PES UNIVERSITY

UE21CS343BB2
Topics in Deep Learning

Dr. Shylaja S S
Director of Cloud Computing & Big Data (CCBD), Centre for Data Sciences & Applied Machine Learning (CDSAML)
Department of Computer Science and Engineering
shylaja.sharath@pes.edu

Ack:Divija L
Teaching Assistant

PES UNIVERSITY

REINFORCEMENT LEARNING

Types of Learning:

Fundamental Types of Learning:

1. **SUPERVISED :**
- Learn a function that returns a label given an input
1. **UNSUPERVISED:**
- Learn a function that recognizes the inherent pattern from an unlabeled dataset
1. **REINFORCEMENT:**
- Learn to make good decisions at each step of a sequence predicting a sequence of actions.

Specialized Variants:

• Semi Supervised • Curriculum • Imitation • Meta Learning • Self Supervised

Reinforcement Learning Involves

- **OPTIMIZATION:**

-> Goal is to find an optimal way to make decisions. Yielding best outcomes or at least very good outcomes

- **DELAYED CONSEQUENCES:**

->Decisions now can impact things much later.Introduces two challenges:

(i) ***When planning:*** decisions involve reasoning about not just immediate benefit of a decision but also its longer term ramifications.

(ii)***When learning***: temporal credit assignment is hard (what caused later high or low rewards?)

Reinforcement Learning Involves

- **EXPLORATION:**

->Learning about the world by making decisions

->Censored data - Only get a reward (label) for decision made

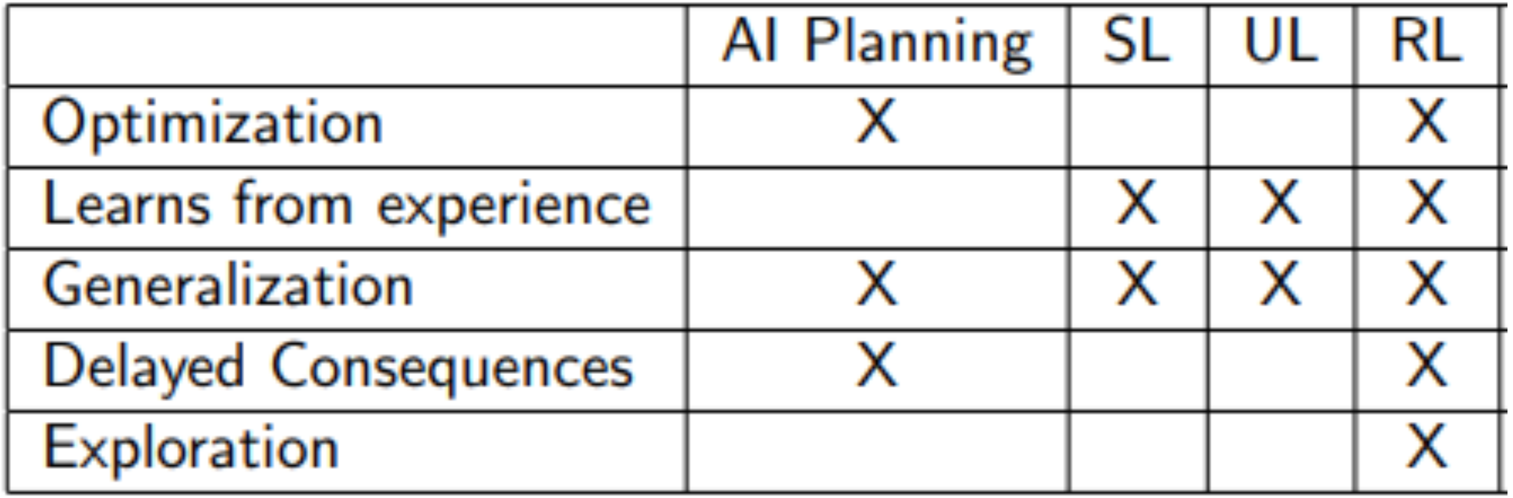
->Decisions impact what we learn about that may yield higher rewards

- **GENERALIZATION:**

->Policy is mapping from past experience to action

->Learning from data in order to generalize the predictions

RL compared to ML and AI



	AI Planning	SL	UL	RL
Optimization	X			X
Learns from experience		X	X	X
Generalization	X	X	X	X
Delayed Consequences	X			X
Exploration				X

SL = Supervised learning; UL = Unsupervised learning; RL = Reinforcement Learning;

Introduction to Reinforcement Learning:

- Reinforcement Learning is a sub topic of Artificial Intelligence modeled around the concept of Agent, Environment interaction.
- Humans and animals **learn by interacting with the environment**
- Learning is **active** as opposed to being passive
- Interactions are **sequential** and can be over a **longer time horizon**
- We can learn **without training examples** of optimal behavior
- We optimize for a **reward** signal

Predicting versus Acting on predictions

- Supervised and Unsupervised learning methods learn to predict under a static setting: e.g. given an image, predict its label.
- Reinforcement Learning is more dynamic and involve interaction with the environment. Such an interaction takes the form of actions, states and rewards.
    - These interactions take the form of sequences that are dynamic and form episodes.
- Though RNNs process sequences, datasets that constitute such sequences are static. E.g. Machine translation.
- Datasets used to train supervised learning based classifiers are iid. The episodes in an RL setting are almost always non iid.

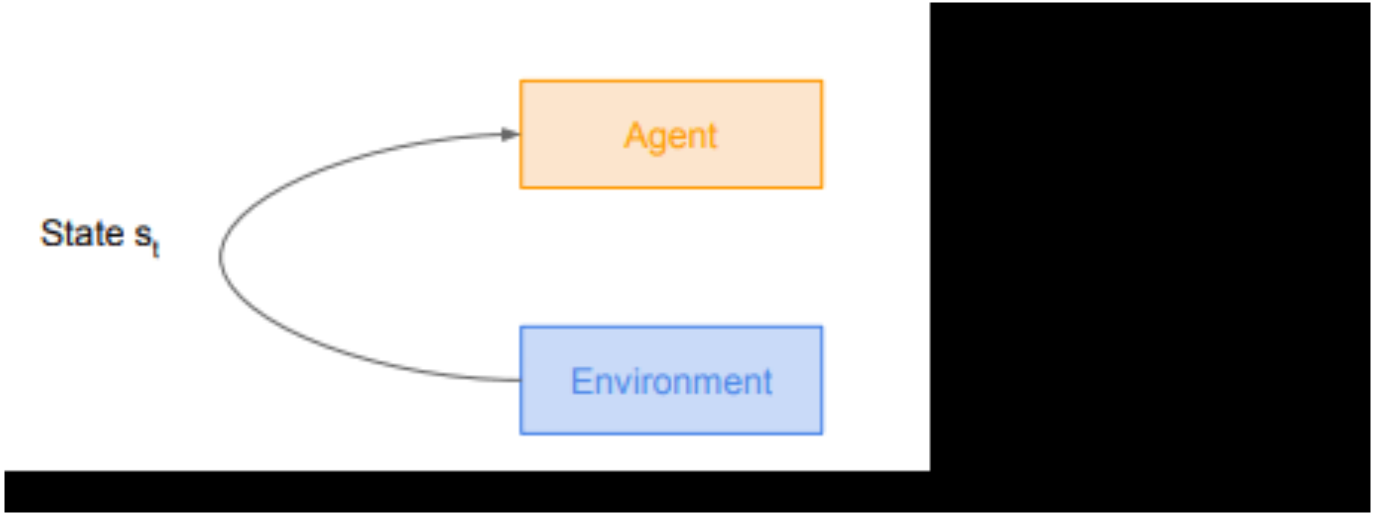
What is special about RL?

- RL is learning how to map states to actions, so as to maximize a numerical reward over time.
- Unlike other forms of learning, it is a multistage decision-making process (often Markovian).
- An RL agent must learn by trial-and-error. (Not entirely supervised, but interactive)
- Actions may affect not only the immediate reward but also subsequent rewards (Delayed effect).

Reinforcement Learning

Problems involving an agent interacting with an environment, which provides numeric reward signals.

Goal: Learn how to take actions in order to maximize reward



[redacted]

Elements of RL

- **States** are a representation of the current world or environment of the task.
- **Actions** are something an RL agent can do to change these states.
- Reward Function it maps each state (or state-action pair) to a real number called as reward.
- **Rewards** are the utility the agent receives for performing the “right” actions.
- **Policy** is a map from state space to action space. It can be stochastic
- **Value Function** : value of a state (or state-action pair) is the total expected reward , starting from that state (or state-action pair).

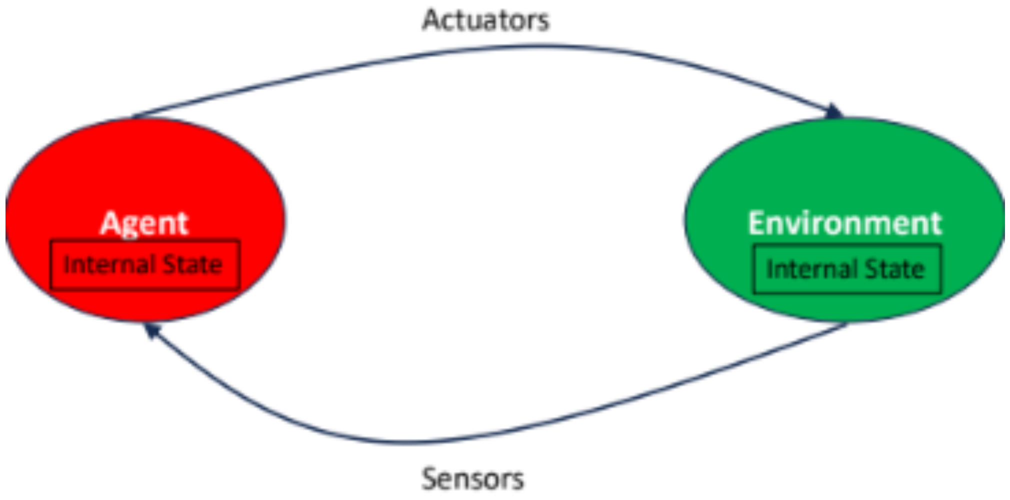
AI: Agent-Environment Model

- Agent perceives the environment (Percepts) and acts upon the environment in order to maximize achievement of the required goal

- An intelligent agent maps the percepts to actions

- The notion of Intelligent Agents is abstract. One can cast a variety of real world problems in to this concept.

- Examples:
- sensing camera feeds, detecting an intruder and raising alarm in a home security system.
- Sensing last 4 hours of stock movement trends and taking an action to buy/sell/hold in order to maximize returns



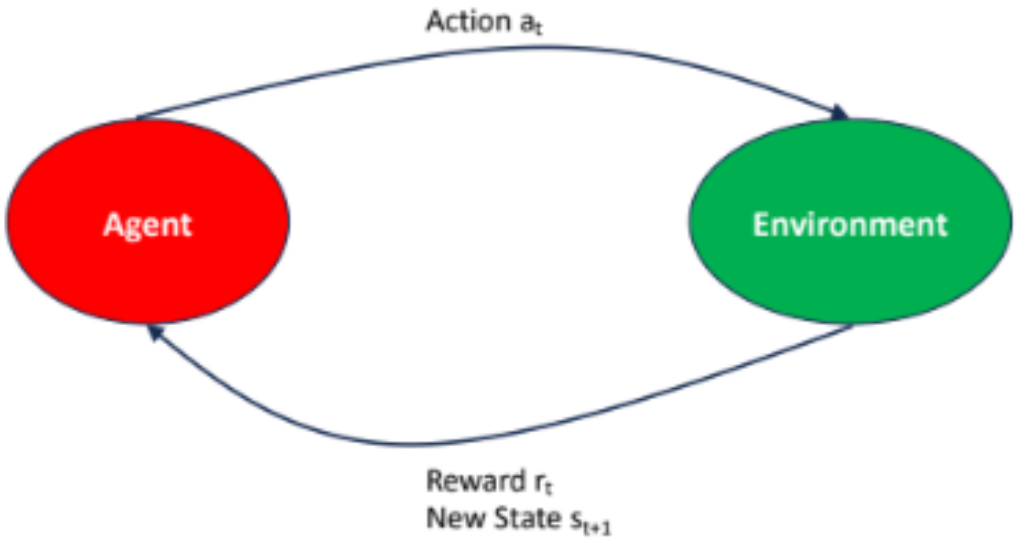
AI: Agent-Environment Model

1. At a time instant t, agent performs an action on the environment

2. The environment on executing the action undergoes a state change **s_t -> s_{t+1}**

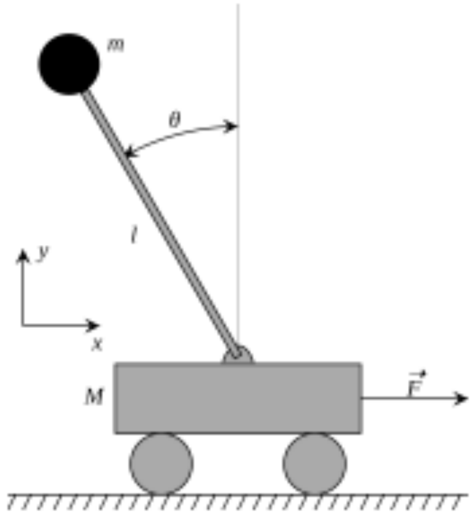
3. Environment returns a reward rt and the next state **s_{t+1}** to the agent.

