DEEP REINFORCEMENT LEARNING IN PHYSICS-BASED SIMULATIONS

A THESIS SUBMITTED TO

THE FACULTY OF ARCHITECTURE AND ENGINEERING

OF

EPOKA UNIVERSITY

BY

DEVID DUMA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR

THE DEGREE OF MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

JUNE, 2023

Abstract

In neuroscience, reinforcement learning is an important concept for the learning process of all organisms. Tunicata, a marine invertebrate animal, has during larval stage a primitive brain and eyes, swims around and learns to find the best rock to attach itself into. In the adult stage it digests its brain, emphasizing that the point of having a brain is to make decisions and take intelligent actions.

In computer science, reinforcement learning (RL) is a mathematical framework based on Markov Decision Processes, concerned with building rational agents that act so as to achieve the best expected outcome, whilst interacting with an environment without an explicit teacher. Deep reinforcement learning (Deep RL) augments the foundational work in RL with neural networks to solve more complicated tasks, like games, physics-based simulations and robotics.

In robotics, physics-based simulations are crucial for training real-life robots. Simulations have seen adoption accelerated by the rapid growth in computational power over the last three decades [1]. Robots are very complicated systems, training them in the real world can be challenging, since execution and feedback is slow. Physics-based simulation allows sampling experience millions times faster than in the real world, making it possible to train very complicated robots.

In the first chapter of this thesis, I give a brief introduction on RL theoretical fundamentals. In the second chapter, I introduce the theoretical background behind deep RL methods. In the third chapter, I evaluate the performance of deep RL methods in physics-based simulations with MuJoCo, an excellent engine for advanced physics-based simulations. In the fourth chapter, I research the application of off-policy learning methods in robotics simulations. I evaluate the performance of off-policy learning methods in Fetch mobile manipulator, a 7-DoF robotic arm with a two-fingered parallel gripper. Finally, I draw concluding remarks.

Keywords: deep reinforcement learning, physics-based simulations, robotics simulations

Abstrakt

Në neuroshkencë, të mësuarit përforcues është një koncept i rëndësishëm për procesin e të mësuarit të të gjithë organizmave. Tunikata, një kafshë jovertebrore detare, ka një tru dhe sy primitiv gjatë fazës së larvës, noton përreth dhe mëson të gjejë shkëmbin më të mirë për t'u lidhur. Kur rritet ajo e konsumon trurin e vet, çka thekson se qëllimi i të pasurit tru është të marrësh vendime dhe të ndërmarrësh veprime inteligjente.

Në shkencën kompjuterike, të mësuarit përforcues (RL) është një kornizë matematikore e bazuar në Proceset e Vendimit Markov, që ka të bëjë me ndërtimin e agjentëve racionalë që veprojnë në mënyrë që të arrijnë rezultatin më të mirë në pritshmëri, ndërkohë që ndërveprojnë me një mjedis, pa një mësues të dedikuar. Të mësuarit përforcues i thellë (Deep RL) shton mbi punën themelore në RL rrjetet neuronale për të zgjidhur detyra më të ndërlikuara, si lojërat, simulimet e bazuara në fizikë dhe simulimet robotike.

Në robotikë, simulimet e bazuara në fizikë janë thelbësore për trajnimin e robotëve në jetën reale. Simulimet kanë parë adoptimin e përshpejtuar nga rritja e shpejtë e fuqisë llogaritëse të sistemeve kompjuterike gjatë tre dekadave të fundit [1]. Robotët janë sisteme shumë të komplikuara, trajnimi i tyre në botën reale mund të jetë sfidues, pasi ekzekutimi dhe reagimet janë të ngadalta. Simulimi i bazuar në fizikë lejon mbledhjen e përvojës miliona herë më shpejt se në botën reale, duke bërë të mundur trajnimin e robotëve shumë të komplikuar.

Në kapitullin e parë të kësaj teze, unë bëj një hyrje të shkurtër mbi bazat teorike të të mësuarit përforcues (RL). Në kapitullin e dytë, unë prezantoj sfondin teorik prapa metodave të të mësuarit përforcues të thellë (Deep RL). Në kapitullin e tretë, unë vlerësoj performancën e metodave të të mësuarit përforcues të thellë në simulimet e bazuara në fizikë me MuJoCo, një projekt motor i shkëlqyer për simulime të avancuara të bazuara në fizikë. Në kapitullin e katërt, unë hulumtoj zbatimin e metodave të të mësuarit off-policy në simulimet e robotikës. Unë vlerësoj performancën e metodave off-policy në manipuluesin e lëvizshëm robotik Fetch, një krah robotik 7-DoF me një kapëse me dy gishta paralele. Së fundi, unë nxjerr vërejtjet përmbyllëse.

Fjalë kyçe: të mësuarit përforcues i thellë, simulime të bazuara në fizikë, simulime robotike

Acknowledgements

Table of Contents

List of Abbreviations

RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
MDP	Markov Decision Process
Q-function	State value function
V-function	State-action value function
MuJoCo	Multi-Joint dynamics with Contact
DoF	Degree of Freedom
TD	Temporal Difference learning
SARSA	State-action-reward-state-action
SGD	Stochastic Gradient Descent
DQN	Deep Q-Network
PG	Policy Gradient method
AC	Actor-Critic method
NPG	Natural Policy Gradient
A2C	Advantage Actor Critic
TRPO	Trust Region Policy Optimization
PPO	Proximal Policy Optimization
DDPG	Deep Deterministic Policy Gradient
TD3	Twin-Delayed DDPG
SAC	Soft Actor-Critic
REDQ	Randomized Ensembled Double Q-Learning

List of Notations

t	Timestep in episode
T	Duration of episode
$s_t \in S$	State s_t in timestep t in state space S
$a_t \in A$	Action of agent in timestep t from action space A
a' or a_{t+1}	Action of agent in the next state, i.e. timestep $t+1$
r_t	Reward of agent in timestep t
r' or r_{t+1}	Reward of agent in the next state, i.e. timestep $t+1$
γ	Discount factor
α	Learning rate
R_t	Forward-looking returns of agent until terminal state
$\pi(a_t \mid s_t)$	Policy of agent given state s in timestep t
$P(s_{t+1} \mid s_t, a_t)$	Transition probabilities to state s_{t+1}
$V^{\pi}(s)$	State value function under policy π
$Q^{\pi}(s,a)$	State-action value function under policy π
$A^{\pi}(s,a)$	Advantage value function under policy π
θ	Parameters of a neural network for the actor
$\bar{\theta}$	Target network for the actor
$\emptyset$	Parameters of a neural network for the critic
$\overline{\emptyset}$	Target network for the critic
D	Experience Replay Buffer
b	Baseline

$\tau \sim \pi_\theta$	Trajectory $\tau = (s_1, s_2 \dots s_n)$ sampled under policy π_θ from initial state s_1 until terminal state s_n
$max\, J(\theta)$	DRL objective: Maximize expected returns of trajectory τ sampled under policy π parameterized by θ
$\nabla_\theta J(\theta)$ or $\nabla\theta$	Gradient descent update w.r.t. parameters θ at the end of an episode
$L(\theta)$	Loss of parameters θ
$\hat{V}_\theta^\pi, \hat{Q}_\theta^\pi, \hat{A}_\theta^\pi$	Approximations of value functions parameterized by θ
$D_{KL}(\pi_\theta \mid \pi_\theta + \nabla\theta)$	KL-Divergence
$H(\theta)$	Hessian Matrix
$F(\theta)$	Fisher Information Matrix
K	Number of backtracking steps in TRPO and PPO

Chapter 1

Reinforcement Learning

1.1 Introduction

In the domain of artificial intelligence, two of its pioneers Stuart Russell and Peter Norvig propose an interesting taxonomy for the approaches taken when studying AI in their famous book "Artificial Intelligence: A Modern Approach" [2]. According to the authors, we can classify artificial intelligence algorithms into one of the following: thinking humanly, thinking rationally, acting humanly and acting rationally.

Thinking Humanly	Thinking Rationally
Cognitive science	Mathematical logic
Acting Humanly	Acting Rationally
Turing test	Intelligent agents

The most successful approach in building Artificial Intelligence agents has proven to be acting rationally. Reinforcement learning follows exactly this acting rationally approach.

Reinforcement learning stands besides supervised learning and unsupervised learning as one of three machine learning paradigms. In reinforcement learning there is no supervisor like in supervised learning, only a reward signal as feedback in each state after executing an action. Moreover, the data is sequential instead of independent and identically distributed. Agent's actions affect the subsequent data it receives.

1.2 Definition of Reinforcement Learning

Reinforcement Learning (RL) is a mathematical framework based on *Markov Decision Processes* [3]. It is concerned with building rational agents that act so as to achieve the *best expected outcome*, whilst interacting with an environment without an explicit instructor.

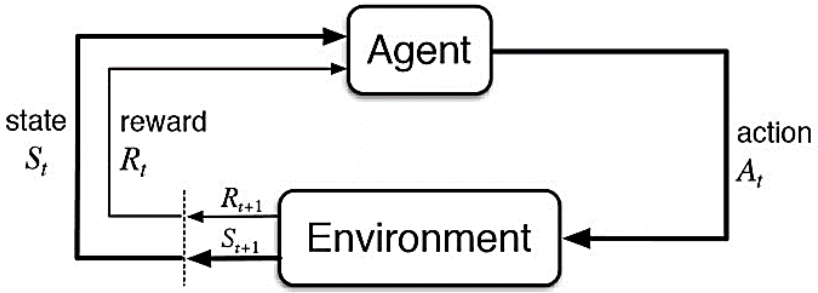


Figure 1-1 Reinforcement Learning loop

Markov Decision Process

Markov Decision Processes are formally defined with the tuple (S, A, P, R, γ):

- S : A finite state space.
- A : A finite set of actions, which are available from each state s.
- P : A transition probability model that specifies $P(s'|s, a)$.
- R : A reward function that maps a state-action pair to rewards (real numbers), i.e. $R: S \times A \rightarrow \mathbb{R}$.
- γ : Discount factor $\gamma \epsilon [0; 1]$.

Markov Property

Markov Decision Processes possess the Markov Property. A stochastic process has the Markov Property, if and only if:

$$P(s_{t+1} \mid s_0, s_1, s_2 \dots s_t) = P(s_{t+1} \mid s_t) \;\; \forall\, t \in \mathbb{N}$$

In other words, e*xpectations over future states are dependent only on the current state, not on past states*. For this reason, RL algorithms are *memoryless* regarding the past.

Return

The agent's objective in an RL algorithm is to maximize the expected returns G_t, i.e. future cumulative reward.

$$G_t = \sum_{i=t} \gamma^i R_i$$

The discount factor γ takes values in the range $[0,1]$. When $\gamma = 1$, we value all future rewards in all future states equally. When $0 < \gamma < 1$, we value immediate future rewards more than future rewards from very far in the future.

Value function

Value functions specify the *expected value* of future cumulative rewards.

- State value function $V_t^\pi : (s) \rightarrow \mathbb{E}[G_t]$ specifies the expected value of future cumulative rewards, starting from state s in timestep t, then following policy π.
- State-action value function $Q_t^\pi : (s, a) \rightarrow \mathbb{E}[G_t]$ specifies the expected value of future cumulative rewards, starting from state s and action a in timestep t, then following policy π.

This is different from the return G_t, which specifies the *real value* of future cumulative rewards. Analytically, we calculate the expectation bootstrapping with the *Bellman backup operator*.

Policy

Policy π determines the best action a to execute in a given state s. The action that delivers the maximal value function is chosen.

$$\pi : s \rightarrow V^{\pi}(s)$$

$$\pi : s \rightarrow Q^{\pi}(s, a)$$

The objective of an RL agent is to choose a policy which maximizes the value function, i.e. the expected returns.

A fundamental theorem in RL states that, given any stationary policy π, we can generate a new deterministic stationary policy π' that is no worse than the existing policy. In other words, we can make step-by-step improvements to a current policy π.

The policy in an RL algorithm is implemented as a *lookup table*.

Causality

Policy $\pi_{\bar{t}}$ at timestep $\bar{t}$ cannot affect rewards from previous timesteps, i.e. reward R_t at timestep t when $t < \bar{t}$.

1.3 Bellman backup operators

Suppose we are given an MDP (S, A, P, R, γ) and a policy π, which can be deterministic or stochastic. Let's assume an infinite horizon, stationary rewards R, stationary transition probabilities P and a stationary policy π.

Let's derive a formula for $Q_t^{\pi}(s, a)$:

$$Q^{\pi}(s,a) = Q_0^{\pi}(s,a) = \mathbb{E}[G_0|s_0 = s, a_0 = a] = \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i R_i | s_0 = s, a_0 = a\right]$$

$$= \mathbb{E}[R_0|s_0 = s, a_0 = a] + \sum_{i=1}^{\infty} \gamma^i \mathbb{E}[R_i|s_0 = s, a_0 = a]$$

$$\overset{(a)}{\leftrightarrow} R(s,a)$$

$$+ \sum_{i=1}^{\infty} \gamma^i \left(\sum_{s' \epsilon S} P(s_1 = s'|s_0 = s, a_0 = a) \mathbb{E}[R_i|s_0 = s, a_0 = a, s_1 = s'] \right)$$

$$\overset{(b)}{\leftrightarrow} R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \left(\sum_{i=1}^{\infty} \gamma^{i-1} \mathbb{E}[r_i|s_1 = s'] \right)$$

$$\overset{(c)}{\leftrightarrow} R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^{\pi}(s')$$

Eq. 1-1

Remarks: (a) is due to the law of total expectation, (b) follows from the Markov Property, (c) follows from linearity of expectation.

Moreover:

$$R^{\pi}(s) = \sum_{a \in A} \pi(a|s) R(s,a)$$

$$P^{\pi}(s'|s) = \sum_{a \in A} \pi(a|s) P(s'|s,a)$$

$$V^{\pi}(s) = \pi(a|s) Q^{\pi}(s,a) = \max_{a' \in A} [Q_t^{\pi}(s,a')]$$

Calculating $V^{\pi}(s) = \pi(a|s) Q^{\pi}(s,a)$ yields:

$$V^{\pi}(s) = R^{\pi}(s) + \gamma \sum_{s' \in S} P^{\pi}(s'|s) V^{\pi}(s')$$

Eq. 1-2

Calculating $V^{\pi}(s) = \underset{a' \in A}{max}[Q_t^{\pi}(s, a')]$ yields:

$$V^{\pi}(s) = \underset{a \in A}{max}\left[R^{\pi}(s) + \gamma \sum_{s' \in S} P^{\pi}(s'|s)V^{\pi}(s')\right]$$

Eq. 1-3

Similar to the formulas we derived in Eq. 1-1, Eq. 1-2 and Eq. 1-3 we define Bellman backup operators:

Bellman expectation backup operator

$$(B^{\pi}U)(s) = R^{\pi}(s) + \gamma \sum_{s' \in S} P^{\pi}(s'|s)U(s'), \ \ \forall s \in S$$

Bellman optimality backup operator

$$(B^{*}U)(s) = \underset{a \in A}{max}\left[R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a)U(s')\right], \ \ \forall s \in S$$

Both operators are *contraction* operators. Applying them iteratively *guarantees convergence* of RL methods towards a single value, a global optimum. Finally, Bellman backup operators have been studied in-depth and contain useful mathematical properties.

1.4 Terminology

Model-based vs model-free

Model-based methods *require* full knowledge of all states and transition dynamics of the environment. We can either build a model of the environment from first principles, or we can learn a model of the environment by performing experiments. Using first principles method, we might result in models that are not accurate, hence the policy learned might be suboptimal. Learning a model from experiments is preferred. Important measures for the efficiency of the models are memory requirements and scalability.

Model-free methods *do not require* full knowledge of all states and transition dynamics of the environment. They scale better to larger applications. Model-free methods are either value function based or policy search. Value function based methods try to learn a value function, then infer an optimal policy from it. Policy search methods directly search in the space of the policy parameters to find an optimal policy.

On-policy vs off-policy

On-policy learning methods attempt to evaluate or improve the policy that is used to make decisions and generate the data.

Off-policy learning methods evaluate or improve a policy *different* from that used to generate the data. The policy being learned about is called the *target policy*, whereas the policy used to sample experience is called the *behavior policy*.

1.5 Dynamic Programming methods

DP methods are model-based and on-policy.

1.5.1 Policy Iteration

Policy Iteration [4] searches over a *policy space*, by iteratively improving on an existing policy, until that policy converges to a global optimum. Iteratively improving on a stationary policy π is possible, because of this theorem:

Theorem: Given any stationary policy π, we can generate a new deterministic stationary policy that is no worse than the existing policy.

Policy Iteration is comprised of Policy Evaluation and Policy Improvement. Policy Evaluation calculates value function for a stationary policy π, with the help of Bellman expectation backup operator. In Policy Improvement, given policy π, we generate a new, improved policy π', with the help of Bellman optimality backup operator.

Policy Iteration repeatedly calls Policy Evaluation and Policy Improvement until the policy stops changing, i.e. the algorithm converges and returns an optimal policy.

algorithm POLICY EVALUATION (M, π, ϵ):

Define $R^{\pi}(s) = \sum_{a \in A} \pi(a|s) R(s,a),\ \forall s \in S$

Define $P^{\pi}(s'|s) = \sum_{a \in A} \pi(a|s) P(s'|s,a),\ \forall s, s' \in S$

Initialize $V'(s) \leftarrow 0, V(s) \leftarrow \infty,\ \forall s \in S$

while $\|V - V'\|_{\infty} > \epsilon$ **do**:

$V \leftarrow V'$

$V'(s) = R^{\pi}(s) + \gamma \sum_{s' \in S} P^{\pi}(s'|s) V(s')$

return $V'(s),\ \forall s \in S$

Algorithm 1 Policy Evaluation algorithm as presented in the literature

algorithm POLICY IMPROVEMENT (M, V^{π}):

$\hat{\pi} \leftarrow \underset{a \in A}{arg\,max}\ [R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^{\pi}(s')], \forall s \in S$

return $\hat{\pi}(s), \forall s \in S$

Algorithm 2 Policy Improvement algorithm as presented in the literature

algorithm POLICY ITERATION (M, ϵ):

Initialize $\pi \leftarrow$ randomly choose a policy $\pi \in \Pi$

while true **do**:

$V^{\pi} \leftarrow$ POLICY EVALUATION (M, π, ϵ)

$\pi^{*} \leftarrow$ POLICY IMPROVEMENT (M, V^{π})

if $\pi^{*}(s) = \pi(s)$ **then**

break

else

$\pi \leftarrow \pi^{*}$

$V^{*} \leftarrow V^{\pi}$

return $V^{*}(s), \pi^{*}(s), \forall s \in S$

Algorithm 3 Policy Iteration algorithm as presented in the literature

1.5.2 Value Iteration

Value Iteration [5] searches over a *value function space*, by applying the *Bellman optimality backup operator* iteratively, until an optimal policy is found.

Value Iteration is guaranteed to converge towards a global optimum.

algorithm VALUE ITERATION (M, ϵ):

Initialize $V'(s) \leftarrow 0, V(s) \leftarrow \infty$

while $\|V - V'\|_\infty > \epsilon$ **do**:

$V \leftarrow V'$

$V'(s) = \max_{a \in A} [R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)V(s')], \ \forall s \in S$

$\mathrm{V}^* \leftarrow V, \ \forall s \in S$

$\pi^* \leftarrow \arg\max_{a \in A} [R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)\mathrm{V}^*(s')], \ \forall s \in S$

return $V^*(s), \ \pi^*(s), \ \forall s \in S$

Algorithm 4 Value Iteration algorithm as presented in the literature

1.6 Monte Carlo methods

Monte Carlo on-policy evaluation [6] uses the Monte Carlo computational method. In Monte Carlo, we first sample an entire episode, then update the V-value or Q-value.

Monte Carlo methods are model-free and on-policy. Compared to dynamic programming methods introduced previously, Monte Carlo methods being model-free *do not require* full knowledge of all states and transition dynamics of the environment, hence scale better to larger applications. Even though Monte Carlo is model-free, it is analytically guaranteed to converge to a global optimum. Moreover, Monte Carlo methods are *agnostic* to the Markov Decision Process setting, hence can be applied to a very wide range of problems.

