BIOMEDICAL
COMPUTER
VISION

Reinforcement Learning

Jaime Cascante, Adrián S. Volcinschi, Juan C. Pérez

Advanced ML

Introduction

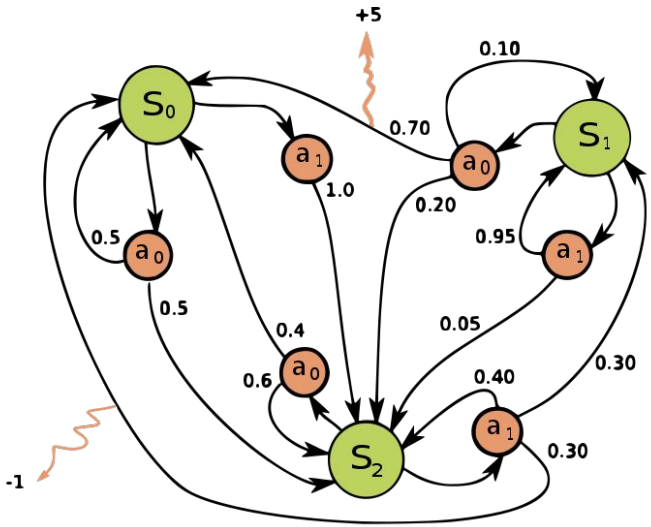
- RL is about *interaction* with the environment
- Concepts of:
  - Cause-and-effect
  - Consequences of actions
  - How to achieve goals
- Learn about:
  - Environment
  - Ourselves!
- Computational approach
- Focused on goal-directed learning
  - Maximize for single variable: von Neumann, Morgenstern and D’Alembert.

Introduction: *what to learn?*

- Want to learn *what to do*:
  - f: S -> A
  - That maximize numerical signal
- Learner will not be told what to do (maybe not even supervisor knows!)
  - Discover actions that yield the most reward by trying them
- Actions and immediate reward
  - Also next situation. Hence: subsequent rewards
- Classical properties of RL:
  - Trial-and-error search
  - Delayed reward

Introduction: how is RL formalized?

- Ideas of dynamical systems theory:
  - Optimal control
  - Incompletely-known Markov Decision Processes
- State of the environment
- Actions that affect the state
- IKMDPs include all these aspects: sensation, action and goal

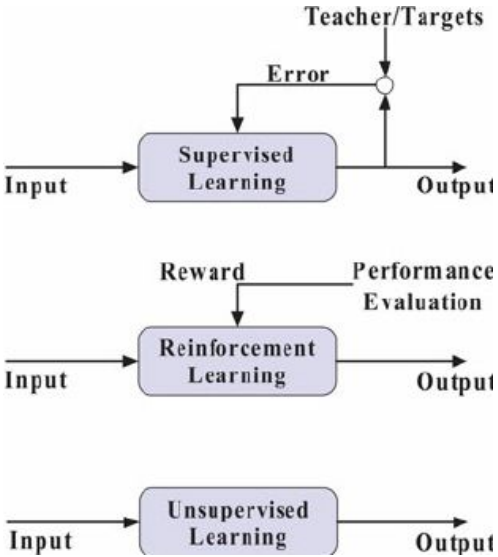


Introduction: RL vs. Supervised Learning (SL)

- SL: learn from set of **labeled examples**
- Objective is to extrapolate/generalize responses: **act correctly in situations not present in training set**
- Alone, not adequate for learning with interaction
    - Need examples of desired behavior that are correct and representative

Introduction: RL vs. Unsupervised Learning (UL)

- UL: find structure in **unlabeled data**
- Similar? RL doesn't rely on examples (exactly)
- But RL *is* trying to maximize something
- UL can be useful for RL
- Sutton & Barto consider RL to be different from both SL and UL



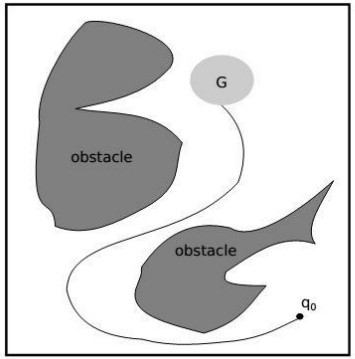
Introduction: particularities of RL

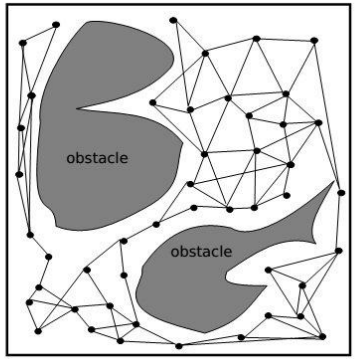
- Exploration vs exploitation
  - Exploration is tougher if system is stochastic.
- RL considers the problem as a whole
  - Yes, study subproblems, but **keep in mind the big picture**
- Real-time may be important
- Agents have **goals**, **senses** and **can interact**
- Usual assumption: **uncertainty** is inherent
- SL can be involved to point out critical capabilities

Introduction: modern RL

- Interacts with engineering and scientific disciplines
- Ability to learn with parameterized approximators addresses the “curse of dimensionality”
- Interaction with psychology and neuroscience
- Sutton & Barto: “RL is the closest to the kind of learning that humans and other animals do”

Introduction: RL examples





Any kind of locomotion!

(very) Brief History

- Animal behavior & learning: learn by trial-and-error
  - The Law of Effect
    - The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond. *Thorndike, 1911*.
- Optimal control and its solution using value functions and dynamic programming (DP)
  - Hamilton & Jacobi -> Bellman: state and value function to define functional equation: The **Bellman equation (BE).**
  - Methods to solve opt. control problems by solving the BE -> DP
  - Bellman introduced MDPs: discrete stochastic version of opt. control problems.
  - Then, partially-observable MDPs
- Turing thought of this kind of learning
  - Intelligent machines reaching configuration for which action is undetermined

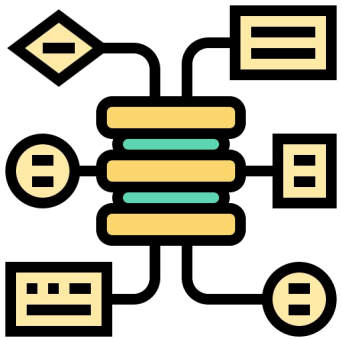
(very) Brief History

- Minsky: how to distribute credit for success among many decisions
- Widrow, Gupta & Maitra, 1973: learn with a **critic**, not a **teacher**
- Klopf: *essential aspects of adaptive behavior are lost as learning researchers focus exclusively on supervised learning*
- Take hedonism into account (?)
- DP suffers from “curse of dimensionality”
- Temporal difference methods

❖ Connection between (opt. control & dyn. prog.) and (learning):

- ➢ Not fast. People wanted accurate system models & analytic solutions to the BE.
- ➢ Chris Watkins (1989) PhD. thesis
- ➢ Bertsekas & Tsitsiklis (1996): DP + neural nets

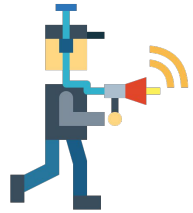
Elements of RL



Agent

Definition
Element that interacts over time with its environment to achieve a goal

Agent

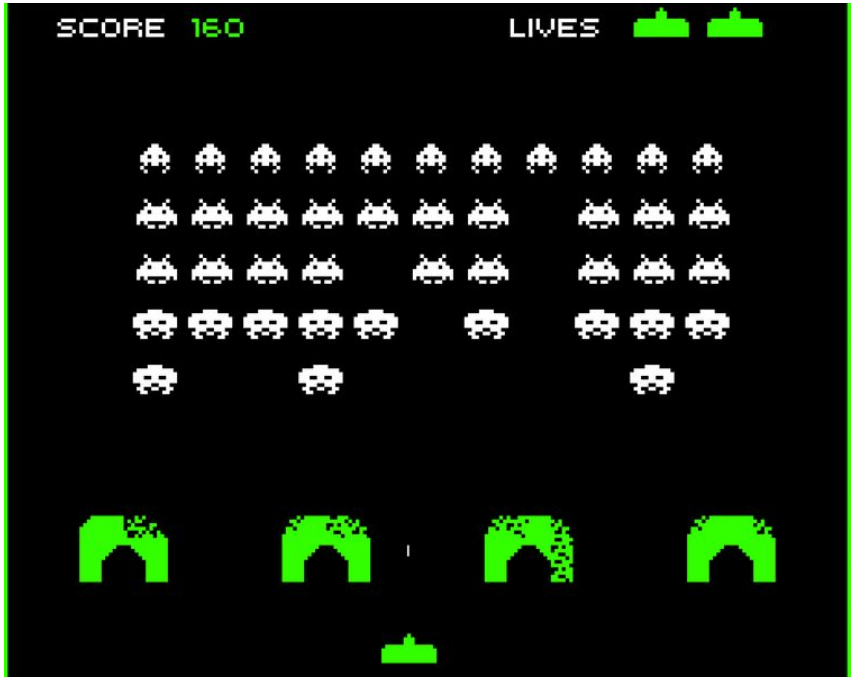


Definition
Element that interacts over time with its environment to achieve a goal

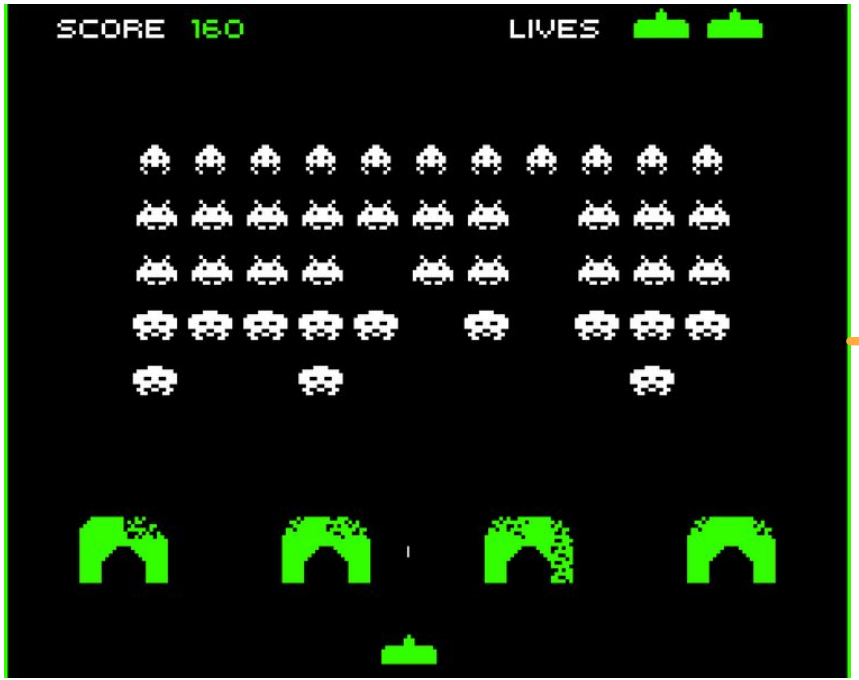
Environment

Definition
Space representation where the agent is and acts

Example: Space invaders

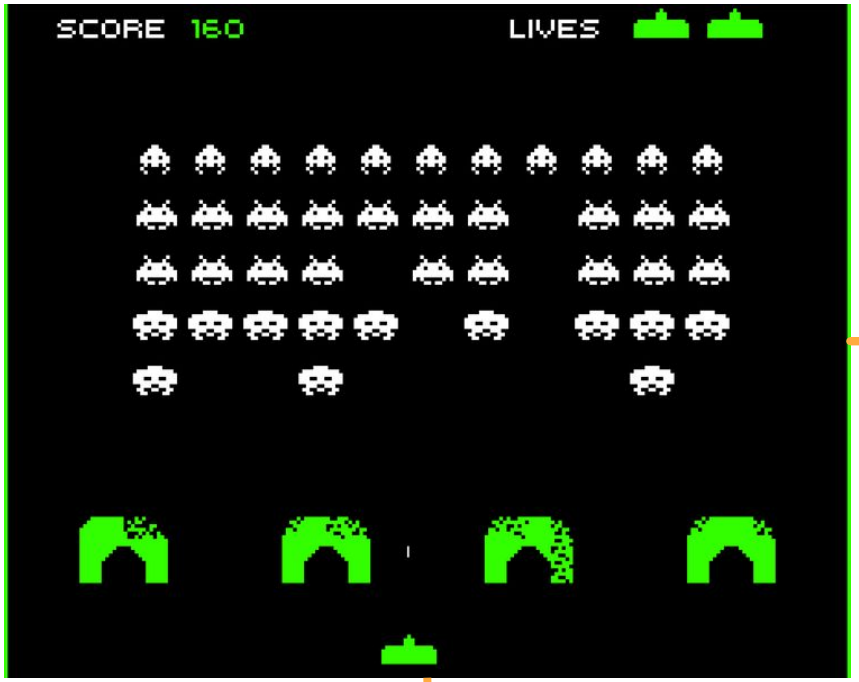


Example: Space invaders



Environment

Example: Space invaders



Environment

Agent

Policy (π)

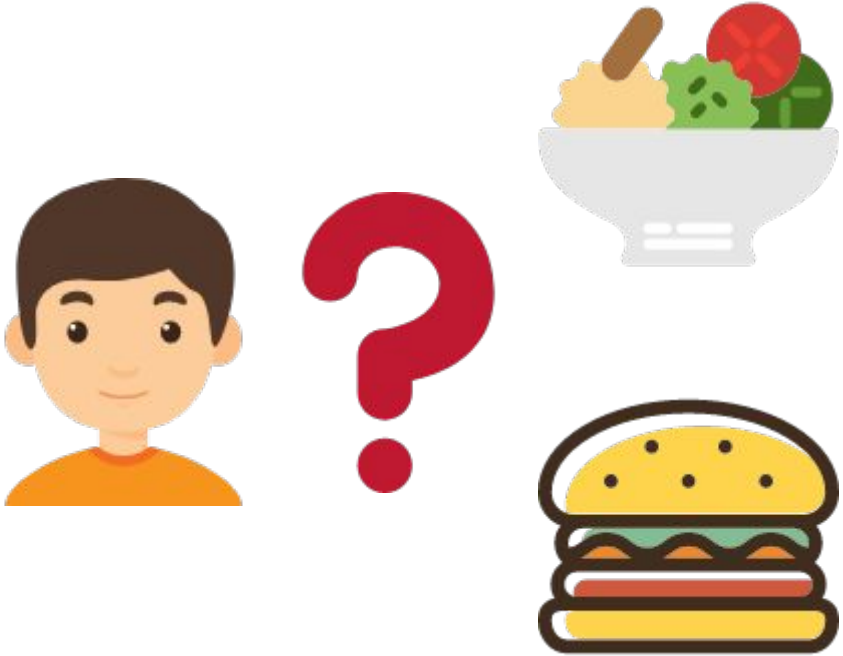
Definition
Element that defines the learning agent's way of behaving at a given time

