###Video: VideoExpected Sarsa

Course 2 - Sample-based Learning Methods

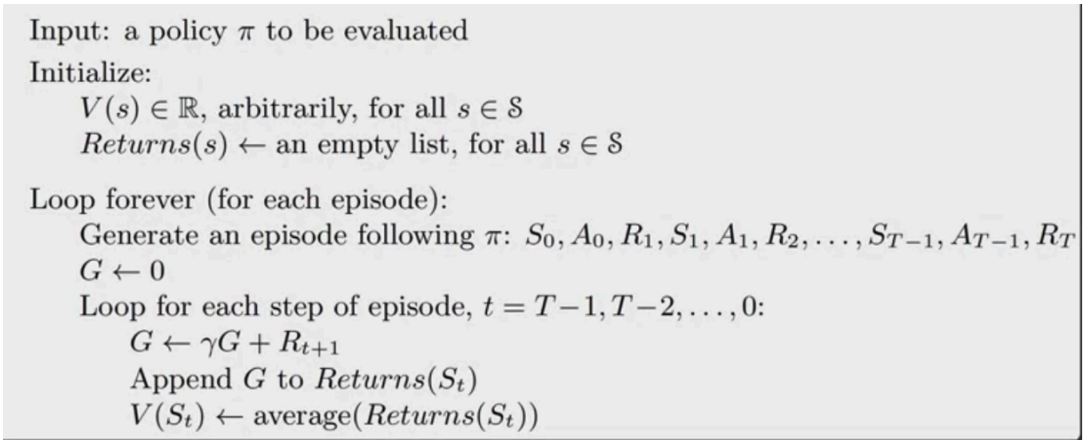
Module 2

Monte Carlo Methods

Monte Carlo methods estimate values by **averaging** over a **large number** of **random samples**. Monte Carlo method for learning a value function would first observe multiple returns from the same state.

It average those observed returns to estimate the expected return from that state

Algorithm for estimating the state value function of a policy



How Monte Carlo Estimates Value Functions

- The agent **observes multiple returns** from the same state and averages them.
- As the number of samples **increases**, the estimated value becomes more accurate.
- **Only works with episodic tasks**, since returns are calculated at the end of episodes.

Differences Between Monte Carlo and Dynamic Programming

- We do not need to keep a **large model** of the environment, but rather learn from experience

- We are estimating the value of an individual state **independently** of the values of other states
- The **computation** needed to update the value of each **state** does **not depend** on the **size** of the **MDP**

Monte-Carlo for Control

Monte-Carlo for Action Value

The method is similar to learning state values: averaging sample returns from a state-action pair.

Learning action values accurately requires trying all possible actions in a given state.

Exploration Strategies

- **Exploring Starts:** Ensures episodes **begin** in **every** possible **state-action pair**, allowing full policy evaluation.
- **Alternative Strategies (e.g., Epsilon-Greedy):** Used when setting starting states manually isn't feasible, especially for stochastic policies.

Exploring starts

- This is a way to maintain exploration.
- In exploring starts, we must guarantee that episodes start in every state-action pair.
- Afterwards, the agent simply follows its policy.

$$s_0, a_0, s_1, a_1, s_2, a_2, \dots$$

Random From π and p

Monte-Carlo for Generalized Policy Iteration

GPI includes a policy evaluation and a policy improvement step. GPI algorithms produce sequences of policies that are at least as good as the policies before them

For the policy improvement step, we can make the policy greedy with respect to the agent's current action value estimates.

For the policy evaluation step, we will use a Monte Carlo method to estimate the action values.

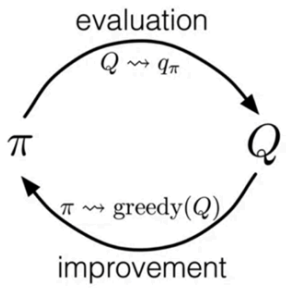
$$\pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \dots$$

Improvement:

$$\pi_{k+1}(s) \doteq \underset{a}{\text{argmax}}\, q_{\pi_k}(s,a)$$

Evaluation:

Monte Carlo Prediction



Monte Carlo method for learning action values

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \dots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$

Steps of the Algorithm

1. **Exploring Starts:** Each episode begins with a **randomly selected state-action pair**, ensuring all possibilities are explored.
2. **Episode Generation:** The agent follows its policy, recording the sequence of **states, actions, and rewards**.
3. **Computing Returns:** At the end of the episode, **returns are calculated** for each state-action pair by summing future rewards with discounting rate.
4. **Updating Action Values:** The **average return** for each state-action pair is computed and used to refine **action value estimates**.
5. **Policy Improvement:** The policy is updated to take the **greedy action**—choosing the best action based on **updated action values**.
6. **Iteration & Convergence:** This cycle continues until the policy **optimally selects actions** across all states.

Exploration Methods for Monte Carlo

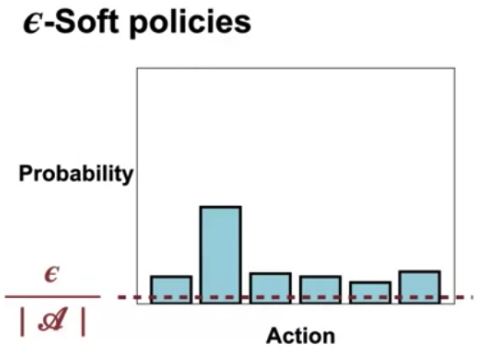
Epsilon-soft policies

Limitations of Exploring Starts

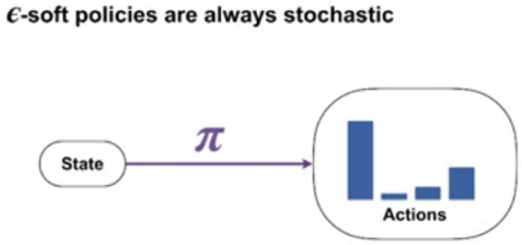
- **Exploring Starts** require starting from every possible **state-action pair**, which is often impractical.
- **Example**: Setting up a **self-driving car** in random configurations on a freeway would be unrealistic.

Epsilon greedy policies are a subset of a larger class of policies called Epsilon soft policies.

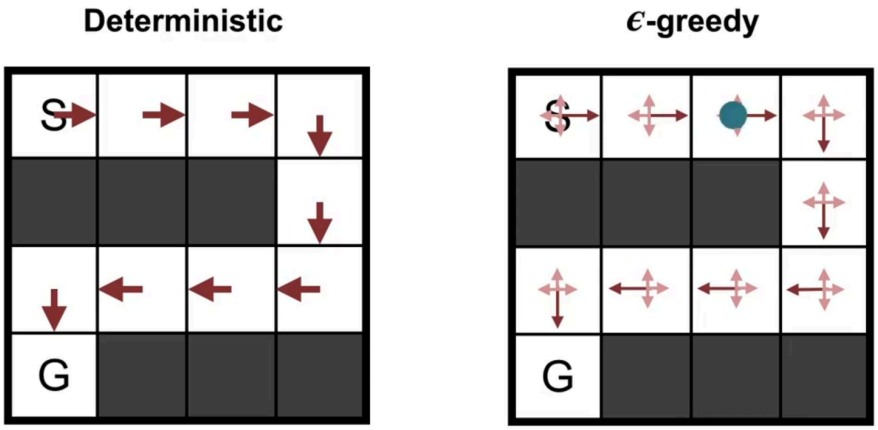
Epsilon soft policies take each action with probability at least ϵ over the number of actions.



