CPE 490 590: Machine Learning for Engineering Applications

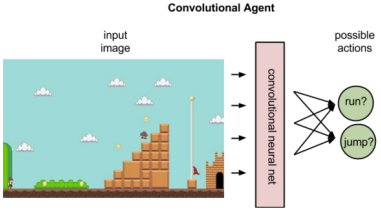
15 Reinforcement Learning

Rahul Bhadani
Electrical & Computer Engineering, The University of Alabama in Huntsville

Outline

Motivation:
Learning in Dynamic Environments

Dynamic Environments



Reinforcement Learning in Robotics

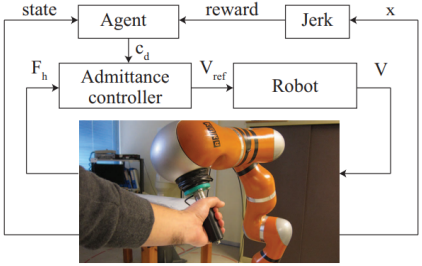


Fig. 1: Reinforcement learning of variable admittance control for human-robot co-manipulation.

Dimeas, Fotios, and Nikos Aspragathos. "Reinforcement learning of variable admittance control for human-robot co-manipulation." 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2015.

Reinforcement Learning in Games

Super Mario: https://youtu.be/OQitI066aI0
Starcraft: https://youtu.be/gEyBzcPU5-w

Class of Learning Problems

Supervised Learning:

- Given training data with previously labeled classes, learn the mapping between the data and their correct classes
- Data: $(\mathbf{x}, \mathbf{y})$
- Goal: learning mapping $\mathbf{x} \rightarrow \mathbf{y}$.
- This is a leaf: 

Unsupervised Learning:

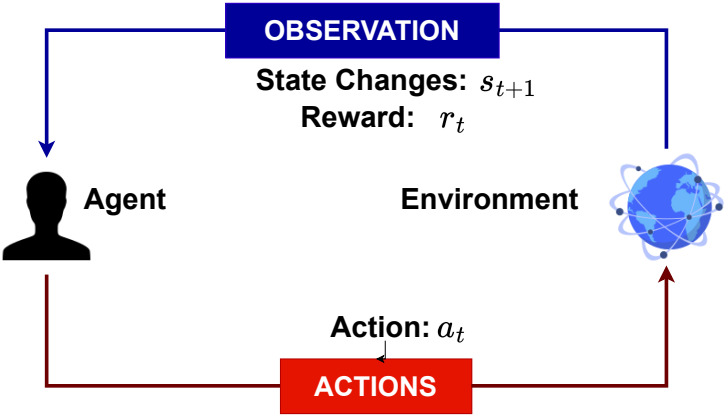
- Given unlabeled data, learn how to group such data into meaningful clusters based on some measure of similarity
- Data: $\mathbf{x}$
- Goal: learn the underlying structure
- These two are similar: 

Reinforcement Learning:

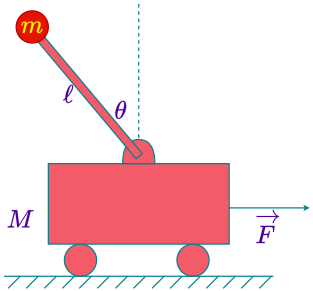
- Given a sequence of outputs, learn a policy to obtain the desired output
- Data: state-action pairs
- Goal: maximize future rewards over many time-steps
- Eat this to stay alive: 

Reinforcement Learning

Reinforcement Learning: A High-level Picture



Reinforcement Learning (RL): Cart-pole Problem



Objective: Balance a pole on the top of a wheeled cart.
State: angle, angular speed, position, horizontal velocity.
Action: horizontal force applied to the car.
Reward: $+10$ at each time step if the pole is upright.

RL: Key Concepts

Agent

that takes actions in the environment and interacts with it and other agents.

Environment

the world in which the agent (or agents) operate(s).

RL: Key Concepts

Action a_t

A move that an agent can make in the environment. Can be continuous (steering angle in car driving) or discrete (chess moves).

RL: Key Concepts

State s_t

A situation that an agent perceives or is in.
Example: current speed of a car, its GPS location: latitude, longitude, acceleration.

RL: Key Concepts

Reward r_t

A feedback by which we measure the success or failure of the agent's action.
Example: Mario jumping to collect coins results in an increased point or gain of an additional lifeline.
Note: Not all actions may result in an immediate reward. It may also be in a delayed fashion.

Total reward/return $R_t = \sum_{i=t}^{\infty} r_i = r_t + r_{t+1} + \cdots + r_{t+n}$.

Discounted reward $R_t = \sum_{i=t}^{\infty} \gamma^i r_i$. This makes future rewards less appealing than the current reward (but also makes infinite sum convergent).

Markov Property

“Markov property: All future states depend on the current state only.”

In other words, the current state completely characterizes the state of the world.

Markov Property

Basic Markov Chain

A standard Markov chain is defined by:

- A set of states $S = \{s_1, s_2, \ldots, s_n\}$
- A transition matrix P where P_{ij} is the probability of transitioning from state i to state j

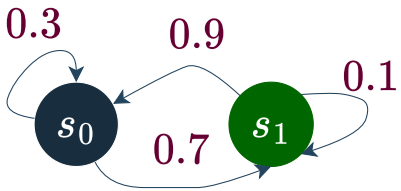
The key property is the Markov property:

$$P(X_{t+1} = s_j | X_t = s_i, X_{t-1} = s_{i-1}, \ldots, X_0 = s_{i_0}) = P(X_{t+1} = s_j | X_t = s_i) = P_{ij}$$

Markov Process/Markov Chain

Definition

A finite state machine where each state is a Markov state (i.e. follows the Markov property).
It consists of a number of states with transition probabilities from one state to another.



Reading: Introduction to Probability Models by Sheldon M. Ross, Chapter 4.

Steady State Distribution Calculation for a Markov Chain

Steady State Distribution

The **steady-state probabilities** (or equilibrium probabilities) of a Markov chain refer to a set of probabilities that remain constant over time, under the condition that the system evolves for a sufficiently long time. These probabilities indicate the proportion of time the process spends in each state in the long run.

Steady State Distribution Calculation for a Markov Chain

Let Ω be the vector of steady-state probabilities, then satisfies the following equation:

$$\begin{aligned} \Omega P &= \Omega \\ \Omega(P - I) &= 0 \\ (P^\top - I)\Omega^\top &= 0 \quad \text{(Writing in the form } Ax = b\text{)} \end{aligned}$$

where P is the transition matrix. It can be solved using the linear algebra solvers. Alternative it is also an Eigenvalue problem and can be solved using standard libraries for Eignevalues.

Now add actions to the state transition.

Markov Decision Process (MDP)

An extension of Markov chain with state transition depending on some action a.