Note: When the state is not fully observable, the returned state from the environment is referred to as observation



RL Applications

Cart-Pole Problem

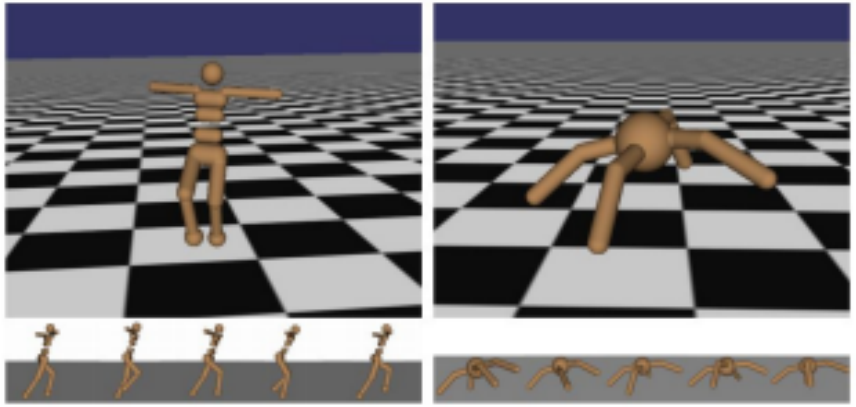


Objective: Balance a pole on top of a movable cart

State: angle, angular speed, position, horizontal velocity
Action: horizontal force applied on the cart
Reward: 1 at each time step if the pole is upright

RL Applications

Robot Locomotion



Objective: Make the robot move forward

State: Angle and position of the joints
Action: Torques applied on joints
Reward: 1 at each time step upright + forward movement

RL Applications

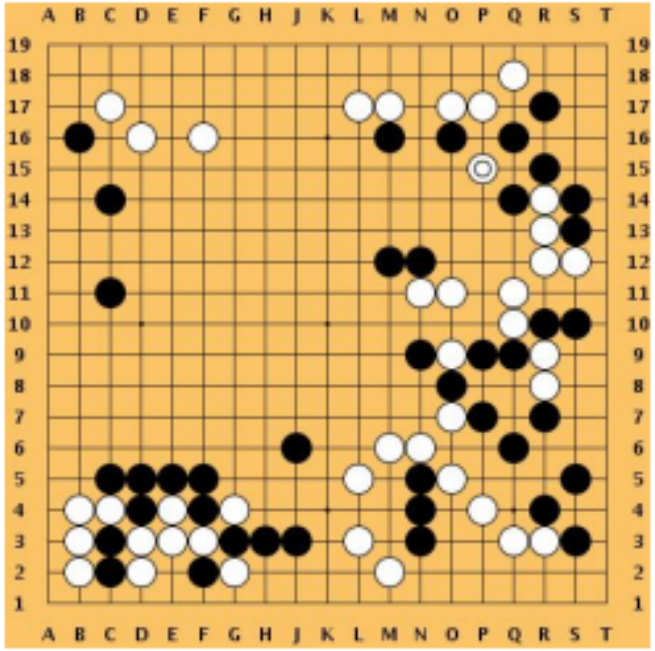
Atari Games

Objective: Complete the game with the highest score

State: Raw pixel inputs of the game state
Action: Game controls e.g. Left, Right, Up, Down
Reward: Score increase/decrease at each time step

RL Applications

Go

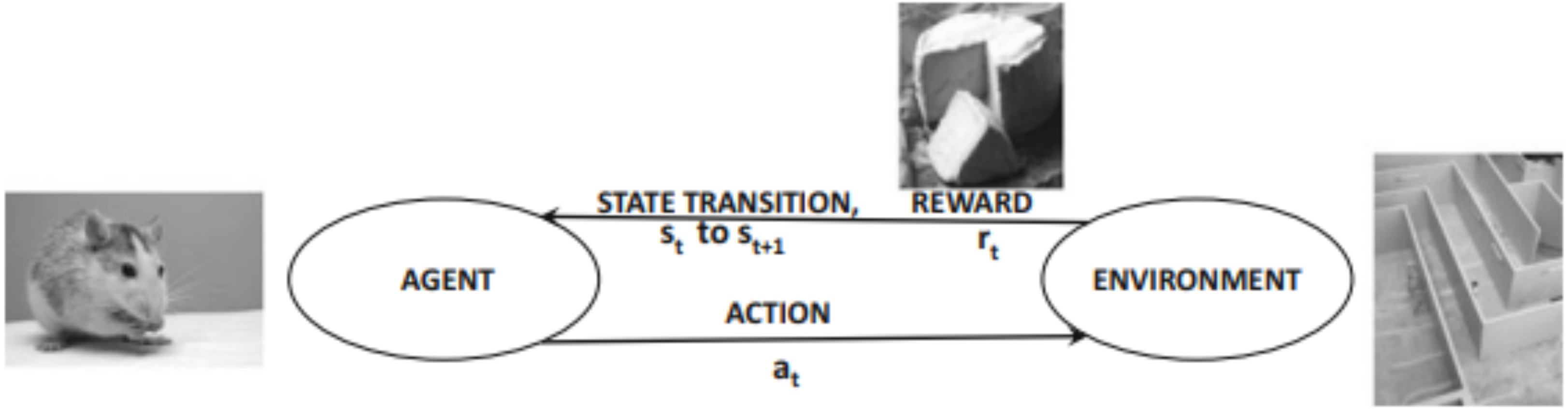


Objective: Win the game!

State: Position of all pieces
Action: Where to put the next piece down
Reward: 1 if win at the end of the game, 0 otherwise

UE22CS645BC2-TDL-Lecture - RL

Framework of RL



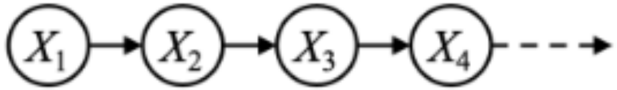
1. AGENT (MOUSE) TAKES AN ACTION a_t (LEFT TURN IN MAZE) FROM STATE (POSITION) s_t
2. ENVIRONMENT GIVES MOUSE REWARD r_t (CHEESE/NO CHEESE)
3. THE STATE OF AGENT IS CHANGED TO s_{t+1}
4. MOUSE'S NEURONS UPDATE SYNAPTIC WEIGHTS BASED ON WHETHER ACTION EARNED CHEESE

OVERALL: AGENT LEARNS OVER TIME TO TAKE STATE-SENSITIVE ACTIONS THAT EARN REWARDS

Figure 9.1: The broad framework of reinforcement learning

Markov Models

- Value of X at a given time is called the state



$$\begin{tikzcd} X_1 \arrow[r] & X_2 \arrow[r] & X_3 \arrow[r] & X_4 \arrow[r, dashed] & {} \end{tikzcd}$$

$P(X_1)$ $\quad$ $P(X_t|X_{t-1})$

- Parameters: called transition probabilities, specify how the state evolves over time (also, initial state probabilities)

- Stationarity assumption: transition probabilities the same at all times

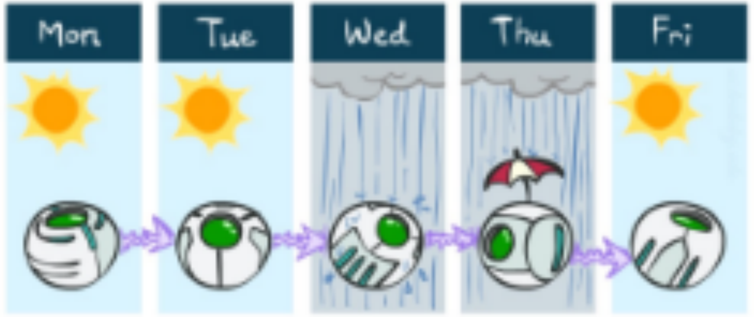
Why is Markov Assumptions popular?

- Simple and often can be satisfied if include some history as part of the state
- In practice often assume most recent observation is sufficient statistic of history: $s_t = o_t$
- State representation has big implications for:
  - Computational complexity
  - Data required
  - Resulting performance

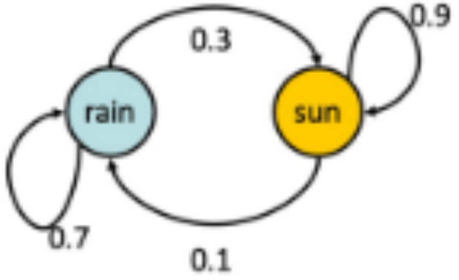
Markov Model Example: Weather sequence

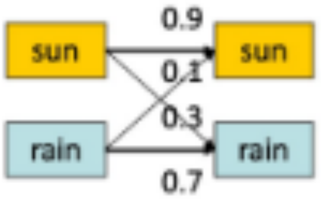
- States: X = {rain, sun}
- Initial distribution: 1.0 sun
- CPT $P(X_t \mid X_{t-1})$:

X_{t-1}	X_t	$P(X_t \mid X_{t-1})$
sun	sun	0.9
sun	rain	0.1
rain	sun	0.3
rain	rain	0.7



Two new ways of representing the same CPT

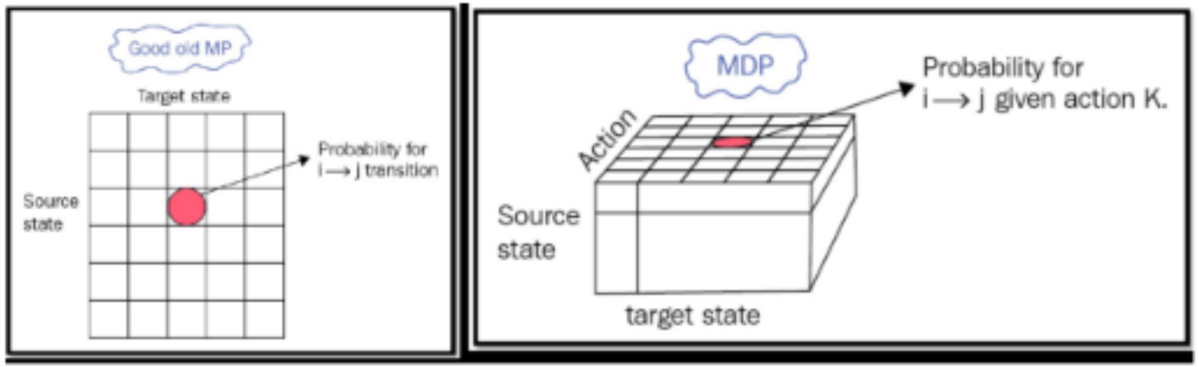




UE22CS645BC2-TDL-Lecture - RL

Markov Models

Adding actions to the state transition table


Good old MP
Target state
Source state
Probability for i ⟶ j transition
MDP
Action
Source state
target state
Probability for i ⟶ j given action K.