ϵ-soft policies are always stochastic (luôn ngẫu nhiên)

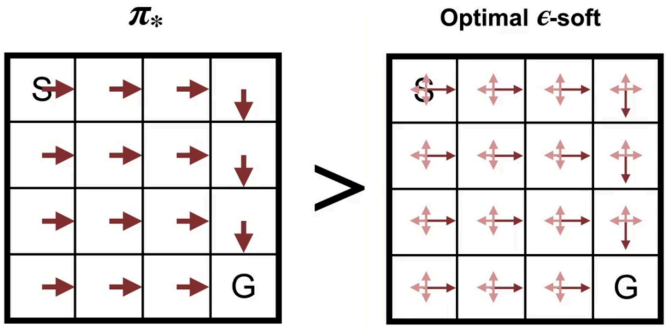


ϵ-greedy policies and deterministic policies



The Epsilon greedy policy has more arrows because every action has some small probability of being selected accordingly. The agent will probably follow a slightly different trajectory every episode.

ϵ-soft policies may not be optimal



Monte Carlo Control with Epsilon-Soft Policies

MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Epsilon-soft policies eliminate the need for **Exploring Starts** in Monte Carlo learning.

The algorithm follows these modified steps:

1. **Start with an Epsilon-soft policy** (e.g., uniform random policy).

2. **Generate an episode** using the Epsilon-soft policy instead of Exploring Starts.
3. **Update action values based on returns** from past experiences.
4. **Improve the policy** by making it **Epsilon-greedy** with respect to learned action values.

Differences Between Exploring Starts & Epsilon-Soft Policies

- **Exploring Starts** find the **optimal deterministic policy** by directly visiting all state-action pairs.
- **Epsilon-soft policies** instead converge to the **optimal Epsilon-soft policy**, which may perform slightly worse but is easier to implement.

Practical Implications

- Although Epsilon-soft policies **may not find the best possible policy**, they allow for safe and flexible exploration.
- Future algorithms like **Q-learning** will build upon these ideas to find **optimal policies**.

Off-policy Learning for Prediction

Off-policy learning is a reinforcement learning method where the agent learns about a **target policy** different from the **behavior policy** used to generate data. This allows learning optimal behaviors without directly following the policy being optimized.

On-Policy vs Off-Policy Learning

- **On-policy:** Agent learns from data generated by the same policy it is improving.
- **Off-policy:** Agent learns from data generated by a different policy (behavior policy), enabling more flexible learning.
- **Target policy (π):** The policy the agent aims to learn or improve.
- **Behavior policy (b):** The policy used to collect experience (may be exploratory or random), must sufficiently explore all actions the target policy might take.

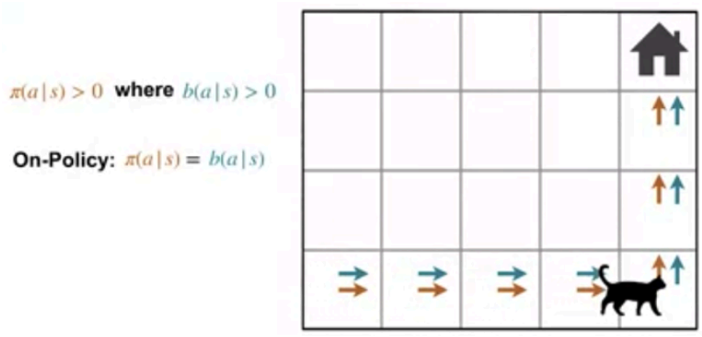
Advantages

- Enables **learning from past experiences or demonstrations** (historical data) collected under different policies.

- Supports **continual exploration** and better coverage of state-action space.
- Allows **sample reuse** and improves **data efficiency**.
- Facilitates **parallel learning** of multiple policies.
- Useful in real-world domains where exploration is costly or risky (e.g., healthcare, robotics, finance).

Key Requirements

- The behavior policy must **cover the target policy's action space** to ensure effective learning.
    - If the target policy assigns a non-zero probability to an action in a given state, the behavior policy must also allow that action. (Note: off policy learning is a strict generalization of on policy learning on policies the specific case where the target policy is equal to the behavior policy)



- Importance sampling or other correction methods are often used to handle the mismatch between behavior and target policies.

Importance Sampling

Derivation of Importance Sampling

Sample: $x \sim b$

Estimate: $\mathrm{E}_{\pi}[X]$

Cause x is sampled from b, cannot use the sample average to compute the expectation under Pi

$$
\begin{aligned}
\mathrm{E}_\pi[X] &\doteq \sum_{x \in \mathrm{X}} x\pi(x) \\
&= \sum_{x \in \mathrm{X}} x\pi(x)\frac{b(x)}{b(x)} \\
&= \sum_{x \in \mathrm{X}} x\frac{\pi(x)}{b(x)}b(x) \\
&= \sum_{x \in \mathrm{X}} x\rho(x)b(x)
\end{aligned}
$$

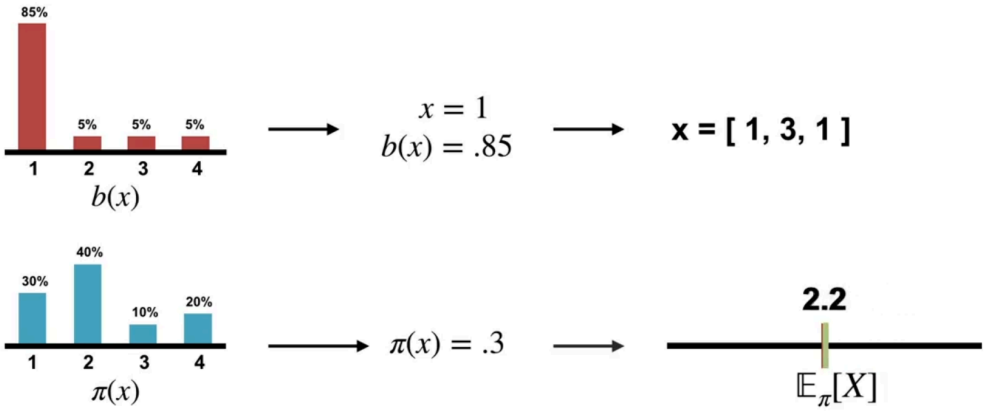
$\rho(x)$ is importance sampling ratio

$$
\mathrm{E}_\pi[X] = \sum_{x \in \mathrm{X}} x\rho(x)b(x) \qquad \rho(x) = \frac{\pi(x)}{b(x)}
$$

$$
= \mathrm{E}_b[X_\rho(X)] \approx \frac{1}{n}\sum_{i=1}^{n} x_i\,\rho(x_i)
$$

$$
x_i \sim b
$$

Example:



$$
\frac{1}{n}\sum_{1}^{n} x\rho(x) \rightarrow \frac{(1 \times \frac{.3}{.85}) + (3 \times \frac{.1}{.05}) + (1 \times \frac{.3}{.85})}{3} \approx 2.24
$$

