UNIVERSIDAD DE GRANADA

DECSAI

Reinforcement Learning: DRL

September 25th – 29th, 2023

E.T.S. de Ingenierías Informática y de Telecomunicación

Universidad de Granada

Manuel Pegalajar Cuellar / Juan Gómez Romero

manupc@ugr.es / jgomez@ugr.es

Departamento de Ciencias de la Computación e Inteligencia Artificial

http://decsai.ugr.es

UNIVERSIDAD DE GRANADA

Reinforcement Learning:
Deep Reinforcement Learning

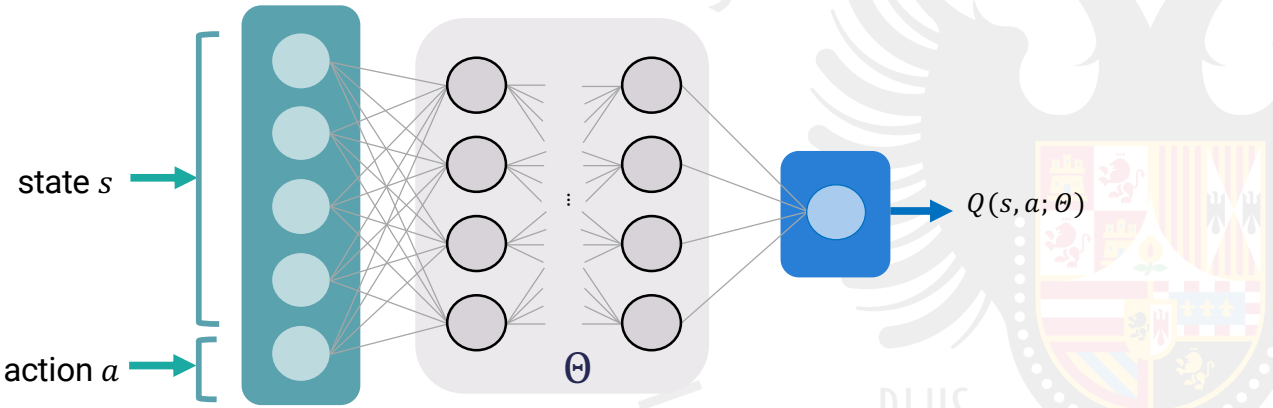
Contents

DECSAI

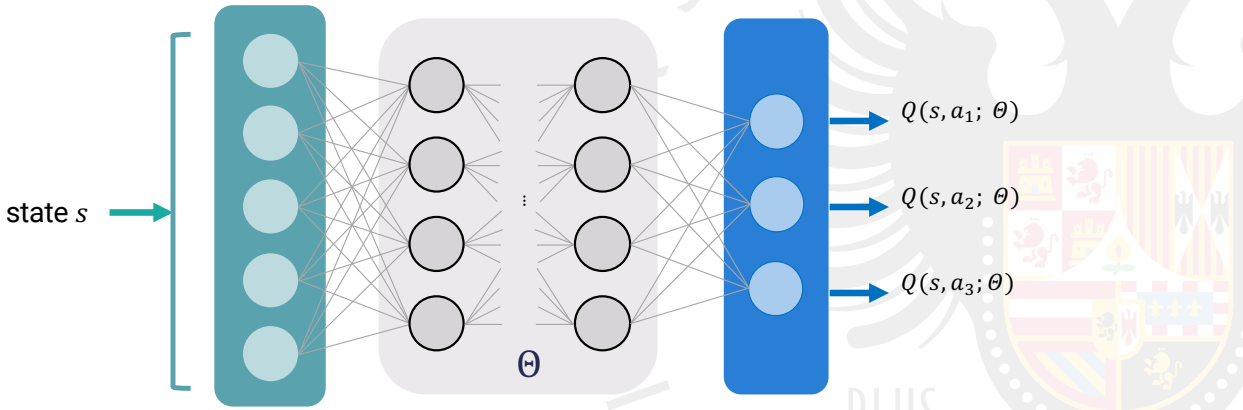
What we know so far

- Fundamental concepts: state, action, reward, policy
- Formulation of a Markov Decision Process
- The value function *V(s)* and state-value function *Q(s, a)*
- Value iteration algorithm
- Q-Learning algorithm
- **Feed forward neural networks**
- **Training loop**
- **PyTorch fundamentals**