- Markov process and the Markov Reward Process do not incorporate the mechanism to alter the Environment
- Performing an action modelled by the MDP allows the agent to affect the probabilities of next state and future dynamics

Formulation of MDP

It is helpful for our understanding to view the MDP as an evolution of:

Markov Process (MP) or Markov Chain

- All possible states of a system is called “State Space”
- States are directly observable, sequences of these “observations” form a chain of states
- Future system dynamics depends only on the current state (Markov Property)
- Notion of state transition table

Markov Rewards Process (MRP)

- All properties same as MP but the notion of rewards included
- Imagine this to be 2 tables: A state transition table and a rewards table

Markov Decision Process (MDP)

- All the properties same as MRP but we add “actions” to the above
- Imagine the state transition table as a cube indexed with (i, j, a) where the index “a” is the action

Markov Decision Process

Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

$\mathcal{S}$: set of possible states
$\mathcal{A}$: set of possible actions
$\mathcal{R}$: distribution of reward given (state, action) pair
$\mathbb{P}$: transition probability i.e. distribution over next state given (state, action) pair
γ : discount factor

Markov Decision Process - Discount Factor

- Mathematically convenient (avoid infinite returns and values)
- Humans often act as if there's a discount factor < 1
- **γ = 0**: Only care about immediate reward
- **γ = 1**: Future reward is as beneficial as immediate reward
- If episode lengths are always finite (H < infinity), can use γ = 1

Markov Decision Process - Rewards

- In each time step, the agent receives a reward from a distribution that depends on the current state and action

$$R_t \sim \mathcal{R}(\cdot \mid S_t, A_t)$$

- For simplicity, we'll assume rewards are deterministic, i.e.

$$R_t = r(S_t, A_t)$$

- What's an example where R_t should depend on A_t?
- The return determines how good was the outcome of an episode.
  - Undiscounted: $G = R_0 + R_1 + R_2 + \cdots$
  - Discounted: $G = R_0 + \gamma R_1 + \gamma^2 R_2$
- The goal is to maximize the expected return, $\mathbb{E}[G]$.
- γ is a hyperparameter called the discount factor which determines how much we care about rewards now vs. rewards later.
  - What is the effect of large or small γ?

Markov Decision Process

- At time step t=0, environment samples initial state $s_0 \sim p(s_0)$
- Then, for t=0 until done:
  - Agent selects action a_t
  - Environment samples reward $r_t \sim R(. \mid s_t, a_t)$
  - Environment samples next state $s_{t+1} \sim P(. \mid s_t, a_t)$
  - Agent receives reward r_t and next state s_{t+1}

- A policy **π** is a function from S to A that specifies what action to take in each state
- **Objective**: find policy **π*** that maximizes cumulative discounted reward: $\sum_{t \geq 0} \gamma^t r_t$

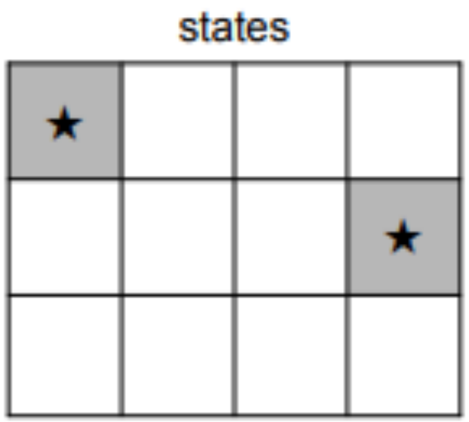
UE22CS645BC2-TDL-Lecture - RL

A simple MDP: Grid World

actions = {

1. right →
2. left ←
3. up ↑
4. down ↓

}

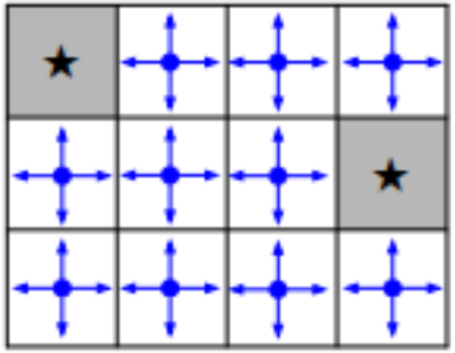


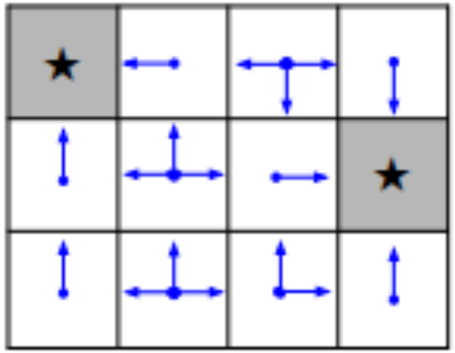
Set a negative "reward" for each transition (e.g. $r = -1$)

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

UE22CS645BC2-TDL-Lecture - RL

A simple MDP: Grid World


Random Policy


Optimal Policy

- Policy π determines how the agent chooses actions
- $\pi : S \rightarrow A$, mapping from states to actions
- Deterministic policy:

$$\pi(s) = a$$

- Stochastic policy:

$$\pi(a|s) = Pr(a_t = a|s_t = s)$$

UE22CS645BC2-TDL-Lecture - RL

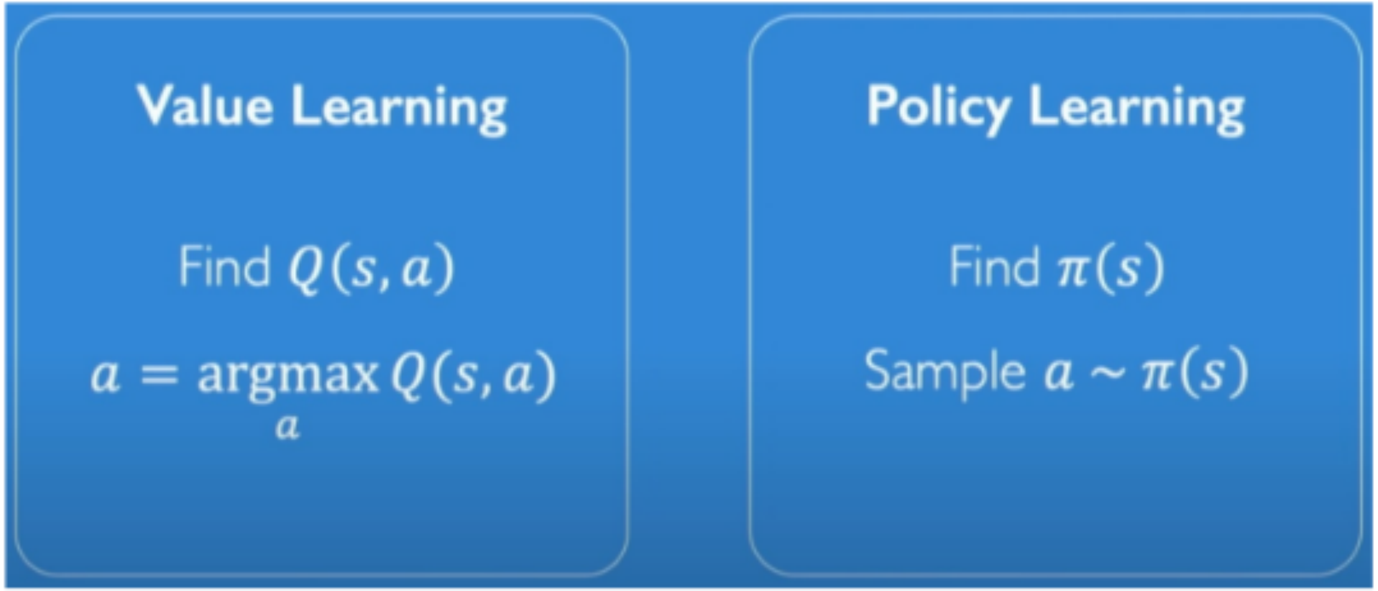
optimal policy **π***

We want to find optimal policy that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability…)?

Formally: $\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right]$ with $s_0 \sim p(s_0), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)$

Two approaches to Deep RL



Value Learning

Find $Q(s, a)$

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

Policy Learning

Find $\pi(s)$

Sample $a \sim \pi(s)$

Credit Assignment Problem

- The credit assignment problem (CAP) is a fundamental challenge in reinforcement learning. It arises when an agent receives a reward for a particular action, but the agent must determine which of its previous actions led to the reward.

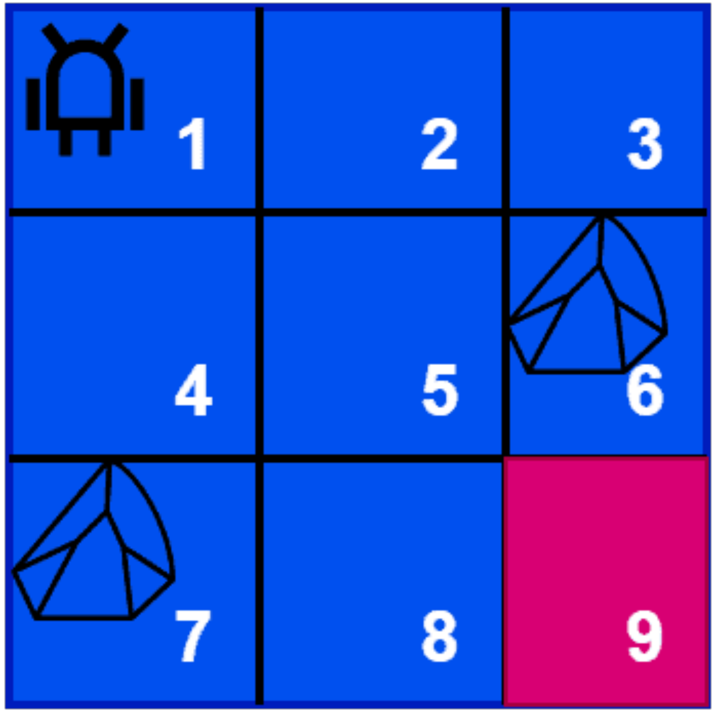
- In reinforcement learning, an agent applies a set of actions in an environment to maximize the overall reward. The agent updates its policy based on feedback received from the environment. It typically includes a scalar reward indicating the quality of the agent’s actions.

- **The credit assignment problem refers to the problem of measuring the influence and impact of an action taken by an agent on future rewards.** The core aim is to guide the agents to take corrective actions which can maximize the reward.

- However, in many cases, the reward signal from the environment doesn’t provide direct information about which specific actions the agent should continue or avoid. This can make it difficult for the agent to build an effective policy.

Credit Assignment Problem Example

- Suppose an agent is playing a game where it must navigate a maze to reach the goal state. We place the agent in the top left corner of the maze. Additionally, we set the goal state in the bottom right corner. The agent can move up, down, left, right, or diagonally. However, it can't move through the states containing stone:



- As the agent explores the maze, it receives a reward of +10 for reaching the goal state.

  Additionally, if it hits a stone, we penalize the action by providing a -10 reward.

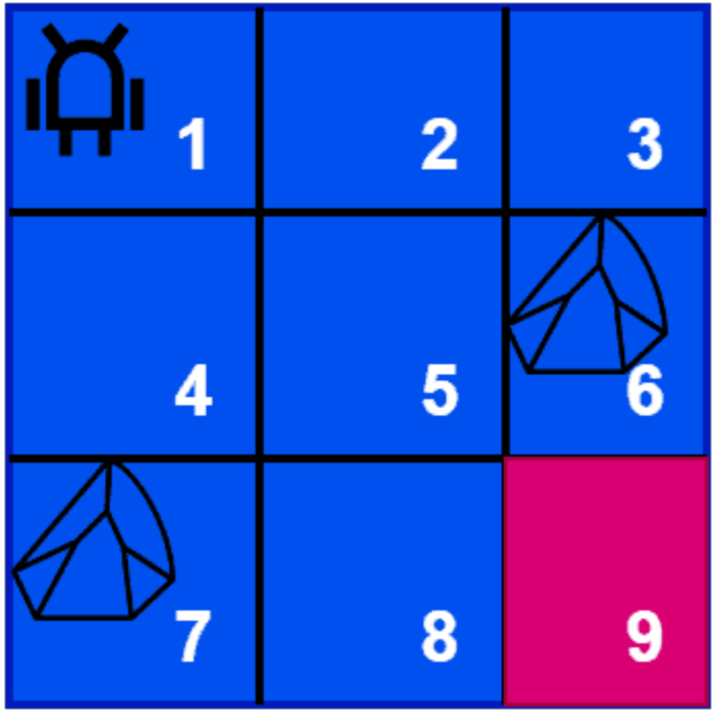
Credit Assignment Problem Example

The goal of the agent is to learn from the rewards and build an optimal policy that maximizes the gross reward over time.

