

Reinforcement Learning (AI644)

Prepared by Dr. Ashraf Elsayed

Lecture 10



Reinforcement Learning

Lecture 10

Prepared by
Dr. Ashraf Elsayed

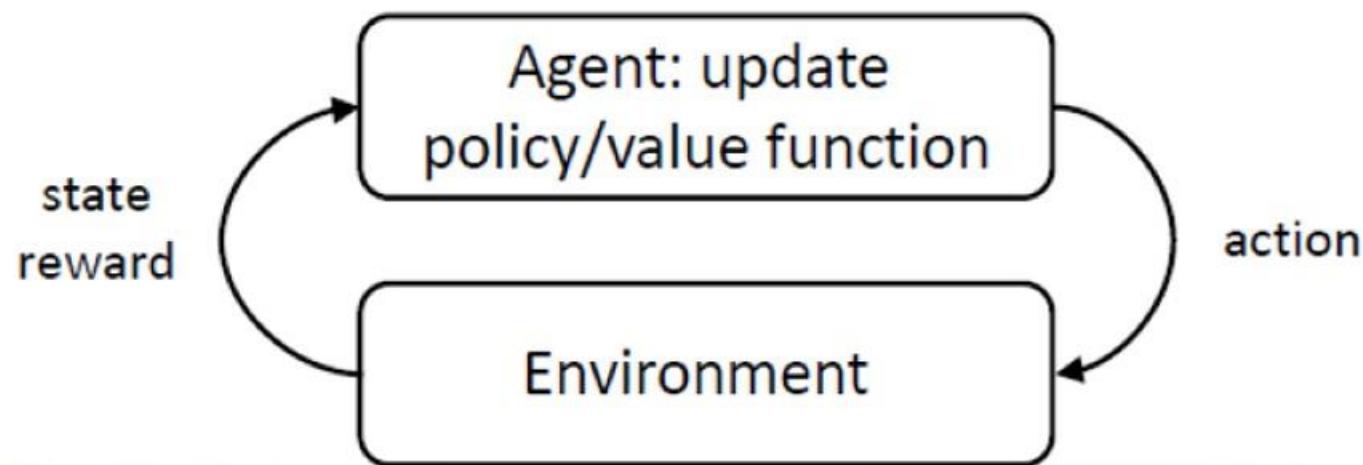
Dyna Architecture

The objectives of this lecture is to:

- Understand how simulating experience from the model differs from interacting with the environment
- Describe how Tabular Dyna-Q works
- Identify the direct-RL and planning updates in Tabular Dyna-Q
- Explain how having an accurate model allows the agent to learn with fewer environment interactions
- Explain the effects of planning with an inaccurate model
- Explain Monte Carlo Tree Search

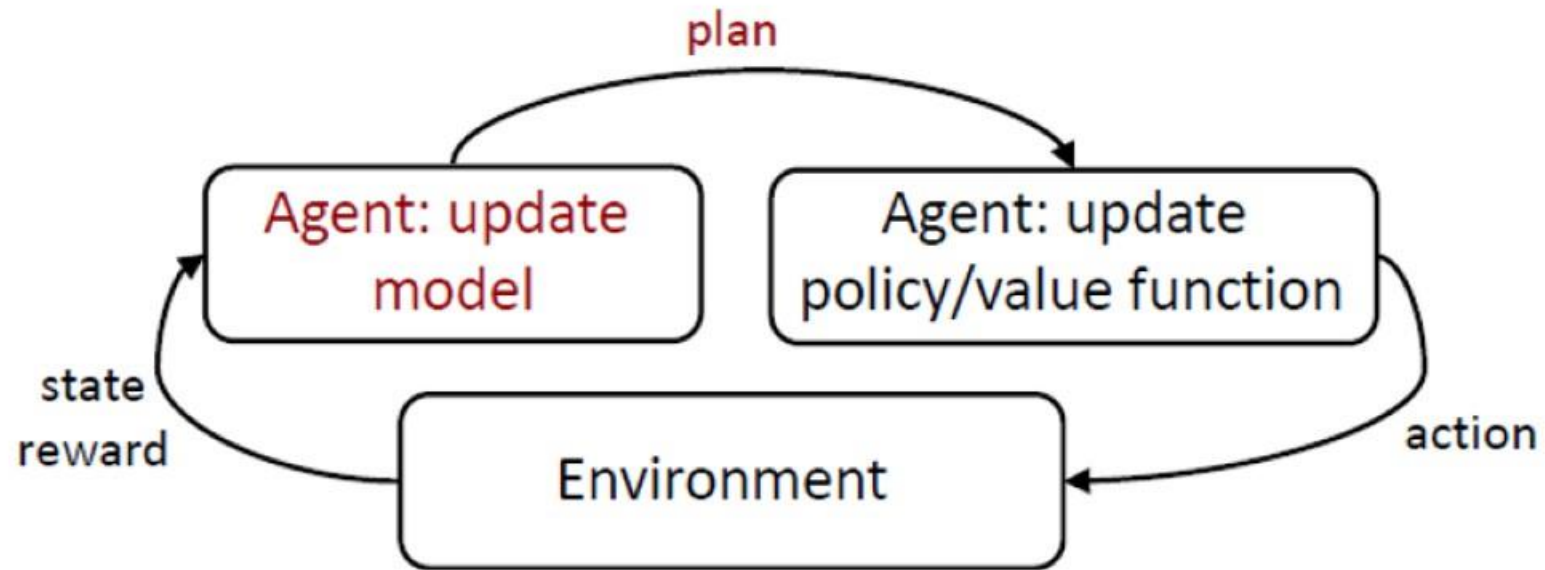
Model-free RL

- No explicit transition or reward models
 - Q-learning: **value-based method**
 - Policy gradient: **policy-based method**
 - Actor critic: **policy and value based method**



Model-based RL

- Learn explicit transition and/or reward model
 - Plan based on the model
 - Benefit: Increased sample efficiency
 - Drawback: Increased complexity



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

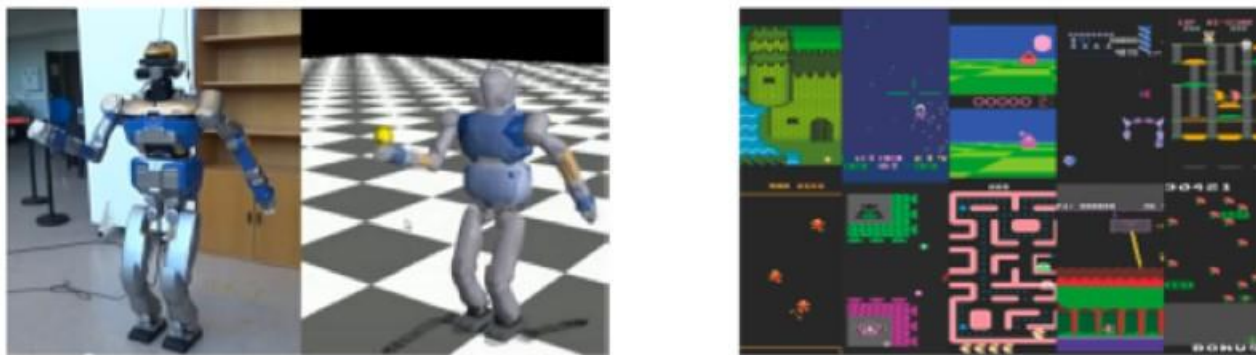
Solve: $V^*(s) = \max_a R(s, a) + \gamma \sum_{s'} \Pr(s'|s, a) V^*(s') \forall s$

$s \leftarrow s'$

Until convergence of V^*

Return V^*

Complex models



- Use function approx. for transition and reward models
 - Linear model: $pdf(s'|s, a) = N(s'|w^T \begin{bmatrix} s \\ a \end{bmatrix}, \sigma^2 I)$
 - Non-linear models:
 - Stochastic (e.g. Gaussian process):

$$pdf(s'|s, a) = GP(s|w^T \begin{bmatrix} s \\ a \end{bmatrix}, \sigma^2 I)$$
 - Deterministic (e.g., neural network): $s' = T(s, a) = NN(s, a)$

Partial Planning

- In complex models, fully optimizing the policy or value function at each time step is intractable
- Consider partial planning
 - A few steps of Q-learning
 - A few steps of policy gradient

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

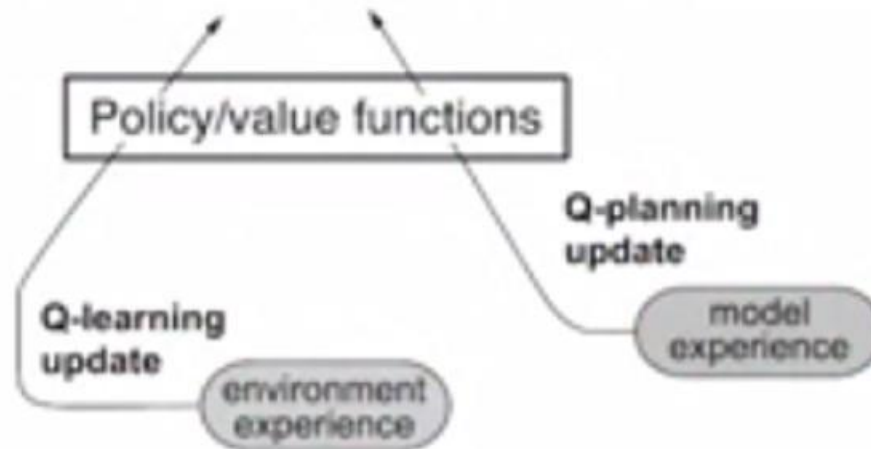
Partial Planning vs Replay Buffer

- Previous algorithm is very similar to Model-free Q-learning with a replay buffer
- Instead of updating Q-function based on samples from replay buffer, generate samples from model
- Replay buffer:
 - Simple, real samples, no generalization to other state-action pairs
- Partial planning with a model
 - Complex, simulated samples, generalization to other state-action pairs (can help or hurt)

Dyna Architecture

- Q-learning performs updates using environment experience.
- Q-planning on the other hand performs updates using simulated experience from the model.
- We can combine these two ideas through the Dyna architecture.