Policy (π)

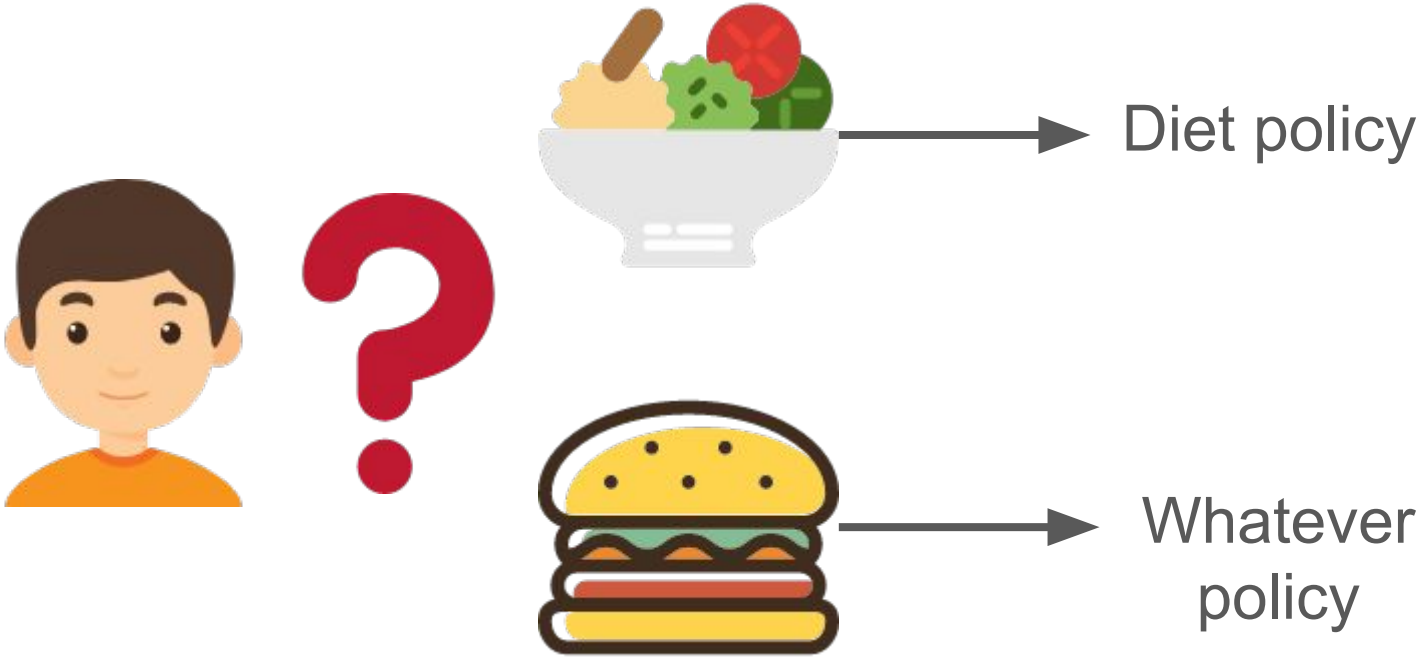
Definition
Element that defines the learning agent's way of behaving at a given time

- Mapping from states to actions
  - f: States $\rightarrow$ Actions
- Can be a simple function, complex function or even a lookup table
- Can be, but not always, stochastic

Example: Diet



Example: Diet



Reward Signal (R)

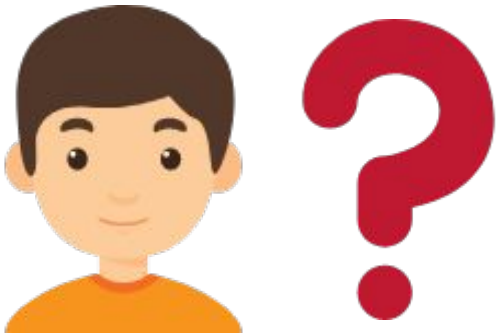
Definition
Scalar number that defines the goal of the agent

Reward Signal (R)

Definition
Scalar number that defines the goal of the agent

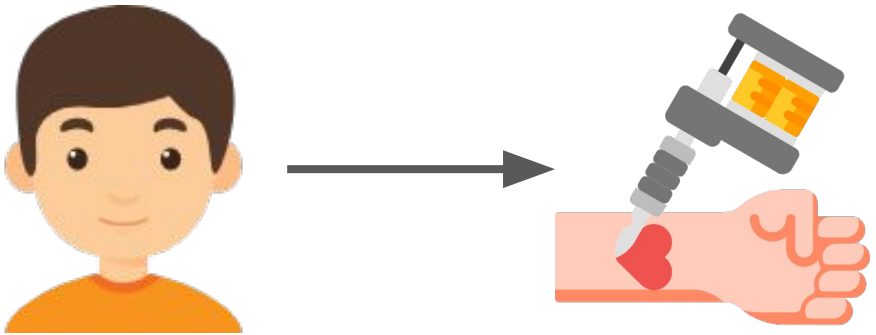
- After each time step the environment sends a **reward** to the learning agent as a "**feedback**" of the state in which the agent is
- Agent's objective: **Maximize** something in regards with the rewards

Example: Pain vs Pleasure

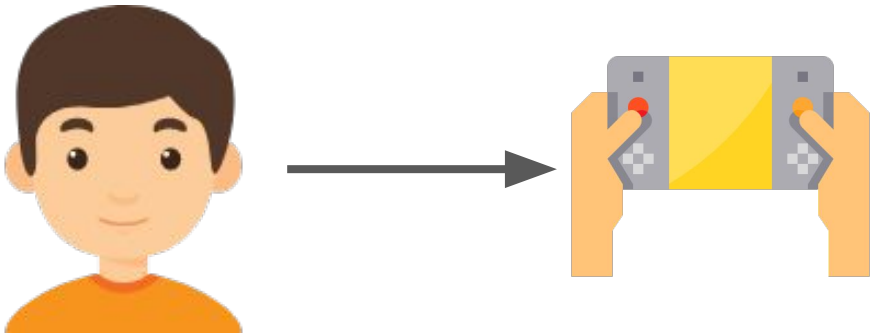


Example: Pain vs Pleasure

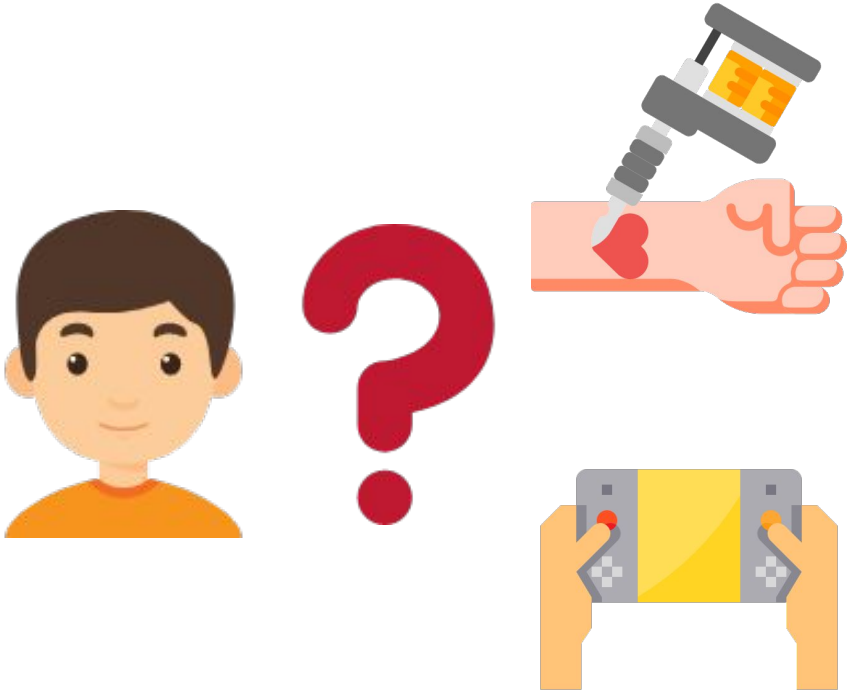
Example: Pain vs Pleasure



Example: Pain vs Pleasure



Example: Pain vs Pleasure



State-Value Function (V)

Definition
Function that returns the value of a state, or in other words the total amount of reward the learning agent can expect to accumulate starting from that state

- This function is going to help the agent decide which state is better to be in and because of it, it is going to help the agent decide which actions to take to get there
- Estimating this function is **really hard** and therefore efficient estimation methods must be used

Action-Value Function (Q)

Definition

Function that returns the value of a state-action pair, or in other words the total amount of reward the learning agent can expect to accumulate starting from that state and taking a given action

Model of the Environment

Definition
Model that predicts how the environment will behave

- This model are used for **planning**
- The existence or not of this model will define the problem as **Model-free** or ***Model-based***

Markov Decision Processes

Markov Property

Definition

A state S_t is Markov if and only if

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t]$$

"The future is independent of the past given the present"

- The state captures all relevant information from the **history**

Markov Process

Definition

A *Markov Process* is a tuple⟨S, P ⟩

- **S is a (finite) set of states**
- **P is a state transition probability matrix**

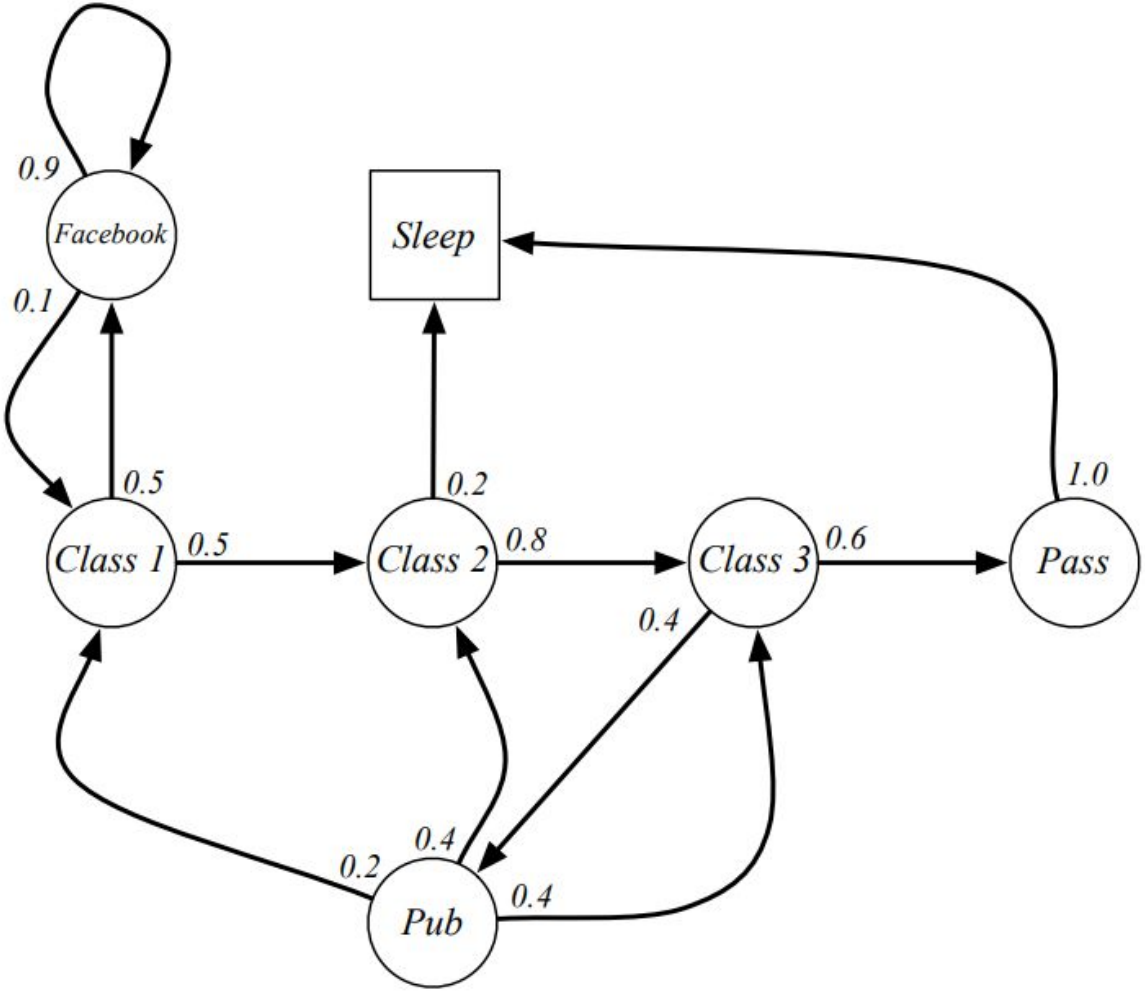
Markov Process

Definition

A *Markov Process* is a tuple⟨S, P ⟩

- **S is a (finite) set of states**
- **P is a state transition probability matrix**

$$P = \begin{bmatrix} P_{1,1} & P_{1,2} & \dots & P_{1,j} & \dots & P_{1,S} \\ P_{2,1} & P_{2,2} & \dots & P_{2,j} & \dots & P_{2,S} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{i,1} & P_{i,2} & \dots & P_{i,j} & \dots & P_{i,S} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{S,1} & P_{S,2} & \dots & P_{S,j} & \dots & P_{S,S} \end{bmatrix}.$$

