Machine Learning

Introduction to Reinforcement Learning

Gabriele Russo Russo   Francesco Lo Presti

Laurea Magistrale in Ingegneria Informatica – A.Y. 2023/24

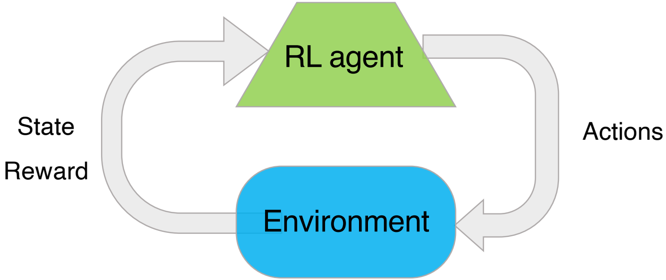
TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Dipartimento di Ingegneria Civile e Ingegneria Informatica

Reinforcement Learning

- Supervised learning
- Unsupervised learning
- **Reinforcement learning**
  - Branch of ML dealing with sequential decision-making

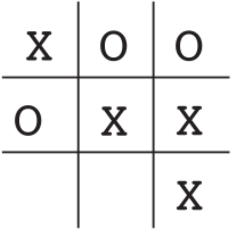
Reinforcement Learning



- Agent interacts with environment through actions
- Feedback in the form of reward (or paid cost)
- Goal: maximizing cumulated reward over the long run
- Trial-and-error experience (no complete knowledge of environment a priori)

Example: Tic-Tac-Toe

- State: representation of the board (3x3 matrix)
- Actions: available cells to mark
- Reward: 1 for a winning move, 0 otherwise



Example: AlphaZero by DeepMind

- Software able to play Go, Chess and Shogi [1]
  - Board games with huge number of legal positions (i.e., state space)
- Trained via self-play and advanced deep RL techniques
- Superhuman level of play with 24-hour training
- First presented in 2017; in 2019 MuZero, generalization to play Atari games and other board games without prior rule knowledge

[1] `https://arxiv.org/abs/1712.01815`

Example: AlphaDev by DeepMind

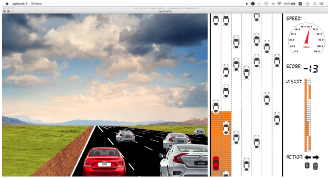
- Announced in 2023[2]
- RL used to develop new C++ sorting algorithm, now accepted in the standard library
- 70% faster on short sequences (2-3 items), 1.7% faster on long sequences
- State: instructions generated so far and state of the CPU
- Actions: assembly instructions to add
- Reward: based on sorting correctness and efficiency

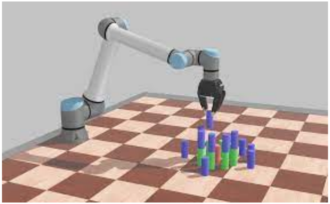
---

[2] `https://www.deepmind.com/blog/alphadev-discovers-faster-sorting-algorithms`

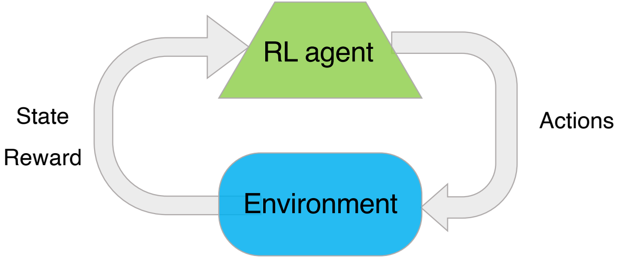
Other Examples

- Autonomous vehicles
- Robot control
- Trading
- Autonomous network and computer systems
- Videogames
- ...





Reinforcement Learning

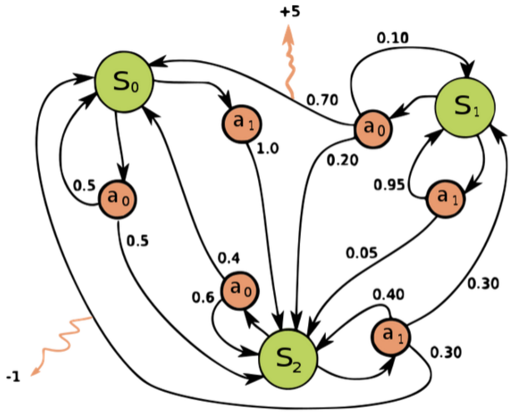


- Agent, environment, actions, state, rewards, . . .
- Modeled depending on the specific task
  - e.g., autonomous car uses different state information compared to chess player
- Formally defined as a Markov Decision Process (MDP)
- A framework to model decision making in situations where outcomes are partly random

Markov Decision Process (MDP)

- Extension of discrete-time Markov chains
- At each time step t, the process is in some state s_t
- The decision maker (the agent) chooses an action a_t among those available in state s_t
  - e.g., robot observes current position and decides direction to move; some directions might be blocked by obstacles
- Following a_t, the process moves to (random) state s_{t+1}
  - e.g., autonomous drone chooses an action to reduce altitude; actual outcome may depend on (unpredictable) wind speed
- Agent receives a reward (or, equivalently, pays a cost)
  - e.g., robot may get a reward for reaching its final destination
  - e.g., chess player rewarded at the end of a match

Example



Example of a simple MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows).

Markov Decision Process (2)

What defines an MDP?

- $\mathcal{S}$: a (finite) set of states
- $\mathcal{A}$: a (finite) set of actions
- p: state transition probabilities

$$p(s'|s,a) = P[S_{t+1} = s'|S_t = s, A_t = a]$$

- r: reward function (or, c: cost function)
  1. $r(s,a) = E[R_t|S_t = s, A_t = a]$
  2. $r(s,a,s') = E[R_t|S_t = s, A_t = a, S_{t+1} = s'] \longrightarrow$
     $r(s,a) = \sum_{s'} p(s'|s,a) r(s,a,s')$

Markov Property

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

Definition

A state S_t is Markov if and only if

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

- The state captures all relevant information from the history
- i.e., the state is a sufficient statistic of the future

Objective: Episodic Tasks

- Informally, we said that the agent aims to maximize the collected reward over time
- Let's consider an episodic task, where the agent-environment interaction naturally terminates at some final time step T
  - e.g., the end of a chess match
  - e.g., the time a robot reaches its destination or runs out of battery
- At time t, we aim to maximize the expected return G_t

$$G_t = R_t + R_{t+1} + \ldots + R_T$$

Objective: Continuing Tasks

- In many cases the agent–environment interaction does not break naturally into identifiable episodes, but goes on continually without limit
  - e.g., an agent managing VM migration in a Cloud datacenter
  - e.g., the control system of RL-based traffic lights
- In this scenario, the goal of the agent is maximizing the expected cumulative discounted reward

$$G_t = R_t + \gamma R_{t+1} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$$

where $\gamma \in [0, 1)$ is the discount factor

Reward vs Cost

- You can either maximimize the expected reward or minimize the expected cost

$$G_t = C_t + \gamma C_{t+1} + \ldots = \sum_{k=0}^{\infty} \gamma^k C_{t+k}$$

- The two formulations are equivalent; you can easily switch between them by setting

$$r(s, a) = -c(s, a)$$

- In the following, we will mostly refer to costs; keep in mind this equivalence

Policy

Definition

A policy π is a distribution over actions given a state s

$$\pi(a|s) = p(A_t = a|S_t = s)$$

- A policy fully defines agent's behavior
- MDP policies depend on the current state only
- Special case: deterministic policy

$$\pi : \mathcal{S} \to \mathcal{A}$$

Example: Deterministic Policy

State	*Action*
s_1	a_1
s_2	a_1
s_3	a_2
s_4	a_1

Value Function

Value function is a prediction of future costs

- can be used to evaluate how good/bad states and/or actions are
- and therefore to select actions e.g.