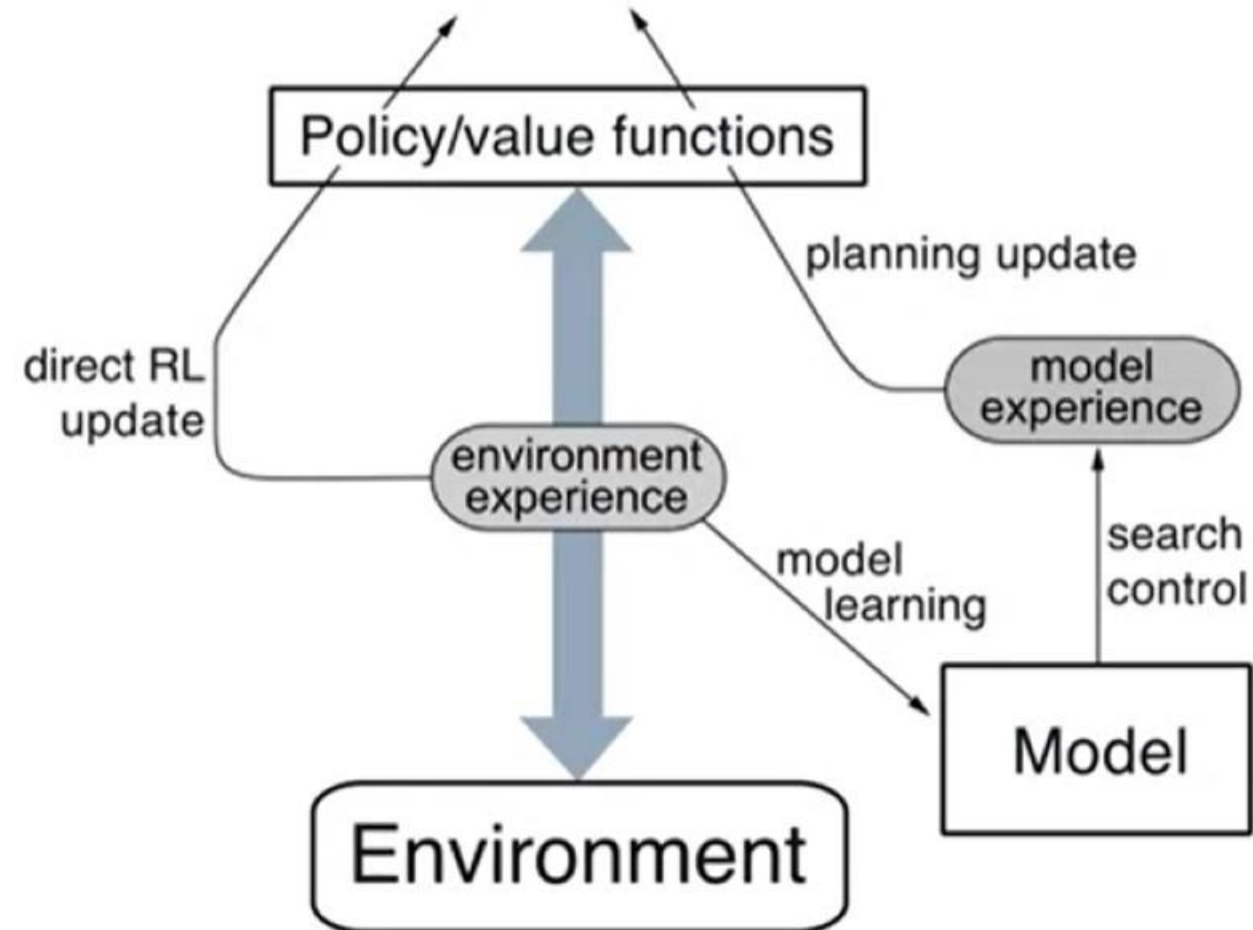
Combining Q-learning and Q-planning



- We have the usual environment and policy. They generate a stream of experience.
- We use this experience to perform direct RL updates.
- To do planning, we need a model. The environment experience can be used to learn the model.
- This model will be used to generate model experience.
- In addition, we want to control how the model generates this simulated experience, what states the agent will plan from. We call this process search control.
- Planning updates are performed using the experience generated by the model

Dyna Architecture

The Dyna Architecture

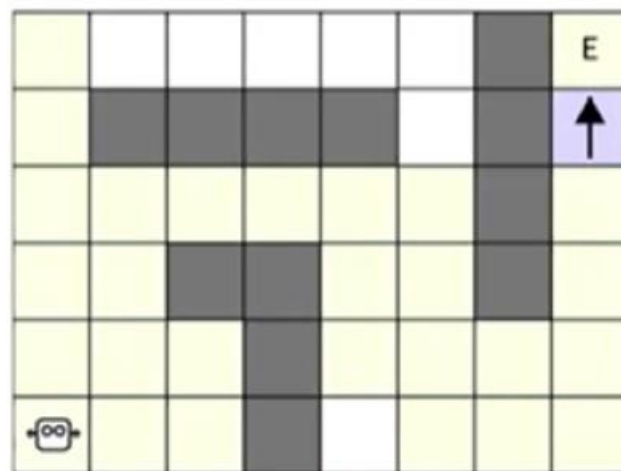


Example: Dyna Architecture

One direct RL update, using Q-learning:

$$Q(s, a) \leftarrow Q(s, a) + \alpha (r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a))$$

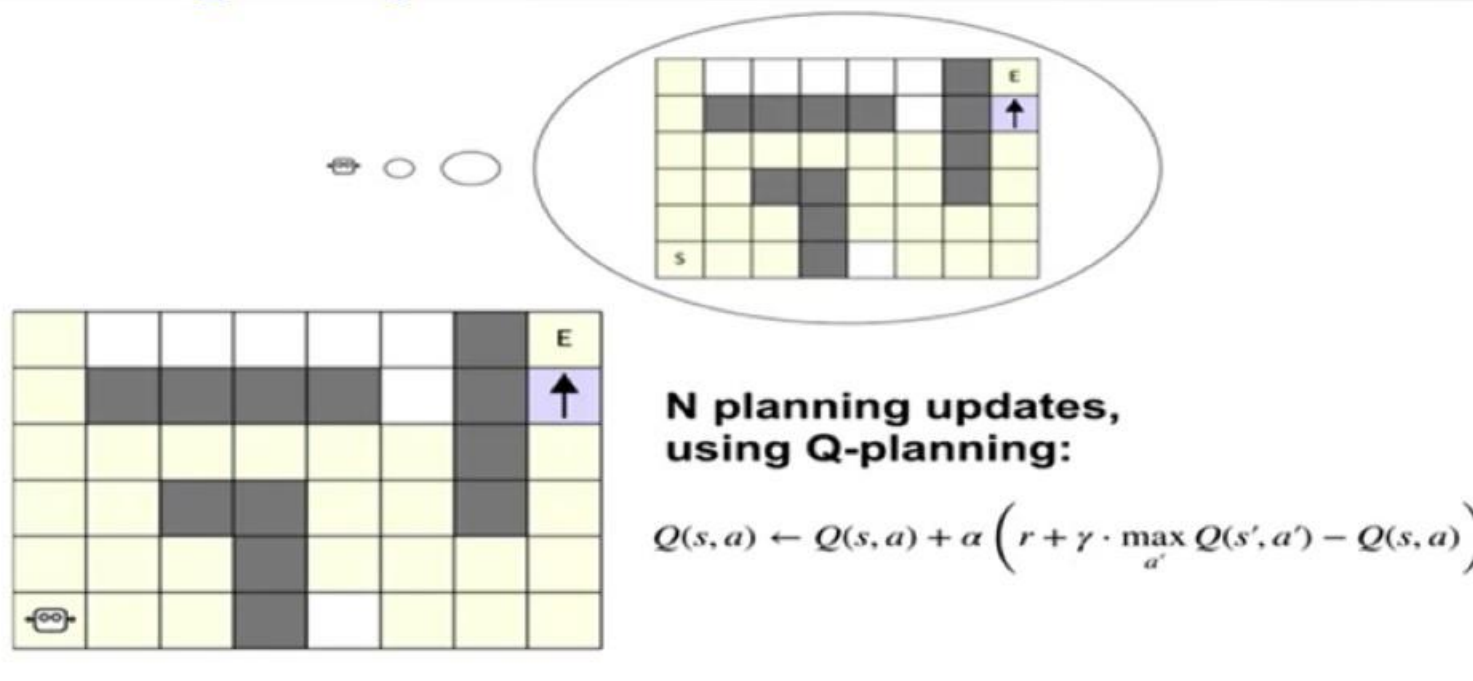
The agent receives 0 reward in all transitions except for the final state, it receives +1



After 1st episode

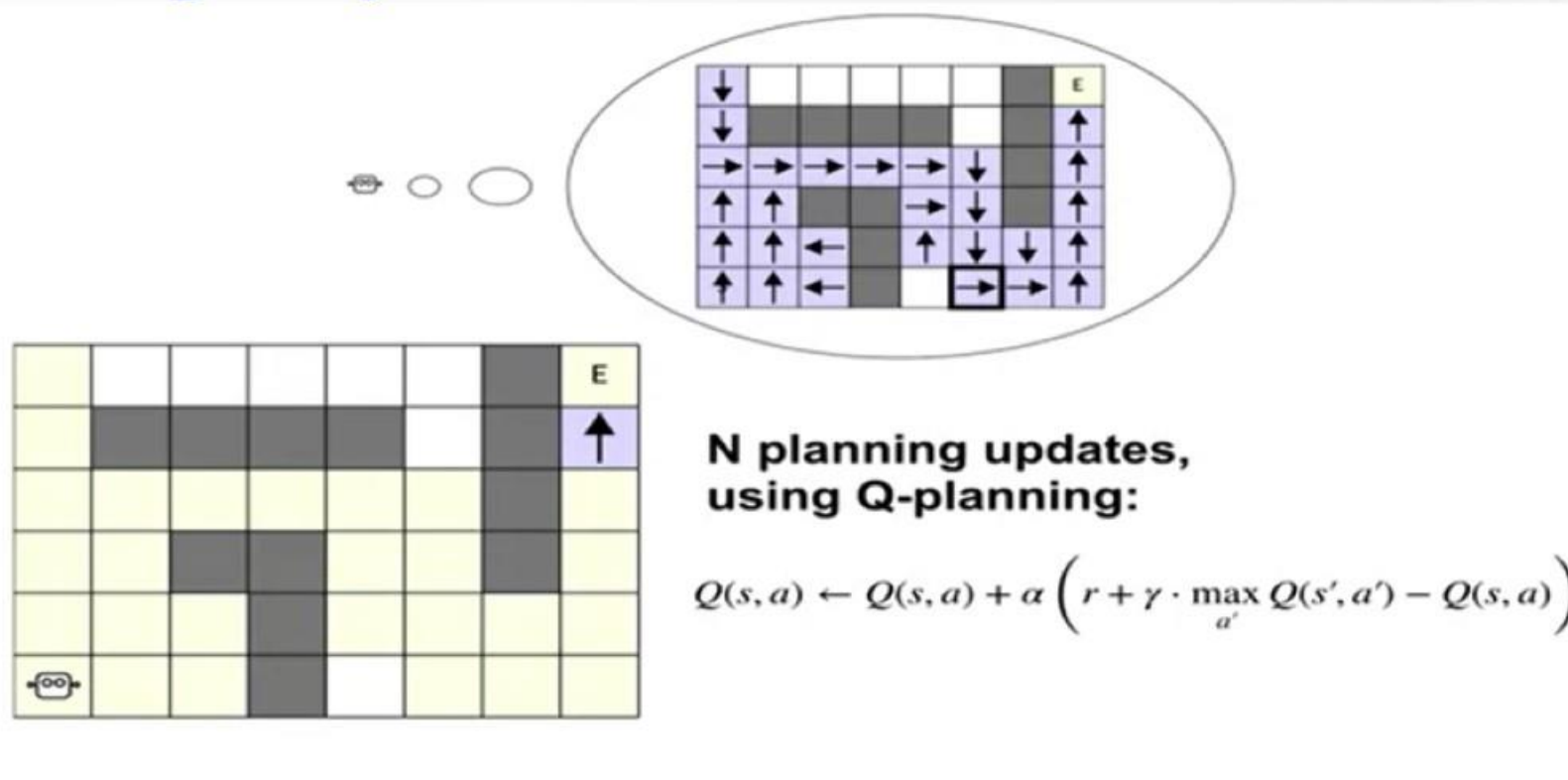
- Dyna makes use of all the experience generated during the first episode to learn a model the environment.
- The yellow states correspond to states visited during the first episode.
- Dyna performs planning on every time step. However, planning has no impact on the policy during 1st episode even though the model became more accurate on each time step.