Off-Policy Monte Carlo Prediction

$$
\begin{aligned}
v_\pi(s) &\doteq \mathrm{E}_\pi\left[G_t \mid S_t = s\right] \\
&\approx \text{average}(\\
&\qquad \rho_0\text{Returns}[0], \\
&\qquad \rho_1\text{Returns}[1], \\
&\qquad \rho_2\text{Returns}[2] \\
&\quad)
\end{aligned}
$$

$$\rho = \frac{\text{P(trajectory under } \pi)}{\text{P(trajectory under } b)}$$

$$v_\pi(s) = \text{E}_b\left[\rho G_t \mid S_t = s\right]$$

Probability distribution over episodic trajectories

Probability of starting in state S_t, taking action A_t, transitioned into state S_{t+1}, and so on ... until reaching terminal state S_T (T is terminal timestep at the end of each tasks episode) under behavior policy b

$$P(A_t, S_{t+1}, A_{t+1}, \dots, S_T \mid S_t, A_{t:T}) =$$

$$b(A_t \mid S_t)\, p(S_{t+1} \mid S_t, A_t)\, b(A_{t+1} \mid S_{t+1})\, p(S_{t+2} \mid S_{t+1}, A_{t+1}) \;\dots\; p(S_T \mid S_{T-1}, A_{T-1})$$

$$\text{P}(\textbf{trajectory under } b) = \prod_{k=t}^{T-1} b(A_k \mid S_k)\, p(S_{k+1} \mid S_k, A_k)$$

$$\begin{aligned}
\rho_{t:T-1} &\doteq \frac{\text{P(trajectory under } \pi)}{\text{P(trajectory under } b)} \\
&\doteq \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)\, p(S_{k+1} \mid S_k, A_k)}{b(A_k \mid S_k)\, p(S_{k+1} \mid S_k, A_k)} \\
&\doteq \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}
\end{aligned}$$

$$\text{E}_b\left[\rho_{t:T-1} G_t \mid S_t = s\right] = v_\pi(s)$$

Off-policy every-visit MC prediction, for estimating $v \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

- $V(s) \in \text{R}$, **arbitrarily, for all** $s \in \text{S}$
- $\text{Returns}(s) \leftarrow$ **an empty list**, for all $s \in \text{S}$

Loop forever (for each episode):

- **Generate an episode following** b: $S_0, A_0, R_1, S_1, \dots, S_{T-1}, A_{T-1}, R_T$
- $G \leftarrow 0 \quad W \leftarrow 1$
- **Loop for each step of episode,** $t = T-1, T-2, \dots, 0$
  - $G \leftarrow \gamma W G + R_{t+1}$

- **Append** G **to** $\text{Returns}(S_t)$
- $V(S_t) \leftarrow \textbf{average}(\text{Returns}(S_t))$
- $W \leftarrow W \frac{\pi(A_t \mid S_t)}{b(A_t \mid S_t)}$

Video: Emma Brunskill: Batch Reinforcement Learning

Domain that require real human-interacting data (clinic, social); need carefully collection

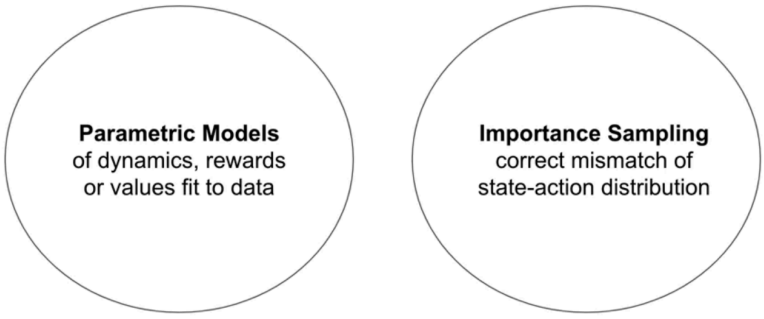
How to minimize data the agent need to learn

Issues:

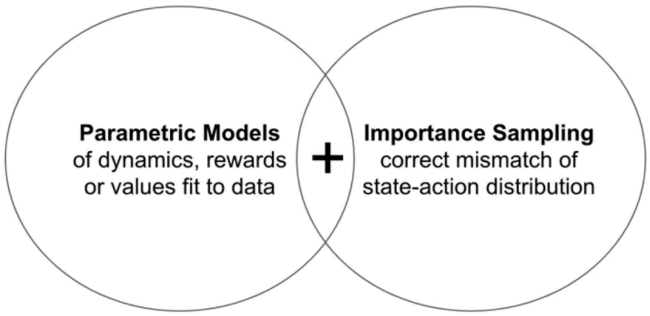
- Collect enough possible scenarios (state-action pair) for agent to learn
- Generalization

Policy Evaluation techniques issues:

- Importance sampling (for distribution mismatching (the existed behavior policy state-action distribution is different from the evaluating/learning target policy's):
  - High variance (although produces an unbiased estimator) causes by too small amount of data
- Parametric model:
  - Lower variance
  - Bias (unless realizable)



- **Doubly Robust (best of both worlds):**



- Smaller variance than importance sampling
- Unbiased if either model realizable or behavior policy known

Module 3

Temporal Difference (TD) learning

Temporal Difference (TD) learning is a reinforcement learning technique that combines elements of Monte Carlo methods and dynamic programming. It is a model-free approach used to estimate value functions or policies directly from experience, without requiring a model of the environment's dynamics.

Review: Estimating Values from Returns

$$v_\pi(s) \doteq \mathrm{E}_\pi\left[G_t \mid S_t = s\right]$$

$$V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right]$$

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

The updated sum $V(S_t)$ is called TD target

TD-error (Temporal Difference Error)

The TD-error is the difference between the expected return and the current estimate. The agent updates the value estimate towards a better estimate of the true value, based on the difference between observed rewards and the predictions made by the current estimate.

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

Where:

- δ_t is the TD error at time step t.
- r_{t+1} is the immediate reward received after taking an action from state s_t and transitioning to state s_{t+1}.

- γ is the discount factor, which represents the importance of future rewards relative to immediate rewards.
- $V(s_t)$ is the estimated value of state s_t at time step t.
- $V(s_{t+1})$ is the estimated value of state s_{t+1} at time step $t+1$.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated

Algorithm parameter: step size $\alpha \in (0, 1]$

Initialize $V(s)$, for all $s \in \mathrm{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop for each episode:

- **Initialize** S
- **Loop for each step of episode:**
  - $A \leftarrow$ action given by π for S
  - Take action A, observe R, S'
  - $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
  - $S \leftarrow S'$
- until S is terminal

Video: Rich Sutton: The Importance of TD Learning

1. What is TD Learning?

- TD Learning is a **prediction-based learning method**.
- It is **highly scalable** with computation, making it crucial for AI.
- Unlike supervised learning, TD Learning does not require labeled training sets or predefined objectives.

2. Prediction Learning vs. Supervised Learning

- **Prediction learning** involves making a prediction, waiting for the outcome, and learning from the result.
- **Supervised learning** requires a teacher or dataset providing correct answers.
- Sutton describes prediction learning as **"unsupervised supervised learning"** because it learns from real-world outcomes.