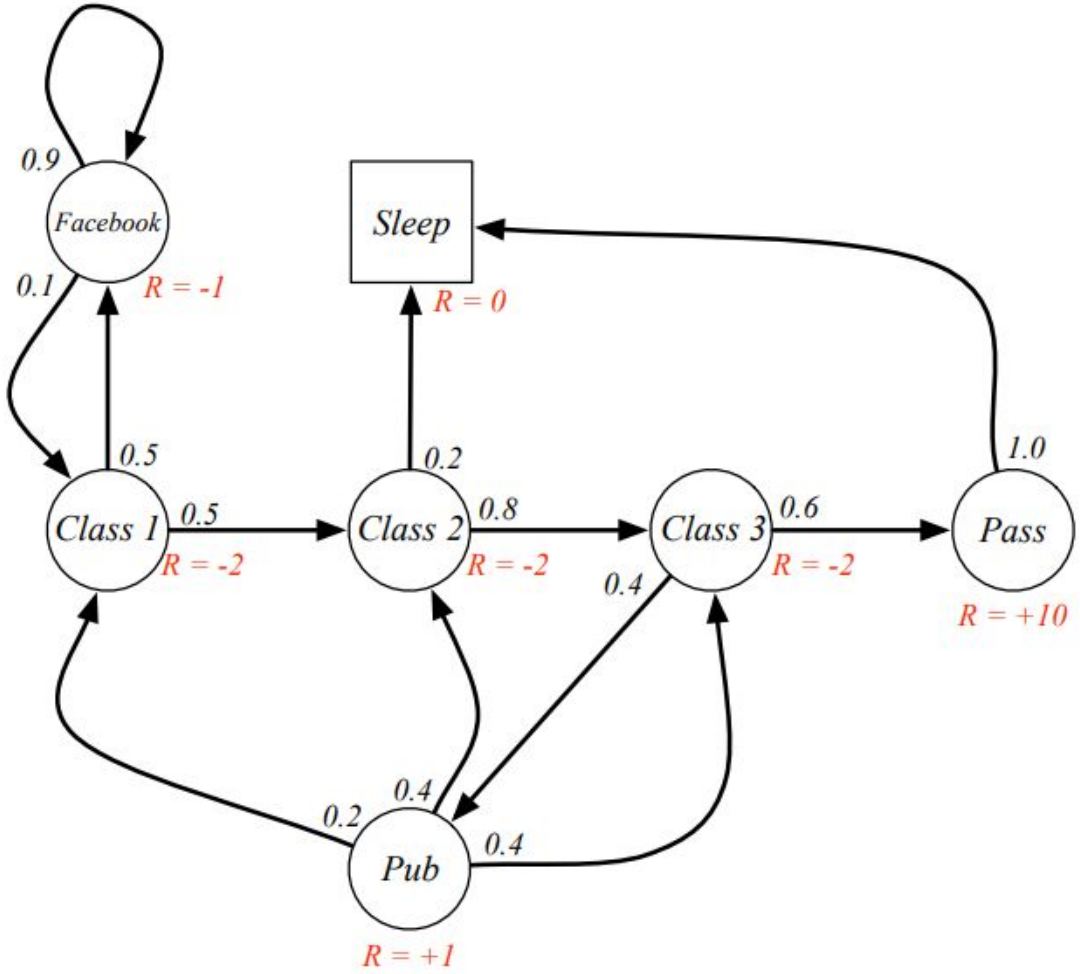

0.9
Facebook
0.1
Sleep
0.5
0.2
1.0
Class 1
0.5
Class 2
0.8
Class 3
0.6
Pass
0.4
0.4
0.2
Pub
0.4

Markov Reward Process

Definition

A *Markov Reward Process* is a tuple⟨S, P, R, γ⟩

- **S is a finite set of states**
- **P is a state transition probability matrix**
- **R is a reward function** $R_s^a = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$
- **γ is a discount factor γ ∈ [0,1]**

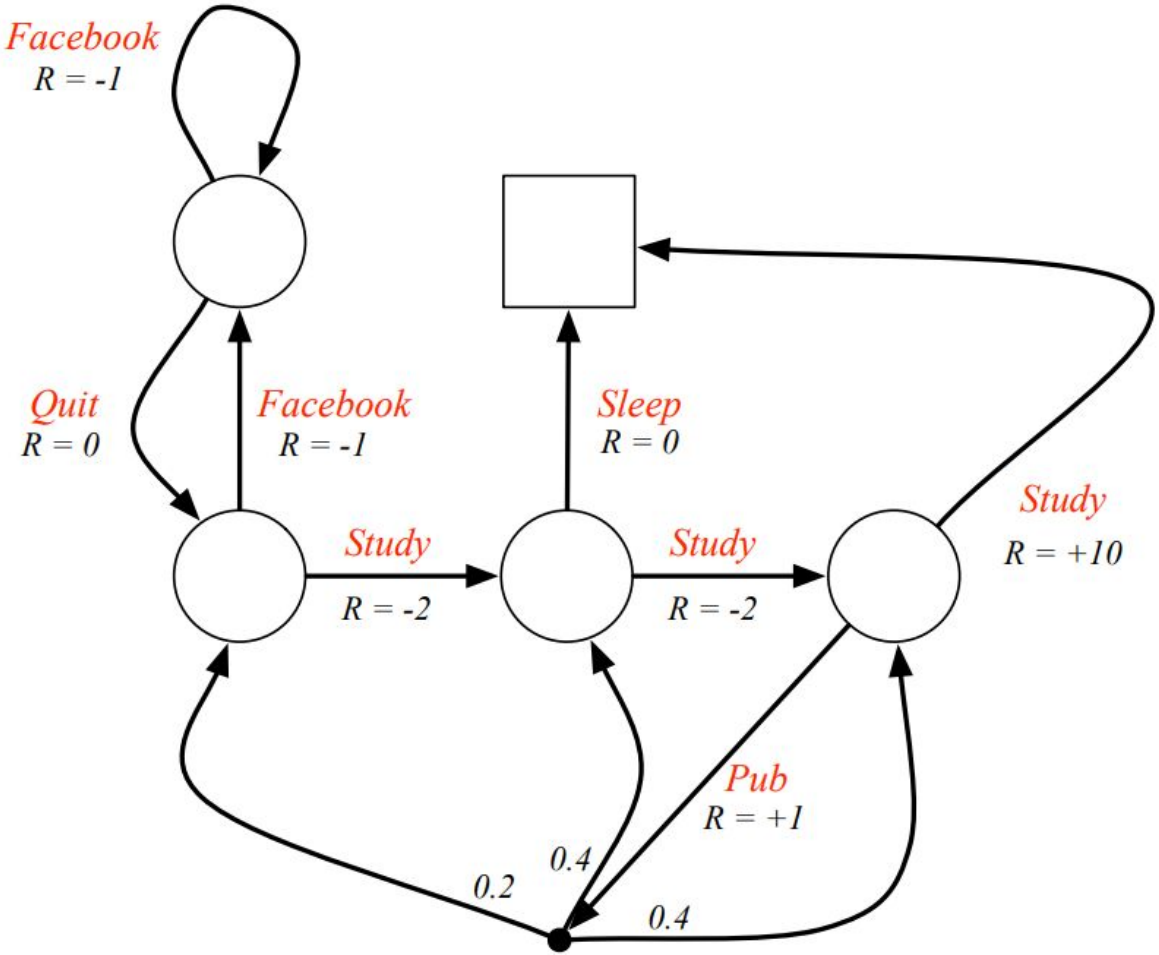

0.9
Facebook
0.1
R = -1
Sleep
R = 0
0.5
0.2
1.0
Class 1
0.5
Class 2
0.8
Class 3
0.6
Pass
R = -2
R = -2
R = -2
0.4
R = +10
0.2
0.4
0.4
Pub
R = +1

Markov Decision Process

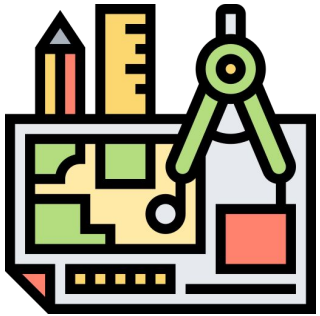
Definition

A *Markov Decision Process* is a tuple⟨S, P, *A,* R, γ⟩

- **S is a finite set of states**
- **P is a state transition probability matrix**
- **A is a finite set of actions**
- **R is a reward function** $R_s^a = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$
- **γ is a discount factor γ ∈ [0,1]**

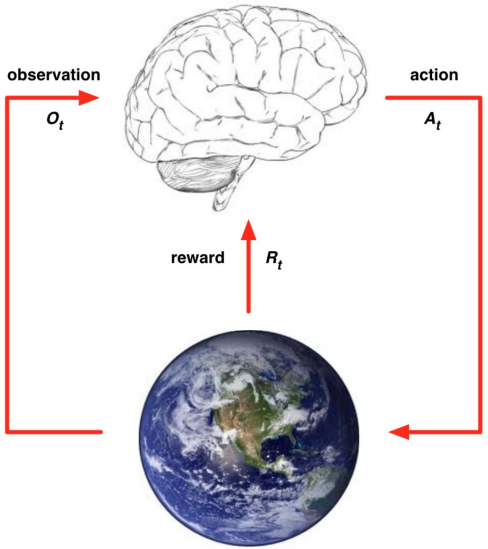

Facebook
R = -1
Quit
R = 0
Facebook
R = -1
Sleep
R = 0
Study
R = -2
Study
R = -2
Study
R = +10
Pub
R = +1
0.2
0.4
0.4

Does model matters?



Planning vs Learning distinction (Optimal Control vs RL)

- Solving a dynamic programming problem with model-free vs model based simulation.

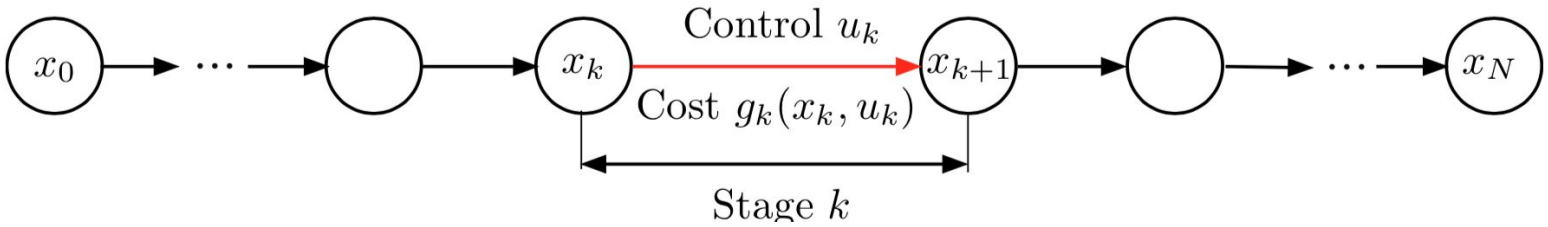


VS

$$(\mathbf{s}_k, a_k) \rightarrow \boxed{\text{f}} \rightarrow \mathbf{s}_{k+1}$$

- F captures environment dynamics

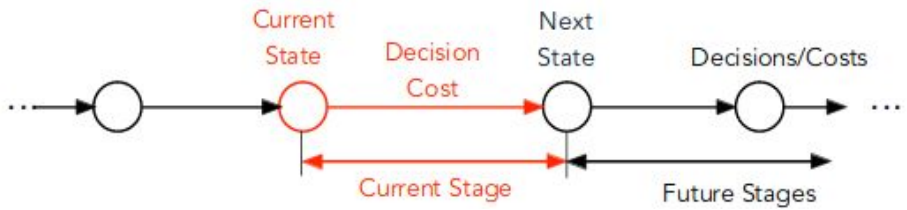
General RL/Control Idea



- Agent = Decision maker or controller.
- Action = Decision or control (u_k)
- Environment = Dynamic system (State x_k)
- Objective: max Reward / min Cost

Model Based (Control/Planning idea)

- Dynamic System / environment evolution is known:



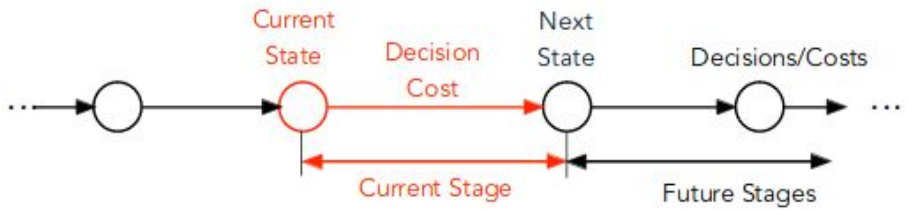
- Sub-optimal policy: At current state, apply decision that minimize:

$$\arg\min_{a} G(s_k, a_k)$$
$$s.t \quad \mathbf{s}_{k+1} = (\mathbf{s}_k, a_k)$$

G is the cost function

Model Based (Control/Planning idea)

- Dynamic System / environment evolution is known:



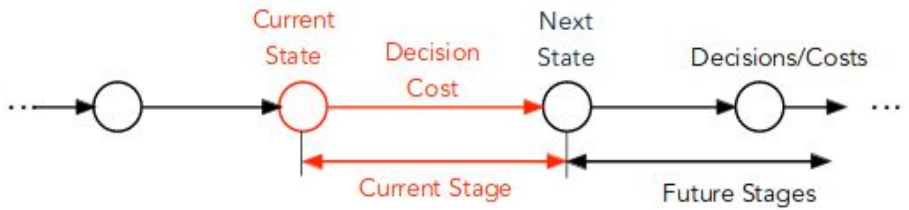
- Sub-optimal policy: At current state, apply decision that minimize:

$$\arg\min_{a} G(s_k, a_k)$$
$$s.t \quad \mathbf{s}_{k+1} = (\mathbf{s}_k, a_k)$$

Why is sub-optimal?