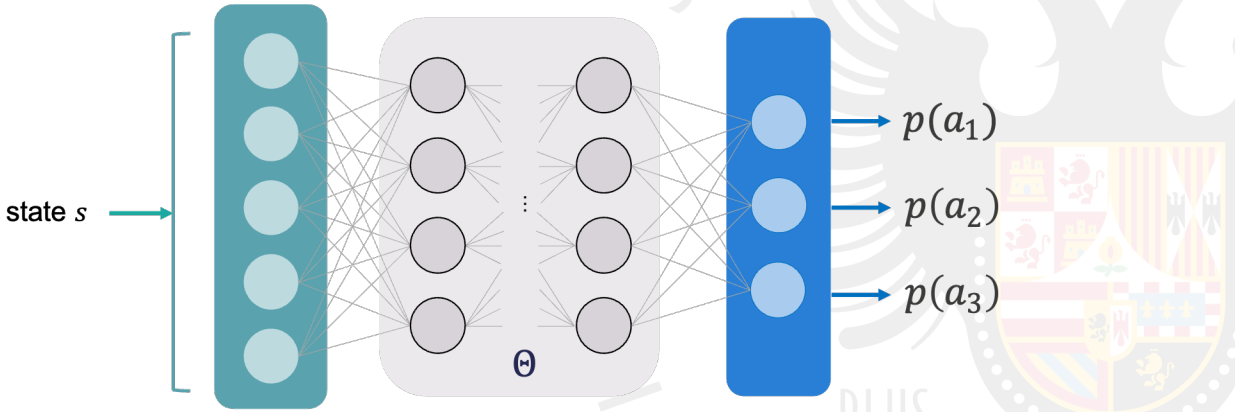
Cliff Walking



Value function approximation



Policy approximation



UNIVERSIDAD DE GRANADA

Reinforcement Learning:
Deep Reinforcement Learning

Contents

DECSAI

Open AI Gym / Gymnasium

Open in Colab

https://github.com/jgromero/rl_seminar_2023/tree/main/code

cliffwalking-qlearning.ipynb

```python
import gymnasium as gym

env = gym.make("CliffWalking-v0", render_mode="ansi")   # change to "human" in local mode

observation, info = env.reset(seed=42)

for i in range(100):
  action = env.action_space.sample()  # this is where you would insert your policy

  observation, reward, terminated, truncated, info = env.step(action)

  if terminated or truncated:
    observation, info = env.reset()

env.close()
```

Q-Learning

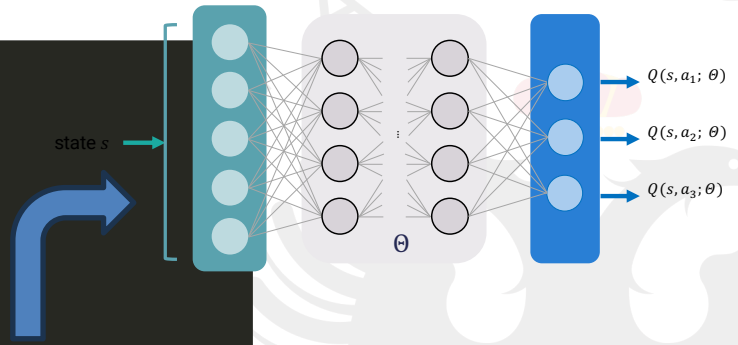
$$\bar{Q}(s,a) \leftarrow \bar{Q}(s,a) + \alpha\Big(r + \gamma \bar{Q}(s',a') - \bar{Q}(s,a)\Big)$$

```
while True :
    # choose action (off policy)
    action = generate_action(env, Q, epsilon, state)

    # apply action and update episode score
    next_state, reward, terminated, _, _ = env.step(action)
    episode_score += reward

    # update Q
    Q[state][action] = Q[state][action] +
                       alpha * (reward + gamma * np.max(Q[next_state]) - Q[state][action])

    # check end of episode
    n_step += 1
    if terminated or n_step > step_limit:
        tmp_scores.append(episode_score)
        break

    # get next state
    state = next_state
```

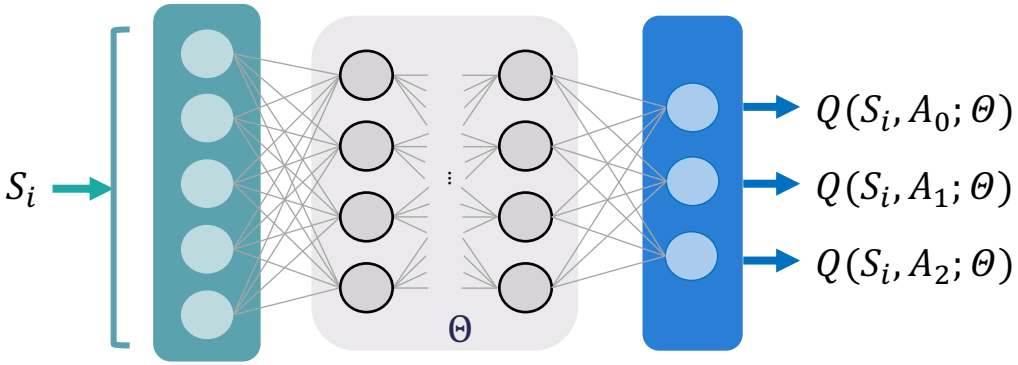
Deep Q-Learning

$$\bar{Q}(s,a) \leftarrow \bar{Q}(s,a) + \alpha\Big(r + \gamma \bar{Q}(s',a') - \bar{Q}(s,a)\Big)$$

```
while True :
    # choose action (off policy)
    action = generate_action(env, Q, epsilon, state)

    # apply action and update episode score
    next_state, reward, terminated, _, _ = env.step(action)
    episode_score += reward

    # update Q
    Q[state][action] = Q[state][action] +
                       alpha * (reward + gamma * np.max(Q[next_state]) - Q[state][action])

    # check end of episode
    n_step += 1
    if terminated or n_step > step_limit:
        tmp_scores.append(episode_score)
        break

    # get next state
    state = next_state
```



Deep Q-Learning



Training

Input: S_i

Output: $Q(S_i, A_j; \Theta)$

Loss function: $\mathcal{L}\big(Q(S_i, A_j), Q(S_i; \Theta)\big)$

We don't know this for sure!