The credit assignment problem arises when the agent reaches the goal after several steps

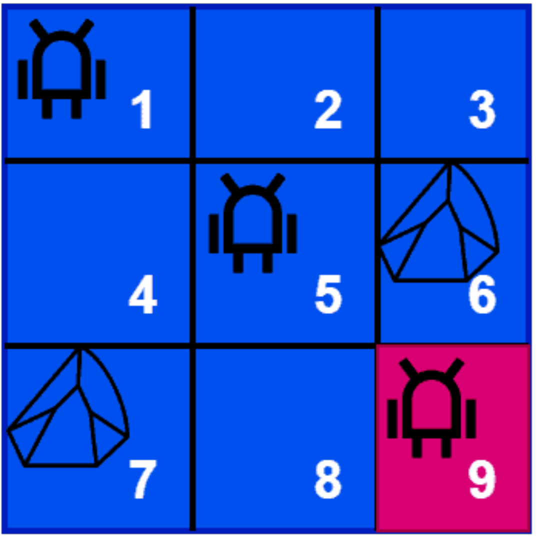
The agent receives a reward of +10 as soon as it reaches the goal state. However, it's not clear which actions are responsible for the Reward.

For example, suppose the agent took a long and winding path to reach the goal. Therefore, we need to determine which actions should receive credit for the reward.

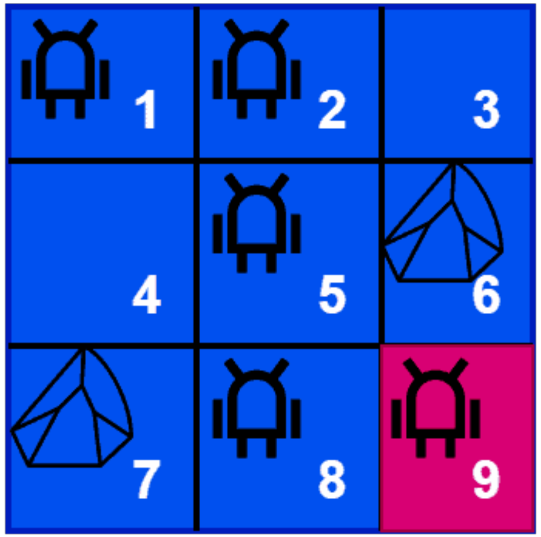


Credit Assignment Problem Example

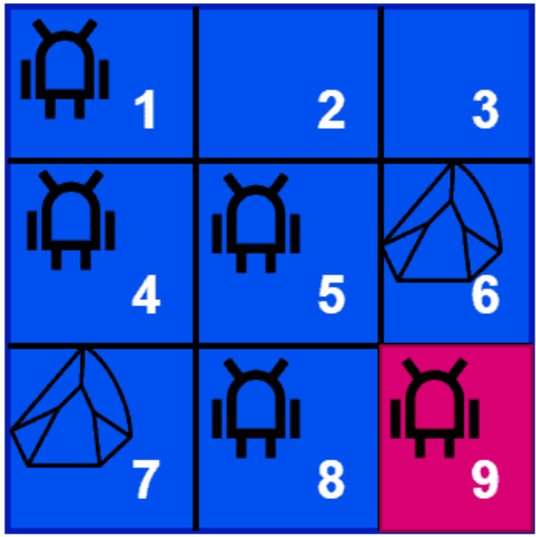
Let's look at some paths which lead the agent to the goal state:



Path 1 (1-5-9)



Path 2 (1-2-5-8-9)



Path 3 (1-4-5-8-9)

As we can see here, **the agent can reach the goal state with three different paths.** Hence, it's challenging to measure the influence of each action. We can see the best path to reach the goal state is path 1.

Approaches to solve Credit Assignment Problem :

Here are the three popular approaches:

1. **Temporal difference (TD) learning**

   -TD learning is a popular RL algorithm that uses a bootstrapping approach to assign credit to past actions.

   -It updates the value function of the policy based on the difference between the predicted reward and the actual reward received at each time step.

   -By bootstrapping the value function from the predicted rewards of future states, TD learning can assign credit to past actions even when the reward is delayed.

Approaches to solve Credit Assignment Problem :

2. Monte Carlo methods

-a class of RL algorithms that use full episodes of experience to assign credit to past actions.

-These methods estimate the expected value of a state by averaging the rewards obtained in the episodes that pass through that state.

-By averaging the rewards obtained over several episodes, Monte Carlo methods can assign credit to actions that led up to the reward, even if the reward is delayed.

Approaches to solve Credit Assignment Problem :

3. Eligibility traces method

- Eligibility traces are a method for assigning credit to past actions based on their recent history.
- Eligibility traces keep track of the recent history of state-action pairs and use a decaying weight to assign credit to each pair based on how recently it occurred.
- By decaying the weight of older state-action pairs, eligibility traces can assign credit to actions that led up to the reward, even if they occurred several steps earlier.

For more info : https://www.tu-chemnitz.de/informatik/KI/scripts/ws0910/ml09_7.pdf

Types of RL agents

1. **Model-based**
- Explicit: Model
- May or may not have policy and/or value function

1. **Model-free**
- Explicit: Value function and/or policy function
- No model
- Among RL's model-free methods is temporal difference (TD) learning, with SARSA and Q-learning (QL) being two of the most used algorithms

Temporal Difference Learning (TD Learning)

Among RL’s model-free methods is temporal difference (TD) learning, with SARSA and Q-learning (QL) being two of the most used algorithms

TD learning is an unsupervised technique to predict a variable's expected value in a sequence of states.

TD uses a mathematical trick to replace complex reasoning about the future with a simple learning procedure that can produce the same results.

$$\underbrace{V(S_t)}_{\text{New value of state t}} \leftarrow \underbrace{V(S_t)}_{\text{Former estimation of value of state t}} + \underbrace{\alpha}_{\text{Learning Rate}}[\underbrace{\underbrace{R_{t+1}}_{\text{Reward}} + \underbrace{\gamma V(S_{t+1})}_{\text{Discounted value of next state}}}_{\text{TD Target}} - \underline{V(S_t)}]$$

Instead of calculating the total future reward, TD tries to predict the combination of immediate reward and its own reward prediction at the next moment in time.

Temporal Difference Learning (TD Learning)

Mathematically, the key concept of TD learning is the discounted return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Where the reward at time t is the combination of discounted rewards in the future.
It implies that future rewards are valued less

TD Error is the difference between the ultimate correct reward (V^*_t) and our current prediction (V_t).

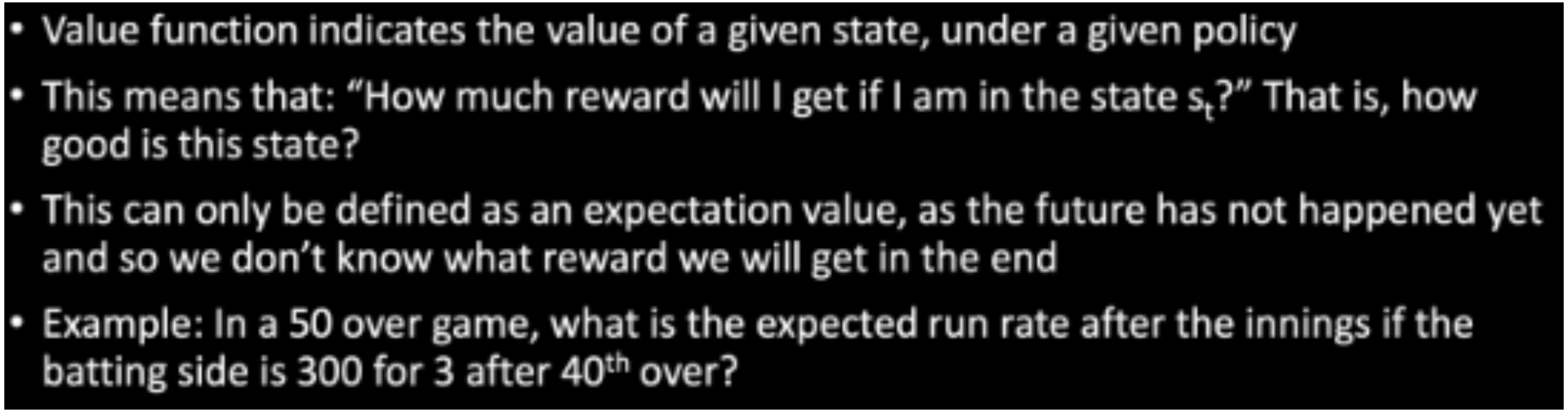
Temporal Difference Learning (TD Learning)

$$
\begin{aligned}
E_t &= V_t^* - V_t \\
&= r_{t+1} + \sum_{k=1}^{\infty} \gamma^k r_{t+k+1} - V_t \\
&= r_{t+1} + \gamma * \left(\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k+1} \right) - V_t \\
&= r_{t+1} + \gamma * \left(\sum_{k=0}^{\infty} \gamma^k r_{(t+1)+k+1} \right) - V_t \\
&= r_{t+1} + \gamma * V_{t+1} - V_t
\end{aligned}
$$

And similar to other optimization methods, the current value will be updated by its value + learning_rate * error:

$$V_t \leftarrow V_t + \alpha * E_t = V_t + \alpha * (r_{t+1} + \gamma * V_{t+1} - V_t)$$

Value Function



- Value function indicates the value of a given state, under a given policy
- This means that: "How much reward will I get if I am in the state s_t?" That is, how good is this state?
- This can only be defined as an expectation value, as the future has not happened yet and so we don't know what reward we will get in the end
- Example: In a 50 over game, what is the expected run rate after the innings if the batting side is 300 for 3 after 40th over?

- Value function V^π: expected discounted sum of future rewards under a particular policy π

$$V^\pi(s_t = s) = \mathbb{E}_\pi[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots | s_t = s]$$

- Discount factor γ weighs immediate vs future rewards
- Can be used to quantify goodness/badness of states and actions
- And decide how to act by comparing policies

Value of a state

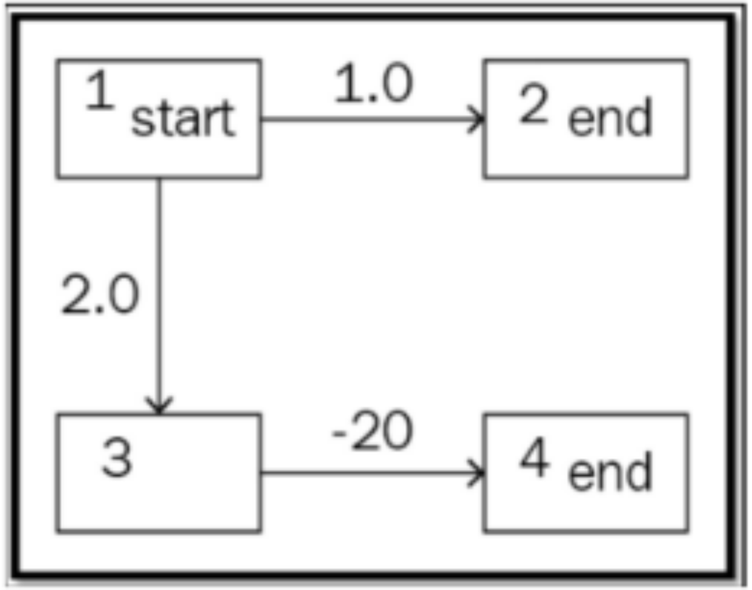
- An episode is a sequence of tuples: (s, a, r, s')
- The goal of RL is to maximize the cumulative reward over the episode
- Rewards in future are discounted with the hyperparameter gamma γ .
- Discounted total reward calculated at time t will be:

$$R_t = \sum_{i=t}^{\infty} \gamma^t r_t + \gamma^{t+1} r_{t+1} + \gamma^{t+2} r_{t+2} + \ldots$$