State	a_1	a_2	a_3
s_1	10	5	3
s_2	8	6	4
s_3	6	5	6
s_4	5	4	6
s_5	4	3	7
s_6	1	5	9
s_7	0	9	15

s	$\pi(s)$
s_1	a_3
s_2	a_3
s_3	a_2
s_4	a_2
s_5	a_2
s_6	a_1
s_7	a_1

Value Functions

Action value function (or, Q function)

Expected cost starting from state s, taking action a and then following policy π

$$Q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$

State value function

Expected cost starting from state s and then following policy π

$$V_\pi(s) = E_\pi[G_t | S_t = s]$$

Action Value Functions

The action value function can be decomposed into two parts:

- immediate cost
- discounted costs from successor state S_{t+1}

$$
\begin{aligned}
Q_\pi(s,a) &= E_\pi[G_t | S_t = s, A_t = a] \\
&= E_\pi[C_t + \gamma C_{t+1} + \gamma^2 C_{t+2} \dots | S_t = s, A_t = a] \\
&= E_\pi[C_t + \gamma (C_{t+1} + \gamma C_{t+2} \dots) | S_t = s, A_t = a] \\
&= E_\pi[C_t + \gamma G_{t+1} | S_t = s, A_t = a] \\
&= c(s,a) + \gamma E_\pi[G_{t+1} | S_t = s, A_t = a]
\end{aligned}
$$

Bellman equation:

$$
Q_\pi(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) Q_\pi(s', \pi(s'))
$$

State Value Functions

The value function can be similarly decomposed into two parts:

- immediate cost C_t
- discounted cost from successor state $V(S_{t+1})$

$$
\begin{aligned}
V_\pi(s) &= E_\pi[G_t | S_t = s] \\
&= E_\pi[C_t + \gamma C_{t+1} + \gamma^2 C_{t+2} \dots | S_t = s] \\
&= E_\pi[C_t + \gamma (C_{t+1} + \gamma C_{t+2} \dots) | S_t = s] \\
&= E_\pi[C_t + \gamma G_{t+1} | S_t = s]
\end{aligned}
$$

Bellman equation:

$$
V_\pi(s) = c(s, \pi(s)) + \gamma \sum_{s'} p(s' | s, \pi(s)) V_\pi(s')
$$

Optimal Value Function

Optimal action value function

$Q^*(s; a)$ is the maximum action-value function over all policies

$$Q^*(s, a) = \max_{\pi} Q_{\pi}(s, a)$$

Optimal state value function

$V^*(s)$ is the minimum value function over all policies

$$V^*(s) = \max_{\pi} V_{\pi}(s)$$

Bellman Optimality Equations

$$Q_\pi(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) Q_\pi(s', \pi(s'))$$

$$\Downarrow$$

$$Q^*(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \min_{a'} Q^*(s',a')$$

$$V^*(s) = \min_a Q^*(s,a)$$

$$Q^*(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V^*(s')$$

Optimal Policy

Given $Q^*(s, a)$ the optimal action when the system is in state s is:

$$\pi^*(s) = a^*(s) = \arg\min_{a \in \mathcal{A}} Q^*(s, a)$$

State	a_1	a_2	a_3
s_1	10	5	3
s_2	8	6	4
s_3	6	5	6
s_4	5	4	6
s_5	4	3	7
s_6	1	5	9
s_7	0	9	15

$$\Longrightarrow$$

Optimal Action
a_3
a_3
a_2
a_2
a_2
a_1
a_1

How to compute V^*?

- If we know the optimal value function, we have an optimal policy!
- But... how do we compute the optimal value function??

Value Iteration

Bellman Equation

$$Q^*(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \min_{a'} Q^*(s',a')$$

- Suppose we know the solution to subproblems $Q^*(s',a')$
- $Q^*(s,a)$ can be computed by one-step lookahead

$$Q^*(s,a) \leftarrow c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \min_{a'} Q^*(s',a')$$

- The idea is to apply these updates iteratively
- Proven to converge (see, Contraction Mapping Theorem in Sutton's book)

Value Iteration: Algorithm

Value Iteration

$i \leftarrow 0$

$Q_i(s,a) \leftarrow 0, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$

repeat

 forall $s \in \mathcal{S}$ **do**

  forall $a \in \mathcal{A}(s)$ **do**

   $Q_{i+1}(s,a) \leftarrow$

    $c(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) \min_{a' \in \mathcal{A}(s')} Q_i(s',a')$

  end

 end

 $i \leftarrow i+1$

until $\max_{s,a} |Q_i(s,a) - Q_{i-1}(s,a)| < \epsilon$

$\pi^*(s) = \arg\min_a Q_i(s,a), \forall s \in \mathcal{S}$

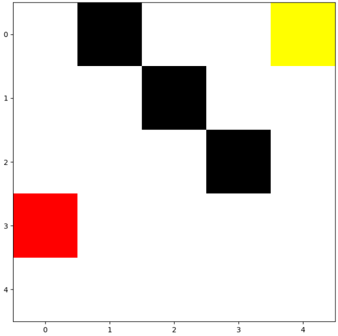
Value Iteration: Alternative Algorithm

Value Iteration - Alternative

$i \leftarrow 0$
$Q_i(s, a) \leftarrow 0, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$
$V_i(s) \leftarrow 0, \forall s \in \mathcal{S}$
repeat
    forall $s \in \mathcal{S}$ **do**
        forall $a \in \mathcal{A}(s)$ **do**
            $Q_{i+1}(s, a) \leftarrow c(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V_i(s)$
        end
        $V_{i+1}(s) = \min_{a' \in \mathcal{A}(s)} Q_{i+1}(s, a')$
    end
    $i \leftarrow i + 1$
until $\max_{s,a} |Q_i(s, a) - Q_{i-1}(s, a)| < \epsilon$
$\pi^*(s) = \arg\min_a Q_i(s, a), \forall s \in \mathcal{S}$