Example: Dyna Architecture



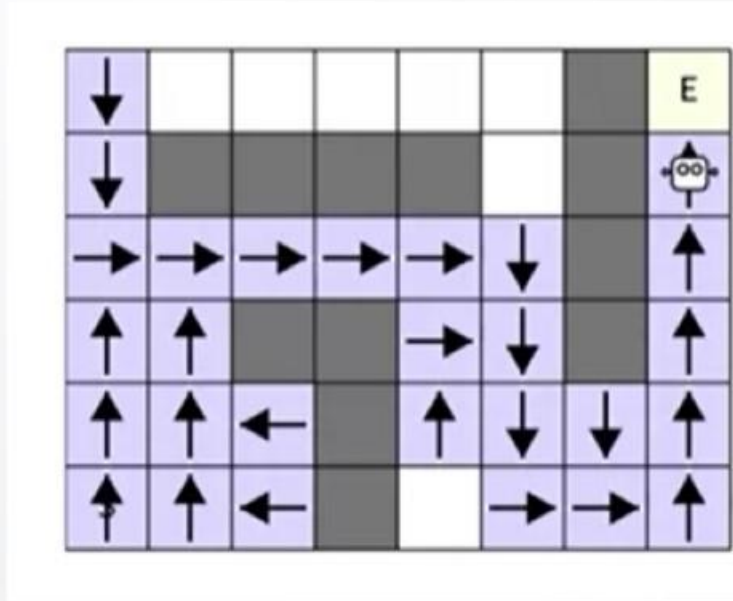
- The model knows about all the states the robot visited during the 1st episode.
- If we run the planning loop at the beginning of the 2nd episode, Dyna can simulate transitions from any of the state action pairs the agent visited during the first episode.
- Planning replays the simulated transitions as if they happened in the real world.
- Each step of planning simply applies the Q-learning update to a simulated transition and updates the value function.

Example: Dyna Architecture



- With enough planning steps, the agent can approve the policy in all the states visited during the 1st episode.
- Here, planning produce a pretty good policy.
- The policy is not perfect since it is still random in states that were not visited before.

Example: Dyna Architecture

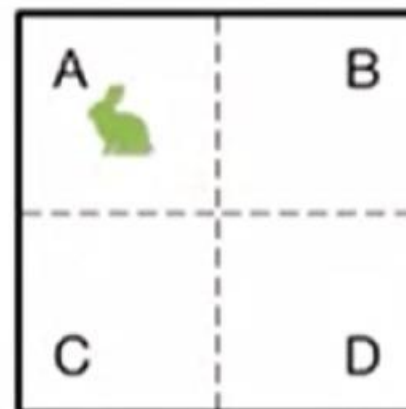
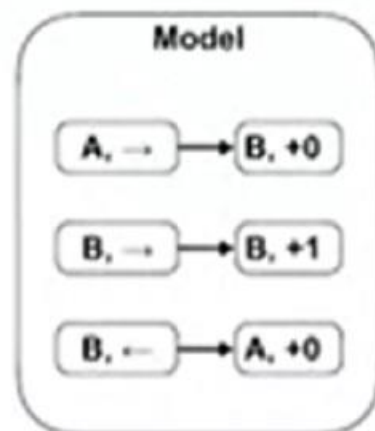


- The robot now knows how to get to the goal quickly after just one episode.
- In comparison, Q-learning require many episodes to achieve similar performance.
- Dyna does more computation per step, but uses limited experience more efficiently.
- The robot would continue exploring and planning on each step of the second episode. So the policy would take random actions once in a while and the policy would continue to change, and improve due to planning.

Tabular Dyna-Q Algorithm

- Tabular Dyna-Q is a specific instance of the Dyna Architecture that assumes **deterministic transitions**.
- For example, when this rabbit chooses to move right in state A, there's only one outcome. The rabbit transitions to state B with a reward of zero.
- Likewise, if the rabbit moves right in state B, it stays in state B with a reward of one.
- The left action in B always takes the rabbit to state A with a reward of zero.

Learning a deterministic model



- After the rabbit has experienced these transitions, the model knows what happens in these state action pairs. If a transition occurred once, it will always unfold that way in the future. The model can be certain about this.

Tabular Dyna-Q Algorithm



Tabular Dyna-Q

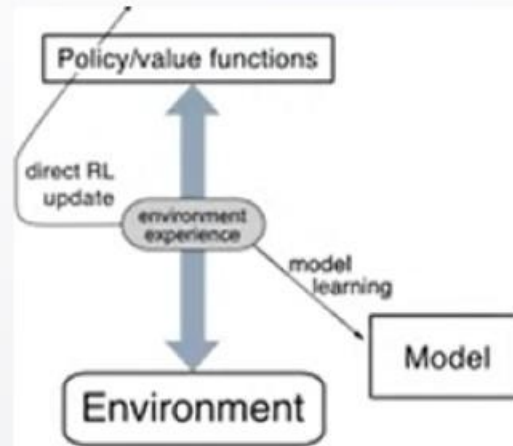
Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Loop forever:

- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \epsilon$ -greedy(S, Q)
- (c) Take action A ; observe resultant reward, R , and state, S'
- (d) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

- It performs a Q-learning update with this transition, **direct-RL**.
- If we were to stop here, we would get exactly the **Q-learning algorithm**.

Tabular Dyna-Q Algorithm



Tabular Dyna-Q

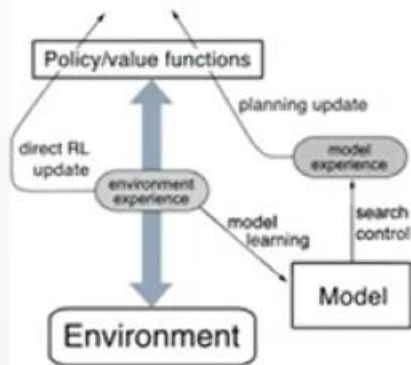
Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Loop forever:

- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \epsilon$ -greedy(S, Q)
- (c) Take action A ; observe resultant reward, R , and state, S'
- (d) $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
- (e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)

- Dyna-Q will take this transition and perform a **model learning step** with it.
- The algorithm memorizes the next state and reward for the given state action pair.
- This works because we assume the environment is deterministic

Tabular Dyna-Q Algorithm



Tabular Dyna-Q

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Loop forever:

- $S \leftarrow$ current (nonterminal) state
- $A \leftarrow \varepsilon\text{-greedy}(S, Q)$
- Take action A ; observe resultant reward, R , and state, S'
- $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
- $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)

$S \leftarrow$ random previously observed state

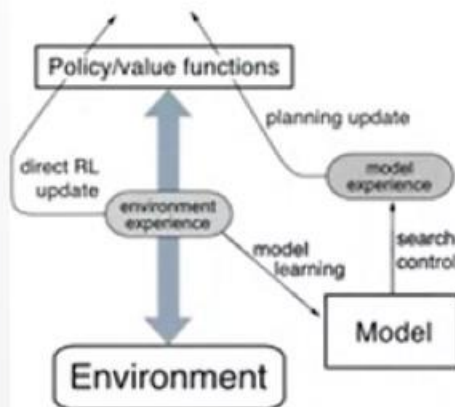
$A \leftarrow$ random action previously taken in S

$R, S' \leftarrow Model(S, A)$

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

- Dyna-Q then performs several steps of planning. Each planning step consists of three steps: **search control**, **model query**, and **value update**.
- The **search control** selects a previously visited state action pair at random. It must be a state action pair the agent has seen before, otherwise, the model would not know what happens next.

Tabular Dyna-Q Algorithm



Tabular Dyna-Q

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Loop forever:

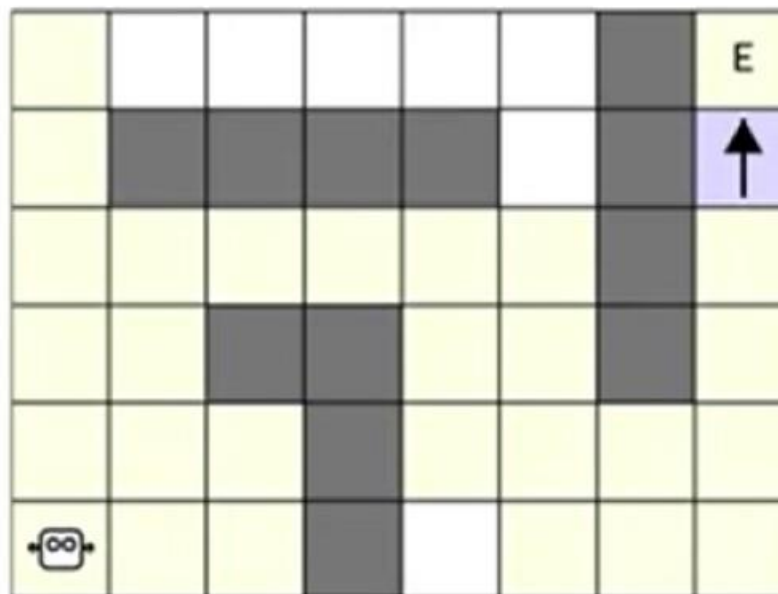
- $S \leftarrow$ current (nonterminal) state
- $A \leftarrow \varepsilon$ -greedy(S, Q)
- Take action A ; observe resultant reward, R , and state, S'
- $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
- $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)
- Loop repeat n times:
 - $S \leftarrow$ random previously observed state
 - $A \leftarrow$ random action previously taken in S
 - $R, S' \leftarrow Model(S, A)$
 - $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

- Finally, Dyna-Q performs a Q-learning update with the simulated transition.
- The planning step is repeated many times.
- The most important thing to remember is that **Dyna-Q performs many planning updates for each environment transition**

Example: Dyna-Q

During the 2nd
episode

Number of actions taken: 184

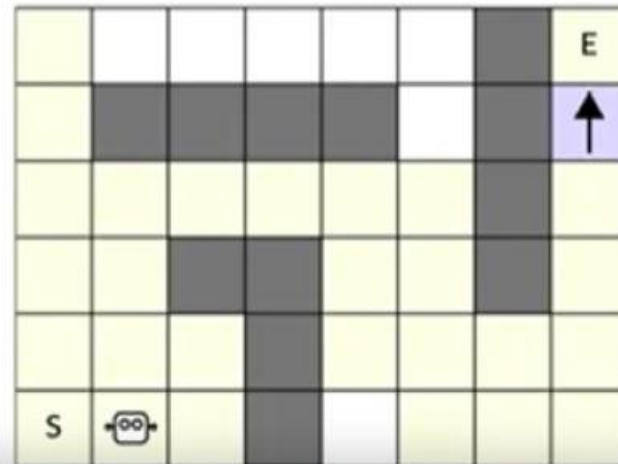


- The first episode takes a 184 steps. In that time, the agent only change the value of the state action pair beside the goal.
- Without planning, the next few episodes would probably be just as long.
- Let's look at the next episode and study the impact of doing 100 steps of planning on every time step.

