ML
MU
KOSICE

http://bit.ly/mlmu-fb-group

http://bit.ly/mlmu-slack

CEAi

IT VALLEY
KOŠICE

T · · ·Systems·

GlobalLogic®

Thank you for the support!

14 Ke 14 Eastcubator

Part I

- RL Formalism
- RL Framework:
    - Objective of Decision Making Agent
    - Policies
    - State-Value Function
    - Optimality
    - Action-Value Function
    - Optimal Policy
    - Policy Evaluation
    - Policy Improvement
    - Policy Iteration

- Monte Carlo Methods
- Temporal Difference Methods - SARSA, Q-Learning

Reinforcement Learning (RL)

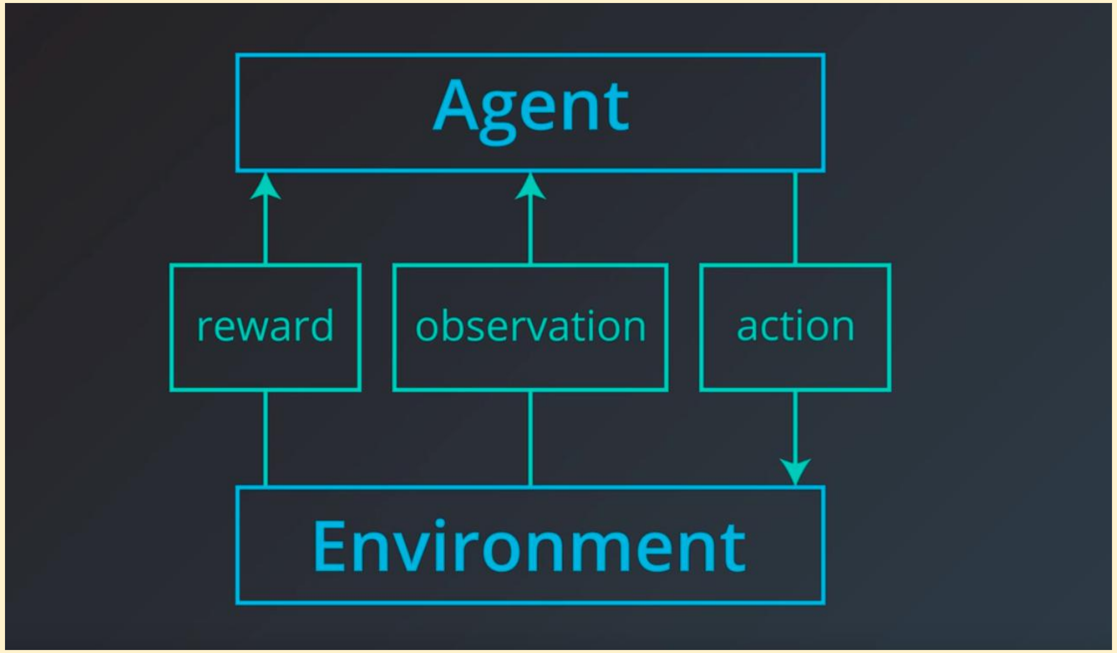
Reinforcement Learning is a subfield of machine learning which addresses the **problem of automatic learning of optimal decisions over time.**

RL is the task of learning through interaction. In this type of task, no human labels data and no human collects or designs the collection of data. These machine learning algorithms can be thought of as agents because of the need for interaction. The agents need to learn to perform a specific task, like in other machine learning paradigms. They also need to collect the most relevant data.

Very often, in RL, you must provide a reward signal. This signal is fundamentally different from the labels in supervised learning. In RL, agents receive reward signals for achieving a goal and not for specific agent behaviors. Additionally, this signal is related to an obviously desired state like winning a game, reaching an objective or location, and so on, which means that humans do not need to intervene by labeling millions of samples.

RL Formalism

Major RL entities: **Agent and Environment**
Communication channels: **Actions, Reward, and Observations**



RL Formalism

Reward
It's just a scalar value (number) we obtain periodically from the environment. It can be positive or negative, large or small. **The purpose of a reward is to give an agent feedback about its success**, and it's an important central thing in RL. **Basically, the term reinforcement comes from the fact that a reward obtained by an agent should reinforce its behavior in a positive or negative way.** Reward is local, meaning, it reflects the success of the agent's recent activity, not all the successes achieved by the agent so far.

The agent
It's somebody or something who/which interacts with the environment by executing certain actions, taking observations, and receiving eventual rewards for this. In most practical RL scenarios, it's our piece of software that is supposed to solve some problem in a more-or-less efficient way.

RL Formalism

Environment
The environment is everything outside of an agent.The environment is external to an agent, and its communication with the environment is limited by rewards (obtained from the environment), actions (executed by the agent and given to the environment), and observations (some information besides the rewards that the agent receives from the environment).

Actions
Actions are things that an agent can do in the environment. In RL, we distinguish between two types of actions: discrete or continuous.

Discrete actions form the finite set of mutually exclusive things an agent could do, such as move left or right.

Continuous actions have some value attached to the action, such as a car's action steer the wheel having an angle and direction of steering.

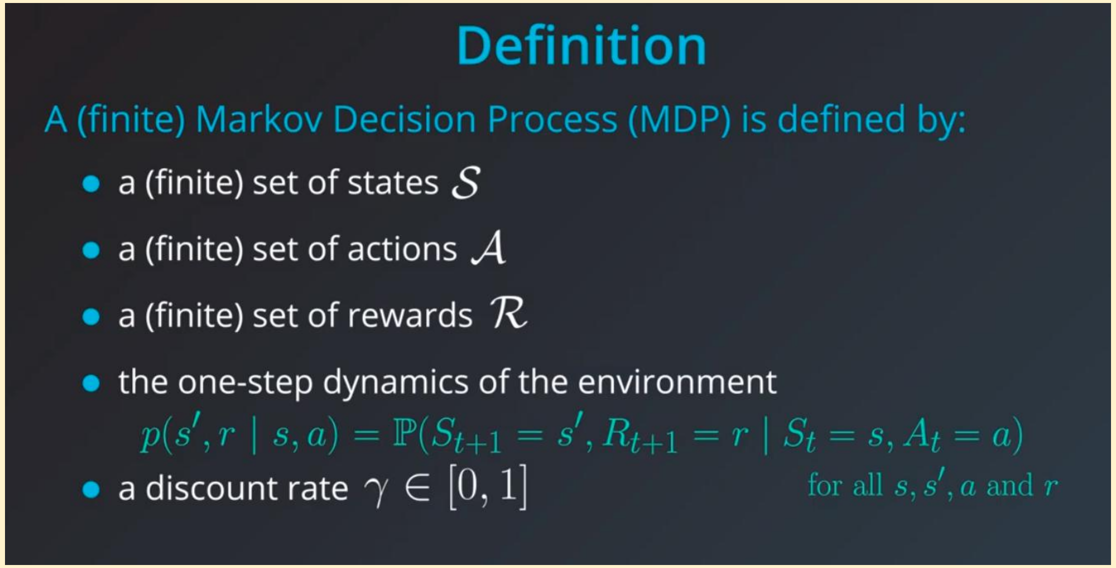
RL Formalism

States/Observations
State/observation of the environment is the second information channel for an agent, with the first being a reward.

State/Observations can be as simple as a bunch of numbers or as complex as several multidimensional tensors containing color images from several cameras, position, velocity, angle and so on. An observation can even be discrete, much like action spaces. An example of such a discrete observation space could be a light bulb, which could be in two states: on or off, given to us as a Boolean value.

Markov Decision Process

The combination of these components—a set of states, a set of actions, the representation of state changes and reward returns as a consequence of agent actions make up a framework known as **Markov Decision Processes** (MDP) and are commonly used to build sequential decision-making problems.



Definition

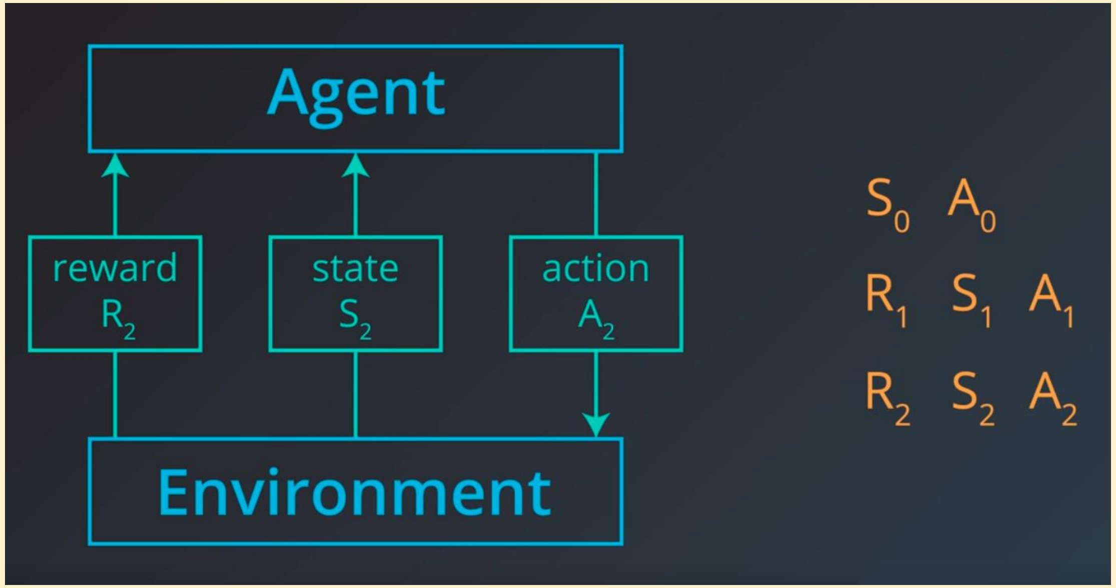
A (finite) Markov Decision Process (MDP) is defined by:

- a (finite) set of states $\mathcal{S}$
- a (finite) set of actions $\mathcal{A}$
- a (finite) set of rewards $\mathcal{R}$
- the one-step dynamics of the environment
  $p(s', r \mid s, a) = \mathbb{P}(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a)$
- a discount rate $\gamma \in [0, 1]$

for all s, s', a and r

RL Framework: Objective of a Decision Making Agent

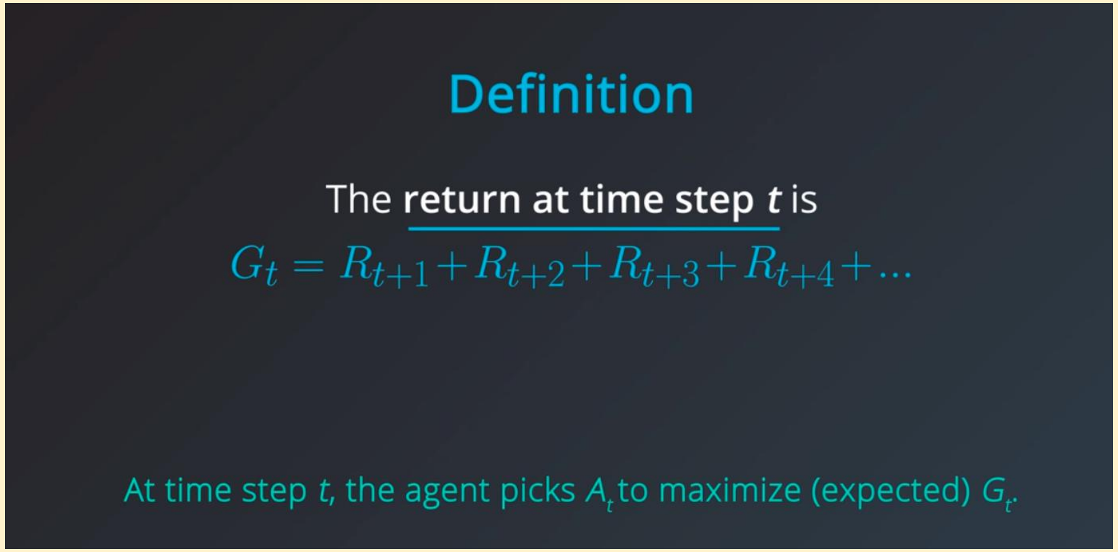
Reinforcement learning is an interaction cycle between an agent, and an environment. The cycle begins with the agent observing the environment. The agent does some internal processing of the observation like learning or memorizing. The agent then takes an action that will affect the environment in some way.



We start from the very beginning at timestep zero. The agent first receives the environment state which we denote by S0, where zero stands for a timestep zero of course. Then, based on that observation the agent chooses an action, A0. The environment transitions to a new state, S1, and gives some reward, R1, to the agent. The agent then chooses an action, A1.

RL Framework: Objective of a Decision Making Agent

At an arbitrary time step t, the agent-environment interaction evolves as a sequence of **states, actions, and rewards. The reward will always be the most relevant quantity to the agent**.



Definition

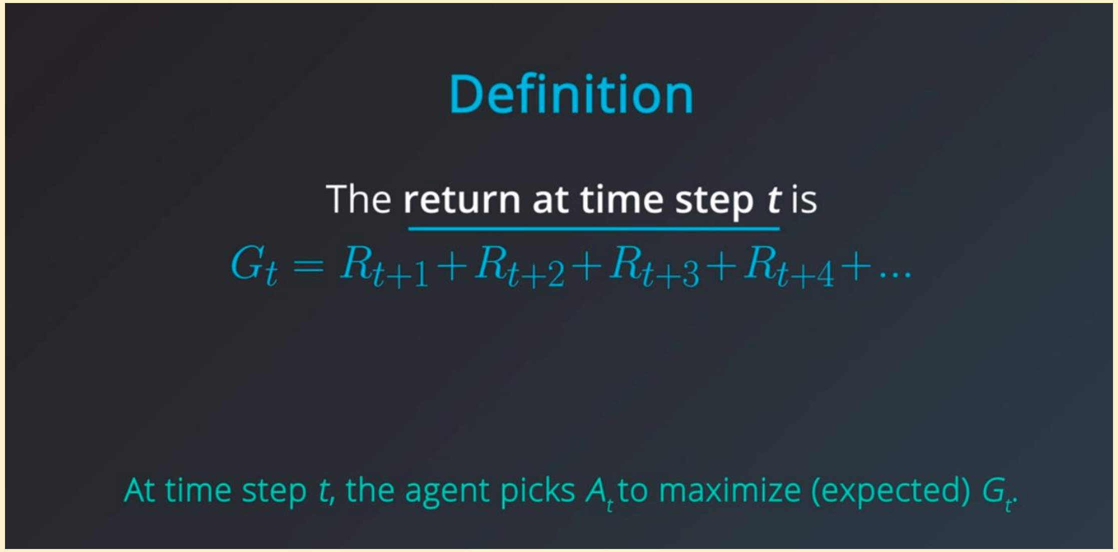
The **return at time step *t*** is

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4} + \ldots$$

At time step t, the agent picks A_t to maximize (expected) G_t.

RL Framework: Objective of a Decision Making Agent

The agent’s goal is to find a sequence of actions that will **maximize the sum of rewards** during the course of an episode or the entire life of the agent, depending on the task. The collection of rewards in a trajectory is called **expected return**.



Definition

The **return at time step *t*** is

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4} + \ldots$$

At time step t, the agent picks A_t to maximize (expected) G_t.

RL Framework: Objective of a Decision Making Agent

- The agent seeks to find the strategy for choosing actions with the cumulative reward is likely to be high.

- The agent can only accomplish this by interacting with the environment. This is because at every timestep, the environment decides how much reward the agent receives. In other words, the agent must play by the rules of the environment.

- Through interaction, the agent can learn those rules and choose appropriate actions to accomplish its goal.

- It's important to emphasize that all of this is just a mathematical model for a real world problem.

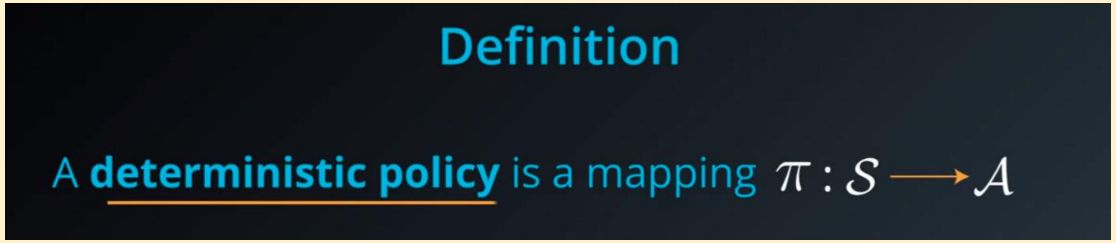
RL Framework: Policies

What the agent needs to come up with is called a **policy**.

A policy is a function that returns an action for any given nonterminal state.

Policies can be:

Deterministic - the policy maps states to actions.



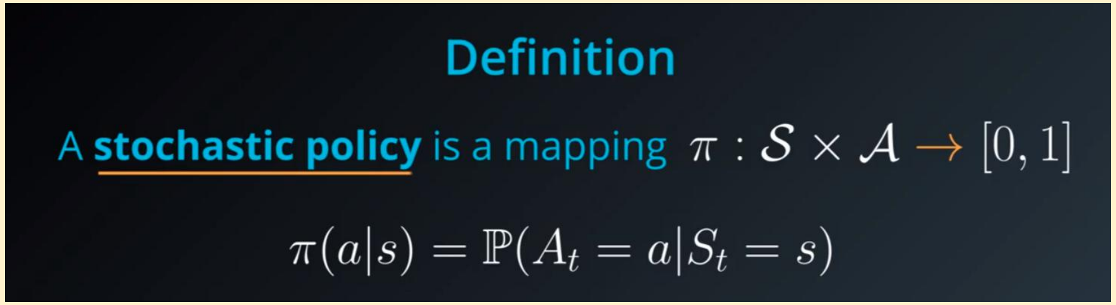
Definition

A **deterministic policy** is a mapping $\pi : \mathcal{S} \longrightarrow \mathcal{A}$

For each state s∈S, it yields the action a∈A that the agent will choose while in state s.