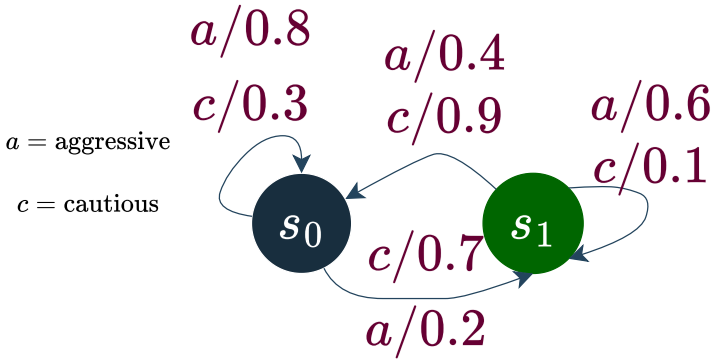
Adding Actions to Markov Chains

When we add actions, we extend the formulation to:

- A set of states $S = \{s_1, s_2, \ldots, s_n\}$
- A set of actions $A = \{a_1, a_2, \ldots, a_m\}$
- A set of transition matrices $\{P^a\}_{a \in A}$ where P^a_{ij} is the probability of transitioning from state i to state j when taking action a

$$P(X_{t+1} = s_j | X_t = s_i, A_t = a) = P^a_{ij}$$

Where A_t is the action taken at time t.


$a/0.8$
$a/0.4$
$c/0.3$
$c/0.9$
$a/0.6$
$c/0.1$
$a =$ aggressive
$c =$ cautious
s_0
s_1
$c/0.7$
$a/0.2$

State Sequences

When generating a sequence of states based on actions, we iteratively apply:

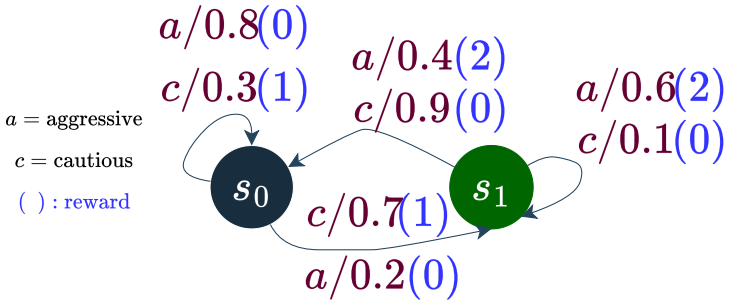
$$X_{t+1} \sim P_{X_t}^{A_t}$$

Where $\sim$ indicates sampling from the probability distribution in row X_t of transition matrix P^{A_t}.

For each time step t:

1. Choose action A_t (predefined or by some policy)
2. Sample next state X_{t+1} from the probability distribution $P_{X_t,\cdot}^{A_t}$
3. Update current state $X_t \leftarrow X_{t+1}$

Now add rewards to the Markov Decision Processes favoring certain actions over another.


$a/0.8(0)$
$c/0.3(1)$
$a/0.4(2)$
$c/0.9(0)$
$a/0.6(2)$
$c/0.1(0)$
a = aggressive
c = cautious
( ) : reward
s_0
s_1
$c/0.7(1)$
$a/0.2(0)$

Markov Decision Process (MDP)

- The transition leads to some corresponding reward.
- An MDP in the context of reinforcement learning is a 5-tuple model $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where
  - $\mathcal{S}$: set of possible states, $s \in \mathcal{S}$ is a state.
  - $\mathcal{A}$: set of possible actions taken while an agent is in a state s, $a \in \mathcal{A}$ is an action.
  - $\mathcal{P}$: a transition probability matrix for defining probabilities of transition to a state s' from a state s under the influence of action a or some other conditional probability density function; written as $\mathcal{P}(s'|s, a)$.
  - $\mathcal{R}$: Distribution of reward given state action pair; a reward $r(s, a) \in \mathcal{R}$ is reward for moving to state s under action a.
  - γ : discount factor, used to emphasize short-term reward over long-term reward; a mathematical trick to make infinite sum finite.

Markov Decision Process

A time step $t = 0$, the environment samples the initial state $s_0 \sim p(s_0)$ (stochastic condition).

For $t = 0$ until done:

- Agent picks an action a_t
- Environment calculates the probable reward $r_t \sim \mathcal{R}(\ \cdot\ |s_t, a_t)$
- Environment samples the next state $s_{t+1} \sim P(\ \cdot\ |s_t, a_t)$
- Agent receives the reward r_t and the next state s_{t+1}
- Increment t.

Reading: Ma, H., Han, S., Hemida, A., Kamhoua, C., & Fu, J. **Adaptive Incentive Design for Markov Decision Processes with Unknown Rewards**.
https://openreview.net/pdf?id=Rwf31BYTAU

Different Types of Rewards

- State-Action Reward $r(s, a)$: reward depends only on the current state and action;
- State-Action-State Reward $r(s, a, s')$: reward depends on the current state, action and the result state;
- State-only Reward $r(s)$: reward depends only on the state you are in.

Policy Function

The policy function, usually denoted by π in the RL literature specifies the mapping from state space $\mathcal{S}$ to Action space $\mathcal{A}$. It tells, what action to perform in each state.

We denote the policy function as $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$. Our ultimate goal in the RL

scenario lies in finding the optimal policy that specifies the correct action to perform in each state, which maximizes the reward.

Episodic and Continuous Tasks

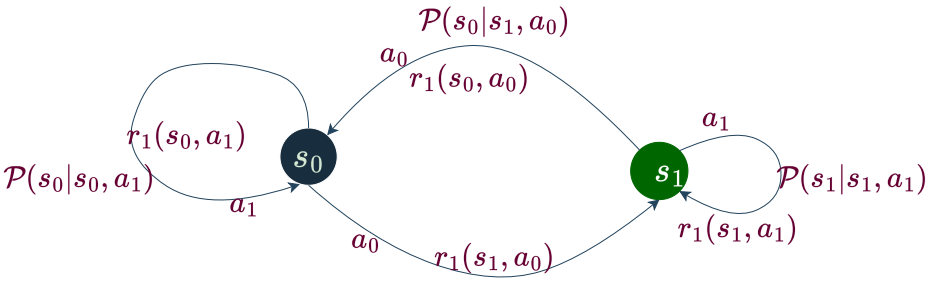
Episodic Tasks

Episodic tasks are the tasks that have a terminal state (end). For example, in a car racing game, the end of the game is a terminal state. Once the game is over, you start the next episode by restarting the game which will be a whole new beginning.

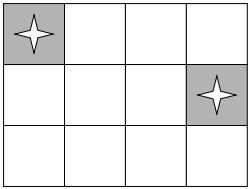
Continuous Tasks

In continuous tasks, there is no terminal state. For example, a robot navigating in a warehouse is a continuous task and there is no defined endpoint as long as operations are required.