Example: Dyna-Q

During the 2nd episode

Number of steps planned: 100
Number of actions taken: 185



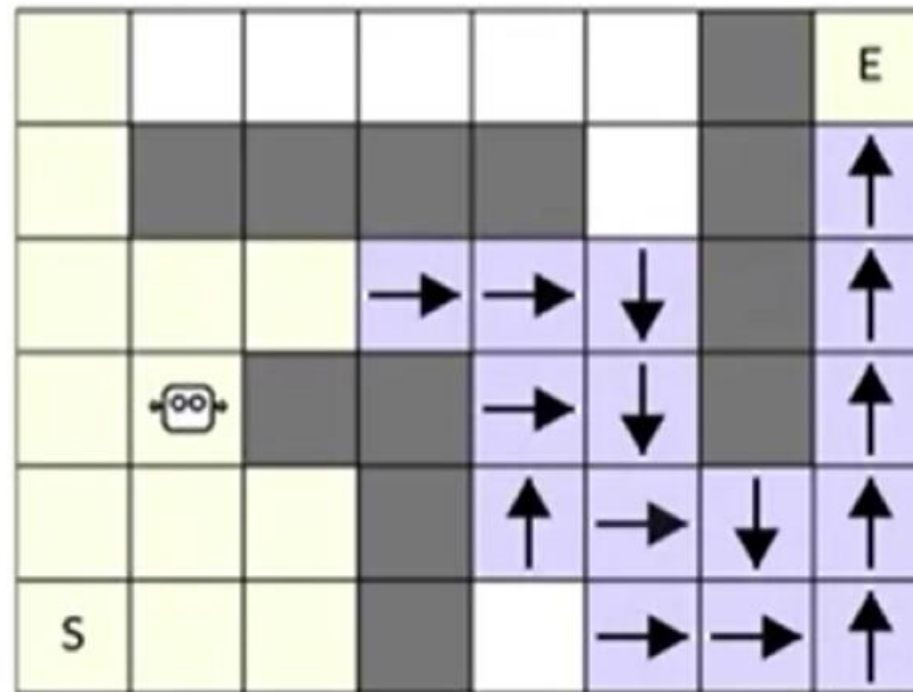
- After just one step, planning updated the values of two more states.
- Now, the agent knows the correct actions in the final corridor.
- Let's run Dyna-Q a bit longer.

Example: Dyna-Q

During the 2nd
episode

Number of steps planned: 1300

Number of actions taken: 197



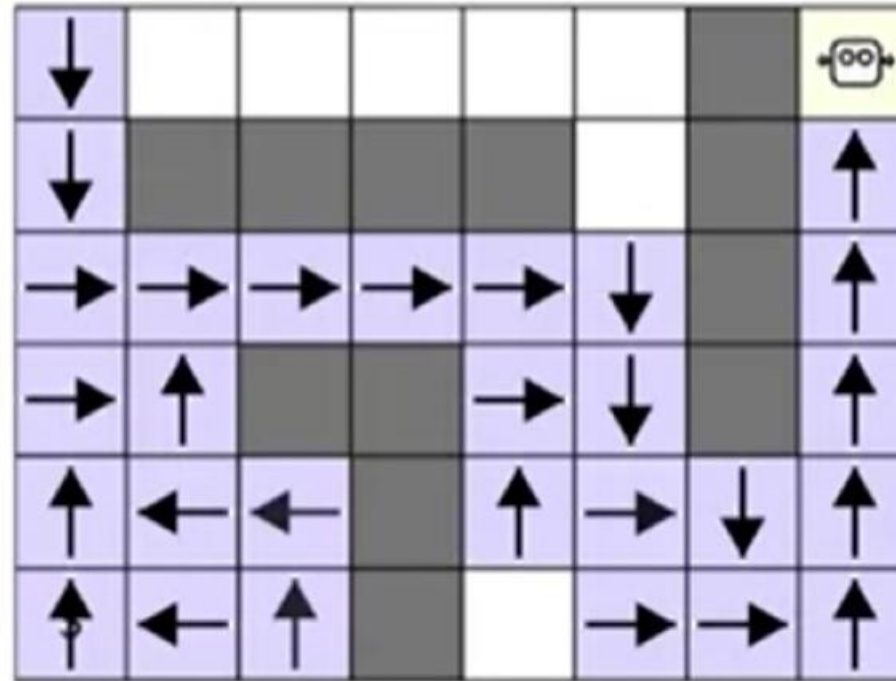
- The impact of planning is quite dramatic. Dyna-Q propagates the reward information across the entire state space.

Example: Dyna-Q

During the 2nd
episode

Number of steps planned: 3800

Number of actions taken: 223



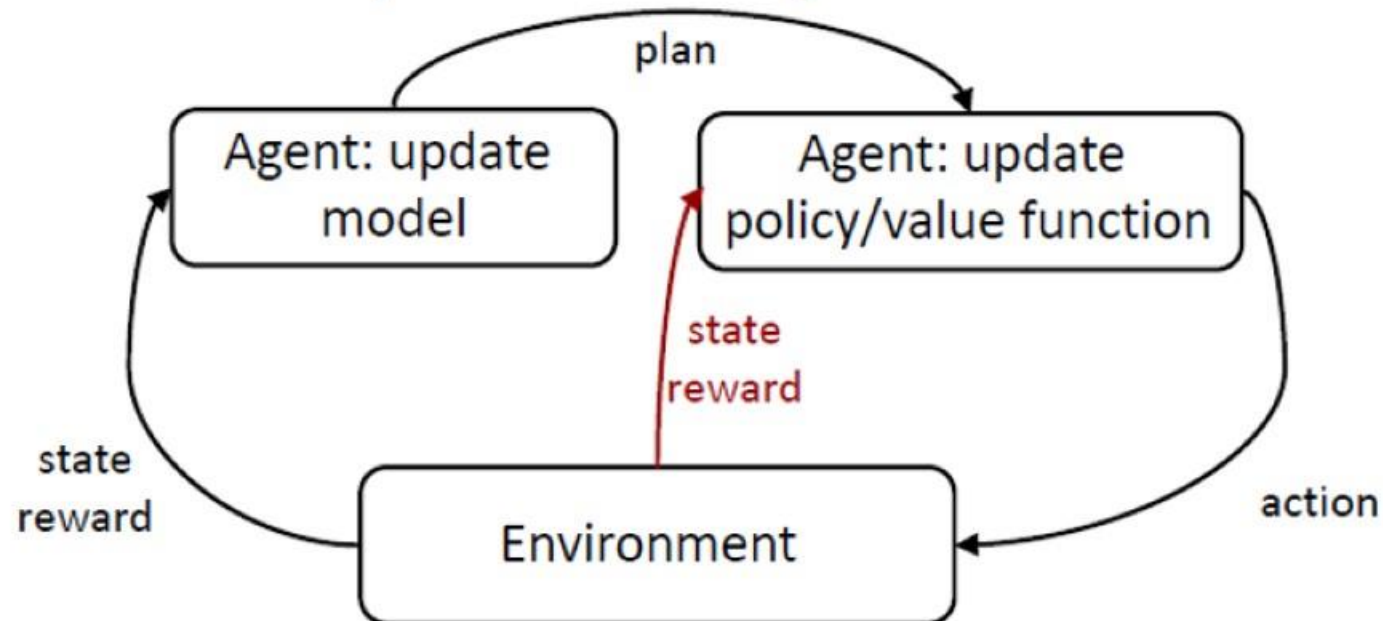
- Eventually, the agent knows an effective policy for navigating to the goal from most states.
- The policy computed by planning only requires 18 steps to reach the goal. After two episodes, that's more than ten times shorter than the first episode.

Tabular Dyna-Q

- Tabular Dyna-Q performed many more updates to the value function than it would have without planning.
- Dyna-Q makes better use of its limited interaction with the environment.
- Tabular Dyna-Q mixes planning, learning, and acting through the value function.

Dyna

- **Learn explicit transition and/or reward model**
 - Plan based on the model
- **Learn directly from real experience**



Dyna-Q

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

$\delta \leftarrow r + \gamma \max_{a'} Q(s', a') - Q(s, a)$

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

Repeat a few times:

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

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

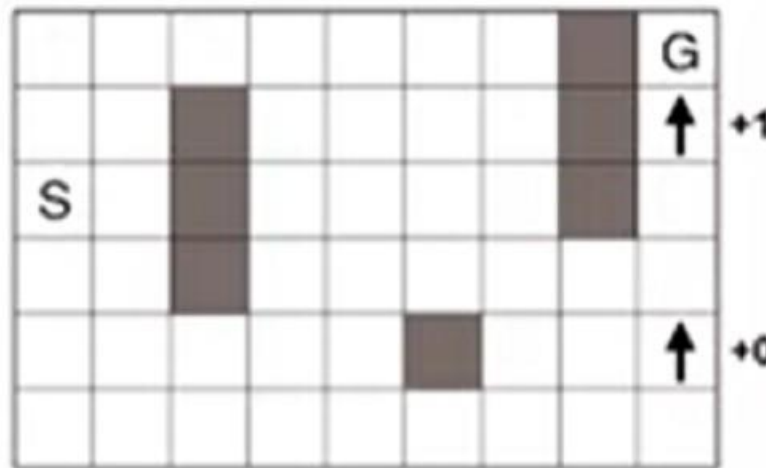
Update Q : $w_Q \leftarrow w_Q - \alpha_Q \delta \nabla_{w_Q} Q(\hat{s}, \hat{a})$