Example: Maze

- Consider a $S \times S$ grid
- Episodes start with agent randomly located in a cell in the first column
- Goal: reaching target cell $(1, S)$
- Some cells are blocked
- Some cells are slippery: when entering, the agent has a probablity p_{slip} of slipping one cell ahead along her current direction

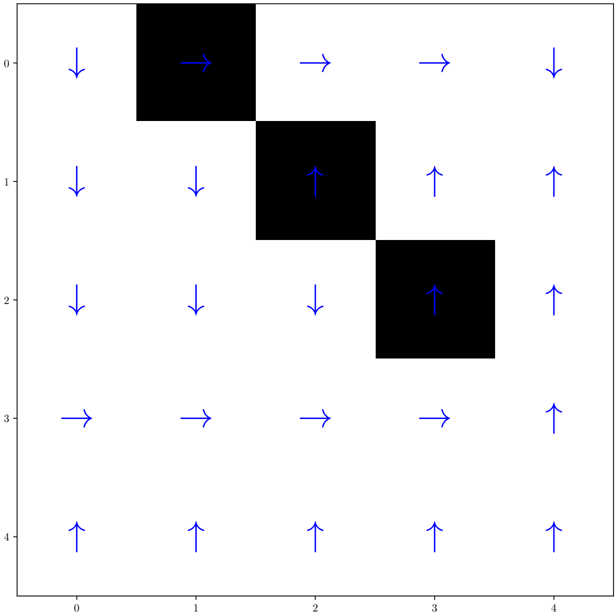


Example: Maze (2)

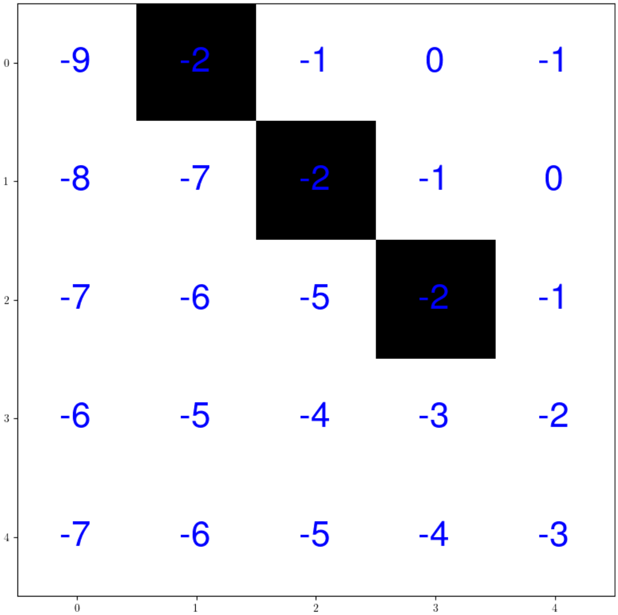
- State: $s = (x, y)$
- Actions: $a \in \{(0,1), (0,-1), (1,0), (-1,0)\}$
- Reward:
  - 0 for entering the goal cell
  - -M for exiting the grid or crashing into a blocked cell ($M \gg 1$)
  - -1 otherwise

maze.py (`--agent mdp`)

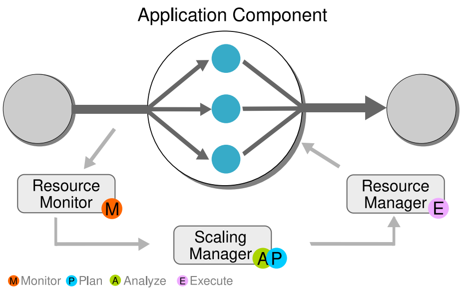
Maze: Optimal Policy



Maze: Optimal Value Function

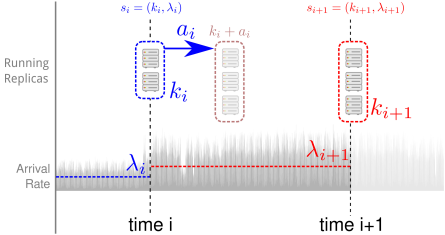


Example: Cloud Auto-scaling



- We periodically make a decision about scaling in/out an app component (a thread, a VM, a container, . . .)
- We are concerned with 3 objectives:
  - Monetary resource cost (or, resource usage in general)
  - Performance req. satisfaction (e.g., max response time)
  - Scaling overhead

Auto-scaling: MDP formulation



- State at time slot i: $s_i = (k_i, \lambda_i)$
  - k_i component parallelism
  - λ_i avg. arrival rate (of requests, jobs, data, ...)
- Action at time slot i: $a_i \in \{0, +1, -1\}$

MDP Model: Transition Probabilities

- State of the system $s = (k, \lambda)$
  - $1 \leq k \leq K^{max}$ Component parallelism
  - λ avg. input rate
    - λ is discretized , i.e., $\lambda_i \in \{0, \Delta\lambda, 2\Delta\lambda, (L-1)\Delta\lambda\}$
    - $\Delta\lambda$ quantization step size, L number of discrete values
- Available actions $\mathcal{A} = \{-1, 0, +1\}$
- Transition probabilities $p(s'|s, a) = p\left((k', \lambda')|(k, \lambda), a\right)$

$$
\begin{aligned}
p(s'|s,a) &= P[s_{t+1} = (k', \lambda')|s_t = (k, \lambda), a_t = a] = \\
&= \begin{cases} P[\lambda_{t+1} = \lambda'|\lambda_t = \lambda] & k' = k + a \\ 0 & \text{otherwise} \end{cases} = \\
&= \mathbb{1}_{\{k'=k+a\}} P[\lambda_{t+1} = \lambda'|\lambda_t = \lambda]
\end{aligned}
$$

MDP Model: Cost Function

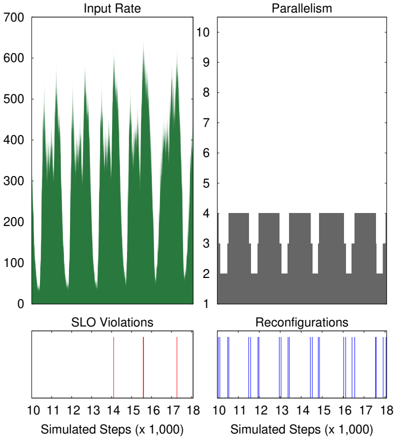
$$c(s, a, s') = \underbrace{w_{res} \frac{k + a}{K^{max}}}_{\text{Resource Cost}} + \underbrace{w_{perf} \mathbb{1}_{\{R(s,a,s') > R^{max}\}}}_{\text{Performance}} + \underbrace{w_{rcf} \mathbb{1}_{\{a \neq 0\}}}_{\text{Reconfig.}}$$

- $w_{res} + w_{perf} + w_{rcf} = 1, w_x \geq 0, x \in \{res, perf, rcf\}$
- $R(s, a, s')$: performance index, e.g, response time
- R^{max}: reference performance value