Model Based (Control/Planning idea)

- Dynamic System / environment evolution is known:



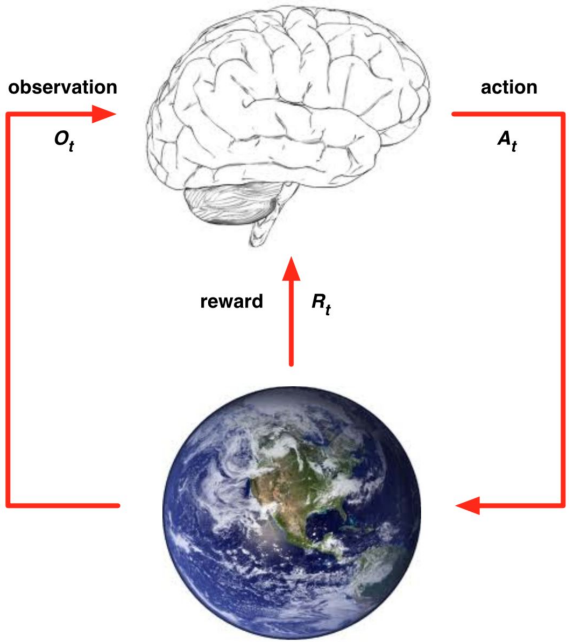
- Sub-optimal policy: At current state, apply decision that minimize:

$$\arg\min_{a} G(s_k, a_k)$$
$$s.t \quad \mathbf{s}_{k+1} = (\mathbf{s}_k, a_k)$$

Why is sub-optimal? -> no horizon prediction

Model free (Learning idea)

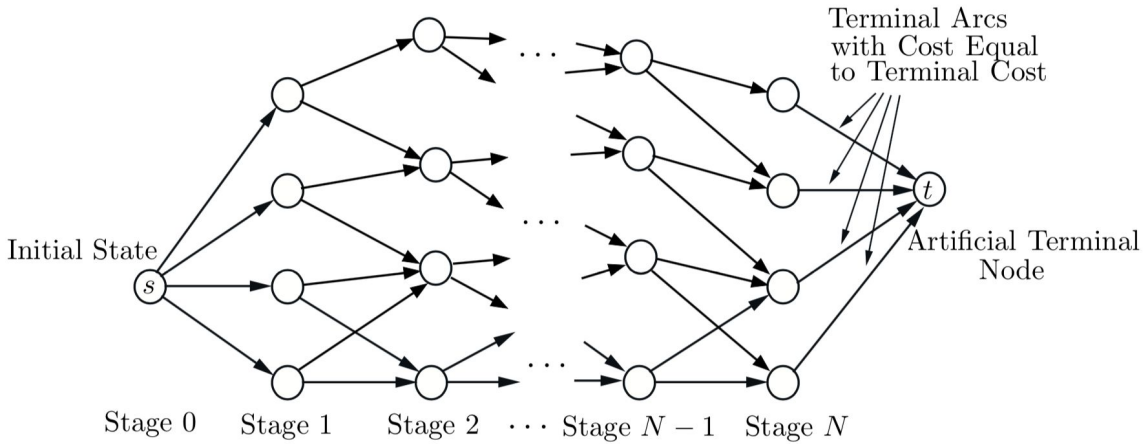
- Dynamic System / environment evolution is completely unknown!



Model free (Learning idea)

- Dynamic System / environment evolution is completely unknown :(

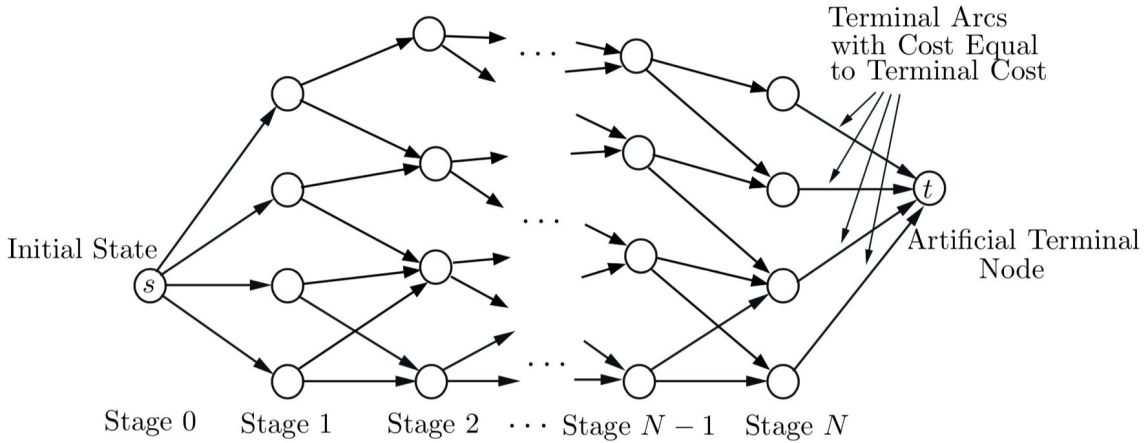
Idea: Learn dynamics sampling from the environment!



Model free (Learning idea)

- Dynamic System / environment evolution is completely unknown :(

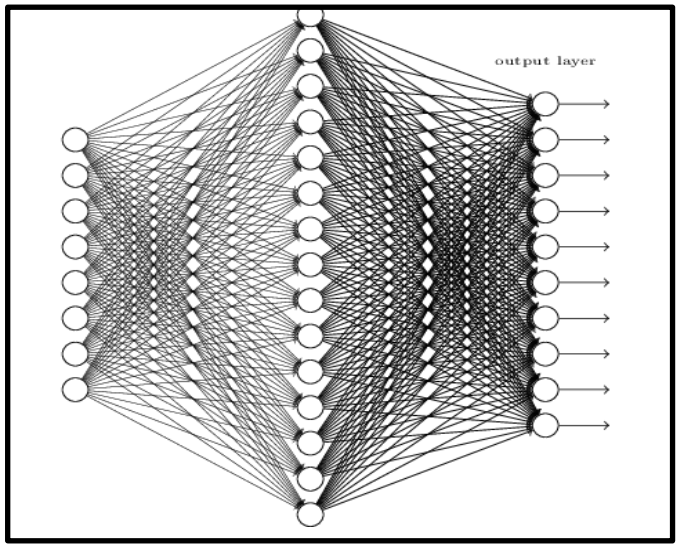
  Problem: States and actions are discretized.



Model free (Learning idea)

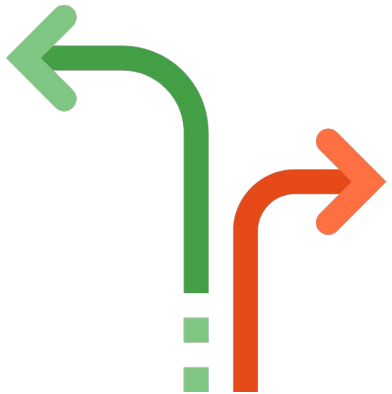
- Dynamic System / environment evolution is completely unknown :(

  Idea: Approximate mapping function f with function approximator

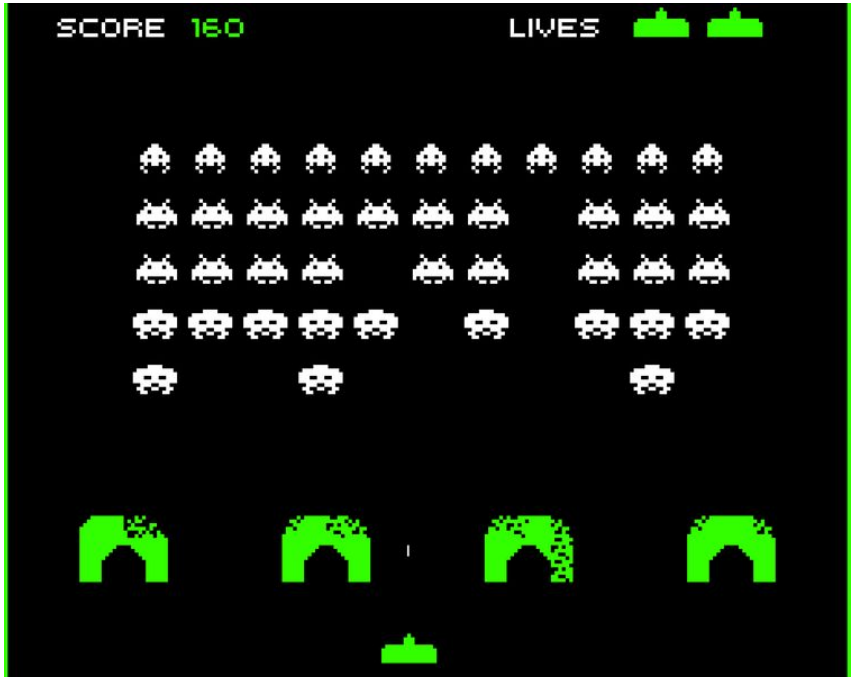
$(\mathbf{s}_k, a_k)$ → 

→ $\mathbf{s}_{k+1}$

Types of problems



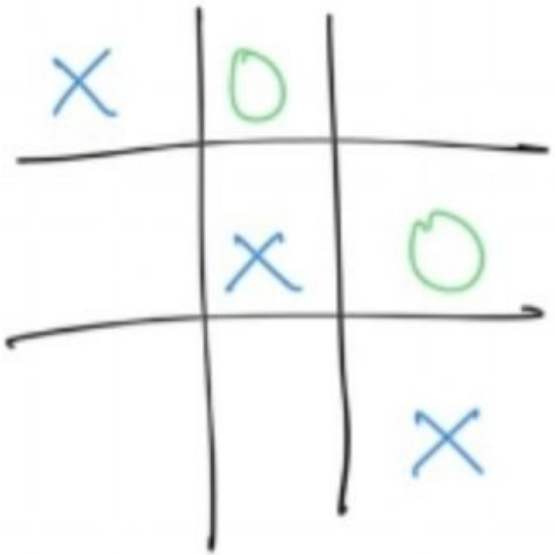
Discrete State-Action problems



- Even dynamics are not known rules are!
- Action set cardinality is 3.
- State set cardinality is huge but finite.

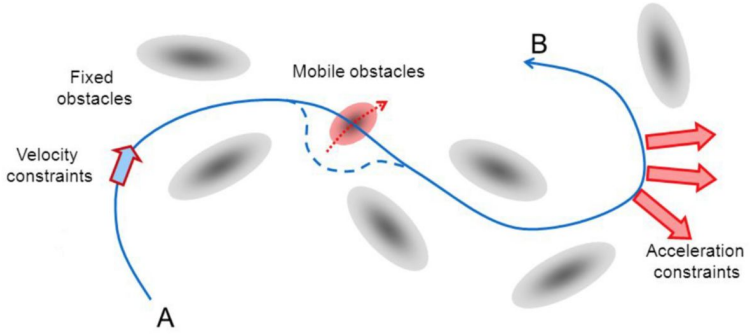
Discrete State-Action problems

- Tic-Tac-Toe



- Action set cardinality is 9.
- State set cardinality is 765 valid states.

Continuous State-Action problems?



PATH PLANNING

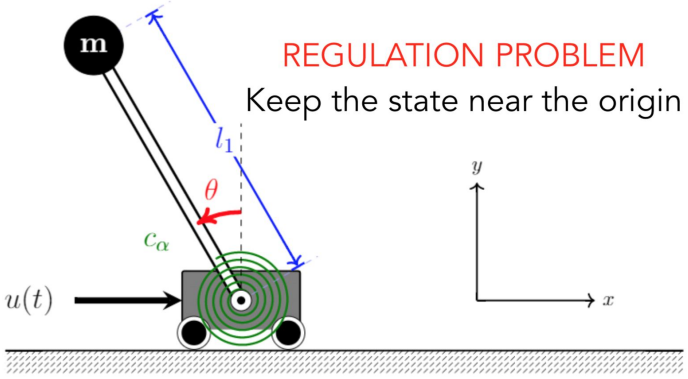
Keep state close to a trajectory

Motion equations

$$x_{k+1} = f_k(x_k, u_k)$$

Penalty for deviating from nominal trajectory

State and control constraints



REGULATION PROBLEM

Keep the state near the origin

How to solve continuous State-Action problems?

PATH PLANNING

Keep state close to a trajectory

More at the end!!!

REGULATION PROBLEM

Keep the state near the origin

Bellman Optimality

Idea of optimality

Definition

An optimal state-value function $v_*(s)$ is the maximum value function over all policies

$$v^\star(s) = \max_\pi v_\pi(s)$$

An optimal action-value function $q_*(s)$ is the maximum action-value function over all policies

$$q^\star(s) = \max_\pi q_\pi(s)$$

Idea of optimality

- Previous definition suppose there exist policy that maximize both state-action value function

Idea of optimality

- Previous definition suppose there exist policy that maximize both state-action value function

- Then we can compare between policies

$$\pi \geq \pi' \text{ if } \quad v_\pi(s) \geq v_{\pi'}(s)$$

Idea of optimality – Optimal Policy

- Then we can compare between policies

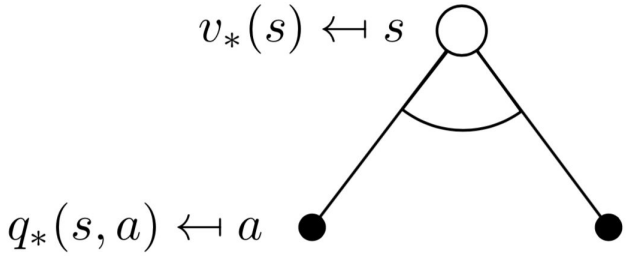
$$\pi \geq \pi' \text{ if } \quad v_\pi(s) \geq v_{\pi'}(s)$$

- If \pi is optimal then:
  - Optimal policy achieve both optimal state-value action-value function

$$q_\pi = q^\star(s) \qquad v_\pi = v^\star(s)$$

Bellman Optimality Equations

- State-value optimality

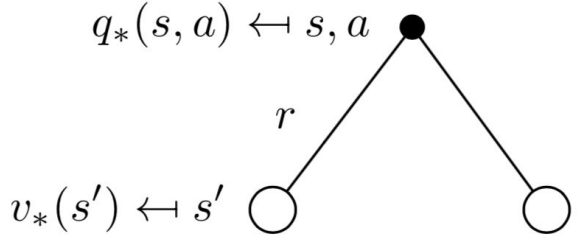


$$v_*(s) = \max_a q_*(s,a)$$

- Action-value optimality

Bellman Optimality Equations

- State-value optimality



- Action-value optimality

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

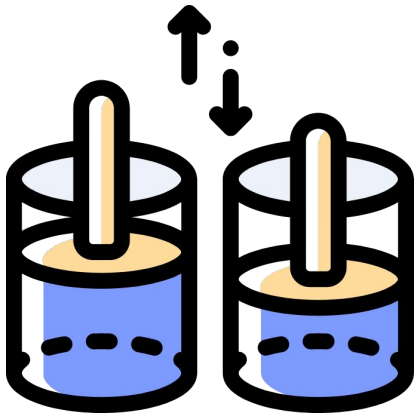
Bellman Optimality Equations

- Combining both for recursivity:

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')$$

$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

Dynamic Programming



Requirements for Dynamic Programming

- Optimal substructure
  - Optimal solution (e.g $v_*(s)$) can be decomposed into subproblems (Bellman equation)

- Overlapping subproblems
  - Subproblems recur many times
  - Solutions can be cached and reused (Value function)

Prediction vs Control

Prediction

Input: MDP (S,A,P,R,γ) and policy π

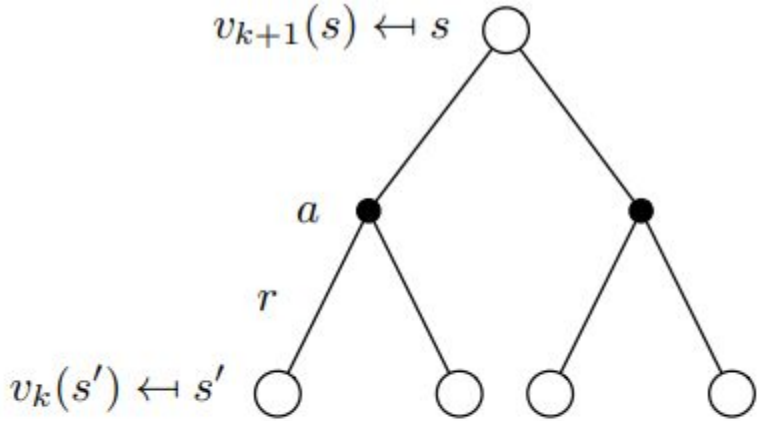
Output: Value function v_π

Control

Input: MDP (S,A,P,R,γ)

Output: Optimal Value function v_* and optimal policy π_*

Prediction: Policy Evaluation



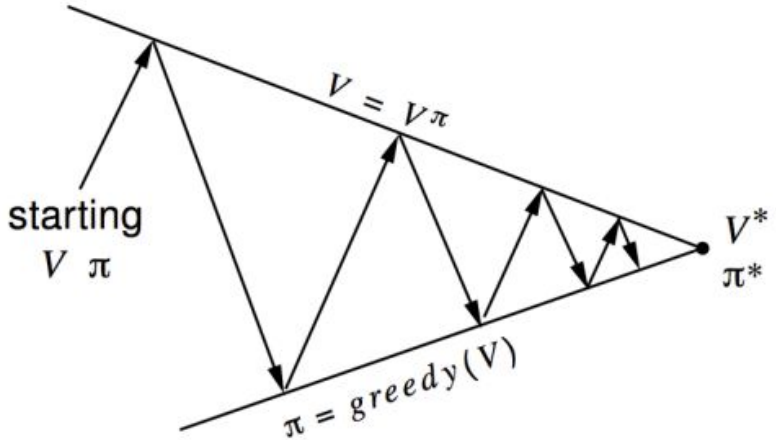
$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

Prediction: Policy Evaluation

Iterative policy evaluation

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

Control: Policy Iteration

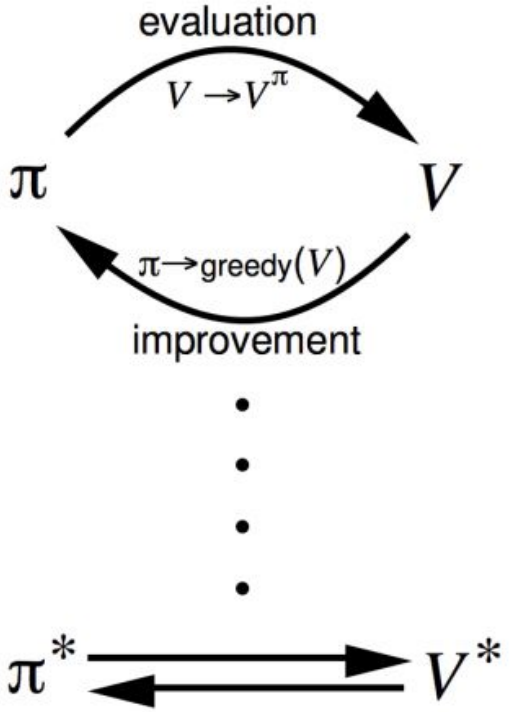


Policy evaluation Estimate v_π

Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$

Greedy policy improvement



Control: Policy Iteration

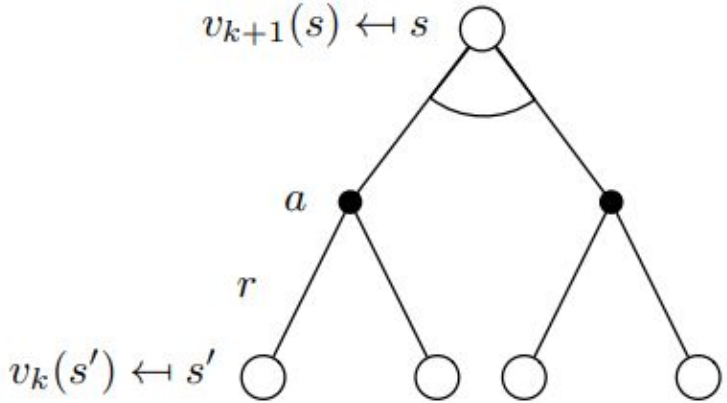
Policy iteration (using iterative policy evaluation)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Control: Value Iteration



$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

Control: Value Iteration

Value iteration

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

Monte-Carlo
vs
Temporal-Difference

Model-Free Prediction

Monte-Carlo

1. Sample an episode under π (Each state S_t has a G_t associated)
2. Define step size α
3. $V(S_t) = V(S_t) + \alpha(G_t - V(S_t))$

Model-Free Prediction

Temporal-Difference

1. Sample a piece of an episode under π
2. Define step size α
3. Define a decay rate γ
4. $V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$

Advantages and Disadvantages

Monte-Carlo

- High variance
- Zero bias
- Not very sensitive to initial values
- Better for non-Markov environments

Temporal-Difference

- Low variance
- Some bias
- More sensitive to initial values
- Better for Markov environments

On-Policy Learning

Definition
Learn about policy π from experience sampled from π

Off-Policy Learning

Definition
Learn about policy π from experience sampled from μ

Model-Free Control

Remember this equation?

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

Model-Free Control

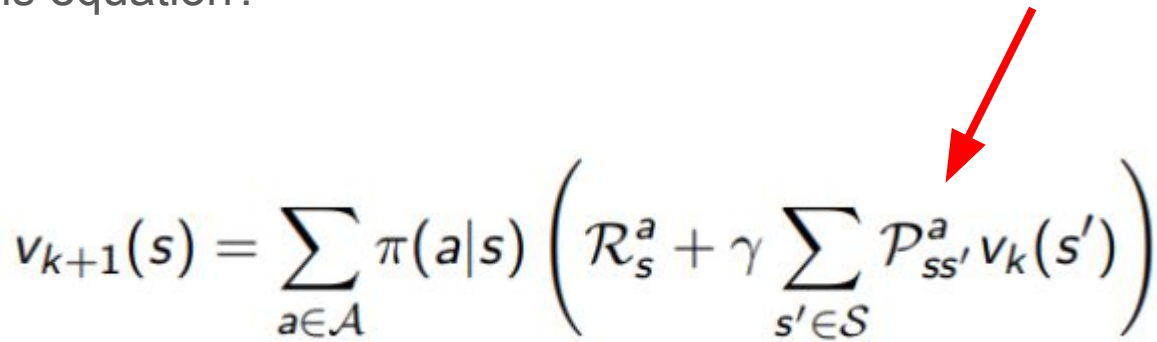
Remember this equation?

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

That’s right, it is the equation of Policy Evaluation. What is the problem when using this here with Monte-Carlo or Temporal-Difference?

Model-Free Control

Remember this equation?



$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

That’s right, it is the equation of Policy Evaluation. What is the problem when using this here with Monte-Carlo or Temporal-Difference?

Model-Free Control

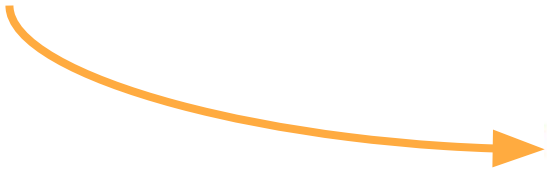
To solve this problem we can write the same equation but in in terms of Q instead of the Rewards R and the transition probabilities P.

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$$

Model-Free Control

To solve this problem we can write the same equation but in in terms of Q instead of the Rewards R and the transition probabilities P.

$$\pi'(s) = \underset{a \in \mathcal{A}}{\text{argmax}}\ \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$$



$$\pi'(s) = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s, a)$$

Model-Free Control

To solve this problem we can write the same equation but in in terms of Q instead of the Rewards R and the transition probabilities P.

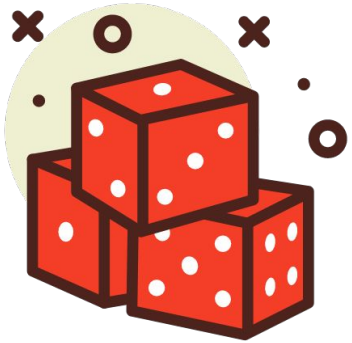
$$\pi'(s) = \underset{a \in \mathcal{A}}{\text{argmax}}\ \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$$

$$\pi'(s) = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s, a)$$

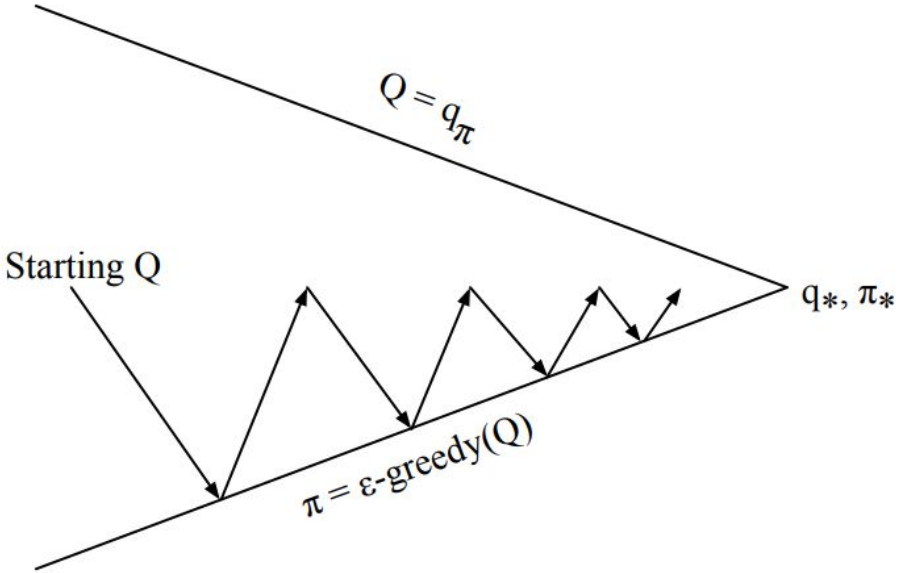
ε-Greedy Exploration

If we use the original Policy Iteration algorithm we would have another problem. Since now we don't have knowledge about the model we can't act greedy all the time.

- 1- ε = P(greedy action)
- ε = P(Random action)



Monte-Carlo Control



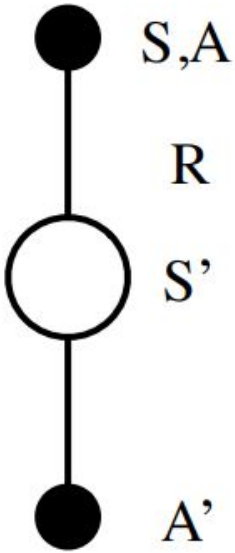
Every episode:

Policy evaluation Monte-Carlo policy evaluation, $Q \approx q_\pi$

Policy improvement ϵ-greedy policy improvement

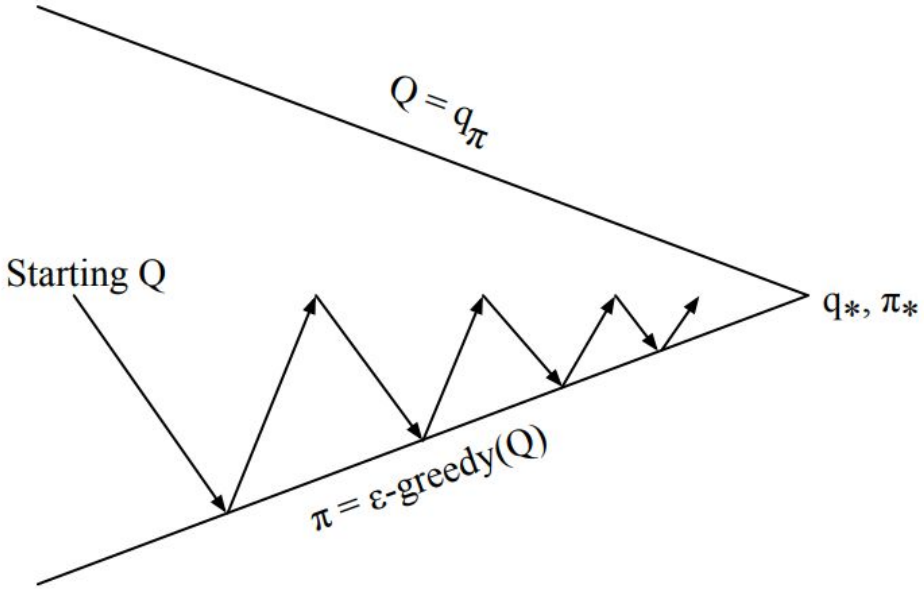
SARSA Algorithm (Control)

Since we prefer TD over MC, we are going to apply TD to Q(S,A) with **ε**-Greedy policy improvement and updating at each time step



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

SARSA Algorithm (Control)



Every time-step:

Policy evaluation Sarsa, $Q \approx q_\pi$

Policy improvement ϵ-greedy policy improvement

SARSA Algorithm (Control)

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

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Q-Learning Algorithm (Control)

- Next action (A_{t+1}) is chosen using behaviour policy μ but we also consider an alternative action A' sampled from the target policy π
- Update $Q(S_t,A_t)$ towards value of A'

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t) \right)$$

Popular Q-Learning

Special case: We let both policies improve

- Target policy $\boldsymbol{\pi}$ is greedy

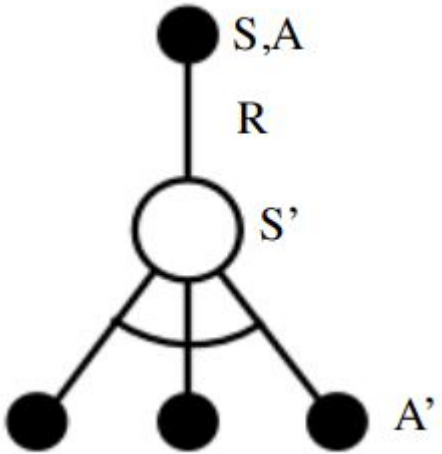
$$\pi(S_{t+1}) = \underset{a'}{\text{argmax}}\ Q(S_{t+1}, a')$$

- Behaviour policy $\boldsymbol{\mu}$ is $\boldsymbol{\varepsilon}$-greedy

$$\begin{aligned} & R_{t+1} + \gamma Q(S_{t+1}, A') \\ =& R_{t+1} + \gamma Q(S_{t+1}, \underset{a'}{\text{argmax}}\ Q(S_{t+1}, a')) \\ =& R_{t+1} + \max_{a'} \gamma Q(S_{t+1}, a') \end{aligned}$$

Q-Learning Algorithm (Control)

- Next action (A_{t+1}) is chosen using behaviour policy μ but we also consider an alternative action A' sampled from the target policy π
- Update $Q(S_t,A_t)$ towards value of A'



$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma \max_{a'} Q(S',a') - Q(S,A) \right)$$

Q-Learning Algorithm (Control)

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

Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Value Function Approximation

Problems so far

- So far we had a value V(s) for each s since it was a lookup table
- Estimating Value functions is really hard and computationally expensive, specially when the MDP is really large (speed and memory (states, actions) suffer).

