Reinforcement Learning

Patryk Neubauer
Gradient Science Club 2022

Plan for Today

- Fundamentals of Reinforcement Learning
  - Elements of RL - high level look
  - Theory behind learning
  - Dynamic Programming - Value/Policy Iteration
- Model-free methods
  - Temporal-Difference Learning (SARSA, Q-learning)
  - Policy Gradient
  - Actor-Critic
- Note on model-based approaches
- Examples

Resources

A lot of hype -> a lot of good resources

- MIT 6.S091: Introduction to Deep RL (Lex Fridman)
- (Ongoing) HuggingFace Deep RL course
- OpenAI Spinning Up
- Deep Dive:
  - Reinforcement Learning: An Introduction (Sutton & Barto book)
  - RL Lecture Series (DeepMind x University College London)
  - Reinforcement Learning Specialization (University of Alberta)

Reinforcement Learning Background

Reinforcement Learning isn't just machine learning

- Neuroscience and psychology:
  - Studying learning and decision-making in animals and humans
  - Used to develop theories of addiction
  - Resemblance to dopamine
- Study of strategic decision-making in Game Theory and Economics
- Works on theory of Optimal Control from mathematics and Dynamic Programming

Fundamentals of Reinforcement Learning

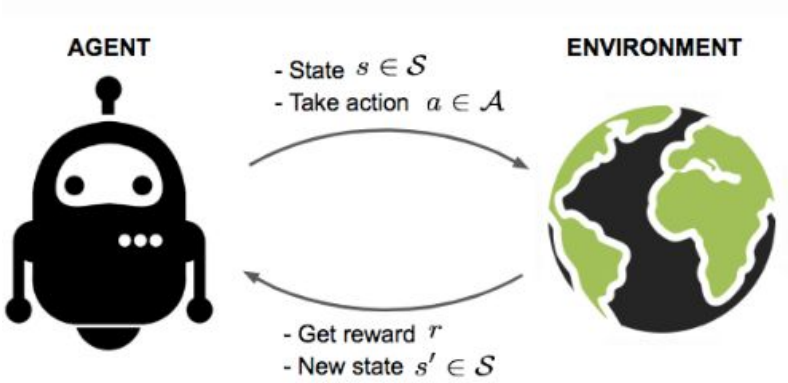
Elements of RL - high level look

The RL Framework

Reinforcement Learning models every problem in this way.

An **agent** interacts with the **environment** through **actions** that he takes based on the **state**, for which he gains **rewards**.

Agents goal is to maximize the **cumulative** reward.

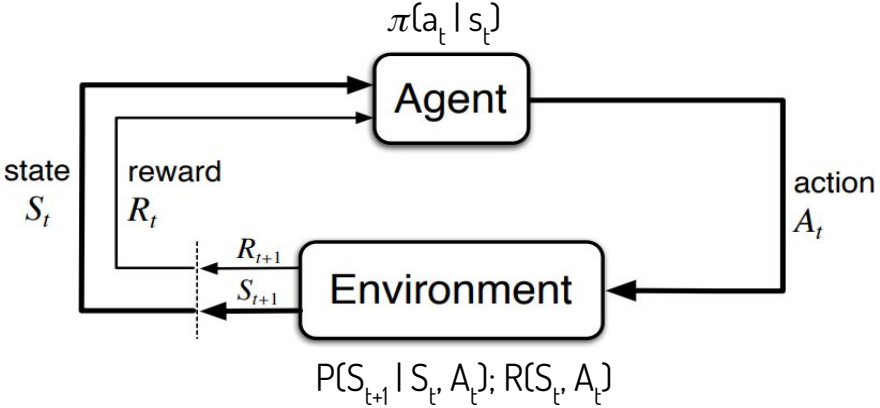


Sequence of $(S_n, A_n, R_{n+1}, ..., A_{t-1}, S_t, R_t)$ is called a **trajectory**.

The RL Framework

At time step t:

- St - current state of the environment
- Rt - reward given by the environment for the previous action
- At - action taken by the agent based on St $\pi(a_t \mid s_t)$ (according to his **policy)**
- St+1 and Rt+1 are the next state and reward given by the environment, affected by At **$P(S_{t+1} \mid S_t, A_t); R(S_t, A_t)$**

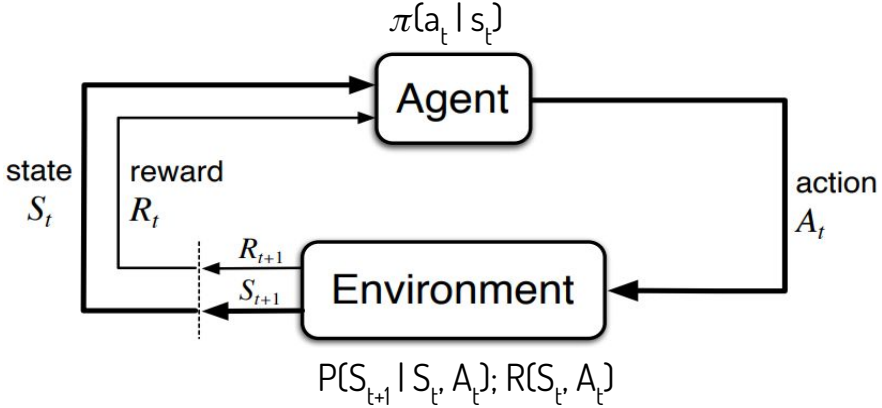


http://incompleteideas.net/book/RLbook2020.pdf

The RL Framework

Alternative notation:

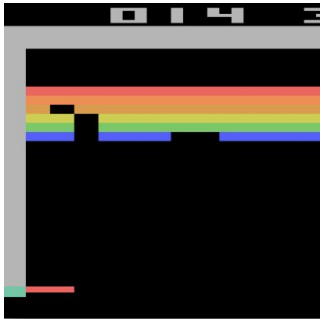
- $S_t = S$
- $A_t = A$
- $R_t = R$
- $S_{t+1} = S'$
- $R_{t+1} = R'$



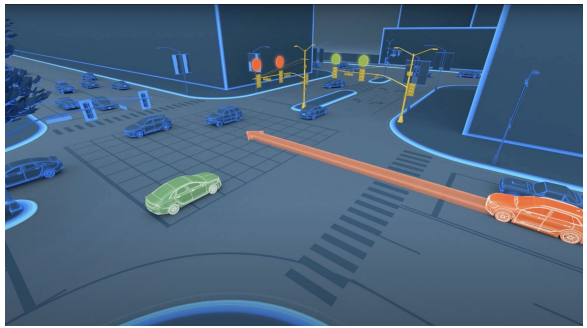
http://incompleteideas.net/book/RLbook2020.pdf

Type of tasks

- Episodic - there's an ending point - **terminal state** - after which an **episode** ends.

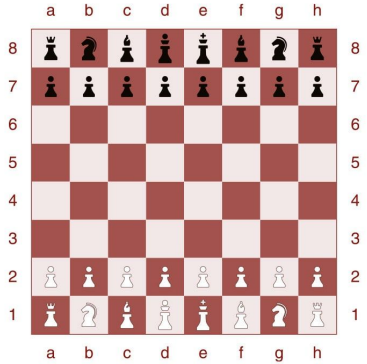


- Continuous - tasks that continue forever - without a terminal state



State Space

- State - description of the environment - information available to the agent, based on which it needs to pick an action.
- It can be basically anything:
  - Board state in a chess game
  - Screen of a video game
  - Camera feed from a robot
  - ...



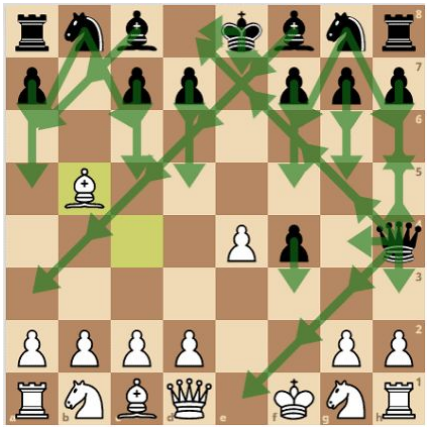
State - complete information

Observation - only some information is available

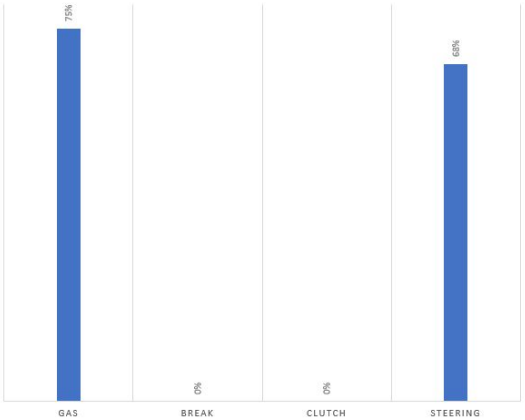
- **State represents the complete information about the environment** (no hidden information), while **observation is a partial description of the world** - often used interchangeably

Action Space

- Action Space / Actions are all of the possible things that agent can do in an environment
- Like with state, it can be basically anything:
  - (Legal) chess moves
  - Keyboard/mouse/Gamepad inputs
  - Steering wheel/gas/break control
  - ...



herculeschess.com



State/Action Space - Continuous vs. Discrete

- Both state and action spaces can be either discrete or continuous.
- If **space is continuous (or discrete, but huge)** we're forced to use **function approximators** (e.g. neural networks) that **approximate the value** function/policy.
- Action space being discrete or continuous, will determine which algorithms can be used
- Examples:
  - Discrete state - Discrete actions -> Tic-Tac-Toe
  - Continuous state - Discrete actions -> Tetris
  - Discrete state - Continuous actions -> Blackjack
  - Continuous state - Continuous actions -> Driving