RL Framework: Policies

Stochastic - the policy will map a state to probabilities of selecting each possible action at that state



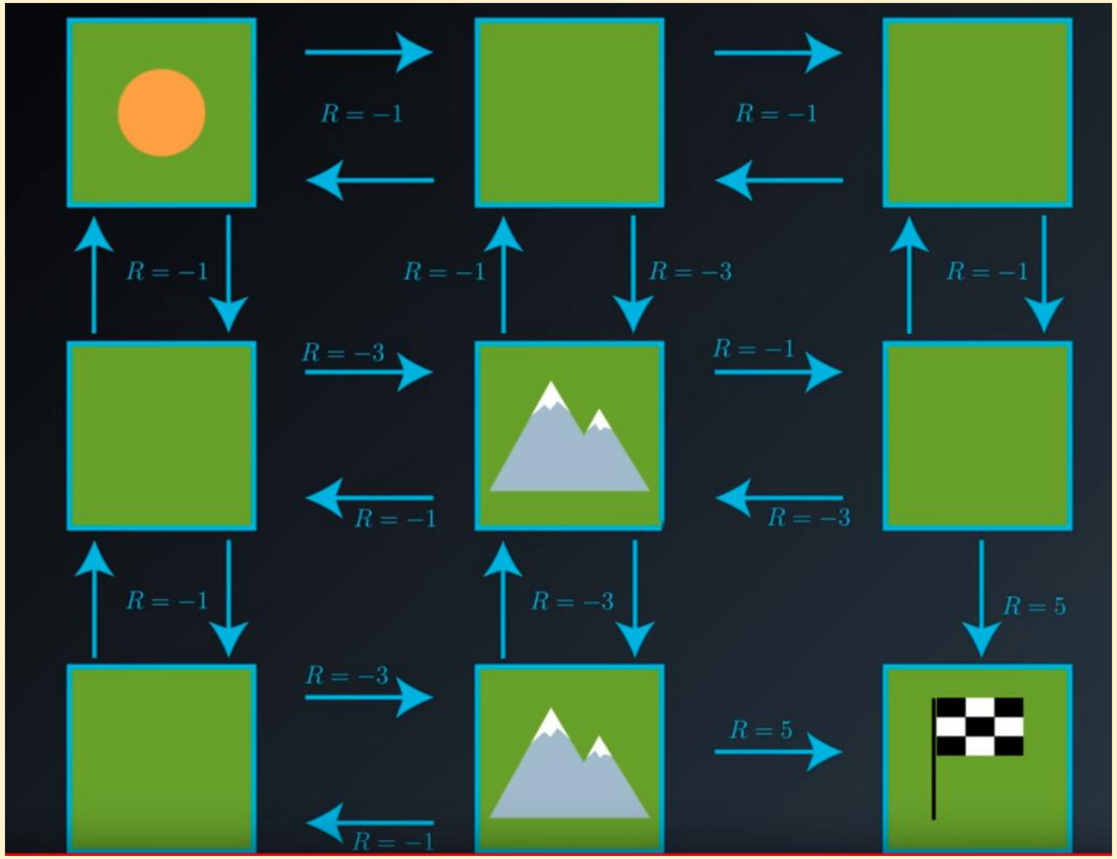
Definition

A **stochastic policy** is a mapping $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$

$$\pi(a|s) = \mathbb{P}(A_t = a|S_t = s)$$

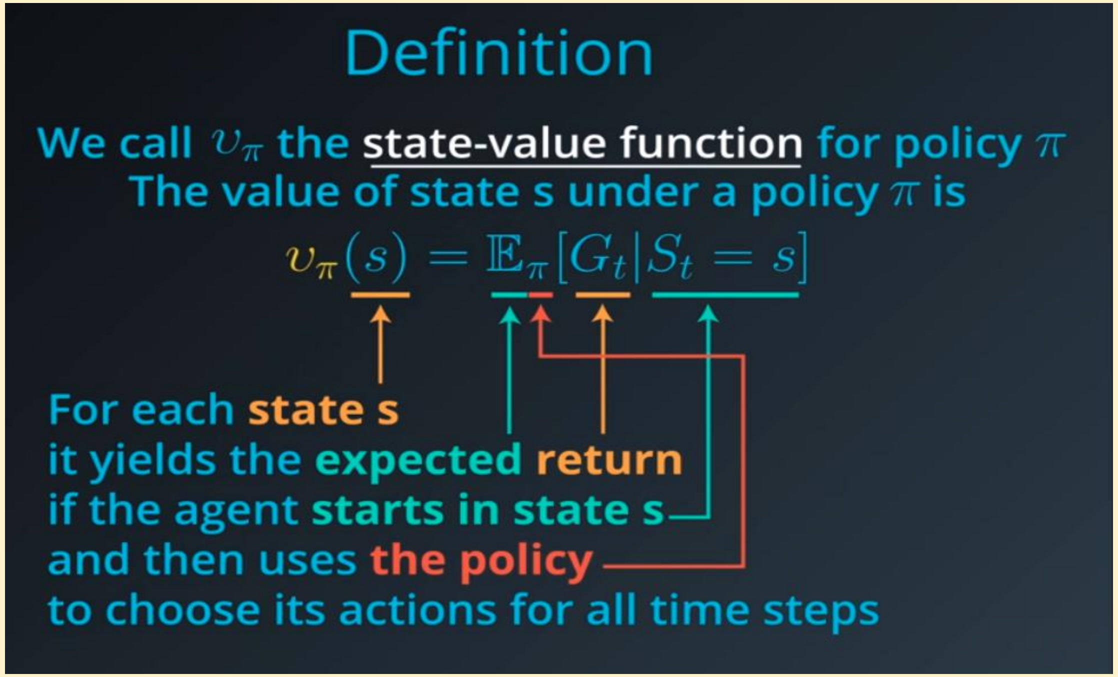
For each state s∈S and action a∈A, it yields the probability π(a|s) that the agent chooses action a while in state s.

Gridworld Example



RL Framework: State-Value Function

We have defined **expected returns** as a **function of future rewards** that the agent is trying to maximize. We now define the **value of states following a policy**: the value of a state s under policy π is the expectation of returns if the agent follows policy π starting from state s. This definition is called the **state-value function** and it simply represents the return an agent can expect to receive given an environment state and following a given policy.



Definition

We call v_π the state-value function for policy π
The value of state s under a policy π is

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

For each state s
it yields the expected return
if the agent starts in state s
and then uses the policy
to choose its actions for all time steps

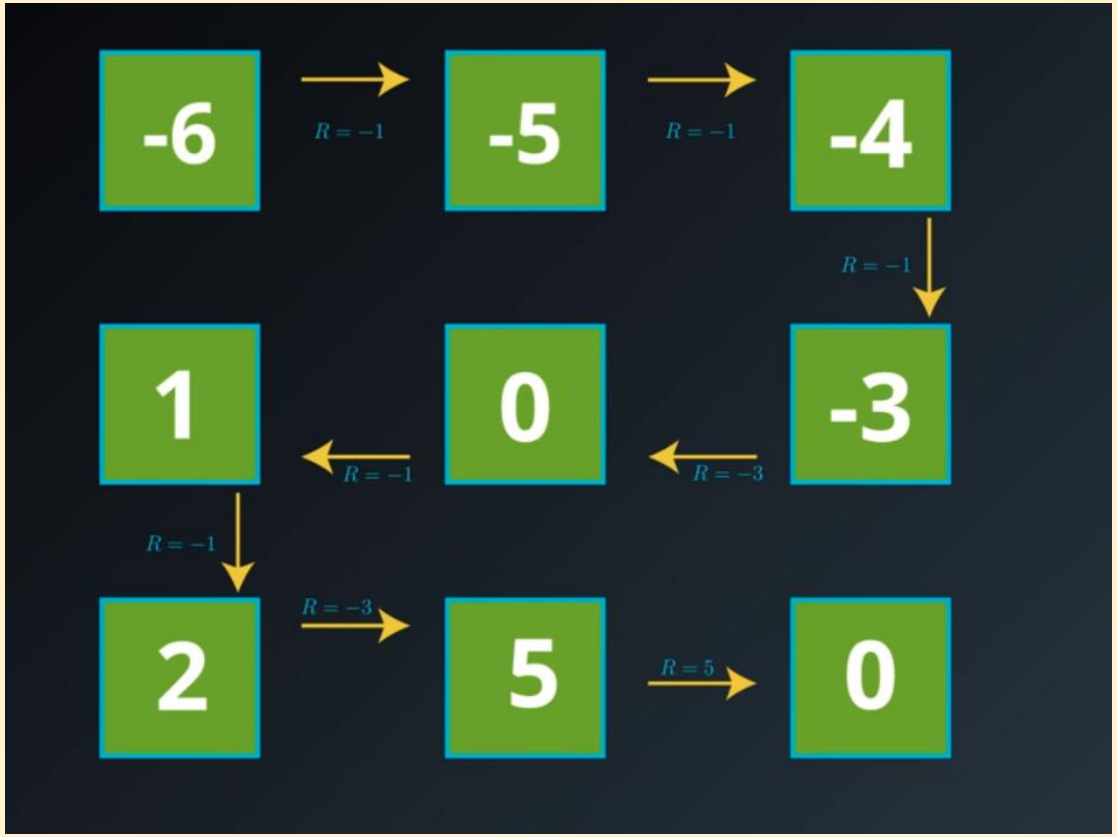
RL Framework: State-Value Function

For a general MDP, we have to work in terms of an expectation, since it's not often the case that the immediate reward and next state can be predicted with certainty. In fact, the reward and next state are chosen according to the one-step dynamics of the MDP.

In case, where the reward r and next state s' are drawn from a (conditional) probability distribution p(s',r|s,a), the **Bellman Expectation Equation expresses the value of any state s in terms of the expected immediate reward and the expected value of the next state:**

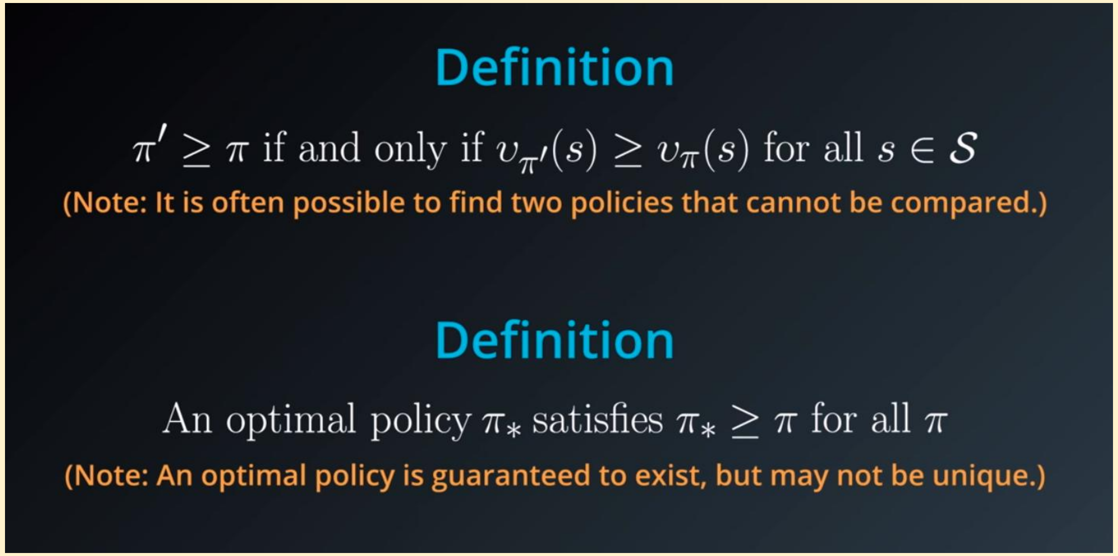
$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s].$$

Gridworld Example - State Values



RL Framework: Optimality

A policy π is defined better than or equal to policy π’ if the expected return is better than or equal to π’ for all states.



Definition

$\pi' \geq \pi$ if and only if $v_{\pi'}(s) \geq v_{\pi}(s)$ for all $s \in \mathcal{S}$

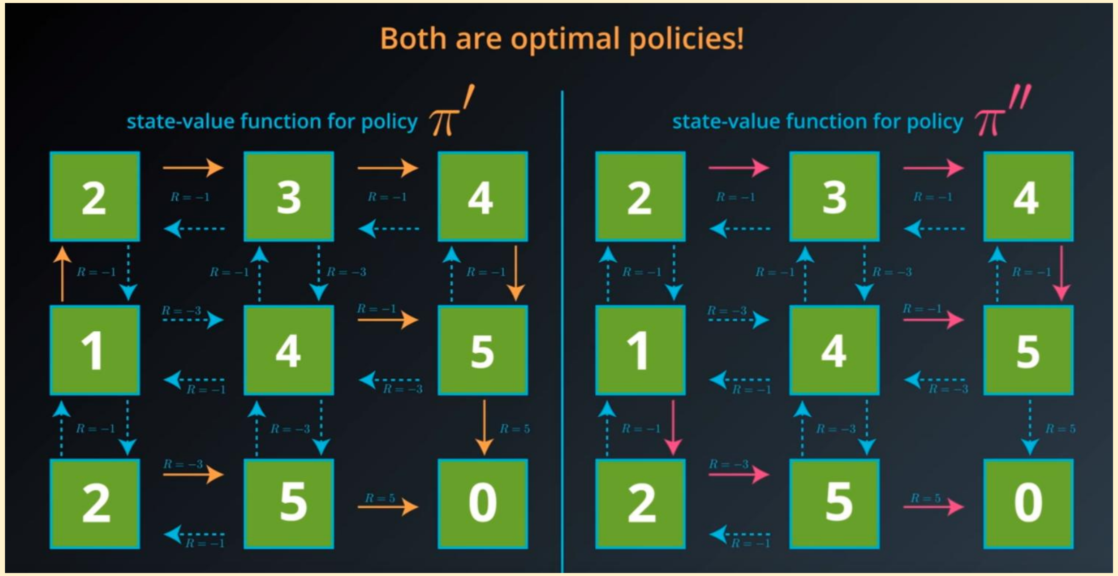
(Note: It is often possible to find two policies that cannot be compared.)

Definition

An optimal policy π_* satisfies $\pi_* \geq \pi$ for all π

(Note: An optimal policy is guaranteed to exist, but may not be unique.)

Different Policies with Equal Value Functions

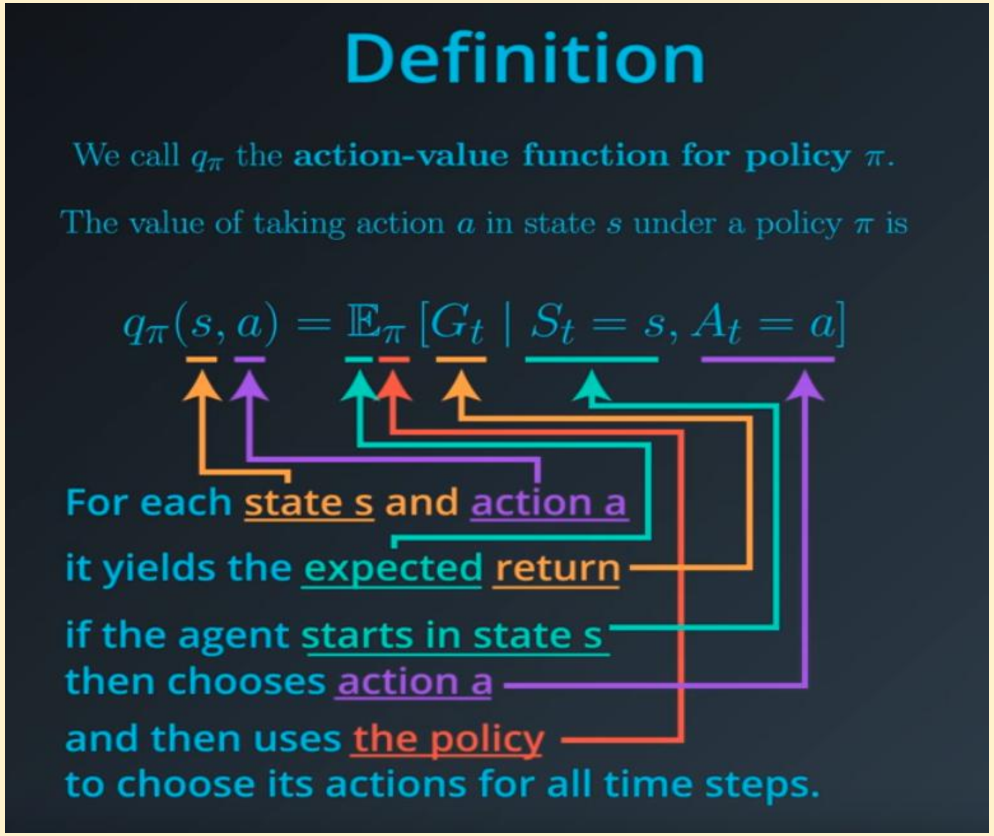


Both are optimal policies!

state-value function for policy π'
2 3 4
1 4 5
2 5 0
state-value function for policy π''
2 3 4
1 4 5
2 5 0
R = −1
R = −3
R = 5

RL Framework: Action-Value Function

The value of a state depends on the value of taking actions in that state. **Therefore, we need a way to compare the values of taking any action in any state, so that we can identify the best action and thereby come up with the best policy.** The action-value function is the expectation of returns if the agent follows policy π starting from state s and taking action a.