3. Biological Relevance

- TD Learning is fundamental in **neuroscience**.
- It plays a major role in **reward systems in the brain**.
- The **dopamine neurotransmitter** carries the TD error signal, driving learning.
- Other species, like bees, use **octopamine** for similar learning processes.

4. AI vs. Neuroscience Perspectives

- In AI, TD Learning is valued for its **powerful and widely applicable learning capabilities**.
- In neuroscience, TD Learning is celebrated for its **accurate modeling of animal learning behavior**.

5. Conclusion

- **Prediction learning is the key to AI**.
- **TD Learning is a specialized method** that leverages the temporal structure of learning.
- It is **important in both AI and neuroscience**, making it a crucial concept in understanding intelligence.

Advantages of TD

1. Introduction to TD Learning

- **Temporal Difference (TD) Learning** combines ideas from **dynamic programming** and **Monte Carlo methods**.
- Like **dynamic programming**, TD uses **bootstrapping** (updating estimates based on other estimates).
- Like **Monte Carlo**, TD learns **directly from experience**.

2. Real-Life Example: Predicting Travel Time

- Example: Predicting how long it takes to drive home.
- Predictions adjust dynamically based on **new observations** (e.g., rain, traffic).
- Learning happens **incrementally**, rather than waiting for the final outcome.

3. Comparison: Monte Carlo vs. TD Learning

- **Monte Carlo methods** update estimates **only at the end of an episode**.
- **TD Learning updates estimates at every step**, making it more efficient.
- TD allows **online learning**, meaning updates happen **without waiting for the final result**.

4. Advantages of TD Learning

- **Does not require a model** of the environment (unlike dynamic programming).
- **Learns directly from experience**.
- **Updates values at every step** (unlike Monte Carlo).
- **Bootstrapping improves efficiency**.
- **Converges faster** than Monte Carlo methods.

Video: Comparing TD and Monte Carlo

1. Introduction

- Temporal Difference (TD) Learning and Monte Carlo (MC) methods are compared in a scientific experiment.
- The goal is to evaluate their empirical benefits in estimating value functions.

2. Experiment Setup

- A **five-state Markov Decision Process (MDP)** is used.
- The agent follows a **uniform random policy**, starting in state C.
- Episodes terminate on the left or right, with a reward of **+1 for right termination**.
- The **discount factor is set to 1**, meaning values represent the probability of terminating on the right.

3. TD vs. Monte Carlo Updates

- **TD updates values at each step**, using the next state's value.
- **Monte Carlo updates values only at the end of an episode**, averaging returns.
- TD initially updates fewer states but eventually adjusts values at every step.

4. Learning Speed and Accuracy

- TD estimates **converge faster** toward true values.
- Monte Carlo waits until the episode ends, making it **slower to adjust**.
- TD performs **consistently better**, reducing error more efficiently.

5. Impact of Learning Rate

- **Higher learning rates** speed up convergence but result in **higher final error**.
- **Lower learning rates** slow learning but achieve **better final accuracy**.

6. Conclusion

- TD Learning **converges faster** and achieves **lower final error** than Monte Carlo in this experiment.

Video: Andy Barto and Rich Sutton: More on the History of RL

1. Early Days of Reinforcement Learning

- Rich Sutton was Andy Barto's first graduate student.
- The field of **reinforcement learning (RL)** was initially unpopular.
- Early research focused on **supervised learning**, overshadowing RL.

2. Distinction Between RL and Supervised Learning

- Early AI research in the **1950s and 1960s** explored reward-based learning but shifted toward **supervised learning**.
- Many researchers misunderstood the difference between **trial-and-error learning** and **error correction**.

3. Struggles and Persistence in RL Research

- RL was **not a recognized field** in the 1980s.
- Sutton and Barto pursued RL despite its **lack of mainstream acceptance**.
- They believed **common-sense RL principles** were being ignored.

4. Influence of Behaviorism

- RL shares elements with **behaviorism**, but pure behaviorists rejected models like **value functions**.
- Sutton learned valuable insights from behaviorist principles, which influenced **Temporal Difference (TD) Learning**.

5. Intrinsic Motivation in Research

- Sutton and Barto were **intrinsically motivated** to study RL, even when it lacked external recognition.
- They avoided **trendy research topics**, focusing on **long-term foundational ideas**.
- Intrinsic motivation plays a role in **skill-building**, similar to how animals develop competencies.

6. Conclusion

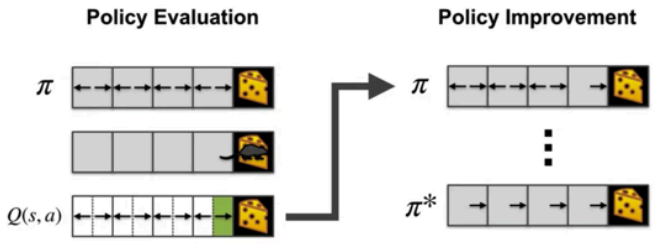
- RL has now achieved **great success**, validating their early persistence.
- Sutton and Barto celebrate the progress of RL as a **recognized and impactful field**.

Module 4

TD for Control

SARSA Algorithm: GPI with Temporal Difference

Monte Carlo does not perform a full policy evaluation step before improvement. Rather, it evaluates and improves after each episode. We could improve the policy after just one policy evaluation step



Sarsa (on-policy TD control) for estimating $Q \approx q_*$

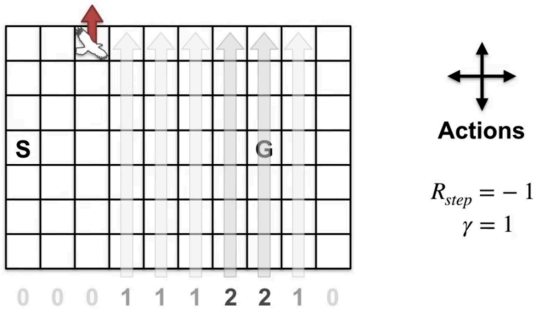
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathrm{S}^{+}, a \in \mathrm{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:

- Initialize S
- Choose A from S using policy derived from Q (e.g., ε-greedy)
- **Loop for each step of episode:**
  - Take action A, observe R, S'
  - Choose A' from S' using policy derived from Q (e.g., ε-greedy)
  - $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
  - $S \leftarrow S'; A \leftarrow A';$
- **until** S is terminal

Video: Sarsa in the Windy Grid World



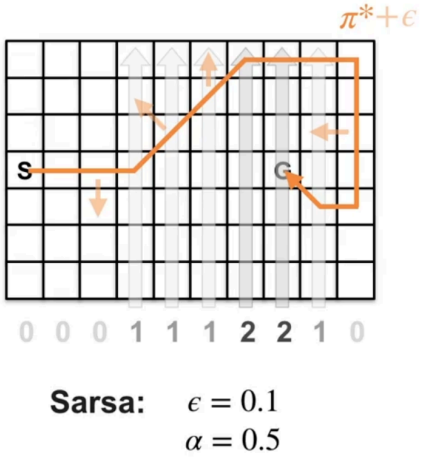
1. **Introduction to Sarsa in Gridworld**
   - The video demonstrates how the **Sarsa control algorithm** operates in a **Markov Decision Process (MDP)**.
   - The goal is to analyze the **performance of a learning algorithm** in a dynamic environment.
2. **Windy Gridworld Setup**
   - The environment consists of a **grid with a start state and a terminal state**.
   - The agent can move in **four directions** and receives a **reward of -1 per step**, encouraging fast escape.
   - A **wind effect** pushes the agent **upward** in certain columns, altering movement dynamics.
3. **Applying Sarsa to the Task**