Monte Carlo on-policy evaluation can be implemented in three versions:

- First visit Monte Carlo: If the current state s is visited for the first time, update $V(s)$ or $Q(s,a)$.
- Every visit Monte Carlo: For every visit of state s, no matter if it is the first visit, update $V(s)$ or $Q(s,a)$.
- Incremental Monte Carlo: Introduces a learning rate α.

Monte Carlo on-policy evaluation can either update $V(s)$ or $Q(s,a)$, depending on the implementation.

algorithm MONTE CARLO EVALUATION (M, s, t, N):

Initialize $i \leftarrow 0$

Initialize $G_t \leftarrow 0$

while $i \neq N$ **do**:

Sample an episode, starting from state s and time t

Using sampled episode, calculate return $g \leftarrow \sum_{i=t}^{H-1} \gamma^{i-t} r_i$

$G_t \leftarrow G_t + g$

$i \leftarrow i + 1$

Update rule for $V(s)$ or $Q(s, a)$

return $V_t(s)$ or $Q_t(s, a)$

Algorithm 5 Monte Carlo evaluation algorithm as presented in the literature

Update rules for First Visit Monte Carlo:

if first visit **then**

$$V_t(s) \leftarrow G_t / N(s)$$

if first visit **then**

$$Q_t(s, a) \leftarrow G_t / N(s, a)$$

Update rules for Every Visit Monte Carlo:

$$V_t(s) \leftarrow G_t / N(s)$$

$$Q_t(s, a) \leftarrow G_t / N(s, a)$$

Update rules for Incremental Monte Carlo:

$$V_t(s) \leftarrow V_t(s) + \alpha[G_t - V_t(s)]$$

$$Q_t(s, a) \leftarrow Q_t(s, a) + \alpha[G_t - Q_t(s, a)]$$

1.7 Temporal Difference learning methods

Temporal Difference (TD) learning [7] *bootstraps* the value functions with Bellman backup operator *while sampling*. This is an improvement over Monte Carlo, which calculates value functions only after sampling an entire episode. For this reason, TD learning methods are a better choice than Monte Carlo methods in MDPs with very long episodes, or non-episodic domains.

TD learning methods are model-free and on-policy. In TD learning, we bootstrap the next state's value estimate to get the current state's value estimate, so the estimate is biased by the estimated value of the next state.

1.7.1 TD-0

TD-0 calculates the TD target $R + \gamma V_{\pi}(s_{t+1})$ every step of the episode, hence bootstraps information while sampling.

algorithm TD-0 (α, n):

Initialize $V^{\pi}(s) \leftarrow 0$
while $n > 0$ **do**:
Begin episode E at state s
while $n > 0$ and episode E has not terminated **do**:
$a \leftarrow$ action at state s under policy π
Take action a in E and observe reward r, next state s'
$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha\big(R + \gamma V^{\pi}(s') - V^{\pi}(s)\big)$
$s \leftarrow s'$
return V^{π}

Algorithm 6 TD-0 algorithm as presented in the literature

1.7.2 SARSA

SARSA [8] calculates the TD target $R_t + \gamma Q_\pi(s_{t+1}, a_{t+1})$ every step of the episode, hence bootstrap the information while sampling. Essentially, SARSA is the same as TD-0, but instead of using V-values in its update rules like in TD-0, SARSA uses Q-values in its update rule.

algorithm SARSA (α_t, ϵ):

Initialize $Q(s, a), \forall s \in S, a \in A$ arbitrarily, except $Q(terminal, \cdot) = 0$

$\pi \leftarrow \epsilon$-greedy policy w.r.t. Q

for each episode **do**:

Set state s_1 as starting state

Choose action a_1 from policy $\pi(s_1)$

while episode E has not terminated **do**:

Take action a_t and observe reward r_t and next state s_{t+1}

Choose action a_{t+1} from policy $\pi(s_{t+1})$

$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$

$\pi \leftarrow \epsilon$-greedy w.r.t. Q (policy improvement)

$t \leftarrow t + 1$

return Q, π

Algorithm 7 SARSA algorithm as presented in the literature

1.8 Q-Learning methods

Q-Learning [9] is an off-policy learning method for Temporal Difference style control. Q-Learning is similar to SARSA, but it *bootstraps the Q-value from the next state*, by checking Q-values of all possible actions, *before* actually choosing and executing that action.

Q-Learning methods are model-free and off-policy.

1.8.1 Q-Learning

algorithm Q-LEARNING $(\epsilon, \alpha, \gamma)$:

Initialize $Q(s,a), \forall s \in S, a \in A$ arbitrarily, except $Q(terminal,\cdot) = 0$

$\pi \leftarrow \epsilon$-greedy policy w.r.t. Q

for each episode **do**:

Set state s_1 as starting state

$t \leftarrow 1$

while episode E has not terminated **do**:

Choose action a_t from policy $\pi(s_t)$

Take action a_t and observe reward r_t, next state s_{t+1}

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

$\pi \leftarrow \epsilon$-greedy w.r.t. Q (policy improvement)

$t \leftarrow t + 1$

return Q, π

Algorithm 8 Q-Learning algorithm as presented in the literature

Q-Learning introduces maximization bias. In other words, some optimistic bias is present towards some actions, making them better than their actual Q-value. Depending on the problem, maximization bias can be beneficial or a drawback.

1.8.2 Double Q-Learning

Double Q-Learning [10] is exactly the same as Q-Learning, with a modified update rule that prevents maximization bias. Double Q-Learning decouples the action selection from Q-value evaluation, by using two independent, unbiased estimates of Q-values: Q_1 and Q_2. Q_1 is used to select the action yielding the maximal returns, whereas Q_2 is used to estimate the value of this maximum return. With 0.5 probability Q_1 is updated and with 0.5 probability Q_2 is updated.

algorithm DOUBLE Q-LEARNING $(\epsilon, \alpha, \gamma)$:

Initialize $Q_1(s,a), Q_2(s,a)\ \ \forall s \in S, a \in A$

$\pi \leftarrow \epsilon$-greedy policy w.r.t. $Q_1 + Q_2$

for each episode **do**:

Set state s_1 as starting state

$t \leftarrow 1$

while episode E has not terminated **do**:

Choose action a_t from policy $\pi(s_t)$

Take action a_t and observe reward r_t and next state s_{t+1}

if (with 0.5 probability) **then**

$$Q_1(s_t,a_t) \leftarrow Q_1(s_t,a_t) + \alpha\left[r_t + \gamma\, Q_2\left(s_{t+1}, \underset{a'}{arg\,max}\ Q_1(s_{t+1},a')\right) - Q_1(s_t,a_t)\right]$$

else

$$Q_2(s_t,a_t) \leftarrow Q_2(s_t,a_t) + \alpha\left[r_t + \gamma\, Q_1\left(s_{t+1}, \underset{a'}{arg\,max}\ Q_2(s_{t+1},a')\right) - Q_2(s_t,a_t)\right]$$

$\pi \leftarrow \epsilon$-greedy w.r.t. $Q_1 + Q_2$ (policy improvement)

$t \leftarrow t + 1$

return $Q_1 + Q_2, \pi$

Algorithm 9 Double Q-Learning algorithm as presented in the literature

Chapter 2

Deep Reinforcement Learning

2.1 Introduction

Traditional RL saves Q-values internally using a *lookup table*. It also approximates the Q-value analytically with Bellman backup operators. This approach might not scale with very large state and action spaces. In other cases, we might prefer quickly learning good approximate value functions over exact value functions. Deep RL parameterizes the policy with weights θ in a neural network, serving as a function approximator for the Q-value. Input features of the network are observations o_t for state s_t and rewards R_t, from some fixed size replay buffer.

Other possible architectures for function approximators could be linear combinations, decision trees, nearest neighbors or Fourier or wavelet bases. Nevertheless, the most popular choice for weights θ are neural networks.

Deep RL allows us to solve more complicated tasks, like physics-based simulations and robotics tasks. However, it is important to note that Deep RL methods are *not guaranteed to converge*. Research in Deep RL has high computational complexity and requires long computational time, in order to find the best hyperparameters that lead to convergence, i.e. solve a task.

2.2 Definition of Deep Reinforcement Learning

Objective

Deep reinforcement learning techniques can be framed as an optimization problem on parameters θ, in order to find the optimal θ^*:

$$\theta^* = \underset{\theta}{arg\,max}\, \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

We can use a variety of optimization techniques, like Stochastic Gradient Descent to optimize this objective.

To be more concrete, let us define the term inside $\underset{\theta}{argmax}$ as a function $J(\theta)$:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[r(\tau)]$$

We could re-write function $J(\theta)$ as:

$$J(\theta) = \int \pi_\theta r(\tau) d\tau$$

With this integral, we can easily take the gradient to perform gradient descent or ascent. Essentially, optimizing the new objective $J(\theta)$ with gradient descent or ascent is equivalent to taking the operator $\underset{\theta}{argmax}$, i.e. finding the optimal parameters θ.

Experience replay buffer

Sampling trajectories τ from the environment consecutively, means that they will be temporally correlated. If we were to feed them to a neural network directly, it would lead to overfitting. Experience replay buffer keeps a fixed number of trajectories in the buffer β, sampled from different timesteps. The temporal correlation between the trajectories in the buffer is broken. Each training step of the policy, trajectories are chosen from the replay buffer at random.

2.3 Types of Deep Reinforcement Learning methods

Value function based

Value based methods estimate V-function or Q-function of the optimal policy. There is no explicit policy, but rather the policy is inferred from the value functions.

Policy Gradient

Policy Gradient methods directly differentiate the RL objective.

$$\nabla_\theta J(\theta) = \nabla_\theta \, \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t R(s_t, a_t) \right]$$

Actor-Critic

Actor-Critic methods estimate value function or Q-function of the current policy, then use it to improve the policy.

Model based

Model based methods estimate the transition model and then use it for planning, with no explicit policy, as well as to improve a policy. As we saw in the previous chapter, model based methods are common in Reinforcement Learning. However, they are not common in Deep Reinforcement Learning.

2.4 Value function methods

2.4.1 Deep Q-Network (DQN)

Deep Q-Network (DQN) [11] is based on Q-Learning, but uses a neural network as a function approximator for the Q-value. The network is trained by minimizing a loss function at every iteration i, given by:

$$L_i(\theta_i) = E_{s\sim\rho_{\pi(\cdot)},a\sim\pi(\cdot)}[R_i - Q(s,a;\theta_i)]^2$$

$$R_i = E_{s'\sim\epsilon}\left[r_i + \gamma \max_{a'} Q(s',a',\theta_{i-1}|s,a)\right]$$

R_i is the target at iteration i, $\pi(s|a)$ is the behavior policy, $\rho_{\pi(\cdot)}$ is the distribution of states under policy π, and ϵ refers to the environment.

To minimize the loss function, the gradient of the loss function is computed w.r.t. the weights:

$$\nabla_{\theta_i} L(\theta_i) = E_{s\sim\rho_{\pi(\cdot)},a\sim\pi(\cdot),s'\sim\epsilon}\left[\left(r_i + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

Stochastic Gradient Descent is used to minimize the loss function. The behavior policy is an $\epsilon-$greedy policy to ensure sufficient exploration.

Deep Q-Networks make use of an *experience replay buffer*.

algorithm DEEP Q-NETWORK:

Initialize replay memory D

Initialize Q-value with random weights w

for each episode **do**:

Observe initial state s_1

for t=1:T **do**:

Select action a_t using Q (e.g. ϵ −greedy)

Take action a_t

Observe reward r_t and new state s_{t+1}

Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D

Sample random transitions (s_i, a_i, r_i, s_{i+1}) from D

for each transition **do**:

Calculate target R_i like below:

if s_{i+1} is terminal **then**

$R_i = r_i$

else

$R_i = r_i + \gamma \underset{a'}{max} Q(s_{i+1}, a'; \theta)$

Train the Q network on $\left(R_i - Q(s_i, a_i; \theta)\right)^2$ using SGD

Algorithm 10 DQN algorithm as presented in the literature

Like in Q-Learning, maximization bias is present.

2.4.2 Double Deep Q-Network (DDQN)

Double deep Q-networks (DDQN) [12] make use of a target network to stabilise training in Deep Q-Networks. The online network is used to evaluate the greedy policy and select an action to execute. The target network is used to calculate the new value function for the update. The update is the same as for DQN, but replacing the target with:

$$R_i = r_i + \gamma \max_{a'} Q\left(s_{i+1};\ \operatorname{argmax}_a Q(s_{i+1}, a, \theta);\ \theta'\right)$$

DDQN is shown to reduce bias and improve performance on the same set of problems that DQN is used.

2.5 Policy Gradient methods

Policy Gradient methods [13] attempt to optimize the policy directly. This is different from Q-value methods described before, which parameterize and optimize the Q-value.

Policies represent action probabilities. Parameterizing the policy directly means that we are estimating the probabilities of the agent to take the action at a specific state, instead of estimating a table of action-state-rewards. Policy Gradient methods train the policy with trajectories directly when they are sampled, then discard the trajectory. They don't use replay buffers or similar techniques to store previous experiences.

Unlike Q-Learning based algorithms, Policy Gradients are capable of functioning in continuous action spaces. Furthermore, Policy Gradients can be employed in scenarios where actions do not need to be executed in discrete steps, like pressing on-off switch. Instead actions are continuous, like a car's steering wheel turn.

2.5.1 Policy Gradient Theorem

Let us derive a mathematical expression for directly calculating the derivate of the policy. This is done by directly differentiating the Deep RL objective:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[r(\tau)] = \int \pi_\theta(\tau) r(\tau) d\tau$$

Let us also recall a convenient identity:

$$\pi_\theta(\tau) \nabla_\theta log \pi_\theta(\tau) = \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} = \nabla_\theta \pi_\theta(\tau)$$

Using this identity, we can take the gradient of $J(\theta)$ in an elegant way:

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta r(\tau) d\tau$$

$$= \int \pi_\theta(\tau) \nabla_\theta log \pi_\theta(\tau) r(\tau) d\tau$$

$$= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [\nabla_\theta log \pi_\theta(\tau) r(\tau)]$$

Recall that, by Bayes' rule:

$$\pi_\theta(\tau) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

Taking the log on both sides, we get:

$$log\ \pi_\theta(\tau) = log\ p(s_1) + \sum_{t=1}^{T} log\ \pi_\theta(a_t|s_t) + log\ p(s_{t+1}|s_t, a_t)$$

We substitute into our original gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \left(log\ p(s_1) + \sum_{t=1}^{T} log\ \pi_\theta(a_t|s_t) + log\ p(s_{t+1}|s_t, a_t) \right) r(\tau) \right]$$

$$= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta log\ \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right]$$

We cancel out $log\ p(s_1)$ and $log\ p(s_{t+1}|s_t, a_t)$, because we are taking the gradient w.r.t. θ. Both these expressions do not depend on θ.

2.5.2 REINFORCE

We derived a mathematical expression for directly calculating the derivate of the policy, which involved an expectation. The above expression is foundational for Policy Gradient methods. However, in most cases we cannot easily obtain this expectation. With increasing complexity of problems and huge state and action spaces, the expectation will involve an intractable integral.

As presented in the previous chapter, Monte Carlo methods are perfect for approximations. We take N samples and average them out:

$$\nabla_\theta J(\theta) \cong \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log\pi_\theta(a_{i,t}|s_{i,t})\left(\sum_{t=1}^{T} r(s_{i,t},a_{i,t})\right)\right)$$

where t means timestep t and i means i-th rollout

With the above gradient, we can do gradient descent on parameters θ, by:

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

REINFORCE algorithm [14] makes use of these gradient derivations, with the additional Monte Carlo technique. It is the simplest Policy Gradient algorithm.

algorithm REINFORCE:

Require: base policy $\pi_\theta(a_t|s_t)$, sample trajectories τ^i

while true **do**:

Sample τ^i from $\pi_\theta(a_t|s_t)$

$\nabla_\theta J(\theta) \cong \frac{1}{N}\sum_i\left(\sum_t \nabla_\theta \log\pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_t r(s_{i,t},a_{i,t})\right)$

Improve policy by $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

return optimal trajectory from gradient ascent as τ^{return}

Algorithm 11 REINFORCE algorithm as presented in the literature

REINFORCE suffers from high variance.

2.5.3 What are Policy Gradients actually doing?

Notice that the first item in the final expectation formula is similar to Maximum Likelihood Estimation:

$$\nabla_\theta J_{ML}(\theta) = \frac{1}{N}\sum \nabla_\theta log\, \pi_\theta(\tau_i)$$

That being said, the intuition behind the derived formula:

$$\nabla_\theta J(\theta) \cong \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta\, log\pi_\theta(a_{i,t}|s_{i,t})\left(\sum_{t=1}^{T} r(s_{i,t},a_{i,t})\right)\right)$$

is that with Policy Gradient methods, we are trying to make:

- good trajectories more likely
- bad trajectories less likely

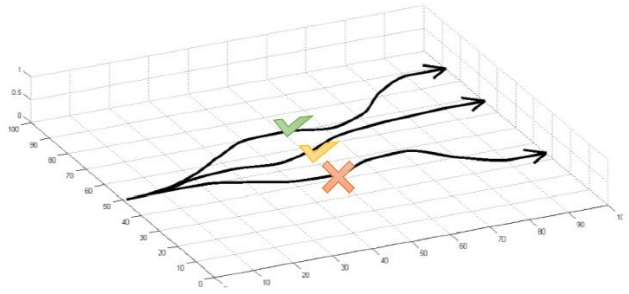


Figure 2-1. Improving the likelihood of good trajectories with Policy Gradients

2.5.4 Improving Policy Gradient

Reward-to-go

In the update rule derived in Policy Gradient Theorem, we can substitute $\sum_{t=1}^{T} r(s_t, a_t)$ with an approximation of Q-function $\hat{Q}_{i,t}$, also called the "reward-to-go". $\hat{Q}_{i,t}$ can be parameterized by $\emptyset$ and we can improve it using gradient descent. We get a new update rule:

$$\nabla_\theta J(\theta) \cong \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_\theta log\,\pi_\theta(a_t|s_t)\right)\hat{Q}_{i,t}^{\emptyset}\right]$$

Reward-to-go and similar techniques are the standard in more advanced methods, like Actor-Critic methods.

Baselines

The goal of Policy Gradient methods is to make good trajectories more likely and worse trajectories less likely. Nevertheless, this leads to high variance. To reduce variance, we could define a better goal: we make *better than average* trajectories more likely and *worse than average* trajectories less likely. We refer to these averages as *baselines*.

Property of baselines: Subtracting a baseline from the gradient update formula is unbiased in expectation, hence we will converge towards the same optimum.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_\theta log\,\pi_\theta(a_t|s_t)\right)(r(\tau)-b)\right]$$

A naïve baseline could be the average of all returns from all episodes:

$$b = \frac{1}{N}\sum_{i=1}^{N} r(\tau)$$

Although not optimal, this baseline is practical in implementation and delivers very good results. This is the baseline that is mostly used.

We could also find an optimal baseline analytically, by analyzing the variance of $\nabla_\theta J(\theta)$ and setting its gradient to 0.

$$\frac{dVar}{db} = \frac{d}{db}\mathbb{E}[\nabla_\theta log\pi_\theta(a_t|s_t)^2(r(\tau)-b)^2]$$

Finally, the optimal baseline which reduces variance is:

$$b = \frac{\mathbb{E}[\nabla_\theta log \pi_\theta(\tau) r(\tau)]}{\mathbb{E}[\nabla_\theta log \pi_\theta]}$$

Advantage Function

Another possible term that could act as a baseline is the V-value function $V(s_t)$. In that case, we choose to subtract it from the reward-to-go. Intuitively, we are calculating how much better choosing action $a_{i,t}$ in state $s_{i,t}$ is to choosing the average action for that state. The update rule becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta log\, \pi_\theta(a_t|s_t) \right) \left(Q(s_{i,t}, a_{i,t}) - V(s_{i,t}) \right) \right]$$

The term $\hat{A}_{i,t} \coloneqq Q(s_{i,t}, a_{i,t}) - V(s_{i,t})$ is called the advantage function.