Definition

We call q_π the **action-value function for policy** π.

The value of taking action a in state s under a policy π is

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right]$$

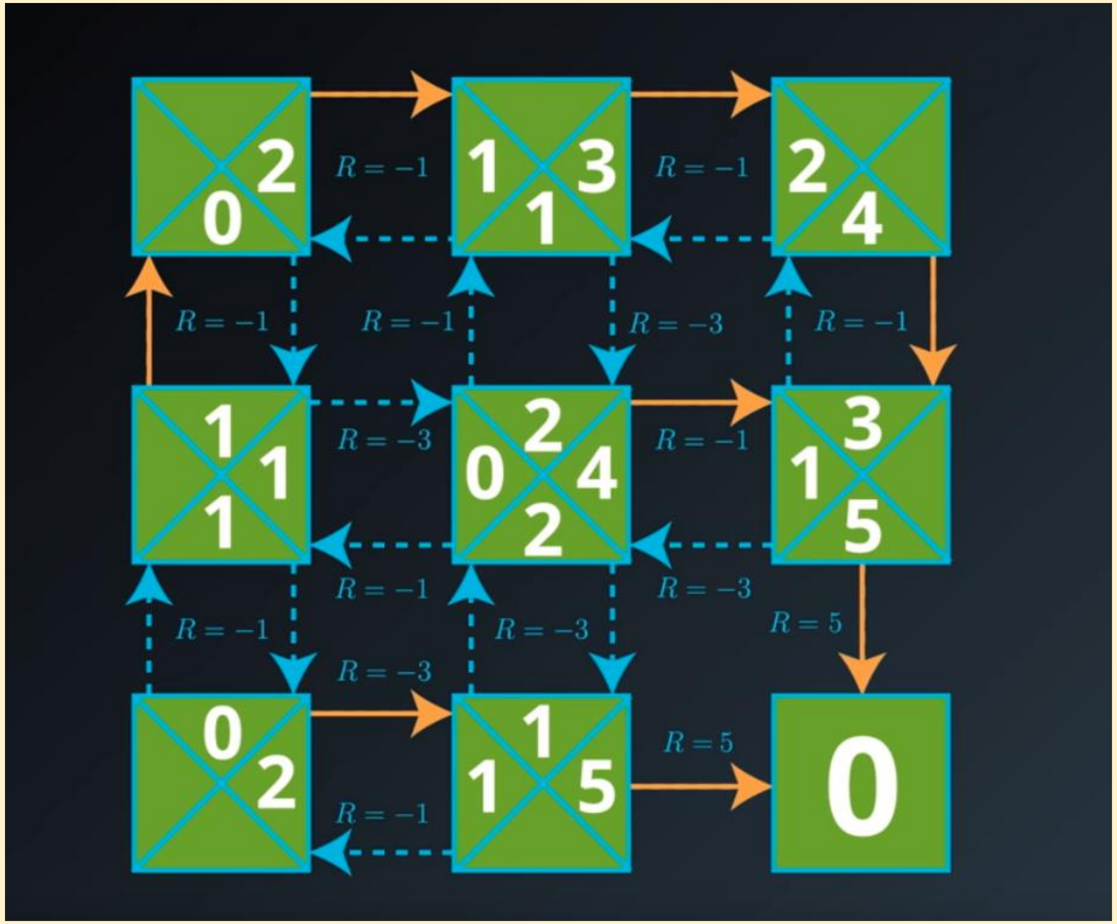
For each state s and action a

it yields the expected return

if the agent starts in state s
then chooses action a

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

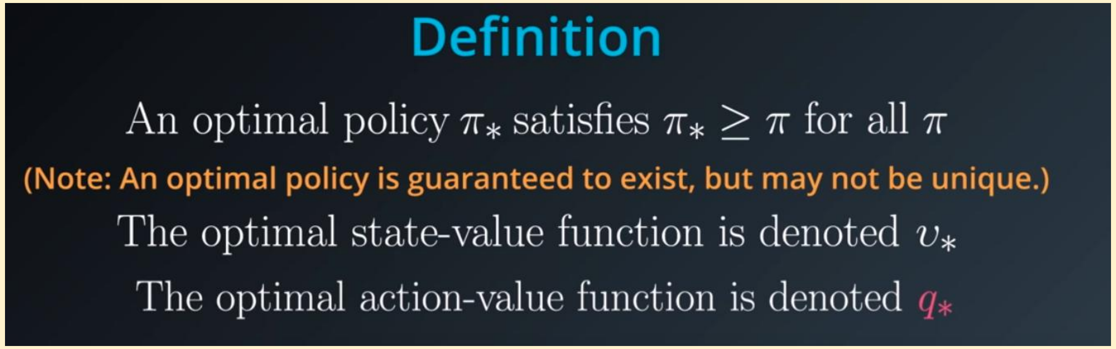
Gridworld Example - Action Values



RL Framework: Optimal Policy

The optimal policy π$_*$ specifies for each environment state how the agent should select an action towards its goal of maximizing reward. The agent could structure its search for an optimal policy by first estimating **the optimal action-value function q$_{*,}$ then, once q$_*$ is known, π$_*$ is obtained =>**

$$\pi_*(s) = \arg\max_{a \in \mathcal{A}(s)} q_*(s, a)$$



Definition

An optimal policy π_* satisfies $\pi_* \geq \pi$ for all π

(Note: An optimal policy is guaranteed to exist, but may not be unique.)

The optimal state-value function is denoted v_*

The optimal action-value function is denoted q_*

RL Framework: Optimal Policy

An optimal policy is guaranteed to exist but may not be unique.

All optimal policies have the same state-value function v_*, called the optimal state-value function.

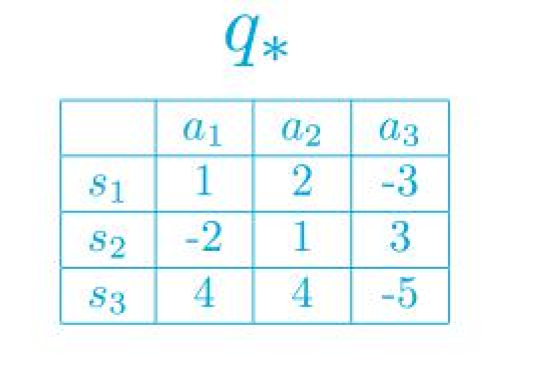
All optimal policies have the same action-value function q_*, called the optimal action-value function.

RL Framework: Optimal Policy

- If the state space S and action space A are finite, we can represent the optimal action-value function q_* in a table, where we have one entry for each possible environment state s∈S and action a∈A.

- This is called a **Q-table** (“Q” for “quality” of the action). The columns will be the actions. The rows will be the states.

- The value for a particular state-action pair s,a is the expected return if the agent starts in state s, takes action a, and then henceforth follows the optimal policy π_*.

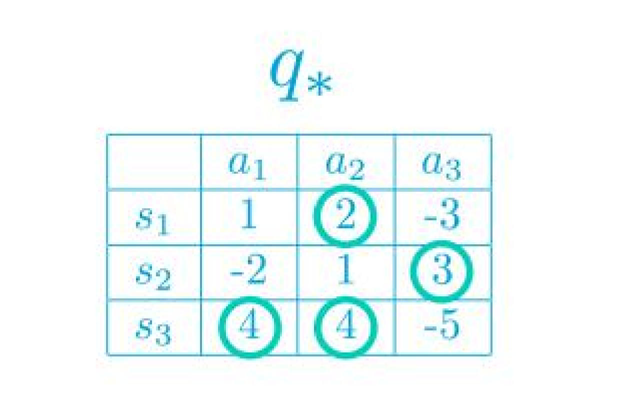


q_*

	a_1	a_2	a_3
s_1	1	2	-3
s_2	-2	1	3
s_3	4	4	-5

RL Framework: Optimal Policy

Towards constructing the optimal policy, we can begin by **selecting the entries that maximize the action-value function**, for each row (or state) - **greedy policy.**



q_*

	a_1	a_2	a_3
s_1	1	2	-3
s_2	-2	1	3
s_3	4	4	-5

Under the optimal policy, the agent must choose action a_2 when in state s_1, and it will choose action a_3 when in state s_2. As for state s_3, the agent can choose either action a_1 or a_2 under the optimal policy, but it can never choose action $a_{3.}$

$$\pi*(s1)=a_2$$
$$\pi*(s2)=a3$$
$$\pi*(a1|s3)=p,\ \pi*(a2|s3)=q,\ \text{and}\ \pi*(a3|s3)=0,$$

RL Framework: Policy Evaluation

Before we can use optimal policy definition, however, we must devise an algorithm for actually evaluating any arbitrary policy.

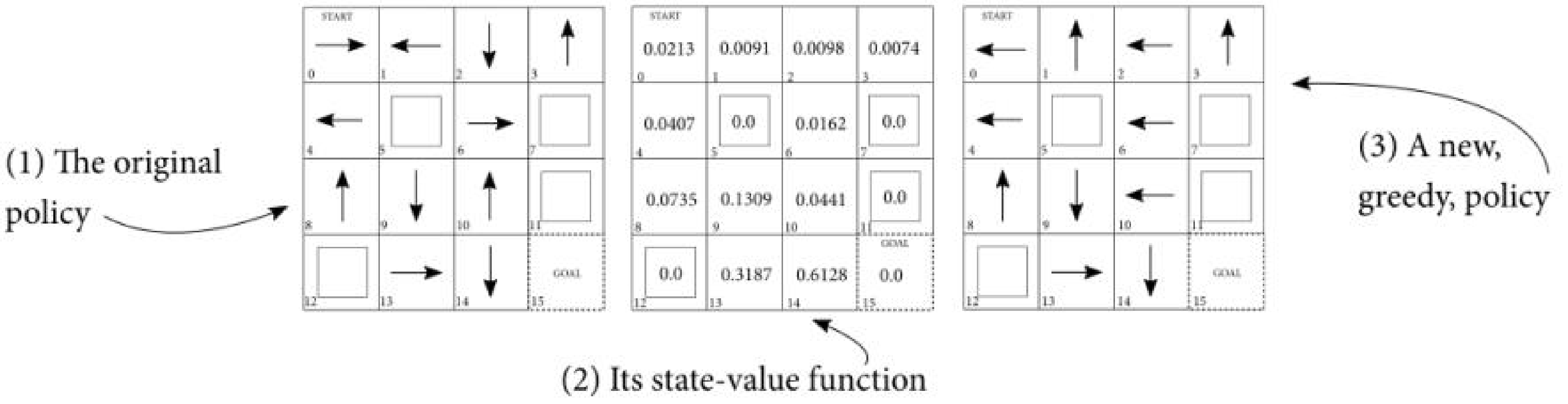
The **policy evaluation** algorithm consists of calculating the state-value function for a given policy by iteratively sweeping through the state space and improving an estimate from an estimate.

We refer to the type of algorithm that takes in a policy and outputs a value function as an algorithm that solves the **prediction problem**; calculating the values of a given policy.

Such algorithm is known as **policy evaluation**.

RL Framework: Policy Improvement

To improve a policy we use state-value function calculated from policy evaluation and then readjust the policy by creating a new policy that takes the action with maximum value.



(1) The original policy

START → 0	← 1	↓ 2	↑ 3
← 4	□ 5	→ 6	□ 7
↑ 8	↓ 9	↑ 10	□ 11
□ 12	→ 13	↓ 14	GOAL 15

(2) Its state-value function

START 0.0213 0	0.0091 1	0.0098 2	0.0074 3
0.0407 4	0.0 5	0.0162 6	0.0 7
0.0735 8	0.1309 9	0.0441 10	0.0 11
0.0 12	0.3187 13	0.6128 14	GOAL 0.0 15

(3) A new, greedy, policy

START ← 0	↑ 1	← 2	↑ 3
← 4	□ 5	← 6	□ 7
↑ 8	↓ 9	← 10	□ 11
□ 12	→ 13	↓ 14	GOAL 15

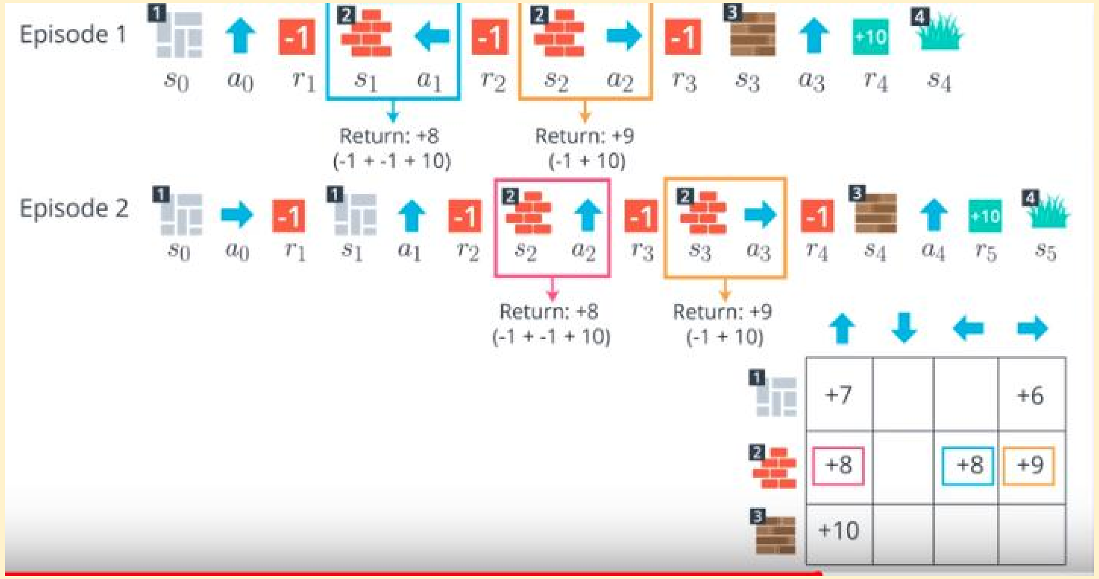
RL Framework: Policy Iteration

We can combine the policy evaluation and policy improvement algorithms to find better and better policies until we find the optimal state-value function and at least one optimal policy.

If we iterate on a random policy between evaluation and improvement, we will monotonically reach a unique fixed point that will indicate we have arrive to the optimal state-value function and therefore an optimal policy. **This algorithm is called policy iteration.**

Monte Carlo Methods

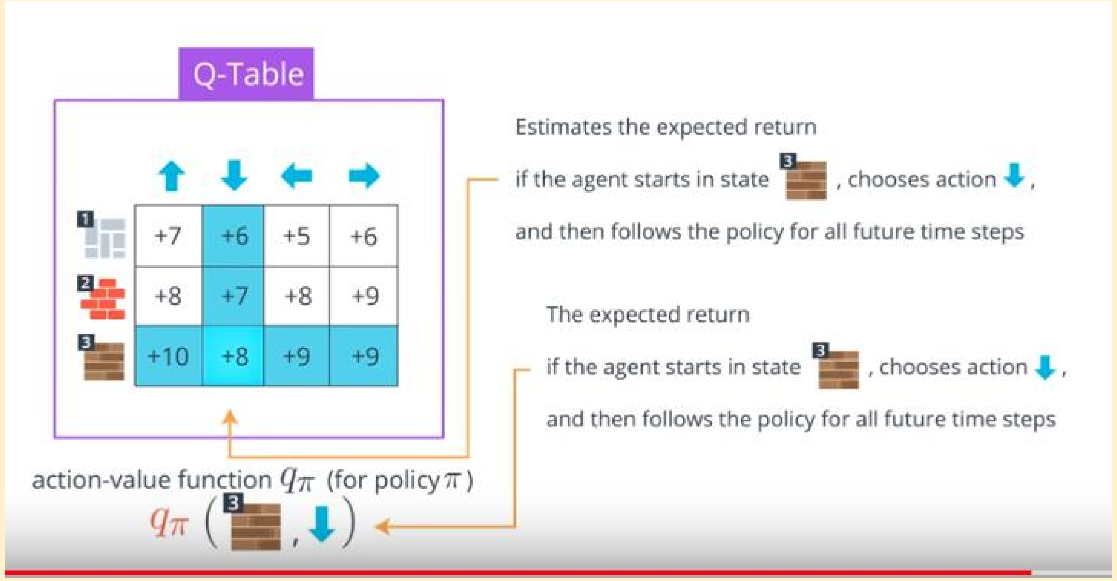
Monte Carlo Prediction - estimates **action-value functions** using **complete episodes.** In the algorithm for MC prediction, we begin by collecting many episodes with the policy. Then, we note that each entry in the Q-table corresponds to a particular state and action. To populate an entry, we use the return that followed when the agent was in that state, and chose the action. In the event that the agent has selected the same action many times from the same state, we need only average the returns.


Episode 1
1 s_0 a_0 -1 r_1 2 s_1 a_1 -1 r_2 2 s_2 a_2 -1 r_3 3 s_3 a_3 +10 r_4 4 s_4
Return: +8
(-1 + -1 + 10)
Return: +9
(-1 + 10)
Episode 2
1 s_0 a_0 -1 r_1 1 s_1 a_1 -1 r_2 2 s_2 a_2 -1 r_3 2 s_3 a_3 -1 r_4 3 s_4 a_4 +10 r_5 4 s_5
Return: +8
(-1 + -1 + 10)
Return: +9
(-1 + 10)

	↑	↓	←	→
1	+7			+6
2	+8		+8	+9
3	+10			

Monte Carlo Methods

Monte Carlo methods require us to complete an entire episode of interaction before updating the **Q-table**.