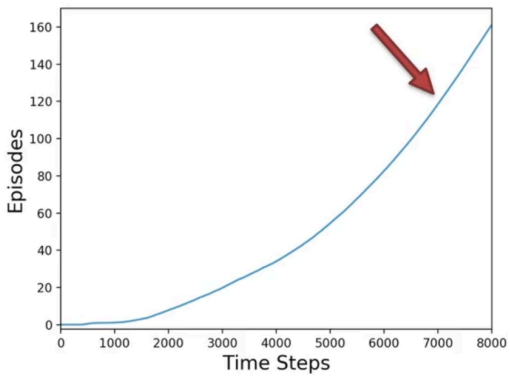
- Uses **epsilon-greedy action selection** with **ε = 0.1**, **α = 0.5**, and **initial values set to 0**.
- **Epsilon-greedy** ensures exploration by taking a **random action 1 in every 10 steps**.
- **Optimistic initialization** encourages systematic exploration.

4. **Performance Analysis**
   - **Early episodes take thousands of steps**, but completion time improves over iterations.
   - Around **7,000 steps**, the **greedy policy stops improving**, stabilizing near optimal.

5. **Comparison with Monte Carlo Methods**
   - **Monte Carlo struggles in this environment** because some policies **never lead to termination**.
   - **Sarsa avoids this trap** by learning during episodes, adjusting policies dynamically.

6. **Conclusion**
   - **Sarsa successfully adapts** to the Windy Gridworld, avoiding pitfalls of deterministic policies.
   - The experiment highlights **Sarsa's ability to learn efficiently in dynamic environments**.





Off-policy TD Control: Q-learning algorithm

Q-Learning is used to learn the optimal action-value function Q∗(s,a), which represents the expected cumulative reward starting from state s and taking action a, under an optimal policy. Q-Learning is an off-policy algorithm (learns from data

generated by an exploratory policy while simultaneously estimating the value of the optimal policy)

The Q-learning algorithm

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathrm{S}^+, a \in \mathrm{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:

- Initialize S
- **Loop for each step of episode:**
  - Choose A from S using policy derived from Q (e.g., ε-greedy)
  - Take action A, observe R, S'
  - $\mathbf{Q(S, A)} \leftarrow \mathbf{Q(S, A)} + \alpha[\mathbf{R} + \gamma \max_{\mathbf{a}} \mathbf{Q(S', a)} - \mathbf{Q(S, A)}]$
  - $S \leftarrow S'$
- **until** S is terminal

Revisiting Bellman equations

- **Sarsa** is a **sample-based** version of **policy iteration**, using **standard Bellman equations** for action values under a **fixed policy**.
- **Q-learning** is a **sample-based** version of **value iteration**, applying the **Bellman Optimality Equation** to continually improve the value function.

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

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

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

$$q_*(s, a) = \sum_{s', r} p(s', r \mid s, a)\big(r + \gamma \max_{a'} q_\pi(s', a')\big)$$

Q-learning **converges to the optimal value function** if the agent **continues to explore** and samples all areas of the **state-action space**.

Q-Learning vs SARSA

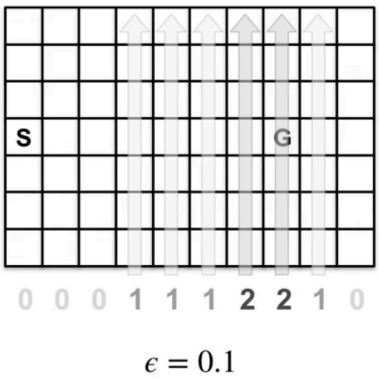
Convergence:

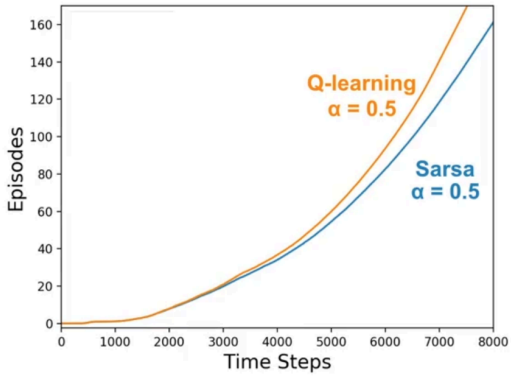
- SARSA: SARSA converges to a policy that is ε-optimal under the current policy being evaluated.
- Q-Learning: Q-Learning converges to the optimal action-value function $Q*(s,a)$ and subsequently the optimal policy $\pi*(s)$.

Safety:

- SARSA: SARSA is generally safer to use in environments where exploration may lead to dangerous actions, as it learns under the current policy.
- Q-Learning: Q-Learning may take riskier actions during exploration, as it learns under an exploratory policy while simultaneously estimating the optimal policy.

Video: Q-learning in the Windy Grid World

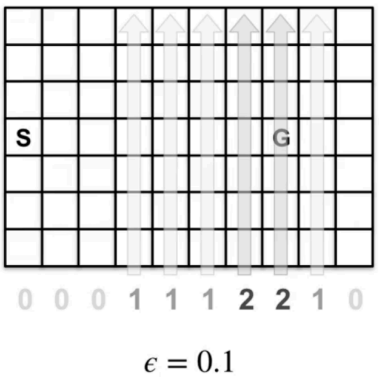


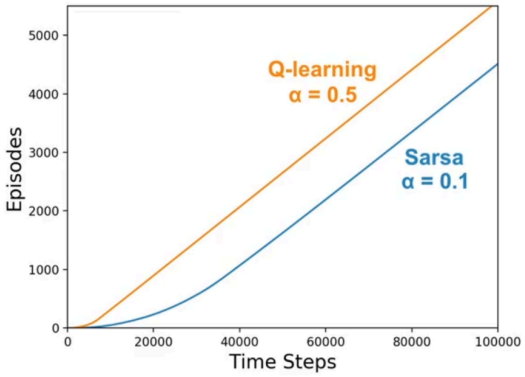


1. **Introduction to Q-learning in Gridworld**
    - The video demonstrates **Q-learning** in a **Markov Decision Process (MDP)**.
    - The environment includes **wind effects**, pushing the agent **upward** in certain columns.
2. **Comparison: Q-learning vs. Sarsa**
    - **Sarsa** learns slowly at first, then improves exponentially before plateauing.
    - **Q-learning** directly learns the **optimal policy's value function**, potentially leading to faster learning.
3. **Experimental Results**

- Initially, **both algorithms learn at a similar pace**.
- **Q-learning achieves a better final policy**, possibly due to its **stable update target**.
- **Sarsa's updates fluctuate** due to exploratory actions affecting its next action value estimate.