Importance Sampling

In importance sampling, we do not use sample trajectories τ from $\pi_\theta(\tau)$ in our calculations of $J(\theta)$ and updates to the parameters θ, but instead we use trajectories from another $\pi_{\hat{\theta}}(\tau)$, where $\pi_{\hat{\theta}}(\tau)$ could be an old policy, or demonstrations from a person. Actually, Policy Gradient with Importance Sampling is off-policy.

In expectation, it is proven analytically that using Importance Sampling, we converge towards the same optimum as without Importance Sampling.

2.5.5 Natural Policy Gradient (NPG)

In Natural Policy Gradient [15], the argument is that a more refined update rule than Gradient Descent is necessary, one that represents the steepest descent direction based on the underlying structure of the parameter space of θ. Natural Policy Gradient introduces a new update rule called the Natural Gradient Descent.

NGD calculates an Advantage function $\hat{A}_t^{\pi_k}$, similar to reward-to-go. Furthermore, a KL-divergence between the Hessian Matrix $\widehat{H}_k$ and Fisher Information Matrix $F(\theta)$ is introduced. Hessian Matrix presents us the second-order derivative of π_θ, which measures how much the policy π_θ changes. Finally, the KL-divergence computes the difference between the policy before and after the update.

The KL-divergence has the form:

$$D_{KL}(\pi_\theta|\pi_\theta + \nabla\theta) = \sum_{x\epsilon X} \pi_\theta(x) log\left(\frac{\pi_\theta(x)}{\pi_\theta + \nabla\theta(x)}\right)$$

The Fisher Matrix has the form:

$$F(\theta) = \mathbb{E}_\theta[\nabla_\theta log\pi_\theta(x)\nabla_\theta log\pi_\theta(x)^T]$$

algorithm NATURAL POLICY GRADIENT (NPG) :

Initialize policy parameters θ_0

for $k = 0{:}\,K$ **do**:

Collect set of trajectories D_k on policy $\pi_k = \pi(\theta_k)$

Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm

Form sample estimates for:

- Policy gradient $\hat{g}_k$ (using advantage estimates)
- KL-divergence Hessian / Fisher Information Matrix $\widehat{H}_k(\theta)$

Compute Natural Policy Gradient update:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{\hat{g}_k^T \widehat{H}_k^{-1}(\theta)\hat{g}_k}}\,\widehat{H}_k^{-1}(\theta)\hat{g}_k$$

Algorithm 12 NPG algorithm (simplified) as presented in the literature

There are downsides to NPG. NGD may misrepresent the actual distance between policies, causing step sizes to be too large. Inversing the Fisher matrix F is a costly operation of $O(N^3)$ complexity. Fisher Information Matrix F is a $|\theta| \cdot |\theta|$ matrix, which may take substantial memory to store. KL-divergence constraint might not be satisfied. Furthermore, policy improvement is not verified. All these issues are addressed by more advanced methods, like TRPO and PPO.

2.6 Actor-Critic methods

Actor-Critic methods [13] build upon the Policy Gradient framework and augment it with learned V-value functions and Q-functions. The methodology behind them is Policy Iteration, which alternates between policy evaluation and policy improvement [16]. In an Actor-Critic method, two key components are present, the actor and the critic.

The actor's parameters θ directly recommend the action that the agent should take. Actor updates its policy taking into account the critics output. Notice that the actor implements a Policy Gradient, for updating parameters θ. The critic's parameters $\emptyset$ estimate Q-values to measure how good the choices made by the actor are. Critics rely on replay buffers. To address large-scale RL applications with continuous action spaces, the Q-value function and policy are optimized jointly, because it is impractical to run either of these steps to convergence.

Actor-Critic methods can either be on-policy or off-policy. On-policy training tends to improve stability, but suffers from high variance and results in poor sample efficiency [16]. Off-policy learning methods reduce variance and are more sample efficient. In off-policy learning methods, the actor needs 2 network parameters, one for the behavior policy used to sample experience and another for the target policy, the policy we update for learning.

2.7 On-policy Actor-Critic methods

On-policy Actor-Critic methods are stable, but are high variance. They are not sample efficient, hence require a lot of training samples.

2.7.1 Advantage Actor-Critic (A2C)

Advantage Actor-Critic (A2C) is the synchronous version of Asynchronous Advantage Actor-Critic (A3C) [17]. A2C calculates the advantage function as a reward-to-go. First, A2C waits for each actor to finish its segment of experience before updating. Then, to calculate the value function it averages over all of the actors. This more effectively uses GPUs due to larger batch sizes.

algorithm ADVANTAGE ACTOR-CRITIC:

Initialize parameters $s, \theta, \emptyset$ and learning rates $\alpha_\theta, \alpha_\emptyset$

while true **do**:

Take action $a \sim \pi_\theta(a|s)$ and observe (s, a, s', r)

Update $\hat{V}_\emptyset^\pi(s)$ using target $y = r + \gamma \hat{V}_\emptyset^\pi(s')$

Evaluate $\hat{A}^\pi(s_i, a_i) = R(s_i, a_i) + \hat{V}_\emptyset(s_i') - \hat{V}_\emptyset(s_i)$

Calculate update $\nabla_\theta J(\theta) \cong \sum_i \nabla_\theta \, log \pi_\theta(a_i|s_i) \hat{A}^\pi(s_i, a_i)$

Improve policy by $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

return optimal policy from gradient ascent as π^{return}

Algorithm 13 A2C algorithm as presented in the literature

2.7.2 Trust Region Policy Optimization (TRPO)

Trust Region Policy Optimization (TRPO) [18] improves over NPG, introducing three improvements.

Conjugate gradient method

In Natural Gradient Descent, computing the inverse Fisher matrix is time-consuming, with $O(N^3)$ time complexity and is often numerically unstable. Instead of calculating the entire inverse Fisher matrix and Hessian matrix, we approximate them with an iterative algorithm.

Conjugate Gradient improves NGD by approximating the Fisher matrix and Hessian matrix iteratively. CG generally converges within $|\theta|$ steps.

Line search

In Natural Gradient Descent, the optimal step size given the constraint placed on KL-divergence may not be satisfied. Line search iteratively reduces the size of the update, until it does not violate the KL-divergence constraint.

Natural policy *presumes* the divergence constraint is met, whereas line search performed in TRPO enforces it. Finally, the trust region shrinks, i.e. the region within we trust the update to actually improve the objective.

algorithm LINE SEARCH FOR TRPO:

Compute proposed policy step $\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k$

for $j = 0,1,2, \dots, L$ **do**:

compute proposed update $\theta = \theta_k + \alpha^j \Delta_k$

if $L(\theta) \geq 0$ and $D_{KL}(\theta|\theta_k) \leq \delta$ **then**

accept the update and set $\theta_{k+1} = \theta_k + \alpha^j \Delta_k$

break

Algorithm 14 Line Search for TRPO as presented in the literature

Improvement check

Rather than *presuming* the update will improve the surrogate advantage $\mathcal{L}(\theta)$, we *verify* whether our update actually improves the policy before accepting it. We compute advantages based on the old policy, using importance sampling to adjust the probabilities.

algorithm TRUST REGION POLICY OPTIMIZATION (TRPO):

Input: initial policy parameters θ_0, initial value function parameters $\emptyset_0$

Hyperparameters: KL-divergence limit δ, learning rate α, maximum number of backtracking steps K

for $k = 0: K$ **do**:

Collect trajectories $D_k = \{\tau_i\}$, by running policy $\pi_k = \pi(\theta_k)$ in the environment

Compute rewards-to-go $\hat{R}_t$

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\emptyset_k}$.

Estimate policy gradient as:

$$\hat{g}_k = \frac{1}{|D_k|} \sum_{\tau \in D_k} \sum_{t=0}^{T} \nabla_\theta \, log \, \pi_\theta(a_t|s_t) \, |_{\theta_k} \hat{A}_t$$

Use the conjugate gradient algorithm to compute:

$$\hat{x}_k \cong \hat{H}_k^{-1} \hat{g}_k$$

where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.

Update the policy by backtracking line search with:

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k$$

where $j \in \{0,1,2 \ldots K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.

Fit value function by regression on mean-squared error:

$$\emptyset_{k+1} = \underset{\emptyset}{arg\,min} \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \left(V_\emptyset(s_t) - \hat{R}_t\right)^2$$

typically via some gradient descent algorithm.

Algorithm 15 TRPO algorithm as presented in the literature

2.7.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [19] [20] is a state-of-the-art method in on-policy actor-critic methods. PPO is similar to TRPO conceptually, although a lot easier to implement, as well as more accurate. It can be formulated in two versions, PPO with adaptive KL penalty, which is the version most similar to TRPO and PPO with clipped objective.

PPO with adaptive KL penalty

PPO reliably determines the scaling parameter β that allows meaningful updates, yet avoids excessive drifts.

$$\Delta\theta^* = \underset{\Delta\theta}{\operatorname{argmax}}\, L_{\theta+\Delta\theta}(\theta + \Delta\theta) - \beta\big(D_{KL}(\pi_\theta|\pi_{\theta+\Delta\theta})\big)$$

It is hard to determine a single value for β that works for multiple problem settings. PPO sets a 'target divergence' δ, large enough to substantially alter the policy, but small enough for updates to be stable.

After each update, PPO checks the size of the update. If the realized KL-divergence exceeds the target divergence by more than 1.5δ, the next iteration we penalize divergence by doubling β. If KL-divergence is less than 0.75δ, the next iteration we expand the trust region by halving β.

algorithm PPO WITH ADAPTIVE KL PENALTY:

Input: policy parameters θ_0, initial KL penalty β_0, target KL-divergence δ

for $k = 0{:}\,K$ **do**:

Collect set of partial trajectories D_k on policy $\pi_k = \pi(\theta_k)$

Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm

Compute policy update

$$\theta_{k+1} = \underset{\theta}{\operatorname{argmax}}\, L_{\theta_k}(\theta) - \beta_k \bar{D}_{KL}(\theta|\theta_k)$$

by taking K steps of minibatch SGD (via Adam)

if $\bar{D}_{KL}(\theta_{k+1}|\theta_k) \geq 1.5\delta$ **then**

$$\beta_{k+1} = 2\beta_k$$

else if $\bar{D}_{KL}(\theta_{k+1}|\theta_k) \leq \delta/1.5$ **then**

$$\beta_{k+1} = \beta_k/2$$

Algorithm 16 PPO with adaptive KL penalty algorithm as presented in the literature

PPO with clipped objective

Instead of changing penalties over time, we restrict the range within which the policy can change. Advantages achieved by updates outside the clipping range are not used for updating purposes. We provide an incentive to stay relatively close to the existing policy.

algorithm PPO WITH CLIPPED OBJECTIVE:

Initial policy parameters θ_0, clipping threshold ϵ

for $k = 0,1,2 \ldots N$ **do**:

Collect set of partial trajectories D_k on policy $\pi_k = \pi(\theta_k)$

Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm

Compute policy update

$$\theta_{k+1} = \underset{\theta}{\operatorname{argmax}}\, L_{\theta_k}^{CLIP}(\theta)$$

by taking K steps of minibatch SGD (via Adam), where

$$L_{\theta_k}^{CLIP}(\theta) = \mathbb{E}_{\tau\sim\pi_k}\left[\sum_{t=0}^{T}\left[min\left(r_t(\theta)\hat{A}_t^{\pi_k}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t^{\pi_k}\right)\right]\right]$$

Algorithm 17 PPO with clipped objective algorithm as presented in the literature

2.8 Off-policy Actor-Critic methods

Off-policy Actor-Critic methods reduce variance, hence are very sample efficient.

2.8.1 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) [21] [22] is the first off-policy actor-critic to be introduced. DDPG is an off-policy algorithm and is trained with samples from a replay buffer to minimize correlations between samples. Moreover, it introduces one target critic Q-network to give consistent targets during bootstrapping of Q-values and improve stability. A key feature of DDPG is its simplicity: it requires only a straightforward actor-critic architecture [21]. Nevertheless, this simplicity makes it very sensitive to hyperparameter tuning.

algorithm DEEP DETERMINISTIC POLICY GRADIENT (DDPG):

Randomly initialize critic network $Q(s, a|\theta)$ and actor $\mu(s|\emptyset)$ with weights θ and $\emptyset$

Initialize target network Q' and μ′ with weights $\theta' \leftarrow \theta$, $\emptyset' \leftarrow \emptyset$

Initialize replay buffer D

for $t = 1{:}T$ **do**:

Take some action $a_t = \mu(s_t|\emptyset)$

Observe (s_t, a_t, s_{t+1}, r_t) and add it to D

Sample random mini-batch of N transitions (s_i, a_i, s_{i+1}, r_i) from D

Select $a_{i+1} = \mu'(s_{i+1}|\emptyset')$

Set $y_i = r_i + \gamma Q'(s_{i+1}, a_{i+1}|\theta')$ using target networks θ' and $\emptyset'$

Update critic by minimizing loss:

$$L = \frac{1}{N}\sum_i \left(y_i - Q(s_i, a_i|\theta)\right)^2$$

Update the actor policy using the sampled policy gradient:

$$\nabla_\emptyset J \cong \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta)|_{s=s_i, a=\mu(s_i)} \nabla_\emptyset \mu(s|\emptyset)|_{s_i}$$

Update the target networks:

$$\emptyset' \leftarrow \emptyset' - \alpha \sum_j \frac{dQ_{\emptyset'}}{d\emptyset'}(s_j, a_j)\left(Q_{\emptyset'}(s_j, a_j) - y_j\right)$$

$$\theta' \leftarrow \theta' + \beta \sum_j \frac{d\mu}{d\theta'}(s_j) \frac{dQ_{\emptyset'}}{da}(s_j, a)$$

Algorithm 18 DDPG algorithm as presented in the literature

2.8.2 Twin-Delayed DDPG (TD3)

Twin-Delayed DDPG (TD3) [23] is essentially the same as DDPG. Instead of updating the policy in each timestep, TD3 updates it every d timesteps. Using this simple technique, TD3 is more stable than DDPG and less sensitive to hyperparameter tuning.

algorithm TWIN-DELAYED DDPG (TD3):

Initialize critic networks $Q_{\theta_1}, Q_{\theta_2}$ and actor network $\pi_\emptyset$ with random parameters $\theta_1, \theta_2, \theta$

Initialize target networks $\theta_1' \leftarrow \theta_1, \theta_2' \leftarrow \theta_2, \emptyset' \leftarrow \emptyset$

Initialize replay buffer D

for t=1:T **do**:

Select action a with exploration noise $a \sim \pi_\emptyset(s) + \epsilon,\ \epsilon \sim \eta(0, \sigma)$ and observe reward r and new state s'

Store transition tuple (s_t, a_t, r_t, s_{t+1}) from D

Sample mini-batch of N transitions (s_i, a_i, r_i, s_{i+1}) from D

Take new actions $a_{i+1} \leftarrow \pi_\emptyset(s') + \epsilon, \epsilon \sim clip(\eta(0, \overline{\sigma}), -c, c)$

Set $y \leftarrow r + \gamma min_{j=1,2} Q_{\theta_j'}(s_{i+1}, a_{i+1})$

Update critics:

$$\theta_j \leftarrow \underset{\theta_j}{argmin} \frac{1}{N} \sum \left(y - Q_{\theta_i}(s, a)\right)^2$$

if t mod d **then**

Update $\emptyset$ by the deterministic policy gradient:

$$\nabla_\theta J(\emptyset) = \frac{1}{N} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\emptyset(s)} \nabla_\emptyset \pi_\emptyset(s)$$

Update target networks:

$$\theta_i' \leftarrow \tau\theta_i + (1 - \tau)\theta_i', \qquad i \in \{1,2\}$$

$$\emptyset' \leftarrow \tau\emptyset + (1 - \tau)\emptyset'$$

Algorithm 19 TD3 algorithm as presented in the literature

2.8.3 Soft Actor-Critic (SAC)

Soft Actor-Critic (SAC) [16] is based on the maximum entropy framework. The entropy appears in both the actor and critic.

SAC introduces a stochastic policy, meant for both maximizing expected reward and maximizing entropy. This prevents premature convergence of the policy variance. Moreover, two critic Q-networks are introduced, to mitigate positive bias in the policy improvement step, which is known to degrade performance of value based methods. The critics encourage exploration by increasing the Q-value of regions of state space that lead to high entropy behaviour. SAC also makes use of two target critic V-networks to give consistent targets during bootstrapping of Q-values and improve stability.

algorithm SOFT ACTOR-CRITIC (SAC):

Input: initial parameters for policy θ and critic $\emptyset_1, \emptyset_2$

Initialize target weights for critic $\bar{\emptyset}_1 \leftarrow \emptyset_1, \bar{\emptyset}_2 \leftarrow \emptyset_2$

Initialize empty replay buffer D

for each iteration **do**:

for each environment step **do**:

Sample action from the policy $a_t \sim \pi_\theta(a_t|s_t)$

Observe new state $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$

Store the tuple in the replay buffer $D \leftarrow D \cup (s_t, a_t, r(s_t, a_t), s_{t+1})$

for each gradient step **do**:

Update the critic parameters $\emptyset_i \leftarrow \emptyset_i - \lambda_Q \nabla_{\emptyset_i} J_Q(\emptyset_i), \quad i \in \{1,2\}$

Update policy weights $\theta \leftarrow \theta - \lambda_\pi \widehat{\nabla}_\theta J_\pi(\theta)$

Update temperature $\alpha \leftarrow \alpha - \lambda \widehat{\nabla}_\alpha J(\alpha)$

Update critic target weights $\bar{\emptyset}_i \leftarrow \tau \emptyset_i + (1-\tau)\bar{\emptyset}_i, \quad i \in \{1,2\}$

Algorithm 20 SAC algorithm as presented in the literature

2.8.4 Randomized Ensembled Double Q-Learning (REDQ)

Randomized Ensembled Double Q-Learning [24] works in conjunction with an underlying off-policy algorithm. It is model-free. It is also very sample efficient, with an Update-To-Data: $G \gg 1$.

REDQ uses an ensemble of N Q-functions. This effectively reduces the variance in the Q-function estimate. Each Q-function is randomly and independently initialized, but updated with the same target. Furthermore, the target for the Q-function includes a minimization over a random subset M of the N Q-functions. This effectively reduces over-estimation bias. The size of the subset M is kept fixed, and is referred to as the *in-target minimization parameter*. The default value $M = 2$, hence the name Double Q-Learning.

algorithm RANDOMIZED ENSEMBLED DOUBLE Q-LEARNING (REDQ):

Initialize policy θ, N Q-functions $\emptyset_i, i = 1 \dots N$, empty replay buffer D.

Set target parameters $\emptyset_{targ,i} \leftarrow \emptyset_i, i = 1 \dots N$

for each episode **do**:

Take action $a_t \sim \pi_\theta(\cdot|s_t)$. Observe reward r_t, new state s_{t+1}.

Add data to buffer: $D \leftarrow D \cup \{(s_t, a_t, r_t, s_{t+1})\}$

for G updates **do**:

Sample a mini-batch $B = \{(s, a, r, s')\}$ from D

Sample a set κ of M distinct indices from $\{1,2, \dots, N\}$

Compute the Q target y (same for all of the N Q-functions):

$$y = r + \gamma \left(\min_{i \in \kappa} Q_{\emptyset_{targ,i}}(s', \bar{a}') - \alpha \, log \pi_\theta(\bar{a}'|s') \right)$$

where $\bar{a}' \sim \pi_\theta(\cdot|s')$

for $i = 1 \dots N$ **do**:

Update $\emptyset_i$ with gradient descent using:

$$\nabla_\emptyset \frac{1}{|B|} \sum_{(s,a,r,s') \in B} \left(Q_{\emptyset_i}(s, a) - y \right)^2$$

Update target networks with:

$$\emptyset_{targ,i} \leftarrow \rho \emptyset_{targ,i} + (1 - \rho) \emptyset_i$$

Update policy parameters θ with gradient ascent using:

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\frac{1}{N} \sum_{i=1}^{N} Q_{\phi_i}(s, \bar{a}_\theta(s)) - \alpha \, log \pi_\theta(\bar{a}_\theta(s)|s) \right)$$

where $\bar{a}_\theta(s) \sim \pi_\theta(\cdot\,|s)$

Algorithm 21 REDQ algorithm as presented in the literature

2.9 Successful applications

Deep Reinforcement Learning methods have seen successful applications in many industry sectors [25], like recommender systems, computer systems, energy management, finance, healthcare, games, robotics and transportation. Furthermore, there have been successful applications in communications, networking and packet routing [26]. Deep RL methods are also used during training of advanced commercial neural network architectures, like ChatGPT.

