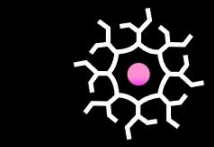
Introdução ao

Reinforcement Learning

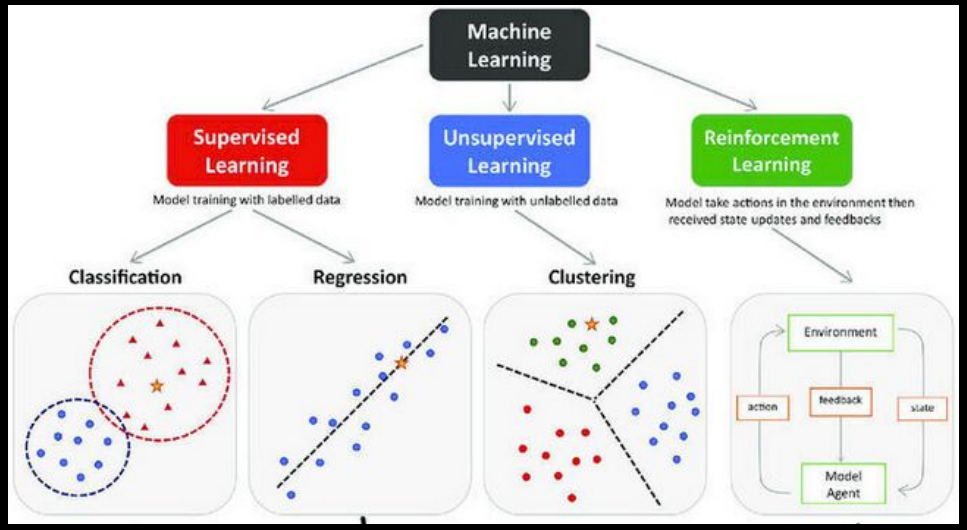
Github do curso: https://github.com/ronanft/rl-course

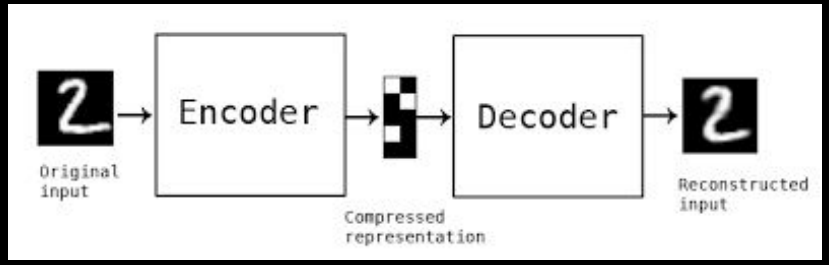


ACTION
CODE

Introdução ao Reinforcement Learning

• Tipos de aprendizado de máquina: supervisionado, não-supervisionado, por reforço.





Introdução ao Reinforcement Learning

- Peculiaridades do RL:
  - feedback por tentativa (versus feedback por *target annotation*).
  - dependência de ambiente
  - retorno retardado
  - *exploration vs. exploitation*
  - *moving target*

Introdução ao Reinforcement Learning

- Aplicações de RL:

robótica, recomendação (produtos, anúncios, etc.), mercado financeiro, llm, etc.

