
a sample next reward, R , and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S' :
$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

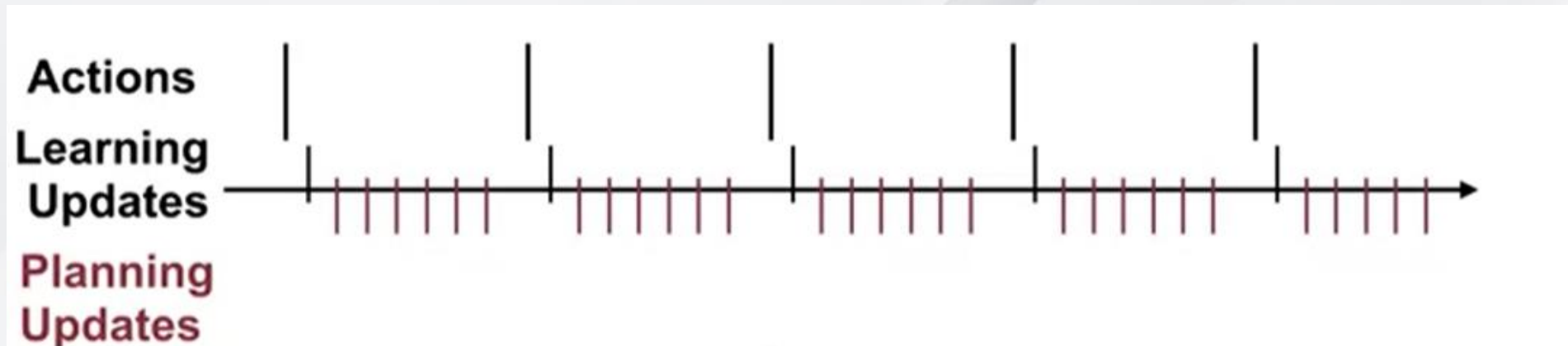
Planning uses imagined experience

- A key point is that this planning method only **uses imagined or simulated experience**.
- All of these updates can be done without behaving in the world or in parallel with the interaction loop.

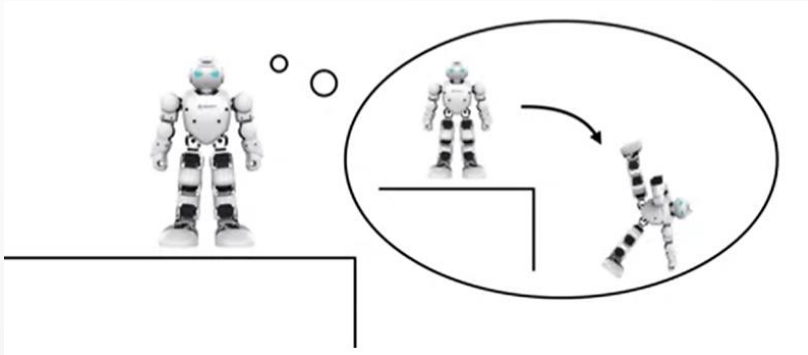


Planning Updates

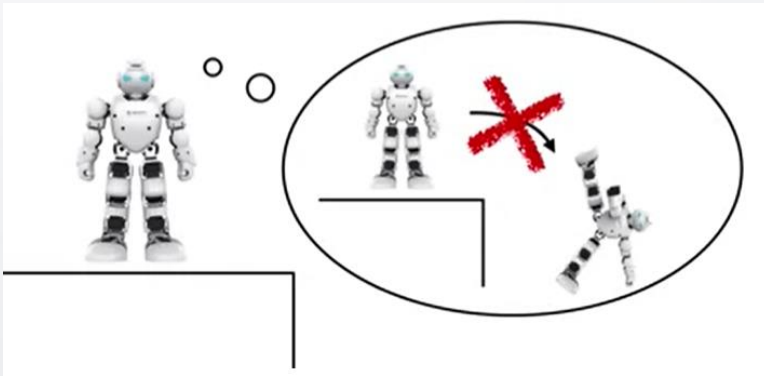
- **Actions** can only be taken at specific time points, but **Learning updates** can be executed **relatively fast**.
- This results in some **waiting time** from after the **Learning Update** and when the **next action** is taken.
- We can fill in this waiting time with **Planning Updates**.



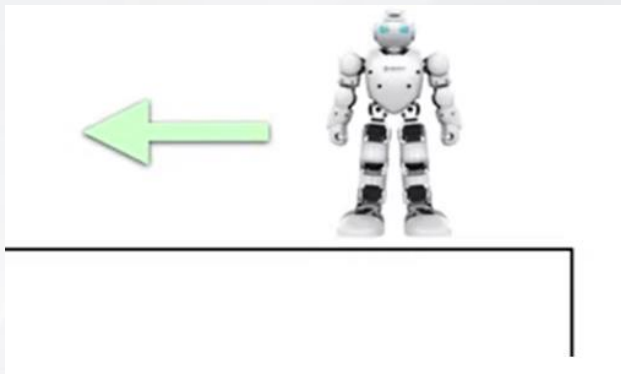
Planning Update Example



- Imagine we have a robot that is standing near a cliff. Let's say its model knows about the outcome of stepping off of the cliff, but it is not accurately represented in its value function or policy.



- It can generate simulated experience of stepping off of the cliff and perform many planning steps on these transitions.



- Its value function now better reflects that stepping off of the cliff is bad, and its policy will have its step away from the cliff.

Summary:

- Describe the Expected SARSA
- Describe the Double Q-learning algorithm
- Understand and describe n-step TD Learning
- Describe what a model is and how they can be used
- Classify models as distribution models or sample models
- Explain how planning is used to improve policies



Thank You