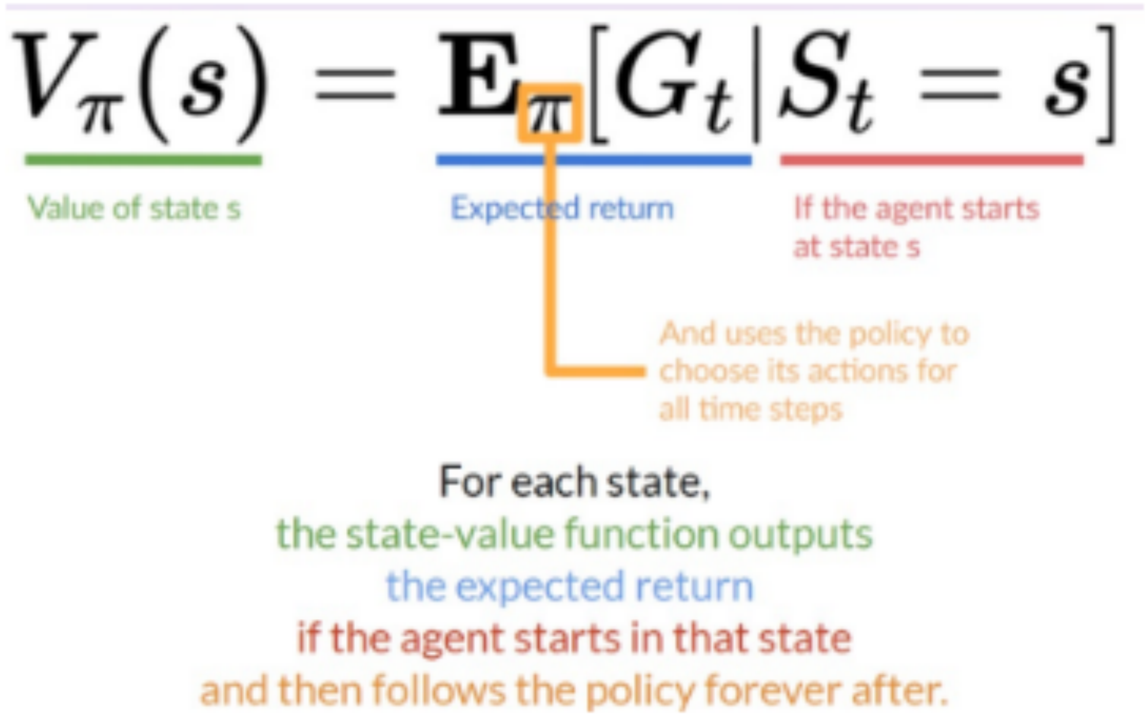
The value of a state s can be defined as: $V(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} r_t \gamma^t\right],$

Example : Value of a state 1

- The "always right" agent is the same: **1.0**
- The "always down" and "right" agent gets 2.0 + (–20) = **–18**
- The 50%/50% agent gets 0.5 * 1.0 + 0.5 * (2.0 + (–20)) = **–8.5**
- The 10%/90% agent gets 0.1 * 1.0 + 0.9 * (2.0 + (–20)) = **–16.1**



Value function of state



$$\underbrace{V_\pi(s)}_{\text{Value of state s}} = \underbrace{\mathbf{E}_{\boxed{\pi}}[G_t}_{\text{Expected return}}|\underbrace{S_t = s}_{\text{If the agent starts at state s}}]$$

Value of state s

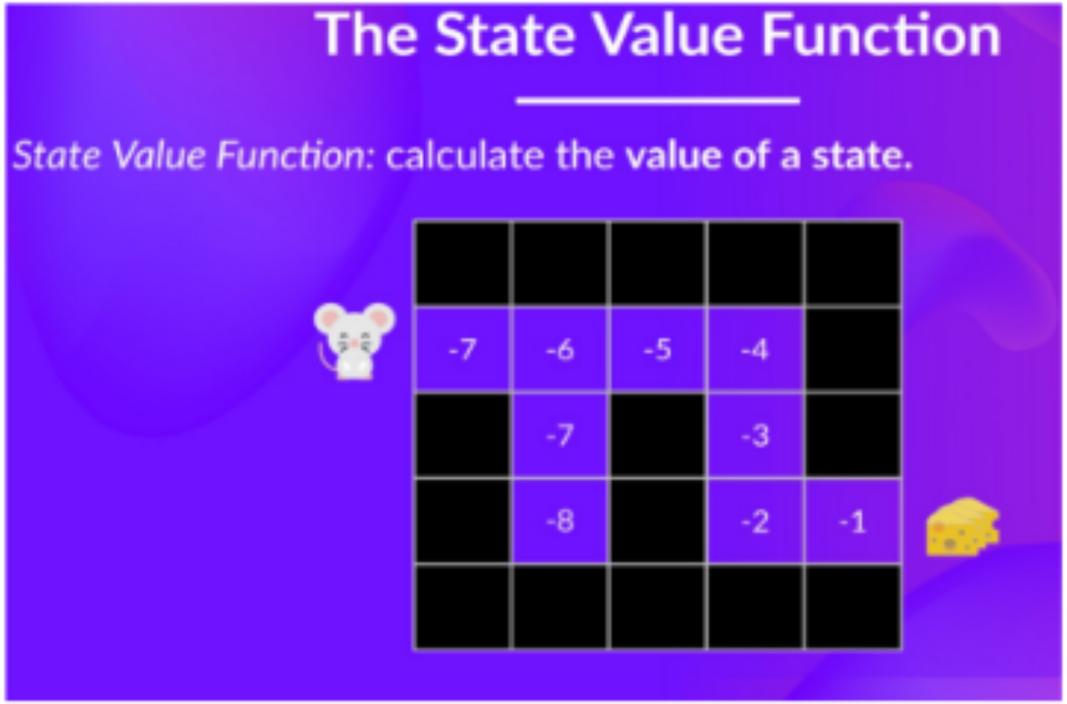
Expected return

If the agent starts at state s

And uses the policy to choose its actions for all time steps

For each state,
the state-value function outputs
the expected return
if the agent starts in that state
and then follows the policy forever after.

Value function of state


The State Value Function

State Value Function: calculate the **value of a state.**

■	■	■	■	■
-7	-6	-5	-4	■
■	-7	■	-3	■
■	-8	■	-2	-1
■	■	■	■	■

If we take the state with value -7: it's the expected return starting at that state and taking actions according to our policy (greedy policy), so right, right, right, down, down, right, right.

Value function and Q-value function

How good is a state?

The **value function** at state s, is the expected cumulative reward from following the policy from state s:

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

How good is a state-action pair?

The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Bellman equations

The idea of the Bellman equation is that instead of calculating each value as the sum of the expected return, which is a long process, we calculate the value as the sum of immediate reward + the discounted value of the state that follows.

Bellman equations

-> The foundation of many RL algorithms is the fact that value functions satisfy a recursive relationship, called the Bellman equation:

The optimal Q-value function Q* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Q* satisfies the following **Bellman equation**:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s, a\right]$$

Intuition: if the optimal state-action values for the next time-step Q*(s',a') are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s',a')$

Solving optimal Policy

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

Q_i will converge to Q^* as i -> infinity

What's the problem with this?

Not scalable. Must compute Q(s,a) for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate Q(s,a). E.g. a neural network!

Solving optimal Policy -> Q-learning

- It is beneficial to define another function that provides the value of the action, Q(s, a).
- This equals the total reward we can get by executing action a in state s and can be defined via V(s).
- Being a much less fundamental entity than V(s), this quantity gave a name to the whole family of methods called Q-learning, because it is more convenient.

Solving optimal Policy -> Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s',a') | s,a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[(y_i - Q(s,a;\theta_i))^2 \right]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) | s,a \right]$

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)) \nabla_{\theta_i} Q(s,a;\theta_i) \right]$$

Solving optimal Policy -> Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s,a\right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) | s,a\right]$

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q* (and optimal policy π^*)

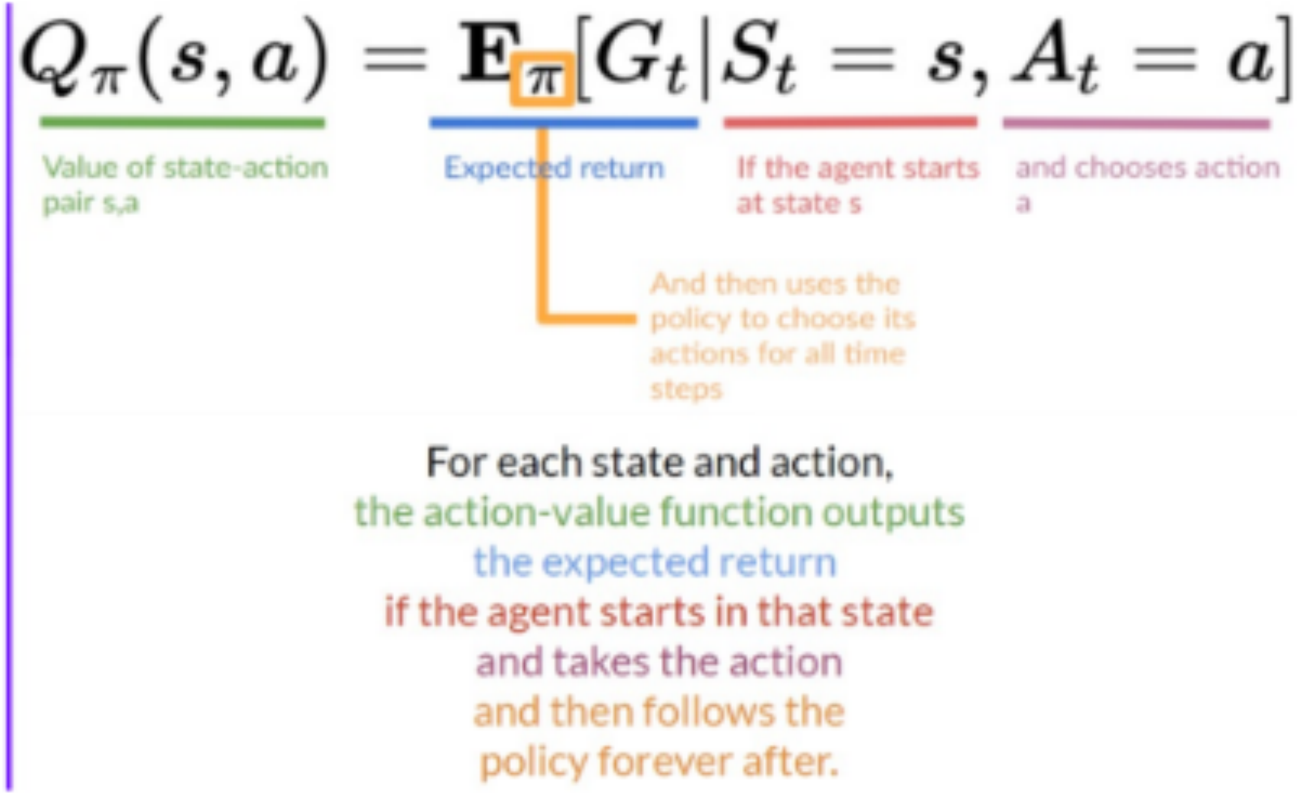
Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

UE22CS645BC2-TDL-Lecture - RL

Action value Function



$$\underbrace{Q_\pi(s,a)}_{\text{Value of state-action pair s,a}} = \underbrace{\mathbf{E}_{\boxed{\pi}}[G_t}_{\text{Expected return}}|\underbrace{S_t = s,}_{\text{If the agent starts at state s}}\ \underbrace{A_t = a]}_{\text{and chooses action a}}$$

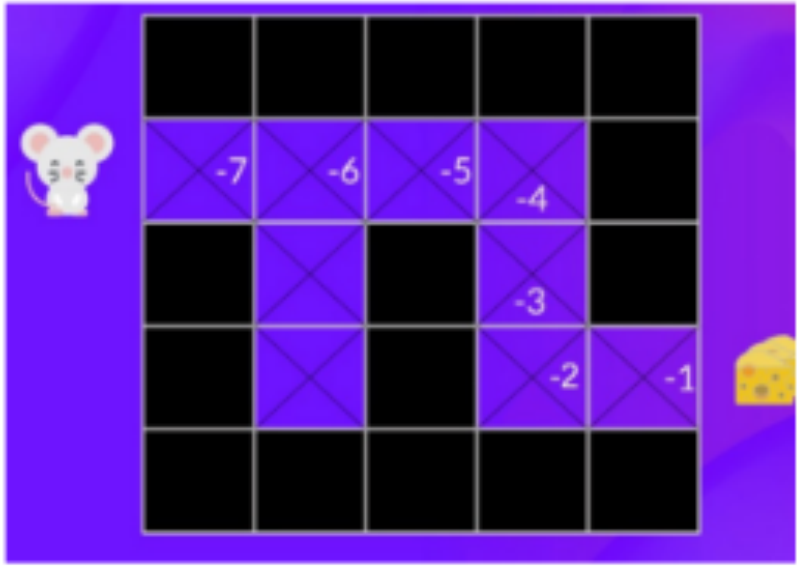
And then uses the policy to choose its actions for all time steps

For each state and action,
the action-value function outputs
the expected return
if the agent starts in that state
and takes the action
and then follows the
policy forever after.