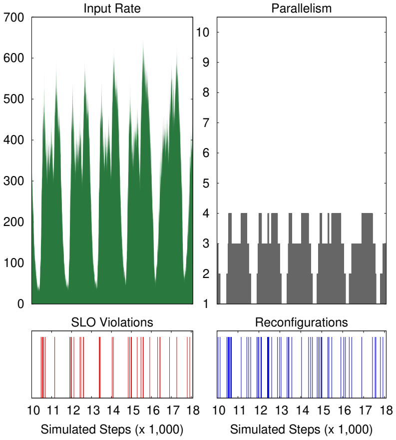
- We want to minimize $\sum_{t=0}^{\infty} \gamma^t c(s_t, a_t, s_{t+1}), \quad \gamma \in [0, 1)$

Trading-off Objectives

w_{perf}=0.6, $w_{res} = w_{rcf}$=0.2



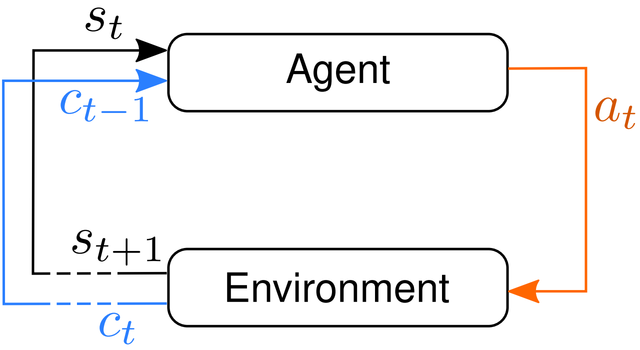
w_{res}=0.6, $w_{perf} = w_{rcf}$=0.2



MDP Resolution

- We can use the Value Iteration algorithm to solve the MDP
  - i.e., finding the optimal policy
- Is this enough?
- Unfortunately, solving the MDP requires exact and complete knowledge of the underlying model
  - state transition probabilities
  - cost function
- In practice, we don't have such information!

Reinforcement Learning



- RL aims to learn the optimal policy through interaction and evaluative feedback

Model-free vs Model-based RL

- **Model-free** RL: no model of the environment is available or used; the optimal policy is learned through experience only
- **Model-based** RL: a (possibly partial) model of the environment is available and used to derive the optimal policy
  - a partial model can boost learning speed
  - RL may also be used in presence of a complete model instead of VI; e.g., with a large number of rarely visited states VI would unnecessarily run for a long time!
  - You may also try to learn the model online and use it to compute a policy

Value-based vs Policy-based RL

- Value-based RL: aims to learn the optimal value function through experience; the policy is derived from it
  - Simplest RL algorithms belong to this group
  - We will mainly focus on this group in the following
- Policy-based RL: aims to directly learn the optimal policy through experience; no explicit computation/learning of the value function
- Hybrid approaches: e.g., the Actor-Critic framework

Simple Value-based RL Algorithm

A simple RL algorithm

$t \leftarrow 0$
Initialize Q
Loop
    $t \leftarrow t + 1$
EndLoop

Q-learning

- Proposed by Chris Watkins in 1989
- One of the most known (and simplest) RL algorithms
- Proven to converge to the optimal policy under mild assumptions
  - …after n steps, with $n \to \infty$

Q-learning: Action Selection

- How to choose an action at every time step?
- Exploration vs Exploitation dilemma

- Exploitation: using available knowledge to maximize reward
  - choose the "best" action, i.e., $a_t = \arg\max_a Q(s_t, a)$
- Exploration: discovering more information about the environment
  - choose other actions to learn more about the environment

Q-learning converges only if all state-action pairs are visited an infinite number of times as $t \to \infty$

- you can't exploit all the time
- you can't explore all the time

ϵ-Greedy Exploration

- Popular approach for the exploration-exploitation dilemma
- With probability $1 - \epsilon$ choose the greedy action $a^* = \arg\max_{a \in \mathcal{A}} Q(s, a)$
- With probability ϵ choose an action at random

- Improvement: ϵ-greedy with decaying ϵ (similar to decaying learning rate in SGD)

Softmax Action Selection

- Alternative to the ϵ-greedy strategy
- All actions assigned non-zero probability of being chosen
- Action $a \in \mathcal{A}$ is selected with probability

$$\pi(a|s) = \frac{exp(Q(s,a)/\tau)}{\sum_{a' \in \mathcal{A}} exp(Q(s,a')/\tau)}$$

- τ is the "temperature"
  - Small τ leads to greedy behavior
  - Large τ leads to random action selection
  - You usually start with a large temperature value and let it decay

Q-learning: Updating Q

With known model, we can compute Q iteratively using:

$$Q(s,a) \leftarrow c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \max_{a'} Q(s',a')$$

Q-learning uses *point estimates* on experience $\{s_t, a_t, c_t, s_{t+1}\}$:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \underbrace{\alpha_t}_{\text{Learning Rate}} \Big[\underbrace{r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1},a')}_{\text{Target}} - Q(s_t,a_t) \Big]$$

Q-learning: Algorithm

Q-learning

$t \rightarrow 0$
Initialize Q (e.g., zero-initialized)
Loop
  choose a_t (e.g., ϵ-greedy or softmax selection)
  observe next state s_{t+1} and reward r_t
  $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left[r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t)\right]$
  $t \leftarrow t + 1$
EndLoop

Example: Maze

- `python maze.py --agent qlearning --episodes N [-- plot_reward]`

maze.py

qlearning.ipynb

SARSA

- Q-learning is an off-policy algorithm
  - The algorithm uses the greedy policy to update Q, but likely chooses action according to another policy (e.g., ϵ-greedy)
- **SARSA**: on-policy algorithm similar to Q-learning
- The same policy is used to choose next action and to update Q

SARSA: Algorithm

SARSA

$t \rightarrow 0$
Initialize Q (e.g., zero-initialized)
choose a_t (e.g., ϵ-greedy or softmax selection)
Loop
  observe next state s_{t+1} and reward r_t
  choose a_{t+1} (e.g., ϵ-greedy or softmax selection)
  $Q(s_t, a_t) \leftarrow$
    $Q(s_t, a_t) + \alpha_t \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]$
  $t \leftarrow t + 1$
EndLoop

Dealing with Large State Spaces: Deep RL

Issues with Tabular RL

- So far, we have considered tabular representations of the value function

State/Action	a_1	a_2	$\ldots$
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	$\ldots$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	$\ldots$
$\ldots$	$\ldots$		
s_n	$Q(s_n, a_1)$	$Q(s_n, a_2)$	$\ldots$

- Not ideal as the state space grows…
- Memory demand: $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$
- No generalization
- How to handle continuous state spaces?