$s \leftarrow s'$

Return Q

Dyna vs Q-learning

The maze

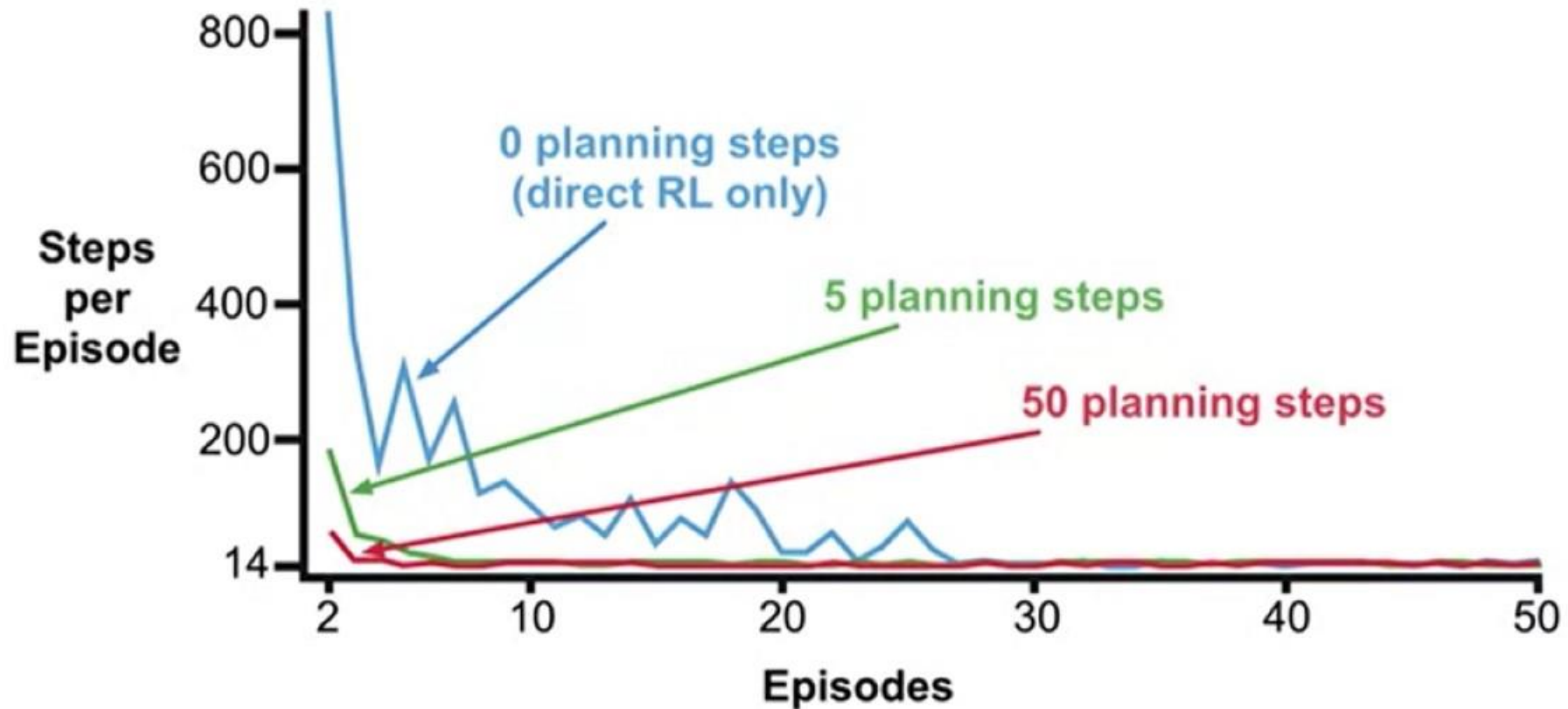


$$\gamma = 0.95$$

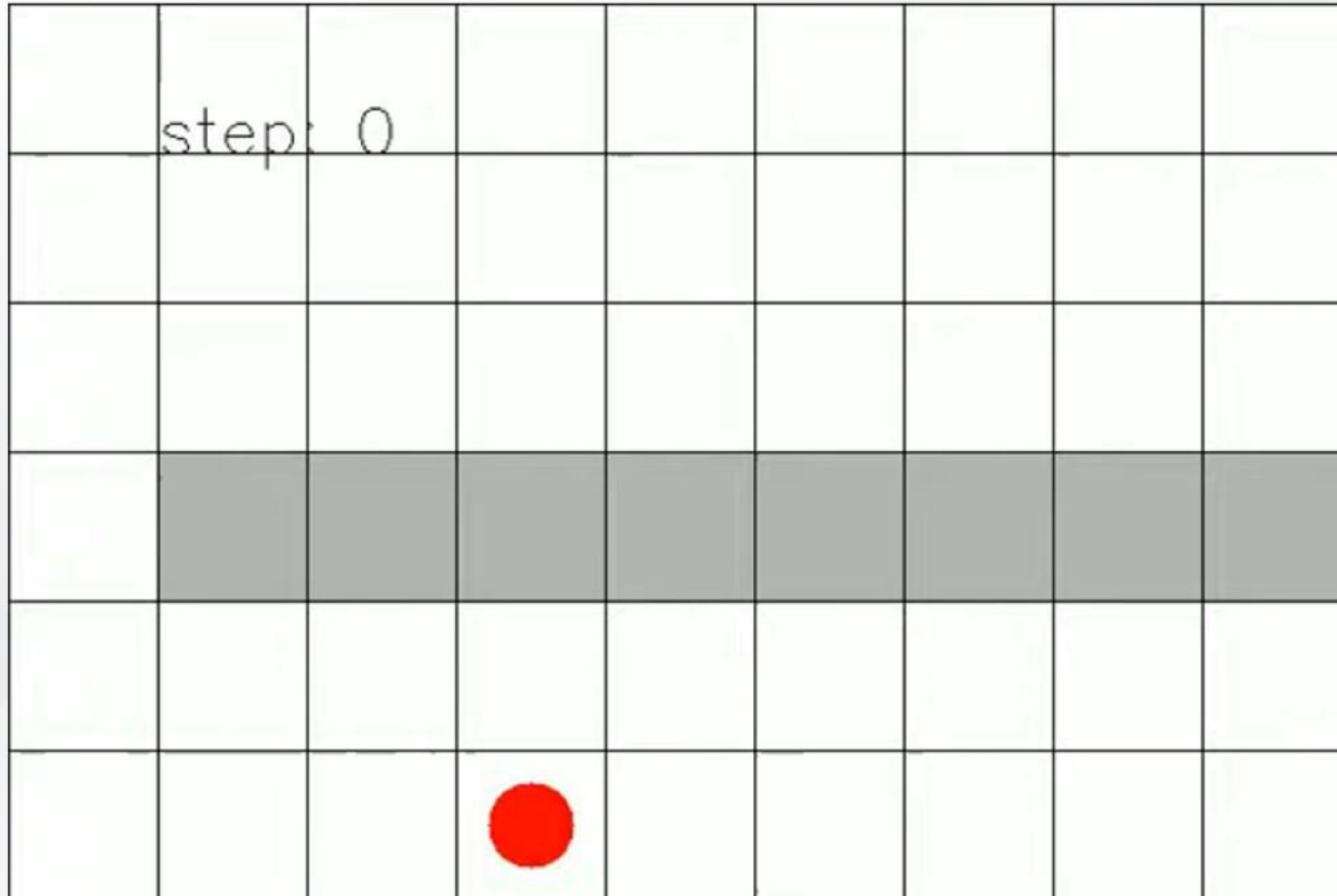
- Run some experiments on this small maze environment.
- Like before, the agent has four actions. Transitioning into the goal state generates a reward of +1. All other transitions generate a reward of 0.
- To compare Dyna-Q with different amounts of planning, three agents are compared.
- Each one will use alpha equal to 0.1 and epsilon equal to 0.1. We initialize the action value estimates to 0 for all three agents. We run the experiment for 50 episodes 30 times and then average the results.

Dyna vs Q-learning

More planning → faster learning



Dyna-Q (Simulation example)



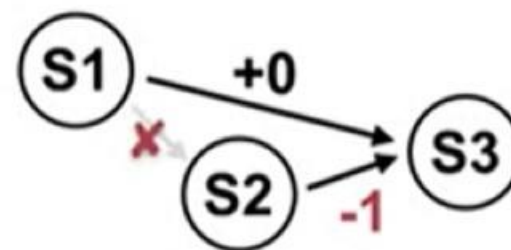
How models can be inaccurate



$S1 \rightarrow S2, +1$

$S2 \rightarrow ??$

Incomplete model

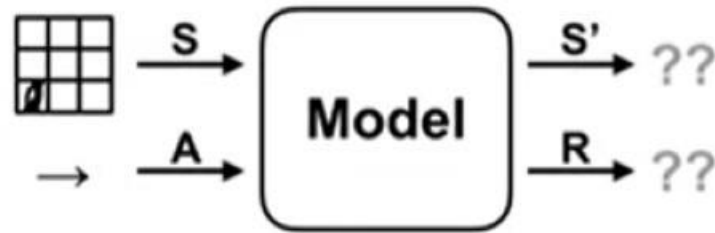


$S1 \rightarrow S2, +1$ **X**

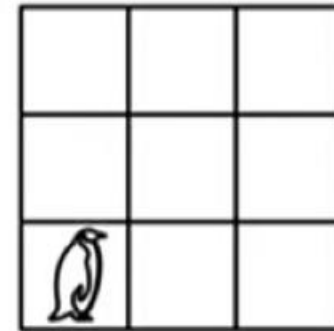
$S2 \rightarrow S3, -1$

Changing environment

Planning with an inaccurate model



$$Q(S, A) \leftarrow Q(S, A) + ??$$



$t = 0$

Incomplete Model

Dyna with an incomplete model

Tabular Dyna-Q

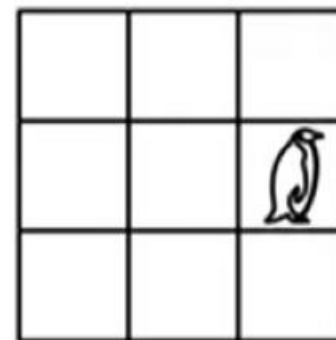
Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Loop forever:

- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \epsilon$ -greedy(S, Q)
- (c) Take action A ; observe resultant reward, R , and state, S'
- (d) $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
- (e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
 - $S \leftarrow$ random previously observed state
 - $A \leftarrow$ random action previously taken in S
 - $R, S' \leftarrow Model(S, A)$
 - $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

Dealing with inaccurate models

Planning with an inaccurate model

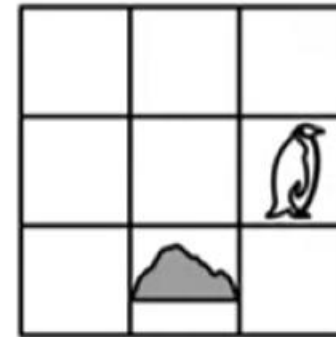
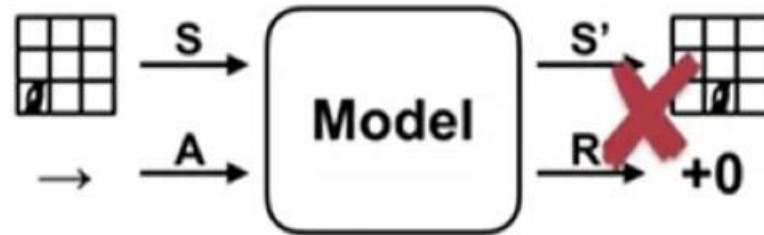


t = 3

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma \max_a Q(S', a) - Q(S, A))$$