Action value Function

- For the state-value function, we calculate **the value of a state** S_t

- For the action-value function, we calculate **the value of the state-action pair** (S_t, A_t) **hence the value of taking that action at that state.**

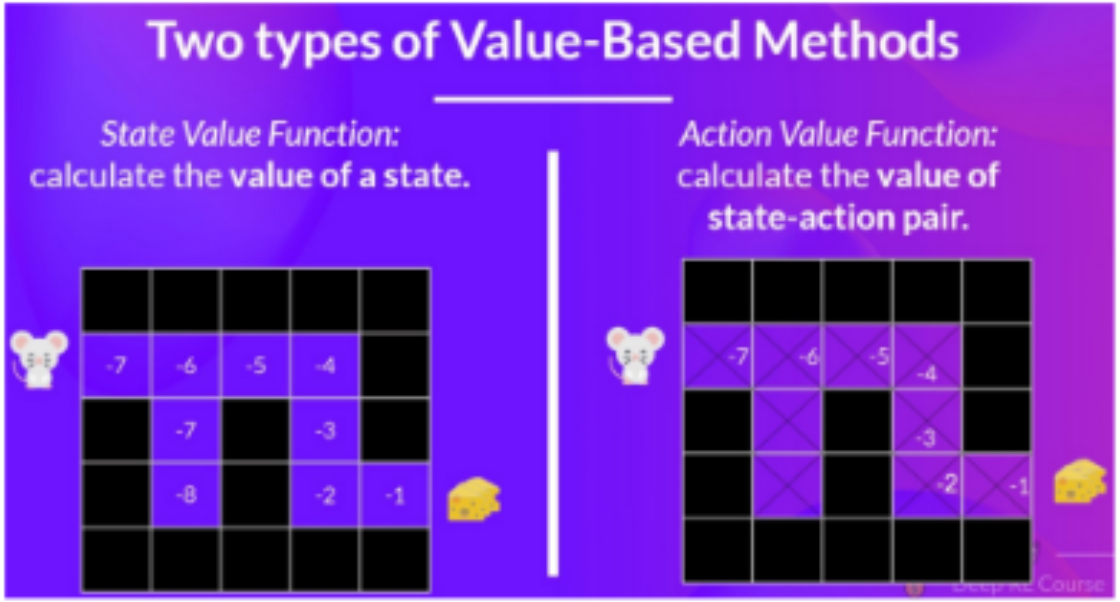


V(s) and Q(s, a) Functions

- Whether we choose state-value or action-value function, the returned value is the expected return.

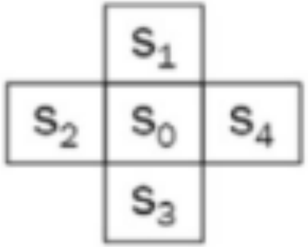
- In order to calculate EACH value of a state or a state-action pair, we need to sum all the rewards an agent can get if it starts at that state.

- This can be a computationally expensive process, and that's where the Bellman equation comes in to help us.


Two types of Value-Based Methods
State Value Function:
calculate the value of a state.
Action Value Function:
calculate the value of state-action pair.
-7
-6
-5
-4
-7
-3
-8
-2
-1
-7
-6
-5
-4
-3
-2
-1

Example: Simplified Frozenlake

- Consider an initial state surrounded by four other states as shown, each having different rewards
- Each action is probabilistic as discussed for Frozenlake environment (probability = 33%)
- Terminal states s_1, s_2, s_3, s_4 do not have outbound transitions and have rewards 1, 2, 3, 4 respectivel Hence: $V1 = 1$, $V2 = 2$, $V3 = 3$, $V4 = 4$
- We can compute Q values for s_0 for each action that can be taken from s_0 as below:



S_0 - initial state

S_1 S_2 S_3 S_4 - final states

$$Q(s_0, up) = 0.33 \cdot V_1 + 0.33 \cdot V_2 + 0.33 \cdot V_4 = 0.33 \cdot 1 + 0.33 \cdot 2 + 0.33 \cdot 4 = 2.31$$

$$Q(s_0, left) = 0.33 \cdot V_1 + 0.33 \cdot V_2 + 0.33 \cdot V_3 = 1.98$$

$$Q(s_0, right) = 0.33 \cdot V_4 + 0.33 \cdot V_1 + 0.33 \cdot V_3 = 2.64$$

$$Q(s_0, down) = 0.33 \cdot V_3 + 0.33 \cdot V_2 + 0.33 \cdot V_4 = 2.97$$

The final value for state s_0 is the maximum of those actions' values, which is 2.97.

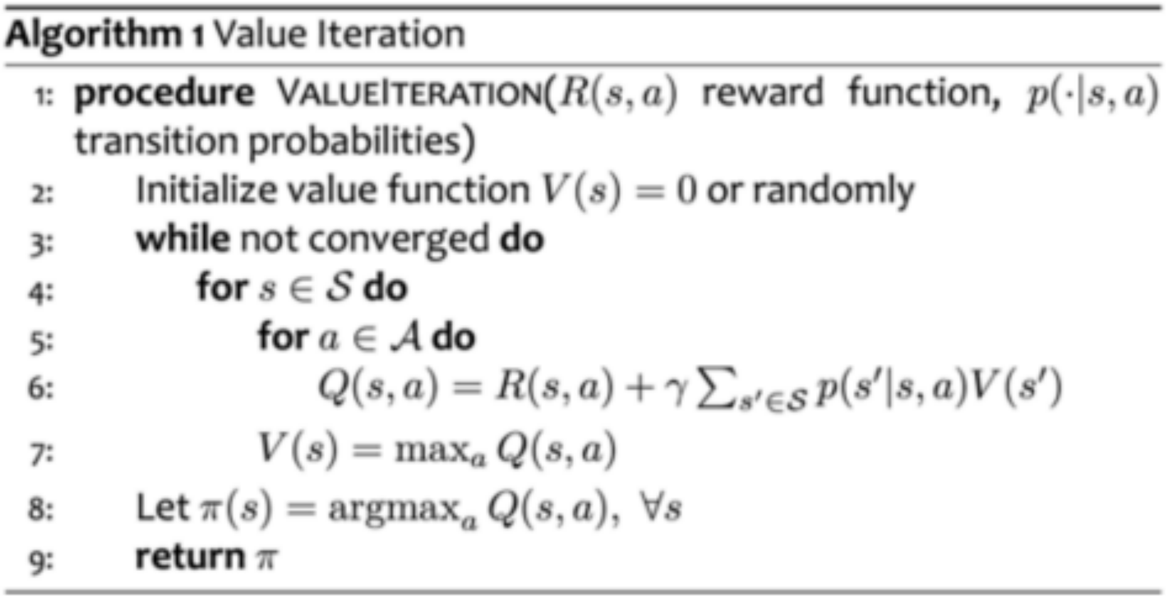
Value Iteration Algorithm

1. Initialize the values of all states, V_i, to some initial value (usually zero)

2. For every state, s, in the MDP, perform the Bellman update:

$$V_s \leftarrow \max_a \sum_{s'} p_{a,s\to s'}(r_{s,a} + \gamma V_{s'})$$

3. Repeat step 2 for some large number of steps or until changes become too small



Algorithm 1 Value Iteration

```
1: procedure VALUEITERATION($R(s,a)$ reward function, $p(\cdot|s,a)$
   transition probabilities)
2:     Initialize value function $V(s) = 0$ or randomly
3:     while not converged do
4:         for $s \in \mathcal{S}$ do
5:             for $a \in \mathcal{A}$ do
6:                 $Q(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)V(s')$
7:             $V(s) = \max_a Q(s,a)$
8:     Let $\pi(s) = \text{argmax}_a Q(s,a), \forall s$
9:     return $\pi$
```

Action-Value Iteration Procedure

1. Initialize every Qs, a to zero

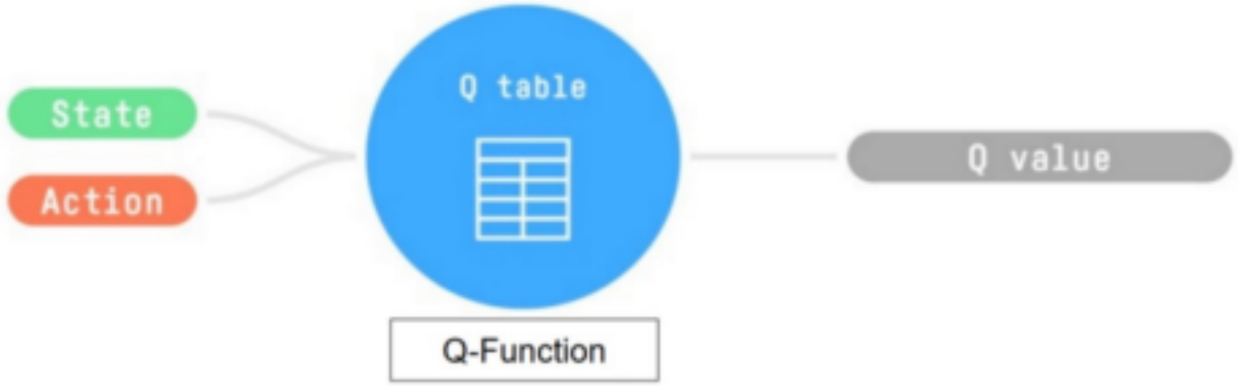
2. For every state, s, and action a, in this state, perform the update:

$$Q_{s,a} \leftarrow \sum_{s'} p_{a,s\rightarrow s'}(r_{s,a} + \gamma \max_{a'} Q_{s',a'})$$

3. Repeat step 2 for some large number of steps or until changes become too small

Tabular Q-Learning

- In the tabular Q-Learning, we use a Q-Table as a central element that maps state, action to a Q-Value
- Training the Q-Function is the process of learning the Q-Table by interacting with the environment
- The algorithm that is used to learn the Q-table is called the Q-Learning algorithm



Using Q Table to derive the optimal policy

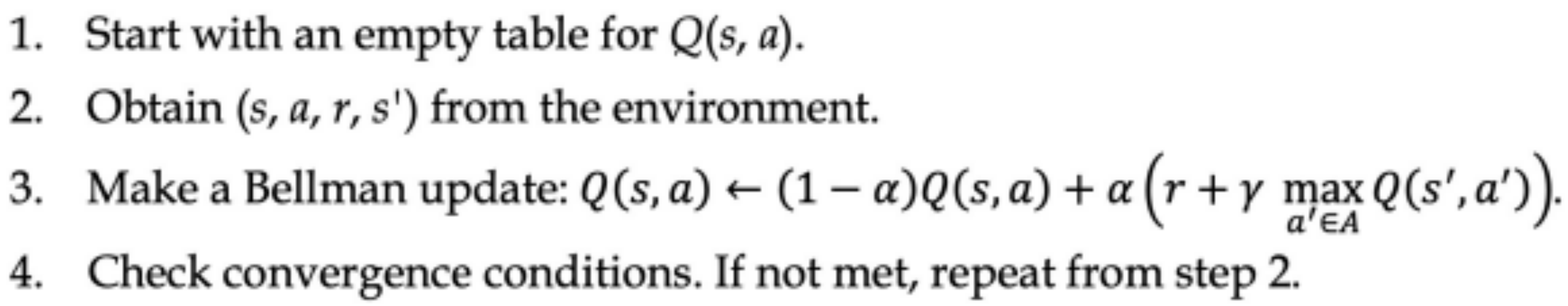
- Suppose we have a trained Q-Table, we then have the value for every possible state-action pair f: (s,a)→ q_value

- Given a (s, a) as input, the agent can retrieve the corresponding q value

- If we have an optimal Q Function represented in the Q Table, we have an optimal policy, since we can determine the best action that can be taken in each state.

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

Q-Learning Algorithm



1. Start with an empty table for $Q(s, a)$.
2. Obtain (s, a, r, s') from the environment.
3. Make a Bellman update: $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \max_{a' \in A} Q(s',a')\right)$.
4. Check convergence conditions. If not met, repeat from step 2.

Deep Q-Learning

- Tabular Q-Learning methods can solve simple problems where the states are discrete and number of states are not too many.
    - E.g. Frozen Lake problem is represented as 4 x 4 grid (16 states)

- Many real world problems may be represented as images and in those cases one may treat an image as an observation.
    - E.g. Atari games where an observation could be 210x160x3, which is 100800 pixels.