Problems so far

- So far we had a value V(s) for each s since it was a lookup table
- Estimating Value functions is really hard and computationally expensive, specially when the MDP is really large (speed and memory (states, actions) suffer).

Solution

We use Value Function approximators!

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$
$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

This way we can generalize from seen states to unseen ones

Value Function Approximation in Model-Free Control

- We want to approximate q_π with some q(S,A,w)

$$\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$$

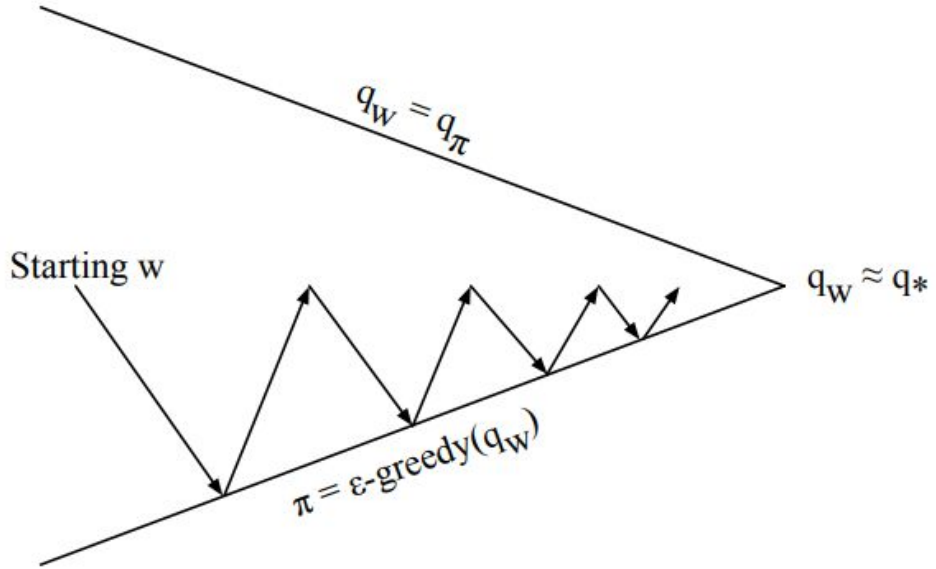
- Minimize MSE between approximation and real value

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2 \right]$$

- Since we like TD the w update is

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

Value Function Approximation in Model-Free Control

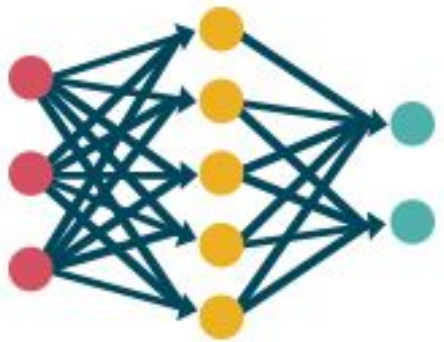


Policy evaluation Approximate policy evaluation, $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$

Policy improvement ϵ-greedy policy improvement

DQN

- Actions are taken according to an **ε**-Greedy policy
- Sample mini-batch of transitions $(S_t, A_t, R_{t+1}, S_{t+1})$
- Compute Q-Learning targets with fixed w
- Optimize MSE with Q-Learning targets and Q-network



$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i} \left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i) \right)^2 \right]$$

Policy Gradients

What if we parametrize the policy

- In value function aproximation we parametrized v(s;θ).

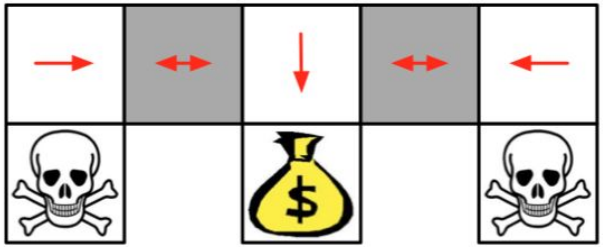
$$V_\theta(s) \approx V^\pi(s)$$
$$Q_\theta(s,a) \approx Q^\pi(s,a)$$

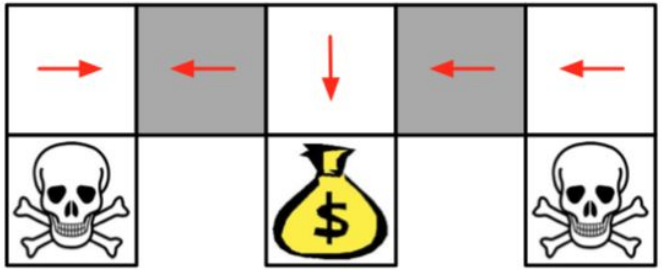
- And optimal policy was obtained from state-value function v(s).

- In Policy gradient we parametrize the policy.

$$\pi_\theta(s,a) = \mathbb{P}\left[a \mid s, \theta\right]$$

Stochastic Policy vs Deterministic Policy

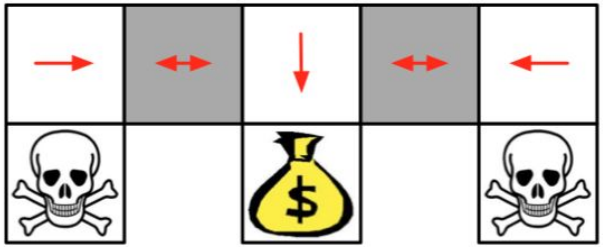


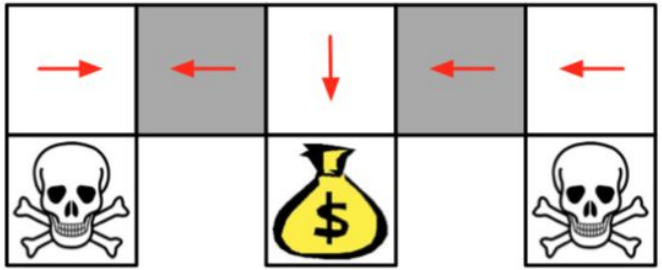


In gray states:

$$\pi_\theta(\text{wall to N and S, move E}) = 0.5$$
$$\pi_\theta(\text{wall to N and S, move W}) = 0.5$$

Stochastic Policy vs Deterministic Policy





In gray states:

$$\pi_\theta(\text{wall to N and S, move E}) = 0.5$$
$$\pi_\theta(\text{wall to N and S, move W}) = 0.5$$

In this **POMDP** stochastic policy beats deterministic one!

Deterministic optimal policy always exists in MDP [S,A,P,R,g] (Act greedly)

How optimize the policy?

- Define cost function
- In *start state environments:*

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1]$$

- In continuous environments (d(s) stationary distribution under policy):

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

- Or average reward per-time step

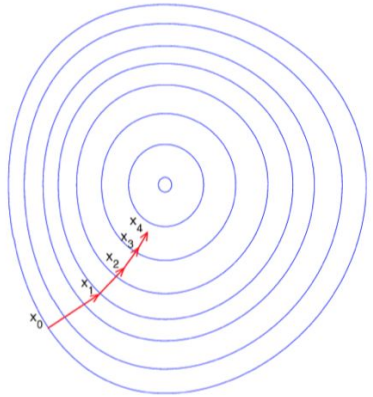
$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) \mathcal{R}_s^a$$

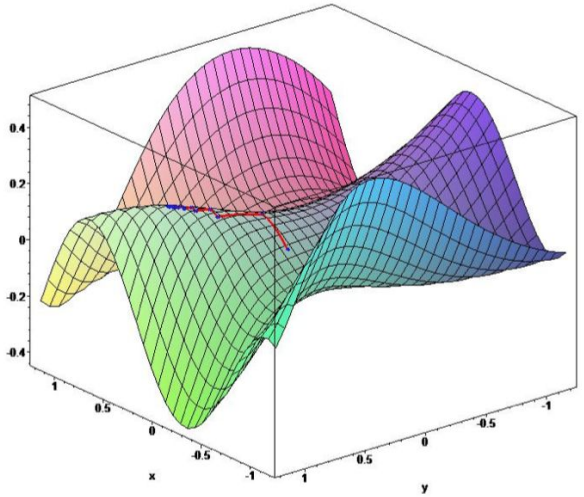
Optimizing the Policy:

- As usual gradient ascent to find optimal parameters.

$$\Delta\theta = \alpha \nabla_\theta J(\theta)$$

- α is LR.





What kind of policies?

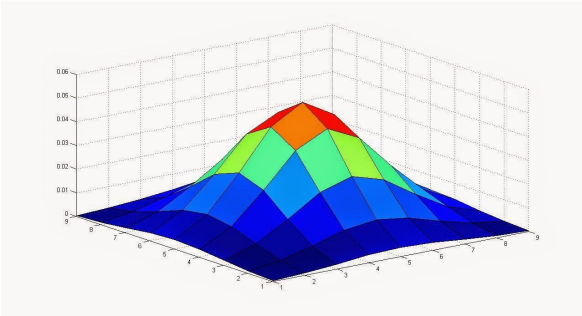
- Linear soft-max policy:
  - Linear combinations of features (s,a):

$$\pi_\theta(s,a) \propto e^{\phi(s,a)^\top \theta}$$

$$\nabla_\theta \log \pi_\theta(s,a) = \phi(s,a) - \mathbb{E}_{\pi_\theta}\left[\phi(s,\cdot)\right]$$

- Gaussian Policy:
  - Policy is gaussian: a~N($\mu(s), \sigma^2$)

$$\nabla_\theta \log \pi_\theta(s,a) = \frac{(a-\mu(s))\phi(s)}{\sigma^2}$$



Policy Gradient Theorem

Theorem

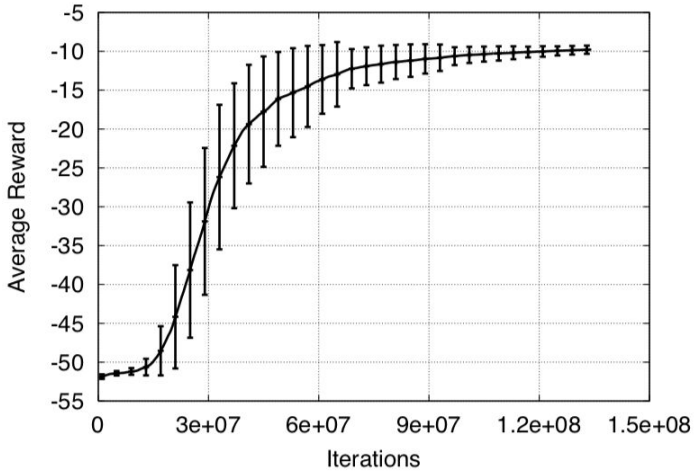
For a parametrized and differenciable policy $\pi_\theta(s,a)$ and for any policy objective function (first start, average…):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla \pi_\theta(s,a) Q^{\pi_\theta}(s,a)]$$

Monte-Carlo Policy Gradient

- v_t is the return (sum of the reward) a sample of $Q^{\pi_\theta}(s, a)$

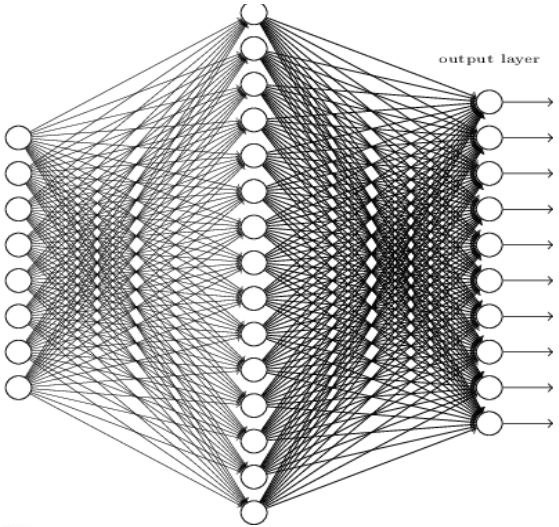
function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
  end for
  return θ
end function



Actor-Critic Policy Gradient

- What if we include action-value function (yep, Q-learning)? (Not using the return)
  - Critic estimate $Q_w(s,a) = Q^{\pi_\theta}(s,a)$

$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$$

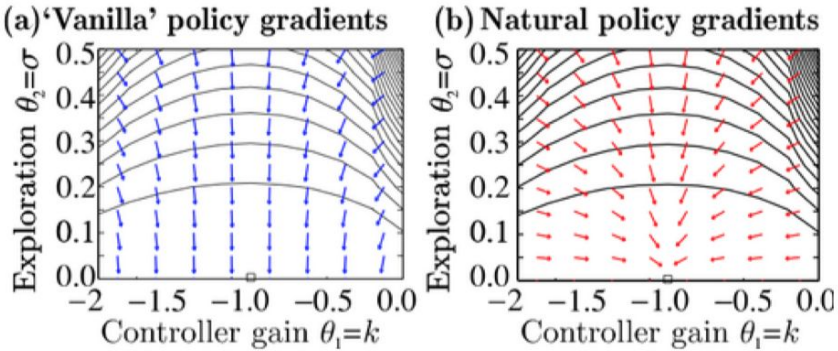


$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ Q_w(s,a)\right]$$
$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s,a)\ Q_w(s,a)$$

Actor-Critic Policy Gradient

- Update critic by linear temporal difference (TD) or monte-carlo
- Update actor parameters using policy gradients

function QAC
  Initialise s, θ
  Sample $a \sim \pi_\theta$
  for each step **do**
    Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
    Sample action $a' \sim \pi_\theta(s', a')$
    $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
    $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
    $w \leftarrow w + \beta \delta \phi(s, a)$
    $a \leftarrow a', s \leftarrow s'$
  end for
end function



Why policy gradients are cool?