Dealing with inaccurate models

Planning with an inaccurate model

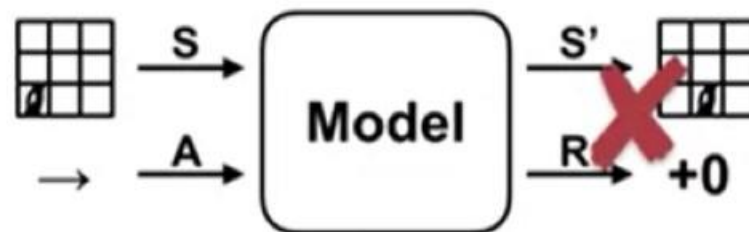


t = 3

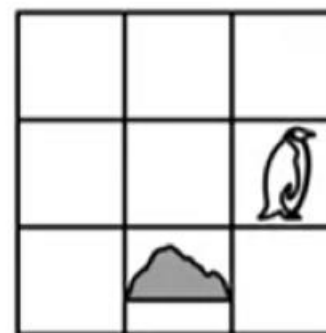
$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma \max_a Q(S', a) - Q(S, A))$$

Dealing with inaccurate models

Planning with an inaccurate model



$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma \max_a Q(S', a) - Q(S, A))$$

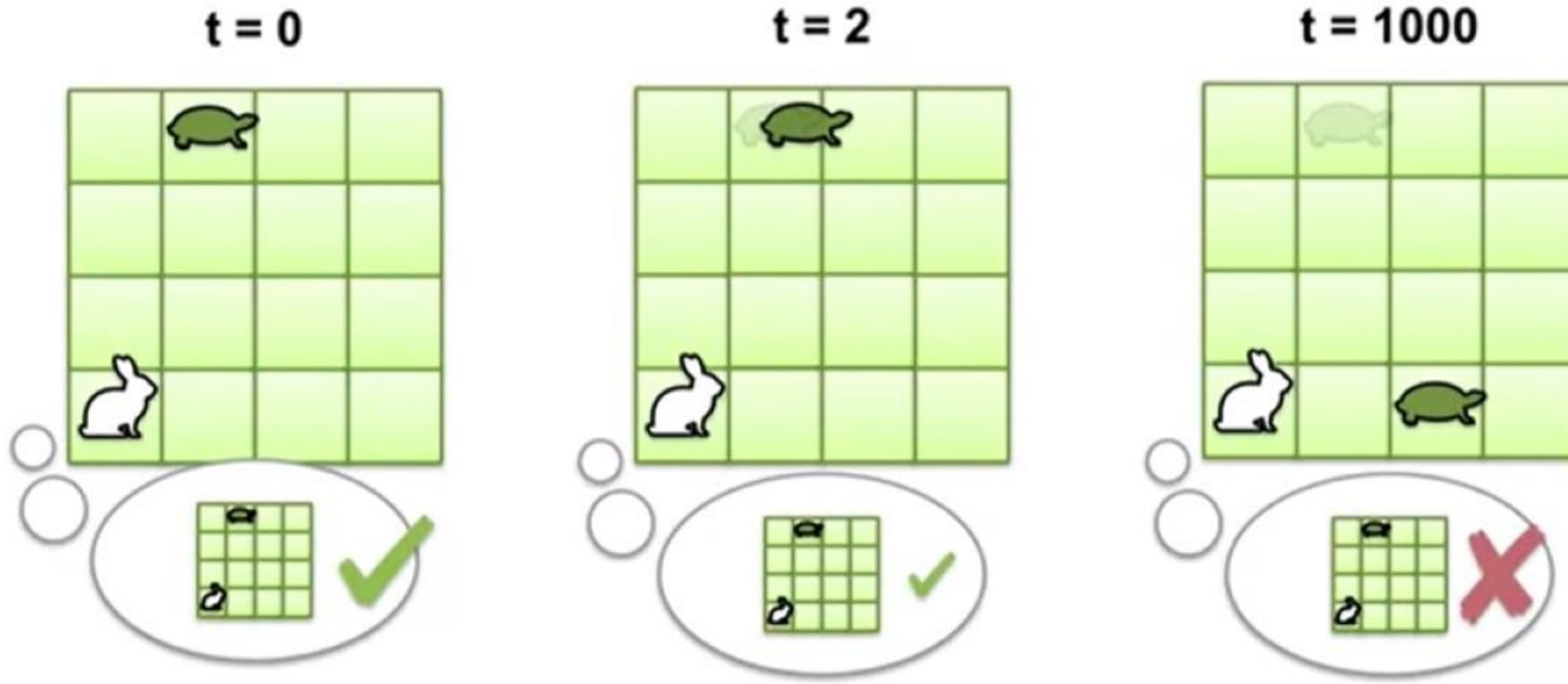


t = 3

The Model becomes outdated when the Environment Changes

Exploration and exploitation for model accuracy

Model inaccuracy and time



Exploration and exploitation for model accuracy

Bonus rewards for exploration

$$\text{New reward} = r + \kappa\sqrt{\tau}$$

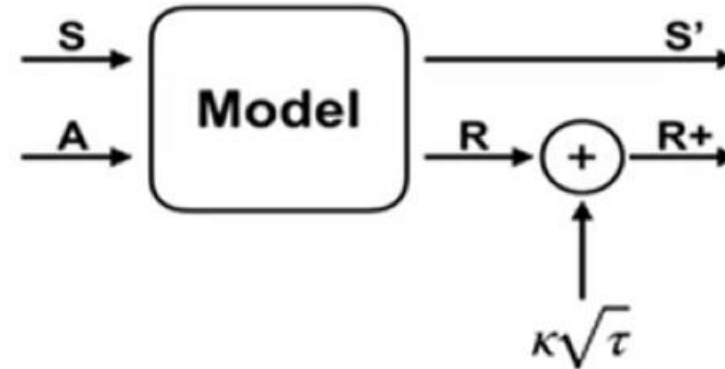
Actual reward

Small constant

Time steps since transition was last tried

The Dyna-Q+ algorithm

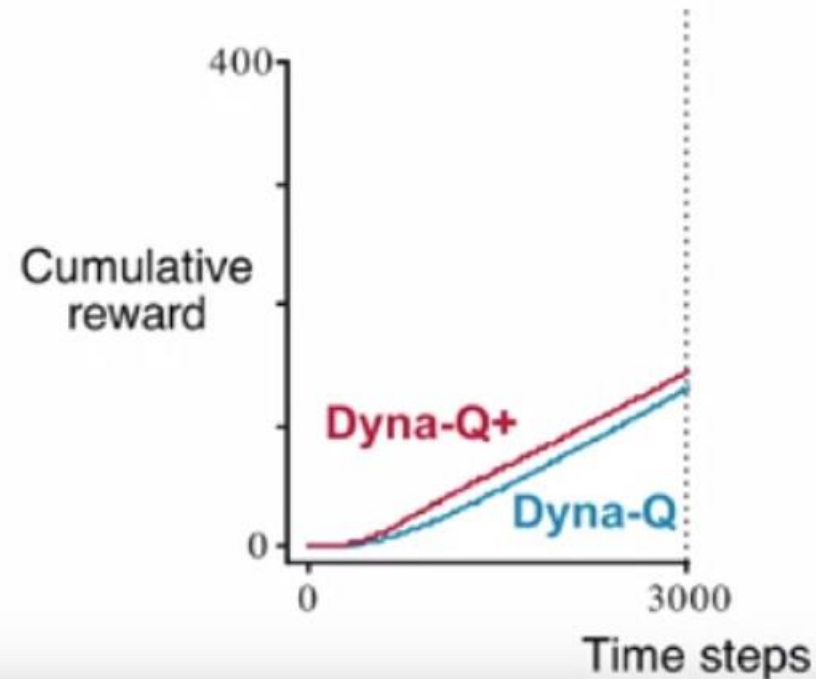
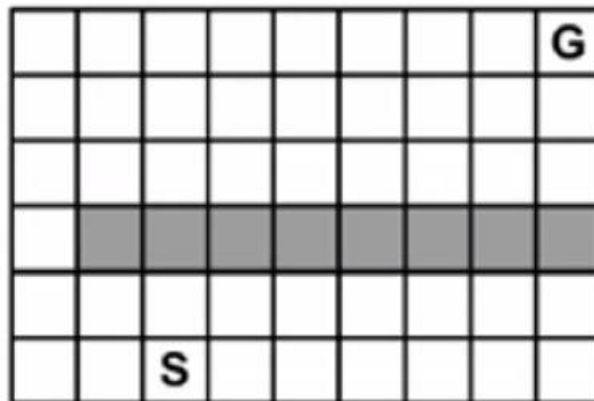
Adding this reward bonus to Dyna-Q's planning updates results in the Dyna-Q+ algorithm



- By artificially increasing the rewards used in planning, we increase the value of state action pairs it haven't been visited recently.
- Imagine a state action pair that has not been visited in a long time. That means τ will be large. As τ grows, the bonus becomes bigger and bigger.
- Eventually, planning will change the policy to go directly to S due to the large bonus.
- When the agent finally visits state S , it might see a big reward, or it might be disappointed. Either way, the model will be updated to reflect the dynamics of the environment

