Reinforcement Learning

Solving the Lunar Lander environment

Nicola Cortinovis, Marta Lucas, Gabriele Pintus

February 7, 2025

University of Trieste

Introduction

Problem Statement

Classic RL problem from OpenAI Gym.

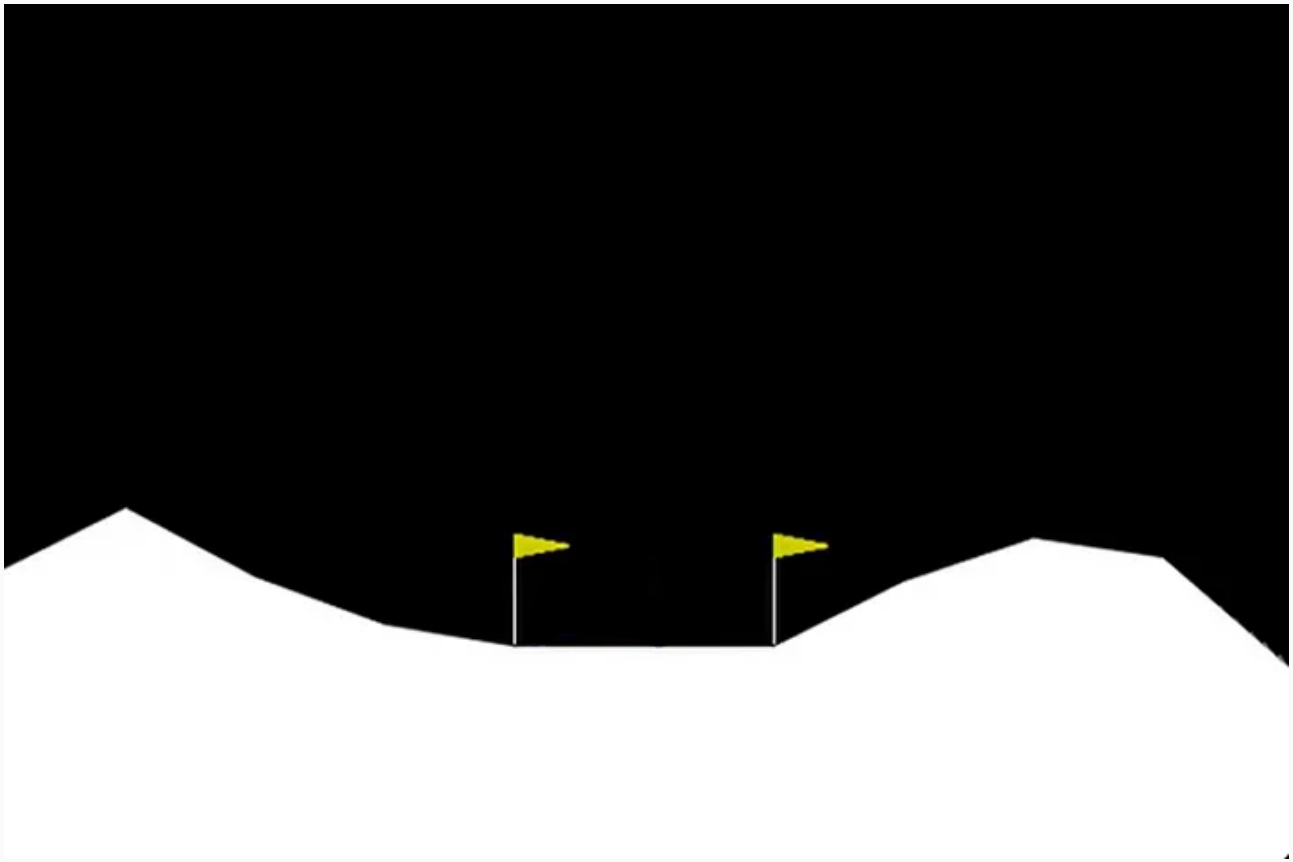
Goal: land the spacecraft safely inside a specific area while optimizing fuel consumption

Challenges:

- Continuous observation space
- Physics constraints

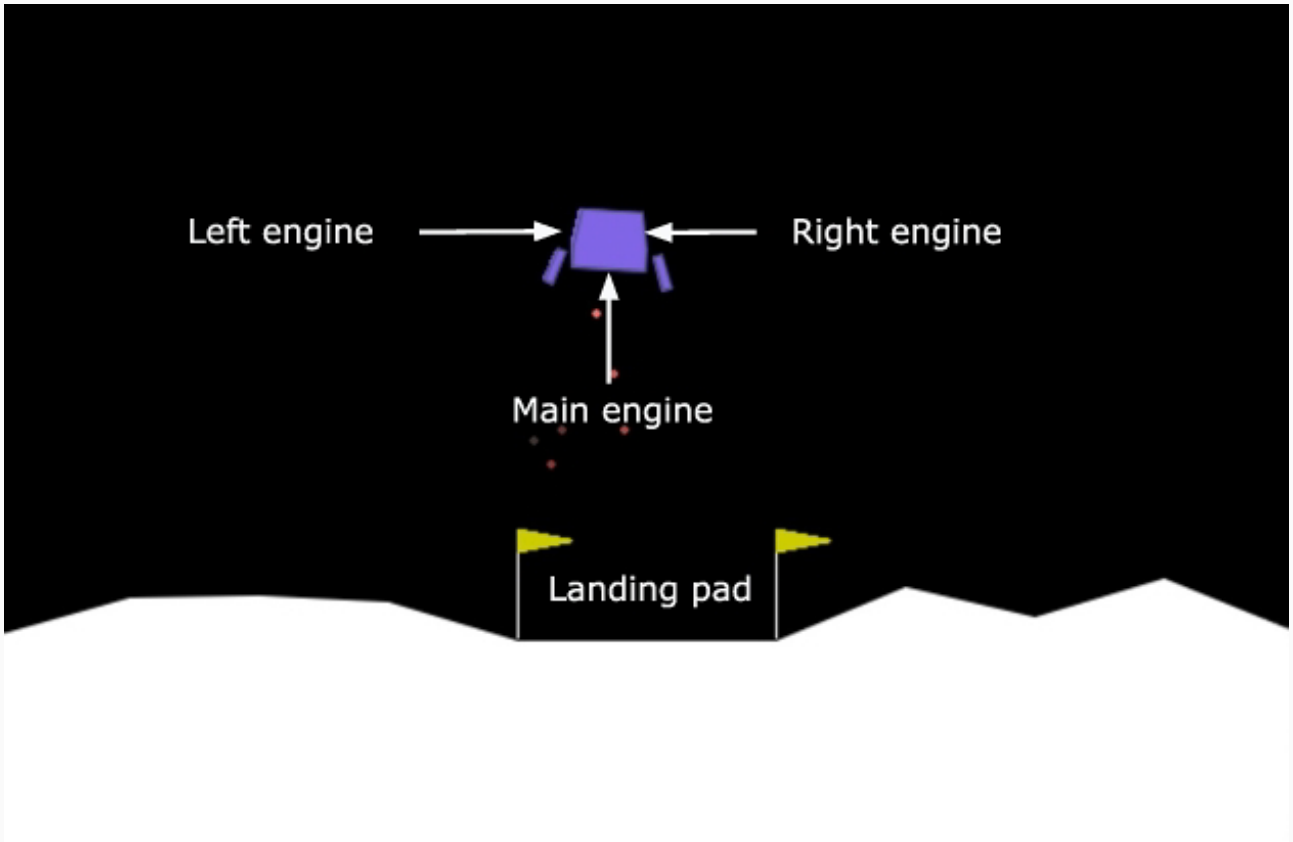
Environment

The yellow flags delimit the landing pad



Agent

The agent is a space shuttle



MDP formulation

To model the problem as an MDP we need:

- State space $\mathcal{S}$
- Action space $\mathcal{A}$
- Transition function $P(s, a, s') = p(s'|a, s)$
- Reward function $R(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$

Observation space

$$\mathcal{S} \subseteq \mathbb{R}^6 \times \{0,1\}^2$$

$$s_t = (x_t, y_t, v_x, v_y, \theta_t, \omega_t, l_t, r_t)$$

Variables	Meaning	Values
x_t, y_t	Position in the $\mathbb{R}^2$ plane	[-2.5,2.5]^2
v_x, v_y	Velocity in the $\mathbb{R}^2$ plane	[-10,10]^2
θ_t	Angle	$[-2\pi, 2\pi]$
ω_t	Angular velocity	$[-10, 10]$
l_t, r_t	Left and right leg contact	{0,1}^{2}

The environment is fully observable, but the agent does not know a-priori where the landing pad is.

Action space

$$\mathcal{A} = \{0, 1, 2, 3\}$$

$$a_t \in \mathcal{A}$$

Action	Effect
0	do nothing
1	fire left orientation engine
2	fire main engine
3	fire right orientation engine

Transition function

$$p(s' \mid a, s) = ?$$

Reward function

Reward	Condition
ΔU_t	Difference in shaping potential
-0.3	Main engine usage per frame penalty (action 2)
-0.03	Side engine usage per frame penalty (actions 1 or 3)
+10	One leg touches ground
+100	Successful landing in target zone
-100	Crash or out-of-bounds

ΔU_t is the *shaping reward* which is computed as follows:

$$U(s_t) = \alpha_1 d(x_t, y_t) + \alpha_2 \|v_t\| + \alpha_3 |\theta_t| \quad \Delta U_t = U(s_{t-1}) - U(s_t)$$

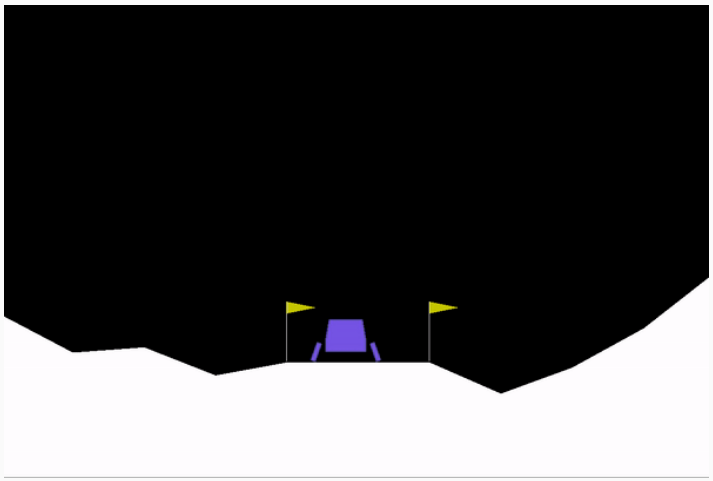
The environment is considered solved if the agent can consistently obtain a final reward of at least **200**

Goal

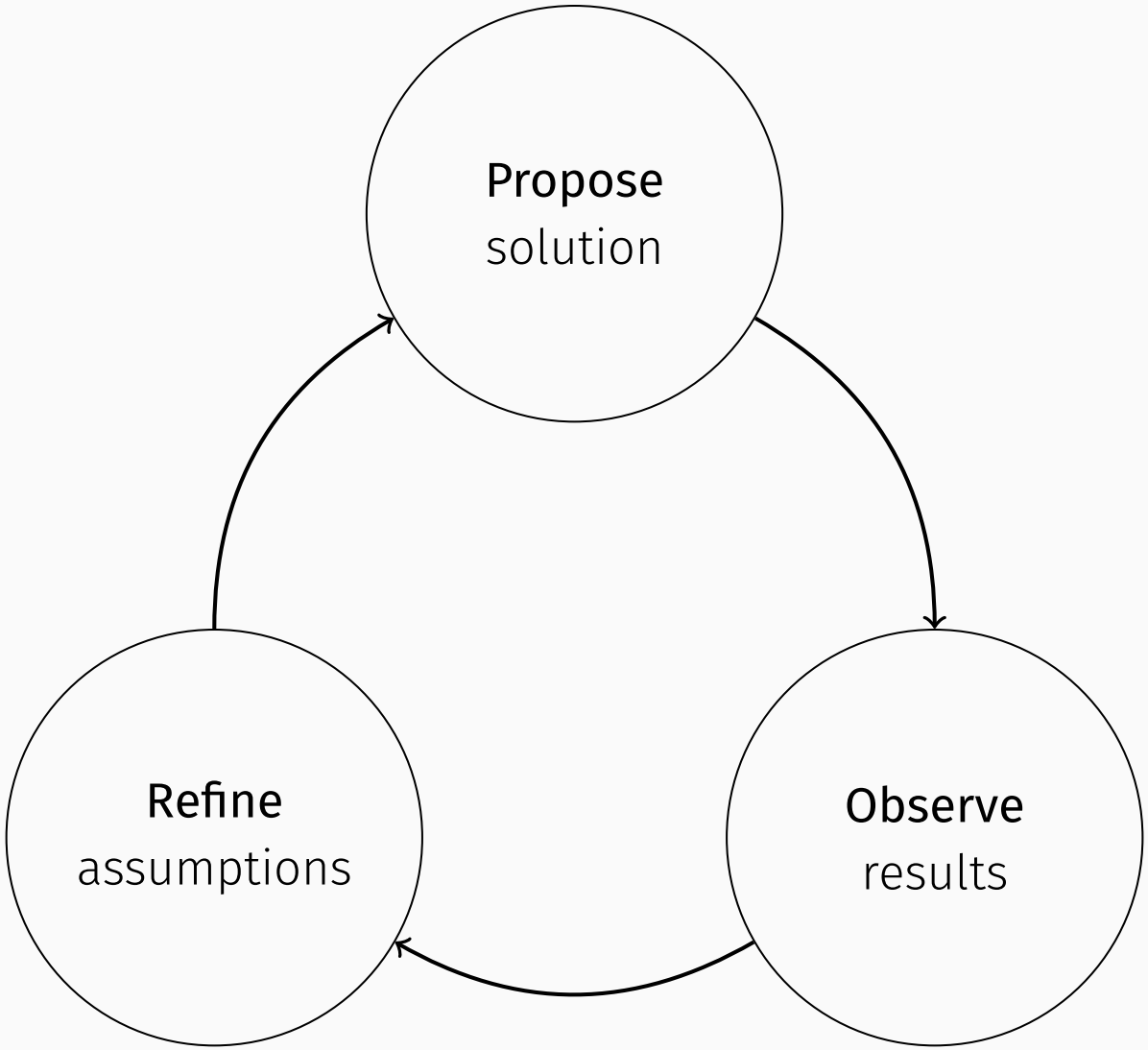
Lunar lander is a control problem: we want to find the best policy π^* which is the one that maximizes the **expected return**:

$$\pi^* = \arg\max_{\pi} v(s) = \arg\max_{\pi} \left(\mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) | S_t = s, A_t = a\right]\right)$$

where $\gamma \in]0, 1]$ is the discount factor.