Example MDP through State Diagram



A simple MDP

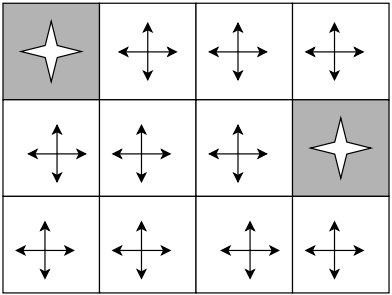


actions $a = \{\uparrow, \downarrow \leftarrow, \rightarrow\}$

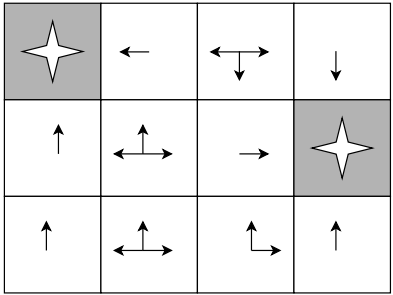
Set a negative reward for each transition, say -1.

Objective: reach one of the terminal states (greyed out) in the least number of actions

A simple MDP



Random Policy



Optimal Policy

Objective in RL

We want to find an optimal policy π^* that maximizes the discounted sum of rewards.

But as there is randomness (due to randomness in the initial state, transition probability, etc.), we say **we want to find an optimal policy π^* that maximizes the expected discounted sum of rewards.**

Formally,

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | \pi\right]$$

How good is a state?

Following a policy, we get a sequence of states and actions (and rewards associated with them): $s_0, a_0, r_0, s_1, a_1, r_1, \cdots$. So how to quantify the goodness of the state the agent arrives?

Value Function

The value function specifies how good it is for an agent to be in a particular state with a policy π. It is an expected cumulative reward following a policy from the state s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^{t} r_t | s_0 = s, \pi\right]$$

Simple Example: 2-State MDP

States:	S1, S2
Actions:	A, B
Discount factor:	$\gamma = 0.9$
Time horizon:	3 steps

Transition Probabilities & Rewards: State S1

When in state S1:

Action A:
80% chance to stay in S1 with reward +1
20% chance to go to S2 with reward +0

Action B:
30% chance to stay in S1 with reward +2
70% chance to go to S2 with reward -1

Transition Probabilities & Rewards: State S2

When in state S2:

Action A:
40% chance to go to S1 with reward +0
60% chance to stay in S2 with reward +1

Action B:
10% chance to go to S1 with reward +5
90% chance to stay in S2 with reward +2

Policy Evaluation

Let's evaluate two different policies:

Policy π_1: Always choose Action A regardless of state

Policy π_2: Always choose Action B regardless of state

Recursive Value Calculation: Policy π_1

For Policy π_1 (always A):

$$V^{\pi_1}(S1) = 1 + 0.9[0.8 \cdot V^{\pi_1}(S1) + 0.2 \cdot V^{\pi_1}(S2)]$$
$$V^{\pi_1}(S2) = 1 + 0.9[0.4 \cdot V^{\pi_1}(S1) + 0.6 \cdot V^{\pi_1}(S2)]$$

$\gamma = 0.9$. Assume initial value of $V^{\pi_1}(S1) = 0$ and $V^{\pi_1}(S2) = 0$.

Starting from S1, over 3 steps: $V^{\pi_1}(S1) \approx 2.71$

Recursive Value Calculation: Policy π_2

For Policy π_2 (always B):

$$V^{\pi_2}(S1) = 2 + 0.9[0.3 \cdot V^{\pi_2}(S1) + 0.7 \cdot V^{\pi_2}(S2)]$$
$$V^{\pi_2}(S2) = 2 + 0.9[0.1 \cdot V^{\pi_2}(S1) + 0.9 \cdot V^{\pi_2}(S2)]$$

$\gamma = 0.9$. Assume initial value of $V^{\pi_1}(S1) = 0$ and $V^{\pi_1}(S2) = 0$.

Starting from S1, over 3 steps: $V^{\pi_2}(S1) \approx$

Finding the Optimal Policy

Policy π_1 (always A) yields $V^{\pi_1}(S1) \approx 2.71$

Policy π_2 (always B) yields $V^{\pi_2}(S1) \approx$

Therefore, $\pi^* =$ is our optimal policy

How good is a state action-pair?

In addition, we also need to quantify the goodness of a state-action pair.

Q Function

The Q function maps each state-action pair to a real number, representing the expected reward over the long run starting from that state and taking that action.

$$Q^{\pi}(s,a) = \mathbb{E}\left[\gamma^{t} r_t | s_0 = s, a_0 = a, \pi\right]$$

Bellman Equation

Q^* is the optimal Q-value which is the maximum expected cumulative reward achievable from the given state-action pair under the given policy:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Q^* satisfies a recursive relation called **Bellman Equation:**

$$Q^*(s,a) = \mathbb{E}_{s_{t+1} \sim P(\cdot|s,a)}\left[r + \gamma \max_{a'} Q^*(s_{t+1}, a_{t+1}) | s, a\right]$$

which essentially tells that if I know the optimal state-action pair for the next time-step, quantified using $Q^*(s_{t+1}, a_{t+1})$, then the optimal strategy is to take the action that maximizes the reward $r + \gamma Q^*(s_{t+1}, a_{t+1})$.
$s_{t+1} \sim P(\cdot|s,a)$ tells you that the next state is chosen stochastically as per the transition probability.

Optimal Policy and Bellman Equation

Hence, the optimal policy π^* corresponds to taking the best action in any state as specified by Q^*.

Solving for Optimal Policy

We use **value iteration algorithm** to solve for an optimal policy iteratively as follows:

$$Q_{i+1} = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s_{t+1}, a_{t+1}) | s, a\right]$$

until $Q_{i+1} \approx Q_i$, i.e. when convergence happens. In this case, we initialize $Q_0(s, a)$ to some arbitrary values for the given state-action pair.

Solving for Optimal Policy: Use Dynamic Programming

The recursion is solved using dynamic programming where a task is divided into multiple smaller tasks and their solutions are recursively combined to get the final solution.

A similar recursive equation can be constructed if we just work with the value function rather than the Q-function.

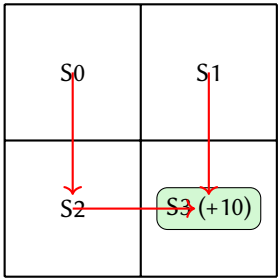
An Example Environment

Grid World Setup

- 4 states (S0, S1, S2, S3)
- S3 is terminal state with reward +10
- All other transitions: reward 0
- Discount factor $\gamma = 0.9$

Available Actions

Up, Down, Left, Right



Transition Model

Stochastic Environment

- Actions succeed with probability 0.8
- Actions fail (move in random other direction) with probability 0.2
- Moving into a wall keeps agent in same state

Reminder

The Q-function represents expected long-term reward when taking action a in state s.

Value Iteration Process

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s_{t+1}, a') \mid s, a\right]$$

Starting condition: $Q_0(s,a) = 0$ for all state-action pairs

Iteration 1

$$\begin{aligned} Q_1(\text{S2}, \text{Right}) &= 0.8 \times (10 + 0.9 \times 0) \\ &\quad + 0.2 \times (0 + 0.9 \times 0) \\ &= 8.0 \end{aligned}$$

Iteration 1

$$\begin{aligned} Q_1(\text{S0}, \text{Down}) &= 0.8 \times (0 + 0.9 \times 0) \\ &\quad + 0.2 \times (0 + 0.9 \times 0) \\ &= 0.0 \end{aligned}$$

Continue Value Iteration

Iteration 2

$$Q_2(\text{S2}, \text{Right}) \approx 8.96$$
$$Q_2(\text{S1}, \text{Down}) \approx 8.96$$
$$Q_2(\text{S0}, \text{Right}) \approx 6.24$$

Iteration 3

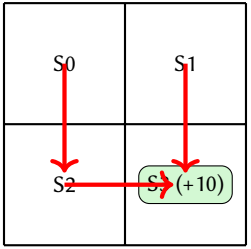
$$Q_3(\text{S0}, \text{Down}) = 7.74$$

Values start propagating backward!