Finally, in a study by Gartner in 2019, Deep Reinforcement Learning was shortlisted among artificial intelligence disciplines that are expected to get a lot of hype in the next 5-10 years [27]. We can expect the field to grow and real-world applications to be numerous in the future.

Chapter 3

Deep Reinforcement Learning in physics-based simulations

3.1 Introduction

Physics-based simulation provides a virtual world for predicting the outcome of real-world phenomena. Many physical systems in the real world are too complex to be investigated via analytical solutions. Through physics-based simulations, we can explore the response and performance of such systems [1]. In order to achieve a 1.7% risk of failure for its mission on Mars with the Curiosity rover, NASA simulated the “seven minutes of terror” millions of times. Moreover, an important milestone was achieved in helping computer-aided design in bioengineering and disease treatment. The life cycle of the world’s smallest free-living bacterium, Mycoplasma genitalium, was simulated.

Over the last three decades, a rapid growth in computational power has accelerated the adoption of physics-based simulation as an important tool. For instance, nowadays a GPU card in 2023 has more than 100 billion transistors. By comparison, an Intel Pentium chip from 1993 had circa 3.1 million transistors. These transistors are organized either as processing units or as cache memories. Such architectures provide incredible amounts of compute power, up to trillions of arithmetic operations per second. Furthermore, they help keeping costs low. Physics-based simulations in robotics use computationally intensive numerical operations, hence can benefit from these hardware advances.

Acquiring robot interaction data through real world experiments requires careful organization. Such experiments can be challenging and risky to the robots and experimenters. Furthermore, the quantity of training data needed for the reinforcement learning algorithms is quite large, so

acquiring this data in the real world is very time consuming. Through physics-based simulation we can generate training data at low cost. It also allows us to gain experience with a wider range of scenarios and operate in a risk-free manner.

Finally, we should note that physics-based simulations are not yet always effective. Many designs produced in simulation fail to deliver in the real world. This is also called the *simulation-to-reality gap*. Nevertheless, up-to-date there are no better alternatives to physics-based simulations in robotics for generating a lot of experience fast, in a risk-free manner while keeping costs low.

3.2 Software components

To train Deep Reinforcement Learning algorithms in physics-based simulations, we need the following software components.

MuJoCo

MuJoCo [28] stands for Multi-Joint dynamics with Contact. It provides a physics engine for running physics-based simulations. It is built with a focus on speed, accuracy and useful features to be used. Moreover, it can compute both forward and inverse dynamics.

Tianshou

Tianshou [29] is an elegant, modular framework for Deep RL, which facilitates research by being flexible and reliable for experimentation. Tianshou provides 20 Deep RL implementations with support for online and offline training, with a unified interface.

Gymnasium

Gymnasium is a standard API for a diverse collection of reference environments, including physics-based simulation environments based on MuJoCo. Gymnasium interface is simple, pythonic, and capable of representing general RL problems.

Jupyter Lab

JupyterLab is a web-based user interface that enables us to work with documents and activities such as Jupyter notebooks [30], text editors, terminals and custom components. JupyterLab is flexible, integrated and extensible. For communicating and performing interactive computing, Jupyter Notebooks are a community standard.

Tensorboard

TensorBoard helps the machine learning workflow by providing measurements and visualizations. It can track metrics like loss and accuracy, or it can visualize the model graph or project embeddings to a lower dimensional space.

3.3 MuJoCo environments

Ant

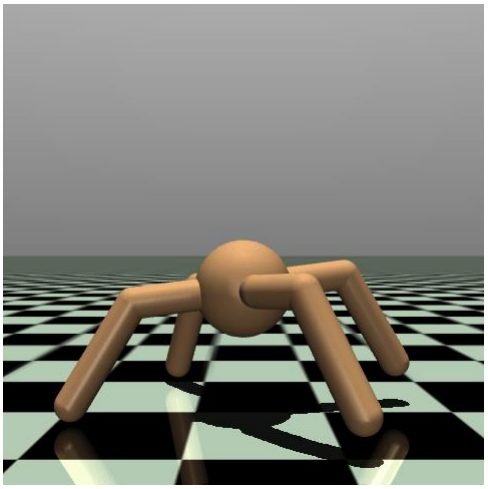


Figure 3-1 Ant environment

Description: Ant [31] is a 3D robot, which consists of one free-rotational torso with four legs attached to it. Each leg has two links.

Task: Coordinate the four legs to move in the forward (right) direction. This is achieved by applying torques on the eight hinges. The hinges connect the two links of each leg and the torso.

Action Space: Action space is a vector of size 8, representing the torques applied at the hinge joints.

Half Cheetah

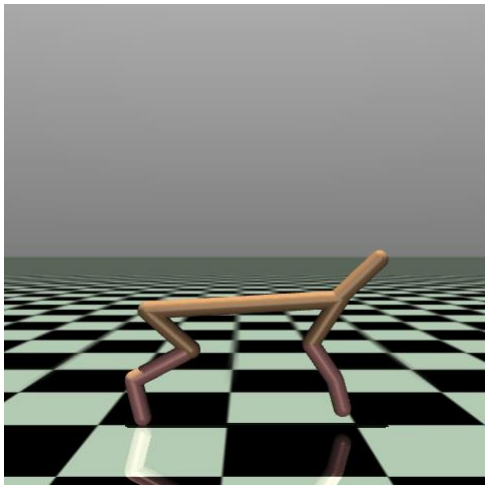


Figure 3-2 Half Cheetah environment

Description: Half Cheetah [32] is a 2-dimensional robot which consists of 9 links, 8 joints and 2 paws.

Task: Apply a torque on the joints to make the cheetah run forward (right) as fast as possible.

Details: The torso and head of the cheetah are fixed. The torque can only be applied on the 6 joints. The joints are over the front and back thighs connecting to the torso, over the shins connecting to the thighs and over the feet connecting to the shins.

Action Space: Action space is a vector of size 6, representing the torques applied between links.

Hopper

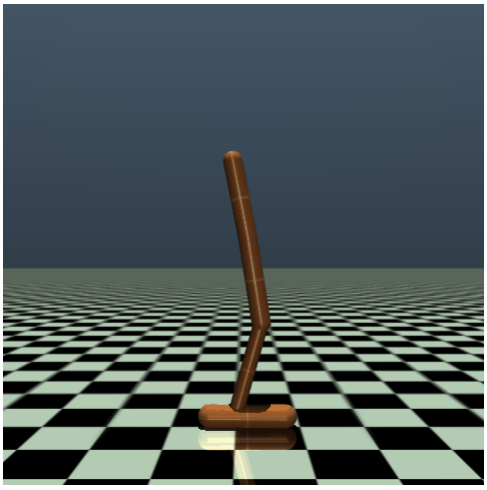


Figure 3-3 Hopper environment

Description: Hopper [33] is a two-dimensional, one-legged figure. It consist of a torso at the top, a thigh in the middle, a leg in the bottom and a single foot on which the entire body rests.

Task: Make hops that move in the forward (right) direction. Apply torques on the three hinges that connect the four body parts.

Action Space: Action space is a vector of size 3, representing the torques applied between links.

Humanoid

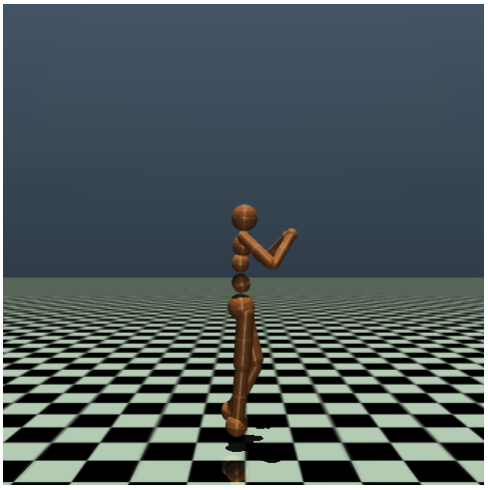


Figure 3-4 Humanoid environment

Description: 3D bipedal Humanoid robot [34] is designed to simulate a human. It has a torso (abdomen) with a pair of legs and arms. The legs each consist of two links and the arms (representing the knees and elbows respectively).

Task: Walk forward as fast as possible without falling over.

Action Space: Action space is a vector of size 17 with continuous values in [-1, 1], representing the torques applied at the hinge joints.

Humanoid Standup

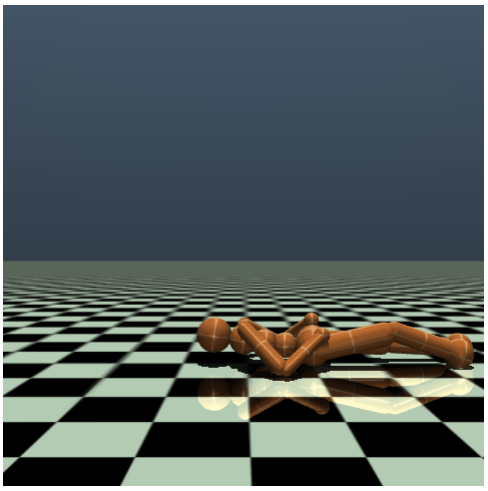


Figure 3-5 Humanoid Standup environment

Description: 3D bipedal robot [34] is designed to simulate a human. The environment starts with the humanoid laying on the ground.

Task: Make the humanoid stand up. Then, keep it standing by applying torques on the various hinges.

Action Space: Action space is a vector of size 17 with continuous values in [-1, 1], representing the torques applied at the hinge joints.

Inverted Double Pendulum

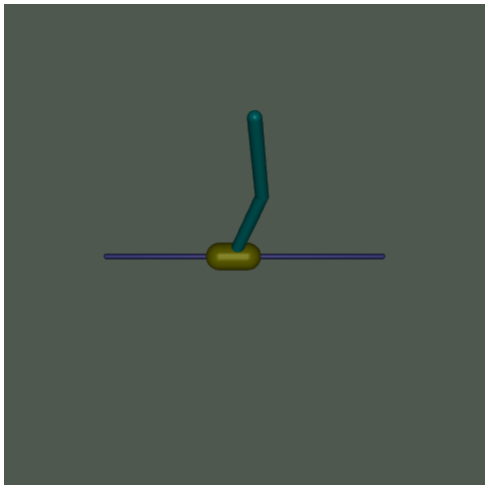


Figure 3-6 Inverted Double Pendulum environment

Description: Inverted Double Pendulum originates from [35] and it involves a cart. The cart can move linearly with a pole fixed on it. A second pole is fixed on the other end of the first one. This leaves the second pole as the only one with one free end. The cart can be pushed left or right.

Task: Balance the second pole on top of the first pole, both on top of the cart. Apply continuous forces on the cart.

Action Space: Action space is a continuous action in [-1, 1]. It represents the numerical force applied to the cart. Magnitude represents the amount of force and sign represents the direction.

Inverted Pendulum

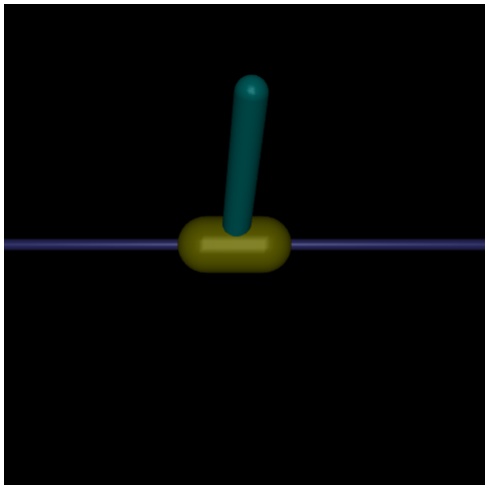


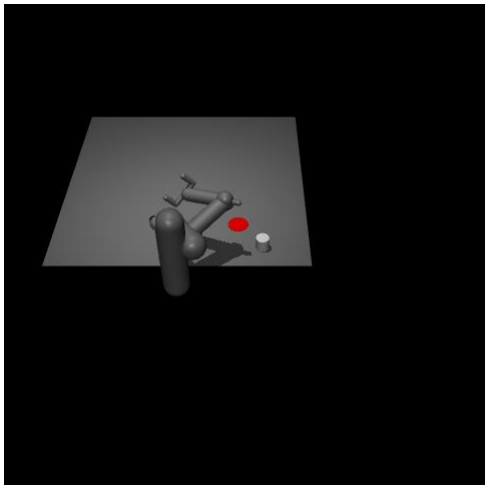
Figure 3-7 Inverted Pendulum environment

Description: This environment originates from [35] and it involves a cart. The cart can move linearly, with a pole fixed on it at one end, having another end free. The cart can be pushed left or right.

Task: Balance the pole on the top of the cart. Apply forces on the cart.

Action Space: Action space is a continuous action in [-3, 3]. Action represents the numerical force applied to the cart. Magnitude represents the amount of force. The sign represents the direction.

Pusher



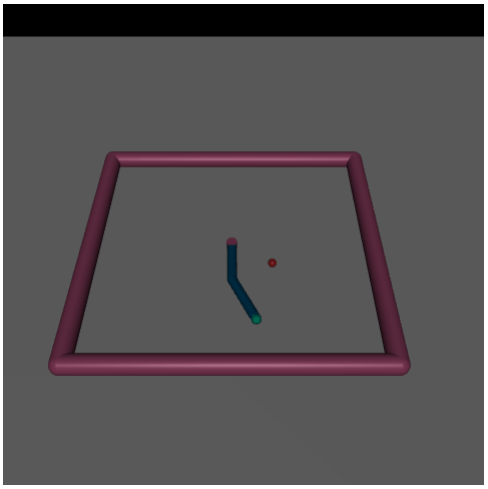
Description: "Pusher" is a multi-jointed robot arm, very similar to that of a human. The robot consists of a shoulder, an elbow, a forearm and wrist joints.

Task: Move a target cylinder, called object, to a goal position. Use the robot's end effector, called fingertip.

Action Space: Action space is a vector of size 7. It represents the torques applied at the hinge joints.

Figure 3-8 Pusher environment

Reacher



Description: Reacher is a two-jointed robot arm.

Task: Move the robot's end effector (called *fingertip*) close to a target that is spawned at a random position.

Action Space: Action space is a vector of size 2, representing the torques applied at the hinge joints.

Figure 3-9 Reacher environment

Swimmer

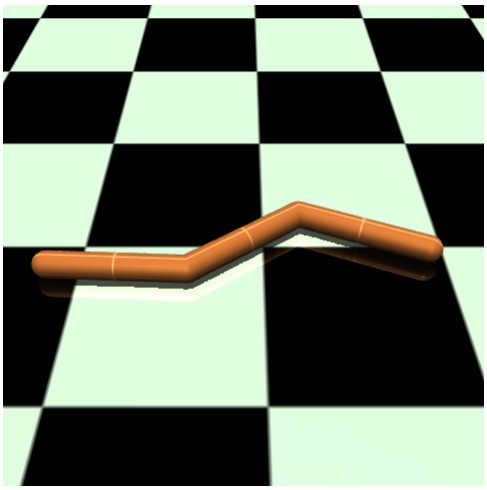


Figure 3-10 Swimmer environment

Description: Swimmer [36] consists of three or more links. The links are connected with rotor joints. Each rotor joint connect exactly two links to form a linear chain. The swimmer is suspended in a two dimensional pool. It always starts in the same position, with some deviation drawn from a uniform distribution.

Task: Move towards the right as fast as possible. Apply torque on the rotors and use the fluids friction.

Action Space: Action space is a vector of size 2, representing the torques applied between links.

Walker

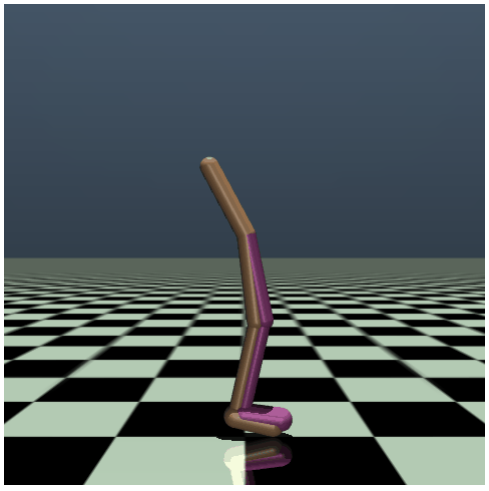


Figure 3-11 Walker environment

Description: Walker builds on Hopper [33], by adding another set of legs. This makes possible for the robot to walk forward instead of hop. There is a single torso at the top, with two legs splitting after the torso. Furthermore, there are two thighs in the middle below the torso. Two legs are in the bottom below the thighs. Finally, two feet are attached to the legs on which the entire body rests.

Task: Coordinate feet, legs and thighs to move in the forward (right) direction. Apply torques on the six hinges, which connect the six body parts.

Action Space: Action space is a vector of size 6. It represents the torques applied at the hinge joints.

3.4 Hardware

To speed up training, we trained on a machine with GPU acceleration. GPU acceleration substantially speeds up training, because of hardware-level parallelization and faster I/O rates.

AMD Ryzen Threadripper 3970X

- Base speed: 3.70 GHz
- 32 Cores and 64 Logical processors
- L1 cache: 2.0 MB, L2 cache: 16.0 MB, L3 cache: 128 MB

NVIDIA RTX A6000

- Shared GPU memory: 64 GB
- NVIDIA Ampere Architecture Based CUDA Cores, Second-Generation RT Cores, Third-Generation Tensor Cores
- 38.7 TFLOPs of FP32 performance

3.5 Methodology

In this chapter, the methodology follows the general principles of the MuJoCo benchmark from Tianshou framework [29]. I benchmarked 5 on-policy learning methods and 4 off-policy learning methods in 11 out of 11 environments from the Gymnasium MuJoCo task suite provided by Gymnasium. Each algorithm keeps the same hyperparameters in all environments.

On-policy learning methods are trained for 100 epochs with 30000 steps per epoch, for a total of 3 million timesteps. The discount factor gamma γ of future rewards is kept at 0.99 for all methods. We choose Adam optimizer as our gradient descent method for training actor and critic methods. Buffer size is kept at 4096. The policy is a two-layer neural network with size $[64, 64]$.

Off-policy learning methods are trained for 200 epochs with 5000 steps per epoch, for a total of 1 million timesteps. The discount factor gamma γ of future rewards is kept at 0.99 for all

methods. We choose Adam optimizer as our gradient descent method for training actor and critic methods. Buffer size is kept at 1000000. Except for REINFORCE, which contains only one actor network, the actor and critic networks are a two-layer neural network with size $[256, 256]$.

Learning rates for on-policy learning methods are presented below:

Table 3-1 Learning rates for on-policy learning methods in MuJoCo experiments

On-policy learning methods	Learning rates α (actor and critic)
REINFORCE	10^{-3} (only actor)
A2C	$7 * 10^{-4}$
NPG	10^{-3}
PPO	$3 * 10^{-4}$
TRPO	10^{-3}

Learning rates for off-policy learning methods are presented below:

Table 3-2 Learning rates for off-policy learning methods in MuJoCo experiments

Off-policy learning methods	Learning rates α (actor)	Learning rates α (critic)
DDPG	10^{-3}	10^{-3}
TD3	$3 * 10^{-4}$	$3 * 10^{-4}$
SAC	10^{-3}	10^{-3}
REDQ	10^{-3}	10^{-3}

The choice of learning rate values is made in accordance with the original papers.