Algorithm	Action Space	State Space
Monte Carlo	Discrete	Discrete
Q-learning	Discrete	Discrete
SARSA	Discrete	Discrete
Q-learning - Lambda	Discrete	Discrete
SARSA - Lambda	Discrete	Discrete
DQN	Discrete	Continuous
DDPG	Continuous	Continuous
A3C	Continuous	Continuous
NAF	Continuous	Continuous
TRPO	Continuous	Continuous
PPO	Continuous	Continuous
TD3	Continuous	Continuous
SAC	Continuous	Continuous

Rewards

- Reward is the **only** feedback the agent gets, it tells whether the taken action was good or bad
- Reward engineering is an important part of reinforcement learning in practice
- Simplest approach can be the best e.g.
  - +1 for winning, -1 for losing in episodic games
  - +1 for each second a car stays on course
- … but often it's not that simple.
- If the reward function is too easy or overengineered, it can be "hacked" and abused.
- Sometimes it's hard to say what the reward should even be:
  - Autonomous driving
  - Controlling a robot arm
  - …

DeepMind – Specification gaming: the flip side of AI ingenuity

Rewards

Balancing Etiquette and Nerve

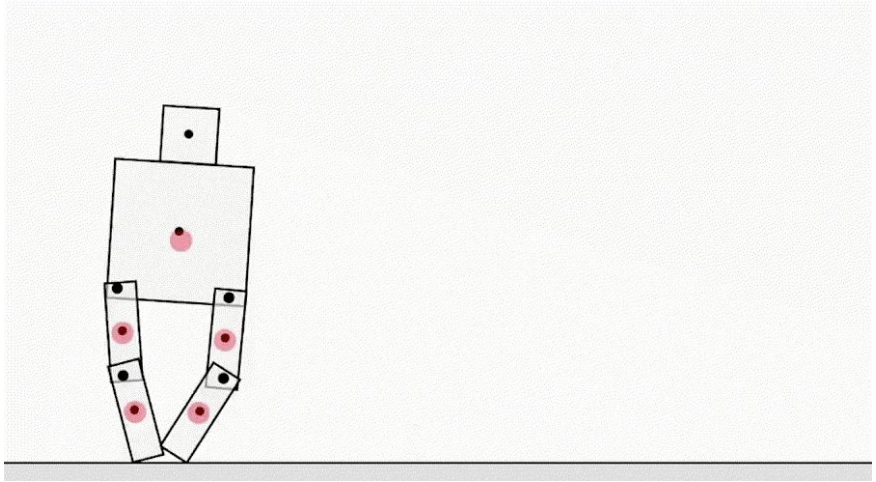
An aggressive agent

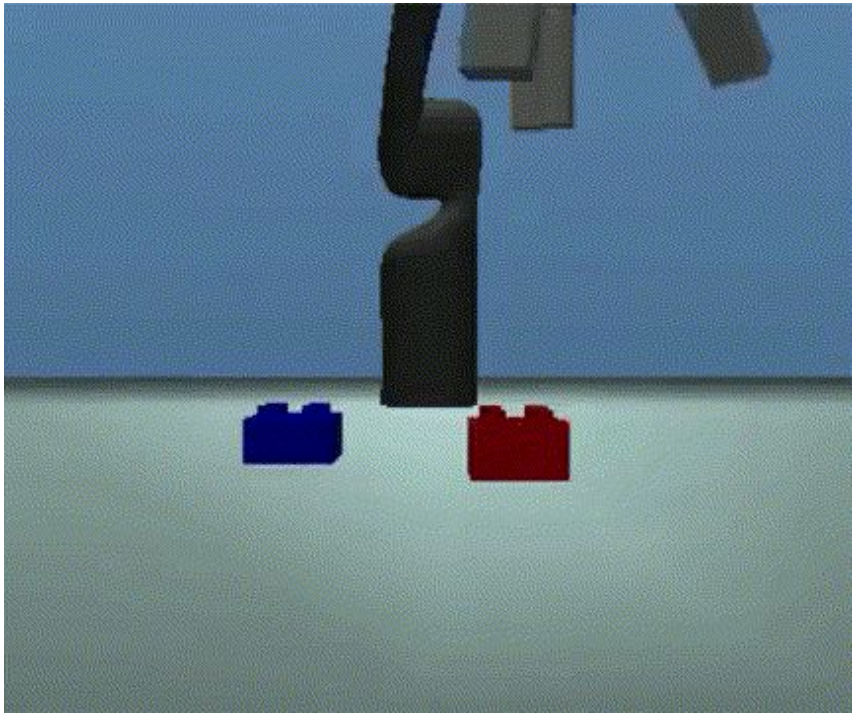
A timid agent

- We controlled this etiquette balance through a series of collision penalty terms
  - **Any collision** – baseline “no fault” penalty
  - **Rear-end collision** – added to the trailing car based on the velocity difference.
  - **Unsporting collision** – constant penalty assessed on sideswipes, rear-ends, and curve collisions that weren’t clearly the opponent’s fault.

Outracing Champion Gran Turismo Drivers with Deep Reinforcement Learning

Rewards





DeepMind - Specification gaming: the flip side of AI ingenuity

Rewards - returns

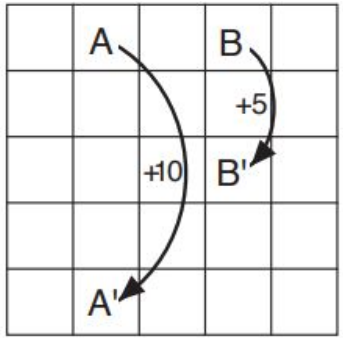
- **Return** is the **cumulative reward** that an agents gets over a trajectory - either in an episode or an finite amount of time in a continuous task.
- **Discounted return** is a variant of the return that takes into account the time value of reward.
- **Expected (discounted) return** is the expected value of the return over all possible trajectories, taking into account the probability of each trajectory occurring.

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$$,$\gamma \in [0, 1]$; if $\gamma = 1$ the return isn't discounted

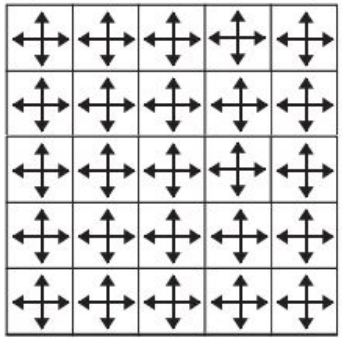
often also represented by $\mathbf{G_t}$

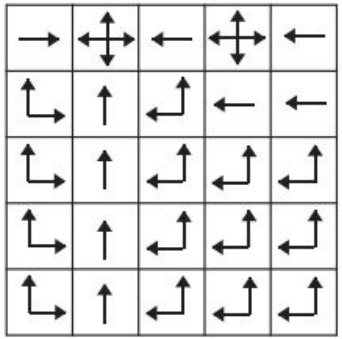
Policies

- **Policy** is the function of agent's behaviour - $\pi(a_t \mid s_t)$.
  - It denotes the **probability** of action a_t in state s_t.
  - Can be stochastic or deterministic.
- **Optimal policy** π_* is a policy that maximizes **expected return** - learning it is the goal of RL.



Grid world example


Random policy


Optimal policy

Value functions

- **State value functions** are used to evaluate the **quality of states**. They represent the expected return that can be achieved by starting from a particular state and **following a particular policy**. The state value function is denoted as **V(s)**, and it maps **states to expected returns**.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S},$$

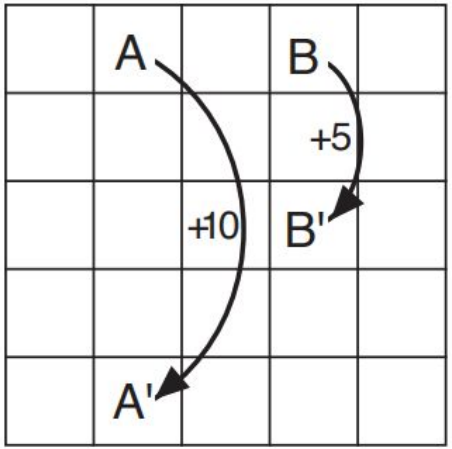
- **Action value functions** are used to evaluate the **quality of actions**. They represent the expected return that can be achieved by taking a particular action in a particular state and **following a particular policy**. The action value function is denoted as **Q(s, a)**, and it maps **state-action pairs to expected returns**.

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

$$V(s) = \max_a Q(s,a)$$

Value functions

- For now we'll assume that everything is in tabular form.



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

State values of a random policy.

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

State values of a optimal policy

*-1 for moving off the grid

Value functions

- For now we'll assume that everything is in tabular form.

	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
s_0	490	490	490	700	490	490	490	490
s_1	490	490	700	490	490	490	490	490
s_2	490	490	490	700	700	490	490	490
s_3	490	490	700	1000	700	700	490	490
s_4	490	490	490	700	700	700	490	490
s_5	490	490	700	700	1000	700	700	490
s_6	490	490	490	490	700	700	490	490
s_7	490	490	490	490	700	1000	700	490
s_8	700	490	490	490	700	700	1000	700
s_9	700	490	490	490	490	700	700	1000
s_{10}	700	490	490	490	490	490	490	490
s_{11}	490	490	490	490	490	700	490	490
s_{12}	490	490	490	490	490	700	700	490
s_{13}	490	490	490	490	490	700	700	700
s_{14}	490	490	490	490	490	490	700	700
s_{15}	490	490	490	490	490	490	490	700

Gradient Descent Methods in Learning Classifier Systems: Improving XCS Performance in Multistep Problems.