Q-Table

	↑	↓	←	→
1	+7	+6	+5	+6
2	+8	+7	+8	+9
3	+10	+8	+9	+9

Estimates the expected return
if the agent starts in state 3, chooses action ↓,
and then follows the policy for all future time steps

The expected return
if the agent starts in state 3, chooses action ↓,
and then follows the policy for all future time steps

action-value function q_π (for policy π)

q_π(3, ↓)

Monte Carlo Methods

Greedy Policies

A policy is greedy with respect to an **action-value function estimate Q** if for every state $s \in S$, it is guaranteed to select an action $a \in A(s)$ such that $a = \text{argmax}_{a \in A(s)} Q(s,a)$. It is common to refer to the selected action as the **greedy action**.

Epsilon-Greedy Policies

A policy is ϵ-greedy with respect to an action-value function estimate Q if for every state $s \in S$,

- with probability $1-\epsilon$, the agent selects the greedy action, and
- with probability ϵ, the agent selects an action uniformly at random from the set of available (non-greedy AND greedy) actions.

Monte Carlo Methods

Once the **Q-table** closely approximates the action-value function q_π, the agent can construct the policy π' that is ϵ-greedy with respect to the **Q-table**, which will yield a policy that is better than the original policy π.

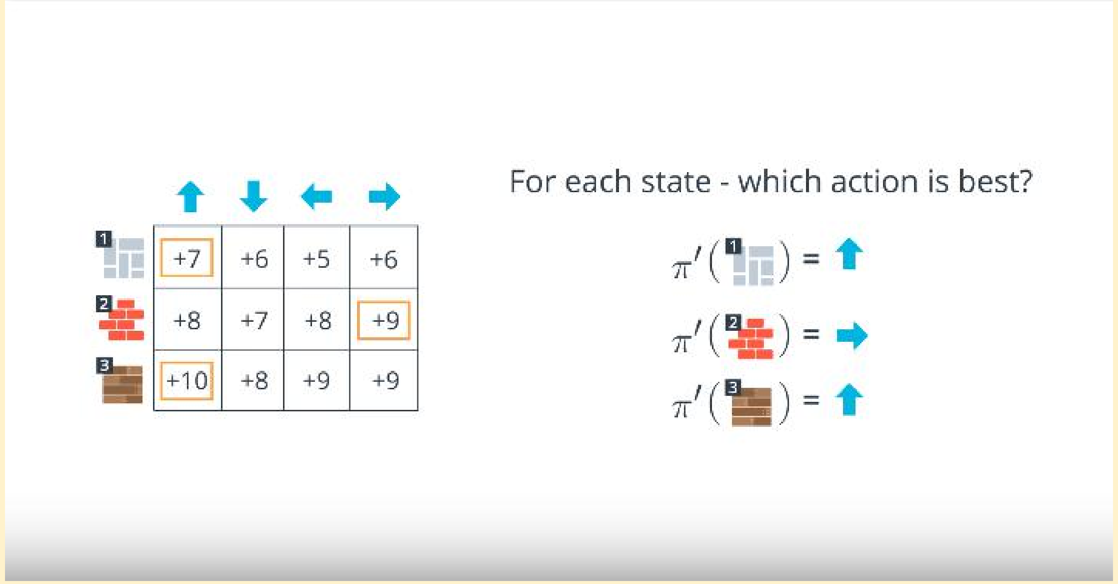
Furthermore, if the agent alternates between these two steps, **policy evaluation and policy improvement:**

> **Step 1:** using the policy π to construct the Q-table, and
> **Step 2:** improving the policy by changing it to be ϵ-greedy with respect to the Q-table

we will eventually obtain the **optimal policy π_*.**

Monte Carlo Methods

To get the greedy **action(s)**, for each row in the table, we need only select the action (or actions) corresponding to the column(s) **that maximize the row**.



	↑	↓	←	→
1	+7	+6	+5	+6
2	+8	+7	+8	+9
3	+10	+8	+9	+9

For each state - which action is best?

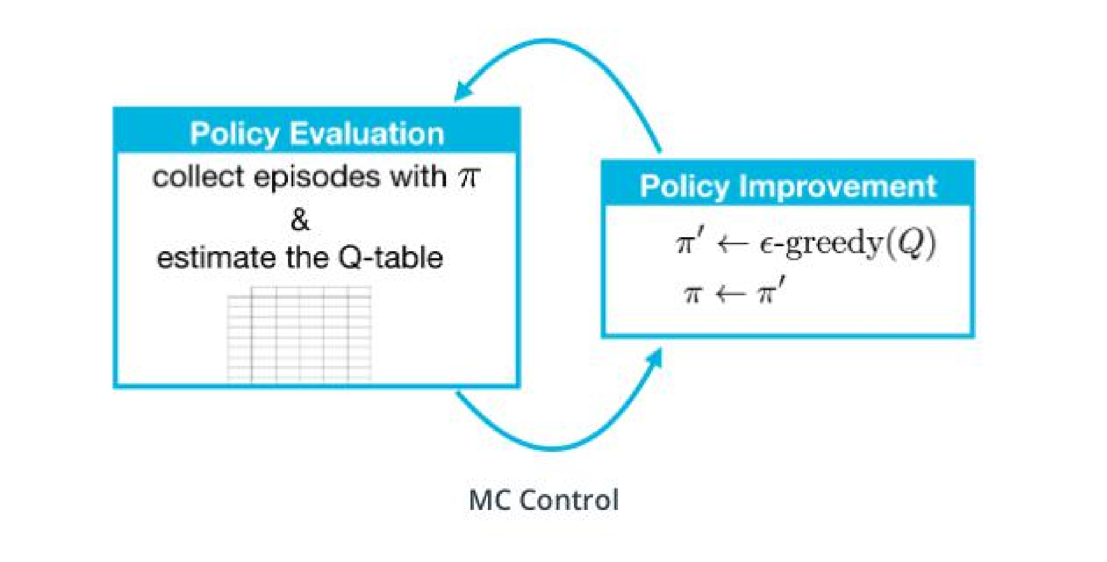
$\pi'(1)$ = ↑

$\pi'(2)$ = →

$\pi'(3)$ = ↑

Monte Carlo Methods

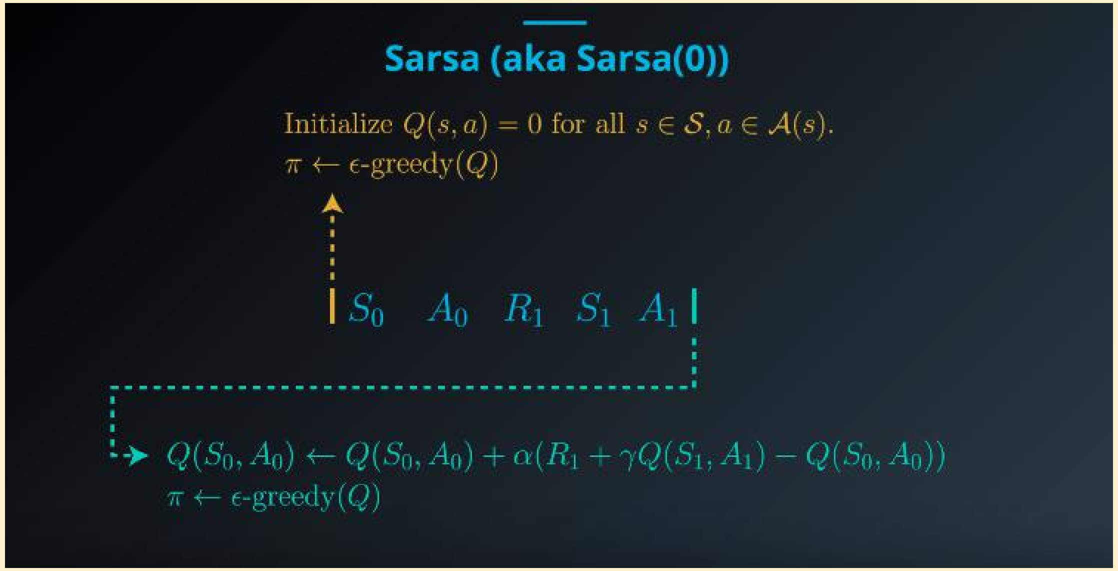
MC Control



Temporal Difference Methods - SARSA

Temporal Difference (TD) methods update the Q-table after every time step.
TD prediction bootstraps a guess from a guess

SARSA Algorithm - it uses the Q-value at the next step of the action as the result of the current policy, to update the Q-value of the current state-action pair.



Sarsa (aka Sarsa(0))

Initialize $Q(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$.

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

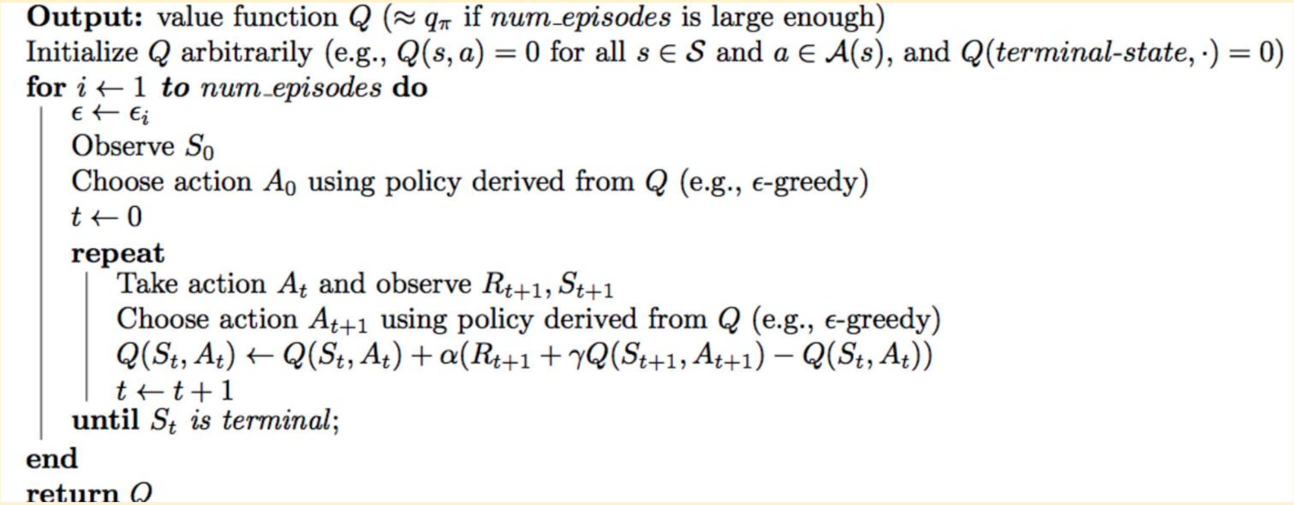
$S_0 \quad A_0 \quad R_1 \quad S_1 \quad A_1$

$Q(S_0,A_0) \leftarrow Q(S_0,A_0) + \alpha(R_1 + \gamma Q(S_1,A_1) - Q(S_0,A_0))$

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

Temporal Difference Methods - SARSA

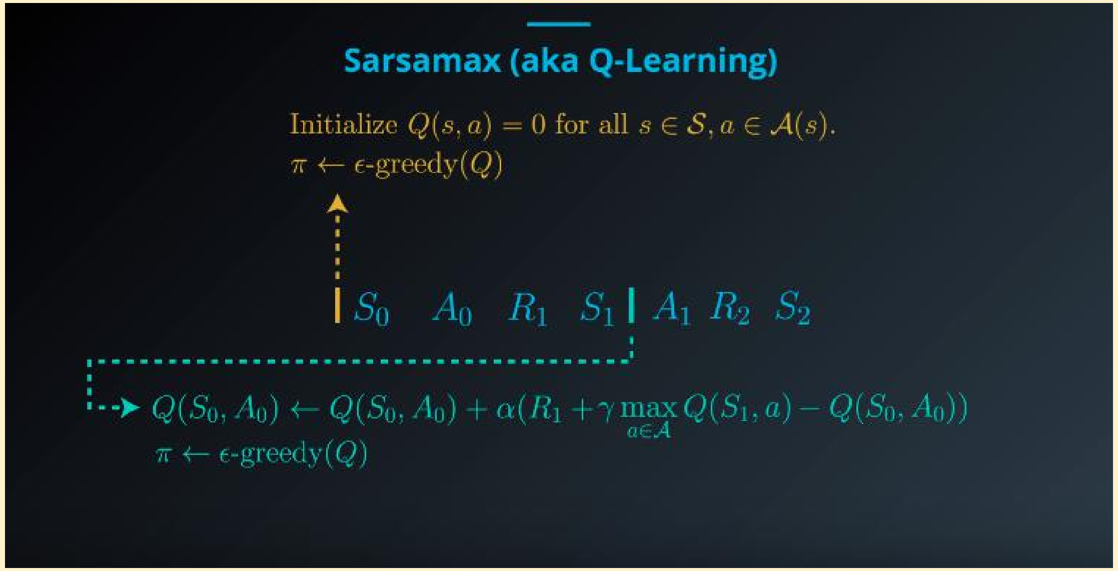
SARSA Algorithm Pseudocode



```
Output: value function $Q$ ($\approx q_\pi$ if $num\_episodes$ is large enough)
Initialize $Q$ arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(terminal\text{-}state, \cdot) = 0$)
for $i \leftarrow 1$ to $num\_episodes$ do
    $\epsilon \leftarrow \epsilon_i$
    Observe $S_0$
    Choose action $A_0$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    $t \leftarrow 0$
    repeat
        Take action $A_t$ and observe $R_{t+1}, S_{t+1}$
        Choose action $A_{t+1}$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$
        $t \leftarrow t + 1$
    until $S_t$ is terminal;
end
return $Q$
```

Temporal Difference Methods - Q-Learning

Q-Learning Algorithm - is a popular off-policy learning algorithm, and it is similar to SARSA, except for one thing. Instead of using the Q value estimate for the new state and the action that the agent took in that new state, it uses the **Q value estimate that corresponds to the action that leads to the maximum obtainable Q value from that new state, S'**.



Sarsamax (aka Q-Learning)

Initialize $Q(s, a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$.

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

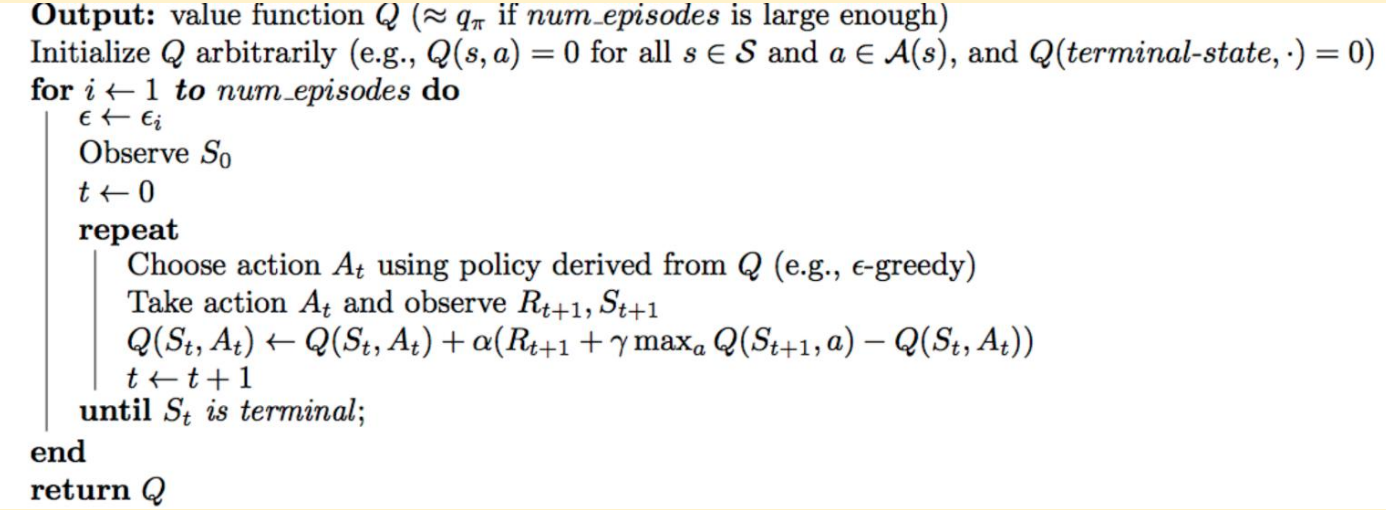
$S_0 \quad A_0 \quad R_1 \quad S_1 \mid A_1 \quad R_2 \quad S_2$

$Q(S_0, A_0) \leftarrow Q(S_0, A_0) + \alpha(R_1 + \gamma \max_{a \in \mathcal{A}} Q(S_1, a) - Q(S_0, A_0))$

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

Temporal Difference Methods - Q-Learning

Q-Learning Algorithm Pseudocode



```
**Output:** value function $Q$ ($\approx q_\pi$ if $num\_episodes$ is large enough)
Initialize $Q$ arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(terminal\text{-}state, \cdot) = 0$)
**for** $i \leftarrow 1$ **to** $num\_episodes$ **do**
    $\epsilon \leftarrow \epsilon_i$
    Observe $S_0$
    $t \leftarrow 0$
    **repeat**
        Choose action $A_t$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $A_t$ and observe $R_{t+1}, S_{t+1}$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$
        $t \leftarrow t + 1$
    **until** $S_t$ *is terminal*;
**end**
**return** $Q$
```

On-policy learning is learning to optimize a policy as we used it and it couples behavior and optimization into the same policy, while off-policy learning decouples learning from behavior, allowing your agent to learn from different sources.