3.6 Experimental results

3.6.1 Ant-v4

On-policy learning methods

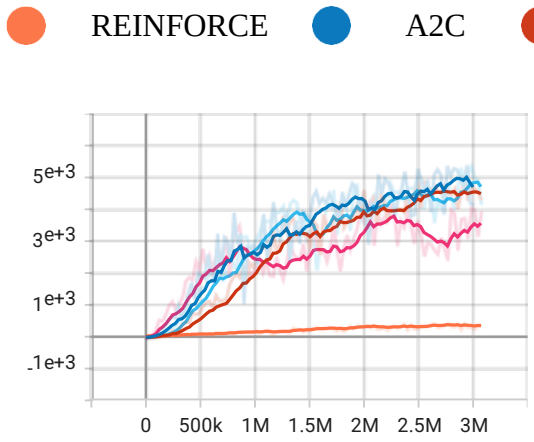


Figure 3-12 Mean rewards (testing) during timesteps

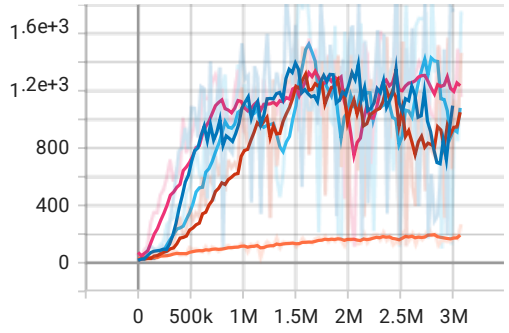


Figure 3-13 Std. deviation of rewards (testing) during timesteps

Off-policy learning methods

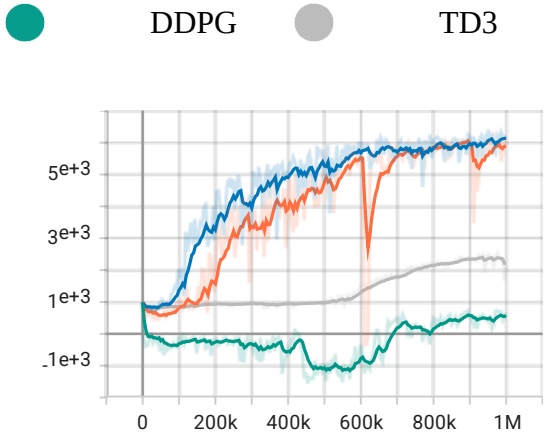


Figure 3-14 Mean rewards (testing) during timesteps

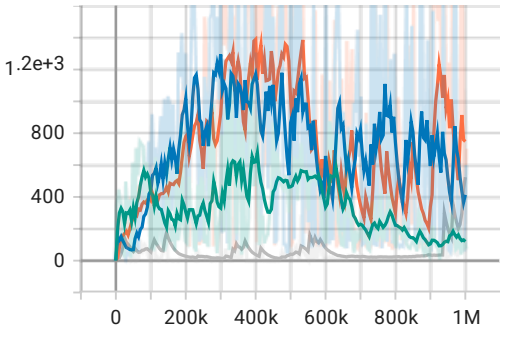


Figure 3-15 Std. deviation of rewards (testing) during timesteps

3.6.2 HalfCheetah-v4

On-policy learning methods

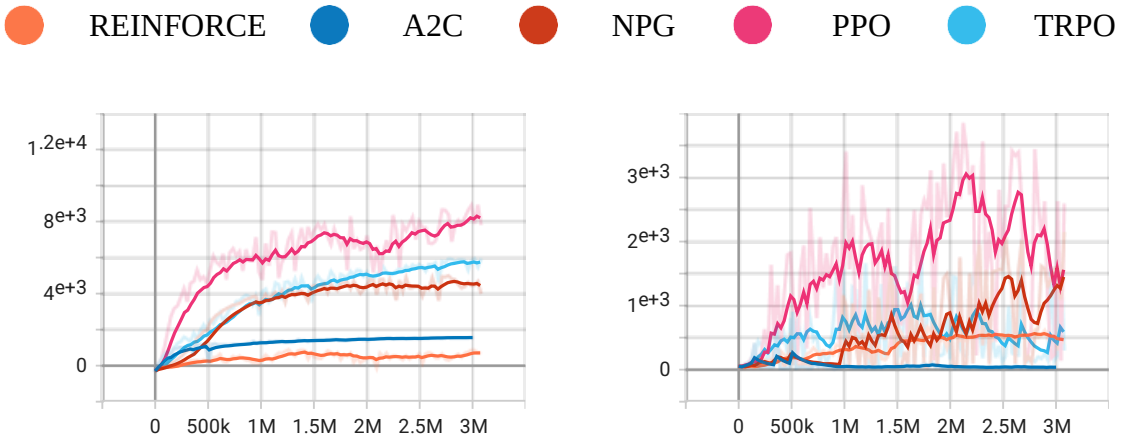


Figure 3-16 Mean rewards (testing) during timesteps

Figure 3-17 Std. deviation of rewards (testing) during timesteps

Off-policy learning methods

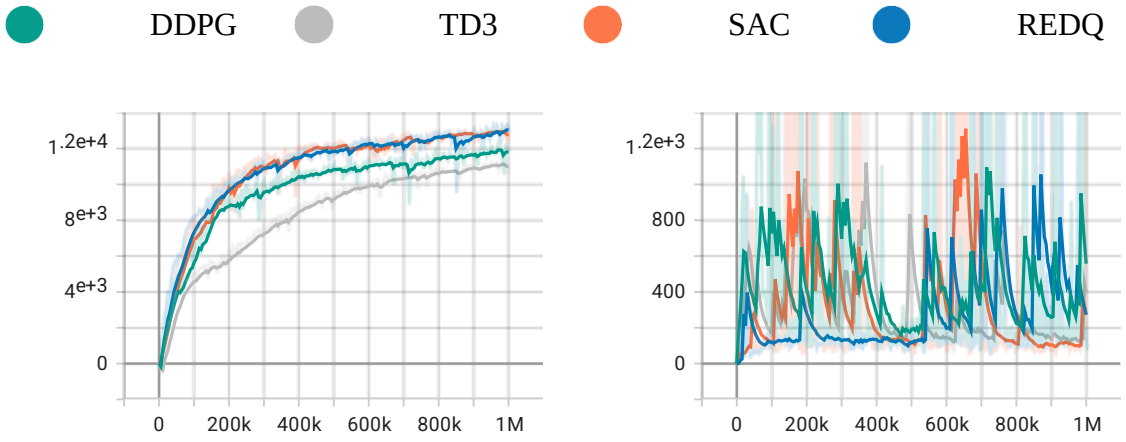


Figure 3-18 Mean rewards (testing) during timesteps

Figure 3-19 Std. deviation of rewards (testing) during timesteps

3.6.3 Hopper-v4

On-policy learning methods

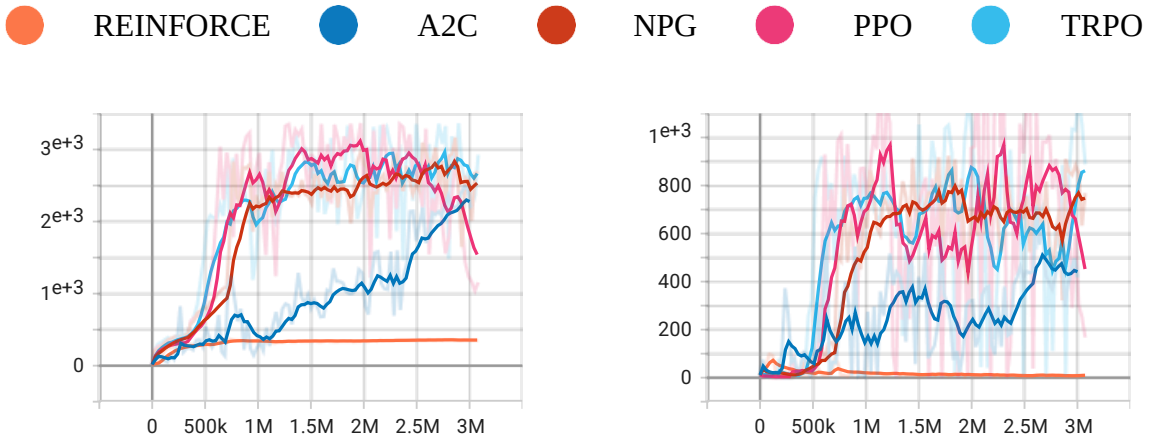


Figure 3-20 Mean rewards (testing) during timesteps

Figure 3-21 Std. deviation of rewards (testing) during timesteps

Off-policy learning methods

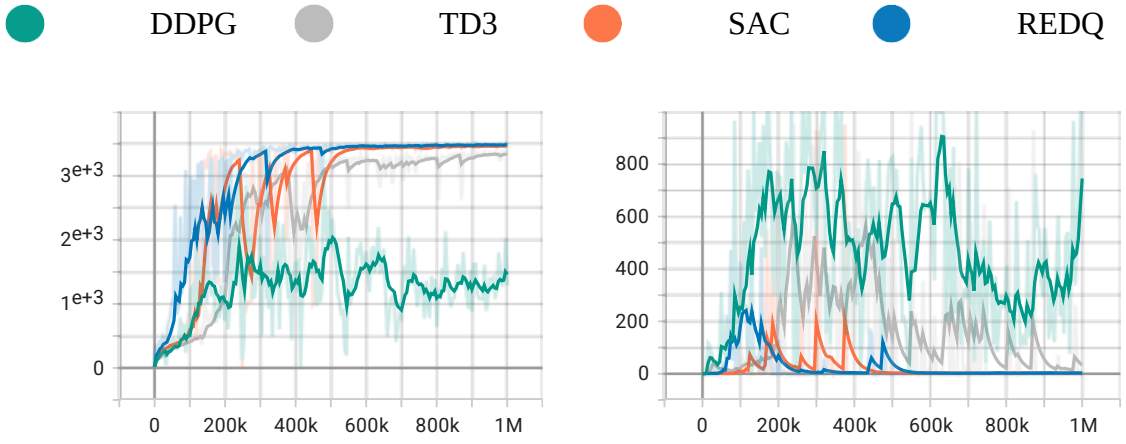


Figure 3-22 Mean rewards (testing) during timesteps

Figure 3-23 Std. deviation of rewards (testing) during timesteps

3.6.4 HumanoidStandup-v4

On-policy learning methods

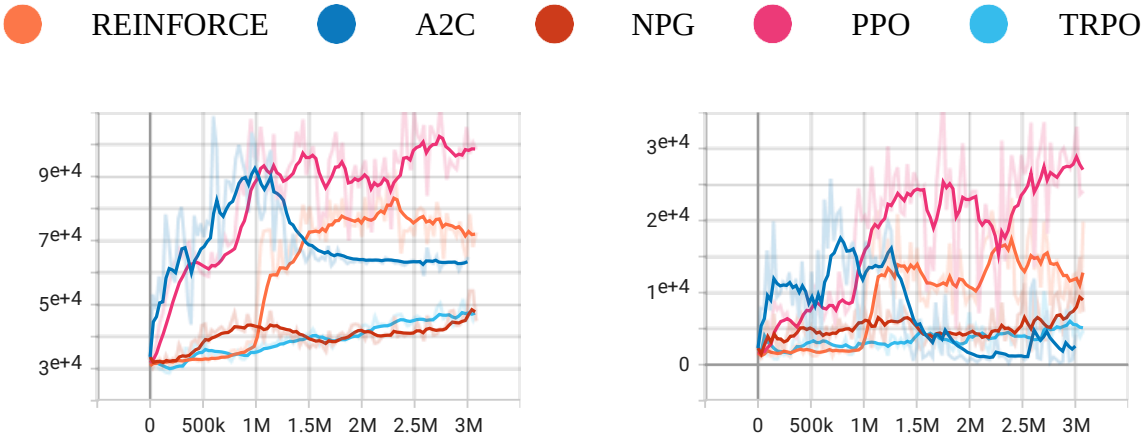


Figure 3-24 Mean rewards (testing) during timesteps

Figure 3-25 Std. deviation of rewards (testing) during timesteps

Off-policy learning methods

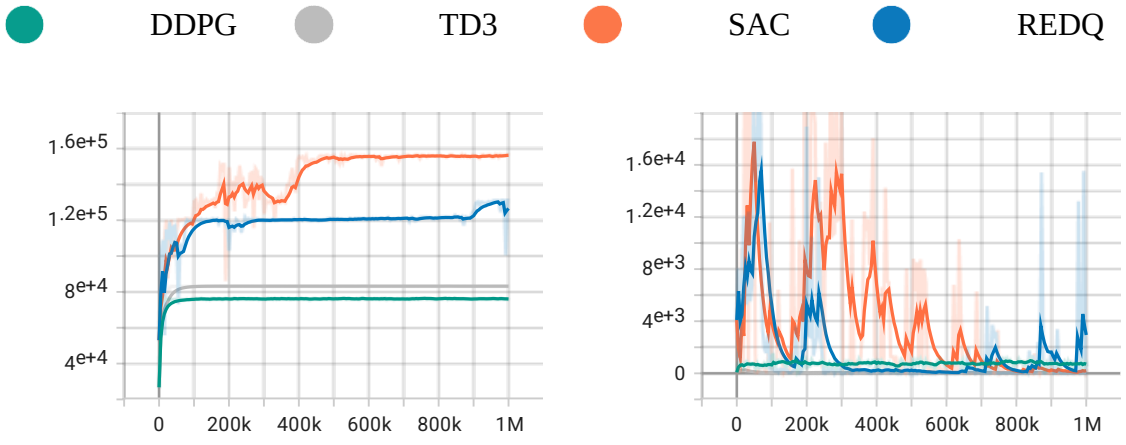


Figure 3-26 Mean rewards (testing) during timesteps

Figure 3-27 Std. deviation of rewards (testing) during timesteps

3.6.5 Humanoid-v4

On-policy learning methods

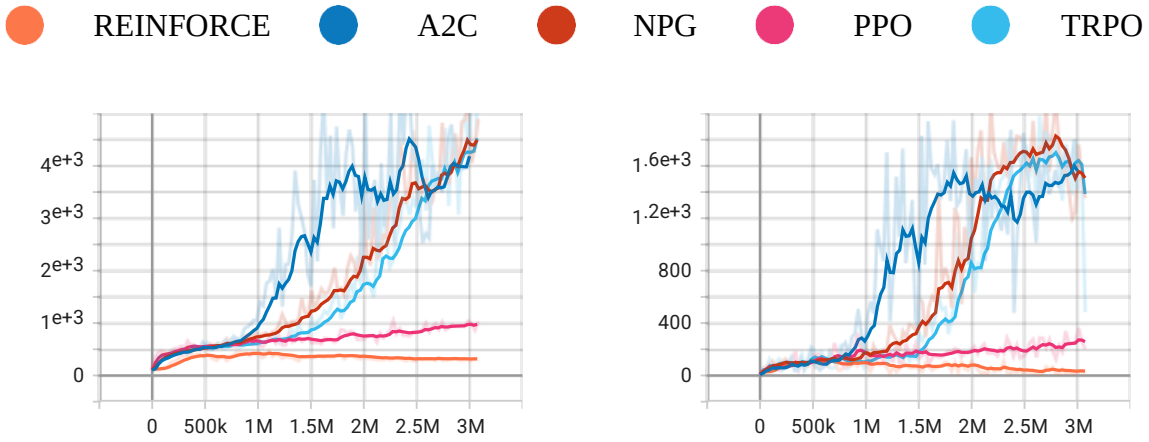


Figure 3-28 Mean rewards (testing) during timesteps

Figure 3-29 Std. deviation of rewards (testing) during timesteps

Off-policy learning methods

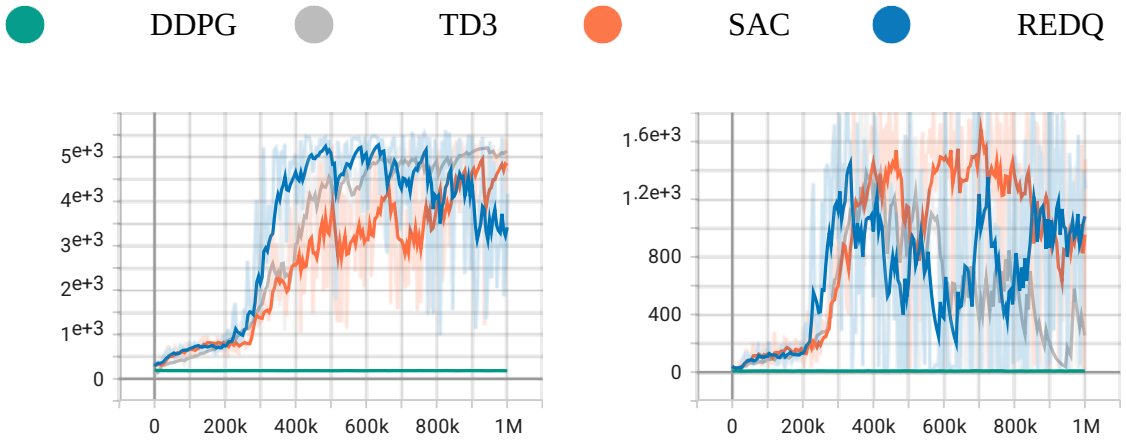


Figure 3-30 Mean rewards (testing) during timesteps

Figure 3-31 Std. deviation of rewards (testing) during timesteps

3.6.6 InvertedDoublePendulum-v4

On-policy learning methods

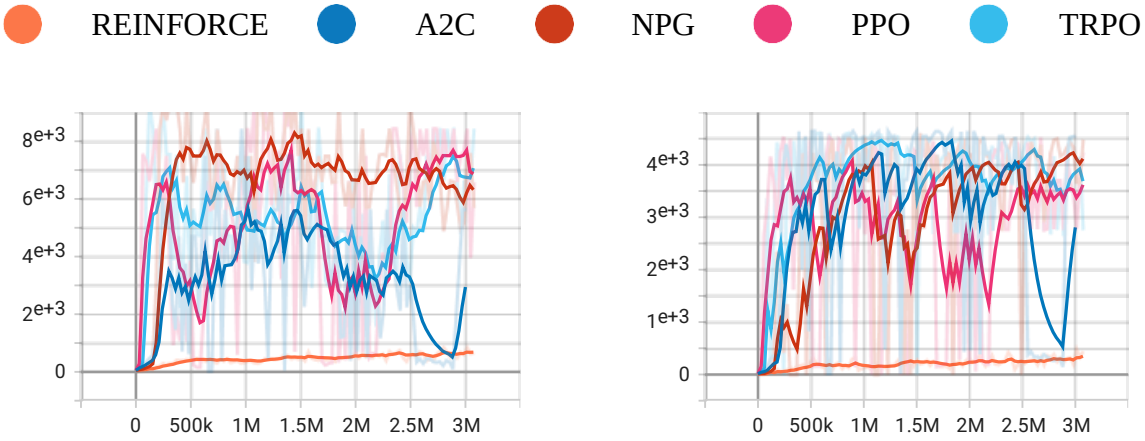


Figure 3-32 Mean rewards (testing) during timesteps

Figure 3-33 Std. deviation of rewards (testing) during timesteps

Off-policy learning methods

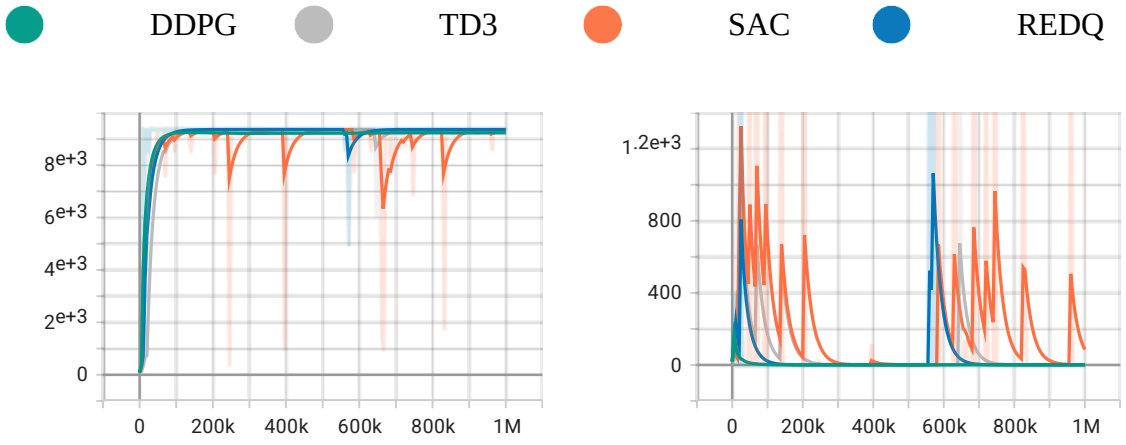


Figure 3-34 Mean rewards (testing) during timesteps

Figure 3-35 Std. deviation of rewards (testing) during timesteps

3.6.7 InvertedPendulum-v4

On-policy learning methods

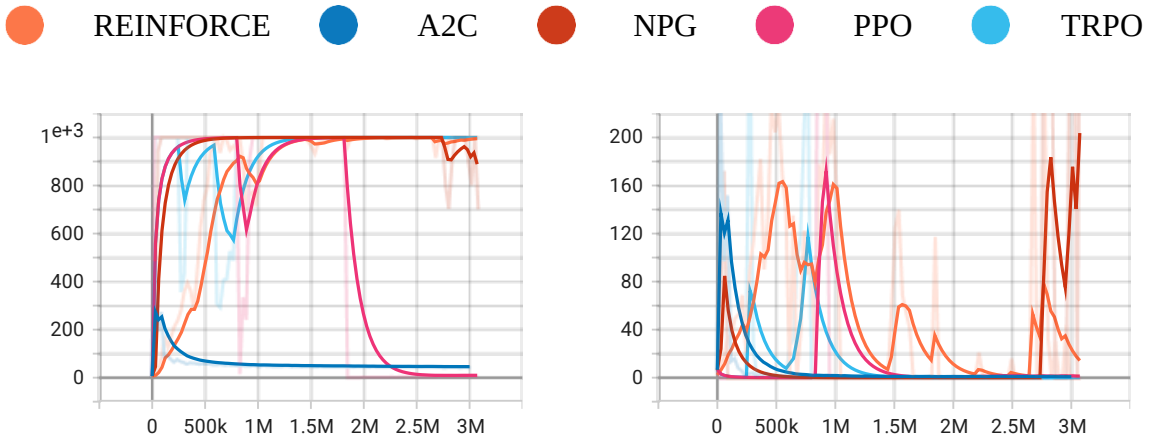


Figure 3-36 Mean rewards (testing) during timesteps

Figure 3-37 Std. deviation of rewards (testing) during timesteps

Off-policy learning methods

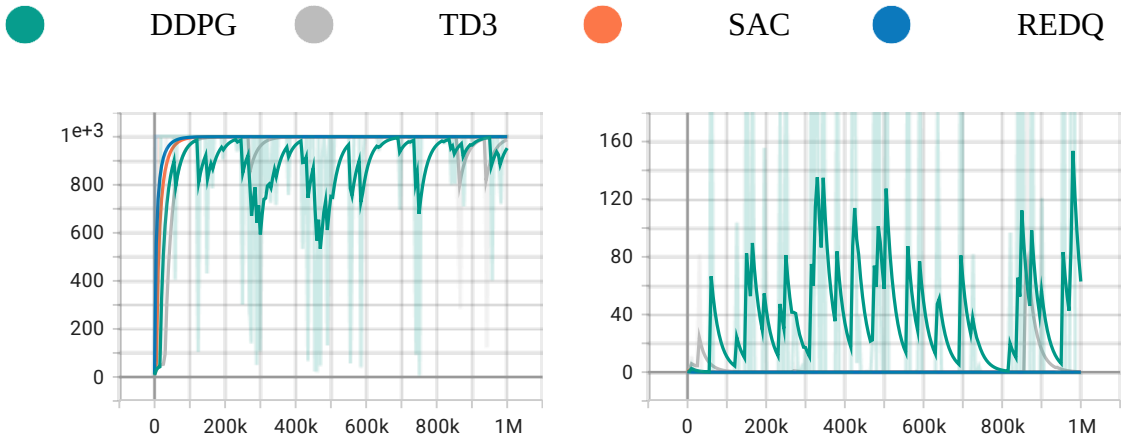


Figure 3-38 Mean rewards (testing) during timesteps

Figure 3-39 Std. deviation of rewards (testing) during timesteps