- As policy is paremetrized π_θ both state-action sets could be **continous**!
- Parametrizing directly policy may require-less parameters.
- May deal with high variance

Paper 1

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**

Daan Wierstra **Martin Riedmiller**

DeepMind Technologies



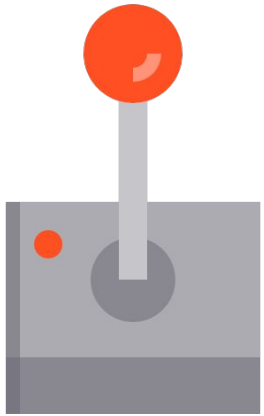

“Database”

Figure 1: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

- 210 × 160 RGB video at 60Hz
- > 50 games, only 7 are used in the paper

Key Points

- Finite action set A = {1,...,K} based on each game

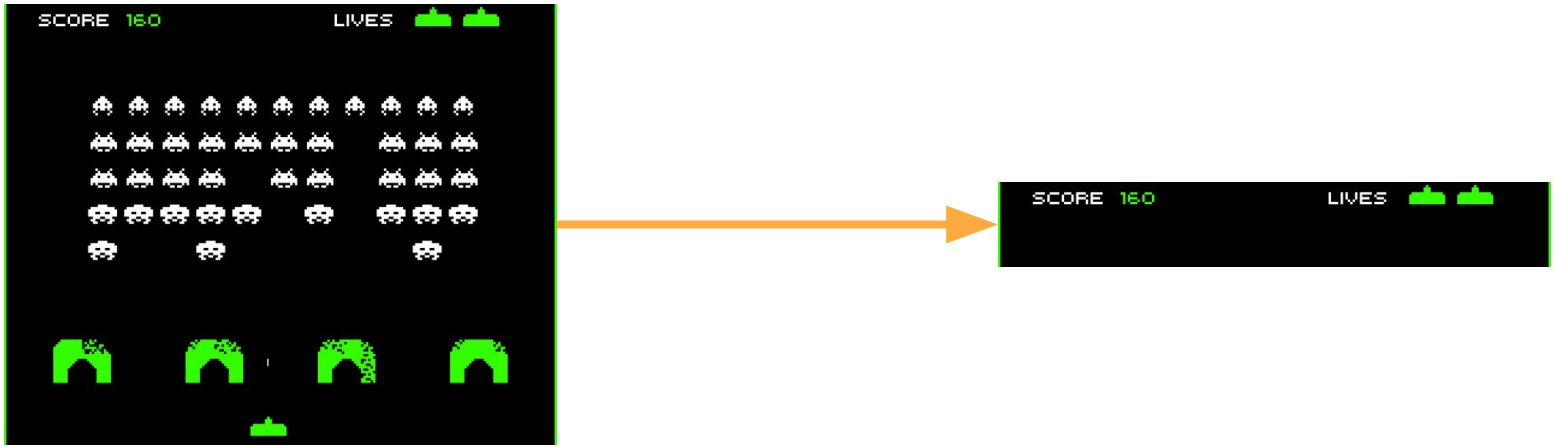


Key Points

- Finite action set A = {1,...,K} based on each game
- Model-free that uses screen images as the observable environment

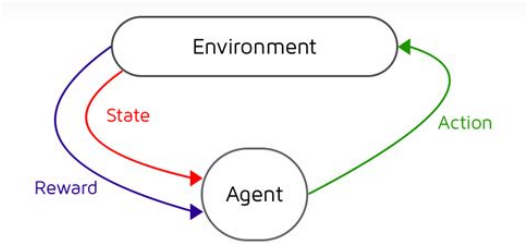
Key Points

- Finite action set A = {1,...,K} based on each game
- Model-free that uses screen images as the observable environment
- Score = Rewards (Discounted)



Key Points

- Finite action set A = {1,...,K} based on each game
- Model-free that uses screen images as the observable environment
- Score = Rewards (Discounted)
- 1 screen image x_t is too little information, therefore a sequence s is used ($s_t = x_1, a_1, x_2, ..., a_{t-1}, x_t$)
- Since the sequence s is finite, the problem can be solved like a very large but finite MDP



Optimization

- We could in theory use the Bellman Equation as an iterative update, right?

Optimization

- We could in theory use the Bellman Equation as an iterative update, right?
  - Remember computation cost!

Optimization

- We could in theory use the Bellman Equation as an iterative update, right?
  - Remember computation cost!
- What do we do? Value Function Approximators!

Optimization

- We could in theory use the Bellman Equation as an iterative update, right?
  - Remember computation cost!
- What do we do? Value Function Approximators!
- Neural Network Approximator with weights $\boldsymbol{\theta}$ = Q-Network

Optimization

- We could in theory use the Bellman Equation as an iterative update, right?
  - Remember computation cost!
- What do we do? Value Function Approximators!
- Neural Network Approximator with weights $\boldsymbol{\theta}$ = Q-Network

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

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

- $\boldsymbol{\rho}$ is a behaviour policy whereas $\boldsymbol{\varepsilon}$ is the learning policy, $\boldsymbol{\varepsilon}$-greedy and greedy respectively

Experience Replay

- Create a dataset of experience D that saves the agent's experiences at each time step $e_t = (s_t, a_t, r_t, s_{t+1})$
- This "history" of experiences D is then used to sample mini-batches for the Q-Learning updates.
- Fixed length of experiences thanks to a function $\boldsymbol{\phi}$

DQN Algorithm

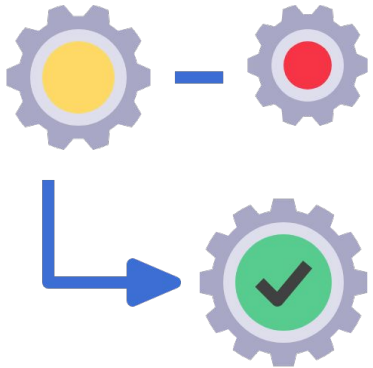
Algorithm 1 Deep Q-learning with Experience Replay

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

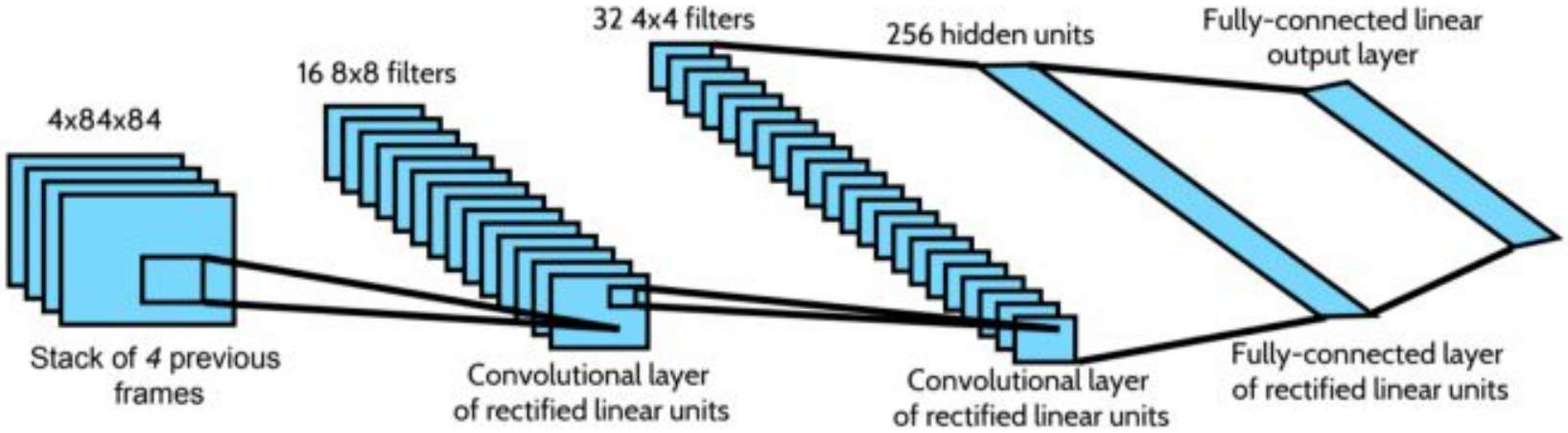
$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right]$$

Preprocessing

- Raw Atari frames (210x160 with 128 color palette)
- Grayscale and 110x84
- ϕ function preprocesses the last 4 frames of the history



Architecture



Experiments

Figure 3: The leftmost plot shows the predicted value function for a 30 frame segment of the game Seaquest. The three screenshots correspond to the frames labeled by A, B, and C respectively.

Experiments

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

Paper 2

CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING

Timothy P. Lillicrap,* Jonathan J. Hunt,* Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver & Daan Wierstra
Google Deepmind
London, UK

DDPG

- This work deals with deep-deterministic policy gradients (DDPG).
  - Any Idea?

DDPG

- This work deals with deep-deterministic policy gradients (DDPG). Idea?
- -> Parametrize using NN

Actor

Critic

$\pi_\theta(s, a) \propto$

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i>t} \sim E, a_{i>t} \sim \pi}\left[R_t | s_t, a_t\right]$$

Recursive bellman eq.

- Under deterministic Policy μ:

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}\left[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1}))\right]$$

Recursive bellman eq.

- Under deterministic Policy μ:

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}\left[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1}))\right]$$

- Off-policy or On-policy?
  - As Q is under deterministic policy we use off-policy.

Actor-critic

- Critic-loss

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E} \left[\left(Q(s_t, a_t | \theta^Q) - y_t \right)^2 \right]$$

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1}) | \theta^Q).$$

- Actor-loss

$$\begin{aligned} \nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_{\theta^\mu} Q(s, a | \theta^Q) |_{s=s_t, a=\mu(s_t|\theta^\mu)} \right] \\ &= \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_a Q(s, a | \theta^Q) |_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s | \theta^\mu) |_{s=s_t} \right] \end{aligned}$$

Heuristics

- Both actor critic networks are updated every certain time τ.
    - Seems to stabalize learning.

DDPG

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

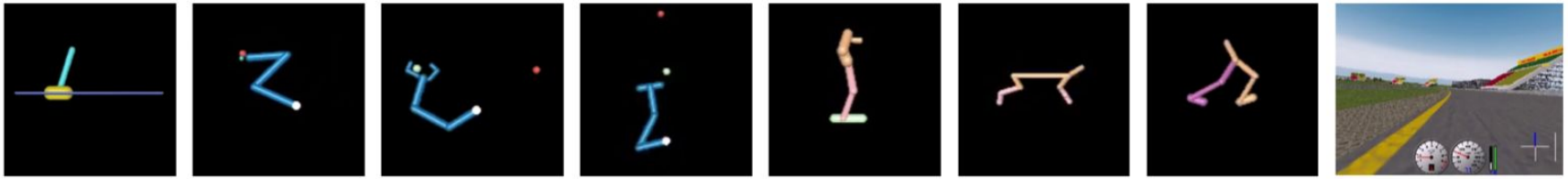
$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

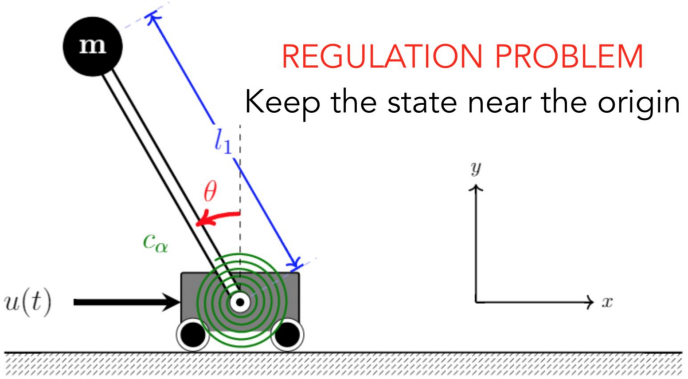
‘Database’



- Continuous actions-state problems.
- Cart, pendulum…

Key-points

- DPG solves action-state continuous spaces.



- DDPG approximates critic and actor using NN.
    - This approach combine the best of Q-learning and Policy gradients.