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

But we can make a guess from samples

$S_0\ A_0\ R_1\ S_1$

Deep Q-Learning

A bit of math…

- Loss function for a given pair (S, A): $\ell\left(x^{(i)}, \Theta\right) = \mathcal{L}(Q(S), Q(S, A; \Theta))$
- MSE expectation given policy π: $\mathrm{E} = \mathbb{E}_{\pi}\left[\left(Q(S, A) - Q(S, A; \Theta)\right)^2\right]$
- Remember that: $\Delta\Theta = \Delta w = -\eta \frac{\partial E}{\partial w}$
- Develop the derivative: $\frac{\partial E}{\partial w} = 2\left(Q(S, A) - Q(S, A; \Theta)\right) \frac{\partial Q(S,A;\Theta)}{\partial w}$

gradient of the weights: ∇_{θ}

- Replace: $\Delta\Theta = -2\eta \left(Q(S, A) - Q(S, A; \Theta)\right) \nabla_{\Theta} Q(S, A; \Theta)$
- Apply Q-Learning expression and reduce:

$$\Delta\Theta = \alpha\left(R + \gamma \max_{a \in \mathcal{A}} Q(S', a; \Theta) - Q(S, A; \Theta)\right) \nabla_{\Theta} Q(S, A; \Theta)$$

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

Deep Q-Learning

Meaning… what?

The weights update rule is a bit different from the typical one in deep neural networks

$$\Delta\Theta = \alpha\left(R + \gamma \max_{a\in\mathcal{A}} Q(S', a; \Theta) - Q(S, A; \Theta)\right)\nabla_\Theta\ Q(S, A; \Theta)$$

We can take a mini-batch of experiences $S_t\ A_t\ R_{t+1}\ S_{t+1}$ from the replay buffer to update the Q network

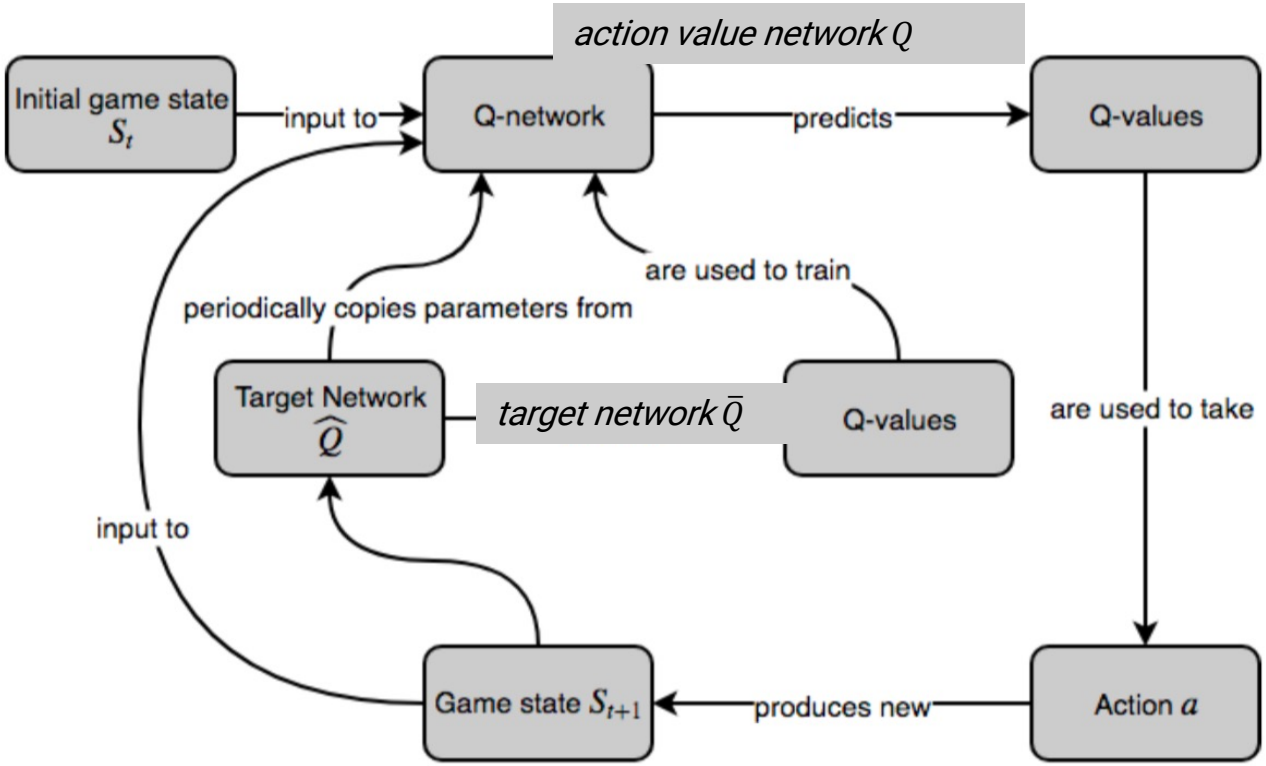
We are approximating Q with an approximation. Therefore, DQN introduces a second $\bar{Q}$ network for the action-value estimations, with the same topology but weights Θ^- are less frequently updated from Θ