Referências

Livro:
Sutton, Barto. http://incompleteideas.net/book/RLbook2020.pdf

Github:
https://github.com/aikorea/awesome-rl

https://gibberblot.github.io/rl-notes/index.html

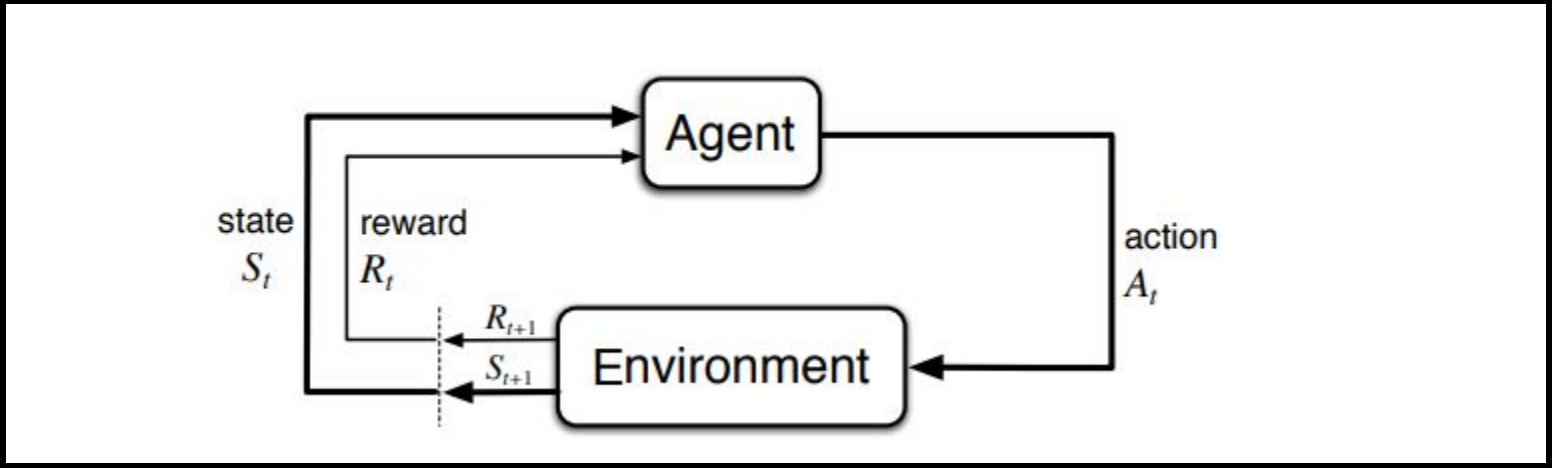
https://github.com/dennybritz/reinforcement-learning