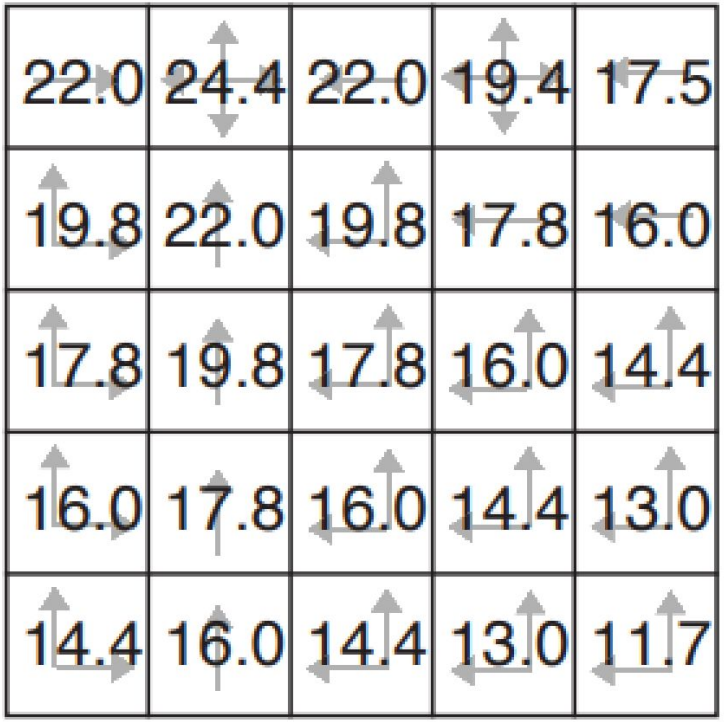
On Optimal Policies and Optimal Value Functions

- To recap:
  - Policies define which actions agent should pick in different states.
  - Value functions represent the expected return *under* a policy of states / state-action pairs.
- We can compare policies with value functions

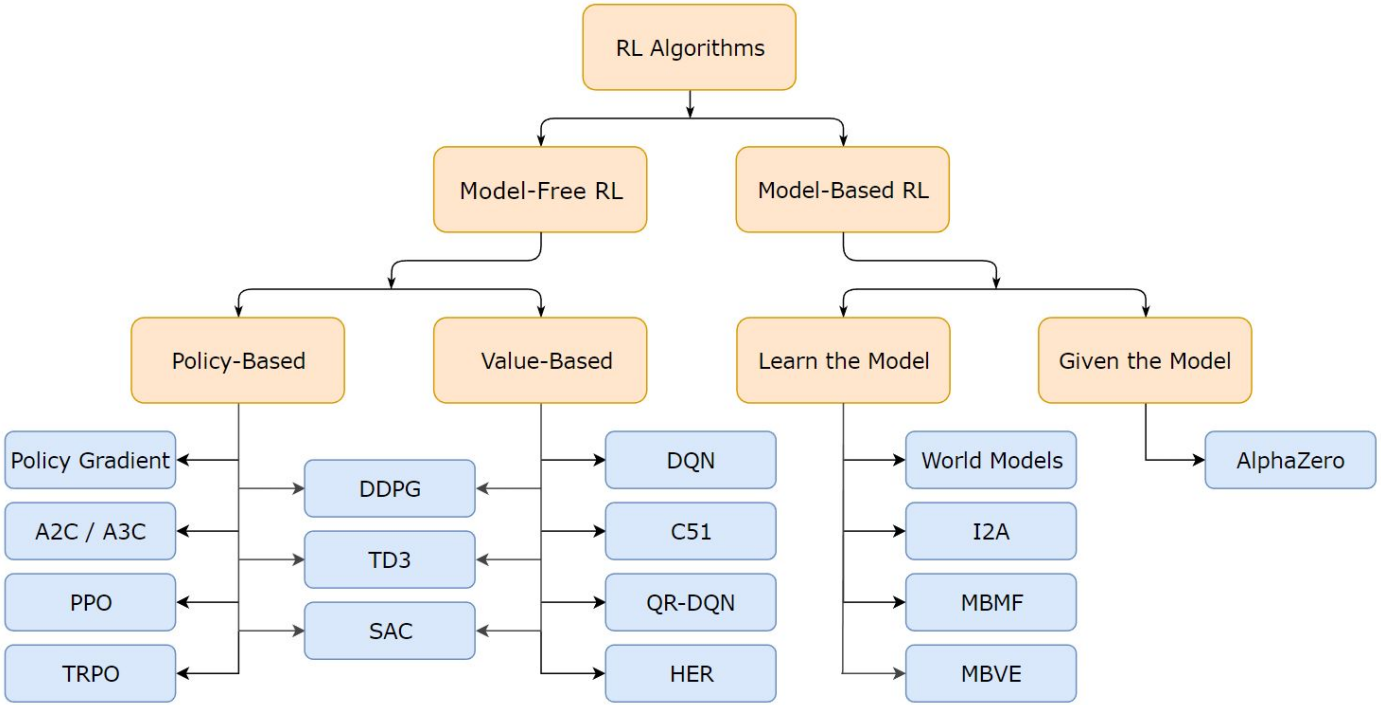
$$\pi >= \pi', \text{ if and only if } v_{\pi}(s) >= v_{\pi'}(s) \text{ for all states}$$

- Optimal policies π_* share the same optimal value functions $v_*(s)$ and $q_*(s,a)$
- **Optimal policy π^* can be determined by using value functions**.

On Optimal Policies and Optimal Value Functions

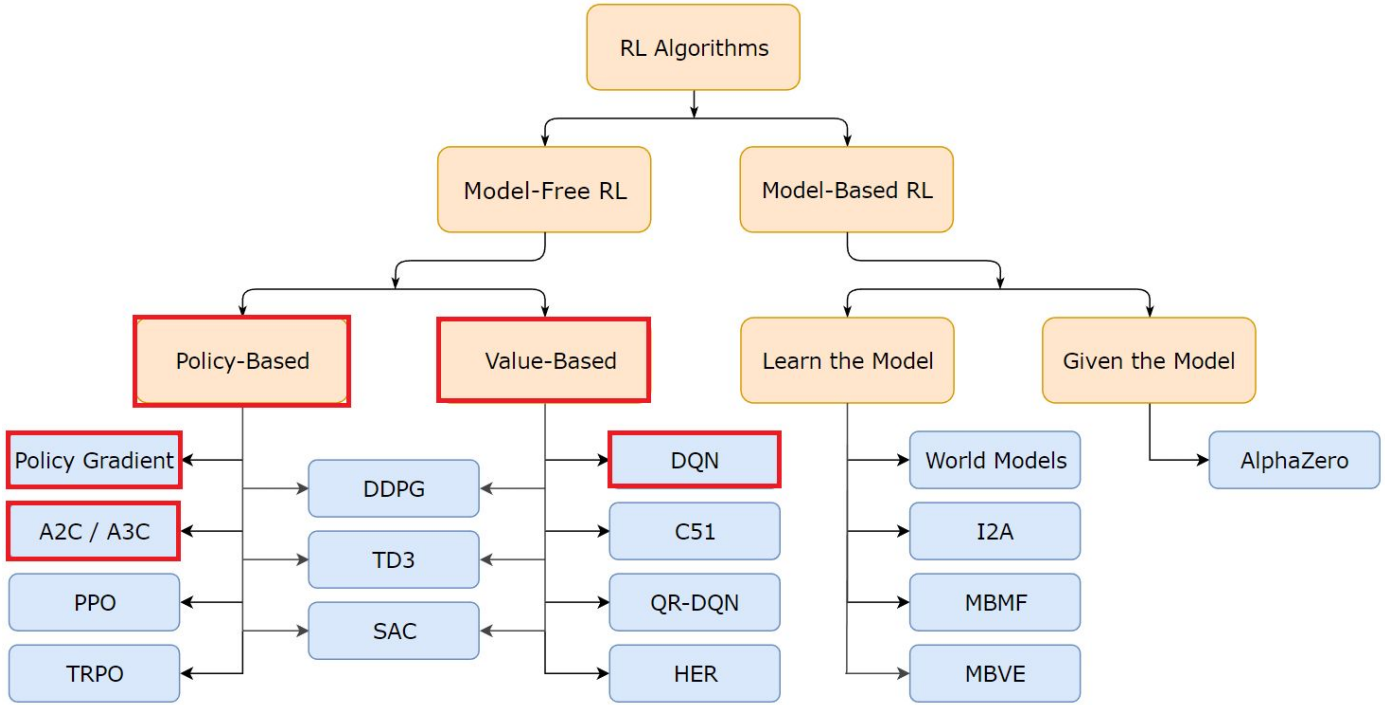


Taxonomy of Reinforcement Learning algorithms



OpenAI SpinningUp

Taxonomy of Reinforcement Learning algorithms



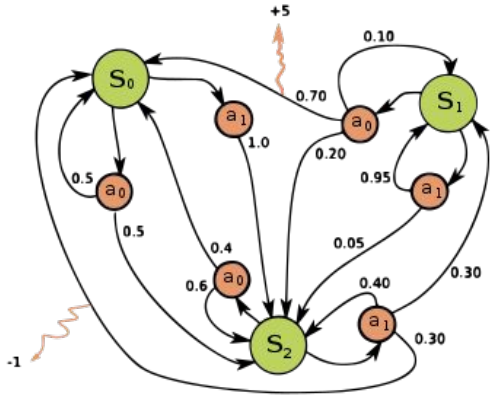
OpenAI SpinningUp

Fundamentals of Reinforcement Learning

Theory behind learning - from math to playing atari.