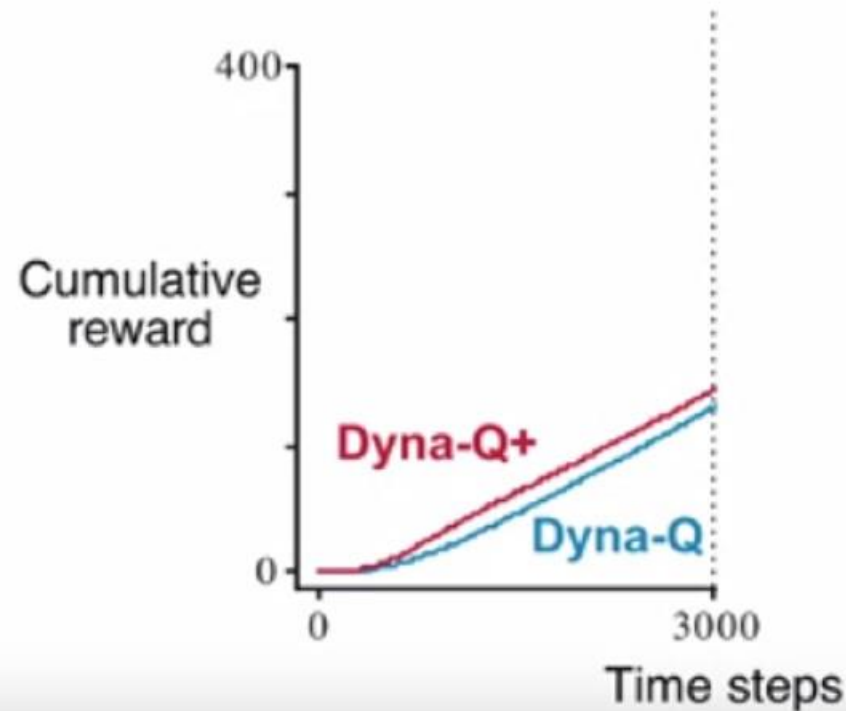
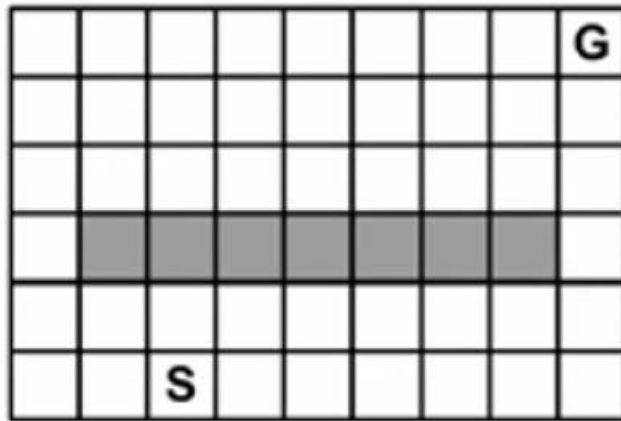
Dyna-Q vs Dyna-Q+

Dyna-Q vs Dyna-Q+ in a changing environment



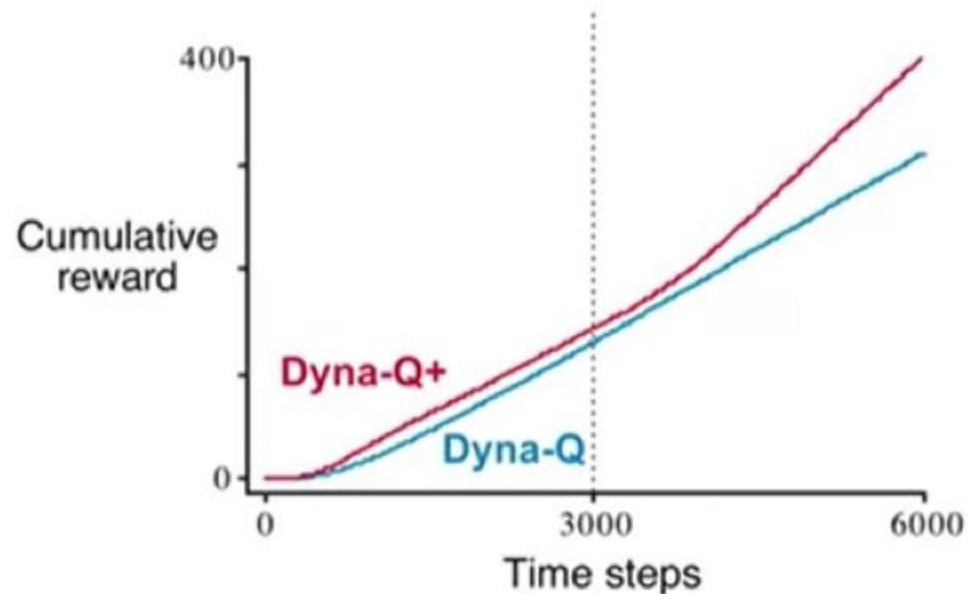
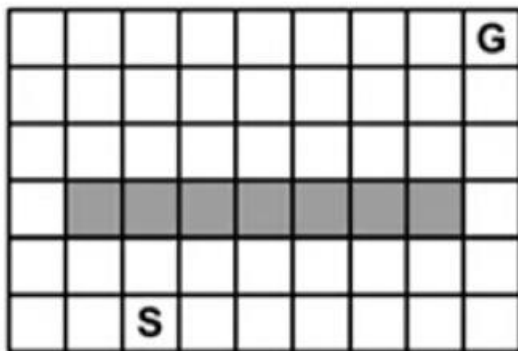
Dyna-Q vs Dyna-Q+

Dyna-Q vs Dyna-Q+ in a changing environment



Dyna-Q vs Dyna-Q+

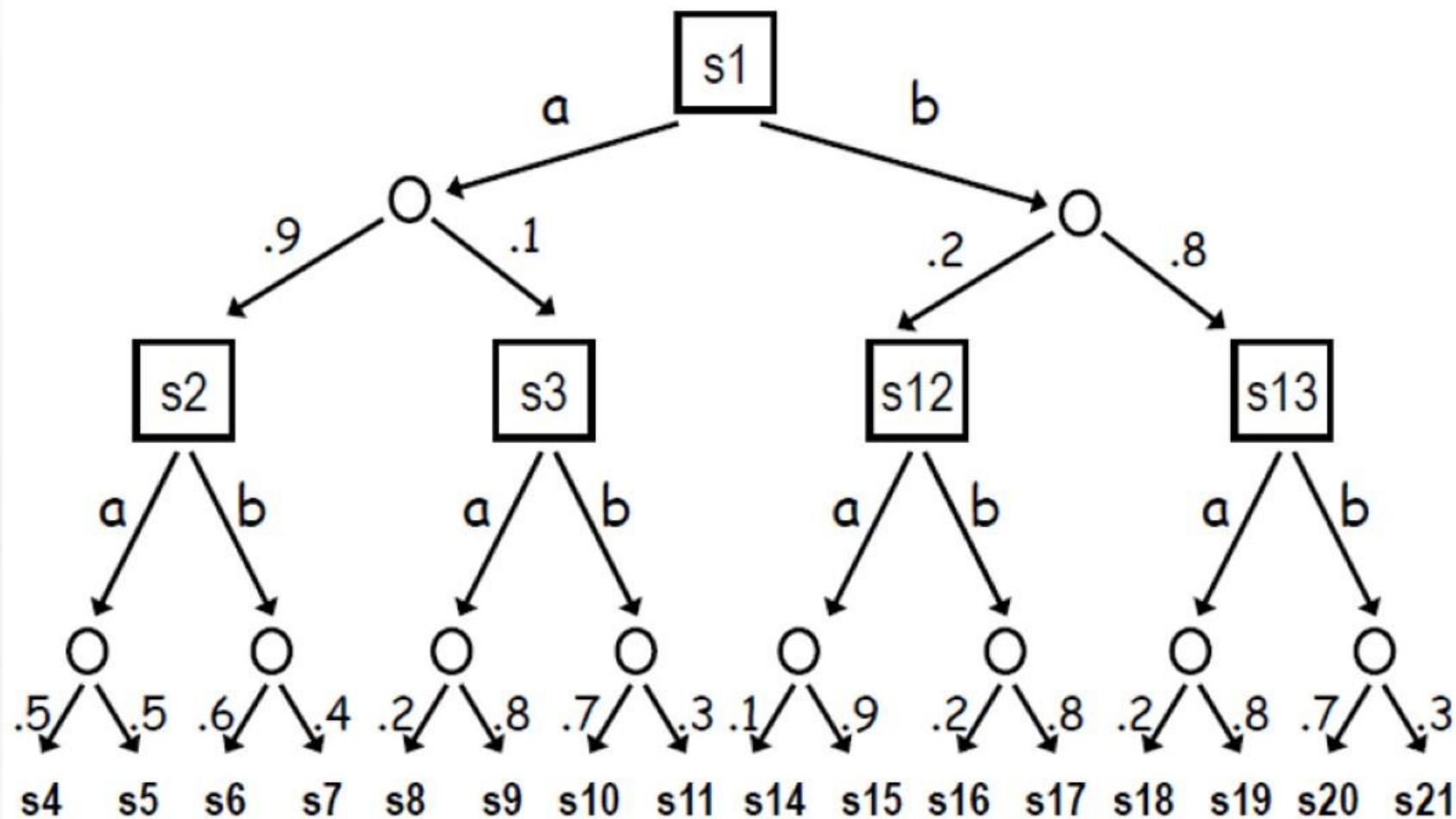
Dyna-Q vs Dyna-Q+ in a changing environment



Planning from current state

- Instead of planning at arbitrary states, plan from the current state
 - This helps improve next action
- Monte Carlo Tree Search

Tree Search



Tractable Tree Search

- Combine 3 ideas:
 - Leaf nodes: approximate leaf values with value of default policy π

$$Q^*(s, a) \approx Q^\pi(s, a) \approx \frac{1}{n(s, a)} \sum_{k=1}^n G_k$$

- Chance nodes: approximate expectation by sampling from transition model

$$Q^*(s, a) \approx R(s, a) + \gamma \frac{1}{n(s, a)} \sum_{s' \sim \Pr(s'|s, a)} V(s')$$

- Decision nodes: expand only most promising actions

$$a^* = \operatorname{argmax}_a Q(s, a) + c \sqrt{\frac{2 \ln n(s)}{n(s, a)}} \quad \text{and} \quad V^*(s) = Q(s, a^*)$$

- Resulting algorithm: Monte Carlo Tree Search

Monte Carlo Tree Search (with upper confidence bound)

UCT(s_0)

```
create root  $node_0$  with state  $state(node_0) \leftarrow s_0$   
while within computational budget do  
   $node_l \leftarrow TreePolicy(node_0)$   
   $value \leftarrow DefaultPolicy(state(node_l))$   
   $Backup(node_l, value)$   
return  $action(BestChild(node_0, 0))$ 
```

TreePolicy($node$)

```
while  $node$  is nonterminal do  
  if  $node$  is not fully expanded do  
    return  $Expand(node)$   
  else  
     $node \leftarrow BestChild(node, C)$   
return  $node$ 
```

Monte Carlo Tree Search (continued)

Expand(*node*)

choose $a \in \text{untried actions of } A(\text{state}(\text{node}))$
 add a new child node' to node
 with $\text{state}(\text{node}') \leftarrow T(\text{state}(\text{node}), a)$
 return node'

deterministic
transition

BestChild(*node*, *c*)

return $\arg \max_{\text{node}' \in \text{children}(\text{node})} V(\text{node}') + c \sqrt{\frac{(2 \ln n(\text{node}))}{n(\text{node}')}}$

DefaultPolicy(*node*)

while node is not terminal do
 sample $a \sim \pi(a | \text{state}(\text{node}))$
 $s' \leftarrow T(\text{state}(\text{node}), a)$
 return $R(s, a)$

Monte Carlo Tree Search (continued)

Single Player

Backup(*node*,*value*)

while *node* is not null do

$$V(\text{node}) \leftarrow \frac{n(\text{node})V(\text{node}) + \text{value}}{n(\text{node}) + 1}$$

$$n(\text{node}) \leftarrow n(\text{node}) + 1$$

$$\text{node} \leftarrow \text{parent}(\text{node})$$

Two Players (adversarial)