Modus operandi

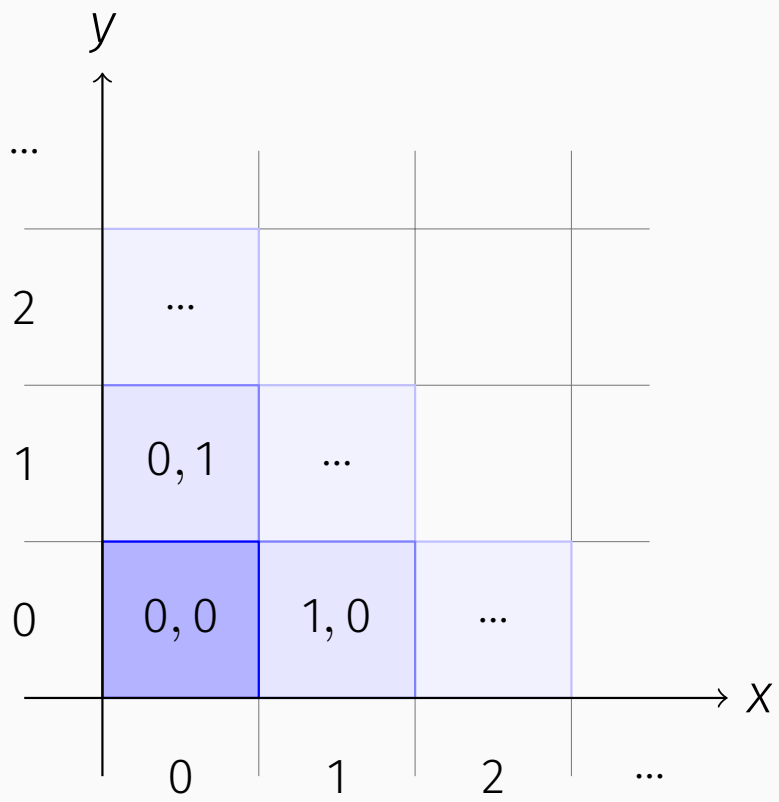


Project outline

- Q-Learning
- Q-Learning with Eligibility Traces
- Double Deep Q-Learning
- Policy Gradient
- Actor-Critic
- Proximal Policy Optimization

Discretization of the observation space

The first two algorithms need a discrete observation space.
Therefore, we divide the continuous state space into a grid.



Q Learning

Q Learning - Motive

Suppose we perfectly estimated $v^*(s)\ \forall s \in \mathcal{S}$. Then, finding the optimal policy is straightforward: act greedily.

$$\begin{aligned}\pi^*(s) &= \arg\max_a q^*(s,a) \\ &= \arg\max_a \sum_{s',r \in S \times R} p(s',r|s,a)\left(r + \gamma v^*(s')\right)\end{aligned}$$

Q Learning - Motive

Suppose we perfectly estimated $v^*(s)\ \forall s \in \mathcal{S}$. Then, finding the optimal policy is straightforward: act greedily.

$$\begin{aligned}\pi^*(s) &= \arg\max_a q^*(s,a) \\ &= \arg\max_a \sum_{s',r \in S \times R} p(s',r|s,a)\left(r + \gamma v^*(s')\right)\end{aligned}$$

However, we have no knowledge of the dynamics function:

$$p(s',r|s,a) = ? \implies \text{estimate } q(s,a) \text{ via experience}$$

Q learning - Update

First, we sample a trajectory from the **behaviour policy** π_b (ε-greedy).

Then we update the state-action values estimate with the **target policy** π_t (greedy).

The update rule is

$$\{S_0, A_0, R_1, S_1, A_1, \ldots, S_{T-1}, A_{T-1}, R_T\} \sim \pi_b$$
$$q(s,a) = q(s,a) + \alpha \left[r + \gamma q(s', \pi_t(s')) - q(s,a)\right]$$

Q learning - Results

	Returns		Steps	
	μ	σ	μ	σ
N = 5	-131.5	42.2	69.9	12.5
N = 10	-131.9	44.1	69.9	12.4
N = 15	-131.0	42.6	70.0	12.7
N = 20	-132.1	47.6	70.1	13.3

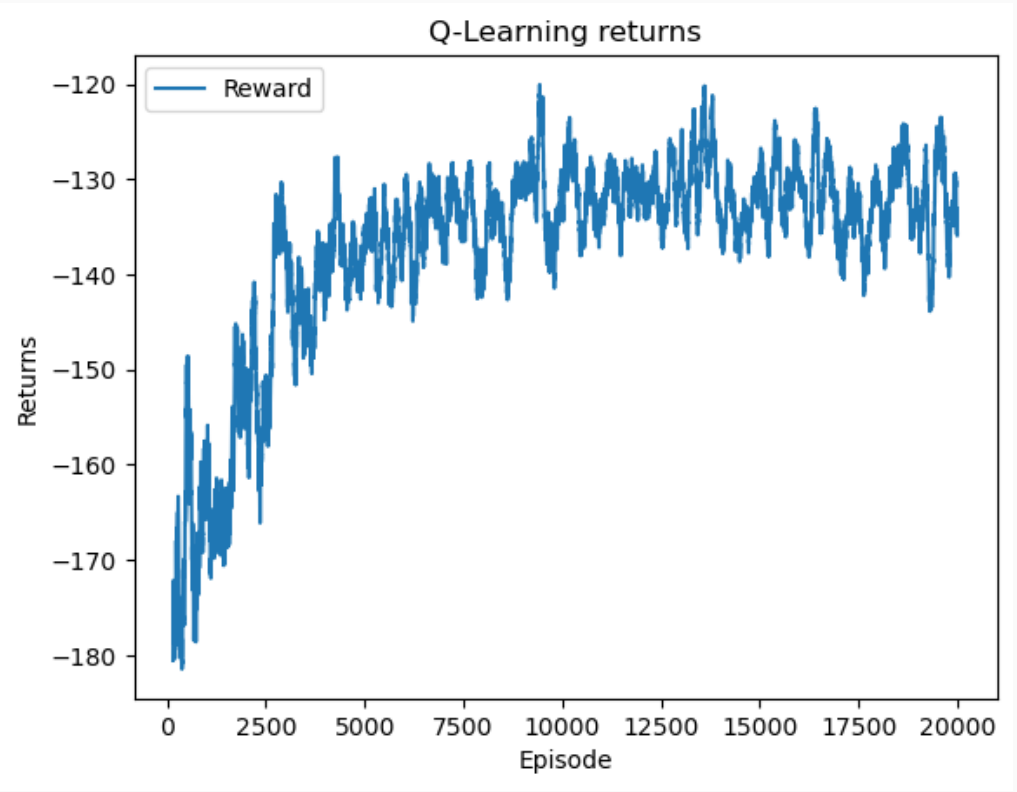


Table 1: Comparison of returns and steps for different values of discretization granularity (*N*). The plot of the training returns over episodes is shown on the right.