$$\Delta\Theta = \alpha\left(R + \gamma \max_{a\in\mathcal{A}} \bar{Q}(S', a; \Theta^-) - Q(S, A; \Theta)\right)\nabla_\Theta\ Q(S, A; \Theta)$$

DQN > Replay memory
Database of sample interactions

DQN > Double network
Avoid "moving goalpost" in training

DQN algorithm



A. Zai, B. Brown (2018) **Deep Reinforcement Learning in Action**. Manning.

DQN algorithm

V. Mnih et al. (2015) *Human-level control through deep reinforcement learning.* Nature 518, 529-533

Deep Mind DQN (2016). Más: https://deepmind.com/research/dqn/

DQN algorithm

Deep Q-Learning Algorithm

params: Q, $\bar{Q}$, D, n_episodes, n_steps, batch_size, γ, C, τ

for i = {0, …, n_episodess}

for t = {0, …, n_steps}

sample

- select action A_t with ϵ-greedy(Q) policy
- apply A_t and retrieve R_{t+1}, S_{t+1}
- store $S_t\, A_t\, R_{t+1}\, S_{t+1}$ in replay buffer D

train Q

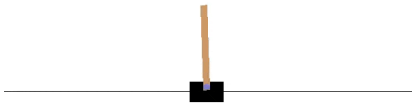
- sample an experience mini-batch $\langle S_j\, A_j\, R_{j+1}\, S_{j+1} \rangle$ of size *batch_size* from D
- get the predicted Q for each experience with Q-Learning expression as:

$$y_j = \begin{cases} R_j \\ R_j + \gamma \max_{a \in \mathcal{A}} \bar{Q}(S_{j+1}, a; \Theta^-) \end{cases}$$ (if the episode ends in *j+1, or otherwise*)

- gradient descent over $\left(y_j - Q(S_j, A_j; \Theta)\right)^2$

update $\bar{Q}$

- every C iterations, update $\bar{Q}$ weights with Q ; e.g., $\Theta^- = \tau\, \Theta^- + (1 - \tau)\, \Theta$



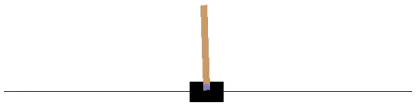
Example: learning to control the cart pole

```
for t in range(max_t):

    # choose action At with e-greedy policy
    action = agent.act(state, eps)

    # apply At and get Rt+1, St+1
    next_state, reward, done, _, _ = env.step(action)

    # store <St, At, Rt+1, St+1>
    agent.memory.add(state, action, reward, next_state, done)

    # train & update
    agent.step(state, action, reward, next_state, done)

    # advance to next state
    state = next_state
    score += reward

    if done:
        break
```