Vídeo:
https://www.youtube.com/watch?v=14BfO5lMiuk

Curso!!!
https://huggingface.co/learn/deep-rl-course/unit0/introduction

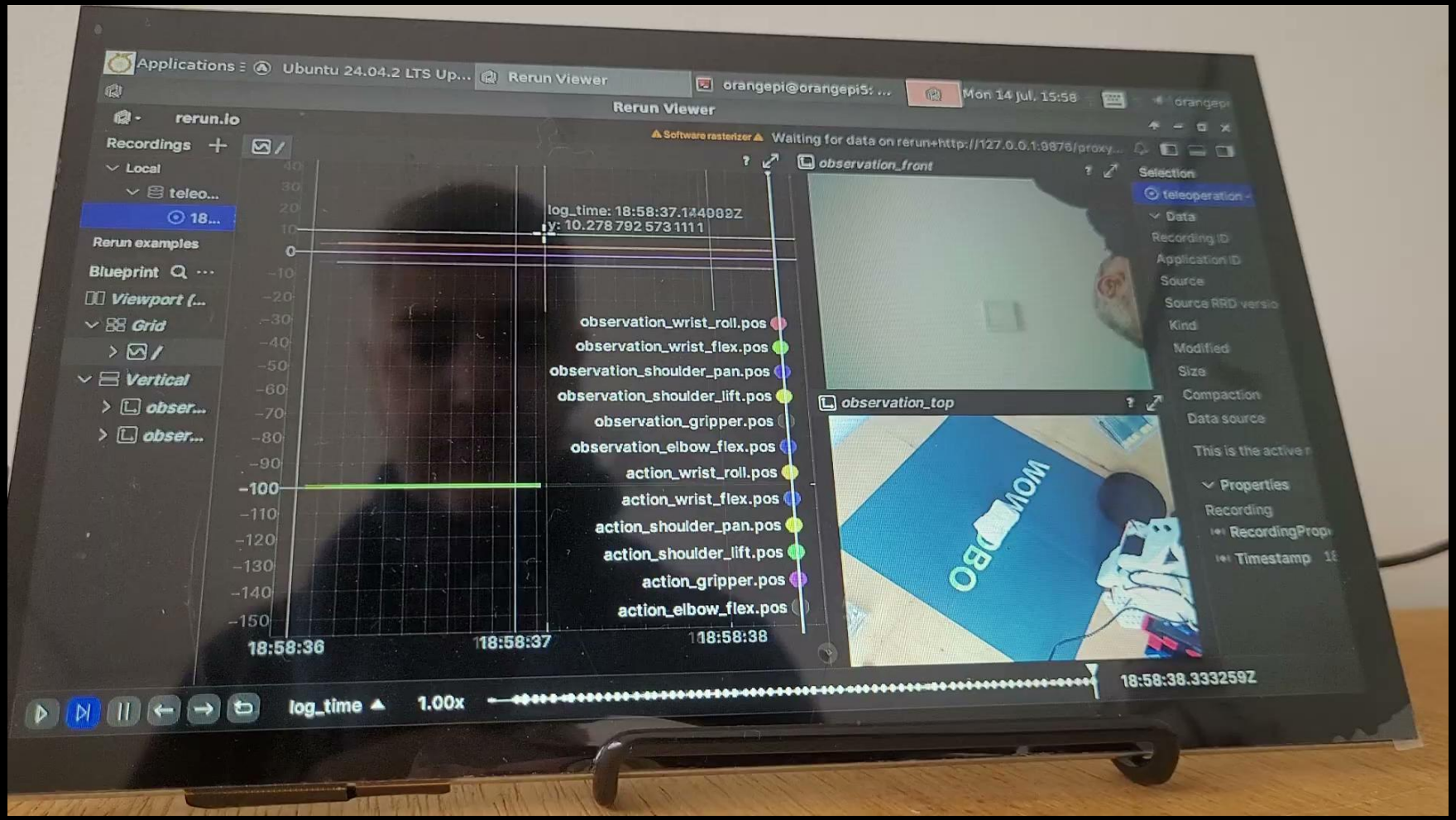
Elementos de RL

- Agente: quem aprende
- Ambiente: onde ocorre a interação (environment)
- Estado ou observação: o que o agente percebe (state)
- Recompensa: o sinal de feedback (reward)
- Taxa de desconto (γ): peso das recompensas futuras (discount rate)



Elementos de RL

O que é observação e o que é ação nesse contexto?



Interação Agente-Ambiente

- Em cada passo t:
  - O agente observa o estado s_t
  - Escolhe uma ação a_t
  - O ambiente devolve nova observação s_{t+1} e recompensa r_{t+1}

Objetivo: aprender uma política ótima que maximize a soma de TODAS as recompensas (não apenas a imediata!).

Loop OpenAI Gym (agora Farama's Gymnasium)

```
episodes = 5
for episode in range(1, episodes+1):
    state = env.reset()
    done = False
    score = 0

    while not done:
        env.render()
        action = random.choice([0,1,2,3,4,5])
        n_state, reward, done, info = env.step(action)
        score+=reward
    print('Episode:{} Score:{}'.format(episode, score))
env.close()
```

16] ⊗ 0.5s

Dynamic Programming

- Requer modelo completo do ambiente (transições e recompensas)
- Requer múltiplas iterações por todos os estados (e ações)
- Algoritmos: *GPI (General Policy Iteration: evaluation* e *improvement) ou Value Iteration*
- Usa *Bellman Equations* para avaliar e melhorar políticas

Bellman Equations

MDP

MDP é premissa das equações: O próximo estado depende apenas do estado atual e da ação, não do histórico anterior

Expectation

$$V(s) = \mathbb{E}\left[G_t \mid S_t = s\right],$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t} R_T.$$

$$V(s) = \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right].$$

$$V(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\left(r + \gamma V(s')\right).$$

Optimallity

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

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

Bellman Equations

Expectation

$$V(s) = \mathbb{E}\left[G_t \mid S_t = s\right],$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t} R_T.$$

$$V(s) = \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right].$$

$$V(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\left(r + \gamma V(s')\right).$$

```
episodes = 5
for episode in range(1, episodes+1):
    state = env.reset()
    done = False
    score = 0

    while not done:
        env.render()
        action = random.choice([0,1,2,3,4,5])
        n_state, reward, done, info = env.step(action)
        score+=reward
    print('Episode:{} Score:{}'.format(episode, score))
env.close()
```

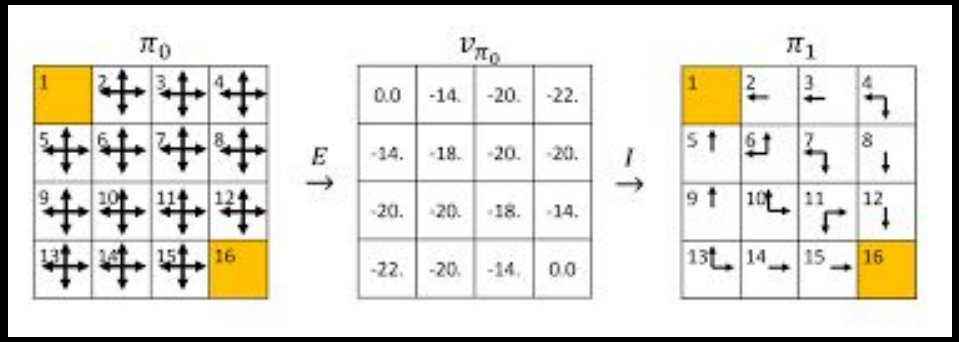
16] ⊗ 0.5s

em dynamic programming: env.P

Bellman Equations

Expectation

$$V(s) = \sum_{a} \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left(r + \gamma V(s')\right).$$



$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

...