RL as Markov-Decision Process

- Markov decision process (MDP) is a **mathematical framework** for modeling decision-making in situations where outcomes are **partly random and partly under the control of a decision maker**. An MDP consists of a set of states, a set of actions, and a reward function.
- The goal in an MDP is to find an optimal policy.



https://en.wikipedia.org/wiki/Markov_decision_process

RL as Markov-Decision Process

The assumptions of MDPs:

1. **The environment is fully observable**: The agent has complete knowledge of the current state of the environment and can observe the probabilities of state transitions and rewards directly - **model.**
2. **The environment is stationary**: The transition probabilities and rewards do not change over time.
3. **The environment is discrete**: The set of states, actions, and rewards is finite and discrete.
4. **The environment is Markovian**: The future is independent of the past given the present, and the optimal policy depends only on the current state and not on the history of previous states.

Rarely the case in practice...

RL as Markov-Decision Process - Bellman equation

The Bellman equation is a fundamental equation in reinforcement learning that is used to solve MDPs.

It is used to **calculate the value of a state** or action, taking into account the **expected value of the next state** and the **reward** that is received for transitioning to that state.

RL as Markov-Decision Process - Bellman equation

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] \Big] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S},
\end{aligned}
$$

"relationship between the value of a state and the values of its successor states"

RL as Markov-Decision Process - Bellman equation

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] \Big] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) \boxed{\Big[r + \gamma v_\pi(s') \Big]}, \quad \text{for all } s \in \mathcal{S},
\end{aligned}
$$

"relationship between the value of a state and the values of its successor states"

RL as Markov-Decision Process - Bellman equation

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] \Big] \\
&= \sum_a \pi(a|s) \boxed{\sum_{s', r} p(s', r|s, a)} \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S},
\end{aligned}
$$

"relationship between the value of a state and the values of its successor states"

RL as Markov-Decision Process - Bellman equation

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] \Big] \\
&= \boxed{\sum_a \pi(a|s)} \sum_{s', r} p(s', r|s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S},
\end{aligned}
$$

"relationship between the value of a state and the values of its successor states"

RL as Markov-Decision Process – Bellman optimality equation

Reminder: π_* optimizes the expected return, so it always picks the best state/action indicated by v_* / q_*

Thus: We can find π_* by finding v_* or q_*.

Bellman optimality equation for v_*:

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_*(s')\right].
\end{aligned}
$$

RL as Markov-Decision Process – Bellman optimality equation

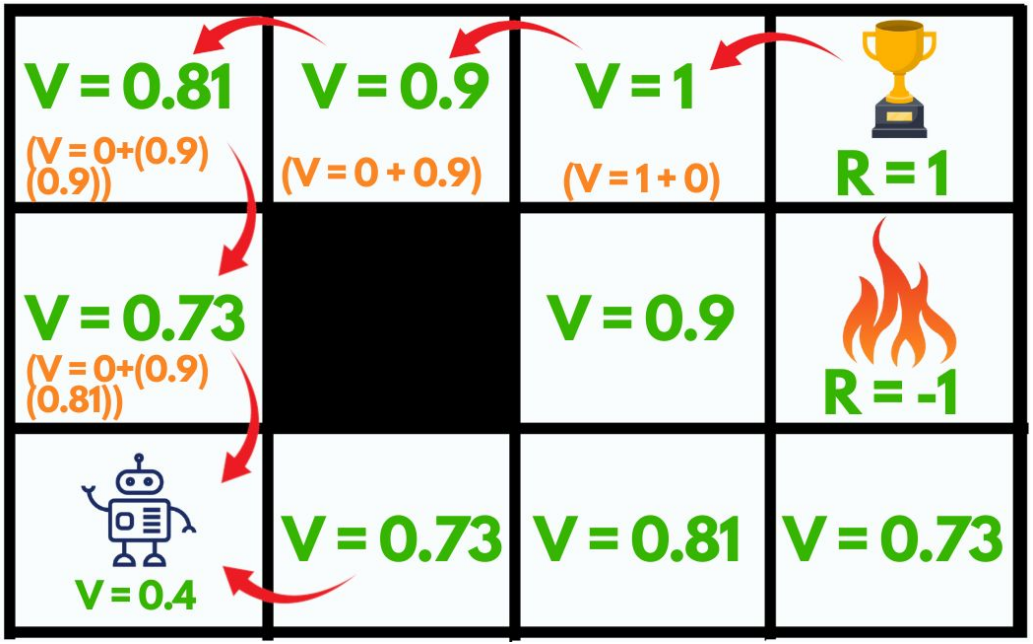
Reminder: π_* optimizes the expected return, so it picks the best state/action indicated by v_* / q_*

Thus: We can find π_* by finding v_* or q_*.

Bellman optimality equation for q_*:

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\middle|\; S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s', r \,|\, s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right].
\end{aligned}
$$

RL as Markov-Decision Process - Bellman equation

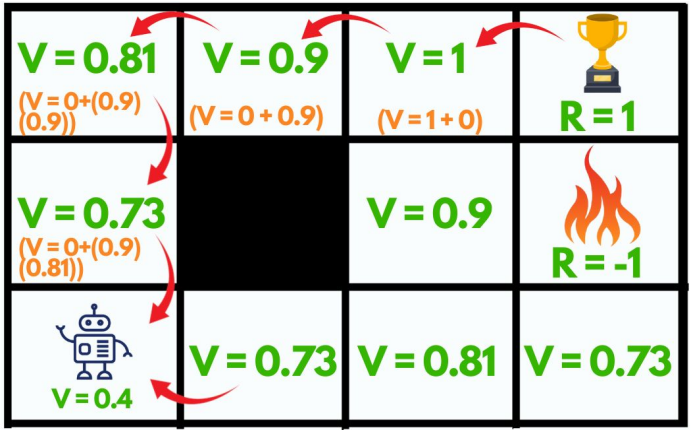


https://www.geeksforgeeks.org/bellman-equation/

RL as Markov-Decision Process – Bellman equation

Recap:

If we assume that we **know probabilities of state transitions** and that the **environment is discrete**, we can get calculate $\mathbf{v}_\pi$/$\mathbf{q}_\pi$.



https://www.geeksforgeeks.org/bellman-equation/

Dynamic Programming

Policy / Value Iteration

Dynamic Programming - Policy Evaluation

- **Dynamic Programming** can be used to find optimal policies in an iterative process.
- Policy Evaluation - approximation of v_{π} by using Bellman equation as an update rule.

$$
\begin{aligned}
v_{k+1}(s) &\doteq \mathbb{E}_{\pi}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \Big[r + \gamma v_k(s') \Big]
\end{aligned}
$$

- With each update as k ->∞, v_k -> v_{π}

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

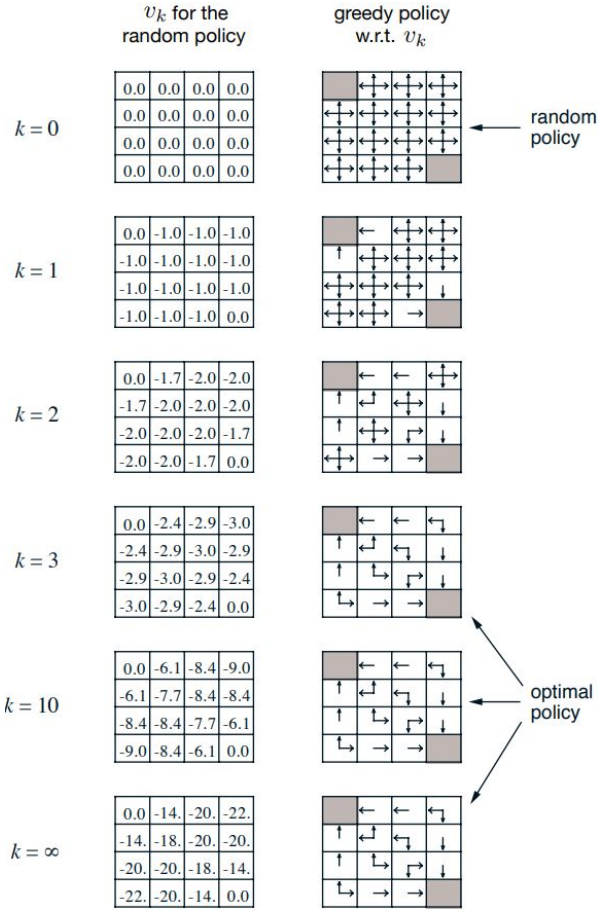
Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$ arbitrarily, for $s \in \mathcal{S}$, and $V(terminal)$ to 0

Loop:
  $\Delta \leftarrow 0$
  Loop 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 $\Delta < \theta$

Dynamic Programming – Policy Iteration

Policy improvement theorem in short:

- Get a random policy π
- Get v_π / q_π.
- Create a policy π' that's simply greedy to v_π – it must as good as π or better
- Repeat that until $\pi' = \pi$, which means we got π_*

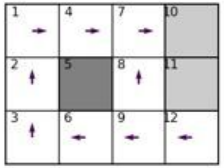


Dynamic Programming

First Iteration:

MDP gridworld

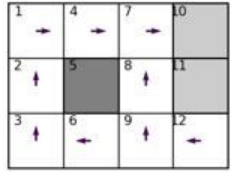
-0.287	-0.169	0.05	1.0
-0.354		-0.479	-1.0
-0.402	-0.451	-0.524	-0.696



Second Iteration:

MDP gridworld

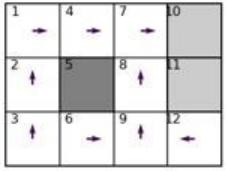
0.509	0.649	0.795	1.0
0.398		0.486	-1.0
0.291	0.207	0.168	-0.009



Third Iteration:

MDP gridworld

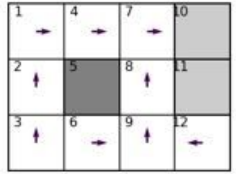
0.509	0.649	0.795	1.0
0.398		0.486	-1.0
0.291	0.207	0.34	0.126



Forth Iteration:

MDP gridworld

0.509	0.649	0.795	1.0
0.398		0.486	-1.0
0.296	0.253	0.344	0.129



Dynamic Programming - Policy Iteration

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(terminal) \doteq 0$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop 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 determining the accuracy of estimation)

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

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

Dynamic Programming - Value Iteration

- Policy Iteration is still not computationally effective - Policy Evaluation ends only with time limit.
- **Value Iteration** is a special case of Policy Iteration, where Policy Evaluation is stopped after just **one step** - notice that update is Bellman Optimality Equation

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$

Policy improvement step in contained within the $\max_a$

Dynamic Programming - Value Iteration

- We can change the update, from **Bellman Equation** to **Bellman Optimality Equation** to get v_*.

Value Iteration, for estimating $\pi \approx \pi_*$

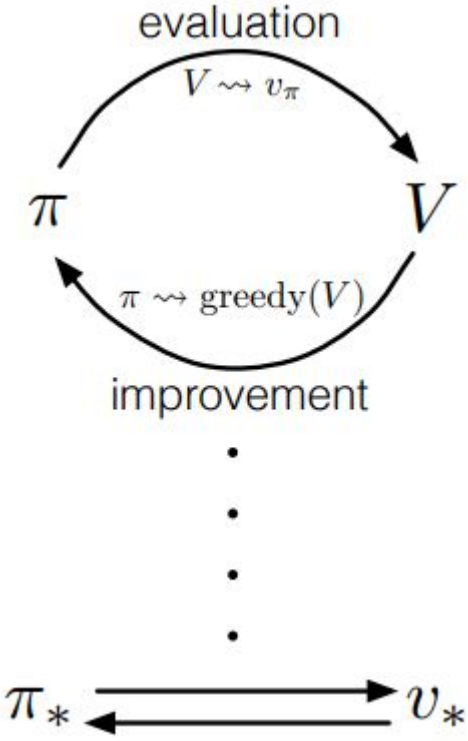
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

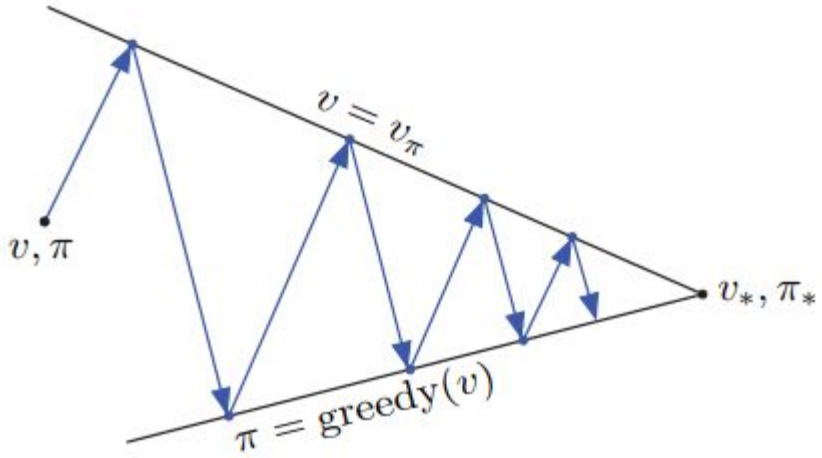
Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

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

Policy improvement step in contained within the $\max_a$

Dynamic Programming - Generalized Policy Iteration





Dynamic Programming

The problem:

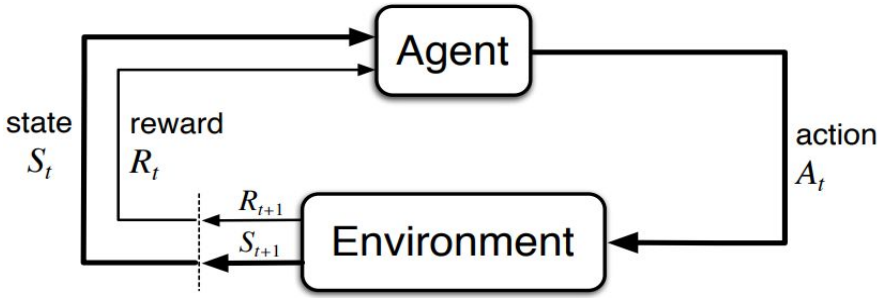
- Requires a complete model of the environment, including the transition probabilities and rewards for all state-action pairs.

Model-free approaches

TD-learning, Policy Gradient methods

Model-free approaches

- Where learning finally comes in.
- The exact model of the environment isn't available most of the time.
- Model-free methods aim to learn from **experience** only.
- Instead of **modeling the environment and solving equations**, we'll **interact with it**.



Model-free approaches - Monte Carlo

- Most basic - Monte Carlo methods - estimates this directly - by **averaging returns**.

$$v(st) = v(st) + \alpha\,[Gt - v(st)]$$ - this will converge to v_{π}

- Simply play out many episodes to get trajectories and calculate the returns.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow$ average$(Returns(S_t))$

Model-free approaches - Monte Carlo

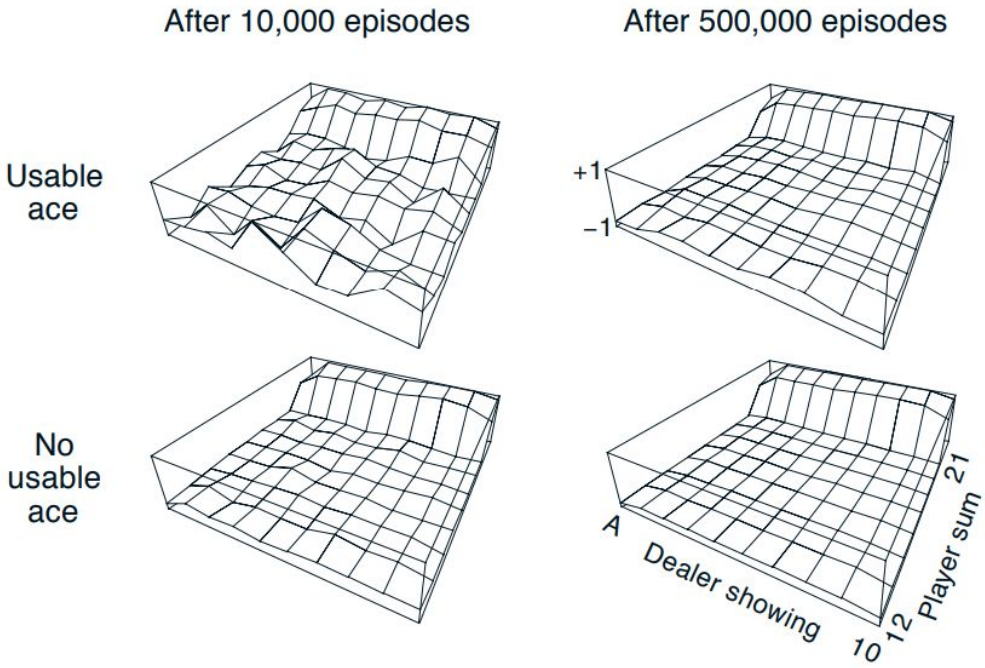


Figure 5.1: Approximate state-value functions for the blackjack policy that sticks only on 20 or 21, computed by Monte Carlo policy evaluation. ■

Model-free approaches - TD-Learning

- While Monte Carlo seems promising, since it needs returns, a **whole episode has to be played out, before an update can be done**.
- **Temporal-Difference (TD) learning** combines ideas from Dynamic Programming with **learning from experience** of Monte Carlo methods.

$$V(s_t) = V(s_t) + \alpha [R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

Model-free approaches - TD-Learning

- While Monte Carlo seems promising, since it needs returns, a **whole episode has to be played out, before an update can be done**.
- **Temporal-Difference (TD) Learning** combines ideas from Dynamic Programming with **learning from experience** of Monte Carlo methods.

$$V(s_t) = V(s_t) + \alpha [R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

  - Can be seen as an error - often called **TD-error**.
  - It's an estimate of the difference between the **estimated value of s_t** and the **better estimate $R_{t+1} + V(s_{t+1})$**

Model-free approaches - TD-Learning

- While Monte Carlo seems promising, since it needs returns, a **whole episode has to be played out, before an update can be done**.
- **Temporal-Difference (TD) Learning** combines ideas from Dynamic Programming with **learning from experience** of Monte Carlo methods.

$$V(s_t) = V(s_t) + \alpha [R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

- Can be seen as an error - often called **TD-error**.
- It's an estimate of the difference between the **estimated value of s_t** and the **better estimate $R_{t+1} + V(s_{t+1})$**
- Practically the same as Monte Carlo - but instead of Gt we look at $V(s_{t+1})$

Model-free approaches - TD-Learning

- While Monte Carlo seems promising, since it needs returns, a **whole episode has to be played out, before an update can be done**.
- **Temporal-Difference (TD) Learning** combines ideas from Dynamic Programming with **learning from experience** of Monte Carlo methods.

$$V(s_t) = V(s_t) + \alpha [R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

"If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning."