Value Function Approximation

Idea: using a parametric approximation of the value function

$$V_\pi(s) \approx \hat{V}(s, \boldsymbol{w}), \text{ or}$$
$$Q_\pi(s, a) \approx \hat{Q}(s, a, \boldsymbol{w})$$

- $\boldsymbol{w} \in \mathbb{R}^d$ is a vector of parameters
- We need to store $\boldsymbol{w}$ instead of the Q table
  - Reduced memory demand if $d < |\mathcal{S}|$ ✓
- Potential generalization ✓
  - The experience gained in a state used to update $\boldsymbol{w}$
  - A single update possibly impacts the value of several states!
  - Can deal with continuous state spaces ✓

Value Function Approximation (2)

- How to choose a function $\hat{Q}$?
- How to determine the value of $\boldsymbol{w}$?
- We search for a function and a vector $\boldsymbol{w}$ so as to approximate V (or Q) “well”

- First of all, what does "well" means?

Function Approximation: Objective

- A simple and natural choice is to minimize MSE:

$$J(\boldsymbol{w}) = \sum_{s \in \mathcal{S}} \mu(s) \left[V_{\pi}(s) - \hat{V}(s, \boldsymbol{w}) \right]^2$$

- $\mu(s) \geq 0$ is a distribution over states
- $\mu(s)$ should reflect the importance or frequency of states

Optimizing Parameters

We can compute parameters $\boldsymbol{w}$ through gradient descent

$$\begin{aligned}\boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \frac{1}{2}\alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w_t}) = \\ &= \boldsymbol{w}_t + \alpha \sum_{s \in \mathcal{S}} \mu(s) \left[V_\pi(s) - \hat{V}(s, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{V}(s, \boldsymbol{w})\end{aligned}$$

Two potential issues:

1. Summation over all states (may be expensive!)
2. We don't have the true values $V_\pi(s)$!

Optimizing Parameters: Issue 1

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \sum_{s \in \mathcal{S}} \mu(s) \left[V_\pi(s) - \hat{V}(s, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{V}(s, \boldsymbol{w})$$

Gradient computed over all states…

Stochastic gradient descent

one (or few) samples $(s_t, V_\pi(s_t))$ at each step

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[V_\pi(s_t) - \hat{V}(s_t, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{V}(s_t, \boldsymbol{w})$$

Optimizing Parameters: Issue 2

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \sum_{s \in \mathcal{S}} \mu(s) \left[V_\pi(s) - \hat{V}(s, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{V}(s, \boldsymbol{w})$$

How to get exact values?

Stochastic semi-gradient descent:
we replace $V_\pi(s_t)$ with a noisy approximation U_t, based on estimated value func.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[U_t - \hat{V}(s_t, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{V}(s_t, \boldsymbol{w})$$

A possible approach (inspired by Q-learning):

$$U_t = r_t + \gamma \hat{V}(s_{t+1}, \boldsymbol{w}_t)$$

Linear Function Approximation

The simplest possible approximation model:

$$\hat{V}(s, \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(s) = \sum_{i=1}^{d} w_i \phi_i(s)$$

Weights
$\mathbf{w} \in \mathbb{R}^d$

Features
$\boldsymbol{\phi} : \mathcal{S} \to \mathbb{R}^d$

Update rule becomes very simple:

$$\nabla_{\mathbf{w}} \hat{V}(s, \mathbf{w}) = \boldsymbol{\phi}(s)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{V}(s_t, \mathbf{w}_t) \right] \boldsymbol{\phi}(s_t)$$

Linear Function Approximation (2)

We have equivalent formulas for Q:

$$\hat{Q}(s, a, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{\phi}(s, a) = \sum_{i=1}^{d} w_i \phi_i(s, a)$$

Weights
$\boldsymbol{w} \in \mathbb{R}^d$

Features
$\boldsymbol{\phi} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$

$$\nabla_{\boldsymbol{w}} \hat{Q}(s, a, \boldsymbol{w}) = \boldsymbol{\phi}(s, a)$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[U_t - \hat{Q}(s_t, a_t, \boldsymbol{w}_t) \right] \boldsymbol{\phi}(s_t, a_t)$$

??

Q-learning + Linear FA

Recall Q-learning update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left[r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

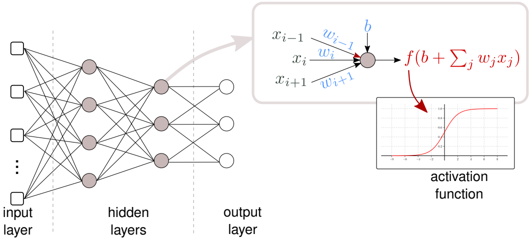
$t \rightarrow 0$
Initialize $\boldsymbol{w}$
Loop
  choose action a_t
  gather experience $\langle s_t, a_t, r_t, s_{t+1} \rangle$
  $U_t \leftarrow r_t + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s_{t+1}, a', \boldsymbol{w_t})$
  $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[U_t - \hat{Q}(s_t, a_t, \boldsymbol{w_t}) \right] \boldsymbol{\phi}(s_t, a_t)$
  $t \leftarrow t + 1$
EndLoop

(Linear) FA: Issues

- Linear FA+RL successfully applied on some tasks
- Nonlinear models (e.g., ANNs) have obtained significant results as well
  - e.g., TD-Gammon (1992)
- Efficacy of these approaches strongly depends on the features in use
  - how states (and actions) are represented
  - domain expertise necessary

Deep RL

- We have seen that the key advancement enabled by DNNs is the ability of learning the features
- Idea: exploiting this ability to learn suitable features for state and action representation



Deep Q Network

- First popular application of DNNs within RL in 2013
  - Mnih et al., "Playing Atari with Deep Reinforcement Learning"
    https://www.cs.toronto.edu/%7Evmnih/docs/dqn.pdf
- Task: playing Atari 2600 games
- Two key innovations:
  - DNN to approximate Q (Deep Q Network)
  - Experience Replay buffer
- Learning algorithm adapted from Q-learning

Example: Atari games

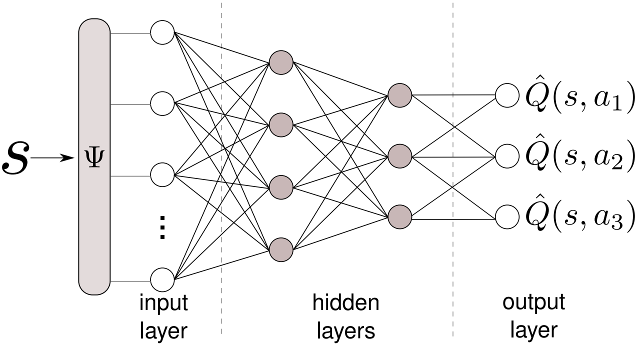
Atari 2600 console (1977–1992)

Example: Atari games

Breakout: `https://www.youtube.com/watch?v=TmPfTpjtdgg`

Deep Q Network

- **Input**: state s (possibly preprocessed)
- **Output**: $\hat{Q}(s, a)$, for every action a