4. **Improving Sarsa's Performance**
   - **Reducing the step-size parameter (α = 0.01)** slows learning but leads to a **better final policy**.
   - **Both algorithms converge to the same policy** when trained longer, as indicated by equal slopes in performance graphs.
5. **Key Takeaways**
   - **Q-learning learns faster** but **Sarsa can reach the same final policy** with proper tuning.
   - **Parameter choices (α, ε, initial values, experiment length) significantly impact learning outcomes**.





Video: How is Q-learning off-policy?

1. **Q-learning is an Off-Policy Algorithm**
   - Unlike **Sarsa**, which learns based on the **actions it actually takes**, Q-learning **learns about the best possible actions** in each state.
   - Q-learning **bootstraps off the maximum action value** in the next state, rather than the action chosen by the behavior policy.
2. **Difference Between Target Policy and Behavior Policy**
   - **Target policy**: Always **greedy** with respect to current values.

- **Behavior policy**: Can be **anything** that ensures all state-action pairs are visited (e.g., **epsilon-greedy**).
- Since the **target and behavior policies differ**, Q-learning is **off-policy**.

3. **Why Q-learning Doesn't Use Importance Sampling**
   - Importance sampling is used when estimating values under a **different policy** than the one generating data.
   - Q-learning **directly estimates action values**, so it **does not need importance sampling ratios** to correct for policy differences.
4. **Cliff Walking Example: Q-learning vs. Sarsa**
   - **Q-learning** learns the **optimal policy**, which moves **right along the cliff** but occasionally falls due to **epsilon-greedy exploration**.
   - **Sarsa** accounts for **exploratory actions**, choosing a **safer but longer path** to avoid falling off the cliff.
   - **Sarsa reaches the goal more reliably**, while **Q-learning optimizes for speed but risks failure**.
5. **Key Takeaways**
   - **Q-learning is off-policy** because it learns about the **best possible actions**, not the ones actually taken.
   - **Sarsa is on-policy**, learning based on the **actions it follows**.
   - **Off-policy vs. on-policy learning affects performance**, depending on the environment and task.

Expected Sarsa

Video: Expected Sarsa

Expected SARSA is a variant of the SARSA algorithm that enhances learning by taking the expected value of action selection instead of selecting a single action deterministically. This approach accounts for the stochastic nature of the policy, providing a smoother and more stable update process compared to SARSA.

Expected Sarsa Q-function Update Equation

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathrm{E}_\pi [Q(S_{t+1}, A_{t+1}) | S_{t+1}] - Q(S_t, A_t) \right]$$

$$\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

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

$Q(S_{t+1}, a') =$	0.0	−1.0	2.0	1.0	
$\pi(a' \mid S_{t+1}) =$	0.1	0.1	0.7	0.1	
$\sum_{a'} \pi(a' \mid S_{t+1}) Q(S_{t+1}, a') =$	0.0	− 0.1	+ 1.4	+ 0.1	= 1.4

Video: Expected Sarsa in the Cliff World

1. **Introduction to Expected Sarsa**
   - Expected Sarsa is a **variant of Sarsa** that explicitly averages over the randomness in its policy.
   - The video demonstrates its behavior in the **Cliff Walk environment** and compares it to **Sarsa and Q-learning**.
2. **Cliff Walk Environment Recap**
   - **Undiscounted episodic grid world** with a **cliff on the bottom edge**.
   - Falling into the cliff **resets the agent to the start** with a **reward of -100**.
   - Otherwise, the agent receives **-1 reward per step**, encouraging faster completion.
3. **Comparison: Expected Sarsa vs. Sarsa**
   - **Sarsa performs better than Q-learning** because it accounts for **its own exploration**.
   - **Expected Sarsa outperforms Sarsa** for almost all values of **step size (α)**.
   - Expected Sarsa **handles larger α values better** because its updates are **deterministic** in this environment.
4. **Long-Term Behavior and Convergence**
   - **Expected Sarsa's updates remain stable**, unaffected by α in the long run.
   - **Sarsa struggles with large α values**, sometimes **failing to converge**.

- As **α decreases**, Sarsa's performance **approaches Expected Sarsa's**.

5. **Key Takeaways**
   - **Expected Sarsa learns a good policy faster** in the Cliff World.
   - **More robust to large step sizes** compared to Sarsa.
   - **Explicitly averages over policy randomness**, making updates more stable.

Video: Generality of Expected Sarsa

1. **Comparison of TD Control Algorithms**
   - Three TD algorithms for control: **Sarsa, Q-learning, and Expected Sarsa**.
   - **Sarsa and Expected Sarsa** approximate the **same Bellman equation**.
2. **Expected Sarsa and Off-Policy Learning**
   - Expected Sarsa can perform **off-policy learning without importance sampling**.
   - The expectation over actions is **computed independently** of the action actually selected in the next state.
3. **Expected Sarsa Generalizes Q-learning**
   - If the **target policy is greedy**, Expected Sarsa **only considers the highest value action**.
   - This is equivalent to **computing the max over actions**, just like **Q-learning**.
   - **Q-learning is a special case of Expected Sarsa** when the target policy is greedy.
4. **Key Takeaways**
   - **Expected Sarsa and Q-learning both use expectations over target policies** in their updates.
   - **Expected Sarsa can learn off-policy**, making it more flexible.
   - **Q-learning is simply Expected Sarsa with a greedy target policy**.

Module 5

What is a Model?

Video: What is a Model?

1. **Introduction to Models in Decision-Making**
    - Models help us **predict possible outcomes** before taking actions.
    - They allow **planning**, improving decision-making without direct experience.
2. **Models in Reinforcement Learning**
    - **Sample-based methods (e.g., TD Learning)** rely only on **past experiences**.
    - **Dynamic Programming (DP)** uses **complete knowledge** of the environment.
    - The **Dyner architecture** unifies both approaches for better efficiency.
3. **How Models Work**
    - A model **stores knowledge** about the environment's dynamics.
    - Given a **state and action**, a model predicts the **next state and reward**.
    - This allows **policy improvement** through simulated experience.
4. **Types of Models**
    - **Sample Models**: Generate **random outcomes** based on probabilities (e.g., flipping a coin).
    - **Distribution Models**: Specify the **probability of all possible outcomes** (e.g., listing all coin flip sequences).
5. **Practical Applications**
    - **Sample models are computationally efficient**, producing single outcomes per trial.
    - **CloudFlare used sample models** from lava lamp patterns to encrypt **10% of global internet traffic** in 2017.
    - **Distribution models contain more information** but can be complex and large.

6. **Conclusion**
    - Models **enhance learning efficiency** by reducing direct interactions with the environment.
    - **Sample models vs. distribution models** differ in complexity and information storage.
    - Next, the course will explore **deeper differences between these models**.