- Can be seen as an error - often called **TD-error**.
- It's an estimate of the difference between the **estimated value of s_t** and the **better estimate $R_{t+1} + V(s_{t+1})$**
- Practically the same as Monte Carlo - but instead of Gt we look at $V(s_{t+1})$

Model-free approaches - TD-Learning

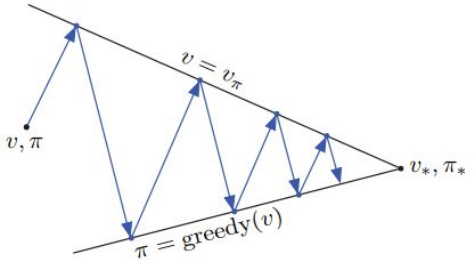
Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

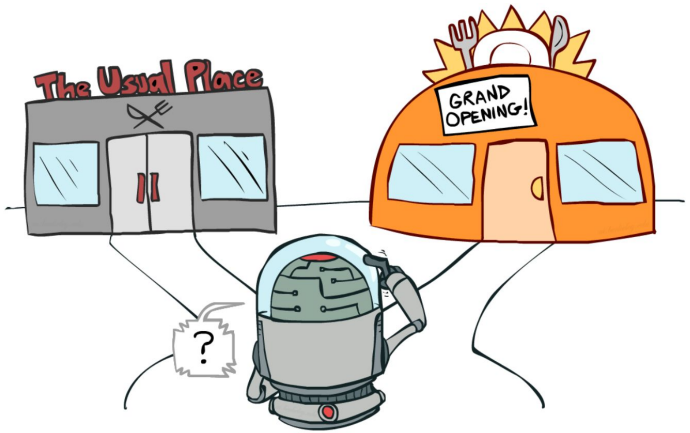
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Model-free approaches - TD-Learning

- Just like with Value Iteration, we could apply **Generalized Policy Iteration** to TD-learning to repeatedly get state values and create a policy that's greedy.



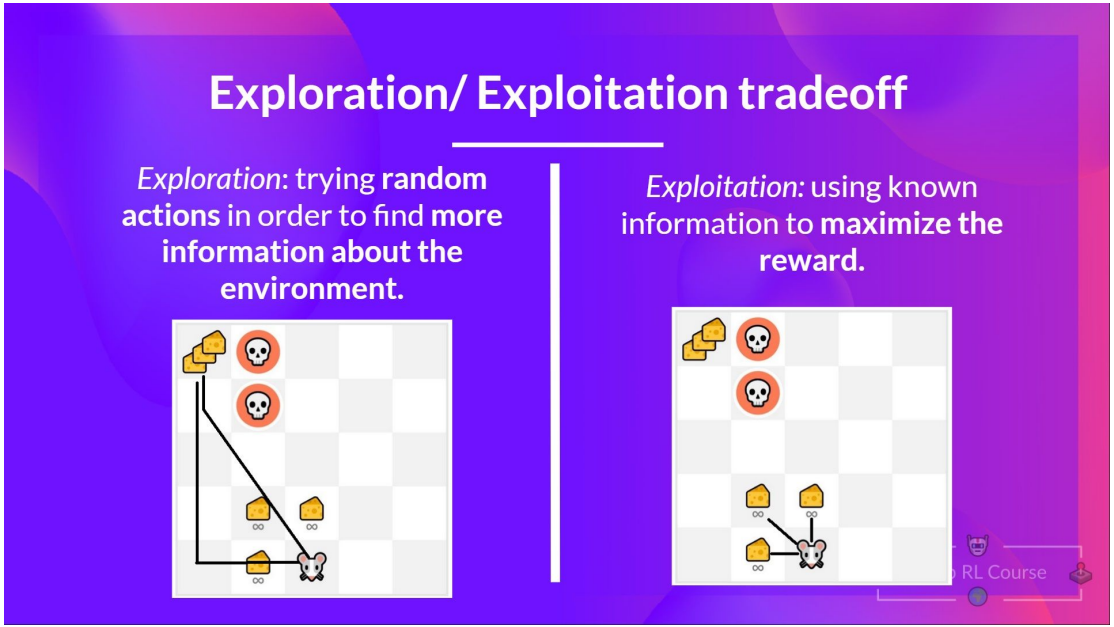
- However - since we're not sweeping over all states and sampling them with our policy instead - there's a new type of problem - **exploration vs. exploitation**.



UC Berkeley AI Course

Exploration vs. Exploitation

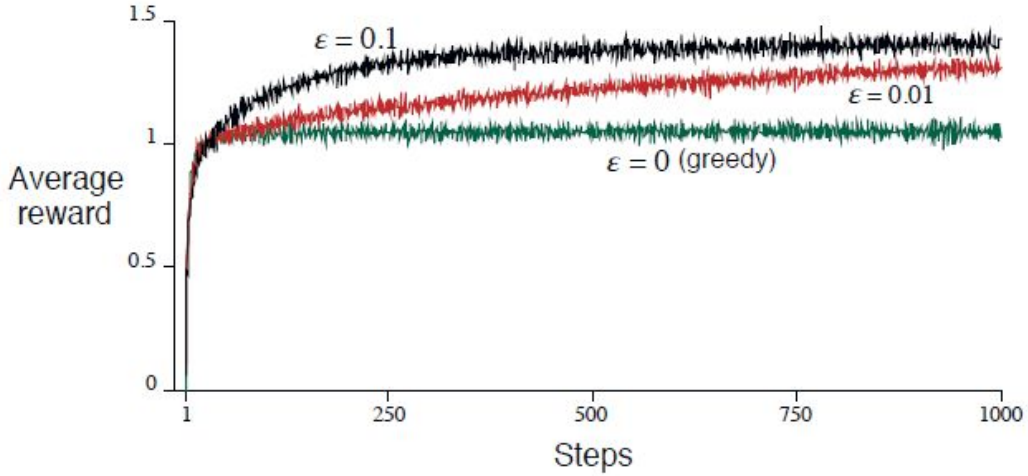
- It's a common challenge in all of Reinforcement Learning - not just TD.



HuggingFace Deep RL Course

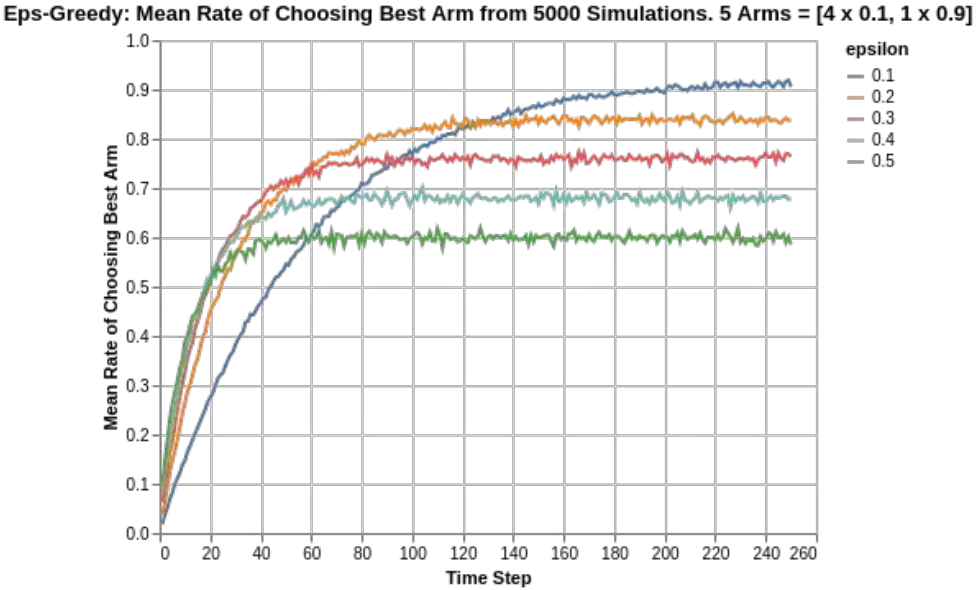
Exploration vs. Exploitation

- Easiest and most common approach - ε-greedy
- The policy is still greedy, but with ε-chance we take a random action.



Exploration vs. Exploitation

- A policy that sometimes takes random actions can't be optimal, so epsilon is often lowered over time.



Multi-Armed Bandit Analysis of Epsilon Greedy Algorithm

SARSA

- One of the algorithms that combines Generalized Policy Iteration with TD-learning.
- Like Value Iteration, SARSA updates it's value function - but with each step, not episode.

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

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

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

SARSA

- Notice that SARSA uses the same policy that it updates - an **On-Policy** approach.
- However we know that it's not an optimal one (since it explores).

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

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

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

Q-Learning

- Simple modification to SARSA - updating with regards to the greedy policy.
- Q-learning has been shown to to converge with probability 1 to q_*.
- **Off-policy** approach - we're updating with a different policy than we're following.