$\mathbb{Q}(\lambda)$

Eligibility Traces - Motive

To tackle the challenge of *temporal credit assignment*, we could use a short-term memory vector to track recently visited state-action pairs.

The **eligibility trace** marks how eligible each state-action pair is for learning.

Its value decays over time, so earlier actions receive less credit.

Q Lambda - Update

Whenever we visit an action-state couple (s, a) we increment the trace value by one

$$e(s, a) = e(s, a) + 1$$

Then, Q-Learning update is computed as

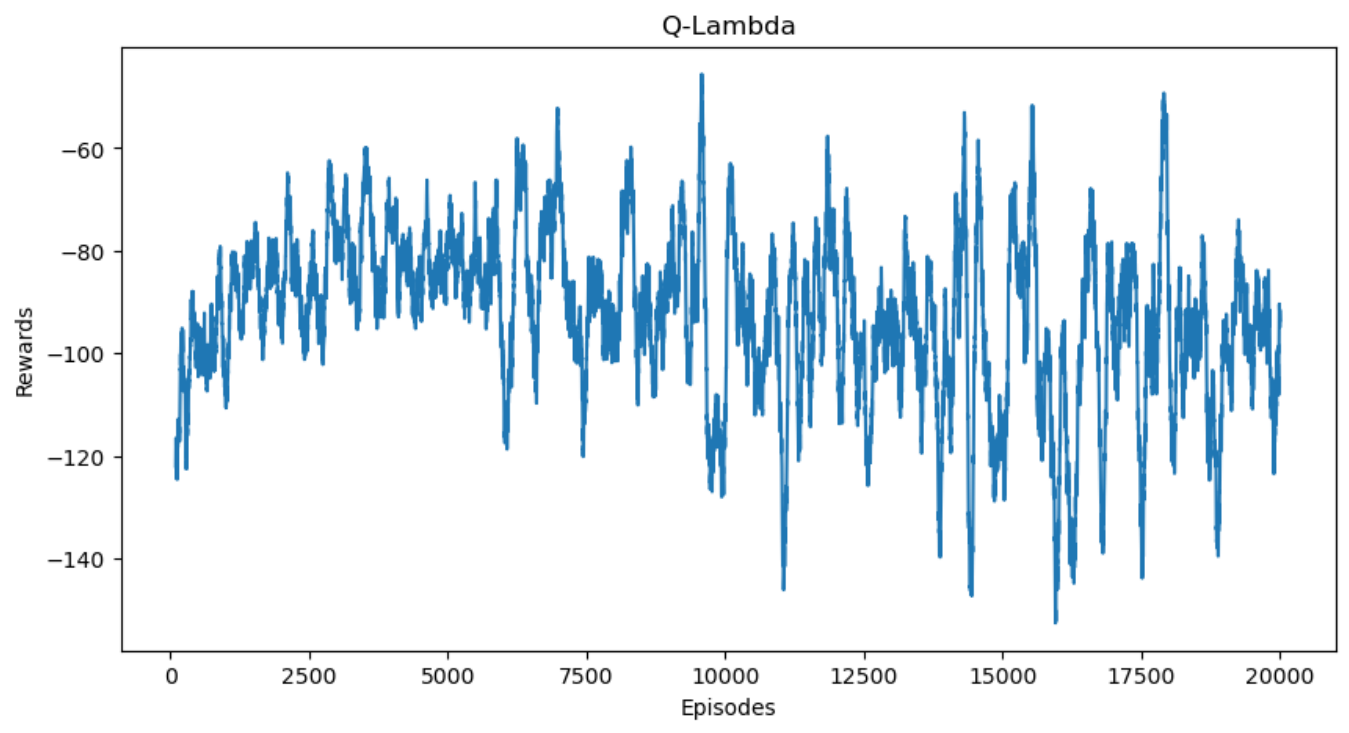
$$q(s, a) = q(s, a) + \alpha \;\; e(s, a) \left[r + \gamma \max_{a'} q(s', a') - q(s, a) \right]$$

And finally the eligibility trace is updated as

$$e(s, a) = \begin{cases} \gamma \lambda \, e(s, a) & \text{if} \quad a' = a^* \\ 0 & \text{otherwise} \end{cases}$$

for all state-action pairs, where $\lambda \in [0, 1]$ is the trace decay.

Q Lambda Results



Training returns over episodes

Deep Q-Learning

Tabular approach issues

Solving a continuos observation problem via a tabular method implies the discretization of the observation space. Consequentially multiple issues arise:

- Less precision in state description
- Zero variability intra-hypercube vs potentially unbounded variability inter-hypercube
- Excessive generalization with coarse discretization vs no generalization with fine discretization

Function approximation

Function approximation avoids the need for explicit discretization of the observation space solving the previously mentioned problems:

- More precise state representation due to the continuity of the approximator
- Smooth changes in action values instead of potentially sharp jumps between discretized states.
- Built-in generalization, the model seamlessly handles unseen states

DQN - Motive

Deep-Q Network approximates the $q(s, a)$ function with a neural network parametrized by $\boldsymbol{\theta}$, obtaining $\hat{q}_{\boldsymbol{\theta}}(s, a)$. Neural networks prove to be a good function approximator since they:

- Generalize by learning from limited data and approximating $q(s, a)$ even for unseen state-action pairs.
- Are *Lipschitz continuous*, meaning no unbounded variability, therefore similar states have similar values
- Handle high-dimensional spaces and don't require handcrafted features

DQN - Sampling experience

We store the experience of multiple episodes obtained by following π_b in a *Replay Buffer* $\mathcal{R}$.

Then, we sample *mini-batches* $\mathcal{B}$ composed by multiple tuples $b_i = (s_t, a_t, r_{t+1} s_{t+1})$ that we obtain from $\mathcal{R}$.

To break the temporal correlation, we sample uniformly by applying the *soft-max* function on the negative values of c_i, a variable that counts how many times a trajectory point i has been previously picked.

$$p(b_i) = \frac{e^{-c_i}}{\sum_i e^{-c_i}}$$

DQN $\rightarrow$ DDQN

We train the neural network via stochastic gradient descent trying to minimizen the Huber loss:

$$L_t(\theta) = \begin{cases} 0.5(y_t - \hat{Q}_{\boldsymbol{\theta}}(s_t, a_t))^2 & \text{if } |y_t - \hat{Q}_{\boldsymbol{\theta}}(s_t, a_t)| < 1 \\ |y_t - \hat{Q}_{\boldsymbol{\theta}}(s_t, a_t)| - 0.5 & \text{otherwise} \end{cases}$$

The *target value* y is actually given by another neural network parametrized by $\boldsymbol{\phi}$ which weights are updated via a linear combination:

$$\boldsymbol{\phi} = \alpha\boldsymbol{\phi} + (1 - \alpha)\boldsymbol{\theta} \qquad \alpha \in [0, 1]$$