The Bellman equation propagates reward information backward from rewarding states.

Q* Values After Convergence (Converged after 9 iterations!)

State	Action	Q* Value
S0	Right	8.57
S0	Down	**8.57**
S1	Left	8.00
S1	Down	**9.68**
S2	Up	8.00
S2	Right	**9.68**



Optimal Policy π^*

For each state, choose action with highest Q* value:

State	Best Action	Q* Value
S0	Down	8.57
S1	Down	9.68
S2	Right	9.68

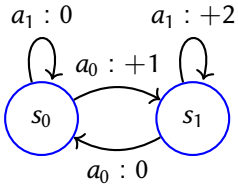
Q-values quantify the "goodness" of each state-action pair in terms of expected long-term reward.

The optimal policy efficiently guides the agent toward the high-reward terminal state.

State-Action Rewards $R(s, a)$

- Rewards depend only on the current state and action taken
- Notation: $R(s, a)$ or R_{sa}
- Mathematical definition:

$$R(s, a) = \mathbb{E}[r_t | s_t = s, a_t = a]$$
$$= \sum_{s' \in S} P(s' | s, a) \cdot r(s, a)$$

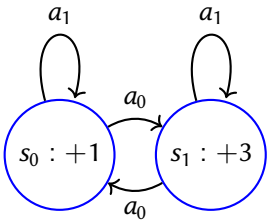


- Example: In a grid world, stepping on a trap (action) while in a specific location (state) incurs a penalty, regardless of where you end up

State-Only Rewards $R(s)$

- Rewards depend only on the state the agent is in
- Notation: $R(s)$ or R_s
- Mathematical definition:

$$R(s) = \mathbb{E}[r_t | s_t = s]$$



- Value functions with state-only rewards:

$$V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s, \pi(s)) V^{\pi}(s')$$

$$Q^{\pi}(s, a) = R(s) + \gamma \sum_{s' \in S} P(s'|s, a) V^{\pi}(s')$$

- Example: Treasure on specific grid locations gives reward just for being there

State-Action-State Rewards $R(s, a, s')$

- Rewards depend on the current state, action taken, and the resulting next state
- Notation: $R(s, a, s')$ or $R_{sas'}$
- Mathematical definition:

$$R(s, a, s') = \mathbb{E}[r_t | s_t = s, a_t = a, s_{t+1} = s']$$

- Bellman equation with state-action-state rewards:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P(s'|s, a)[R(s, a, s') + \gamma V^{\pi}(s')]$$

- Example: Reward for successfully jumping over a pit (transition from initial state to goal state via jump action)

State-Action-State Rewards: Matrix Form

For an MDP with 2 states and 2 actions, $R(s, a, s')$ can be represented as matrices:
For action a_0:

$$R_{a_0} = \begin{bmatrix} R(s_0, a_0, s_0) & R(s_0, a_0, s_1) \\ R(s_1, a_0, s_0) & R(s_1, a_0, s_1) \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ -1 & 0 \end{bmatrix}$$

For action a_1:

$$R_{a_1} = \begin{bmatrix} R(s_0, a_1, s_0) & R(s_0, a_1, s_1) \\ R(s_1, a_1, s_0) & R(s_1, a_1, s_1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 3 & 2 \end{bmatrix}$$

The expected reward for taking action a in state s is:

$$R(s, a) = \sum_{s' \in S} P(s'|s, a) \cdot R(s, a, s')$$

Expected Rewards: Converting Between Forms

- Converting from $R(s, a, s')$ to $R(s, a)$:

$$R(s, a) = \sum_{s' \in S} P(s'|s, a) \cdot R(s, a, s')$$

- Converting from $R(s, a)$ to expected rewards:

$$R(s) = \sum_{a \in A} \pi(a|s) \cdot R(s, a)$$

- Bellman equation using $R(s, a)$:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \left[R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^{\pi}(s') \right]$$

Practical Example: Grid World Rewards

State-Action Rewards $R(s, a)$

$$R(s, \text{up}) = \begin{cases} -1, & \text{if } s \text{ is regular cell} \\ -10, & \text{if } s \text{ is adjacent to pit} \end{cases}$$

$$R(s, \text{right}) = \begin{cases} -1, & \text{if } s \text{ is regular cell} \\ +10, & \text{if } s \text{ is adjacent to goal} \end{cases}$$

State-Only Rewards $R(s)$

$$R(s) = \begin{cases} -1, & \text{if } s \text{ is regular cell} \\ -100, & \text{if } s \text{ is pit} \\ +100, & \text{if } s \text{ is goal} \end{cases}$$

State-Action-State Rewards $R(s, a, s')$

$$R(s, a, s') = \begin{cases} -1, & \text{if } s' \text{ is regular cell} \\ -100, & \text{if } s' \text{ is pit} \\ +100, & \text{if } s' \text{ is goal} \\ -10, & \text{if action fails (hits wall)} \end{cases}$$

Comparison of Reward Structures

Reward Type	Advantages	Disadvantages
State-Only $R(s)$	Simple to understand and implement	Cannot reward specific behaviors or transitions
State-Action $R(s, a)$	Can encourage/discourage specific actions in specific states	Cannot distinguish between different outcomes of the same action
State-Action-State $R(s, a, s')$	Most expressive; can reward specific transitions	More complex to design and implement

- Choice of reward structure depends on:
  - Task requirements
  - Available information
  - Desired agent behavior
- Any reward structure can be converted to another with proper mathematical transformations

Relationship to Optimal Policies

The optimal policy π^* maximizes the expected discounted sum of rewards:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$
$$= \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right]$$

Optimal value functions:

$$V^*(s) = \max_a \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s')\right]$$
$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \max_{a'} Q^*(s', a')$$

The choice of reward structure can significantly impact the learned policy!

Reward Design Principles

- **Parsimony**: Use the simplest reward structure that captures the task requirements
- **Alignment**: Ensure rewards truly represent the desired behaviors and outcomes
- **Consistency**: Similar states/actions should receive similar rewards
- **Sparsity vs. Density**: Trade-off between sparse rewards (clearer signal) and dense rewards (easier learning)
- **Scale**: The magnitude of rewards should be proportional to their importance
- **Horizons**: Consider discount factor γ in relation to reward time scales

Reward design is often considered an art as much as a science in reinforcement learning!

Let's do some coding

https://github.com/rahulbhadani/CPE490_590_Sp2025/blob/master/Code/Chapter_15_Reinforcement_Learning.ipynb

Deep Reinforcement Learning

Why Deep Reinforcement Learning?