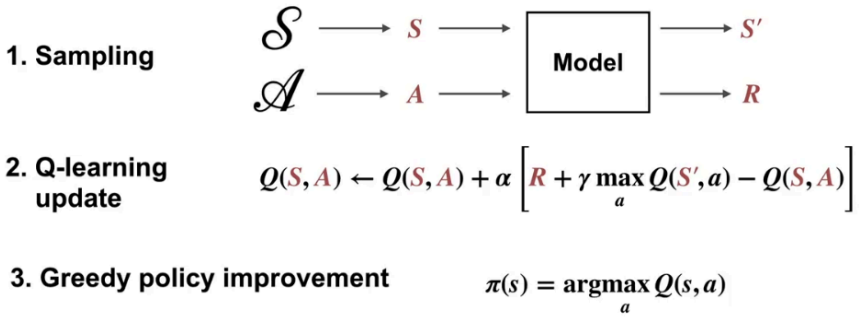
Video: Comparing Sample and Distribution Models

1. **Introduction to Models**
    - Models represent **knowledge of how the world works**.
    - Two types: **Sample models** and **Distribution models**.
2. **Example: Rolling 12 Dice**
    - **Sample model**: Roll a single dice **12 times**, generating **random numbers**.
    - **Distribution model**: Compute **all possible outcomes** and assign **probabilities**.
    - **Scaling issue**: With **one dice**, there are **6 outcomes**; with **12 dice**, over **2 billion outcomes**.
3. **Advantages of Each Model**
    - **Sample models** require **less memory** and are **computationally efficient**.
    - **Distribution models** provide **exact probabilities**, allowing **precise expectation calculations**.
4. **Real-World Applications**
    - **Sample models approximate expected outcomes** by averaging multiple trials.
    - **Distribution models assess risk**, useful in fields like **medicine** (e.g., predicting side effects).
5. **Conclusion**
    - **Sample models are compact**, while **distribution models provide precise probabilities**.

- Each model has **different strengths**, depending on the application.

Planning

Random-sample one-step tabular Q-planning



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

$$\pi(s) = \operatorname{argmax}_a Q(s,a)$$

1. **Introduction to Planning in Reinforcement Learning**
   - Planning uses **models** to improve decision-making **without interacting with the real world**.
   - It leverages **simulated experience** to refine policies.
2. **Q-planning vs. Q-learning**
   - **Q-learning** updates policies using **real-world experience**.
   - **Q-planning** updates policies using **model-generated experience**.
3. **Random-Sample One-Step Tabular Q-planning**
   - Assumes a **sample model** of transition dynamics.
   - **Randomly selects state-action pairs**, queries the model, and performs a **Q-learning update**.
   - Improves policy by **acting greedily with respect to updated values**.
4. **Advantages of Planning Updates**
   - **Executed without interacting with the real world**, reducing risk.
   - Can be performed **in parallel with real-world interactions**, filling waiting time.
5. **Example: Robot Near a Cliff**
   - The robot's **model knows stepping off the cliff is bad**, but its **policy does not reflect this yet**.

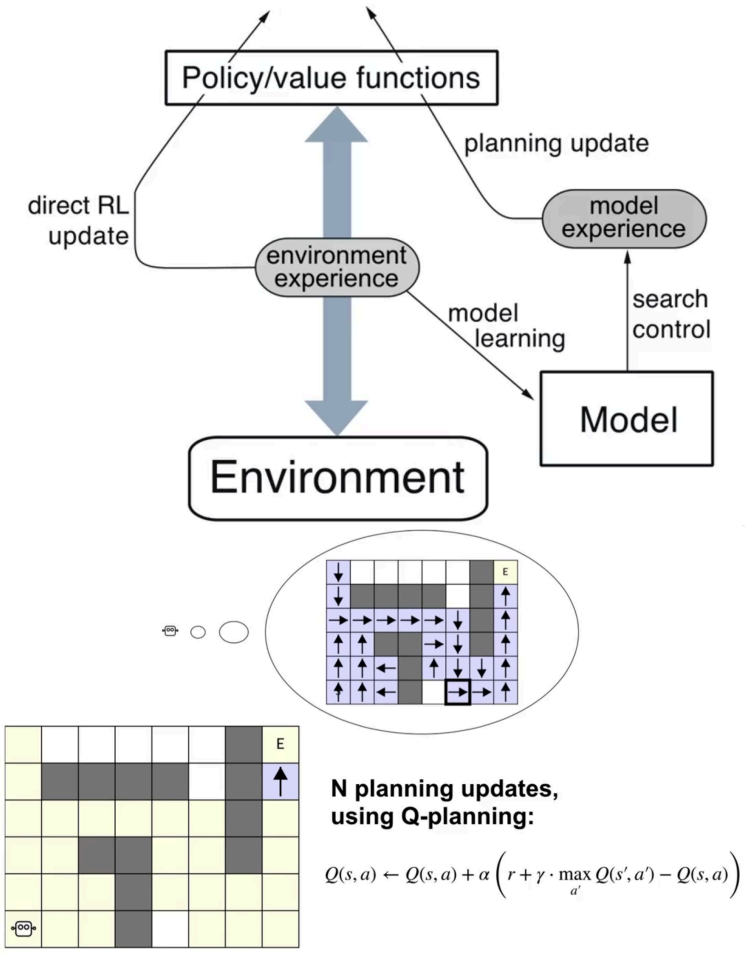
- **Simulated experience** helps the robot **learn to avoid the cliff** before actually encountering it.

6. **Conclusion**

- Planning methods use **simulated experience** to improve policies.
- **Random-sample Q-planning** generates experience and updates values using **Q-learning principles**.

Dyna as a formalism for planning

Video: The Dyna Architecture



1. **Introduction to Dyna Architecture**

- Dyna combines **direct reinforcement learning (RL)** and **planning**.
- The agent learns from **real-world experience** and **simulated experience** generated by a model.

2. **Components of Dyna**

- **Environment & Policy**: Generate real-world experience for direct RL updates.
- **Model**: Learned from environment experience, used to generate simulated experience (planning).
- **Search Control**: Determines which states the agent will plan from.

3. **Example: Robot in a Maze**
   - The robot **starts with no knowledge** and explores randomly.
   - After the first episode, **Dyna uses past experience to build a model**.
   - **Planning replays simulated transitions**, improving the policy **without additional real-world interactions**.
4. **Comparison with Q-learning**
   - **Q-learning requires many episodes** to learn an optimal policy.
   - **Dyna improves policy faster** by using **limited experience more efficiently**.
   - **More computation per step**, but **faster learning** overall.
5. **Conclusion**
   - **Dyna accelerates learning** by combining **direct RL updates** with **planning updates**.
   - The next lecture will discuss **how to implement Dyna** in practice.

Video: The Dyna Algorithm

Tabular Dyna-Q algorithm

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

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state

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

1. **Introduction to Dyna-Q**
    - **Dyna unifies planning, learning, and acting** in reinforcement learning.
    - The video focuses on **Tabular Dyna-Q**, a specific instance of the Dyna architecture.
2. **How Tabular Dyna-Q Learns a Model**
    - Assumes **deterministic transitions** (e.g., a rabbit moving right in state A always reaches state B).
    - The model **memorizes transitions** and assumes they will always unfold the same way.
3. **Dyna-Q Algorithm Steps**
    - **Agent interacts with the environment**, selecting actions via **epsilon-greedy policy**.
    - **Direct RL update**: Performs a **Q-learning update** using observed transitions.
    - **Model learning step**: Stores the **next state and reward** for each state-action pair.
    - **Planning step** (repeated multiple times):
        - **Search control** selects a previously visited state-action pair.
        - **Model query** generates the next state and reward.
        - **Value update** applies a **Q-learning update** using simulated transitions.
4. **Impact of Planning in Dyna-Q**

- **Without planning**, early episodes take **many steps** to reach the goal.
- **With planning**, reward information propagates **across the state space**, improving policy efficiency.
- **Example**:
  - First episode takes **184 steps**.
  - After **two episodes with planning**, the policy requires only **18 steps** to reach the goal.