Open in Colab

https://github.com/jgromero/rl_seminar_2023/tree/main/code

cartpole-dqn.ipynb



Example: learning to control the cart pole

```
# Get max predicted Q values (for next states) from target model
# - qnetwork_target : apply forward pass for the whole mini-batch
# - detach : do not backpropagate
# - max : get maximizing action for each sample of the mini-batch (dim=1)
# - [0].unsqueeze(1) : transform output into a flat array
Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

# Compute Q targets for current states (y)
# - dones : detect if the episode has finished
Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

# Get expected Q values from local model (Q(Sj, Aj, w))
# - gather : for each sample select only the output value for action Aj
Q_expected = self.qnetwork_local(states).gather(1, actions)

# Optimize over (yj-Q(Sj, Aj, w))^2
# * compute loss
loss = F.mse_loss(Q_expected, Q_targets)
# * minimize the loss
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

DQN and beyond

Convergence

Z. Yang, Y. Xie, Z. Wang (2019) *A Theoretical Analysis of Deep Q-Learning.* https://arxiv.org/abs/1901.00137v2
S.J. Lee (2019) *The Deep Q-Network Book.* https://www.dqnbook.com

Double DQN with variable update period

DQN tends to overestimate the $\bar{Q}$ at the first training episodes **>>** Use a variable C parameter to update Θ^-

Priority experience replay memory
The replay memory has a fixed length and old experiences may be lost to store new experiences **>>** Keep old *valuable* experiences
Replay buffer is randomly sample (uniform) **>>** Use a biased distribution

Dueling networks
Use different architectures for each network, such as: $V(s) = Adv(s) + Q(s,a)$, whereas $Adv(s)$ is the *advantage* function

Extensions
Distributional DQN, Noisy DQN, Rainbow

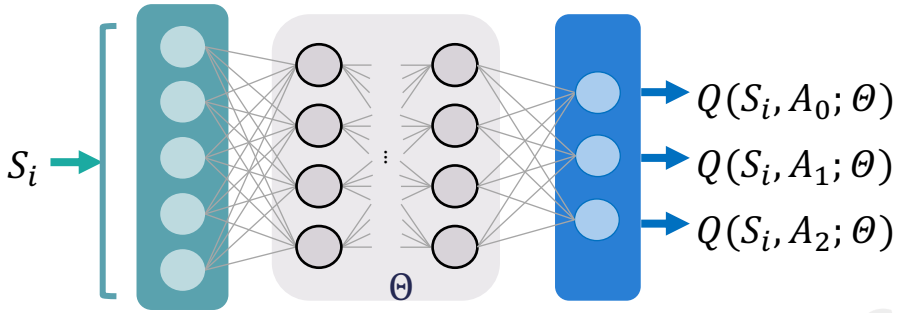
UNIVERSIDAD
DE GRANADA

Reinforcement Learning:
Deep Reinforcement Learning

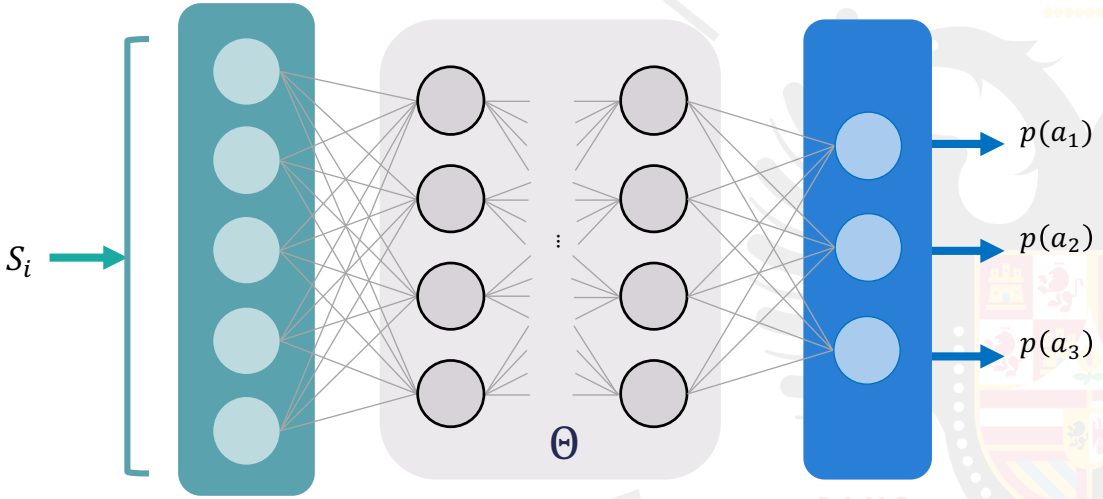
Contents

DECSAI

DQN
Getting π_* from Q



Policy-gradient
Getting π_* from Θ



Hill climbing

Hill-Climbing Algorithm

params: Θ, S_0

$G_{max} \leftarrow 0$, $\Theta_{max} \leftarrow \Theta$

while not solved
    run episode and generate trajectory according to policy π_Θ
    calculate gain reward G for the trajectory from S_0
    update G_{max}, Θ_{max}
    modify randomly Θ

Improvements

- Do not modify Θ at random
- Use Θ_{max} to generate better trajectories
- Apply optimization techniques
  - Simulated annealing
  - Adaptive noise
  - Evolutionary computation
  - **Define a loss function and apply gradient descent**

REINFORCE

"Maximum reward" function to optimize $U(\Theta)$:

$$U(\Theta) = \sum_x P(x;\Theta)R(x)$$

$U(\Theta)$ gradient:

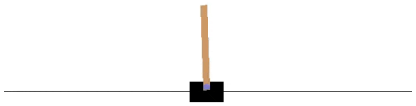
$$\nabla_\Theta U(\Theta) \approx \frac{1}{K}\sum_{i=1}^{K}\sum_{t=1}^{n} \nabla_\Theta \, log\pi_\Theta \left(A_t^{(i)} \middle| S_t^{(i)}\right) R(x^{(i)})$$

R.S. Sutton, A.G. Barto (2018) **Reinforcement Learning**. Chapter 13: Policy Gradient Methods. MIT Press.

REINFORCE Algorithm

params: Θ

while not solved
    generate K trajectories $x^{(i)}$ with length n according to policy π_Θ
    calculate the reward R^i for each $x^{(i)}$
    update Θ by applying gradient over $U(\Theta)$



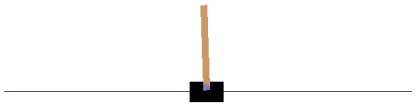
Example: learning to control the cart pole