This stabilizes training by reducing value overestimation and improving convergence. The final update rule for y_t is:

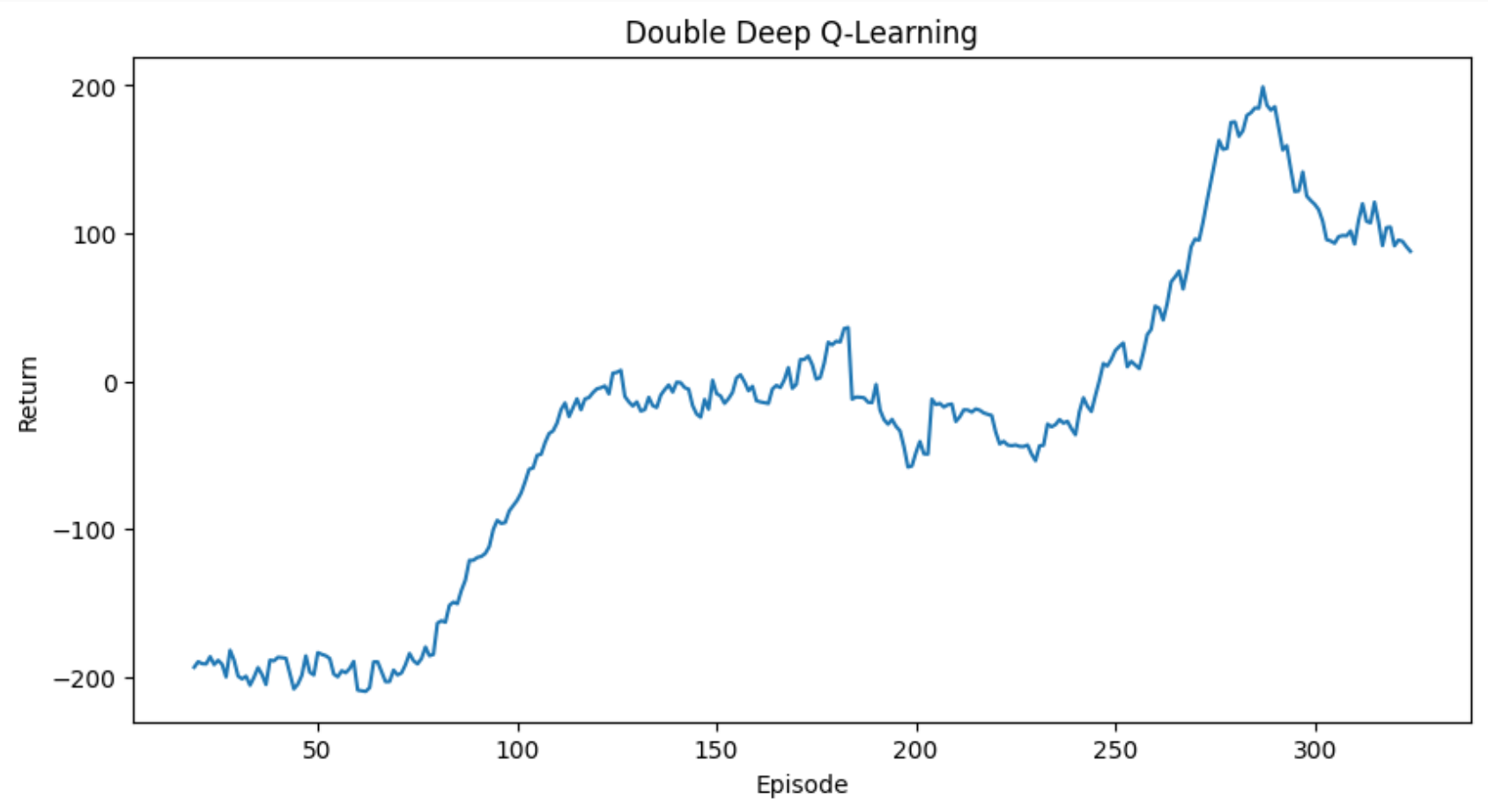
$$y_t = r_t + \gamma \max_{a'} \hat{q}_{\boldsymbol{\phi}}(s_{t+1}, a')$$

DDQN - Update

Algorithm 1 Double Deep Q-learning with Experience Replay

Initialize $\mathcal{R}$ to capacity N, batch size B and $\boldsymbol{\theta}, \boldsymbol{\phi}$
for episode $= 1$ to M **do**
  while not terminal **and** $t < T$ **do**
    Populate $\mathcal{R}$ with experiences $(s_t, a_t, r_{t+1}, s_{t+1})$ following π_b
    if $|\mathcal{R}| > B$ **then**
      Sample random minibatch $(s_j, a_j, r_j, s_{j+1})^B$ from $\mathcal{R}$
      GD step on Huber loss
    end if
  end while
  $\boldsymbol{\phi} \leftarrow \alpha\boldsymbol{\phi} + (1 - \alpha)\boldsymbol{\theta}$
end for

DDQN - Results



Training returns over episodes

Policy Gradient

Policy Gradient - Motive

> *"When solving a given problem, try to avoid solving a more general problem as an intermediate step"*
>
> – *V. Vapnik "The Nature of Statistical Learning Theory"*

Following this principle, in **Policy Gradient** we directly approximate the policy function.

Policy Gradient Theorem

The performance measure is defined as

$$J(\theta) = \mathbb{E}_{\kappa \sim \pi_\theta}[G_\kappa] = v_{\pi_\theta}(s_0) \qquad \kappa = \{S_0, A_0, R_1, \ldots, R_t\}$$

The *Policy Gradient Theorem* lets us write

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi_\theta(a|s)$$

where $\mu(s)$ is the state distribution.

REINFORCE

One of the simplest policy gradient based algorithm is **Monte Carlo Policy Gradient**.

$$\nabla_\theta J(\theta) \propto \mathbb{E}_\pi \left[\sum_a q_\pi(s_t, a) \nabla_\theta \pi_\theta(a|s_t) \right]$$

Substitute the action value function with an estimate of the returns:

$$\mathbb{E}_\pi \left[G_T | S_t = s_t, A_t = a_t \right] = q_\pi(s_t, a_t)$$

Apply the log trick for the derivative and substituting:

$$\mathbb{E}_\pi \left[G_t \nabla_\theta \log \pi_\theta(a_t|s_t) \right]$$

Leading to the update rule for θ:

$$\theta_{t+1} = \theta_t + \alpha \cdot \sum_{t=0}^{T-1} \left[G_t \cdot \nabla_\theta \log \pi_\theta(a_t|s_t) \right]$$

Actor-Critic methods

Actor-Critic methods

In Monte Carlo Policy Gradient, we estimate the policy gradient through sampling. To approximate the expected value, we use the sample mean, which is an unbiased estimator. However, the higher the variance the less accurate the estimate.

Actor-Critic methods

In Monte Carlo Policy Gradient, we estimate the policy gradient through sampling. To approximate the expected value, we use the sample mean, which is an unbiased estimator. However, the higher the variance the less accurate the estimate.

Can we derive a family of policy gradient methods with reduced variance?