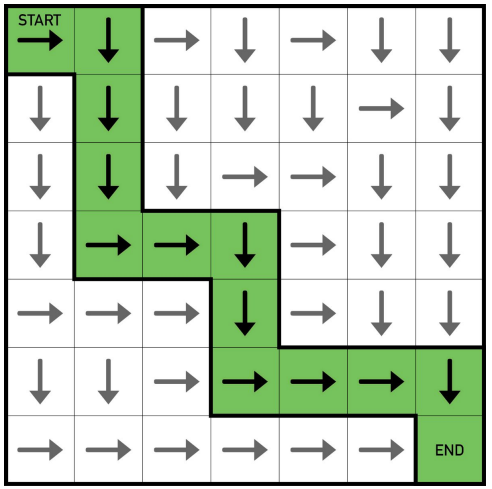
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

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

Tabular representation

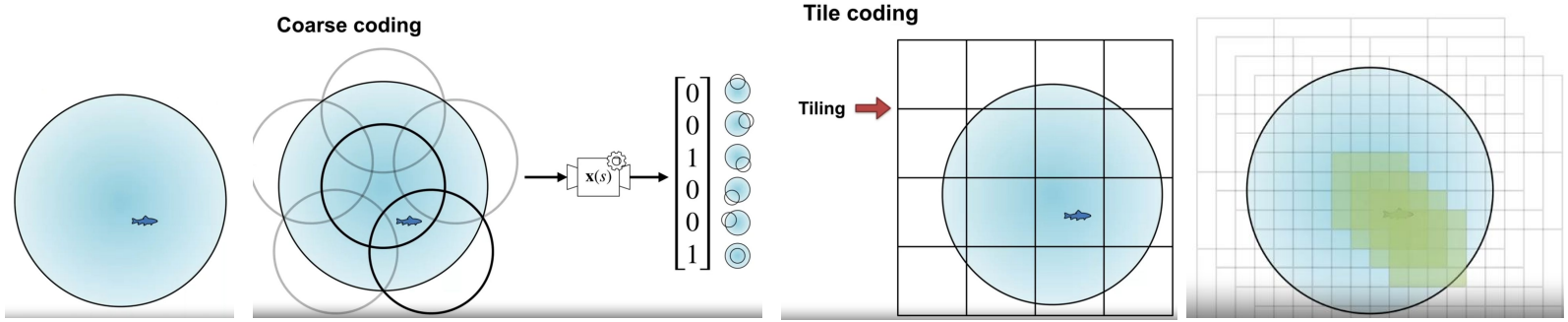
- It's all great, but we still have the assumption of keeping **state-action values in a table**.
- We can still directly apply it to some problems.
- But most of the time state space is either too big or continuous.



Training an Agent to beat Grid World

Discretization, state aggregation

- We can easily map continuous spaces to discrete ones.



- Similarly, we can aggregate huge discrete states into smaller ones.

Aggregatedstates	S1′			S2′			S3′			S4′		
Original states	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12

Q-Learning - Approximation, Deep RL

- Since it's practically feature engineering - a much better solution is to instead use approximation
- We can approximate $v_\pi(s)$ or $q_\pi(s,a)$ with a function approximator - e.g. linear function or a neural network.

$$v_w(s) = \hat{v}(s; w) \approx v_\pi(s)$$

$$q_w(s,a) = \hat{q}(s,a; w) \approx q_\pi(s,a)$$

- But what should the loss function be?

$$w = w + \alpha \left(v_\pi(s) - \hat{v}(s; w)\right) \nabla \hat{v}(s; w)$$

Q-Learning - Approximation, Deep RL

- But what should the loss function be?

$$w = w + \alpha \left(v_{\pi}(s) - \hat{v}(s; w)\right) \nabla \hat{v}(s; w)$$

- Don't have access to $v_{\pi}(s)$, but if we look back at TD-Learning:

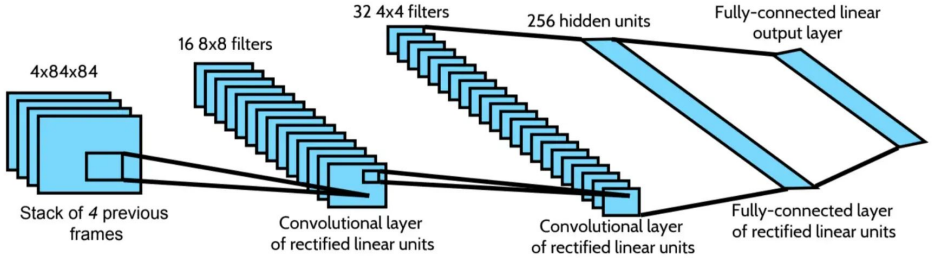
$$V(s_t) = V(s_t) + \alpha \left[\mathbf{R_{t+1} + \gamma V(s_{t+1})} - V(s_t)\right]$$

- We can use the same methodology here

$$w = w + \alpha \left(\mathbf{R_{t+1} + \gamma \hat{v}(s_{t+1})} - \hat{v}(s_t; w)\right) \nabla \hat{v}(s_t; w)$$

Q-Learning - Approximation, Deep RL

- This is exactly what DQN does.



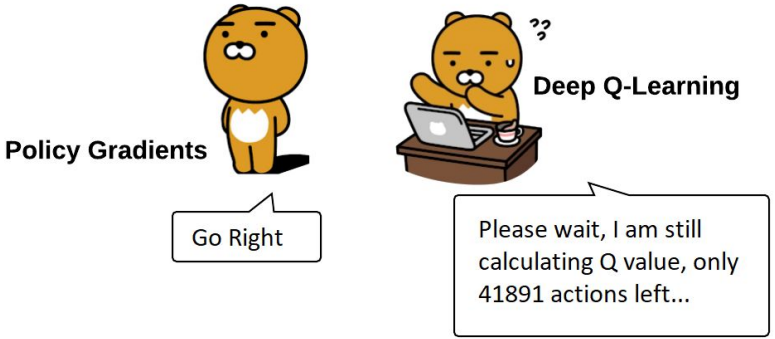
DeepMind Atari DQN

+ few simple trick:

- Skipping frames
- Experience replay
- Reward clipping
- Fixed target network

Policy Gradient Methods

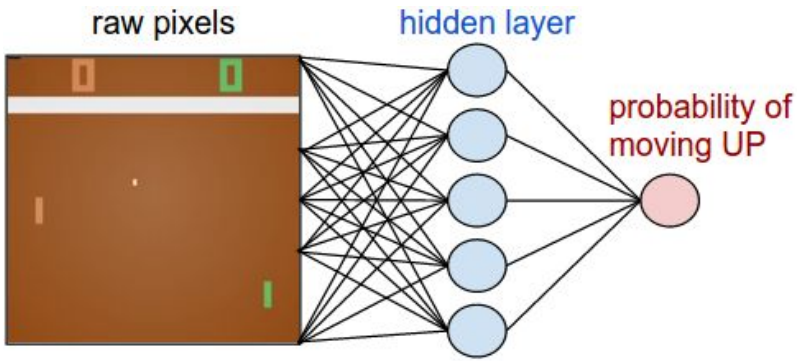
Learning the policy directly



Introduction to Reinforcement Learning Policy Gradient

Policy Gradient Methods

- What if instead of training a neural network to predict state values, we predict actions directly instead?
- Policy Gradient methods seek to learn $\pi(a_t \mid s_t)$ directly, omitting value functions.

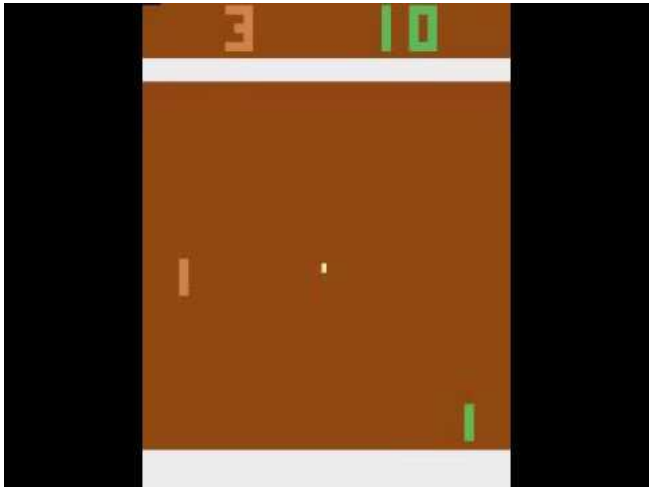


Deep Reinforcement Learning: Pong from Pixels

Policy Gradient Methods

- Objective: Learn weights θ for $\pi(a \mid s,\theta) = P(A_t=a \mid S_t=s, \theta_t=\theta)$
- Gradient ascent: $\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$
- J - the objective function, in RL is maximizing the return
- **REINFORCE** - Uses Monte Carlo approach as before, play out episodes and average the returns.

Loop forever (for each episode):
Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$



Policy Gradient Methods – REINFORCE

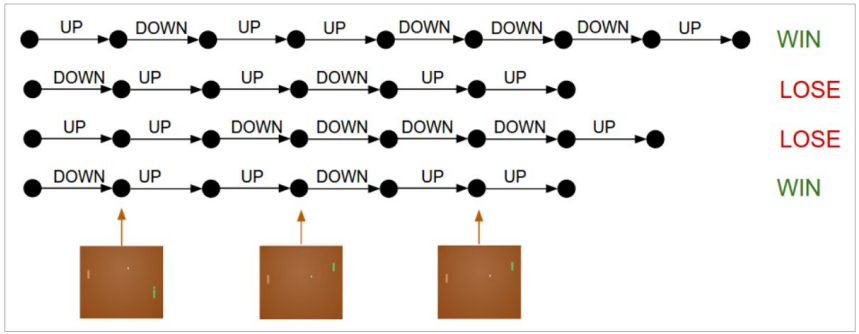
Pros over DQN:

- **Messy world:** In some environments, Q functions might be too complex to learn.
- **Speed:** Faster convergence
- **Stochastic Policies:** DQN learns deterministic only.
- **Continuous Actions:** Naturally models continuous action space. (probability distribution)

Cons over DQN:

- **Data inefficient:** Needs more data
- **Stability:** Less stable training.
- **Awarding actions:** Calculating rewards at the end – all actions are averaged as good/bad depending on the reward.