The use of the Bellman Equation and Dynamic Programming to find optimal policy requires computing $Q(s, a)$ for every state-action pair. This can be computationally intractable for most problems such as a autonomous driving where the state is continuous (denoted by speed, acceleration, etc.9+).

In such a case, we use a deep neural network as a function approximation for $Q(s, a)$.

Deep Learning Algorithms

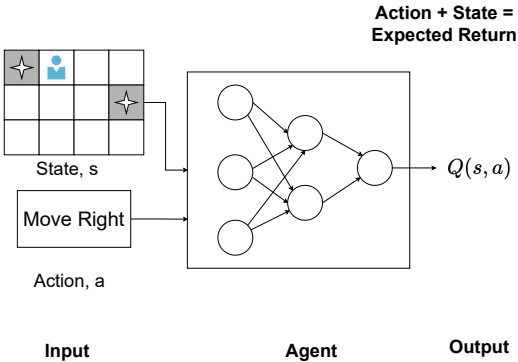
Value Learning (Q-Learning)

- Model $Q(s, a)$ using deep learning,
- then $a = \arg\max_a Q(s, a)$

Policy Learning

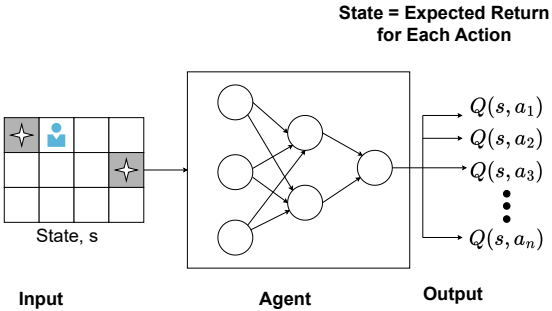
- Model $\pi(s)$ using deep learning,
- then Sample $a \sim \pi(s)$

Q-Learning



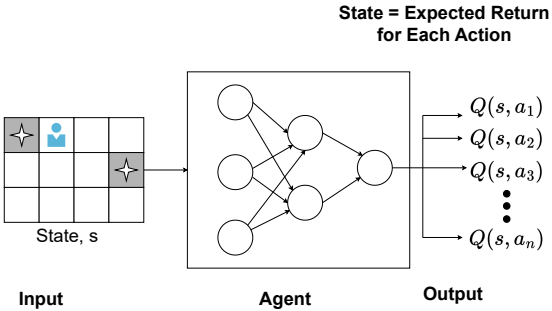
Very inefficient for a large set of actions.

Q-Learning



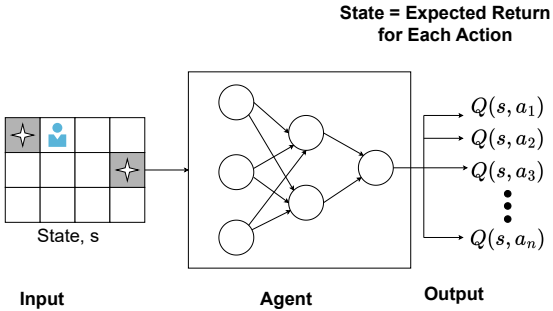
Take action based on the highest value of Q.

Q-Learning



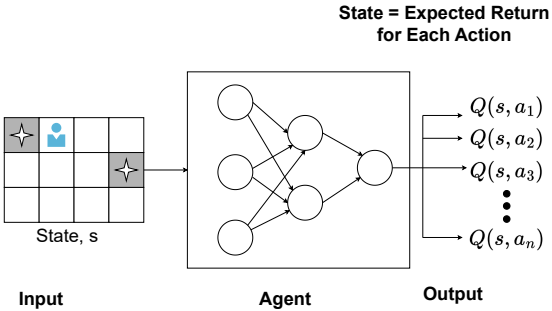
What happens when we take all the best actions?

Q-Learning



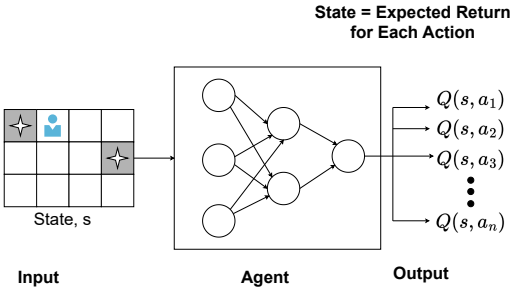
This would mean that the target return (what we are trying to predict) will always be maximized.

Q-Learning



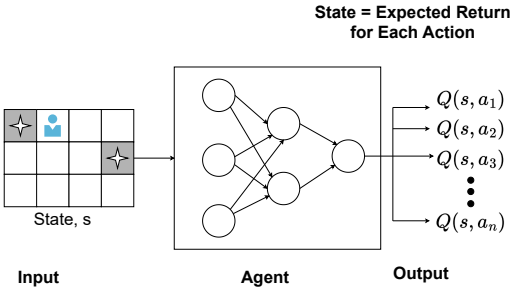
That can serve as a ground truth to the agent.

Q-Learning



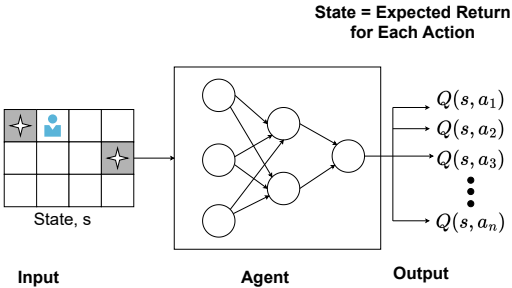
Maximize target return $\rightarrow$ train the agent.

Q-Learning



To do this, we want to formulate a loss function that will essentially return an expected return if we were able to take all of the best actions.

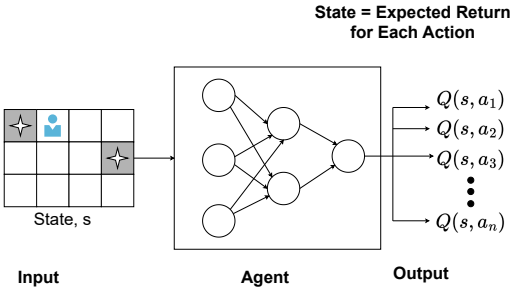
Q-Learning



Essentially, in this case, the target is

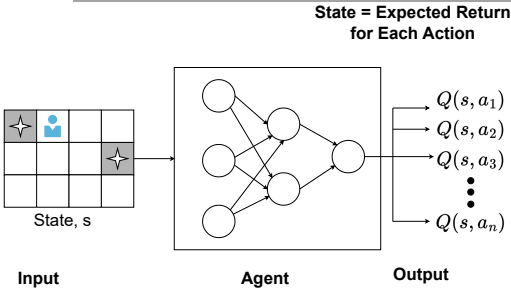
$$r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$

Q-Learning



And, the predicted variable would denoted by $Q(s, a)$.

Q-Learning



And, the predicted variable would denoted by $Q(s, a)$.