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

...

$k = \infty$

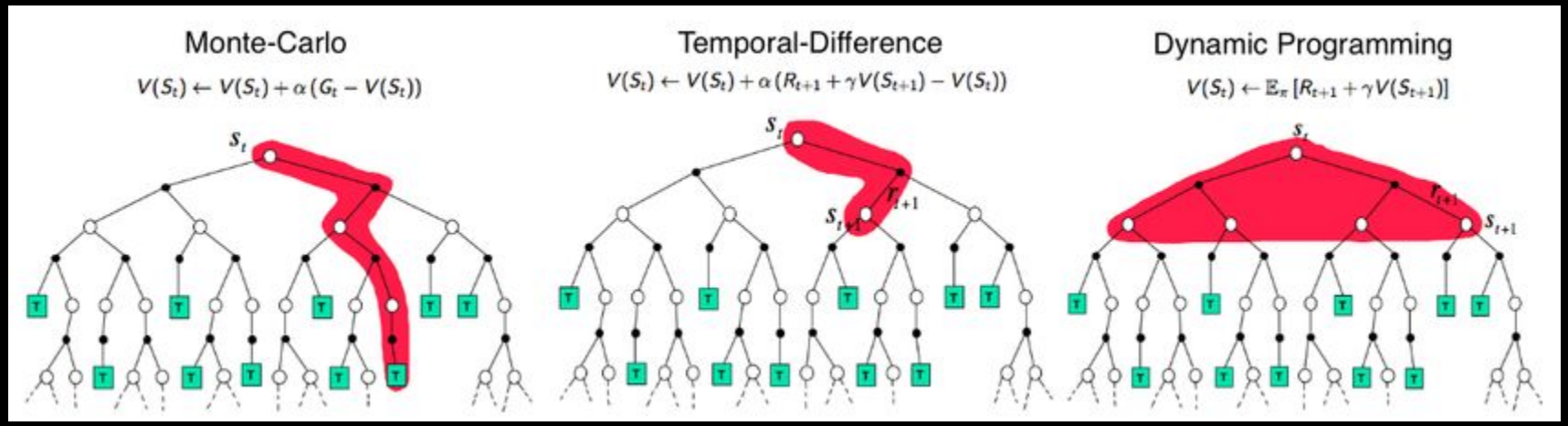
0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

$\leftarrow v_\pi$

Monte Carlo e TD Learning

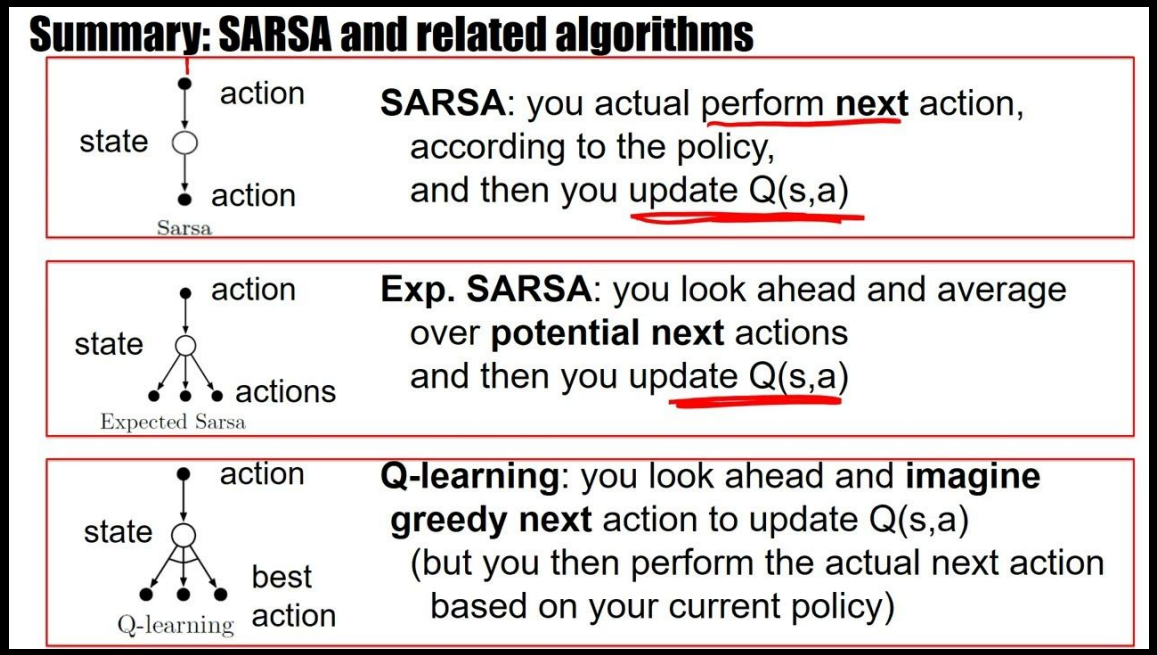
Mundo Real: env.step(), model free, ambos usam value function e só aprendem na medida em que conseguem rewards:

- Monte Carlo: aprende *value function* por episódios completos e recalcula valores de trás para frente, por meio de médias.
- Temporal Difference (TD): atualiza após cada passo (*bootstrapping*)



SARSA vs. Q-Learning

- SARSA: *on-policy* (atualiza *value function* e age conforme mesma política)
- *Q-Learning: off-policy* (usa a melhor ação na atualização)
- Ambos usam TD para atualizar Q(s,a)
- Muito usados em ambientes com ações discretas (necessário) e estados com pouca dimensionalidade



Resumo comparação métodos de RL

- DP: requer modelo (normalmente não dispomos), computacionalmente caro
- Monte Carlo: sem modelo, usa episódios completos
- TD: sem modelo, atualiza por passo
- Q-Learning: off-policy, converge para política ótima (pode ter viés otimista, principalmente se ambiente for estocástico!)
- SARSA: on-policy, mais conservador

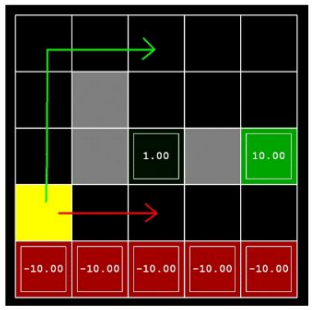
Exploração vs. “Exploitação” (rs)

- Exploração: experimentar ações novas para aprender
- Exploitação: escolher a melhor ação conhecida
- ε decrescente é muito usado em SARSA e Q-Learning

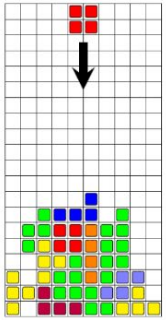
Deep Q-Network (DQN)

Substitui a tabela Q por uma rede neural (Q-table inviável com muitos estados).

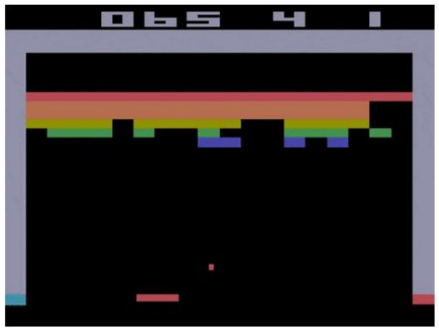
- Discrete environments



Gridworld
10^1

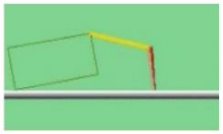


Tetris
10^60

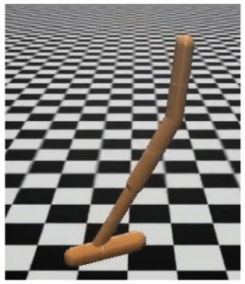


Atari
10^308 (ram) 10^16992 (pixels)

Continuous environments (by crude discretization)