Actor-Critic methods

In Monte Carlo Policy Gradient, we estimate the policy gradient through sampling. To approximate the expected value, we use the sample mean, which is an unbiased estimator. However, the higher the variance the less accurate the estimate.

Can we derive a family of policy gradient methods with reduced variance?

Yes, by appling **Control Variates**.

Control Variates

Let $\hat{m}$ be an unbiased estimator of the parameter m, and let $\hat{t}$ be an unbiased estimator of the parameter t.
The unbiased "cv estimator" for the parameter m is defined as

$$m^* = \hat{m} - \alpha(\hat{t} - t)$$

Where α is a coefficient. Now, if one expands $\mathbb{V}[m^*]$, a second order equation in the variable α is obtained. Then, one can minimize the variance of m^* by choosing

$$\alpha = -\frac{\text{Cov}\left[\hat{m}, \hat{t}\right]}{\mathbb{V}\left[\hat{t}\right]}$$

and obtain

$$\mathbb{V}[m^*] = (1 - \rho^2_{\hat{m},\hat{t}})\mathbb{V}[\hat{m}] \implies \mathbb{V}[m^*] \leq \mathbb{V}[\hat{m}]$$

CV on the gradient

We define the baseline function $b(s)$, which substitutes $\alpha(\hat{t} - t)$, and the gradient estimator becomes

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\tau \left[\nabla_\theta \log \pi_\theta(a|s) \cdot (q_{\pi_\theta}(s,a) - b(s))\right] \\ &= \mathbb{E}_\tau \left[\nabla_\theta \log \pi_\theta(a|s) \cdot A(s,a)\right]\end{aligned}$$

where A is the advantage function if $b(s)$ is chosen to be the value function $v(s)$.
In practice one uses the approximated value function $\hat{v}_\phi(s)$.

One can show that requiring b to depend only on s is a sufficient condition to guarantee the unbiasedness of the cv estimator.

Actor and Critic

The **Actor** network approximates the policy $\pi_\theta(a \mid s)$ and decides which action to take.

The **Critic** network estimates the value function $v^\pi(s)$ to evaluate the Actor's decisions, providing feedback by evaluating how good the action was.

Actor-Critic - Update

Algorithm 2 MC Advantage Actor–Critic (A2C)

1: **Initialize** parameters θ, ϕ.
2: **for** episode = 1 to M **do**
3: **Sample** trajectory κ
4: **Compute** returns G_t
5: **Update** critic

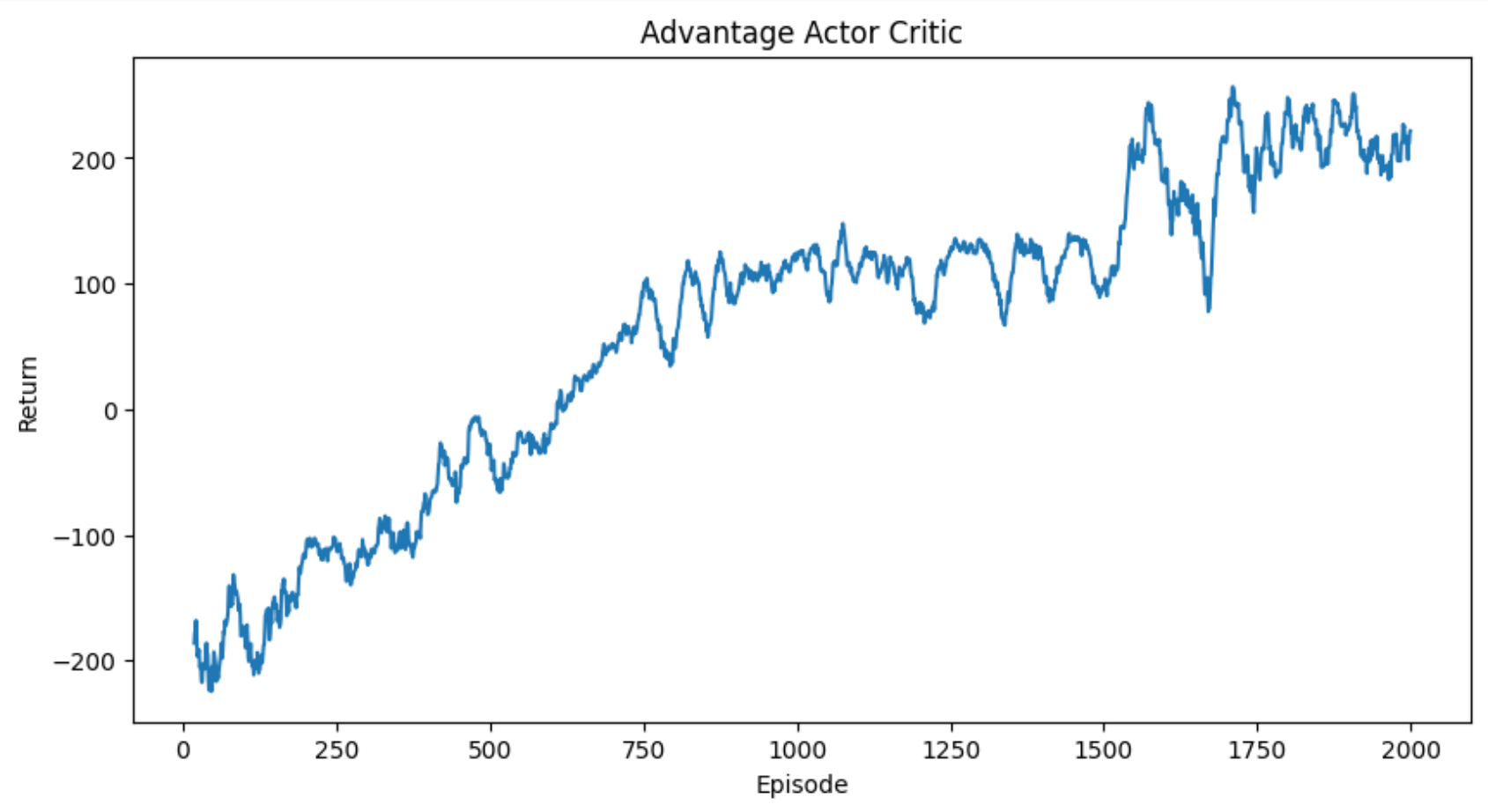
$$\phi \leftarrow \phi + \nabla_\phi L(\phi)$$

6: **Update** actor

$$\theta \leftarrow \theta + \nabla_\theta J(\theta)$$

where L is the *Huber* loss.

Actor-Critic - Results



Training returns over episodes

Proximal Policy Optimization (PPO)

PPO - Motive

Even with a critic, updates on the policy can be excessively large, potentially degrading performance before the agent has a chance to readapt.

To mitigate this, we would like to constrain how much the policy can change.