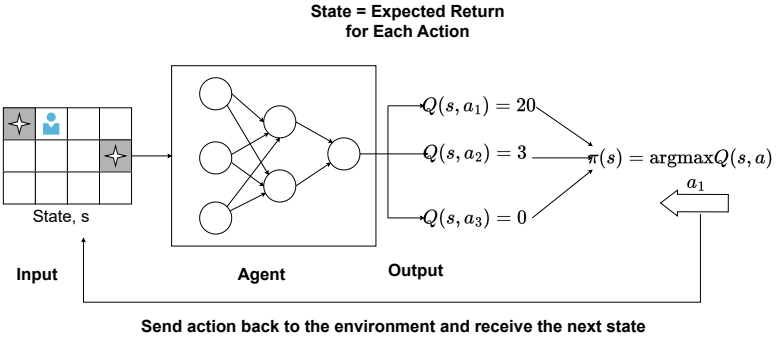
Then, we can define Q-loss as

$$\mathcal{L} = \mathbb{E}\left[\left|\left|\left(r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})\right) - Q(s, a)\right|\right|^2\right]$$

Q-Learning

Using NN, we can learn the Q-function and then infer the optimal policy $\pi(s)$ using $\pi(s) = \arg\max_a Q(s, a)$.

Q-Learning


State = Expected Return
for Each Action
$Q(s, a_1) = 20$
$Q(s, a_2) = 3$
$Q(s, a_3) = 0$
$\pi(s) = \text{argmax} Q(s, a)$
a_1
State, s
Input
Agent
Output
Send action back to the environment and receive the next state

Solving Optimal Policy

Assuming that the neural network will be parametrized by θ (i.e function weights, the thing we want to keep updating in backward propagation),

Forward propagation:

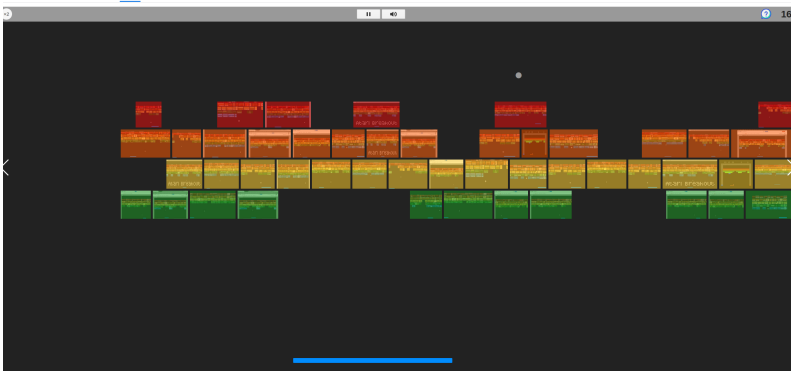
$$\mathcal{L}(\theta_i) = \mathbb{E}\left[\left|\left|\left(r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}, \theta_{i-1})\right) - Q(s, a, \theta_i)\right|\right|^2\right]$$

Iteratively make $Q(s, a, \theta_i)$ close to the target (the first time in the squared difference above).

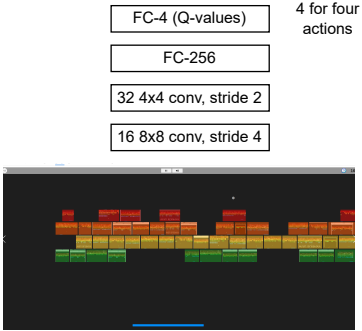
Backward propagation:

$$\nabla_{\theta_i}\mathcal{L}(\theta_i)\mathbb{E}\left[\left(r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}, \theta_{i-1})\right) - Q(s, a, \theta_i)\nabla_{\theta_i} Q(s, a, \theta_i)\right]$$

Case Study: Atari Breakout Game



Case Study: Atari Breakout Game



Current state s_t 84x84x4 stack of last 4 frames

Convert to grayscale, downsample and crop

Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated $\rightarrow$ inefficient learning.
- Current Q-network parameters determine the next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) $\rightarrow$ can lead to bad feedback loops.

Training the Q-network: Experience Replay

We address these problems using **experience replay**:

- Continually update a replay memory table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
- Train Q-network on random mini-batches of transitions from the replay memory, instead of consecutive samples
- Each transition can also contribute to multiple weight updates

From [Mnih et al. NIPS Workshop 2013; Nature 2015]

Original paper: https://arxiv.org/pdf/1312.5602.pdf

Experience Replay Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
for $t = 1, T$ **do**
With probability ϵ select a random action a_t
otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
Execute action a_t in emulator and observe reward r_t and image x_{t+1}
Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
end for
end for

Play M episodes (full games)

Experience Replay Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Initialize state (starting game screen pixels) at the beginning of each episode

Experience Replay Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

For each timestep t of the game

Experience Replay Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

With small probability, select a random action (explore), otherwise, select greedy action from current policy

Experience Replay Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Take the action (a_t), and observe the reward r_t and next state s_{t+1}

Experience Replay Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Store transition in replay memory

Experience Replay Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Experience Replay: Sample a random minibatch of transitions from replay memory and perform a gradient descent step

Let's do some coding

Problem with Q-learning

Problem with Q-learning

- Q-learning can model scenarios where the action space is discrete and small.
- The formulation we saw can't handle continuous action spaces.
- Policy is deterministically calculated by maximizing the reward $\rightarrow$ cannot learn stochastic policy.

Problem with Q-learning

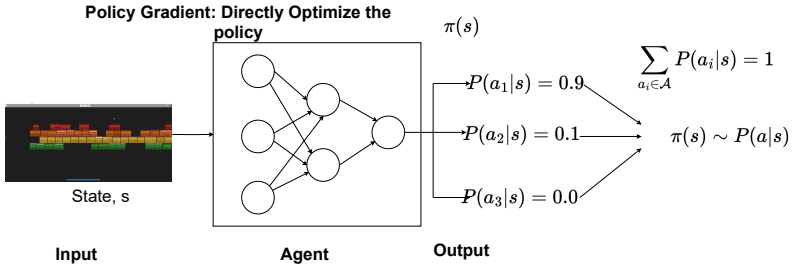
Problem with Q-learning

The Q-function can be very complicated!

Example: a robot grasping an object has a very high-dimensional state $\rightarrow$ hard to learn the exact value of every (state, action) pair.

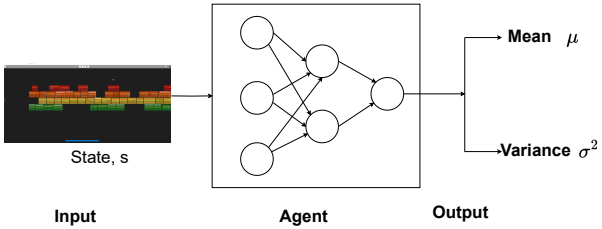
But the policy can be much simpler: simply close your hand.
Can we learn a policy directly, e.g. finding the best policy from a collection of policies?

Formulating Policy Gradients



Sample from $\pi(s)$, explore the environment, and obtain some stochasticity.