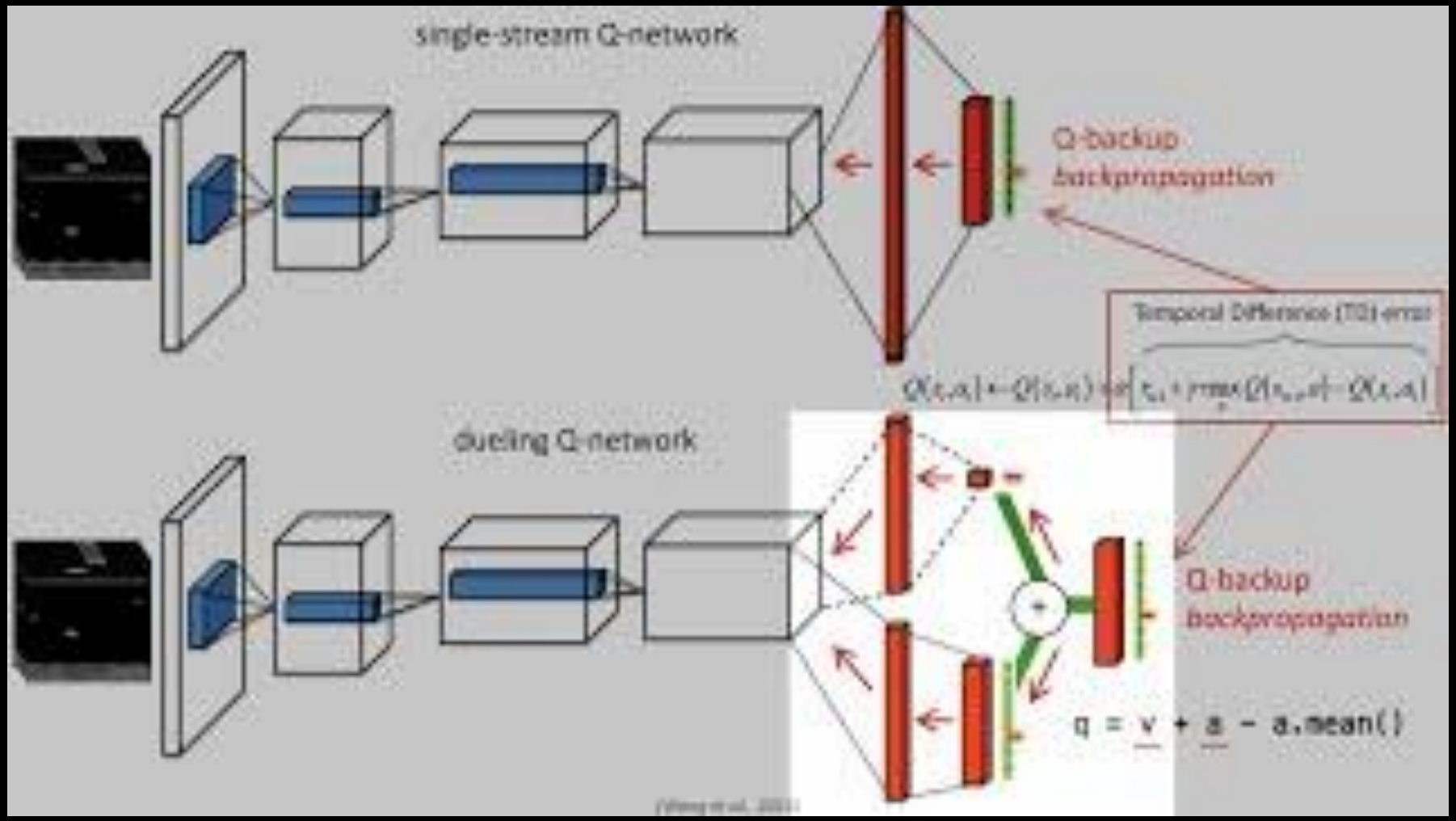
Crawler
10^2



Hopper
10^10

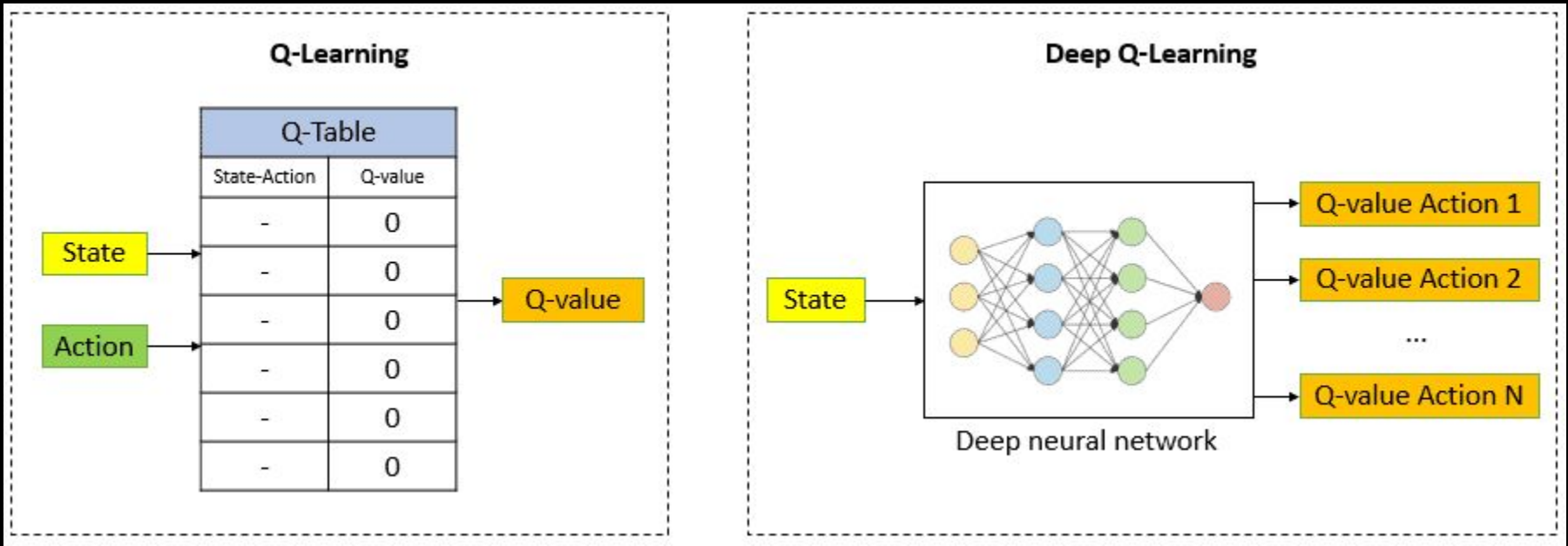
Humanoid
10^100

Deep Q-Network (DQN)



Deep Q-Network (DQN)

- Substitui a tabela Q por uma rede neural (Q-table inviável com muitos estados)
- Replay buffer: armazena e reutiliza experiências
- Target network: reduz instabilidade no treinamento
- Aprendizado eficiente mesmo com estados complexos



Deep Q-Network (DQN)

- Basic Q-Learning keeps a table of all q-values
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the q-tables in memory
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar situations
  - This is a fundamental idea in machine learning

Instead of a table, we have a ***parametrized Q function***: $Q_\theta(s,a)$

- Can be a linear function in features:

$$Q_\theta(s,a) = \theta_0 f_0(s,a) + \theta_1 f_1(s,a) + \cdots + \theta_n f_n(s,a)$$

- Or a neural net, decision tree, etc.

Learning rule:

- Remember: $\text{target}(s') = R(s,a,s') + \gamma \max_{a'} Q_{\theta_k}(s',a')$