```
for t in range(max_t):

    # choose action At with the policy network
    action, log_prob = policy.act(state)

    # save actions' probabilities
    saved_log_probs.append(log_prob)

    # apply At and get Rt+1, St+1
    next_state, reward, done, _, _ = env.step(action)

    # store reward
    rewards.append(reward)

    # advance to next state
    state = next_state
    score += reward

    if done:
        break
```

Open in Colab

https://github.com/jgromero/rl_seminar_2023/tree/main/code

cartpole-reinforce.ipynb



Example: learning to control the cart pole

```
    # calculate discounted reward for the episode
    discounts = [gamma**i for i in range(len(rewards)+1)]
    R = sum([a*b for a,b in zip(discounts, rewards)])

    # train & update
    policy_loss = []
    for log_prob in saved_log_probs:
        policy_loss.append(-log_prob * R)
    policy_loss = torch.cat(policy_loss).sum()

    optimizer.zero_grad()
    policy_loss.backward()
    optimizer.step()
```

UNIVERSIDAD DE GRANADA

Reinforcement Learning:
Deep Reinforcement Learning

Contents

DECSAI

Actor-critic approach

Policy-gradient methods (REINFORCE)

Calculate the probability that an action will lead to a good result (ACT)

Problem: a trajectory leading to a bad result may contain good actions

+

Value-based methods (DQN)

Guess the value of the expected reward after taking an action in a state (ESTIMATE)

Problem: estimations are built from estimations, which may make training unstable and biased

=

Actor-critic methods

Combination of both approaches

Two networks:

actor (similar to the REINFORCE network - π)

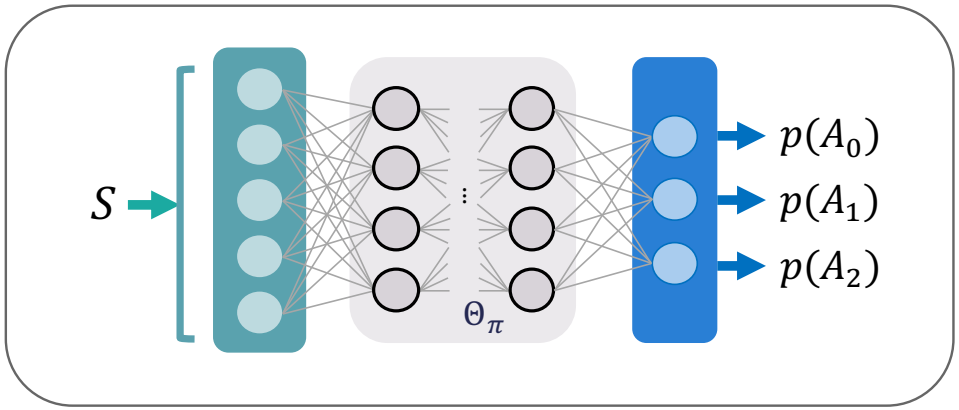
critic (similar to the DQN network - V)

Advantage: faster, more stable and less biased training

Actor-critic architecture

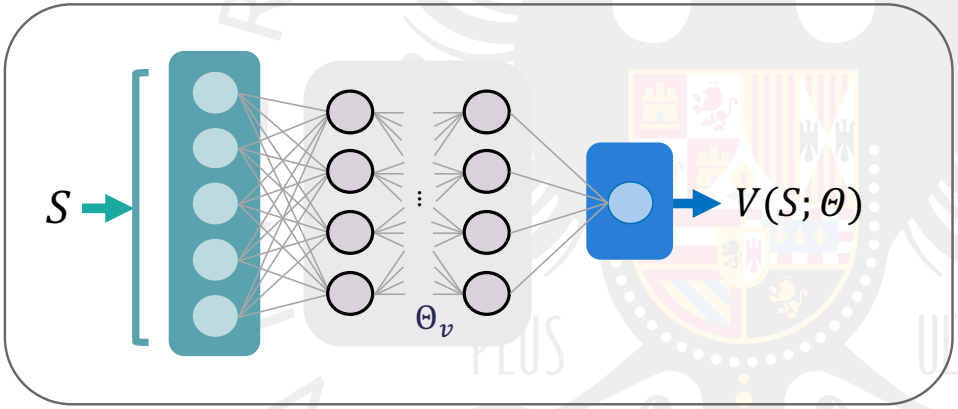
actor

$\pi(A|S;\Theta_\pi)$

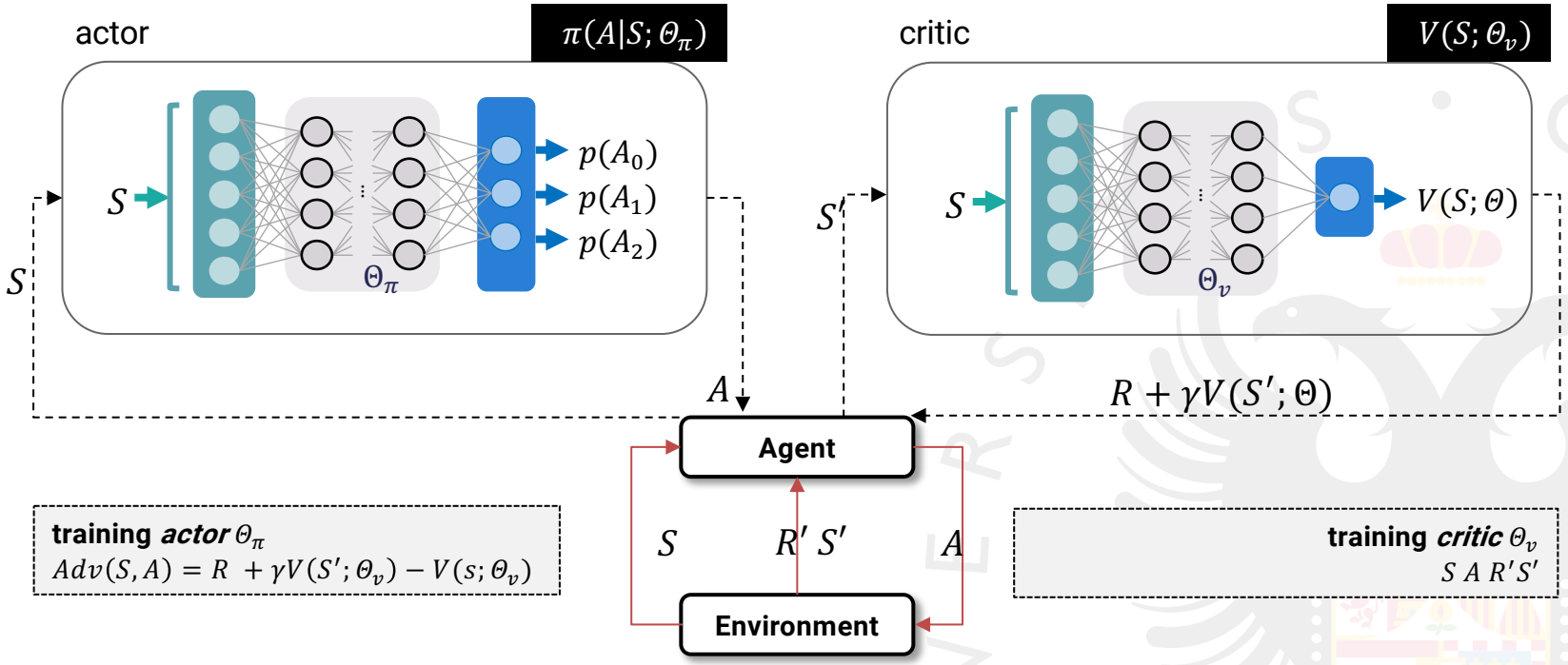


critic

$V(S;\Theta_v)$



Actor-critic algorithm



Actor-critic and beyond

Parallel & asynchronous architectures with multiple agents

- *A3C : Asynchronous Advantage Actor-Critic* (2016)
  Each agent has their copy of the *actor-critic* network
  Periodically, each agent asynchronously updates a global *actor-critic* network
- *A2C : Advantage Actor-Critic (2016)*
  The agents synchronously update the global *actor-critic* network

V. Mnih et al. (2016) *Asynchronous Methods for Deep Reinforcement Learning.*
https://arxiv.org/abs/1602.01783

Extensions for continuous action spaces

- *DDPG : Deep Deterministic Policy Gradient Continuous Action Space* (2015)

The actor network outputs a continuous action, the critic estimates Q values

T. Lillicrap et al. (2015) *Continuous control with Deep Reinforcement Learning.*
https://arxiv.org/abs/1509.02971

Actor-critic and beyond

State of the art algorithms

- *PPO : Proximal Policy Optimization* (2017)

  REINFORCE but replacing U with Adv

  J. Schulman et al. (2017) *Proximal Policy Optimization Algorithms.* https://arxiv.org/abs/1707.06347

- *SAC : Soft Actor Critic* (2017)

  Actor critic architecture but with two value networks for Q and V

  T. Haarnoja et al. (2017) *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.* https://arxiv.org/abs/1801.01290