Formulating Policy Gradients

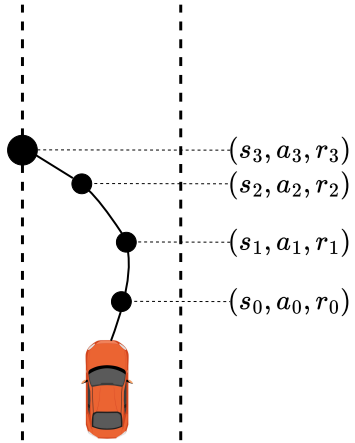


$$P(a|s) \sim \mathcal{N}(\mu, \sigma^2).$$

https://link.springer.com/article/10.1007/BF00992696

Example: Self-driving Cars

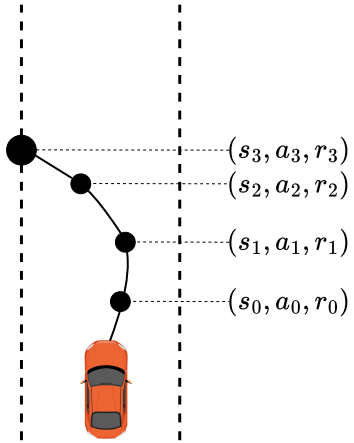
- **Agent:** vehicle
- **State:** comes from sensor measurements: camera, Lidar, etc.
- **Action:** steering wheel angle
- **Reward:** distance traveled



Training Policy Gradients

Training Algorithm

- Initialize the agent
- Run a policy until termination
- Record all states, actions, rewards
- Decrease the probability of actions that resulted in low reward.
- Increase probability of actions that lead to high rewards



REINFORCE Algorithm for Policy Gradients

function REINFORCE
Initialize θ
for $episode \sim \pi_\theta$
$\{s_i, a_i, r_i\}_{i=1}^{T-1} \leftarrow episode$
for t = 1 to T-1
$\nabla \leftarrow \nabla_\theta \log \pi_\theta(a_t|s_t)\, R_t$
$\theta \leftarrow \theta + \alpha\nabla$
return θ

log-likelihood of action

$$\nabla_\theta \log \pi_\theta(a_t|s_t)\, R_t$$

reward

Code Demo of Snake RL Game

https://github.com/rahulbhadani/CPE490_590_Sp2025/tree/master/Code/RL_Snake

Run `python agent.py` to train aan RL agent.
Run `python snake_game_original.py` and use arrow keys to play game by yourself.

The idea behind using Q-learning in Snake Game

Originally, Q-learning would maintain a table of values that gives us the expected future reward for taking action a in state s

$$Q(s,a) = Q(s,a) + \eta[r + \gamma max(Q(s',a')) - Q(s,a)]$$

where η is the learning rate. $max(Q(s',a'))$ is the maximum expected future reward from the new state.

An Example 4×5 Grid

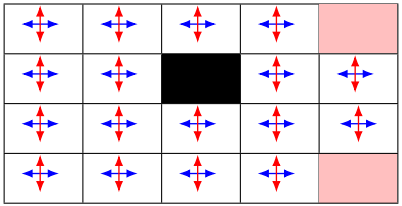
Grid

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2, 4)
(3,0)	(3,1)	(3,2)	(3,3)	(3, 4)

Reward

-1	-1	-1	-1	+100
-1	-100	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	+100

Movement



When the movement causes to hit the wall, state doesn't change.

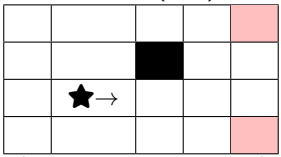
Consider $\eta = 0.5$, $\gamma = 0.9$.

An Example 4×5 Grid

Initial Q values to zero. $Q(s, a) = 0.0$ for all s and a.

Episode 1

Start at $(2, 1)$.



Randomly pick an action: Say Right ($a =$ Right).

Transition: $(2, 1) \rightarrow (2, 2)$.

Reward received: $r = -1$.

New state: $s' = (2, 2)$.

An Example 4×5 Grid: Q-value Calculation

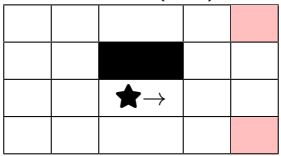
$$
\begin{aligned}
Q((2,1), \text{Right}) &= Q((2,1), \text{Right}) + \eta[r + \gamma \max_{a'} Q(s' = (2,2), a') - Q((2,1), \text{Right})] \\
&= 0 + 0.5[-1 + 0.9 \max_{a'} Q((2,2), a') - 0] \\
&= 0 + 0.5[-1 + 0.9 \times 0 - 0] \quad \text{(Since all initial Q values are 0)} \\
&= 0.5[-1] \\
&= -0.5
\end{aligned}
$$

An Example 4×5 Grid: Continue

At $(2, 2)$,

Episode 1

Start at $(2, 1)$.



Randomly pick an action: Say Right ($a =$ Right).

Transition: $(2, 2) \rightarrow (2, 3)$.

Reward received: $r = -1$.

New state: $s' = (2, 3)$.

An Example 4×5 Grid: Q-value Calculation

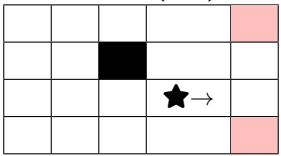
$$
\begin{aligned}
Q((2,2), \text{Right}) &= Q((2,2), \text{Right}) + \eta[r + \gamma \max_{a'} Q(s' = (2,3), a') - Q((2,2), \text{Right})] \\
&= 0 + 0.5[-1 + 0.9 \max_{a'} Q((2,3), a') - 0] \\
&= 0 + 0.5[-1 + 0.9 \times 0 - 0] \quad \text{(Since all initial Q values are 0)} \\
&= 0.5[-1] \\
&= -0.5
\end{aligned}
$$

An Example 4×5 Grid: Continue

At $(2, 3)$,

Episode 1

Start at $(2, 1)$.



Randomly pick an action: Say Right ($a =$ Right).

Transition: $(2, 3) \rightarrow (2, 4)$.

Reward received: $r = -1$.

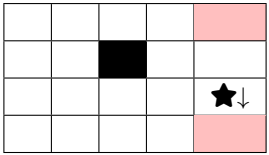
New state: $s' = (2, 4)$, and $Q((2, 3), \text{Right}) = -0.5$

An Example 4×5 Grid: Continue

At $(2, 4)$,

Episode 1

Start at $(2, 1)$.



Randomly pick an action: Say Down ($a =$ Right).

Transition: $(2, 4) \rightarrow (3, 4)$.

Reward received: $r = +100$.

New state: $s' = (3, 4)$, and $Q((2, 3), \text{Right}) = +50$

We reach our goal, episode 1 ends.

An Example 4×5 Grid: Episode 2

- Episode 2 uses updated $Q(s, a)$ values from Episode 1.
- How does action probability change?
-
  - Epsilon-greedy Action Selection
  - Softmax (Boltzmann) Action Selection

The Case of Exploitation vs Exploration

Exploration

- Trying out different actions to discover more about the environment and potential rewards.
- Analogous to randomly exploring the levers of slot machines to see which one gives the best payout.