- Update:

$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \left[\frac{1}{2} (Q_\theta(s,a) - \text{target}(s'))^2 \right] \Big|_{\theta=\theta_k}$$

Encerramento e Revisão

- RL: interação, recompensa, aprendizado com tentativa e erro
- Evolução:

  DP → MC/TD → Q-Learning → DQN
- Conceitos-chave: política, valor, exploração, estabilidade
- Aplicações reais
- Próximos passos: policy gradient, action space contínuo, etc.

RL Algorithms

This table displays the RL algorithms that are implemented in the Stable Baselines3 project, along with some useful characteristics: support for discrete/continuous actions, multiprocessing.

Name	Box	Discrete	MultiDiscrete	MultiBinary	Multi Processing
ARS [1]	✔	✔	✘	✘	✔
A2C	✔	✔	✔	✔	✔
CrossQ [1]	✔	✘	✘	✘	✔
DDPG	✔	✘	✘	✘	✔
DQN	✘	✔	✘	✘	✔
HER	✔	✔	✘	✘	✔
PPO	✔	✔	✔	✔	✔
QR-DQN [1]	✘	✔	✘	✘	✔
RecurrentPPO [1]	✔	✔	✔	✔	✔
SAC	✔	✘	✘	✘	✔
TD3	✔	✘	✘	✘	✔
TQC [1]	✔	✘	✘	✘	✔
TRPO [1]	✔	✔	✔	✔	✔
Maskable PPO [1]	✘	✔	✔	✔	✔