PPO - Update

We define the surrogate objective

$$L^{\text{CPI}} = \hat{\mathbb{E}}_\tau \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \cdot \hat{A}_\phi(a_t, s_t) \right] = \hat{\mathbb{E}}_\tau \left[r_t(\theta) \cdot \hat{A}_\phi(a_t, s_t) \right]$$

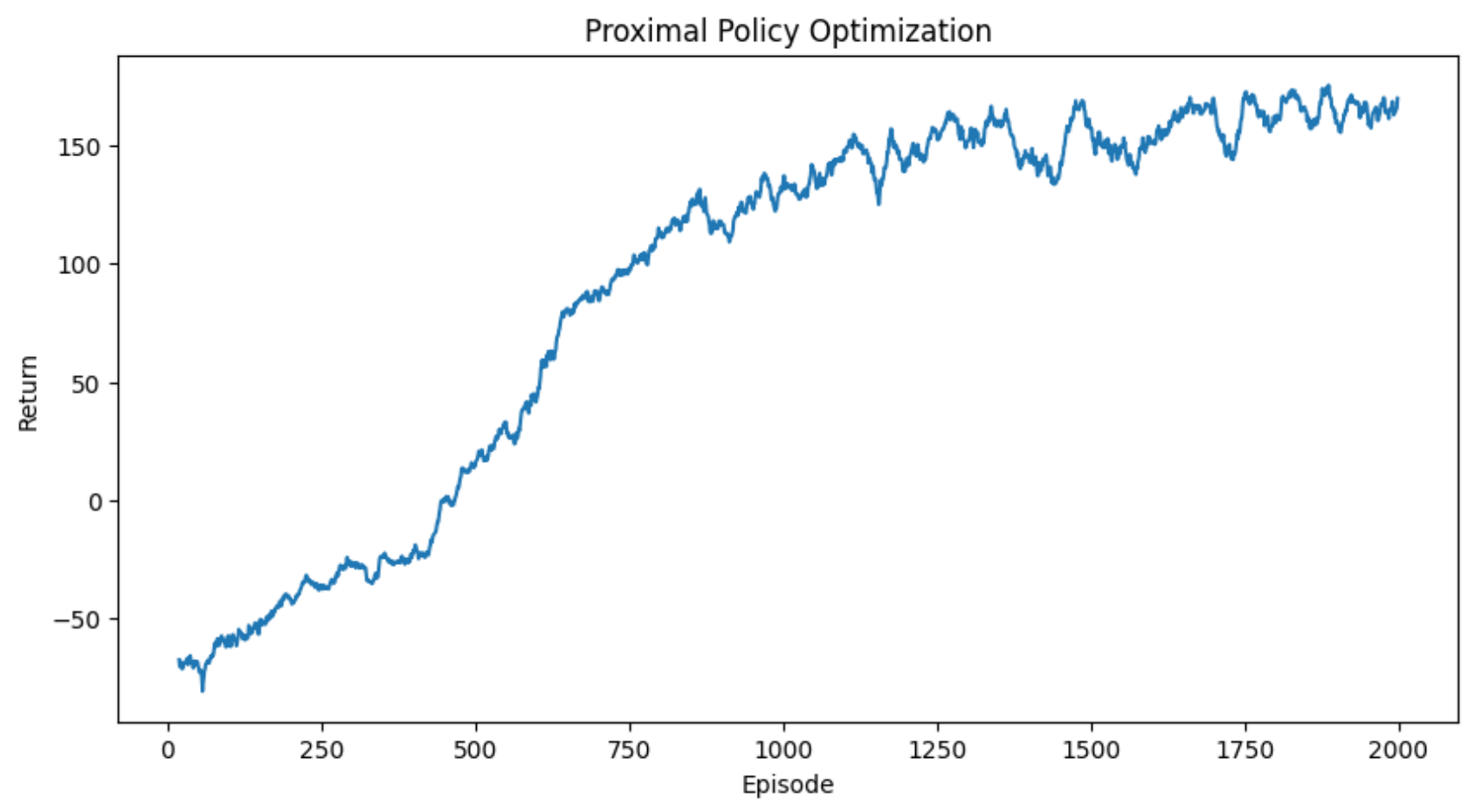
Where the *ratio function* measures how the probability distribution of the actions over the states changes from the old policy $\pi_{\theta_{old}}$ to the new π_θ.
Then, we constrain this variation by *clipping* the ratio

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\textbf{min} \left(r_t(\theta)\hat{A}_\phi(a_t, s_t), \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_\phi(a_t, s_t) \right) \right]$$

and eventually obtain the actual loss function.

$$\theta_{t+1} = \theta_t + \nabla_\theta L^{\text{clip}}(\theta)$$

PPO - Results



Training returns over episodes

Implementation details

ε-greedy policy

In the Q-learning based algorithms, the behaviour policy π_b from which we sample the trajectories is an ε-greedy policy defined as

$$\pi_b(s) = \begin{cases} \text{random action from } \mathcal{A} & \text{with probability } \varepsilon, \\ \arg\max_{a \in \mathcal{A}} Q(s, a) & \text{with probability } 1 - \varepsilon. \end{cases}$$

Initially we take a fully explorative approach, so $\varepsilon = 1$. Then at the end of each episode we decrease this value with the following rule

$$\varepsilon_{t+1} = \min(\eta \cdot \varepsilon_t, \varepsilon_{\min})$$

where $\eta \in [0, 1]$ is the decay rate and $\varepsilon_{\min}$ is the minimum exploration rate we want to keep.

Neural Network

We use a fully connected neural network as our general function approximator. The architecture details are the following:

- `n_layers`: 2
- `hidden_dim`: 64
- `activation`: ReLU
- `dropout`: $\in [0.1, 0.2]$

The dropout mechanism helps avoiding overfitting and makes the network more robust.

The network is trained with the Adam optimizer and gradient clipping is applied before the parameter update.