The Case of Exploitation vs Exploration

Exploitation

- Using the current knowledge to choose the action that is believed to yield the highest reward.
- Analogous to repeatedly playing the slot machine that has given the best payout so far.

Reference: Deep Reinforcement Learning in Action By Alexander Zai, Brandon Brown

Epsilon-greedy Action Selection

After each episode, the agent learn something new. Hence, after episode 1, it has capability to exploit (based on what it learnt) in addition to explore.

ϵ-greedy Approach

- With probability $1 - \epsilon$, choose the best action (with highest Q value)
- With probability ϵ, pick randomly among all actions

Mathematically,

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|}, & \text{if best action} \\ \frac{\epsilon}{|\mathcal{A}|}, & \text{otherwise} \end{cases}$$

Reference: Section 5.4 Reinforcement Learning An Introduction 2nd ed, Richard S. Sutton and Andrew G. Barto

Actor-Critic Algorithm: Motivation

The algorithms we saw above are of two types:

- **Actor-only:** methods work with a parameterized family of policies (policy gradients). The gradient of the performance, with respect to the actor parameters, is directly estimated by simulation, and the parameters are updated in the direction of improvement.
- **Critic-only:** rely exclusively on value function approximation and aim at learning an approximate solution to the Bellman equation, which will then hopefully prescribe a near-optimal policy (Q-learning).

Actor-critic combines the strong points of actor-only and critic-only methods.

Actor-critic Algorithms

- Estimate the policy, estimate the value function.
- Critic-only methods estimate the value function, policy is implicit.
- Actor-only methods estimate the policy and we don't work with value functions

You can consider that this algorithm has two parts: actor and critic. The actor decides which action should be taken and the critic informs the actor how good the action is and how it should adjust.

This type of architecture also appears in GAN (Generative Adversarial Network) where 'discriminator' and 'generator' play games.

How does Actor-critic Algorithm works

- The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust
- Also alleviates the task of the critic as it only has to learn the values of (state, action) pairs generated by the policy
- Can also incorporate Q-learning tricks e.g. experience replay

How to define how better the action was?

How should we quantify how much an action was better than expected?

Advantage function

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

Pseudo Code for Actor-Critic Algorithm

Initialize policy parameters θ, critic parameters ϕ

For iteration=1, 2 … **do**

Sample m trajectories under the current policy

$\Delta\theta \leftarrow 0$

For i=1, ..., m **do**

For t=1, ... , T **do**

$$A_t = \sum_{t' \geq t} \gamma^{t'-t} r_t^i - V_\phi(s_t^i)$$

$$\Delta\theta \leftarrow \Delta\theta + A_t \nabla_\theta \log(a_t^i | s_t^i)$$

$$\Delta\phi \leftarrow \sum_i \sum_t \nabla_\phi ||A_t^i||^2$$

$\theta \leftarrow \alpha\Delta\theta$

$\phi \leftarrow \beta\Delta\phi$

End for

Proximal Policy Optimization (PPO)

Improves agent's training stability by avoiding policy updates that are too large. To do that, we use a ratio that indicates the difference between our current and old policy and clip this ratio to a specific range $[1-\epsilon, 1+\epsilon]$.

The intuition behind PPO

The idea with Proximal Policy Optimization (PPO) is that we want to improve the training stability of the policy by limiting the change you make to the policy at each training epoch: we want to avoid having too large of a policy update.
For two reasons:

- We know empirically that smaller policy updates during training are more likely to converge to an optimal solution.
- A too-big step in a policy update can result in falling "off the cliff" (getting a bad policy) and taking a long time or even having no possibility to recover.

PPO

To achieve this, we compare the new policy to the old one by computing the ratio of their action probabilities.
This ratio tells us how much the policy has changed.
To prevent the new policy from straying too far from the previous one, we clip this ratio within a fixed range $[1 - \epsilon, 1 + \epsilon]$.

This clipping mechanism limits the size of policy updates, encouraging the new policy to stay "proximal" to the old policy and promoting more stable training.

The Policy Objective Function

$$L^{PG}(\theta) = \mathbb{E}_t[\ \overbrace{\log \pi_\theta(a_t|s_t)}^{\text{the log probabilityof taking action}} * \underbrace{A_t}_{\substack{\text{the advantage term (positive when the action is better than other possible actions at that state)}}}\ ]$$

The idea was that by taking a gradient ascent step on this function (equivalent to taking gradient descent of the negative of this function), we would **push our agent to** take actions that lead to higher rewards and avoid harmful actions.
However, the problem comes from the step size:

- Too small, **the training process was too slow**
- Too high, **there was too much variability in the training**

With PPO, the idea is to constrain our policy update with a new objective function called the *Clipped surrogate objective function* that **will constrain** the policy change in a small range using a clip.

Importance Sampling in PPO

Probability Ratio

Instead of using the same policy, compare new and old policies using:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

where:

- θ_{old} are the parameters before the update.
- $r_t(\theta)$ measures how much the policy changed.

Clipped Surrogate Objective Function

Preventing Large Policy Changes

Define the clipped objective:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) A_t,\ \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) A_t \right) \right]$$

- If $r_t(\theta)$ changes too much, clipping ensures stable updates.
- ϵ is a hyperparameter (e.g., 0.2).

Policy Update Rule

Updating Policy Safely

After optimizing the clipped surrogate loss:

$$\theta_{\text{old}} \leftarrow \theta$$

Repeat data collection and optimization to improve the policy iteratively.

Full PPO Objective

Total Loss Function

PPO optimizes the combined objective:

$$L(\theta) = L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t)$$

where:

- $L^{\text{CLIP}}(\theta)$: Clipped surrogate policy loss.
- $L^{\text{VF}}(\theta)$: Value function loss (MSE between $V(s_t)$ and returns R_t).
- $S[\pi_\theta](s_t)$: Entropy bonus to encourage exploration.
- c_1, c_2: Coefficients to balance the terms.

PPO: Example

Environment Setup

- 1 state s, 2 actions: left and right.
- Agent picks **right**, receives reward $r = 1$.
- Old policy: $\pi_{\theta_{\text{old}}}(\text{right}|s) = 0.4$
- New policy: $\pi_{\theta}(\text{right}|s) = 0.6$
- Estimated advantage: $\hat{A} = 2.0$

PPO: Example

Step-by-Step Computation

$$
\begin{aligned}
r(\theta) &= \frac{0.6}{0.4} = 1.5 \\
L_1 &= r(\theta) \cdot \hat{A} = 1.5 \cdot 2.0 = 3.0 \\
r_{\text{clip}} &= \text{clip}(1.5, 0.8, 1.2) = 1.2 \\
L_2 &= r_{\text{clip}} \cdot \hat{A} = 1.2 \cdot 2.0 = 2.4 \\
L^{\text{CLIP}} &= \min(3.0, 2.4) = \boxed{2.4}
\end{aligned}
$$

That's the loss that would be used to update the policy network!

The End