Training

- NN training usually based on (large) training set
  - collection of examples $(\boldsymbol{x}_i, \boldsymbol{y}_i)$
- To train a DQN we would need many examples $(s_i, [Q(s_i, a_1) \cdots Q(s_i, a_n)]^T)$
- Problem: we don't have true examples of $Q(s, a)$ to use!
  - agent only collects immediate rewards on-line
- We need to estimate Q on-line based on experience (as usual in RL)

Training (2)

Experience

$\langle s_t, a_t, s_{t+1}, r_t \rangle$
$\langle s_{t-1}, a_{t-1}, s_t, r_{t-1} \rangle$
$\langle s_{t-2}, a_{t-2}, s_{t-1}, r_{t-2} \rangle$
$\ldots$

Training Sample

$(s_t, a_t) \to r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a', \boldsymbol{w})$
$(s_{t-1}, a_{t-1}) \to r_{t-1} + \gamma \max_{a'} \hat{Q}(s_t, a', \boldsymbol{w})$
$(s_{t-2}, a_{t-2}) \to r_{t-2} + \gamma \max_{a'} \hat{Q}(s_{t-1}, a', \boldsymbol{w})$

- Naive idea: pick mini-batches of last *b* experience tuples and train the NN
  - i.e., at each iteration, train on most recent experience
- sequential observations likely correlated X
- less recent experience possibly forgotten X

Experience Replay

- Smarter approach: experience replay buffer
- Circular FIFO buffer with capacity $B > b$
- At each training iteration, b tuples drawn randomly from the buffer
- correlation between observations reduced/removed ✓
- if B is large, old observations are “seen” more than once ✓
  - improved data efficiency

Deep Q-learning (DQL)

Initialize $\boldsymbol{w}$
Initialize empty buffer $\mathcal{B}$
$i \leftarrow 0$
Loop
  choose action a_i
  gather experience $\langle s_i, a_i, r_i, s_{i+1} \rangle$ and add to $\mathcal{B}$
  sample minibatch of b $\langle s_j, a_j, r_j, s_{j+1} \rangle$ tuples from $\mathcal{B}$
  $y^{(j)} \leftarrow r_j + \gamma \max_{a'} \hat{Q}(s_{i+1}, a', \boldsymbol{w}), j = 1, \ldots, b$
  $\mathcal{L}^{(j)} = (y^{(j)} - \hat{Q}(s_j, a_j, \boldsymbol{w}))^2$ /* Loss */
  update $\boldsymbol{w}$ using, e.g., SGD on the minibatch
  $i \leftarrow i + 1$
EndLoop

Example: Atari

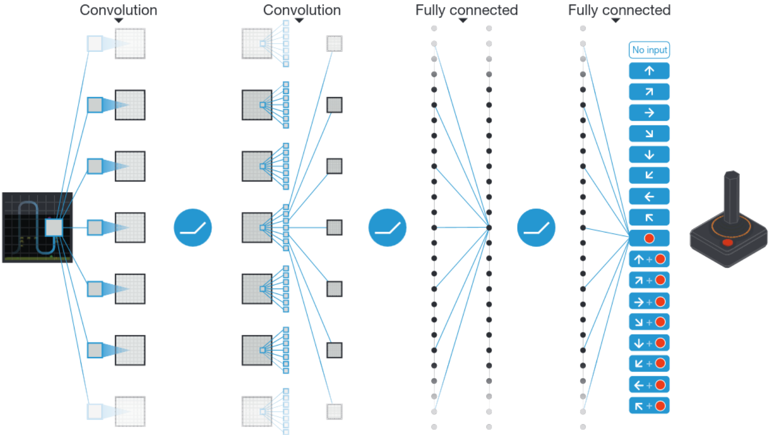
- Frames are 210×160 pixel images with a 128 color palette
- Input dimensionality reduced via preprocessing
  - RGB to gray-scale conversion
  - down-sampling to 110×84
  - cropped to 84x84 to ease implementation
- State comprises last 4 frames
  - why?

Example: Atari

- NN input: $84 \times 84 \times 4$ image produced by preprocessing
- Conv. layer with 16 8x8 filters with ReLU
- Conv. layer with 32 4x4 filters with ReLU
- Fully-connected layer with 256 ReLU units
- Linear output layer with one unit for each valid action (from 4 to 18 in the considered games)

- Trained using RMSProp for a total of 50 million frames (around 38 days of game experience in total)
- Replay memory stores 1 million most recent frames

Example: Atari



Target Network

- DQN may suffer from instability during training, possibly preventing the algorithm to converge
- In traditional NN training, the training targets do not change over time
- In DRL, since we don't have ground-truth Q values, we use the approximated $\hat{Q}$ in the update target value:

$$y^{(j)} \leftarrow r_j + \gamma \min_{a'} \hat{Q}(s_{i+1}, a', \boldsymbol{w})$$

- ......but we keep changing $\boldsymbol{w}$ at each iteration
- Let's use a second neural network to stabilize the targets

Deep Q-learning with Target Network

Initialize $\boldsymbol{w}$ and $\boldsymbol{w}^- = \boldsymbol{w}$
Initialize empty buffer $\mathcal{B}$
$i \leftarrow 0$
Loop
    choose action a_i
    gather experience $\langle s_i, a_i, r_i, s_{i+1} \rangle$ and add to $\mathcal{B}$
    sample minibatch of b $\langle s_j, a_j, r_j, s_{j+1} \rangle$ tuples from $\mathcal{B}$
    $y^{(j)} \leftarrow r_j + \gamma \min_{a'} \hat{Q}(s_{i+1}, a', \boldsymbol{w}^-), j = 1, \ldots, b$
    $\mathcal{L}^{(j)} = (y^{(j)} - \hat{Q}(s_j, a_j, \boldsymbol{w}))^2$
    update $\boldsymbol{w}$ using, e.g., SGD on the minibatch
    every C steps: $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$
    $i \leftarrow i + 1$
EndLoop

Remark

- DQN can seamlessly work with continuous state spaces
- Action space must be finite

Example: CartPole with DQN

- Environment provided by OpenAI Gym
  - Large collection of ready-to-use environments
- DQN implemented using TF-Agents
  - RL library part of Tensorflow ecosystem
- `https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial?hl=en`

Policy-based RL

- So far, we have considered value-based RL algorithms
  - Learn the value function; get a policy from it
- Now we turn our attention to policy-based RL (or, policy gradient methods)
  - Directly learn a policy
  - Algorithms may still learn the value function, but it is not used to derive the policy

Policy Gradient Methods

- Algorithms learn a parameterized policy

$$\pi(a|s, \boldsymbol{\theta}) = P(A_t = a | S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta})$$

  $\boldsymbol{\theta} \in \mathbb{R}^m$ is the vector of policy parameters

- $\pi(a|s, \boldsymbol{\theta})$ can be any function, as long as it is differentiable w.r.t. parameters $\boldsymbol{\theta}$

Note

To avoid ambiguity, we will keep using $\boldsymbol{w} \in \mathbb{R}^d$ to denote the vector of parameters used to approximate the value function, if necessary (e.g., $V(s, \boldsymbol{w})$)

Policy Gradient Methods (2)

- Suppose that $J(\boldsymbol{\theta})$ is a performance measure of the policy resulting from parameters $\boldsymbol{\theta}$ (the higher the better)
- To maximize performance, we can update $\boldsymbol{\theta}$ by gradient ascent:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla J(\boldsymbol{\theta}_t)$$

- The expression "policy gradient" refers to all the methods based on the idea introduced above

Policy Approximation: Why?