BackupMinMax(*node*,*value*)

while *node* is not null do

$$V(\text{node}) \leftarrow \frac{n(\text{node})V(\text{node}) + \text{value}}{n(\text{node}) + 1}$$

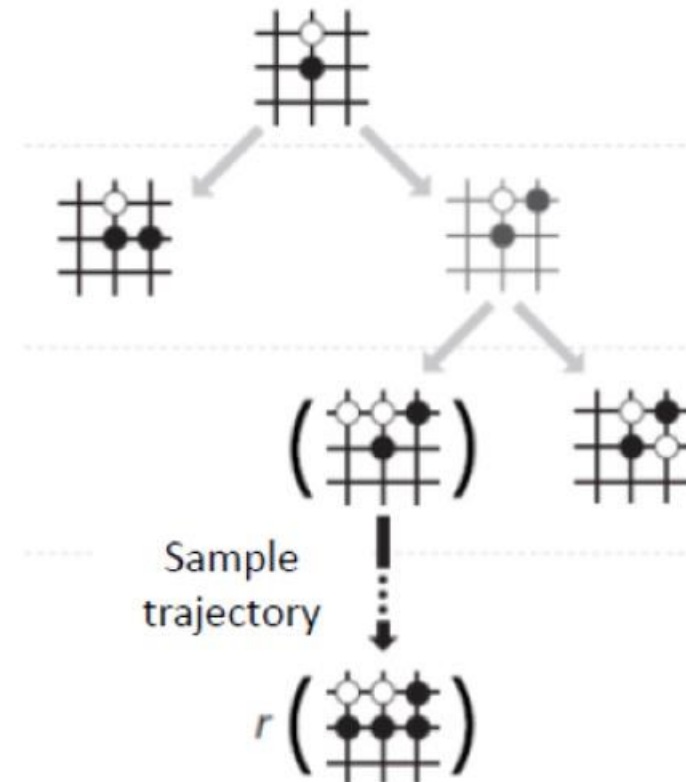
$$n(\text{node}) \leftarrow n(\text{node}) + 1$$

$$\text{value} \leftarrow -\text{value}$$

$$\text{node} \leftarrow \text{parent}(\text{node})$$

Search Tree

- At each edge store
 $Q(s, a), \pi(a|s), n(s, a)$
- Where $n(s, a)$ is the
visit count of (s, a)



Simulation

- At each node, select edge a^* that maximizes

$$a^* = \operatorname{argmax}_a Q(s, a) + u(s, a)$$

- where $u(s, a) \propto \frac{\pi(a|s)}{1+n(s,a)}$ is an exploration bonus

$$Q(s, a) = \frac{1}{n(s,a)} \sum_i 1_i(s, a) [\lambda V_w(s) + (1 - \lambda)G_i]$$

$$1_i(s, a) = \begin{cases} 1 & \text{if } (s, a) \text{ was visited at iteration } i \\ 0 & \text{otherwise} \end{cases}$$

Summary:

- Understand how simulating experience from the model differs from interacting with the environment
- Describe how Tabular Dyna-Q works
- Identify the direct-RL and planning updates in Tabular Dyna-Q
- Explain how having an accurate model allows the agent to learn with fewer environment interactions
- Explain the effects of planning with an inaccurate model
- Explain Monte Carlo Tree Search

Recent Progress in Reinforcement Learning: Focus on Deep RL

- Advancements shaping intelligent decision-making systems today

Core Concepts and Early Algorithms

Markov Decision Processes

MDPs model environments using states, actions, and rewards forming the basis of reinforcement learning.

Early RL Algorithms

Q-learning and SARSA used tabular methods to estimate value functions and derive optimal policies.

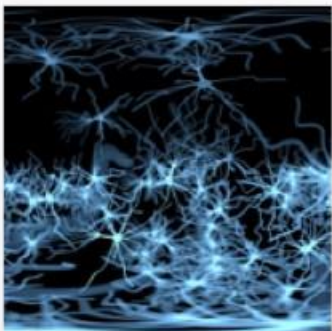
Exploration-Exploitation Tradeoff

Early methods introduced key challenges like balancing exploration and exploitation in learning.

Transition to Deep RL

Neural networks were integrated to overcome scalability limits, enabling deep reinforcement learning advancements.

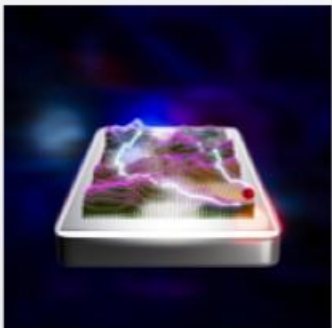
Motivation and Breakthroughs



Addressing RL Limitations

DRL addresses traditional RL limits by handling large, continuous state spaces using deep neural networks.

Deep RL addresses challenges with high-dimensional state spaces that traditional reinforcement learning cannot handle effectively.

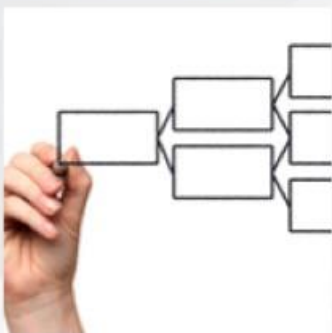


Deep Q-Network Breakthrough

Deep Q-Networks combined convolutional networks with RL to achieve human-level performance on Atari games.

Combining RL with deep neural networks enables agents to learn directly from raw sensory inputs like images and signals.

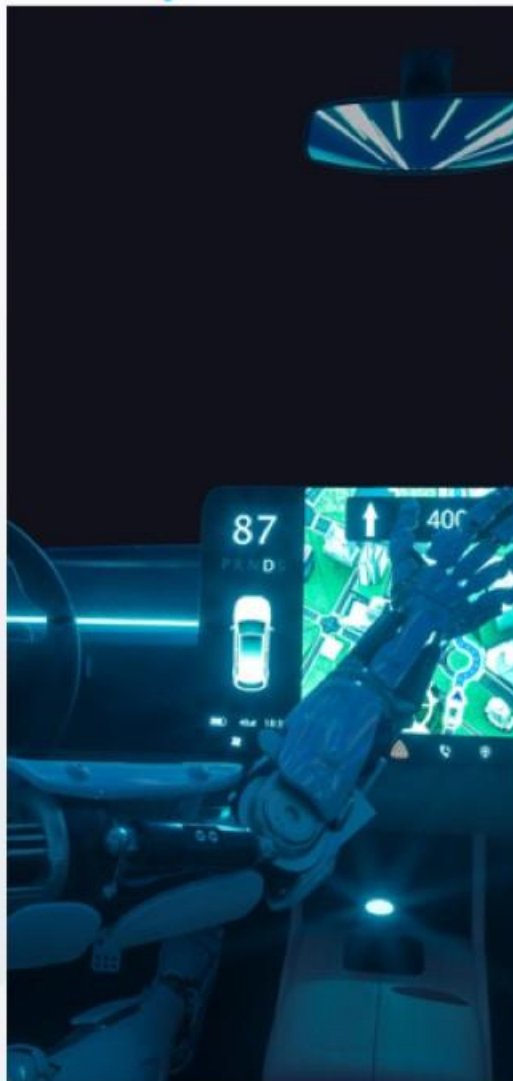
DQN introduced experience replay and target networks to stabilize training, marking a key breakthrough in DRL development.



Advanced Architectures

New architectures like policy gradients and actor-critic frameworks improved DRL stability and sample efficiency.

Key Breakthroughs in DRL



Real-World Use Cases

Gaming Achievements

Deep reinforcement learning powered AI like AlphaGo and AlphaZero that outperformed human champions in Go and Chess.

Robotics Control

DRL enables robots to adaptively control manipulation and locomotion tasks in changing environments.

Autonomous Driving

Reinforcement learning enhances decision-making under uncertainty, improving autonomous vehicle safety and efficiency.

Finance and Healthcare

RL optimizes trading strategies and personalizes treatments like chemotherapy dosing for better outcomes.

Major Algorithmic Advances

Deep Q-Networks Breakthrough

DQN enabled learning from raw pixels, achieving human-level performance on Atari games, marking a major DRL milestone.

Policy Gradient Methods

Policy gradients introduced stochastic policies for continuous action spaces, enhancing DRL's flexibility and power.

Actor-Critic Architectures

Actor-critic models combine value estimation and policy learning, improving stability and performance in DRL.

Advanced Optimization Techniques

Algorithms like PPO and SAC enhance training stability and sample efficiency, enabling complex real-world DRL applications.

Components and Algorithms

ALGORITHM	TYPE	KEY FEATURES
DQN	Value-based	Experience replay, target network
PPO	Policy-based	Clipped objective for stability
SAC	Actor-Critic	Entropy regularization for exploration

Advancements from 2024 to 2025

Continual Reinforcement Learning

Allows agents to learn sequential tasks, enhancing adaptability and long-term knowledge retention.

Model-Based DRL Approaches

Incorporate predictive environment models to increase sample efficiency and decision accuracy.

Hybrid RL Techniques

Combine policy gradients with value-based methods to improve learning stability and performance.

Reinforcement Learning from Human Feedback

Critical for aligning AI models with human preferences, merging reinforcement learning and generative AI.

Challenges and Future Directions

Current Limitations and Opportunities

Challenges in Deep RL

Sample inefficiency and safety concerns limit deep reinforcement learning applications in critical fields.

Sustainability and Computation

High computational costs and energy consumption raise sustainability issues in reinforcement learning.

Future Research Directions

Advancements like hierarchical RL, transfer learning, and sim-to-real adaptation address current limitations.

Ethics and Autonomous Agents

Fairness, transparency, and AI integration with generative models shape future autonomous intelligent agents.

Future Research Directions

Multi-Agent Reinforcement Learning

Multiple agents interact collaboratively or competitively in shared environments to solve complex problems.

Hierarchical Reinforcement Learning

Layered decision-making frameworks enhance scalability for handling complex tasks efficiently.

Offline Reinforcement Learning

Learning from pre-collected datasets reduces dependency on real-time costly simulations.

Sim-to-Real Transfer and AI Integration

Bridging virtual training with physical deployment and combining DRL with generative and language models.

Timeline and Key Milestones

Evolution of RL and DRL



Foundations in the 1950s

Dynamic programming and the Bellman equation established the theoretical basis of reinforcement learning in the 1950s.

Algorithmic Advances in the 1980s

The 1980s introduced Q-learning and temporal difference methods that advanced RL algorithms significantly.

Birth of Deep RL (2013-2015)

Integration of deep learning led to Deep Q-Networks (DQN), marking the beginning of deep reinforcement learning.

Modern Innovations and Future Trends

Recent advances include AlphaGo's success, PPO, SAC algorithms, RLHF, multi-agent systems, and continual learning driving RL forward.



Thank You