3.6.8 Pusher-v4

On-policy learning methods

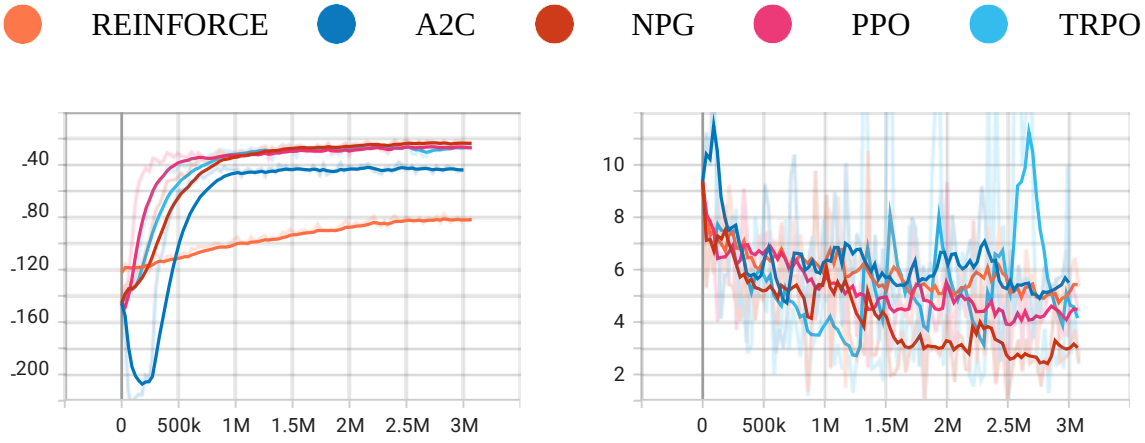


Figure 3-40 Mean rewards (testing) during timesteps

Figure 3-41 Std. deviation of rewards (testing) during timesteps

Off-policy learning methods

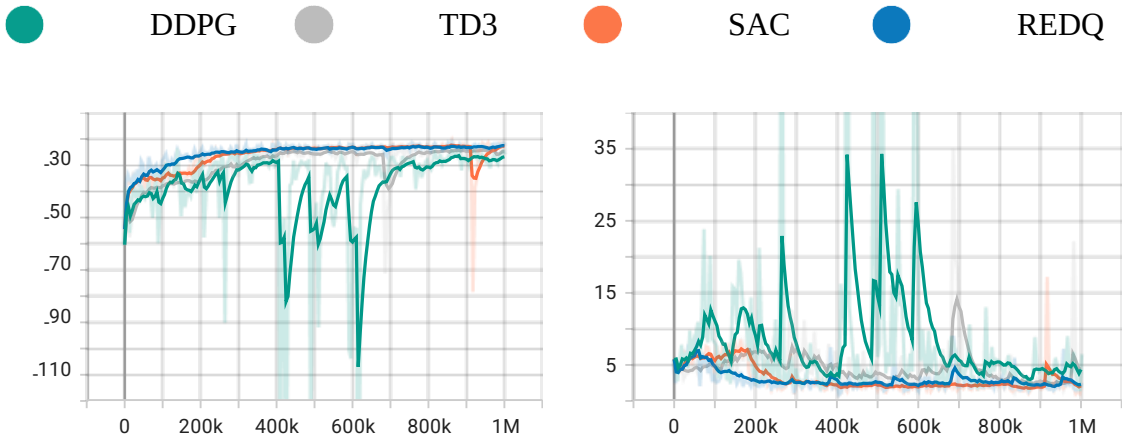


Figure 3-42 Mean rewards (testing) during timesteps

Figure 3-43 Std. deviation of rewards (testing) during timesteps

3.6.9 Reacher-v4

On-policy learning methods

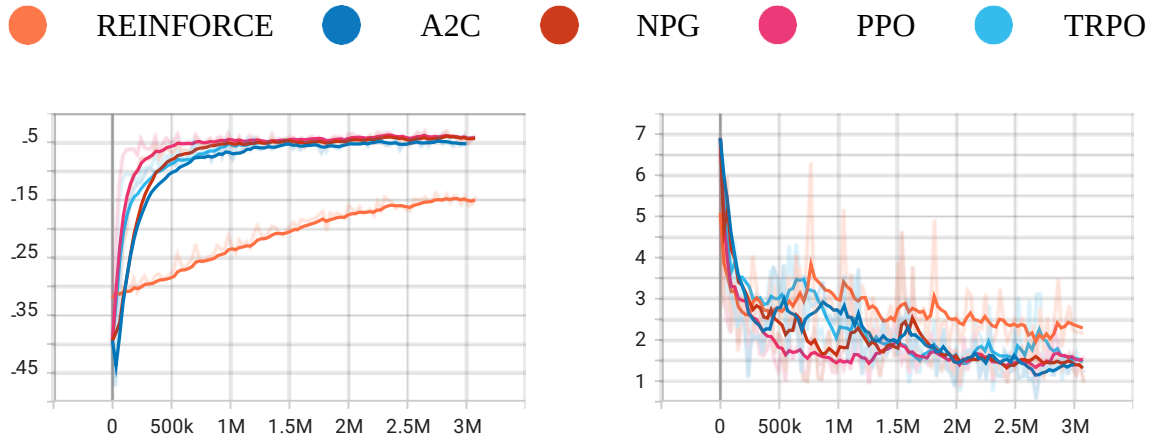


Figure 3-44 Mean rewards (testing) during timesteps

Figure 3-45 Std. deviation of rewards (testing) during timesteps

Off-policy learning methods

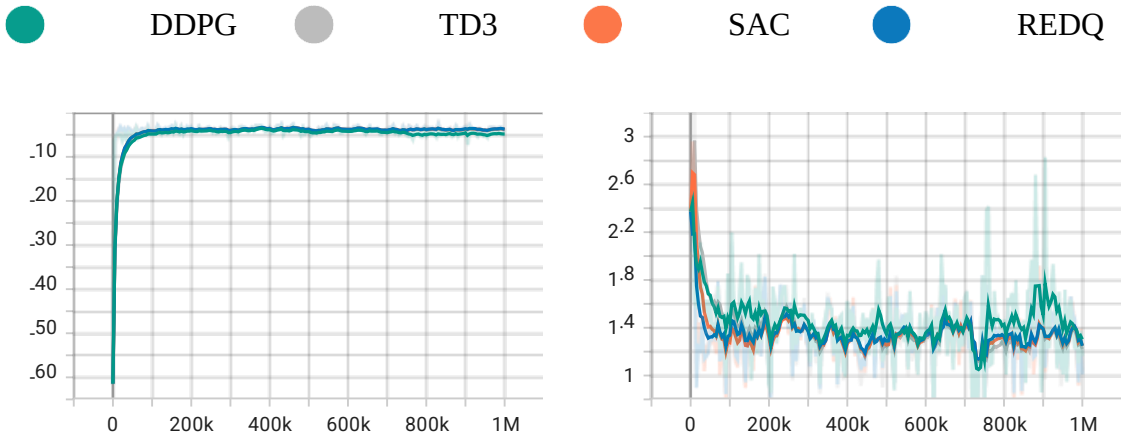


Figure 3-46 Mean rewards (testing) during timesteps

Figure 3-47 Std. deviation of rewards (testing) during timesteps

3.6.10 Swimmer-v4

On-policy learning methods

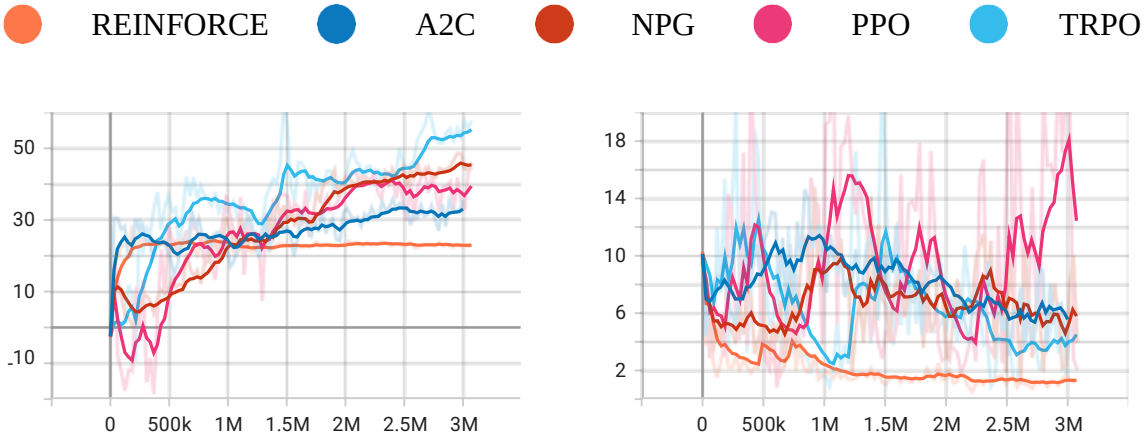


Figure 3-48 Mean rewards (testing) during timesteps

Figure 3-49 Std. deviation of rewards (testing) during timesteps

Off-policy learning methods

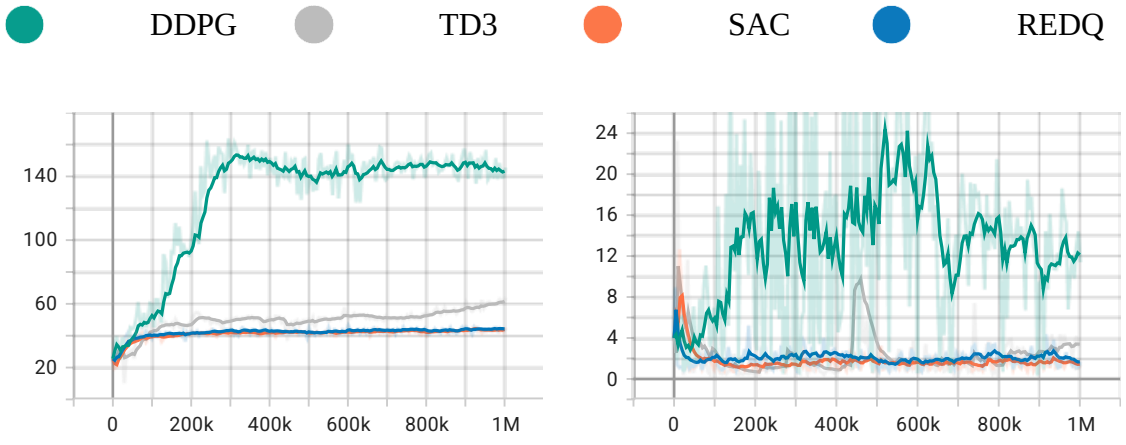


Figure 3-50 Mean rewards (testing) during timesteps

Figure 3-51 Std. deviation of rewards (testing) during timesteps

3.6.11 Walker2d-v4

On-policy learning methods

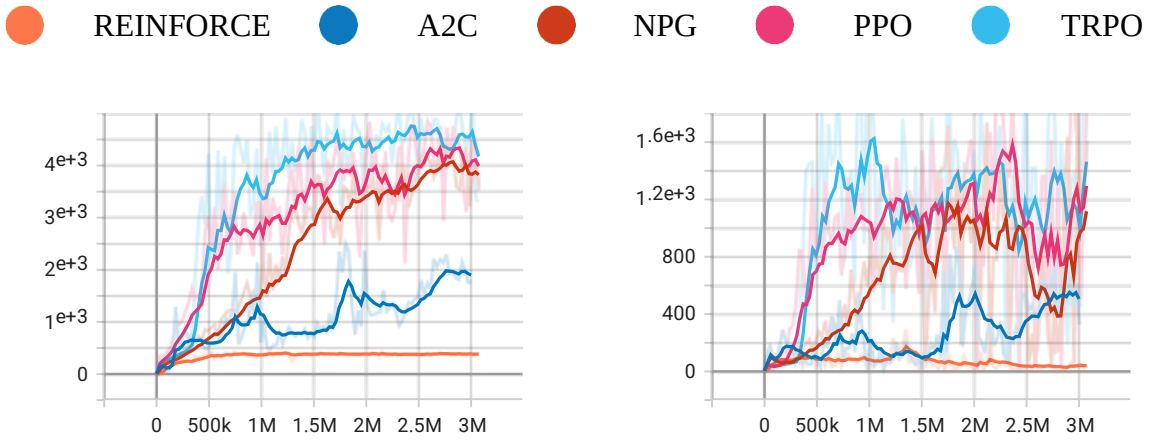


Figure 3-52 Mean rewards (testing) during timesteps

Figure 3-53 Std. deviation of rewards (testing) during timesteps

Off-policy learning methods

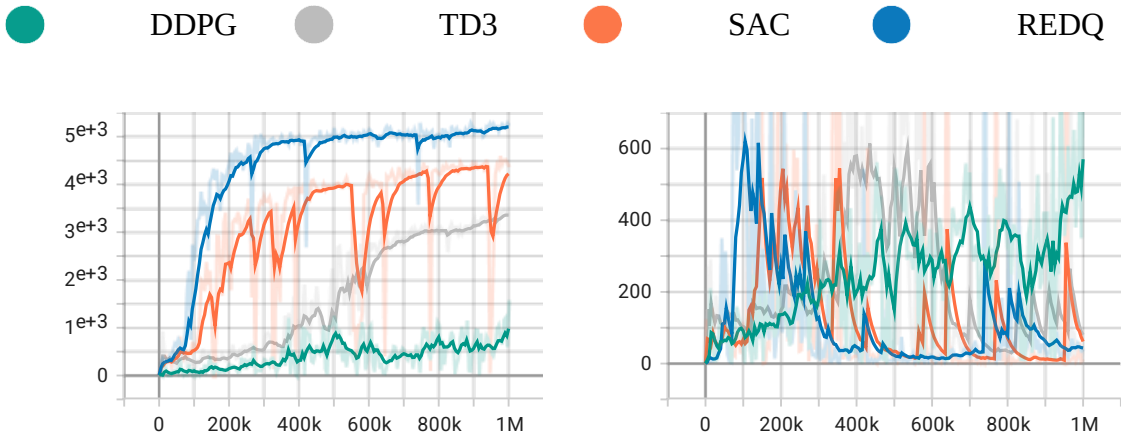


Figure 3-54 Mean rewards (testing) during timesteps

Figure 3-55 Std. deviation of rewards (testing) during timesteps

3.6.12 Best scores

On-policy learning methods

Table 3-3 Best scores achieved by on-policy learning methods in MuJoCo experiments

	REINFORCE	A2C	NPG	TRPO	PPO
Ant-v4	443.07 ± 245.38	5377.74 ± 194.95	4898.47 ± 120.45	5340.45 ± 132.30	4303.13 ± 116.90
HalfCheetah-v4	909.51 ± 125.96	1575.62 ± 29.65	5007.75 ± 117.07	5949.44 ± 232.49	8919.43 ± 143.83
Hopper-v4	377.14 ± 65.59	2485.17 ± 449.64	3169.62 ± 323.34	3366.23 ± 4.74	3368.42 ± 13.95
Humanoid-v4	484.12 ± 109.99	5688.02 ± 485.07	5256.30 ± 1253.43	5508.71 ± 490.57	1060.80 ± 208.83
HumanoidStandup-v4	92332.42 ± 16033.78	108764.14 ± 3925.53	54425.88 ± 14916.48	51864.43 ± 8294.39	117003.29 ± 29036.77
InvertedDoublePendulum-v4	879.02 ± 387.30	8429.23 ± 2788.62	9357.32 ± 0.92	9358.51 ± 1.34	9345.79 ± 0.37
InvertedPendulum-v4	1000.00 ± 0.00	486.10 ± 240.95	1000.00 ± 0.00	1000.00 ± 0.00	1000.00 ± 0.00
Pusher-v4	-77.75 ± 5.64	-38.17 ± 2.49	-21.63 ± 1.65	-23.96 ± 1.55	-22.99 ± 2.10
Reacher-v4	-13.79 ± 1.71	-4.04 ± 1.32	-3.14 ± 1.47	-3.31 ± 1.45	-3.06 ± 1.16
Swimmer-v4	28.48 ± 4.17	35.91 ± 5.61	48.55 ± 2.20	64.28 ± 22.67	44.87 ± 2.07
Walker2d-v4	450.75 ± 81.01	2509.91 ± 473.37	4350.40 ± 68.21	5275.59 ± 34.14	4853.45 ± 52.62
Total scores: (excluding HumanoidStandup-v4)	**4480.55 ± 1026.75**	**26545.49 ± 4671.67**	**33063.64 ± 1888.74**	**35835.94 ± 921.25**	**32869.84 ± 541.83**