Total states = $256^{210x160x3} = 256^{100800}$

- It is intractable to handle such a huge state space representing them as state action tables

- **Key Idea:** Replace Q-Table with a Deep Neural Network that maps (s, a) in to a Q value

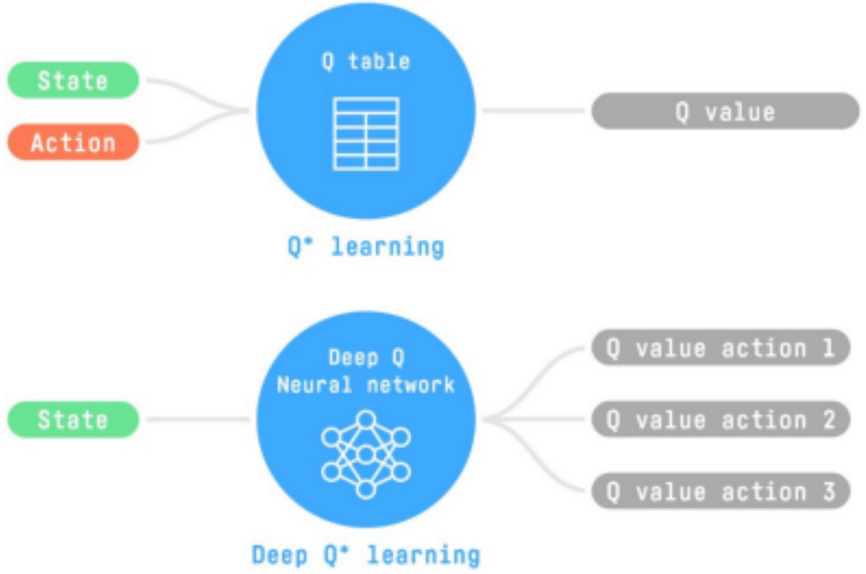
Deep Q-Learning

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

function parameters (weights)

If the function approximator is a deep neural network => **deep q-learning**!



Types of RL algorithms

- **Policy gradients:** directly differentiate the above objective

- **Value-based:** estimate value function or Q-function of the optimal policy (no explicit policy)

- **Actor-critic:** estimate value function or Q-function of the current policy, use it to improve policy

- **Model-based RL**: estimate the transition model, and then…
  - Use it for planning (no explicit policy)
  - Use it to improve a policy
  - Something else

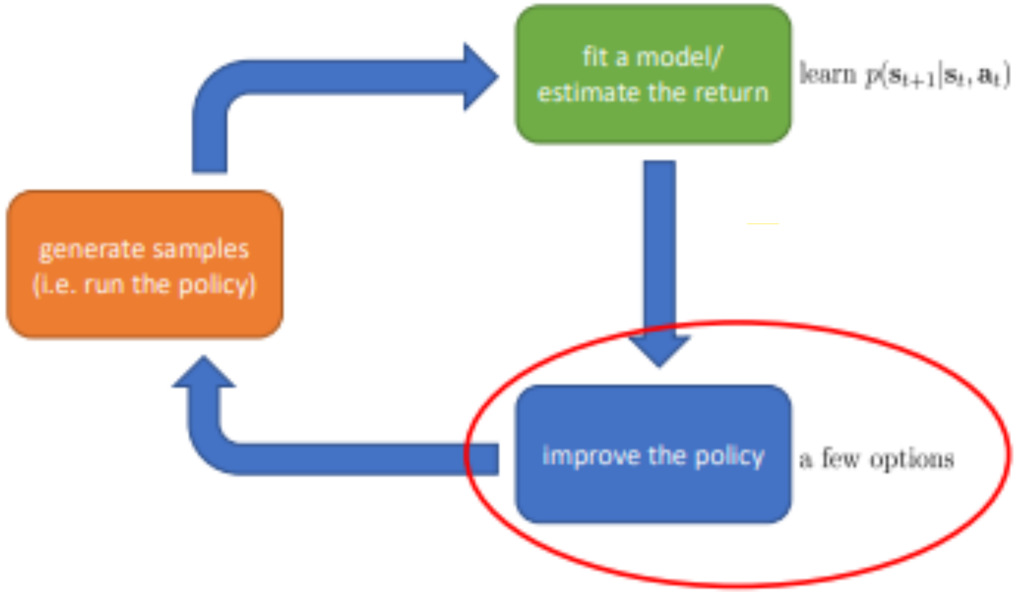
Model-based RL algorithms

A few options to improve policy

1. Just use the model to plan (no policy)
• Trajectory optimization/optimal control
(primarily in continuous spaces)
essentially backpropagation to optimize over a
• Discrete planning in discrete action spaces –
e.g., Monte Carlo tree search

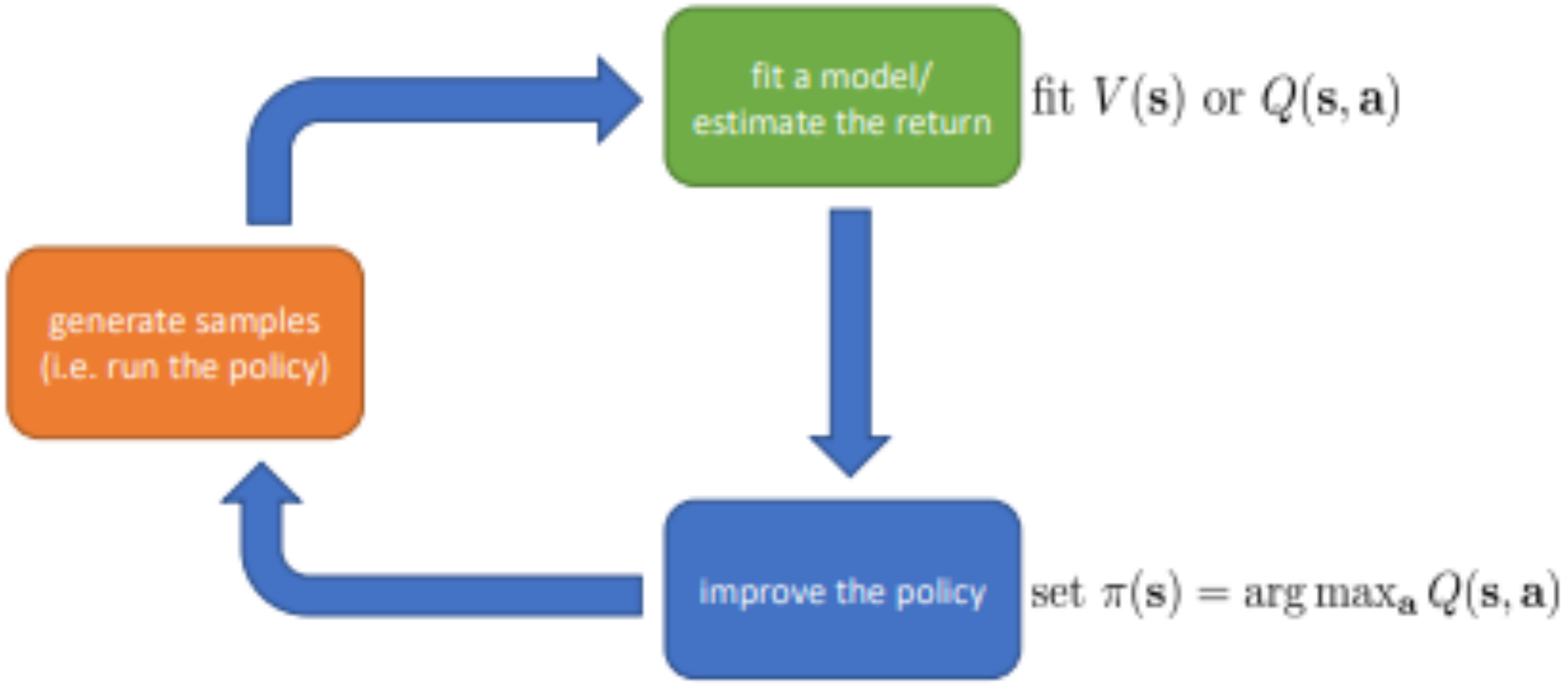
2. Backpropagate gradients into the policy
• Requires some tricks to make it work

3. Use the model to learn a value function
• Dynamic programming
• Generate simulated experience for model-free learner


fit a model/ estimate the return
learn $p(s_{t+1}|s_t, a_t)$
generate samples (i.e. run the policy)
improve the policy
a few options

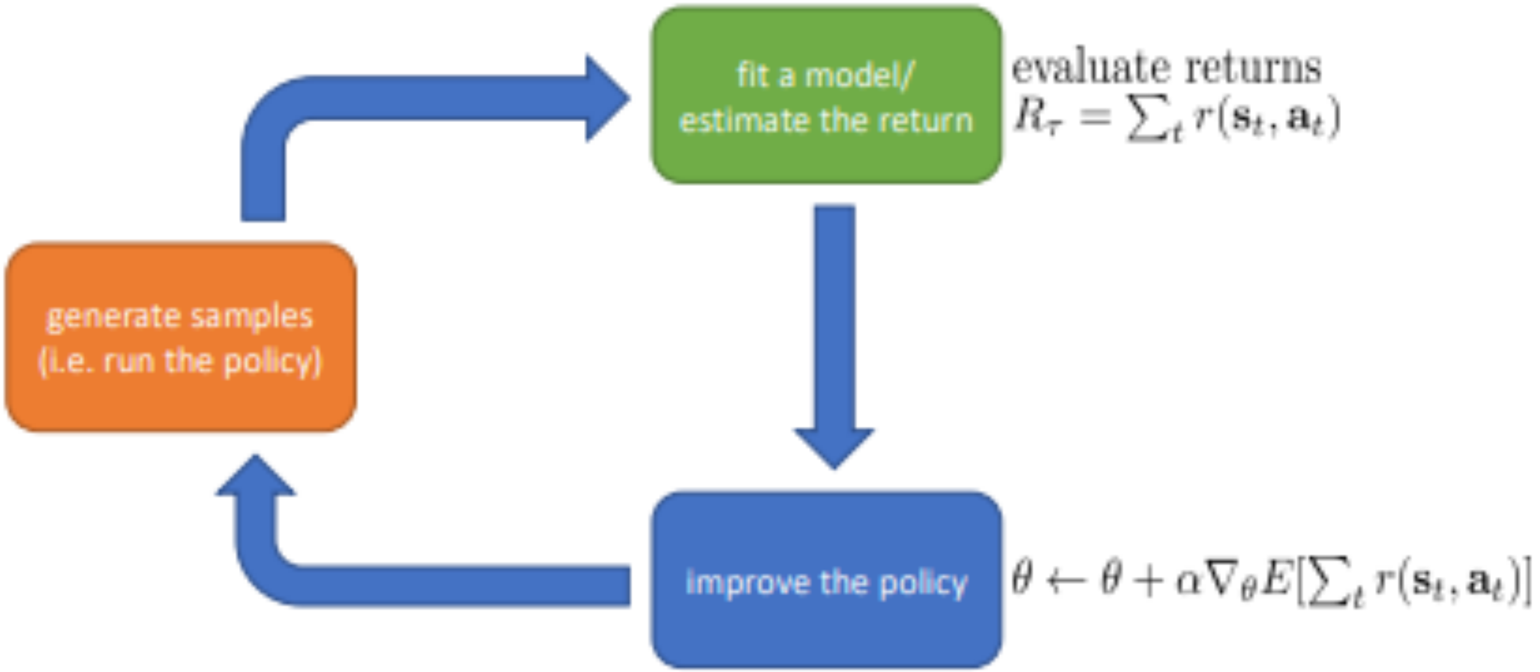
UE22CS645BC2-TDL-Lecture - RL

Value function based algorithms


fit a model/ estimate the return
fit $V(\mathbf{s})$ or $Q(\mathbf{s}, \mathbf{a})$
improve the policy
set $\pi(\mathbf{s}) = \arg\max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a})$
generate samples (i.e. run the policy)