The off-policy TD control algorithm is called Q-learning, and it is probably the most popular version of "tabular" reinforcement learning.

It is also the base of perhaps the most popular deep reinforcement learning algorithm, **Deep Q-networks (DQN).**

Monte Carlo vs Temporal Difference Methods

- Return in Monte Carlo method depends on many random variables within a single episode: you take multiple actions, transition multiple times, and receive multiple rewards. That adds **high variance** because the randomness accumulates within the same episode and estimates could wildly differ from one episode to another.

- The Temporal Difference updates have much **lower variance** than the return we use in Monte-Carlo updates. The Temporal Difference algorithm depends only on a single action, a single transition, a single reward, and a single next-state estimate. Temporal Difference is said to be a biased estimate of the true state-value function.

Part II

- **From Reinforcement Learning to Deep Reinforcement Learning**

- **Value Based Methods**
  - **Deep Q-Learning Agent**

- **Policy Gradient Methods**
  - **DDPG - Deep Deterministic Policy Gradient Agent**
  - **MADDPG - Multi-Agent Deep Deterministic Policy Gradient**

- **Examples of DRL agents using ML Agents environment**

Part II

- **Stabilizing Deep Reinforcement Learning**
  - **Experience Replay**
  - **Fixed Q-Targets**

- **Algorithm Description**
- **Model (Network) Architecture**
- **Annotated Source code implementation in PyTorch**

From Reinforcement Learning to Deep Reinforcement Learning

High-Dimensional State Space

- The main drawback of 'tabular' reinforcement learning is that the use of a table to represent value functions is no longer practical in complex problems. Environments can have high-dimensional state spaces, meaning that the number of variables that comprise a single state is very large.

Continuous State Space

- Environments can also have continuous variables, meaning that the number of values a single variable can be is infinite. For instance, the position and angles of a robot can be of infinitesimal precision; it could be 1.56 or 1.5683 or 1.5683256 and so on.

And of course, there are environments that have both **high-dimensional and continuous variables**

From Reinforcement Learning to Deep Reinforcement Learning

Function Approximators - Using neural networks to approximate Q-functions

- Neural networks are shown to be effective as universal function approximators. In fact, there is a universal approximation theorem that states that a single hidden layer feedforward neural network can approximate any continuous function that is closed and bounded in. It basically means that even simple (shallow) neural networks can approximate several functions.

- In environments with high-dimensional or continuous state spaces there is really no practical reason to create a table to store value functions. Sure, discretizing or binning the values could make tables possible. But, again, even if we could engineer a way to use tables and store our value functions there, by doing so, we'd be missing out on the advantages of generalization.

From Reinforcement Learning to Deep Reinforcement Learning

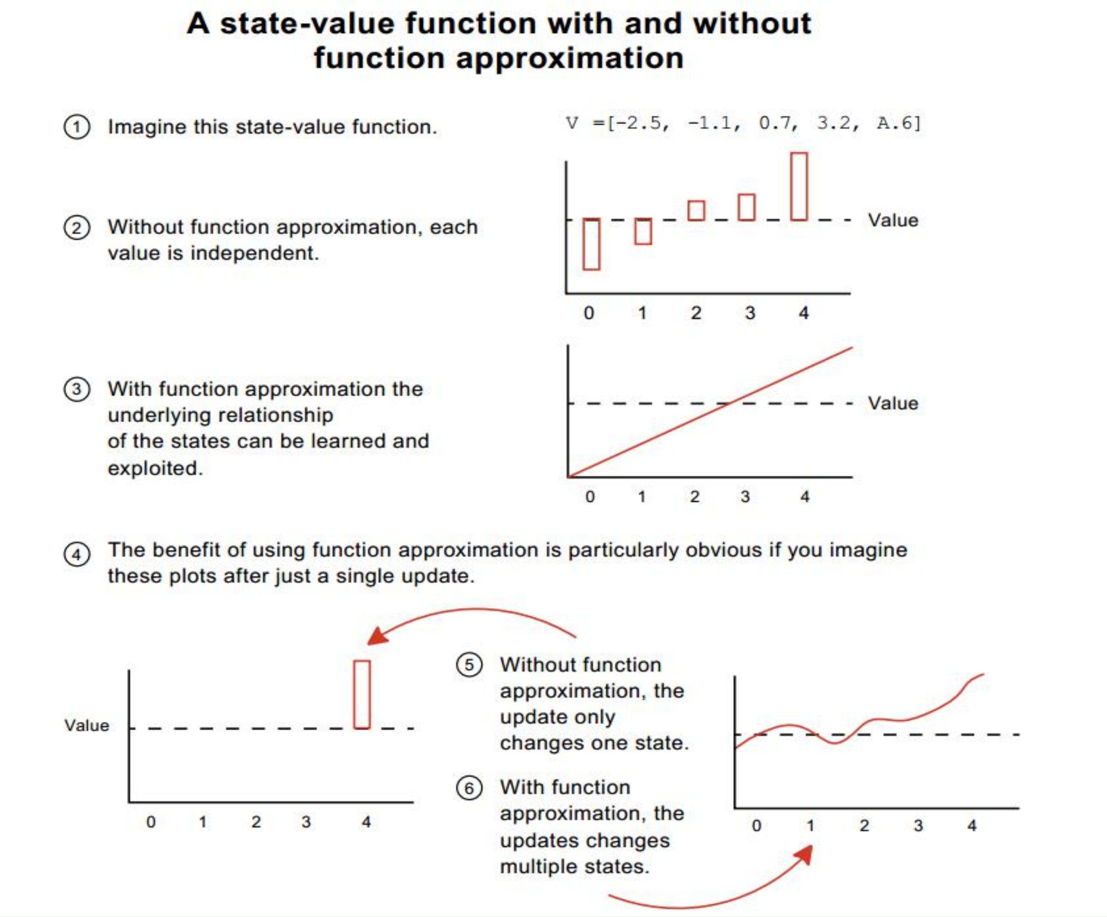
Generalization

- We would like our agents to generalize for example that 0.1 units away from the center is similar to 0.2, at least more so than 2.4.

- Action-value functions often have some underlying relationship that agents can exploit.

- We want to use generalization because it is a more efficient use of experiences. With function approximation, such as neural networks, agents learn and exploit patterns with less data (and perhaps faster).

From Reinforcement Learning to Deep Reinforcement Learning



A state-value function with and without function approximation

1. Imagine this state-value function.

V =[-2.5, -1.1, 0.7, 3.2, A.6]

2. Without function approximation, each value is independent.

3. With function approximation the underlying relationship of the states can be learned and exploited.

4. The benefit of using function approximation is particularly obvious if you imagine these plots after just a single update.

5. Without function approximation, the update only changes one state.

6. With function approximation, the updates changes multiple states.

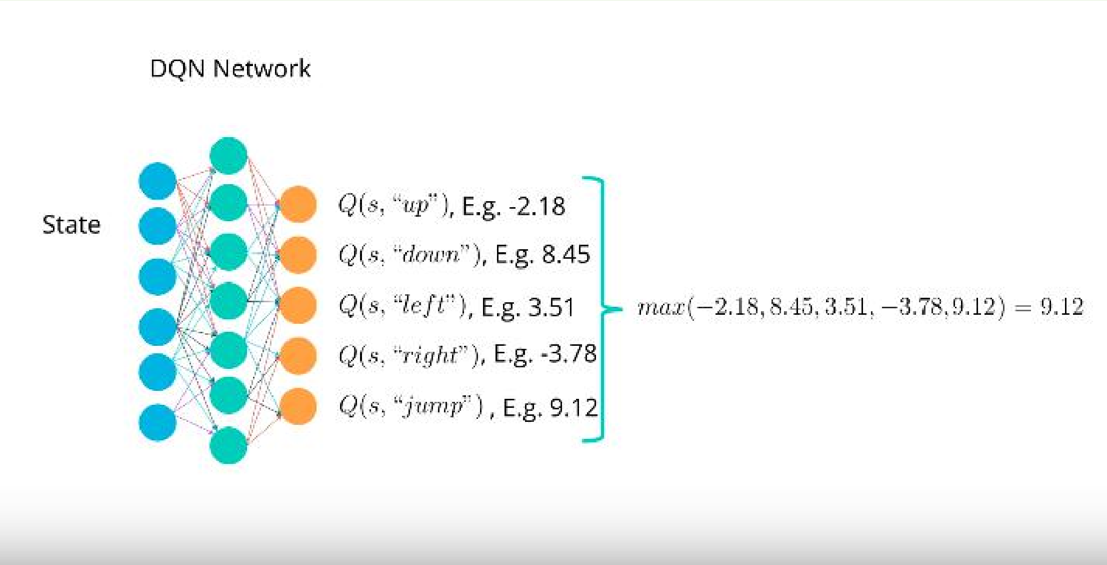
Value Based Methods

Value Functions - the state-value function **V(s)**, though useful for many purposes, is not sufficient on its own to solve the control problem. Finding V(s) helps you know how much expected cumulative discounted future rewards you will obtain from state "s" and using policy π thereafter.

The **action-value function Q(s,a)** solves the control problem. If we had the values of state and action pairs, we could differentiate the actions that would lead us to, either gain information, in the case of an exploratory action, or maximize total expected reward, in the case of a greedy action.

Value Based Methods

Deep Q-Network - we will be approximating **the action-value function Q(s,a)**, just like in Q-learning. We refer to the approximate action-value function **as Q(s,a; w); that mean Q is parameterized by "w", the weights of a neural network, and indexed by a state "s" and an action "a".**


DQN Network
State
$Q(s, \text{“}up\text{”})$, E.g. -2.18
$Q(s, \text{“}down\text{”})$, E.g. 8.45
$Q(s, \text{“}left\text{”})$, E.g. 3.51
$Q(s, \text{“}right\text{”})$, E.g. -3.78
$Q(s, \text{“}jump\text{”})$, E.g. 9.12
$max(-2.18, 8.45, 3.51, -3.78, 9.12) = 9.12$

Value Based Methods

What to optimize

We can use the same principles as for **policy iteration in which we alternate between policy evaluation and policy improvement**.

We will start with a randomly initialized action-value function (and policy), evaluate it by sampling actions from it, improve it with an exploration strategy such as epsilon-greedy, then evaluate again, improve it, and so on.

There are multiple ways we can evaluate any given policy. More specifically, there are different targets we can use for estimating the values of a policy π.

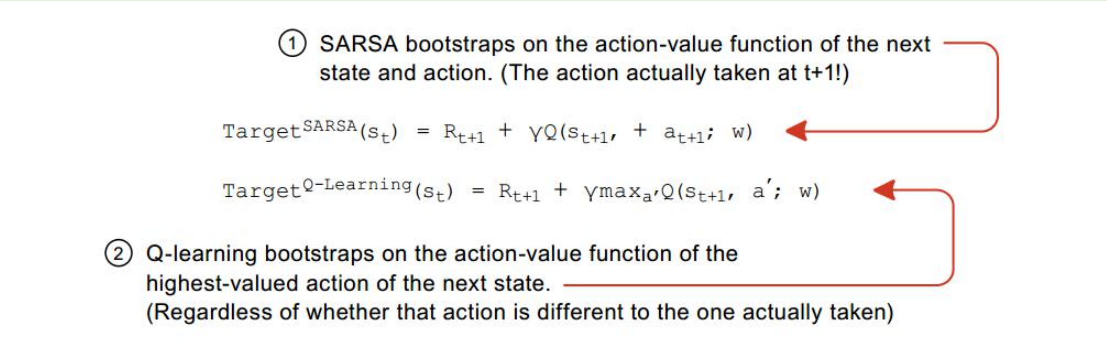
The core targets in reinforcement learning are the Monte-Carlo (MC) target and the Temporal-Difference (TD) target.

Value Based Methods

TD - Temporal Difference targets

There are two main ways to bootstrap the TD target we can either **use the action-value function of the action actually taken at landing state, or we can use the value of the best action.**

On-policy and off-policy TD targets



① SARSA bootstraps on the action-value function of the next state and action. (The action actually taken at t+1!)

$$\text{Target}^{\text{SARSA}}(s_t) = R_{t+1} + \gamma Q(s_{t+1}, + a_{t+1}; w)$$

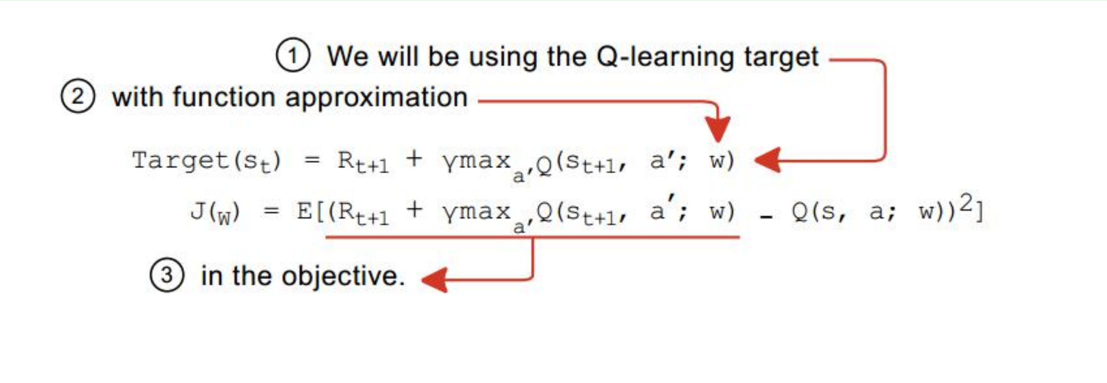
$$\text{Target}^{\text{Q-Learning}}(s_t) = R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; w)$$

② Q-learning bootstraps on the action-value function of the highest-valued action of the next state.
(Regardless of whether that action is different to the one actually taken)

Value Based Methods

Q-Learning Target

We will use the off-policy TD target used by the Q-learning algorithm. To get an objective function, we simply need to substitute the true optimal action-value function q*(s,a) in the ideal objective equation by this target.



① We will be using the Q-learning target

② with function approximation

$$\text{Target}(s_t) = R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; w)$$

$$J(w) = E[(\underbrace{R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; w)}_{} - Q(s, a; w))^2]$$

③ in the objective.

Value Based Methods

Loss Function

A loss function is a measure of how well our neural network predictions are. In supervised learning, it is more straightforward to interpret the loss function: given a batch of predictions and their corresponding true values, the loss function computes a distance score indicating how well the network has done in this batch.

The challenge in reinforcement learning, as compared to supervised learning is that our "true values" are also predictions coming from the network. **MSE** (or L2 loss) is defined as the **average squared difference between the predicted and true values**; in our case, the “predicted values” are the predicted values of the action-value function. **But the “true values” are, the TD targets.**

Value Based Methods

Deep Q-Learning Algorithm:

1. Initialize parameters for $Q(s, a)$ and $\hat{Q}(s, a)$ with random weights, epsilon ← 1.0, and empty replay buffer ($\hat{Q}(s, a)$ is target network)

2. With probability epsilon, select a random action a, otherwise $a = \arg\max_a Q_{s,a}$

3. Execute action in environment and observe reward r and the next state s′

4. Store transition (s, a, r, s′) in the replay buffer (buffer size=10000)

5. Sample a random minibatch (batch size = 64) of transitions from the replay buffer

6. For every transition in the buffer, calculate target $y = r$ if the episode has ended at this step or $y = r + \gamma \max_{a' \in Aa'} \hat{Q}_{s',a'}$ otherwise A discount factor - gamma of 0.99 is applied.

7. Calculate MSE loss: $L = (Q_{s,a} - y)^2$

8. Update Q(s, a) using the SGD algorithm by minimizing the loss in respect to model parameters. The learning rate for Adam optimizer (LR) = 0.0001.

9. Every 4 steps copy weights from local Q-network to target Q-network. The target network is frozen for several time steps and then the target network weights are updated by copying the weights from the actual Q network. Freezing the target network for a while and then updating its weights with the actual Q network weights stabilizes the training. The fractional update was controlled by the parameter tau which was set to 0.001.

Value Based Methods

Stabilizing Deep Reinforcement Learning

Unfortunately, reinforcement learning is notoriously unstable when neural networks are used to represent the action values.

Common problems in value-based deep reinforcement learning

- The first problem is the **non-stationarity of the target values**. These are the targets we use to train our network with, but these targets are calculated using the network itself.
- The second problem is the **non-compliance with the IID assumption of the data**. Samples collected from trajectories are correlated and also not identically distributed as they depend on the policy that generates the actions.

The **Deep Q-Learning algorithm** addresses these instabilities by using two key features:

- **Experience Replay**
- **Fixed Q-Targets**

Value Based Methods

Fixed Q-Targets

In supervised learning, **the targets are the labels on your dataset and are fixed throughout training.** In reinforcement learning these targets would move freely with every training step of the network. At every update, we improve the value function and therefore change the shape of possibly the entire function. That means the target values change as well. Which means, our estimates are invalid with every update, since they have already changed.

A straightforward way to make target values more stationary is to have a separate network which we can fix for multiple steps and use it to calculate more stable target values. This network is **called target network**, as it is used to calculate targets.

By having a target network and fixing our target values, we mitigate the **"chasing your own tail"** issue by artificially creating multiple small supervised learning problems: **Our targets are now fixed for as many steps as we fix our target network.**

Value Based Methods

Experience Replay

- By sampling at random we increase the probability that our updates to the neural network will have less variance.

- When we use the batch, most of the samples in that batch were correlated and similar. Updating with similar samples concentrates the updates we make to our neural network to a limited area of our function, and it potentially over-emphasizes the magnitude of the updates.

- If we sample uniformly at random from a very large buffer, on the other hand, chances are our updates to the network will be better distributed all across, and therefore more representative of the true function.