5. **Conclusion**
   - **Dyna-Q accelerates learning** by mixing **planning, learning, and acting**.
   - **Planning significantly reduces the number of steps needed** to find an optimal policy.

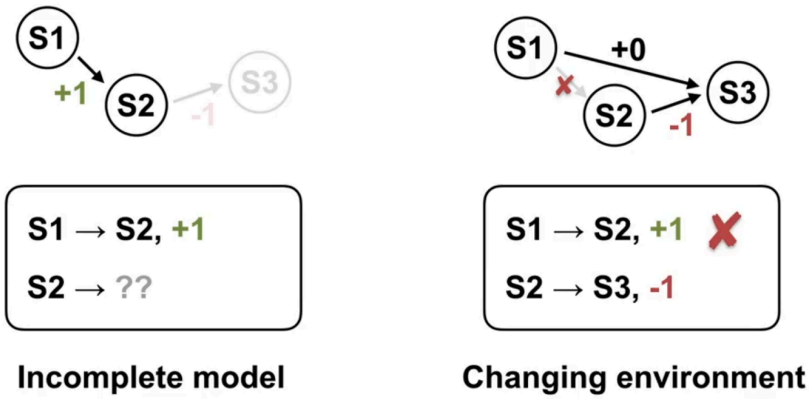
Video: Dyna & Q-learning in a Simple Maze

1. **Introduction to Dyna-Q vs. Q-learning**
   - The experiment compares **tabular Dyna-Q** and **model-free Q-learning** in a **small grid world**.
   - The goal is to analyze how **learning from both real and model experience** impacts performance.
2. **Maze Environment Setup**
   - The agent has **four actions** and receives **+1 reward for reaching the goal**, **0 otherwise**.
   - The problem is **episodic**, with a **discount factor of 0.95**.
   - Three agents are tested, all using **$\alpha = 0.1$, $\varepsilon = 0.1$**, and initializing **action values to 0**.
3. **Performance Comparison: Dyna-Q vs. Q-learning**
   - **Q-learning (0 planning steps)** improves slowly, plateauing at **14 steps per episode**.
   - **Dyna-Q (5 planning steps)** reaches the same performance **much faster**.
   - **Dyna-Q (50 planning steps)** finds a good policy in **just three episodes**, showing **higher sample efficiency**.

4. **Impact of Search Control in Planning**
    - **Random search control** leads to **inefficient updates**, requiring many planning steps.
    - **Ordered search control** significantly reduces planning updates needed.
    - In **larger environments**, random search control becomes even more problematic.
5. **Conclusion**
    - **Planning dramatically speeds up learning**, making Dyna-Q more efficient than Q-learning.
    - **Better search control strategies** can further improve planning efficiency.

Dealing with inaccurate models

Video: What if the model is inaccurate?



1. **Introduction to Model Accuracy**
    - Models help agents **plan using simulated experience**, but they can be **inaccurate**.
    - Inaccuracy occurs when **stored transitions differ from actual environment transitions**.
2. **Types of Model Inaccuracy**
    - **Incomplete Models**: Missing transitions because the agent **hasn't explored certain actions**.
    - **Outdated Models**: Environment changes, making **previously stored transitions incorrect**.

3. **Effects of Planning with an Inaccurate Model**
    - **Incomplete models** prevent planning initially but improve as the agent **gathers more experience**.
    - **Outdated models** lead to incorrect updates, making the **policy worse** relative to the real environment.
4. **How Dyna-Q Handles Incomplete Models**
    - **Dyna-Q only plans using previously visited state-action pairs**.
    - Early in learning, it **reuses the same transitions** frequently.
    - As exploration increases, **planning updates become more evenly distributed**.
5. **Conclusion**
    - **Planning improves policy based on the model, not the real environment**.
    - **Dyna-Q mitigates incomplete models** by focusing on **visited state-action pairs**.

Video: In-depth with changing environments

1. **Introduction to Model Accuracy in Changing Environments**
    - Agents rely on **models to plan**, but models can become **inaccurate when environments change**.
    - Inaccurate models **worsen policy performance**, requiring **exploration to update them**.
2. **Exploration vs. Exploitation Trade-off**
    - Agents must **choose between exploring to update the model** or **exploiting the current model** for optimal policy.
    - **Revisiting states periodically** helps maintain an accurate model.
3. **Dyna-Q+ Algorithm for Exploration**
    - **Adds an exploration bonus** to planning updates, encouraging agents to revisit states.

- Bonus formula: **Kappa × sqrt(Tau)**, where **Tau** is the time since the state-action pair was last visited.
- **Higher Tau values increase the bonus**, driving exploration.

4. **Shortcut Maze Experiment: Dyna-Q vs. Dyna-Q+**
   - Initially, both algorithms perform **similarly**.
   - After a **shortcut appears**, **Dyna-Q+ quickly finds the new path**, while **Dyna-Q struggles**.
   - **Persistent exploration in Dyna-Q+ helps adapt to environment changes faster**.
5. **Conclusion**
   - **Model accuracy affects learning efficiency**, requiring **exploration to maintain correctness**.
   - **Dyna-Q+ improves exploration** by **rewarding visits to rarely explored states**.

Video: Drew Bagnell: self-driving, robotics, and Model Based RL

1. **Introduction to Robotics and Machine Learning**
   - Robotics is undergoing a **major transformation**, from **manipulation to drones to self-driving cars**.
   - **Machine learning is essential** for perception, decision-making, and control in robotics.
2. **Machine Learning in Autonomous Vehicles**
   - **Perception systems** detect and track objects like **vehicles, cyclists, and pedestrians**.
   - **Decision-making in urban environments** is complex due to **context and interactions** with other actors.
3. **Model-Based Reinforcement Learning (RL)**
   - **Models predict future states** based on current state and action.
   - **Model-based RL reduces sample complexity**, requiring **fewer interactions** to learn effectively.

- **Example**: Acrobatic autonomous helicopter control using **continuous state-action models**.

4. **Quadratic Value Function Approximation**
    - Used in **optimal control** since the 1960s, applied to **drones, medical robots, and self-driving vehicles**.
    - **Exact for linear systems**, and **a strong local approximation** for convex problems.
    - Enables **rapid computation of optimal policies** in continuous action spaces.
5. **Dynamic Programming and Bellman Equation in RL**
    - **Backward iteration** refines the model using **Taylor series approximations**.
    - **Differential dynamic programming** improves policy optimization over time.
6. **Conclusion**
    - **Model-based RL accelerates learning** by leveraging **structured models**.
    - Next steps involve **building models to solve RL problems efficiently**.