- Often (but not always), the policy is an easier function to approximate compared to the value function
- Policy parameterization lets action probabilities change smoothly as a function of the learned parameters, while they can change dramatically for a small change in the action values (if a different action gets the highest value)
  - stronger convergence guarantees are available
- Stochastic policies can be learned
- Continuous action spaces are supported

Policy Approximation via Action Preferences

- Let's suppose that the action space is discrete (and not too large)
  - We will discuss later other scenarios
- A natural choice for policy approximation is softmax in action preferences:

$$\pi(a|s, \boldsymbol{\theta}) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_{a'} e^{h(s,a',\boldsymbol{\theta})}}$$

- $h(s, a, \boldsymbol{\theta})$ is a parameterized numerical preference value for every state-action pair

Policy Approximation via Action Preferences

- Note that we don't need any specific strategy to determine the preference values $h(s, a, \boldsymbol{\theta})$
- They are just a convenient way to parameterize the policy $\pi(a|s, \boldsymbol{\theta})$
- For instance, we could use a DNN with a softmax output layer to approximate the policy
  - Hidden layers compute the action preferences
  - Output layer produces the policy probabilities

Action Preferences vs. Action Values

- Action values $Q(s, a)$ may differ by a small amount
  - Softmax based on Q may struggle to approach a deterministic policy (unless a very small temperature coefficient is used)
- Action preferences instead do not need to convergence to specific values (e.g., the optimal value function), but rather to the best values for the policy to learn
  - if a deterministic policy is optimal, preference for the optimal action will be as higher as possible than the other actions
  - if a stochastic policy is optimal, more than one action will have a high preference value (e.g., card games with incomplete information) ...
  - ...and we can learn *arbitrary* probabilities for actions

Policy Gradient in Episodic Tasks

- Let's consider an episodic task starting in state s_0
- In this case, performance of the policy can be evaluated as

$$J(\boldsymbol{\theta}) = V_{\pi_{\boldsymbol{\theta}}}(s_0)$$

- How to update $\boldsymbol{\theta}$ to improve performance?
- Performance depends both on (1) action selection and (2) the distribution of states occurring in the episode
- Both depend on the parameters!
- (2) is particularly difficult as it also depends on the environment

Policy Gradient Theorem

Policy Gradient Theorem

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a Q_\pi(s, a) \nabla_{\boldsymbol{\theta}} \pi(a|s, \boldsymbol{\theta})$$

- Proportionality constant is the average length of an episode
  - in gradient ascent, constant absorbed by step size α ✓
- we don't need the derivative of $\mu(s)$ ✓

Policy Gradient Theorem: Proof

To simplify notation, we leave it implicit that π is a function of $\boldsymbol{\theta}$, and that gradients are w.r.t. $\boldsymbol{\theta}$

$$\nabla V_\pi(s) = \nabla \left[\sum_a \pi(a|s) Q_\pi(s,a) \right] =$$

$$= \sum_a \nabla [\pi(a|s) Q_\pi(s,a)] =$$

$$= \sum_a [\nabla \pi(a|s) Q_\pi(s,a) + \pi(a|s) \nabla Q_\pi(s,a)] =$$

$$= \sum_a \left[\nabla \pi(a|s) Q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r'} p(s',r'|s,a)(r + V_\pi(s')) \right] =$$

Proof (2)

$$= \sum_a \left[\nabla \pi(a|s) Q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r'} p(s',r'|s,a)(r + V_\pi(s')) \right] =$$

1) Reward does not depend on $\boldsymbol{\theta}$ (gradient is 0)
2) $\sum_{s'} \sum_{r'} p(s',r'|s,a) V_\pi(s') = \sum_{s'} p(s'|s,a) V_\pi(s')$

$$= \sum_a \left[\nabla \pi(a|s) Q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \nabla V_\pi(s') \right] =$$

Note: we are computing $\nabla V_\pi(s)$ and now we have a recursive term $\nabla V_\pi(s')$! Let's unroll the recursion…

Proof (3)

$$= \sum_a \Big[\nabla \pi(a|s) Q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \cdot$$

$$\sum_{a'} \left(\nabla \pi(a'|s') Q_\pi(s',a') + \pi(a'|s') \sum_{s''} p(s''|s',a') \nabla V_\pi(s'') \right) \Big] =$$

After repeated unrolling …

$$= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} P(s \to x, k, \pi) \sum_a [\nabla \pi(a|x) Q_\pi(x,a)]$$

where $P(s \to x, k, \pi)$ is the probability of transitioning from s to x in k steps under policy π.

Proof (4)

We can now write an expression for the gradient of J

$$\nabla J(\boldsymbol{\theta}) = \nabla V_\pi(s_0) = \sum_s \sum_{k=0}^{\infty} P(s_0 \to s, k, \pi) \sum_a [\nabla \pi(a|s) Q_\pi(s,a)] =$$

$$= \sum_s \eta(s) \sum_a [\nabla \pi(a|s) Q_\pi(s,a)] =$$

$\eta(s)$: avg. number of steps spent in s within an episode

$$= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a [\nabla \pi(a|s) Q_\pi(s,a)] =$$

Proof (5)

$$= \sum_{s'} \eta(s') \sum_{s} \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_{a} [\nabla \pi(a|s) Q_{\pi}(s, a)] =$$

$$= \sum_{s'} \eta(s') \sum_{s} \mu(s) \sum_{a} [\nabla \pi(a|s) Q_{\pi}(s, a)]$$

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s} \mu(s) \sum_{a} [\nabla \pi(a|s) Q_{\pi}(s, a)]$$

REINFORCE

- $\mu(s)$ is the on-policy distribution of states under π
  - if π is followed, states will occur in that proportion

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a Q_\pi(s, a) \nabla_{\boldsymbol{\theta}} \pi(a|s, \boldsymbol{\theta}) =$$
$$= E_\pi \left[\sum_a Q_\pi(s_t, a) \nabla_{\boldsymbol{\theta}} \pi(a|s_t, \boldsymbol{\theta}) \right]$$

- as we did to approximate the value function, we can perform a stochastic gradient ascent on occurring states s_t