- Using a replay buffer also gives the impression our data is **IID**, so optimization methods will be better behaved. Samples will seem independent and identically distributed because we will be sampling from multiple trajectories and policies at once.

Value Based Methods

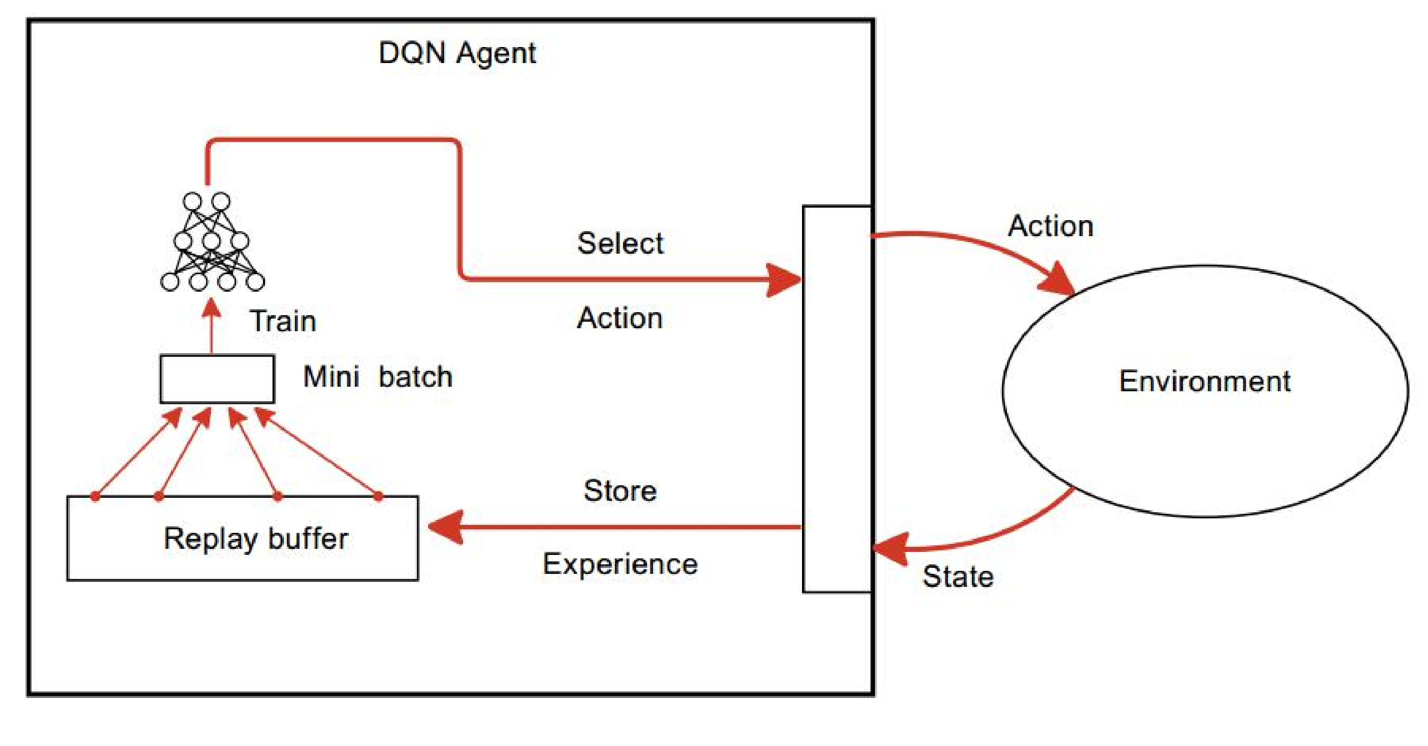
Prioritized Experience Replay

- The replay buffer is used to break the correlation between immediate transitions in our episodes. The examples agent experiences during the episode will be highly correlated, as most of the time the environment is "smooth" and doesn't change much according to our actions. However, the SGD method assumes that the data we use for training has a i.i.d. property. To solve this problem, the classic DQN method used a large buffer of transitions, randomly sampled to get the next training batch.

- The main concept of prioritized replay is the criterion by which the importance of each transition is measured. One idealised criterion would be the amount the RL agent can learn from a transition in its current state. A reasonable proxy for this measure is the magnitude of a transition’s TD error δ, which indicates how ‘surprising’ or unexpected the transition is.

Value Based Methods

Experience/Replay Buffer



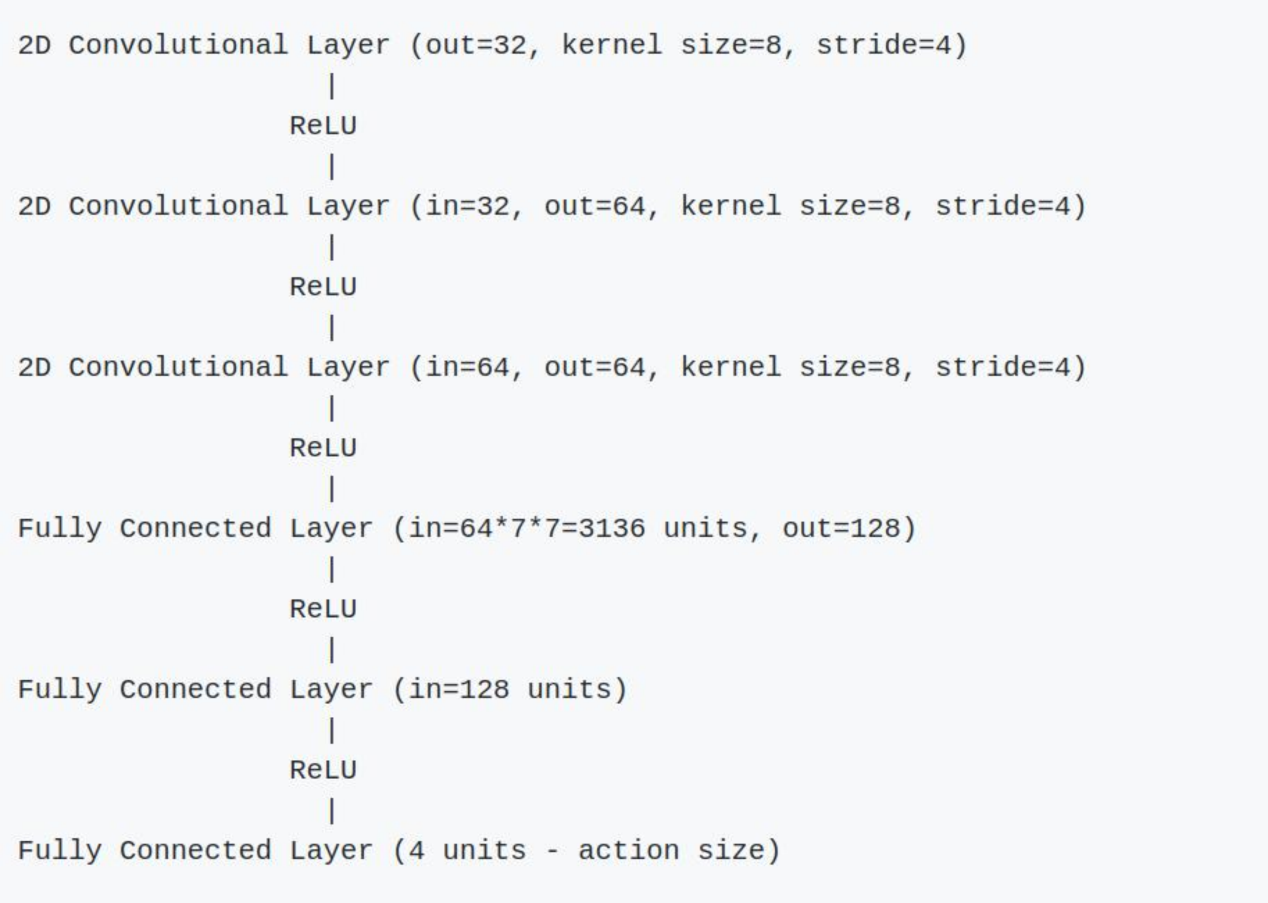
Deep Q-Learning Agent

The Deep Convolutional Q-Network

```
class QNetwork(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(QNetwork, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        conv_out_size = self._get_conv_out(input_shape)
        self.fc = nn.Sequential(
            nn.Linear(conv_out_size, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        )

    def _get_conv_out(self, shape):
        o = self.conv(torch.zeros(1, *shape))
        return int(np.prod(o.size()))

    def forward(self, x):
        conv_out = self.conv(x).view(x.size()[0], -1)
        return self.fc(conv_out)
```

Deep Q-Learning Agent

The Deep Convolutional Q-Network



```
2D Convolutional Layer (out=32, kernel size=8, stride=4)
                  |
                ReLU
                  |
2D Convolutional Layer (in=32, out=64, kernel size=8, stride=4)
                  |
                ReLU
                  |
2D Convolutional Layer (in=64, out=64, kernel size=8, stride=4)
                  |
                ReLU
                  |
Fully Connected Layer (in=64*7*7=3136 units, out=128)
                  |
                ReLU
                  |
Fully Connected Layer (in=128 units)
                  |
                ReLU
                  |
Fully Connected Layer (4 units - action size)
```

Deep Q-Learning Agent

The Deep Q-Learning Agent

```
class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, input_shape, action_size, seed):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            seed (int): random seed
        """
        self.state_size = input_shape
        self.action_size = action_size
        self.seed = random.seed(seed)

        # Q-Network
        self.qnetwork_local = QNetwork(input_shape, action_size).to(device)
        self.qnetwork_target = QNetwork(input_shape, action_size).to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        self.prioritized_replay_alpha = 0.6
        self.prioritized_replay_beta0 = 0.4
        self.prioritized_replay_beta_iters = 100000

        # Replay memory
        self.memory = PrioritizedReplayBuffer(BUFFER_SIZE, alpha=self.prioritized_replay_alpha)
        self.beta_schedule = LinearSchedule(self.prioritized_replay_beta_iters,
                                             initial_p=self.prioritized_replay_beta0,
                                             final_p=1.0)

        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0
```

Deep Q-Learning Agent

The Deep Q-Learning Agent

```
def step(self, state, action, reward, next_state, done):
    # Save experience in replay memory
    self.memory.add(state, action, reward, next_state, done)

    # Learn every UPDATE_EVERY time steps.
    self.t_step = (self.t_step + 1) % UPDATE_EVERY
    if self.t_step == 0:
        # If enough samples are available in memory, get random subset and learn
        if len(self.memory) > BATCH_SIZE:
            experiences = self.memory.sample(BATCH_SIZE, beta=self.beta_schedule.value(len(self.memory)))
            self.learn(experiences, GAMMA)

  def act(self, state, eps=0.):
    """Returns actions for given state as per current policy.

    Params
    ======
        state (array_like): current state
        eps (float): epsilon, for epsilon-greedy action selection
    """
    self.qnetwork_local.eval()
    with torch.no_grad():
         action_values = self.qnetwork_local(state)
    self.qnetwork_local.train()

    # Epsilon-greedy action selection
    if random.random() > eps:
        return np.argmax(action_values.cpu().data.numpy())
    else:
        return random.choice(np.arange(self.action_size))
```

Deep Q-Learning Agent

The Deep Q-Learning Agent

```
    def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples.
        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
            gamma (float): discount factor
        """
        states, actions, rewards, next_states, dones, weights, idxes = experiences

        # Get max predicted Q values (for next states) from target model
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Compute loss
        losses_v = weights * (Q_expected - Q_targets) ** 2
        loss = losses_v.mean()
        prios = losses_v + 1e-5

        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # Update replay buffer priorities
        self.memory.update_priorities(idxes, prios.data.cpu().numpy())

        # Update target network
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)
```

Deep Q-Learning Agent

The Deep Q-Learning Agent

```
def soft_update(self, local_model, target_model, tau):
    """Soft update model parameters.
    θ_target = τ*θ_local + (1 - τ)*θ_target
    Params
    ======
        local_model (PyTorch model): weights will be copied from
        target_model (PyTorch model): weights will be copied to
        tau (float): interpolation parameter
    """

    for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
        target_param.data.copy_(tau * local_param.data + (1.0 - tau) * target_param.data)
```

Policy Gradient Methods

Policy Gradients - Introduction

In Deep Q-Learning based intelligent agent implementation, we use a **deep neural network as the function approximator to represent the action-value function**. The agent then uses the action-value function to come up with a policy based on the value function. In particular, we use the **epsilon-greedy algorithm**.

Ultimately the agent has to know what actions are good to take given an observation/state. **Instead of parametrizing or approximating a state/action action function and then deriving a policy based on that function, can we not parametrize the policy directly?** Yes we can! That is the exact idea behind policy gradient methods.

Policy Gradient Methods

Why Policy?

There are several reasons why policy might be an interesting topic to explore.

- Policy is what we’re looking for when we’re solving a Reinforcement Learning (RL) problem. When the agent obtains the observation and needs to make a decision about what to do next, we need policy, not the value of the state or particular action. We do care about the total reward, but at every state, we may have little interest in the exact value of the state.

- Environments with lots of actions or, in the extreme, with a continuous action space. To be able to decide on the best action to take having Q(s, a), we need to solve a small optimization problem finding a, which maximizes Q(s, a).

- Stochasticity - policy is naturally represented as the probability of actions.

Policy Gradient Methods

Convergence

Policy-based methods have better convergence properties.

The problem with value-based methods is that they can have a big oscillation while training. This is because the choice of action may change dramatically for an arbitrarily small change in the estimated action values.

On the other hand, with policy gradient, we just follow the gradient to find the best parameters. We see a smooth update of our policy at each step.

Because we follow the gradient to find the best parameters, we’re guaranteed to converge on a local maximum (worst case) or global maximum (best case).

Policy Gradient Methods

Policy gradients are more effective in high dimensional action spaces

Another advantage is that policy gradients are more effective in high dimensional action spaces, or when using continuous actions.

The problem with Deep Q-learning is that their predictions assign a score (maximum expected future reward) for each possible action, at each time step, given the current state.

But what if we have an infinite possibility of actions?
For instance, with a self driving car, at each state you can have a (near) infinite choice of actions (turning the wheel at 15°, 17.2°, 19,4°, honk…). We'll need to output a Q-value for each possible action!

On the other hand, in policy-based methods, you just adjust the parameters directly: thanks to that you'll start to understand what the maximum will be, rather than computing (estimating) the maximum directly at every step.

Policy Gradient Methods

Policy gradients can learn stochastic policies

A third advantage is that policy gradient can learn a stochastic policy, while value functions can’t.

We don’t need to implement an exploration/exploitation trade off. A stochastic policy allows our agent to explore the state space without always taking the same action. This is because it outputs a probability distribution over actions. As a consequence, it handles the exploration/exploitation trade off without hard coding it.

Policy Gradient Methods

In policy gradient based methods, **the policy is represented by using a neural network**, and the goal is to find the best set of parameters . This can be intuitively seen as an optimization problem where we are trying to optimize the objective of the policy to find the best-performing policy.

Objective of the agent's policy
We know that the agent should achieve maximum rewards in the long term, in order to complete the task or achieve the goal. If we can formulate that objective mathematically, we can use optimization techniques to find the best policy for the agent to follow for the given task.

The Policy Gradient Theorem

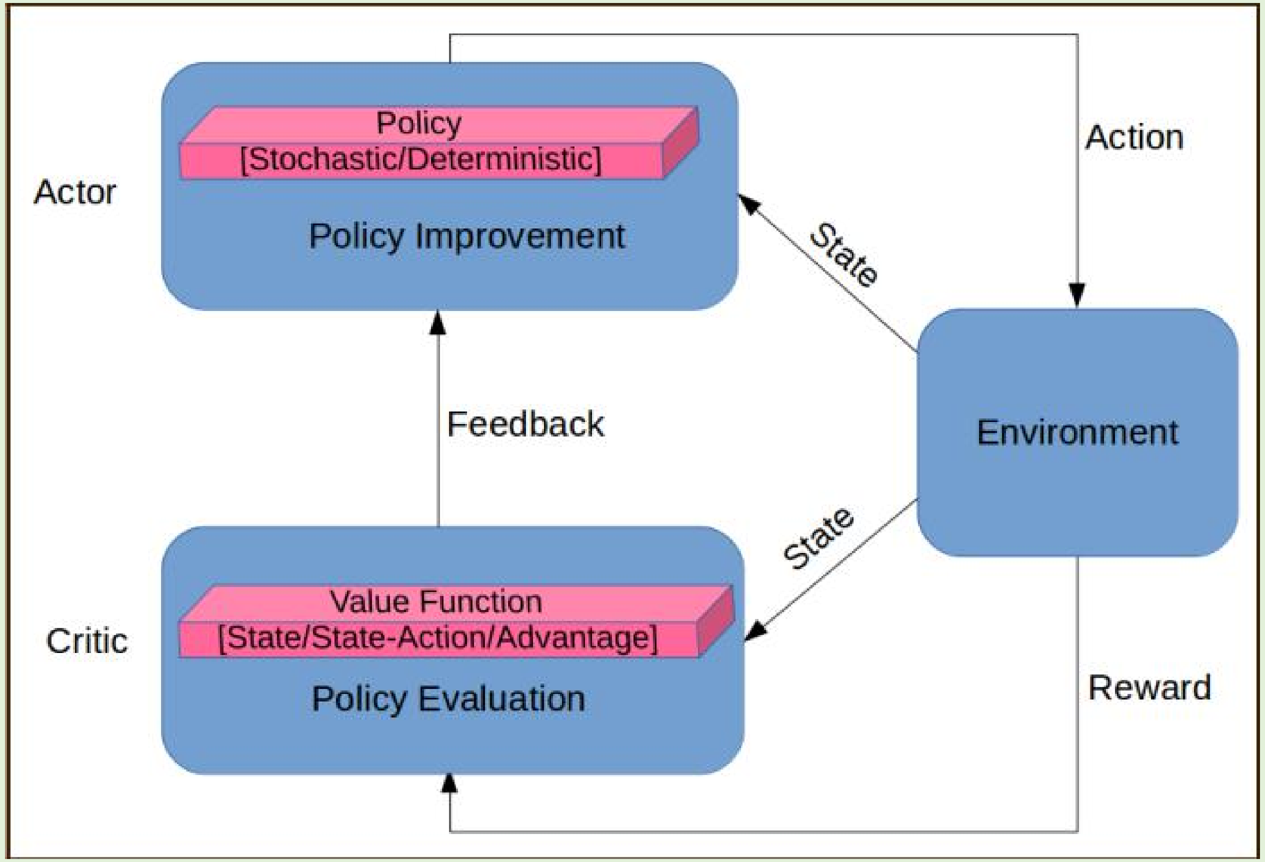
$$\nabla J \approx E[Q(s, a)\nabla \log \pi(a|s)]$$

PG defines the direction in which we need to change our network’s parameters to improve the policy in terms of the accumulated total reward. The scale of the gradient is proportional to the value of the action taken, which is Q(s, a) and the gradient itself is equal to the gradient of log-probability of the action taken. Intuitively, **this means that we’re trying to increase the probability of actions that have given us good total reward and decrease the probability of actions with bad final outcomes.**

Actor-Critic Methods

- There are two components in the **actor-critic algorithm**.
- **The actor** is responsible for acting in the environment, which involves taking actions, given observations about the environment and based on the agent's policy. The actor can be thought of as the policy holder/maker.
- **The critic** takes care of estimating **the state-value, or state-action-value, or advantage-value function** (depending on the variant of the actor-critic algorithm used).