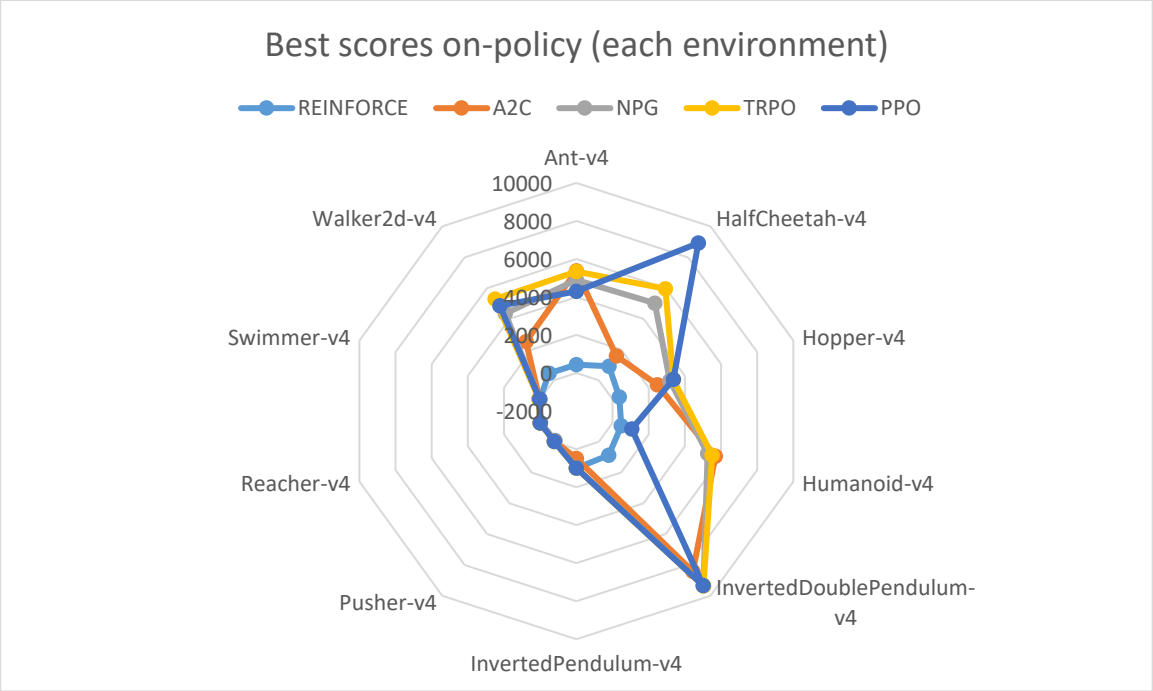


Figure 3-56 Best scores of on-policy learning methods in each environment

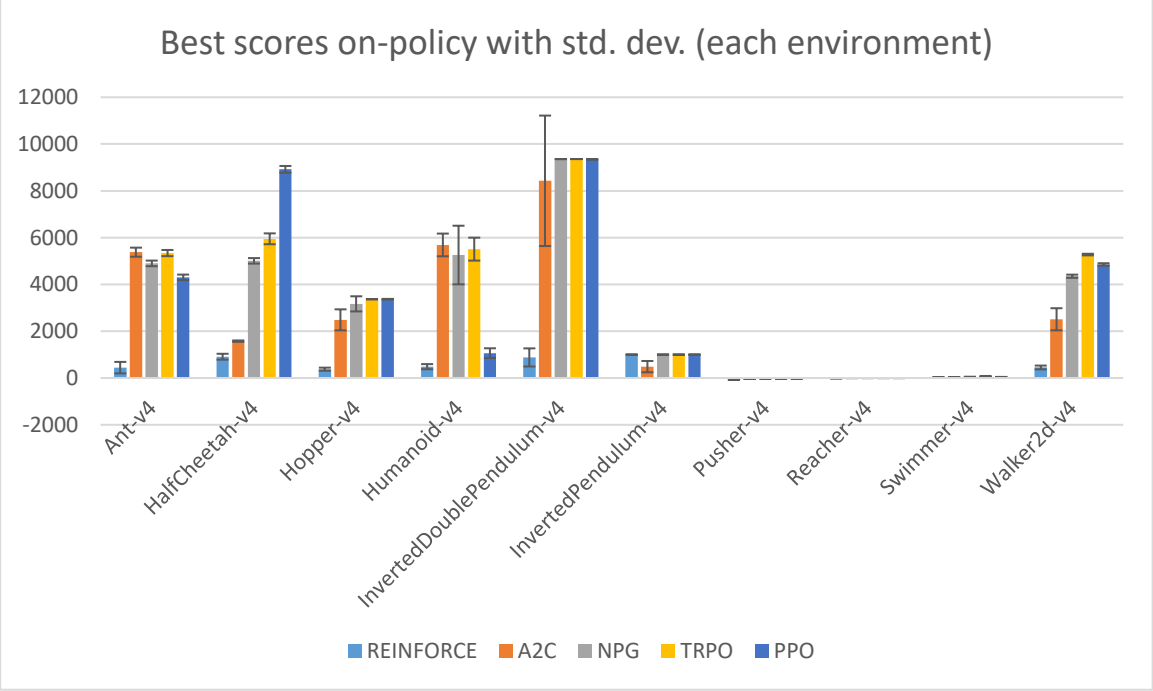


Figure 3-57 Best scores of on-policy learning methods in each environment, with standard deviations as confidence bounds

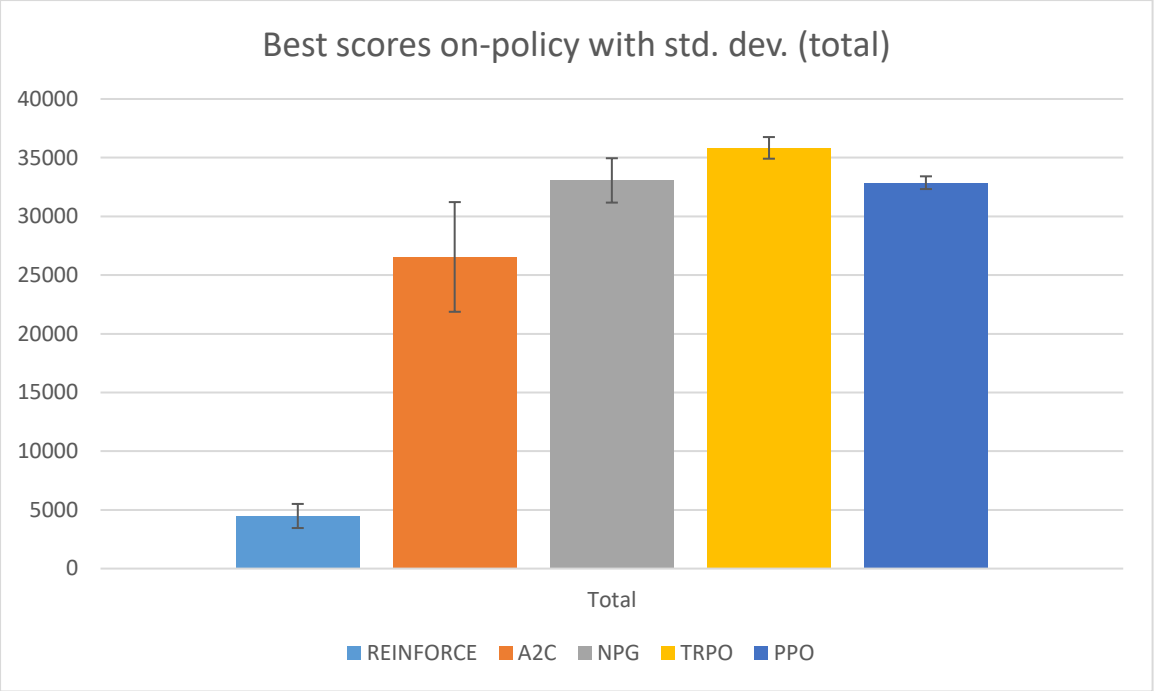


Figure 3-58 Best scores of on-policy learning methods in total (all environments), with standard deviations as confidence bounds

We train on-policy learning methods for 3M timesteps in each environment. Best scores were achieved by TRPO, then NPG, then PPO, then A2C and finally REINFORCE. Actually, TRPO, NPG and PPO are all high performing and deliver very similar best scores. The lowest standard deviations were achieved by PPO, then TRPO, then NPG, then A2C and finally REINFORCE. The lower the standard deviation, the less variance and more accurate the methods are. Actually, PPO and TRPO deliver similar standard deviations, with NPG a close third.

The difference of best scores between PPO and NPG is very small and PPO is more accurate than NPG. When accuracy is important, we can expect PPO to be a better choice than NPG in most cases. On the downside, we show in subsequent graphs that PPO is 3 times slower than NPG.

Off-policy learning methods

Table 3-4 Best scores achieved by off-policy learning methods in MuJoCo experiments

	DDPG	TD3	SAC	REDQ
Ant-v4	986.55 ± 1.43	2548.25 ± 38.28	6256.40 ± 110.89	6405.20 ± 75.33
HalfCheetah-v4	12441.85 ± 134.25	11335.22 ± 169.44	13071.48 ± 146.27	13314.53 ± 161.15
Hopper-v4	3134.90 ± 21.06	3380.07 ± 74.12	3493.66 ± 4.21	3512.44 ± 3.39
Humanoid-v4	201.05 ± 4.12	5254.40 ± 22.15	5487.84 ± 24.51	5601.18 ± 18.28
HumanoidStandup-v4	76912.81 ± 1190.52	83283.39 ± 55.78	156740.09 ± 10.37	131165.58 ± 66.21
InvertedDoublePendulum-v4	9348.58 ± 1.19	9359.86 ± 0.09	9359.87 ± 0.09	9359.96 ± 0.03
InvertedPendulum-v4	1000.00 ± 0.00	1000.00 ± 0.00	1000.00 ± 0.00	1000.00 ± 0.00
Pusher-v4	-23.94 ± 2.55	-22.38 ± 1.58	-20.69 ± 1.61	-21.42 ± 2.05
Reacher-v4	-3.11 ± 1.51	-2.78 ± 1.57	-2.63 ± 1.63	-2.56 ± 1.64
Swimmer-v4	163.47 ± 1.32	64.20 ± 3.37	45.89 ± 2.14	46.89 ± 1.70
Walker2d-v4	1581.38 ± 879.84	3458.46 ± 16.18	4584.96 ± 20.26	5275.41 ± 29.25
Total scores: (excluding HumanoidStandup-v4)	**28830.73 ± 1047.27**	**36375.3 ± 326.78**	**43276.78 ± 311.61**	**44491.63 ± 292.82**

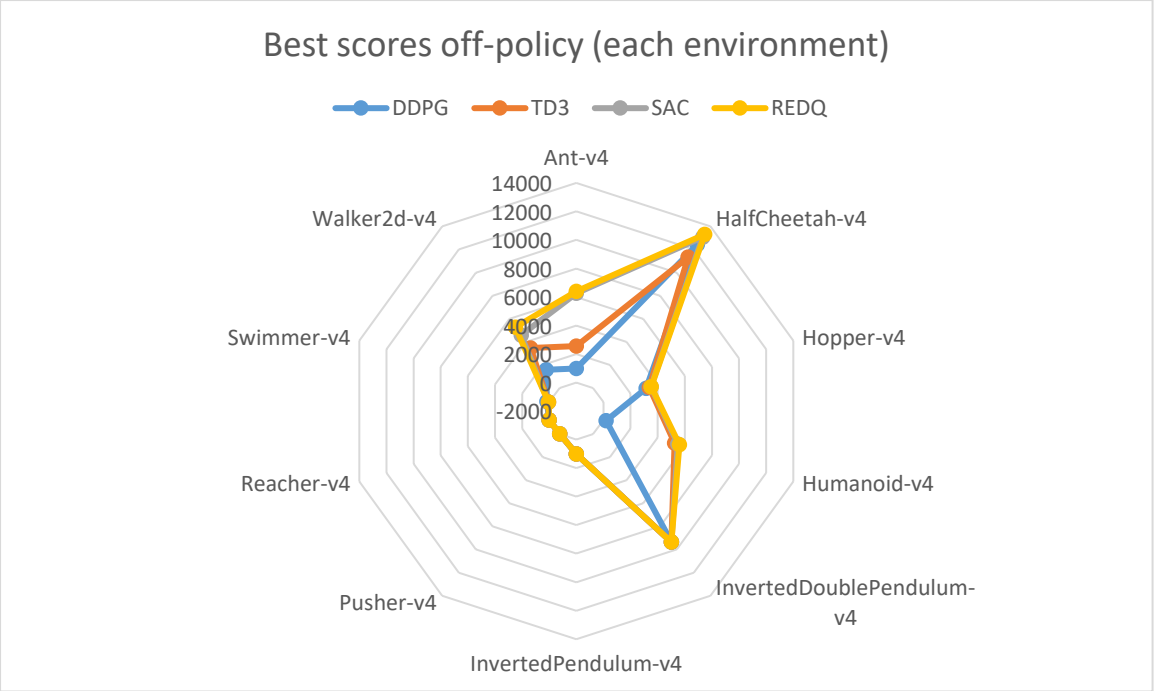


Figure 3-59 Best scores of off-policy learning methods in each environment

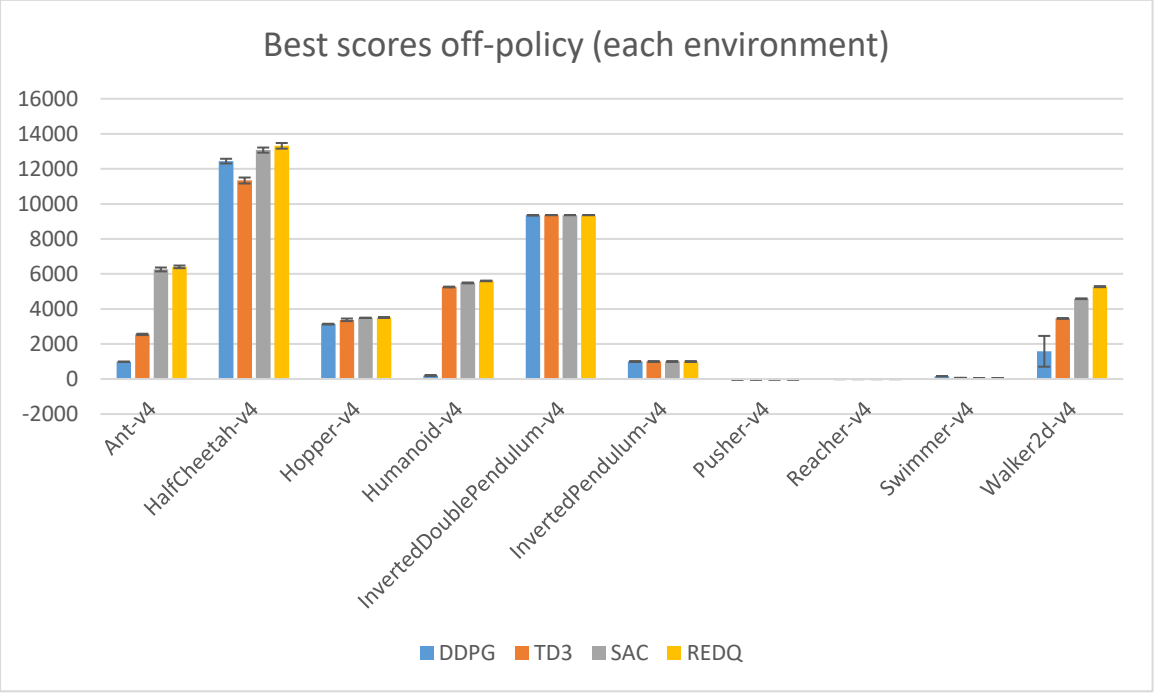


Figure 3-60 Best scores of off-policy learning methods in each environment, with standard deviations as confidence bounds

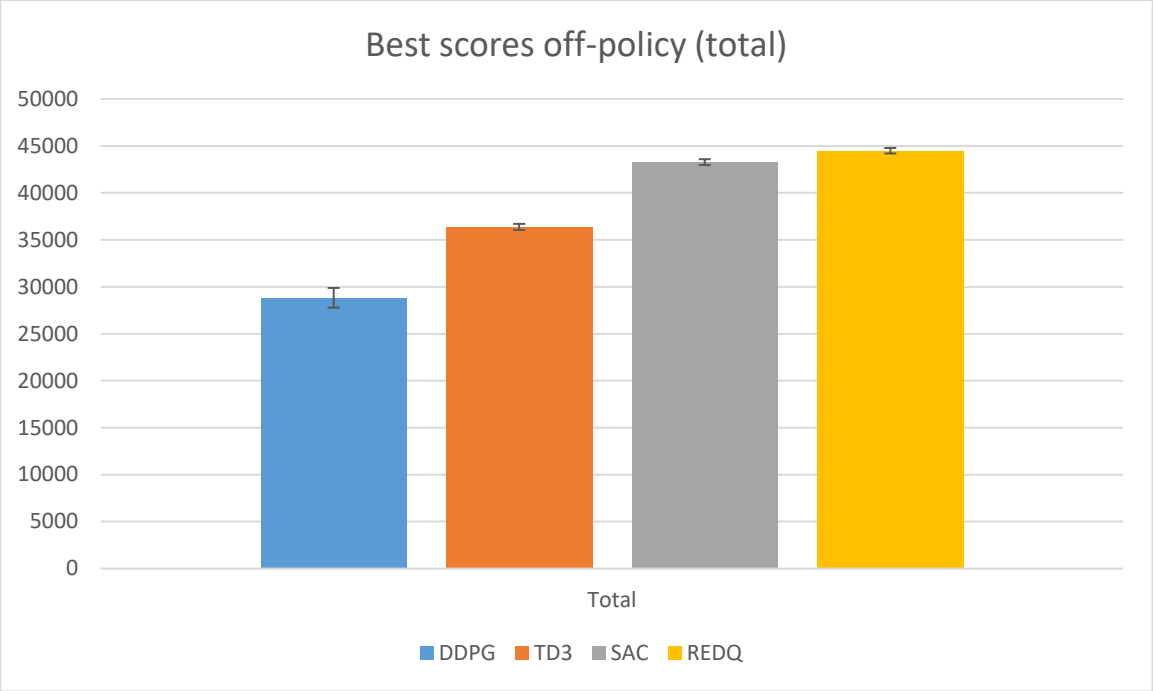


Figure 3-61 Best scores of off-policy learning methods in total (all environments), with standard deviations as confidence bounds

We train off-policy learning methods for 1M timesteps in each environment. Best scores were achieved by REDQ, then SAC, then TD3 and finally DDPG. Actually, REDQ and SAC are both high performing and deliver very similar best scores. The lowest standard deviations were achieved by REDQ, then SAC, then TD3 and finally DDPG. The lower the standard deviation, the less variance and more accurate the methods are. Actually, REDQ and SAC deliver similar standard deviations, with TD3 a close third.

Compared to on-policy learning methods, off-policy learning methods achieve better scores with lower standard deviation. We conclude that off-policy learning methods have better accuracy and lower variance than on-policy learning methods. Furthermore, off-policy learning methods are able to surpass the scores of on-policy learning methods within less training timesteps, which means they have better sample efficiency.