UE22CS645BC2-TDL-Lecture - RL

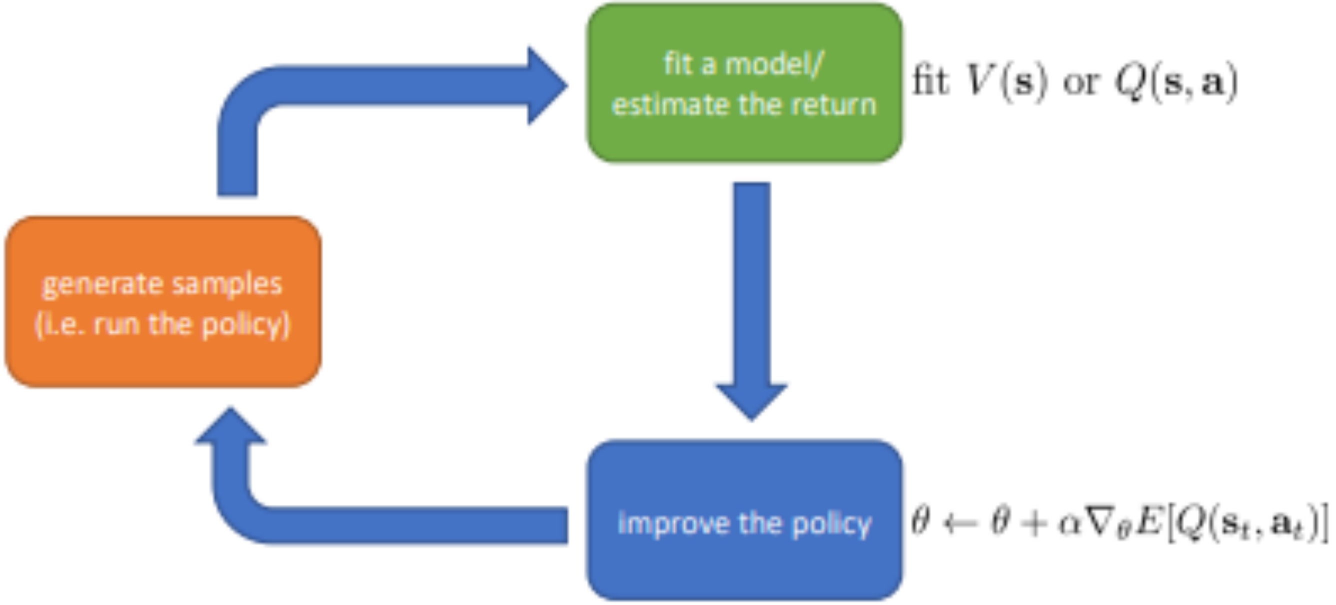
Direct policy gradients



fit a model/ estimate the return
evaluate returns $R_\tau = \sum_t r(\mathbf{s}_t, \mathbf{a}_t)$
improve the policy
$\theta \leftarrow \theta + \alpha \nabla_\theta E[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)]$
generate samples (i.e. run the policy)

UE22CS645BC2-TDL-Lecture - RL

Actor-critic: value functions + policy gradients

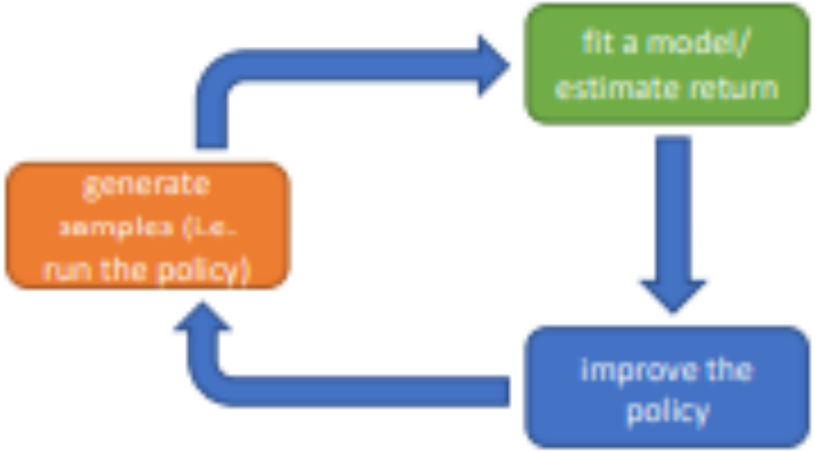

fit a model/ estimate the return
fit $V(\mathbf{s})$ or $Q(\mathbf{s}, \mathbf{a})$
improve the policy
$\theta \leftarrow \theta + \alpha \nabla_\theta E[Q(\mathbf{s}_t, \mathbf{a}_t)]$
generate samples (i.e. run the policy)

Tradeoffs Between Algorithms

Why so many RL algorithms?

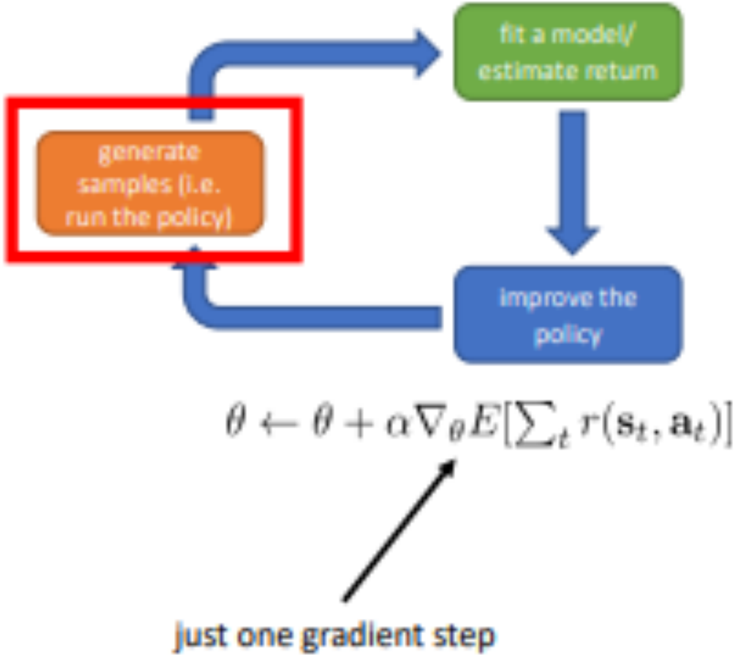
- Different tradeoffs
  - Sample efficiency
  - Stability & ease of use
- Different assumptions
  - Stochastic or deterministic?
  - Continuous or discrete?
  - Episodic or infinite horizon?

- Different things are easy or hard in different settings
  - Easier to represent the policy?
  - Easier to represent the model?


fit a model/ estimate return
improve the policy
generate samples (i.e. run the policy)

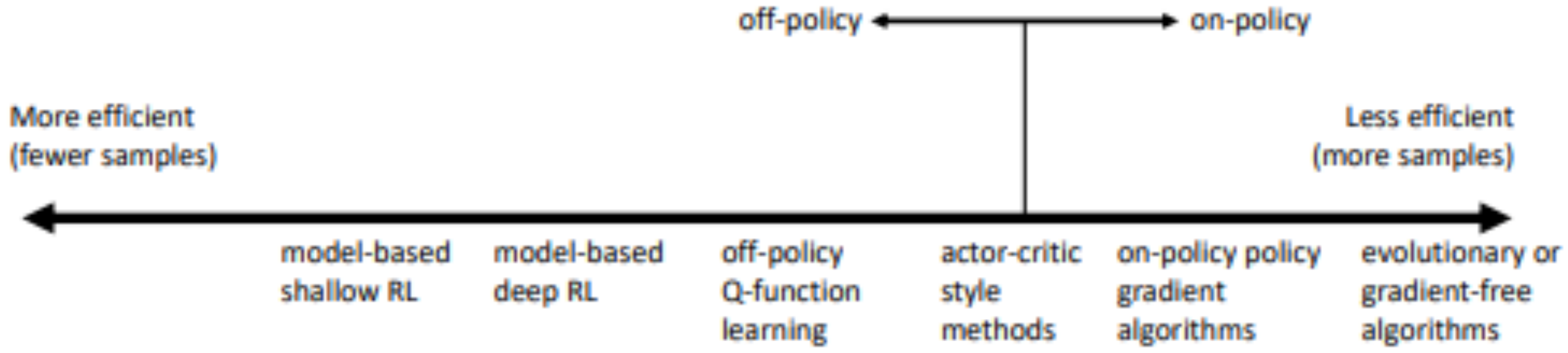
Comparison: sample efficiency

- Sample efficiency = how many samples do we need to get a good policy?
- Most important question: is the algorithm *off policy*?
  - Off policy: able to improve the policy without generating new samples from that policy
  - On policy: each time the policy is changed, even a little bit, we need to generate new samples


fit a model/ estimate return
generate samples (i.e. run the policy)
improve the policy
just one gradient step

$$\theta \leftarrow \theta + \alpha \nabla_\theta E[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)]$$

Comparison: sample efficiency


off-policy
on-policy
More efficient (fewer samples)
Less efficient (more samples)
model-based shallow RL
model-based deep RL
off-policy Q-function learning
actor-critic style methods
on-policy policy gradient algorithms
evolutionary or gradient-free algorithms

Why would we use a *less* efficient algorithm?

Wall clock time is not the same as efficiency!

Comparison: stability and ease of use

- Does it converge?
- And if it converges, to what?
- And does it converge every time?

Why is any of this even a question???

- Supervised learning: almost *always* gradient descent
- Reinforcement learning: often *not* gradient descent
  - Q-learning: fixed point iteration
  - Model-based RL: model is not optimized for expected reward
  - Policy gradient: *is* gradient descent, but also often the least efficient!

Comparison: stability and ease of use

- **Value function fitting**
  - At best, minimizes error of fit (“Bellman error”)
    - Not the same as expected reward
  - At worst, doesn’t optimize anything
    - Many popular deep RL value fitting algorithms are not guaranteed to converge to anything in the nonlinear case
- **Model-based RL**
  - Model minimizes error of fit
    - This will converge
  - No guarantee that better model = better policy

- **Policy gradient**
  - The only one that actually performs gradient descent (ascent) on the true objective

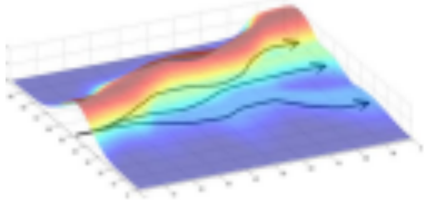
UE22CS645BC2-TDL-Lecture - RL

Comparison: assumptions

- Common assumption #1: **full observability**
  - Generally assumed by value function fitting methods
  - Can be mitigated by adding recurrence

- Common assumption #2: **episodic learning**
  - Often assumed by pure policy gradient methods
  - Assumed by some model-based RL methods

- Common assumption #3: **continuity or smoothness**
  - Assumed by some continuous value function learning methods
  - Often assumed by some model-based RL methods



Examples of specific algorithms

- Value function fitting methods
    - Q-learning, DQN
    - Temporal difference learning
    - Fitted value iteration
- Policy gradient methods
    - REINFORCE
    - Natural policy gradient
    - Trust region policy optimization
- Actor-critic algorithms
    - Asynchronous advantage actor-critic (A3C)
    - Soft actor-critic (SAC)
- Model-based RL algorithms
    - Dyna
    - Guided policy search

Example 1: Atari games with Q-functions

- Playing Atari with deep reinforcement learning, Mnih et al. ‘13

- Q-learning with convolutional neural networks

Example 2: robots and model-based RL

- End-to-end training of deep visuomotor policies, L.* , Finn* '16

- Guided policy search (model-based RL) for image-based robotic manipulation

Example 3: walking with policy gradients

- High-dimensional continuous control with generalized advantage estimation, Schulman et al. ‘16

- Trust region policy optimization with value function approximation

Example 4: robotic grasping with Q-functions

- QT-Opt, Kalashnikov et al. ‘18

- Q-learning from images for real-world robotic grasping

UE22CS645BC2-TDL-Lecture - RL

References:

- https://web.stanford.edu/class/cs234/slides/lecture1.pdf
- https://towardsdatascience.com/states-actions-rewards-the-intuition-behind-reinforcement-learning-33d4aa2bbfaa
- https://www.engati.com/glossary/temporal-difference-learning#:~:text=Temporal%20Difference%20Learning%20aims%20to,Dynamic%20Programming%20(DP)%20method.
- https://towardsdatascience.com/intro-to-reinforcement-learning-temporal-difference-learning-sarsa-vs-q-learning-8b4184bb4978

PES UNIVERSITY

Thank You

Dr. Shylaja S S
Director of Cloud Computing & Big Data (CCBD), Centre for Data Sciences & Applied Machine Learning (CDSAML)
Department of Computer Science and Engineering
shylaja.sharath@pes.edu

Ack:Divija L
Teaching Assistant