Actor-Critic Methods



Actor-Critic Methods

A2C - Advantage Actor-Critic

- The action-value actor-critic (where critic estimates action-value function) algorithm still has high variance. We can reduce the variance by subtracting a baseline function, B(s), from the policy gradient. **A good baseline is the state value function.**

- The PG variance can be further reduced by making the baselines state-dependent, which, intuitively, is a good idea, as different states could have very different **baselines.** Indeed, to decide about the suitability of a particular action in some state, we're using the discounted total reward of the action.

- **The total reward itself could be represented as a value of the state plus advantage of the action: Q(s, a) = V(s) + A(s, a).**

DDPG - Deterministic Deep Policy Gradient

- **DDPG** is a variation of the actor-critic method, but has a very nice property of being off-policy. In actor-critic, the actor estimates the stochastic policy, which returns the probability distribution over discrete actions or for continuous action spaces the parameters of normal distribution.

- **Deterministic policy gradients belongs to the actor-critic family, but the policy is deterministic, which means that it directly provides us with the action to take from the state.** This makes it possible to apply the chain rule to the Q-value, and by maximizing the Q, the policy will be improved as well.

- **The role of actor is to return the action to take for every given state**. In a continuous action domain, every action is a number, so the actor network will take the state as an input and return N values, one for every action. This mapping will be deterministic, as the same network always returns the same output if the input is the same.

- **The role of the critic is to estimate the Q-value**, which is a discounted reward of the action taken in some state. **The critic net accepts two inputs: the state and the action. The output from the critic will be the single number, which corresponds to the Q-value.**

DDPG - Deterministic Deep Policy Gradient

- **We have two functions represented by deep neural networks, one is the actor, let's call it μ(s), which converts the state into the action and the other is the critic, by the state and the action giving us the Q-value: Q(s, a)**. We can substitute the actor function into the critic and get the expression with only one input parameter of our state: Q(s, μ(s)).

- **The output of the critic gives us the approximation of the entity we're interested in maximizing in the first place: the discounted total reward.** This value depends not only on the input state, but also on parameters of the θ_μ actor and the θ_Q critic networks. At every step of our optimization, we want to change the actor's weights to improve the total reward that we want to get. In mathematical terms, we want the gradient of our policy.

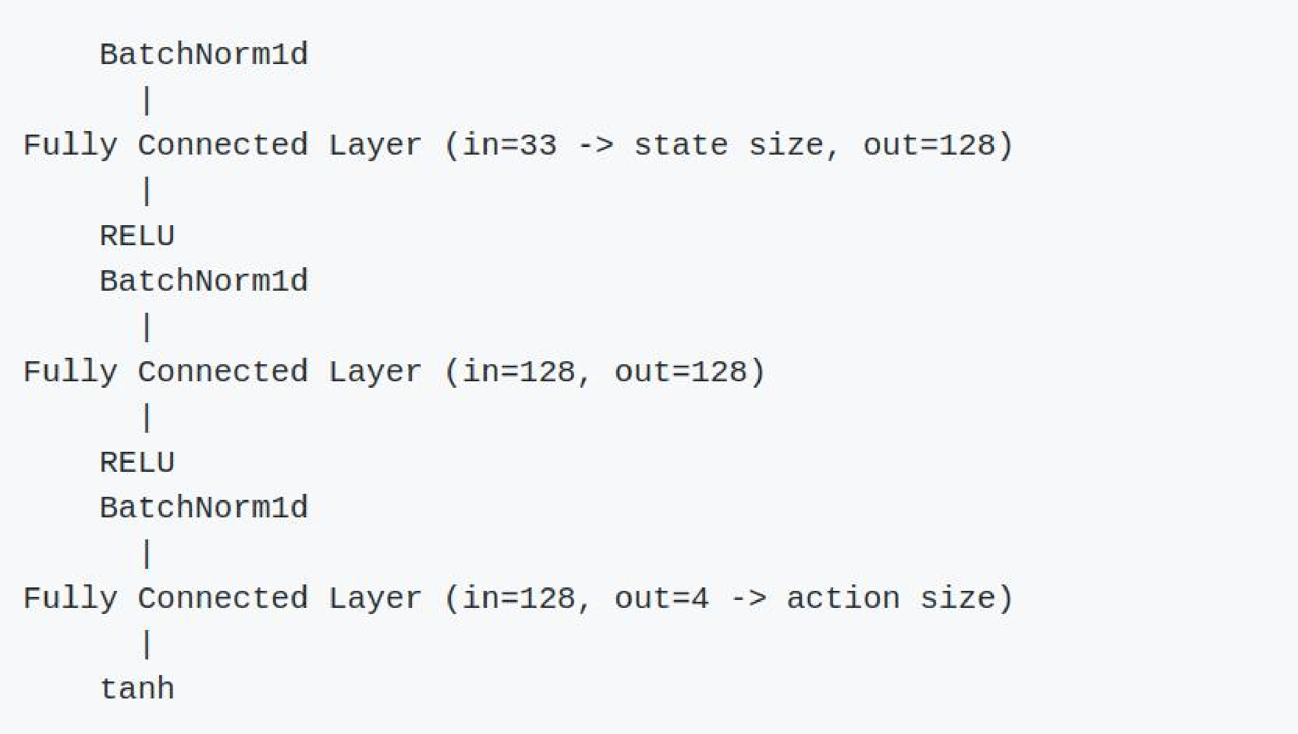
- In his deterministic policy gradient theorem, David Silver has proved that **stochastic policy gradient is equivalent to the deterministic policy gradient**. In other words, to improve the policy, we just need to calculate the gradient of the Q(s, μ(s)) function.

- By applying the chain rule, we get the gradient: **$\nabla_a Q(s,a) \nabla_{\theta\mu} \mu(s)$.**

DDPG - Deterministic Deep Policy Gradient

- **The DDPG model consists of two separate networks for the actor and critic**

- **The actor is fully connected feed-forward network** with two hidden layers, RELU activation and Batch Normalization. The input is an observation/state vector, while the output is a vector with N values, one for each action. The output actions are transformed with hyperbolic tangent non-linearity to squeeze the values to the -1..1 range.



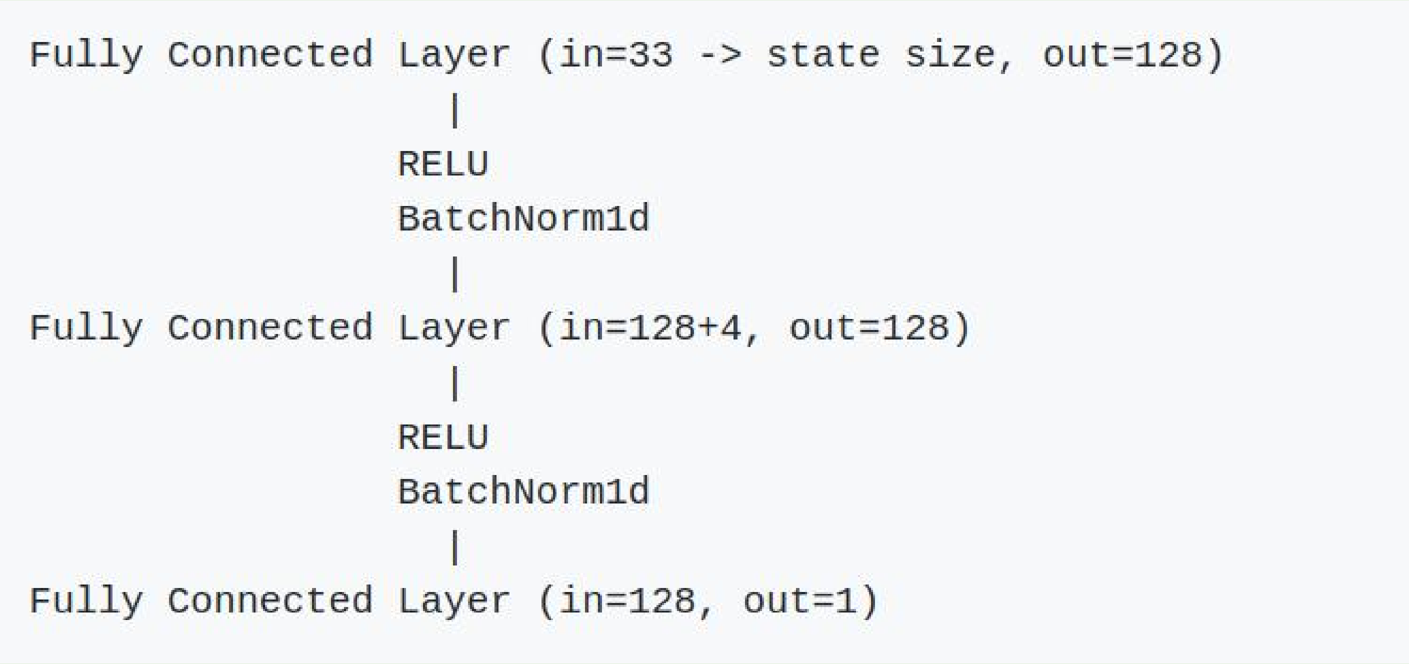
```
    BatchNorm1d
      |
Fully Connected Layer (in=33 -> state size, out=128)
      |
    RELU
    BatchNorm1d
      |
Fully Connected Layer (in=128, out=128)
      |
    RELU
    BatchNorm1d
      |
Fully Connected Layer (in=128, out=4 -> action size)
      |
    tanh
```

DDPG - Deterministic Deep Policy Gradient

```
class Actor(nn.Module):
    """Actor (Policy) Model."""
    def __init__(self, state_size, action_size, seed, fc1_units=128, fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)

        self.model = nn.Sequential(
            nn.BatchNorm1d(state_size),
            nn.Linear(state_size, fc1_units),
            nn.ReLU(),
            nn.BatchNorm1d(fc1_units),
            nn.Linear(fc1_units, fc2_units),
            nn.ReLU(),
            nn.BatchNorm1d(fc2_units),
            nn.Linear(fc2_units, action_size),
            nn.Tanh()
        )

        self.model.apply(self.init_weights)

    def init_weights(self, m):
        if type(m) == nn.Linear:
            nn.init.xavier_uniform_(m.weight)
            m.bias.data.fill_(0.1)

    def forward(self, state): #takes in state/observation and outputs continuous action values
        return self.model(state)
```

DDPG - Deterministic Deep Policy Gradient

- **The critic includes two separate paths for observation and the actions**, and those paths are concatenated together to be transformed into the critic output of one number. The forward() function of the critic first transforms the observations with its first network - model_input, then concatenates the output and given actions to transform them using second network – model_output into one single value of Q.



```
Fully Connected Layer (in=33 -> state size, out=128)
                          |
                      RELU
                      BatchNorm1d
                          |
Fully Connected Layer (in=128+4, out=128)
                          |
                      RELU
                      BatchNorm1d
                          |
Fully Connected Layer (in=128, out=1)
```

DDPG - Deterministic Deep Policy Gradient

```
class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=128, fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()

        self.seed = torch.manual_seed(seed)

        self.model_input = nn.Sequential(
            nn.Linear(state_size, fc1_units),
            nn.ReLU(),
            nn.BatchNorm1d(fc1_units),
        )

        self.model_output = nn.Sequential(
            nn.Linear(fc1_units + action_size, fc2_units),
            nn.ReLU(),
            nn.Linear(fc2_units, 1),
        )

        self.model_input.apply(self.init_weights)
        self.model_output.apply(self.init_weights)

    def forward(self, state, action):  #takes in state/observation and action and outputs action value
        i = torch.cat([self.model_input(state), action], dim=1)
        return self.model_output(i)
```

DDPG - Deterministic Deep Policy Gradient

The Agent.act method - converts the observations into the appropriate form and ask the actor network to convert them into deterministic actions. Then it adds the exploration noise by applying the OU process. Lastly, it clips the actions to enforce them to fall into the -1..1 range.

```
def act(self, state, add_noise=True):

    """Returns actions for given state as per current policy."""
    state = torch.from_numpy(state).float().unsqueeze(0).to(device)

    self.actor_local.eval()
    with torch.no_grad():
                action = self.actor_local(state).cpu().data.numpy()
    self.actor_local.train()

    if add_noise:
         action += self.noise.sample()

     return np.clip(action, -1, 1)
```

DDPG - Deterministic Deep Policy Gradient

The Agent.learn method - on every iteration, we store the experience into the replay buffer and sample the training batch. Agent performs two separate training steps.

1.) To train the critic, we need to calculate the target Q-value using the one-step Bellman equation, with the target critic network as the approximation of the next state, then we calculate the MSE loss and ask the critic's optimizer to tweak the critic weights.

```
# ---------------------------- update critic ---------------------------- #

# Get predicted next-state actions and Q values from target models
actions_next = self.actor_target(next_states)
Q_targets_next = self.critic_target(next_states, actions_next)

# Compute Q targets for current states (y_i)
Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

# Compute critic loss
Q_expected = self.critic_local(states, actions)
critic_loss = F.mse_loss(Q_expected, Q_targets)

# Minimize the loss
self.critic_optimizer.zero_grad()
critic_loss.backward()
torch.nn.utils.clip_grad_norm(self.critic_local.parameters(), 1)
self.critic_optimizer.step()
```

DDPG - Deterministic Deep Policy Gradient

2.) To train actor, we need to update the actor's weights in a direction that will increase the critic's output. As both the actor and critic are represented as differentiable functions, what we need to do is just pass the actor's output to the critic and then minimize the negated value returned by the critic.

The negated output of the critic could be used as a loss to backpropagate it to the critic network and, finally, the actor. We don't want to touch the critic's weights, so it's important to ask only the actor's optimizer to do the optimization step. The weights of the critic will still keep the gradients from this call, but they will be discarded on the next optimization step.

```
# ---------------------------- update actor ---------------------------- #

# Compute actor loss
actions_pred = self.actor_local(states)
actor_loss = -self.critic_local(states, actions_pred).mean()

# Minimize the loss
self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()
```

DDPG - Deterministic Deep Policy Gradient

As the last step of the training loop, we perform **the target networks update using so called ‘soft sync’.** The soft sync is carried out on every step, but only a small ratio of the optimized network's weights are added to the target network. This makes a smooth and slow transition from old weight to the new ones.