REINFORCE (2)

- We replaced a sum over states with an expectation under π
- We want to do the same with the sum over actions
- First, we need to weigh actions by $\pi(a|s, \boldsymbol{\theta})$

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto E_{\pi}\left[\sum_a Q_{\pi}(s_t, a) \nabla_{\boldsymbol{\theta}} \pi(a|s_t, \boldsymbol{\theta})\right] &= \\
= E_{\pi}\left[\sum_a \pi(a|s_t, \boldsymbol{\theta}) Q_{\pi}(s_t, a) \frac{\nabla_{\boldsymbol{\theta}} \pi(a|s_t, \boldsymbol{\theta})}{\pi(a|s_t, \boldsymbol{\theta})}\right] &= \\
= E_{\pi}\left[Q_{\pi}(s_t, a_t) \frac{\nabla_{\boldsymbol{\theta}} \pi(a_t|s_t, \boldsymbol{\theta})}{\pi(a_t|s_t, \boldsymbol{\theta})}\right] &= \\
= E_{\pi}\left[G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(a_t|s_t, \boldsymbol{\theta})}{\pi(a_t|s_t, \boldsymbol{\theta})}\right]
\end{aligned}$$

REINFORCE (3)

$$\nabla J(\boldsymbol{\theta}) \propto E_\pi \left[G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(a_t | s_t, \boldsymbol{\theta})}{\pi(a_t | s_t, \boldsymbol{\theta})} \right]$$

- We can sample G_t for each time step, and we got an expression proportional to the gradient
- We can use stochastic gradient ascent to update parameters
- The resulting algorithm is REINFORCE (1992)

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha G_t \nabla_{\boldsymbol{\theta}} \ln \pi(a_t | s_t, \boldsymbol{\theta})$$

$$\frac{\nabla_{\boldsymbol{\theta}} \pi(a_t | s_t, \boldsymbol{\theta})}{\pi(a_t | s_t, \boldsymbol{\theta})} = \nabla_{\boldsymbol{\theta}} \ln \pi(a_t | s_t, \boldsymbol{\theta})$$

REINFORCE: Algorithm

We also include a discount factor γ

Initialize $\boldsymbol{\theta}$ (e.g., to 0)
Loop
    generate episode $s_0, a_0, r_1, s_1, \ldots, r_T$ following π
    for t=0,1,...,T **do**
        $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G_t \nabla_{\boldsymbol{\theta}} \ln \pi(a_t | s_t, \boldsymbol{\theta})$
    end
EndLoop

Policy Gradient: Another Perspective

- Consider a NN used for multi-class classification
- Softmax final layer outputs a probability y_c for all classes c
- Gradient of cross-entropy loss used for training

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} \left[\sum_c \bar{y}_c \ln y_c \right]$$

- Intuitively, training will increase prob. y_c for the class c labeled as correct (ground truth)
  - likelihood maximization
- Replacing classes with "actions", the NN outputs $\pi(a|s, \boldsymbol{\theta})$
- REINFORCE update is proportional to $\nabla_{\boldsymbol{\theta}} \ln \pi(a_t|s_t, \boldsymbol{\theta})$
  - without a ground truth, prob. is increased/decreased based on return

REINFORCE with Baseline

- A slight generalization of REINFORCE involves the use of a baseline $b(s)$
- We compare the value of each action to $b(s)$
- The policy gradient theorem remains true

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s} \mu(s) \sum_{a} \left(Q(s,a) - b(s)\right) \nabla \pi(a|s,\boldsymbol{\theta})$$

- Baseline can be useful to reduce the variance of the update and speed learning

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left(G_t - b(s_t)\right) \nabla \ln \pi(a_t|s_t,\boldsymbol{\theta})$$

Actor-Critic

- Idea to avoid high variance of returns used by REINFORCE
- Using the one-step return $G_{t:t+1}$ instead

$$G_{t:t+1} = r_t + \gamma V(s_{t+1})$$

- We call critic the role of the value function used in this way
- We call the resulting approach actor-critic

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left(G_{t:t+1} - \hat{V}(s_t, \boldsymbol{w}) \right) \nabla \ln \pi(a_t | s_t, \boldsymbol{\theta}) =$$

$$= \boldsymbol{\theta}_t + \alpha \left(r_{t+1} + \gamma \hat{V}(s_{t+1}, \boldsymbol{w}) - \hat{V}(s_t, \boldsymbol{w}) \right) \nabla \ln \pi(a_t | s_t, \boldsymbol{\theta})$$

Actor-Critic (2)

Initialize $\boldsymbol{\theta}$ and $\boldsymbol{w}$ (e.g., to 0)

Loop

  Initialize s as first state of the episode

  $I \leftarrow 1$

  while *s not terminal* **do**

    choose action a according to $\pi(\cdot|s, \boldsymbol{\theta})$

    observe s' and r

    $\delta \leftarrow r + \gamma \hat{V}(s', \boldsymbol{w}) - \hat{V}(s, \boldsymbol{w})$

    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{V}(s, \boldsymbol{w})$

    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + I \alpha^{\boldsymbol{\theta}} \delta \nabla \ln \pi(a|s, \boldsymbol{\theta})$

    $s \leftarrow s'$

  end

EndLoop

The Continuing Case

- Policy gradient theorem holds for continuing tasks as well
- The performance measure $J(\boldsymbol{\theta})$ must be changed to the average reward
- Proof and updated Actor-Critic alg. in Sutton-Barto, 13.6

Continuous or Large Action Spaces

- Policy gradient methods can be useful in presence of very large or continuous action spaces
  - Computing the learned probability for every action can be expensive/unfeasible
- The solution: learn the parameters of a probability distribution, instead of the probability of choosing each action
- Example: policy defined as the normal probability density

$$\pi(a|s, \boldsymbol{\theta}) = \frac{1}{\sigma(s, \boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(a - \mu(s, \boldsymbol{\theta})^2)}{2\sigma(s, \boldsymbol{\theta})^2}\right)$$

Example

CartPole with Actor-Critic in TensorFlow:
https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic?hl=en

Advanced Policy Methods

- Deterministic Policy Gradient (DPG): similar to the theorem proven above, but for deterministic policies
  - Silver et al. (2014), "Deterministic Policy Gradient Algorithm", `http://proceedings.mlr.press/v32/silver14.pdf`
- Deep Deterministic Policy Gradient (DDPG): DPG with DNNs
  - Lillicrap et al. (2016), "Continuous control with deep reinforcement learning", `https://arxiv.org/abs/1509.02971`
- ...
- Proximal Policy Optimization (PPO): state-of-the-art algorithm
  - Schulman et al., "Proximal Policy Optimization Algorithms", `https://arxiv.org/pdf/1707.06347.pdf`

Advanced RL Topics

- Multi-agent RL
- Hierarchical RL
- RL + Heuristic Tree Search
- Transfer RL
- ...