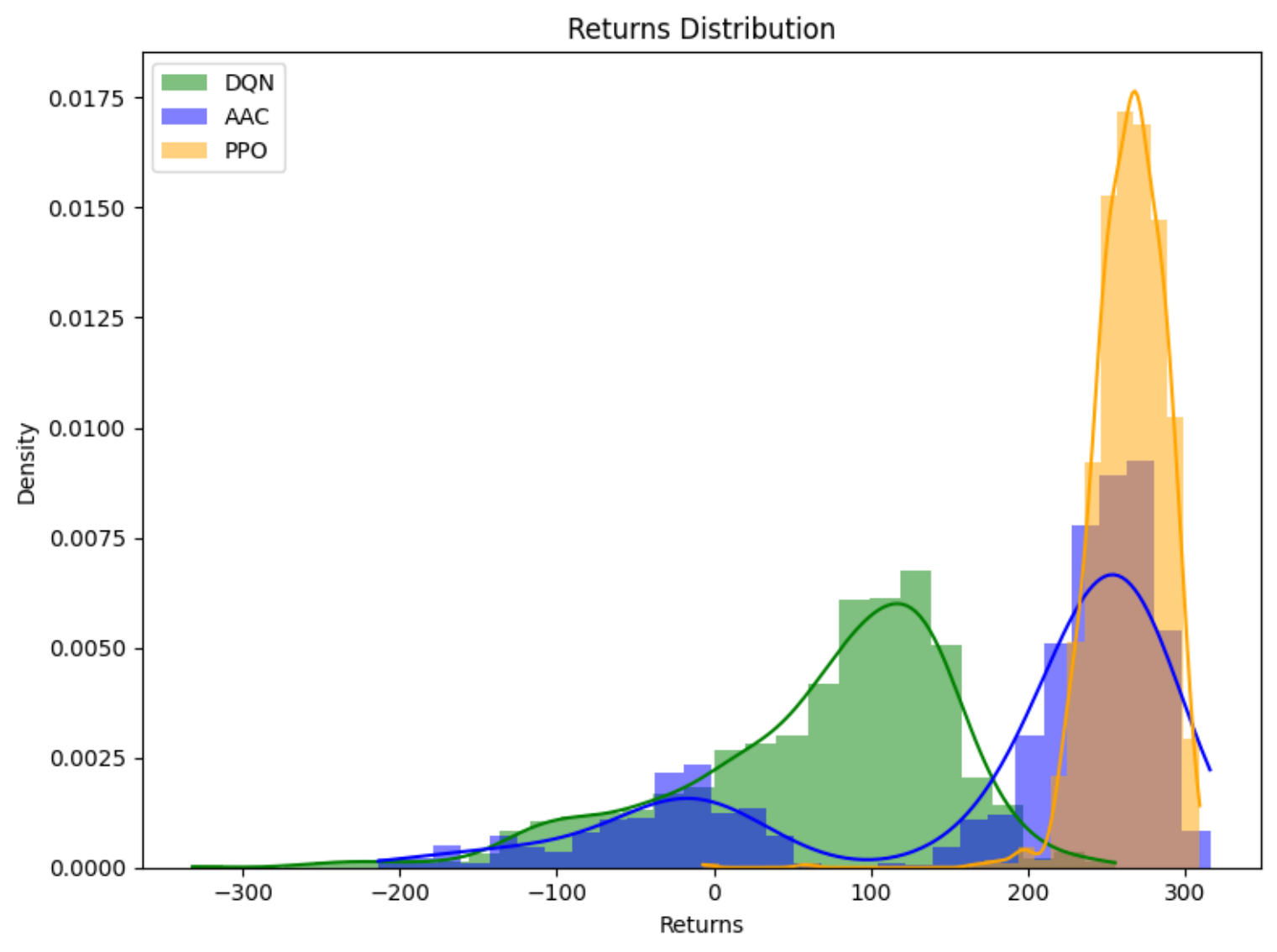
Result comparison

Test comparison

Algorithm	Returns	
	μ	σ
Q	-131.0	42.6
$Q(\lambda)$	-93.6	110.0
DDQN	66.4	85.0
AAC	242	56.0
PPO	**264**	**23.6**

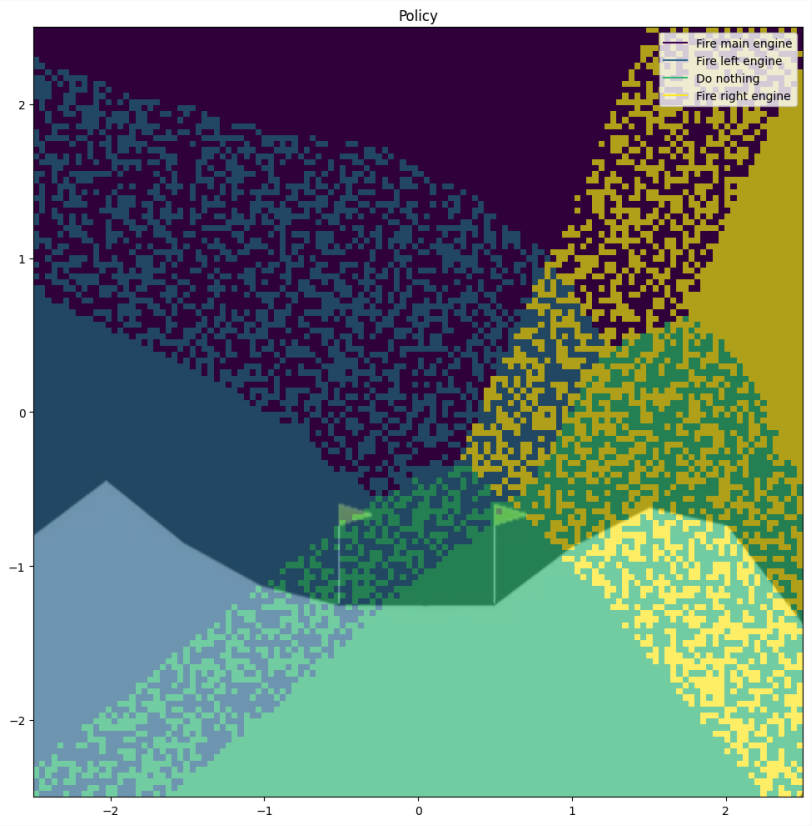
Average and standard deviation for the returns on test runs

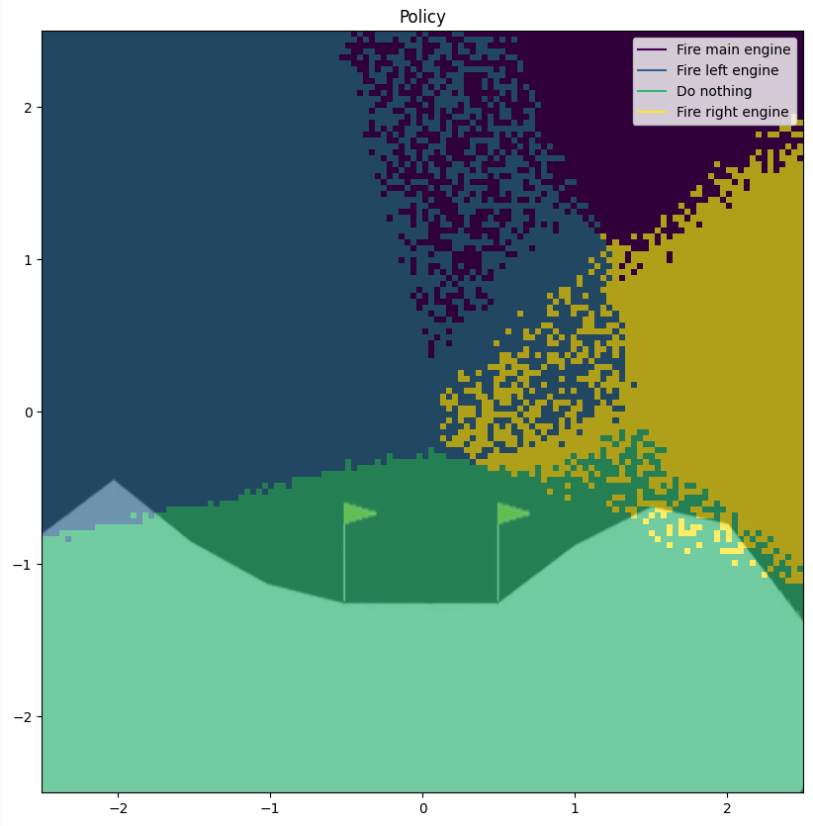
Test comparison



Distribution of returns on test runs for the best algorithms

Policy interpretation - AC vs PPO





Comparison of best-performing actions for AC and PPO.

Thank you for the attention!

References

- Sutton, R. S., Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.
- Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. Springer.
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning.
- Benhamou, E. (2019). Variance Reduction in Actor Critic Methods (ACM).
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). Proximal Policy Optimization Algorithms.