- *TD3 : Twin Delayed DDPG* (2018)

  DDPG with double Q-network, delayed policy updates (not only for Q) and Q value smoothing

  S. Fujimoto et al. (2010) *Addressing Function Approximation Error in Actor-Critic Methods .* https://arxiv.org/abs/1802.09477

UNIVERSIDAD DE GRANADA

Reinforcement Learning:
Deep Reinforcement Learning

Contents

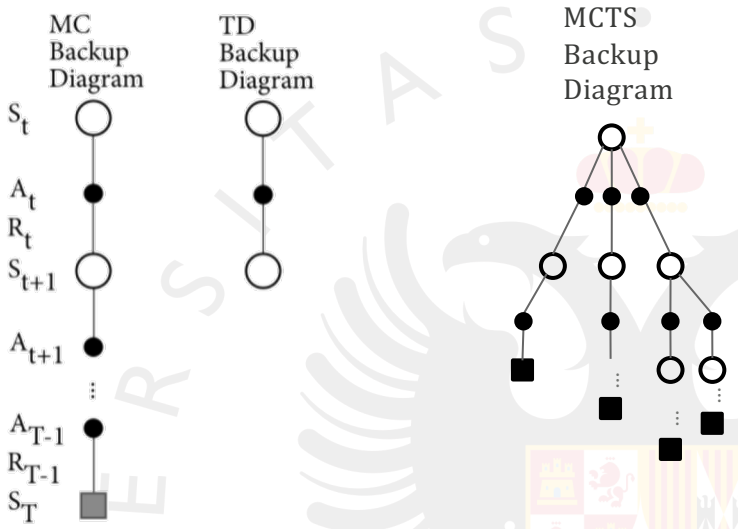
DECSAI

Principles

DRL agent that defeated humans playing *Go*

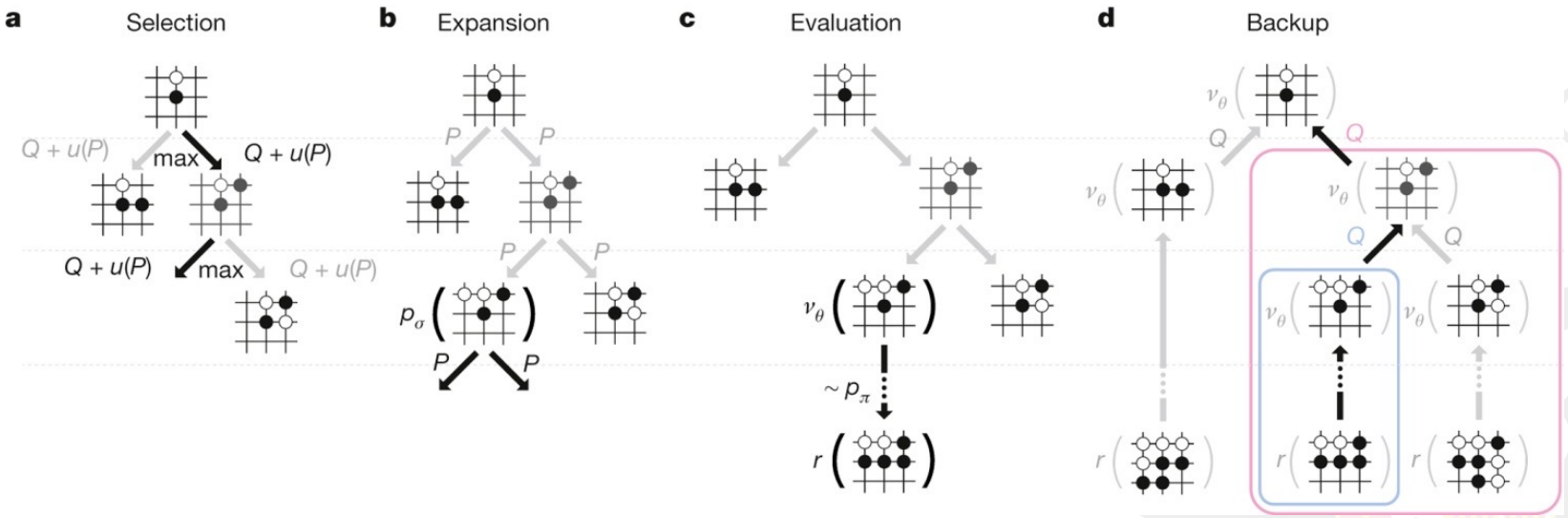
Approach

Game resolution with search trees
+
Deep reinforcement learning
+
Rollout using Monte Carlo Tree Search (MCTS)



D. Silver et al. (2016) *Mastering the game of Go with Deep Neural Networks & Tree Search.* Nature 529, 484-489

Principles



D. Silver et al. (2016) *Mastering the game of Go with Deep Neural Networks & Tree Search.* Nature 529, 484-489

M. Pumperla, K. Ferguson (2019) *Deep Learning and the Game of Go*. Manning.

AlphaZero (2017)

Learn by self-play

D. Silver et al. (2017) *A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play.* Science 362(6419), 1140-1144

AlphaStar (2019)

Can defeat humans in StarCraft 2.0

O. Vinyals et al. (2019) *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II.* https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/

OpenAI DOTA 2 (2019)

Can defeat humans in DOTA 2

J. Pachocki et al. (2019) *OpenAI Five.* https://openai.com/five/

Pluribus (2019)

Can defeat humans in Poker

N. Brown, T. Sandholm (2019) *Superhuman AI for multiplayer poker.* Science 365(6456), 885-890.