Results

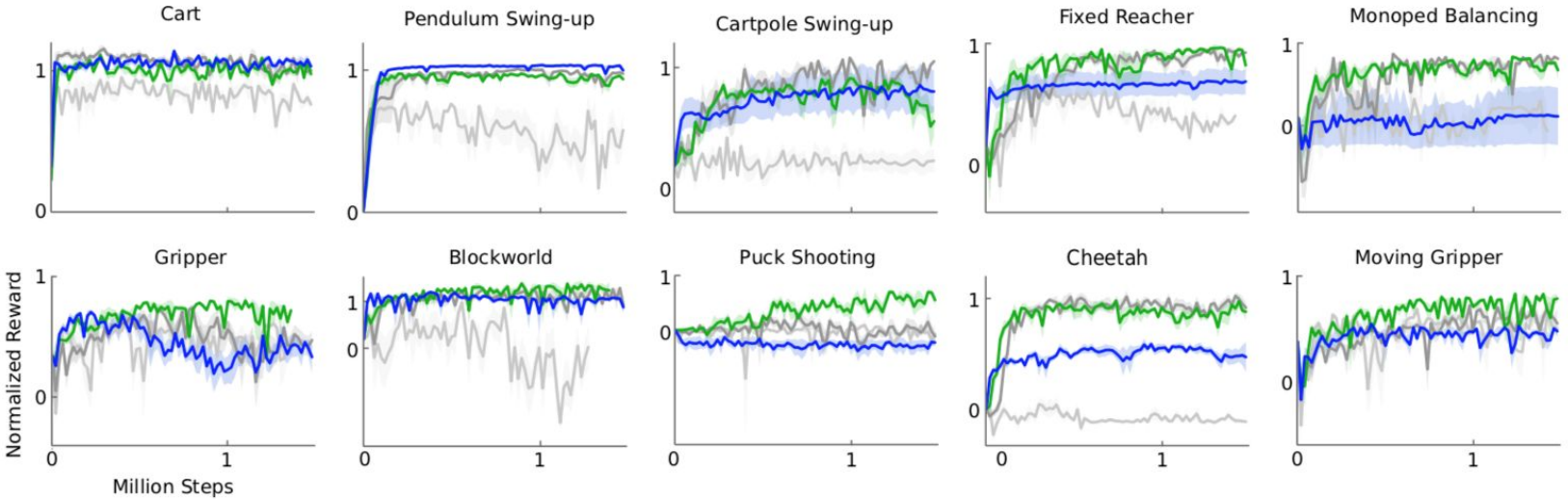


Figure 2: Performance curves for a selection of domains using variants of DPG: original DPG algorithm (minibatch NFQCA) with batch normalization (light grey), with target network (dark grey), with target networks and batch normalization (green), with target networks from pixel-only inputs (blue). Target networks are crucial.

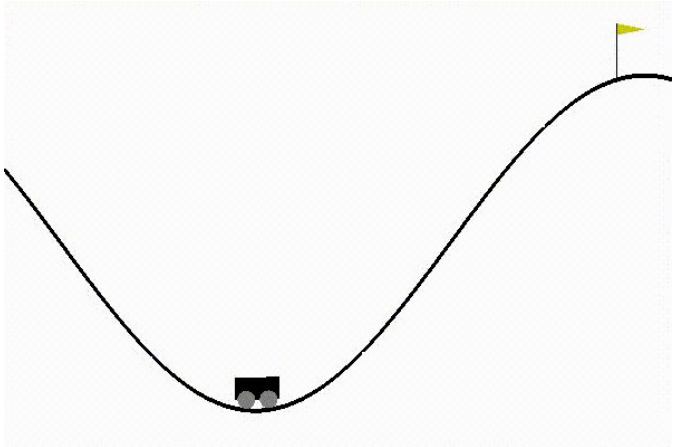
Architecture and Hyperparameters

- Actor an critic are both 2 layer network with 300 and 400 hidden states.
- Hyper-parameters in tutorial

DQN Tutorial

Problem

- Two actions: Forward and backward
- Objective: Get the car to the goal
- Input to the agent: Car position and velocity
- Most important imports *gym, matplotlib,* and *torch*



Hyperparameters

- Epsilon (exploration rate)
- Gamma (reward decay rate)
- LR
- Memory_Capacity (Number of sequences stored)
  sequence = State, Action, Reward and Next State
- Q_Network_Iteration (Number of iteration between target and eval policies)
- Batch_Size (Amount of sequences sampled from memory to learn)

```
#hyper parameters
EPSILON = 0.9
GAMMA = 0.9
LR = 0.01
MEMORY_CAPACITY = 2000
Q_NETWORK_ITERATION = 100
BATCH_SIZE = 32
```

Loading the Environment

```
EPISODES = 400
env = gym.make('MountainCar-v0')
env = env.unwrapped
NUM_STATES = env.observation_space.shape[0] # 2
NUM_ACTIONS = env.action_space.n
```

- Set the amount of episodes
- Load the MountainCar environment
- Get amount of states and actions in this problem

Neural Network

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.fc1 = nn.Linear(NUM_STATES, 30)
        self.fc1.weight.data.normal_(0, 0.1)
        self.fc2 = nn.Linear(30, NUM_ACTIONS)
        self.fc2.weight.data.normal_(0, 0.1)


    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)

        return x
```

Main

1. Iterate over the amount of episodes defined
2. For each time step get the action based on the state and then get the next state and reward
3. Store the sequence (State, Action, Reward, Next State)
4. If the Memory counter overpasses the Memory capacity **learn**

```python
def main():
    net = Dqn()
    print("The DQN is collecting experience...")
    step_counter_list = []
    for episode in range(EPISODES):
        state = env.reset()
        step_counter = 0
        while True:
            step_counter +=1
            env.render()
            action = net.choose_action(state)
            next_state, reward, done, info = env.step(action)
            reward = reward * 100 if reward >0 else reward * 5
            net.store_trans(state, action, reward, next_state)

            if net.memory_counter >= MEMORY_CAPACITY:
                net.learn()
                if done:
                    print(f"episode {episode}, the reward is {round(reward, 3)}")
            if done:
                step_counter_list.append(step_counter)
                net.plot(net.ax, step_counter_list)
                break

            state = next_state
```

DQN Part 1

```python
class Dqn():
    def __init__(self):
        self.eval_net, self.target_net = Net(), Net()
        self.memory = np.zeros((MEMORY_CAPACITY, NUM_STATES *2 +2))
        # state, action ,reward and next state
        self.memory_counter = 0
        self.learn_counter = 0
        self.optimizer = optim.Adam(self.eval_net.parameters(), LR)
        self.loss = nn.MSELoss()

        self.fig, self.ax = plt.subplots()

    def store_trans(self, state, action, reward, next_state):
        if self.memory_counter % 500 ==0:
            print(f"The experience pool collects {self.memory_counter} time experience")
        index = self.memory_counter % MEMORY_CAPACITY
        trans = np.hstack((state, [action], [reward], next_state))
        self.memory[index,] = trans
        self.memory_counter += 1

    def choose_action(self, state):
        # notation that the function return the action's index nor the real action
        # EPSILON
        state = torch.unsqueeze(torch.FloatTensor(state) ,0)
        if np.random.randn() <= EPSILON:
            action_value = self.eval_net.forward(state)
            action = torch.max(action_value, 1)[1].data.numpy() # get action whose q is max
            action = action[0] #get the action index
        else:
            action = np.random.randint(0,NUM_ACTIONS)
        return action

    def plot(self, ax, x):
        ax.cla()
        ax.set_xlabel("episode")
        ax.set_ylabel("total reward")
        ax.plot(x, 'b-')
        plt.pause(0.000000000000001)
```

- Init: Init Memory space, optimizer, loss, counters and both networks (Target and Eval)
- Store_trans: Saves into the Memory a sequence of (State, Action, Reward, Next State)
- Choose_action: Picks an action based on an ε-greedy policy
- Plot: Plot Total Reward vs Episode

DQN Part 2 (Learning)

```
def learn(self):
    # learn 100 times then the target network update
    if self.learn_counter % Q_NETWORK_ITERATION ==0:
        self.target_net.load_state_dict(self.eval_net.state_dict())
    self.learn_counter+=1

    sample_index = np.random.choice(MEMORY_CAPACITY, BATCH_SIZE)
    batch_memory = self.memory[sample_index, :]
    batch_state = torch.FloatTensor(batch_memory[:, :NUM_STATES])
    #note that the action must be a int
    batch_action = torch.LongTensor(batch_memory[:, NUM_STATES:NUM_STATES+1].astype(int))
    batch_reward = torch.FloatTensor(batch_memory[:, NUM_STATES+1: NUM_STATES+2])
    batch_next_state = torch.FloatTensor(batch_memory[:, -NUM_STATES:])

    q_eval = self.eval_net(batch_state).gather(1, batch_action)
    q_next = self.target_net(batch_next_state).detach()
    q_target = batch_reward + GAMMA*q_next.max(1)[0].view(BATCH_SIZE, 1)

    loss = self.loss(q_eval, q_target)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

- QNETWORKITER
- Sample batches of state, action, reward and next state (sample sequences)
- Q value of initial state based
- Q value of next state based on target policy
- Q value for the update
- Backpropagation

DQN Homework

1. Play with the hyperparameters and show their corresponding graphs. Which parameter caused the most change? Which one didn't affect that much? Discuss briefly your results
2. Anneal the ε hyperparameter to decay linearly instead of being fixed. Did it help at all? Why?
3. Try two different architectures and report any results

DDPG Tutorial

Problem

- Continuous action space and state space
- Objective: swing up problem (Classic control problem).
- State-space: Tip-position and tip-velocity.
- Action-space: Applied torque (-2 , 2)
- Reward: - cost `angle_normalize(theta)**2 + .1*theta_dot**2 + .001*(u**2)`
- Reward penalize both actions and control law.
- Pendulum starts in random position.

Critic and actor networks

- Actor maps from state_dim to action_dim (actor output action)
- Critic estimate action-value function.

```python
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(Actor, self).__init__()

        self.l1 = nn.Linear(state_dim, 400)
        self.l2 = nn.Linear(400, 300)
        self.l3 = nn.Linear(300, action_dim)

        self.max_action = max_action

    def forward(self, x):
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = self.max_action * torch.tanh(self.l3(x))
        return x


class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()

        self.l1 = nn.Linear(state_dim + action_dim, 400)
        self.l2 = nn.Linear(400 , 300)
        self.l3 = nn.Linear(300, 1)

    def forward(self, x, u):
        x = F.relu(self.l1(torch.cat([x, u], 1)))
        x = F.relu(self.l2(x))
        x = self.l3(x)
        return x
```

DDPG

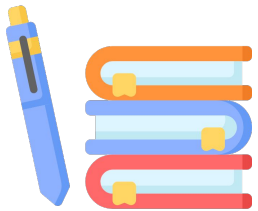
- Initialize critic and actor networks.
- Copy actor and critic to target networks (heuristic!)
- Replay_buffer (save things and sample)

```python
class DDPG(object):
    def __init__(self, state_dim, action_dim, max_action):
        self.actor = Actor(state_dim, action_dim, max_action).to(device)
        self.actor_target = Actor(state_dim, action_dim, max_action).to(device)
        self.actor_target.load_state_dict(self.actor.state_dict())
        self.actor_optimizer = optim.Adam(self.actor.parameters(), args.learning_rate)

        self.critic = Critic(state_dim, action_dim).to(device)
        self.critic_target = Critic(state_dim, action_dim).to(device)
        self.critic_target.load_state_dict(self.critic.state_dict())
        self.critic_optimizer = optim.Adam(self.critic.parameters(), args.learning_rate)
        self.replay_buffer = Replay_buffer()
        self.writer = SummaryWriter(directory)
        self.num_critic_update_iteration = 0
        self.num_actor_update_iteration = 0
        self.num_training = 0

    def select_action(self, state):
        state = torch.FloatTensor(state.reshape(1, -1)).to(device)
        return self.actor(state).cpu().data.numpy().flatten()

    def update(self):

        for it in range(args.update_iteration):
            # Sample replay buffer
            x, y, u, r, d = self.replay_buffer.sample(args.batch_size)
            state = torch.FloatTensor(x).to(device)
            action = torch.FloatTensor(u).to(device)
            next_state = torch.FloatTensor(y).to(device)
            done = torch.FloatTensor(d).to(device)
            reward = torch.FloatTensor(r).to(device)

            # Compute the target Q value
            target_Q = self.critic_target(next_state, self.actor_target(next_state))
            target_Q = reward + ((1 - done) * args.gamma * target_Q).detach()

            # Get current Q estimate
            current_Q = self.critic(state, action)

            # Compute critic loss
            critic_loss = F.mse_loss(current_Q, target_Q)
            self.writer.add_scalar('Loss/critic_loss', critic_loss, global_step=self.num_critic_update_iteration)
            # Optimize the critic
            self.critic_optimizer.zero_grad()
            critic_loss.backward()
            self.critic_optimizer.step()

            # Compute actor loss
            actor_loss = -self.critic(state, self.actor(state)).mean()
            self.writer.add_scalar('Loss/actor_loss', actor_loss, global_step=self.num_actor_update_iteration)

            # Optimize the actor
            self.actor_optimizer.zero_grad()
            actor_loss.backward()
            self.actor_optimizer.step()

            # Update the frozen target models
            for param, target_param in zip(self.critic.parameters(), self.critic_target.parameters()):
                target_param.data.copy_(args.tau * param.data + (1 - args.tau) * target_param.data)

            for param, target_param in zip(self.actor.parameters(), self.actor_target.parameters()):
                target_param.data.copy_(args.tau * param.data + (1 - args.tau) * target_param.data)

            self.num_actor_update_iteration += 1
            self.num_critic_update_iteration += 1
```

DDPG Homework

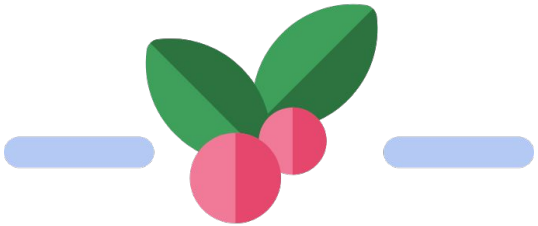
1. Change DDPG to Mountain car, (May tune a bit the hyperparameters as constant time systems are different). Compare with DQN as the environment is the same.

2. (Optional) As you see reward/cost penalize control law/actions change it so it penalize more control energy used and plot u(t) (actions in time) for different initial positions of the pendulum.



References

- Sutton, R. S., Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press. Available at: http://incompleteideas.net/book/bookdraft2017nov5.pdf (BIBLE)
-
- D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic Policy Gradient Algorithms,” Proc. 31st Int. Conf. Mach. Learn., pp. 387–395, 2014. (DPG)
- V. Mnih et al., “Playing Atari with Deep Reinforcement Learning,” arXiv, pp. 1–9, 2013. (DQN)
- T. P. Lillicrap et al., “Continuous control with deep reinforcement learning,” arXiv, 2015. (DDPG)



THANK YOU