Policy Gradients: Run a policy for a while. See what actions led to high rewards. Increase their probability.



Deep Reinforcement Learning: Pong from Pixels

Advantage Actor-Critic (A2C)

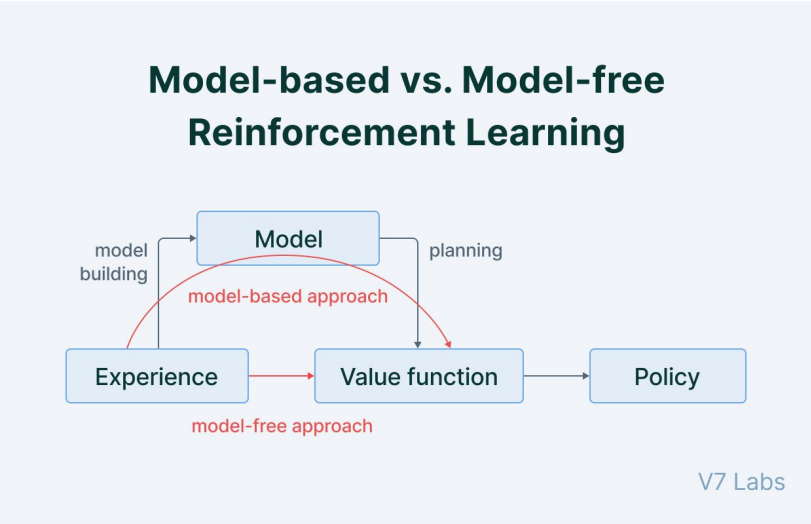
Advantage Actor-Critic (A2C)

- Combines **TD-Learning** with **REINFORCE**
- To learn the policy (**"actor"**), instead of averaging returns Gt (Monte Carlo), we approximate the value function q_π (**"critic"**) the same way as we did before in TD-Learning.

Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$ (if S' is terminal, then $\hat{v}(S', \mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S, \mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S, \boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

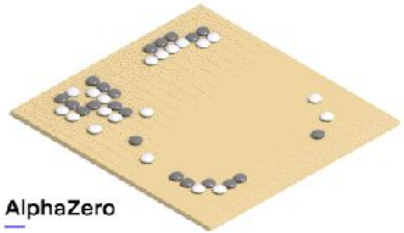
Model-based approaches

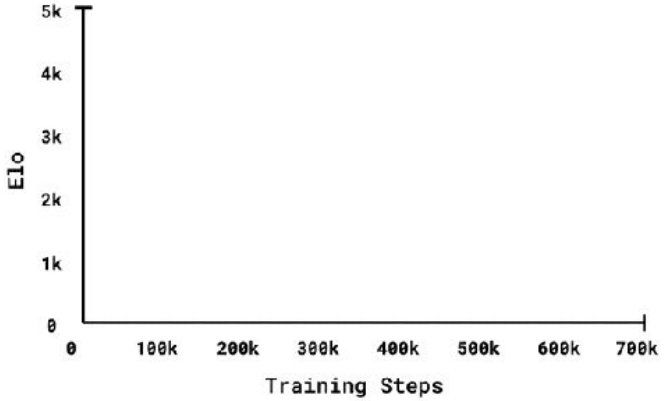
- Similar techniques and approaches.
- Main difference is that through having the **model**, they can use **planning** - “playing out” their actions and seeing which lead to the best future states.



AlphaZero

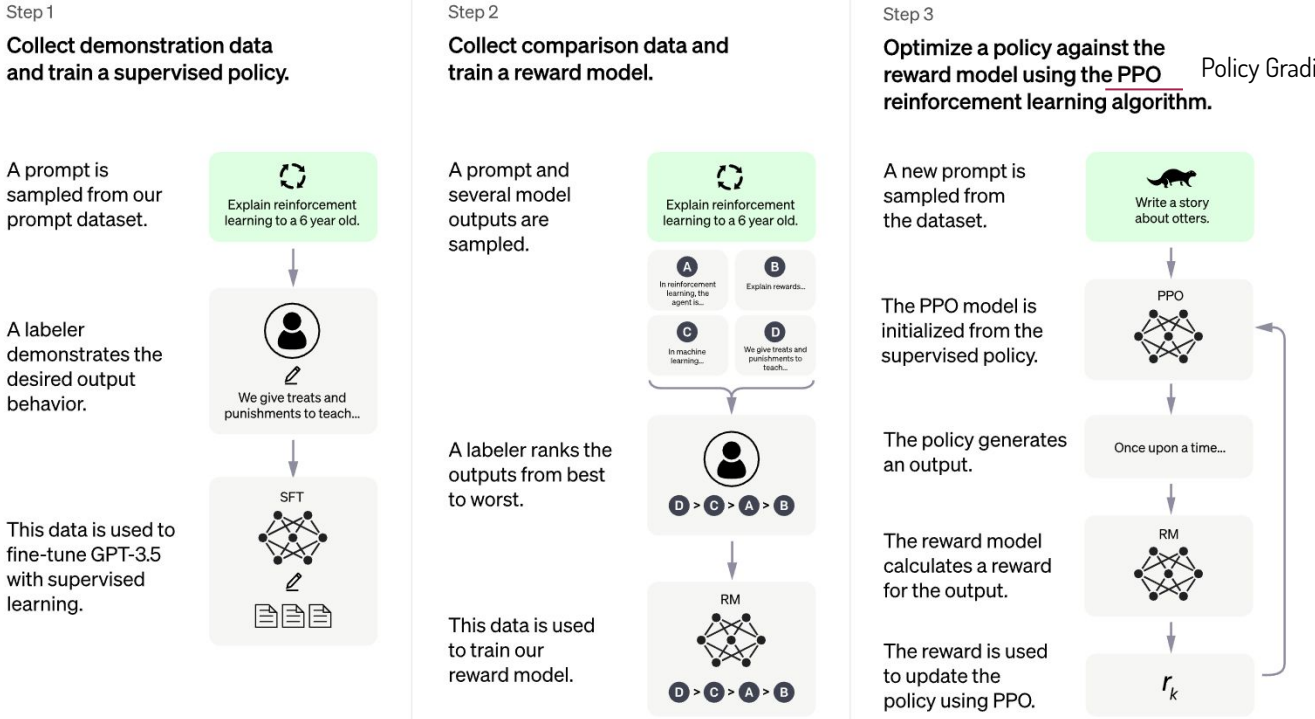
- **Monte Carlo** tree search (MCTS) to guide its search for the best moves.
- Neural network to estimate **values** of a board under the policies from policy network
- The policy network, to predict the action that is most likely to lead to the best outcome in a given state.





DeepMind's AlphaZero beats state-of-the-art chess and shogi game engines

Other examples



ChatGPT

Learning Agile Locomotion For Quadruped Robots

Policy Gradient based

Learning Agile Locomotion For Quadruped Robots

Autonomous Overtaking in Gran Turismo Sport Using Curriculum Reinforcement Learning

Yunlong Song*, HaoChih Lin*, Elia Kaufmann, Peter Dürr, and Davide Scaramuzza

***Abstract*— Professional race-car drivers can execute extreme overtaking maneuvers. However, existing algorithms for autonomous overtaking either rely on simplified assumptions about the vehicle dynamics or try to solve expensive trajectory-optimization problems online. When the vehicle approaches its physical limits, existing model-based controllers struggle to handle highly nonlinear dynamics, and cannot leverage the large volume of data generated by simulation or real-world driving. To circumvent these limitations, we propose a new learning-based method to tackle the autonomous overtaking problem. We evaluate our approach in the popular car racing game Gran Turismo Sport, which is known for its detailed modeling of various cars and tracks. By leveraging curriculum learning, our approach leads to faster convergence as well as increased performance compared to vanilla reinforcement learning. As a result, the trained controller outperforms the built-in model-based game AI and achieves comparable overtaking performance with an experienced human driver.**

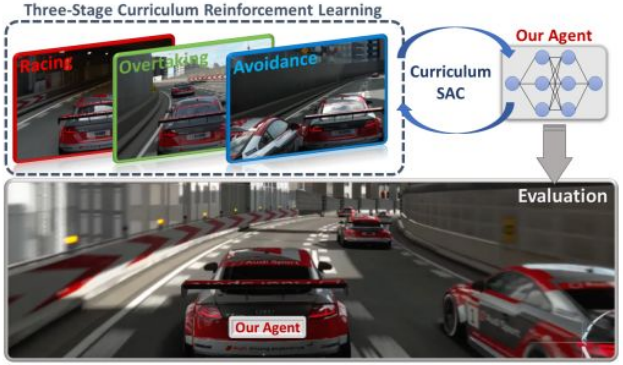


Fig. 1: A system overview of the proposed curriculum reinforcement learning method for addressing the autonomous overtaking problem in Gran Turismo Sport.

Actor-Critic version

Questions
&
Discussion

Thank you!

See you next week on ML Implementation.