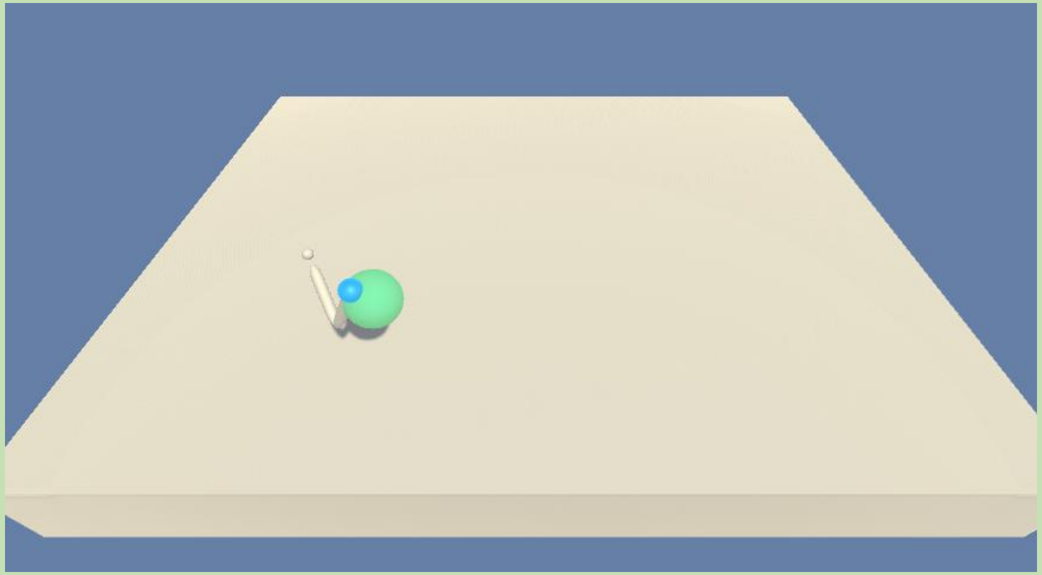
```
def soft_update(self, local_model, target_model, tau):
    """Soft update model parameters.
    θ_target = τ*θ_local + (1 - τ)*θ_target

    Params
    ======
        local_model: PyTorch model (weights will be copied from)
        target_model: PyTorch model (weights will be copied to)
        tau (float): interpolation parameter
    """
    for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
        target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

DDPG - Deterministic Deep Policy Gradient - Example: Reacher

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm.

Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector must be a number between -1 and 1.



MADDPG - Multi-Agent Deterministic Deep Policy Gradient

- Traditional reinforcement learning approaches such as Q-Learning or policy gradient are poorly suited to multi-agent environments. One issue is that each agent's policy is changing as training progresses, and the environment becomes non-stationary from the perspective of any individual agent (in a way that is not explainable by changes in the agent's own policy). This presents learning stability challenges and prevents the straightforward use of past experience replay, which is crucial for stabilizing deep Q-learning. Policy gradient methods, on the other hand, usually exhibit very high variance when coordination of multiple agents is required.

- The goal of **Multi-Agent DDPG algorithm** is to operate under under the following constraints:

  1.) the learned policies can only use local information (i.e. their own observations) at execution time
  2.) it does not assume a differentiable model of the environment dynamics
  3.) it does not assume any particular structure on the communication method between agents

- The Multi-Agent DDPG algorithm accomplishes the above mentioned constraints by adopting the framework of **centralized training with decentralized execution.**

MADDPG - Multi-Agent Deterministic Deep Policy Gradient

- **The Multi-Agent DDPG algorithm is an extension of actor-critic policy gradient methods where the critic is augmented with extra information about the policies of other agents.**

- A primary motivation /behind MADDPG is that, if we know the actions taken by all agents, the environment is stationary even as the policies change. This is not the case if we do not explicitly condition on the actions of other agents, as done for most traditional RL methods.

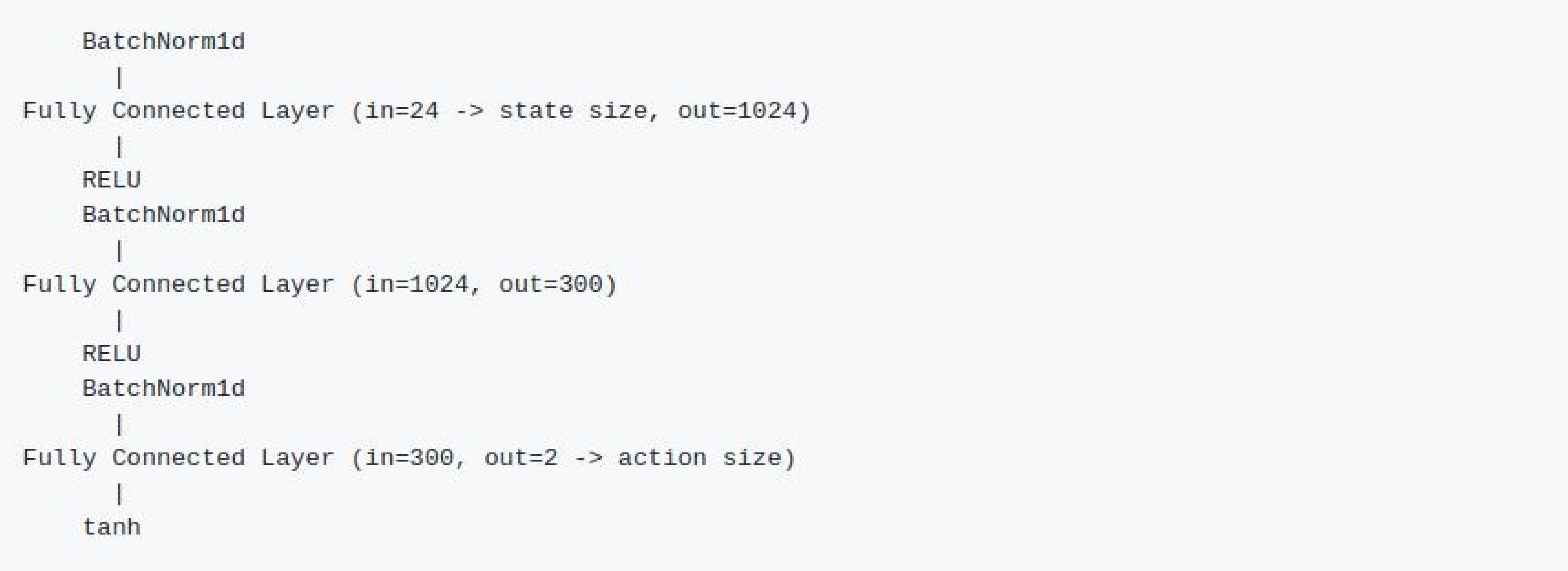
- MADDPG is a multi-agent version of DDPG. DDPG is well suited to continuous control tasks and this just extends it to a multi-agent scenario. More details can be found in the https://arxiv.org/abs/1706.02275.

- In MADDPG, each agent has it's own actor and it's own critic. Agents share a common experience replay buffer which contains tuples with states and actions from all agents. Each agent does it's own sampling from shared replay buffer. This allows agents to learn their own reward function and incorporate the actions of other agents in their learning. Therefore, we can train them in collaborative, competitive, or mixed environments.

- The model consists of two separate networks for the actor and critic.

MADDPG - Multi-Agent Deterministic Deep Policy Gradient

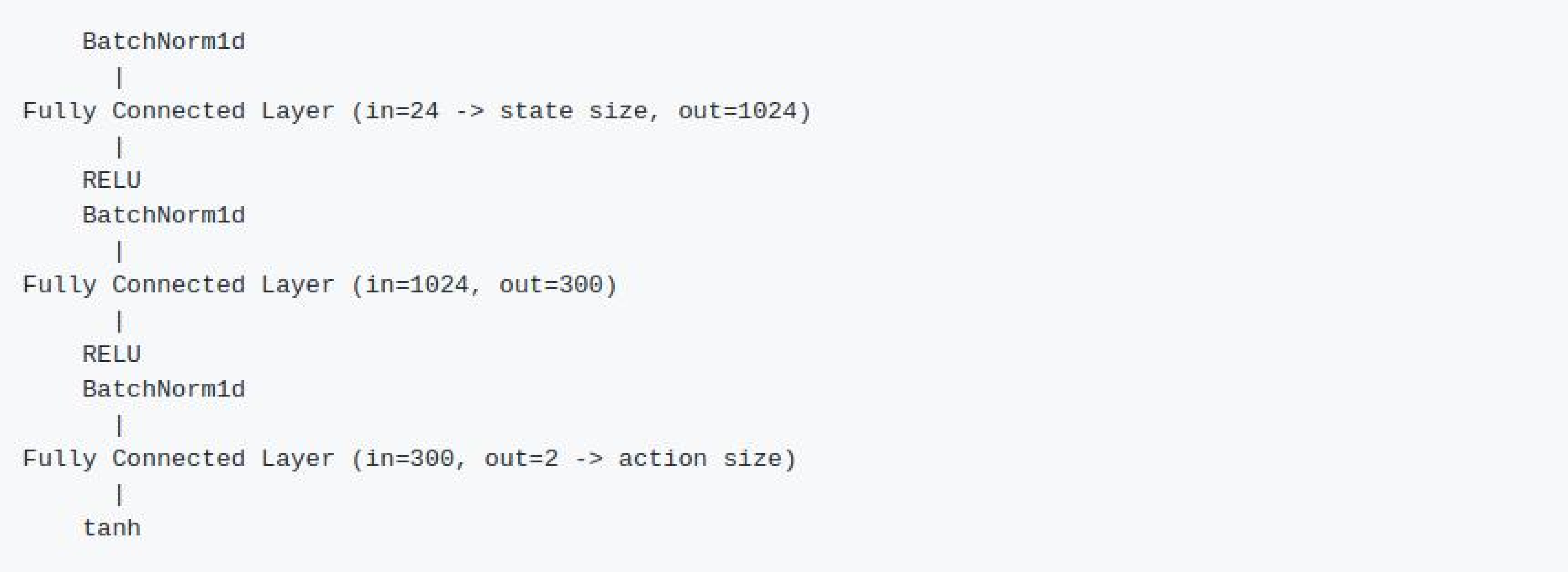
- **The Actor** is fully connected feed-forward network with two hidden layers, RELU activation and Batch Normalization. The input is an state vector(dim=24), while the output is a vector with 2 values, one for each action. The output actions are transformed with hyperbolic tangent non-linearity to squeeze the values to the -1..1 range.



```
    BatchNorm1d
      |
Fully Connected Layer (in=24 -> state size, out=1024)
      |
    RELU
    BatchNorm1d
      |
Fully Connected Layer (in=1024, out=300)
      |
    RELU
    BatchNorm1d
      |
Fully Connected Layer (in=300, out=2 -> action size)
      |
    tanh
```

MADDPG - Multi-Agent Deterministic Deep Policy Gradient

- **The Critic** is fully connected feed-forward network with two hidden layers, RELU activation and Batch Normalization in the input layer. **The critic network estimates centralized action-value function that takes as input the observations of all agents and the actions of all agents, and outputs the Q-value for agent i.** Since each action-value function is learned separately, agents can have arbitrary reward structures, including conflicting rewards in a competitive setting.

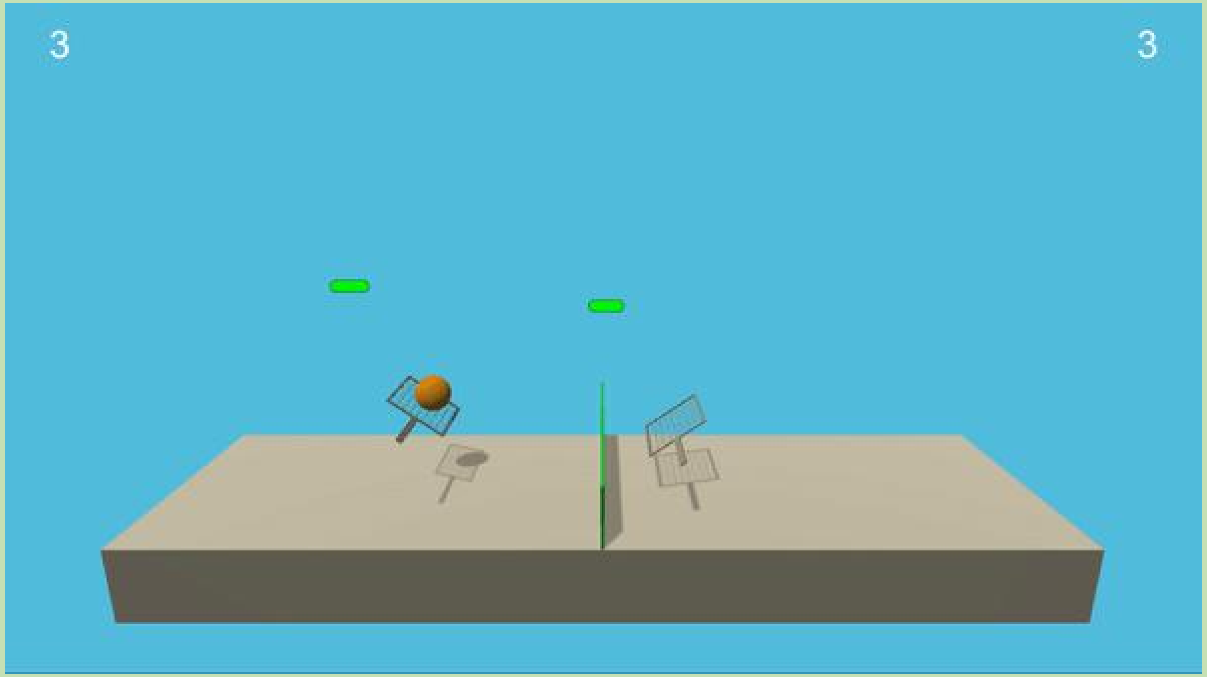


```
    BatchNorm1d
      |
Fully Connected Layer (in=24 -> state size, out=1024)
      |
    RELU
    BatchNorm1d
      |
Fully Connected Layer (in=1024, out=300)
      |
    RELU
    BatchNorm1d
      |
Fully Connected Layer (in=300, out=2 -> action size)
      |
    tanh
```

MADDPG - Multi-Agent Deterministic Deep Policy Gradient - Example: Tennis

- In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

- The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.



SAC - Soft Actor-Critic Agent

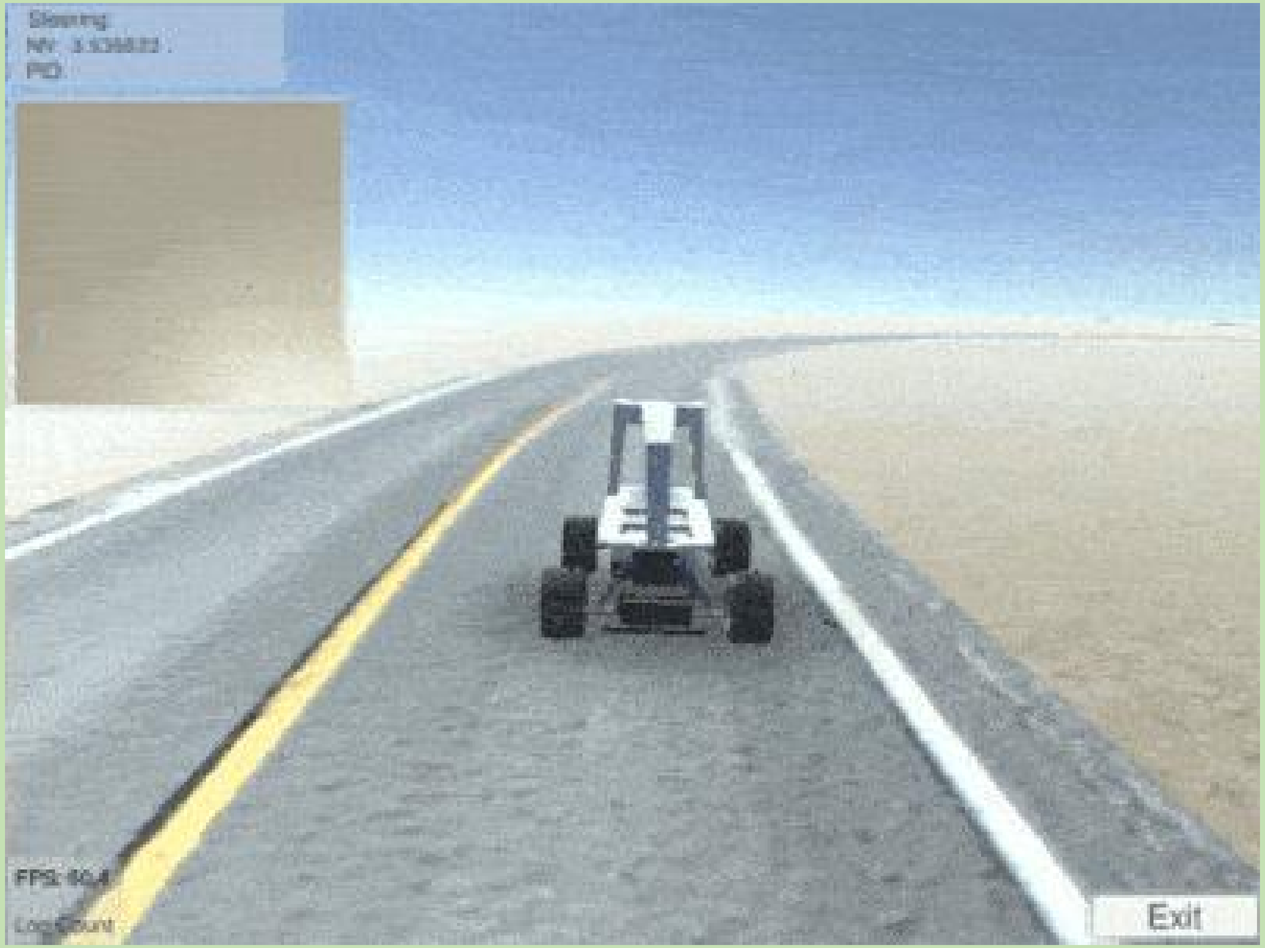
- Some of the most successful RL algorithms in recent years such as Trust Region Policy Optimization (TRPO), Proximal Policy Optimization (PPO) and Asynchronous Actor-Critic Agents (A3C) suffer from **sample inefficiency**. This is because they learn in an “on-policy” manner, i.e. they need completely new samples after each policy update.

- In contrast, Q-learning based “off-policy” methods such as Deep Deterministic Policy Gradient (DDPG) and Twin Delayed Deep Deterministic Policy Gradient (TD3PG) are able to learn efficiently from past samples using experience replay buffers. However, the problem with these methods is that they are very sensitive to hyperparameters and require a lot of tuning to get them to converge.

- Soft Actor-Critic follows in the tradition of the latter type of algorithms and adds methods to combat the convergence brittleness.

- A central feature of SAC is entropy regularization. The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. This has a close connection to the exploration-exploitation trade-off:

SAC - Soft Actor-Critic Agent - Learning to Drive


Steering
NN:
PID:
FPS:
Log Count
Exit