3.6.13 Running times

On-policy learning methods

Table 3-5 Running times for on-policy learning methods in MuJoCo experiments

Running time (h:mm:ss)	REINFORCE	A2C	NPG	TRPO	PPO
Ant-v4	0:13:20	0:22:30	0:20:59	0:20:55	1:07:36
HalfCheetah-v4	0:12:26	0:18:27	0:17:15	0:17:11	1:05:03
Hopper-v4	0:07:20	0:18:26	0:17:44	0:18:02	1:03:06
Humanoid-v4	0:08:32	0:22:23	0:22:07	0:22:05	1:07:37
HumanoidStandup-v4	0:16:44	0:25:04	0:29:26	0:28:07	1:19:32
InvertedDoublePendulum-v4	0:06:07	0:17:21	0:17:31	0:17:41	1:04:02
InvertedPendulum-v4	0:09:08	0:14:26	0:16:47	0:17:13	1:02:22
Pusher-v4	0:09:45	0:15:35	0:13:54	0:17:45	1:09:01
Reacher-v4	0:06:02	0:14:01	0:12:29	0:12:43	0:58:12
Swimmer-v4	0:11:52	0:20:46	0:19:12	0:19:24	1:07:40
Walker2d-v4	0:07:48	0:18:21	0:17:48	0:18:17	1:04:57
Average	**0:09:55**	**0:18:51**	**0:18:39**	**0:19:02**	**1:06:17**
Total	**1:49:03**	**3:27:20**	**3:25:11**	**3:29:22**	**12:09:07**

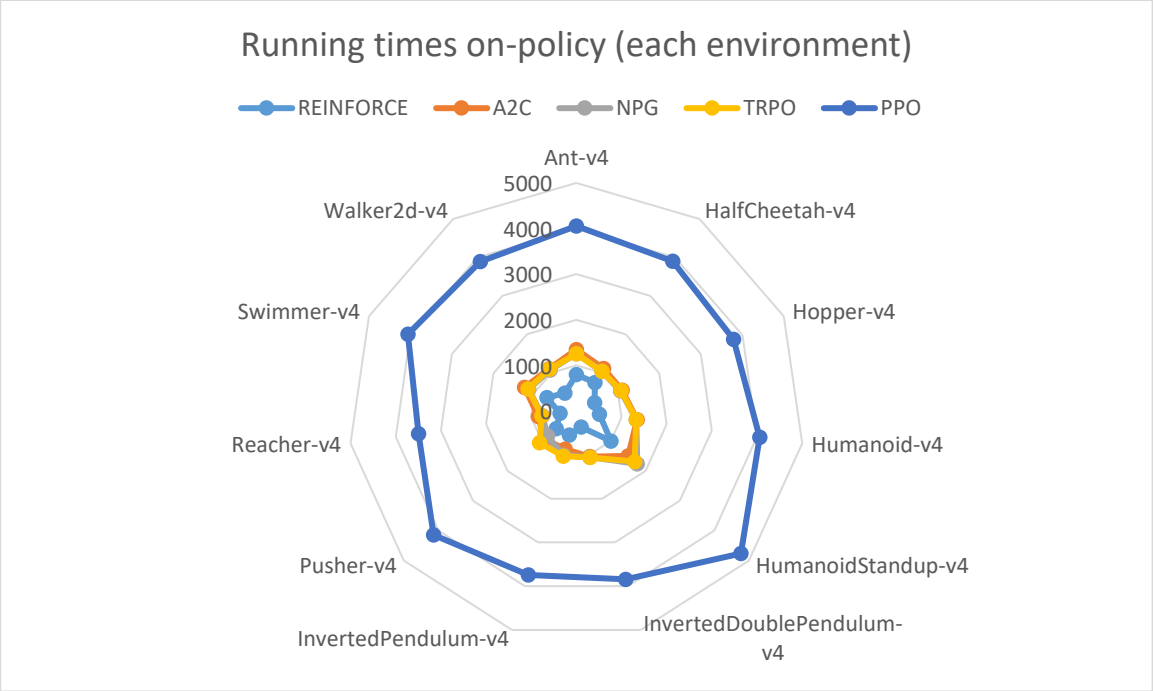


Figure 3-62 Running times of on-policy learning methods in each environment

In running time, we take into consideration both training time and testing time. Notice that length of an episode does not influence running time. In each epoch, we train for a fixed amount of timesteps, which are later saved in an experience replay buffer. If episodes are short, that means we will sample more episodes in an epoch.

On-policy learning methods take the same time to execute through all environments. This is expected, given that we train each on-policy learning method for the same number of timesteps (3M timesteps) in each environment. Furthermore, each environment runs on the same MuJoCo platform, hence they all have the same refresh frequency. Neural networks for the actor and critic are similar in all methods and require approximately the same amount of time to train.

In total over all environments, PPO took longest to train with 12:09:07, followed by TRPO with 3:29:22, A2C with 3:27:20, NPG with 3:25:11 and finally REINFORCE with 1:49:03.

Off-policy learning methods

Table 3-6 Running times for off-policy learning methods in MuJoCo experiments

Running time (h:mm:ss)	DDPG	TD3	SAC	REDQ
Ant-v4	2:31:51	2:54:48	4:53:48	3:57:05
HalfCheetah-v4	2:25:55	2:44:12	4:43:20	4:04:11
Hopper-v4	2:23:21	2:43:19	4:42:37	4:04:41
Humanoid-v4	3:04:00	0:21:39	5:39:44	4:09:10
HumanoidStandup-v4	3:54:52	4:21:57	6:44:36	5:24:34
InvertedDoublePendulum-v4	2:31:43	2:53:04	4:52:37	4:07:57
InvertedPendulum-v4	2:28:20	2:49:49	4:49:22	4:08:49
Pusher-v4	3:06:16	3:31:35	5:44:29	4:55:44
Reacher-v4	2:18:38	2:38:16	4:33:30	3:55:00
Swimmer-v4	2:27:37	2:48:07	4:48:09	3:52:32
Walker2d-v4	2:24:40	2:43:20	4:48:05	3:56:15
Average	**2:41:34**	**2:46:22**	**5:07:18**	**4:14:11**
Total	**1d 5:37:12**	**1d 6:30:05**	**2d 8:20:18**	**1d 22:35:57**

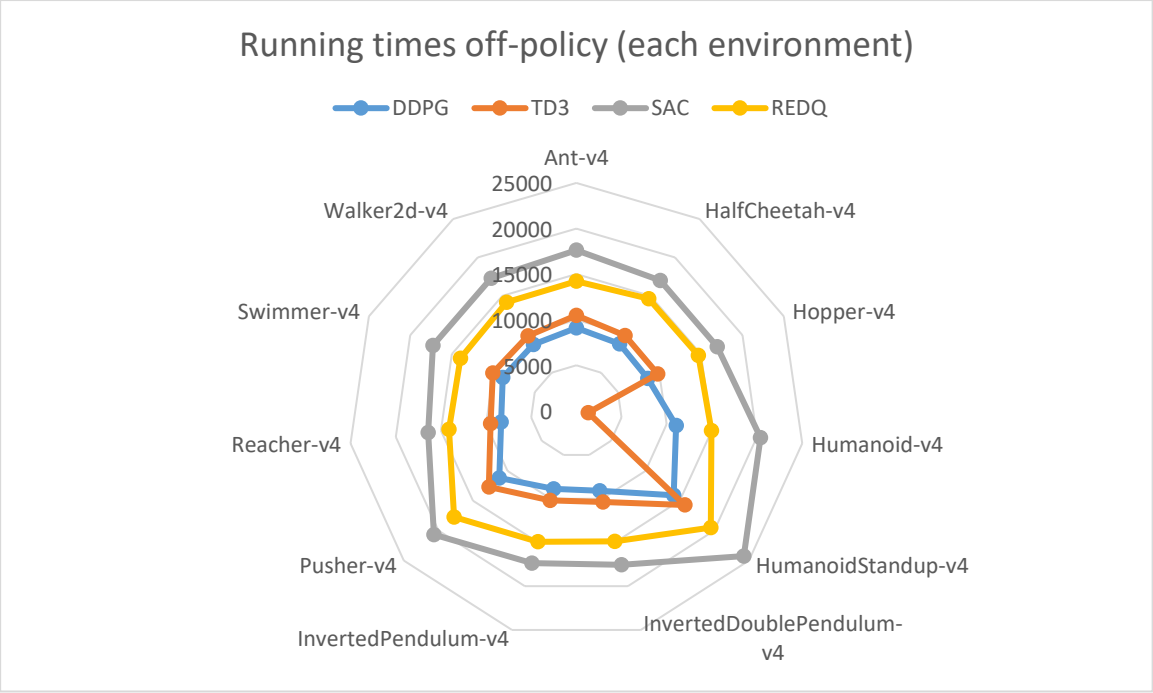


Figure 3-63 Running times of off-policy learning methods in each environment

In running time, we take into consideration both training time and testing time. Notice that length of an episode does not influence running time. In each epoch, we train for a fixed amount of timesteps, which are later saved in an experience replay buffer. If episodes are short, that means we will sample more episodes in an epoch.

Off-policy learning methods take the same time to execute through all environments. This is expected, given that we train each on-policy learning method for the same number of timesteps (1M timesteps) in each environment. Furthermore, each environment runs on the same MuJoCo platform, hence they all have the same refresh frequency. Neural networks for the actor and critic are similar in all methods and require approximately the same amount of time to train.

In total over all environments, SAC took longest to train with 2d 8:20:18, followed by REDQ with 1d 22:35:57, TD3 with 1d 6:30:05 and finally DDPG with 1d 5:37:12. As we can see, off-policy learning methods take considerably more time to execute than on-policy learning methods.

3.6.14 Speed of training

On-policy learning methods

Table 3-7 Speed of training for on-policy learning methods in MuJoCo experiments

Speed of training	REINFORCE	A2C	NPG	TRPO	PPO
Ant-v4	6353.34 step/s	2902.25 step/s	3248.03 step/s	3237.90 step/s	823.21 step/s
HalfCheetah-v4	7224.49 step/s	3629.41 step/s	4053.47 step/s	4066.65 step/s	849.23 step/s
Hopper-v4	7875.17 step/s	3141.04 step/s	3640.36 step/s	3625.83 step/s	863.48 step/s
Humanoid-v4	6482.98 step/s	2662.57 step/s	2668.36 step/s	2641.95 step/s	770.13 step/s
HumanoidStandup-v4	5736.29 step/s	2663.10 step/s	2178.34 step/s	2311.14 step/s	700.39 step/s
InvertedDoublePendulum-v4	9156.07 step/s	3442.99 step/s	3798.12 step/s	3744.96 step/s	846.51 step/s
InvertedPendulum-v4	10606.20 step/s	3534.57 step/s	4044.53 step/s	3900.47 step/s	854.99 step/s
Pusher-v4	5782.97 step/s	3308.72 step/s	3807.38 step/s	2980.63 step/s	748.11 step/s
Reacher-v4	8898.79 step/s	3626.81 step/s	4179.37 step/s	4099.40 step/s	883.28 step/s
Swimmer-v4	8092.04 step/s	3171.26 step/s	3603.20 step/s	3553.75 step/s	819.67 step/s
Walker2d-v4	8238.03 step/s	3219.25 step/s	3632.49 step/s	3623.19 step/s	844.42 step/s
Average	**7676.94 step/s**	**3209.27 step/s**	**3532.15 step/s**	**3435.08 step/s**	**818.49 step/s**

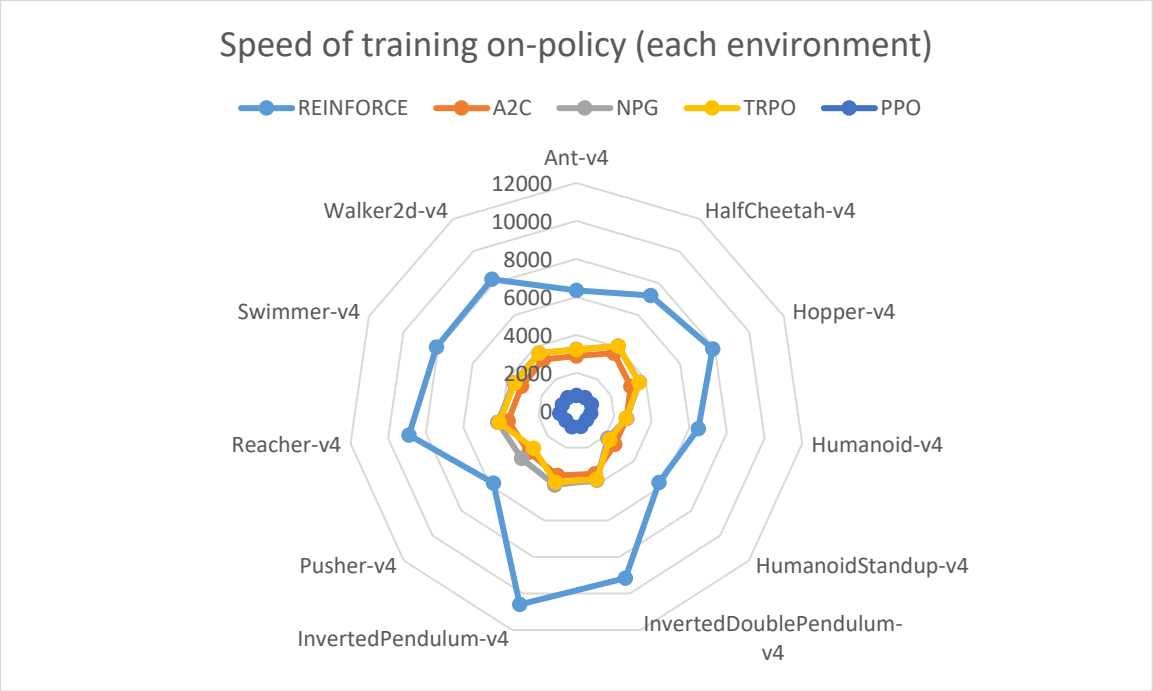


Figure 3-64 Speed of training of on-policy learning methods in each environment

Speed of training is similar, although not the same as running time. In running time, we take into consideration both training time and testing time. In speed of training, we are concerned only with training and not testing. Notice that length of an episode does not influence running time. In each epoch, we train for a fixed amount of timesteps, which are later saved in an experience replay buffer. If episodes are short, that means we will sample more episodes in an epoch.

On-policy learning methods have almost the same training speed through all environments. This is expected, given that we train each on-policy learning method for the same number of timesteps (3M timesteps) in each environment. Furthermore, each environment runs on the same MuJoCo platform, hence they all have the same refresh frequency. Neural networks for the actor and critic are similar in all methods and require approximately the same amount of time to train.

On average, the fastest on-policy learning method was REINFORCE with 7676 step/s, followed by NPG with 3532 step/s, TRPO with 3435 step/s, A2C with 3209 step/s and finally PPO with 818 step/s.

Off-policy learning methods

Table 3-8 Speed of training for off-policy learning methods in MuJoCo experiments

Speed of training	DDPG	TD3	SAC	REDQ
Ant-v4	114.08 step/s	99.67 step/s	58.97 step/s	73.66 step/s
HalfCheetah-v4	119.12 step/s	105.35 step/s	60.98 step/s	71.20 step/s
Hopper-v4	119.21 step/s	105.32 step/s	60.85 step/s	70.81 step/s
Humanoid-v4	90.76 step/s	79.34 step/s	50.33 step/s	69.03 step/s
HumanoidStandup-v4	73.92 step/s	66.05 step/s	42.54 step/s	53.47 step/s
InvertedDoublePendulum-v4	114.28 step/s	99.69 step/s	58.92 step/s	70.03 step/s
InvertedPendulum-v4	116.23 step/s	101.29 step/s	59.54 step/s	69.67 step/s
Pusher-v4	89.97 step/s	79.18 step/s	48.60 step/s	56.66 step/s
Reacher-v4	120.49 step/s	105.51 step/s	61.05 step/s	71.08 step/s
Swimmer-v4	118.20 step/s	103.25 step/s	60.04 step/s	74.93 step/s
Walker2d-v4	118.10 step/s	104.88 step/s	59.69 step/s	73.43 step/s
Average	**108.58 step/s**	**95.42 step/s**	**56.51 step/s**	**68.55 step/s**

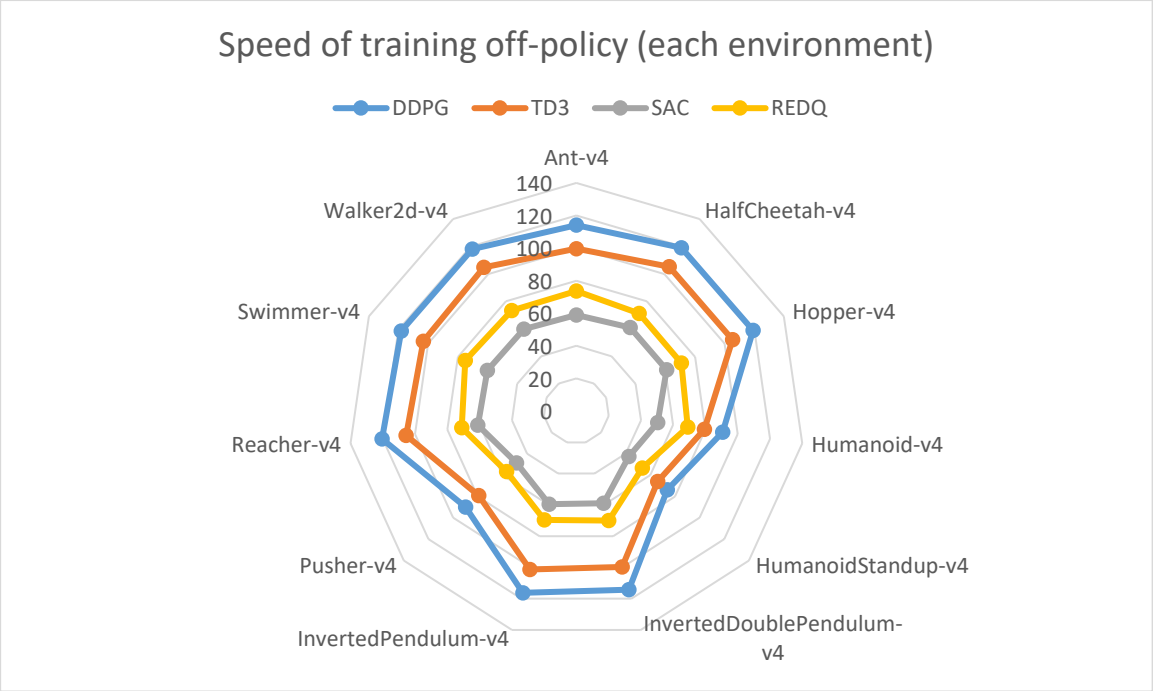


Figure 3-65 Speed of training of off-policy learning methods in each environment

Speed of training is similar, although not the same as running time. In running time, we take into consideration both training time and testing time. In speed of training, we are concerned only with training and not testing. Notice that length of an episode does not influence running time. In each epoch, we train for a fixed amount of timesteps, which are later saved in an experience replay buffer. If episodes are short, that means we will sample more episodes in an epoch.

Off-policy learning methods take the same time to execute through all environments. This is expected, given that we train each on-policy learning method for the same number of timesteps (1M timesteps) in each environment. Furthermore, each environment runs on the same MuJoCo platform, hence they all have the same refresh frequency. Neural networks for the actor and critic are similar in all methods and require approximately the same amount of time to train.

On average, the fastest off-policy learning method was DDPG with 108 step/s, followed by TD3 with 95 step/s, REDQ with 68 step/s and finally SAC with 56 step/s.

3.7 Summary

Off-policy learning methods significantly outperform on-policy learning methods in all environments. They require less timesteps to converge to an optimal policy, hence are the most sample efficient and achieve much higher scores. Each method compared to the other methods takes proportionally the same time in each environment. The reason for that is that all 11 environments we experimented with have the same refresh frequency, given that they run on the MuJoCo platform.

A drawback of on-policy learning methods is that they use a lot more computational power compared to off-policy learning methods. The reason is that they try to help the gradient descent towards the steepest direction by calculating a Hessian matrix or a similar, partial form of a Hessian. Such calculations need a lot of computational resources.

A drawback of off-policy learning methods is that they take longer to train compared to on-policy learning methods. Most importantly, this has to do with the implementation details. Another reason is that in off-policy learning methods, in one epoch we sample less timesteps of experience before a gradient update. This means that during training, we do gradient updates more frequently than in on-policy learning methods, which slows down training.

Best scores for on-policy learning methods were achieved by TRPO, then NPG, then PPO, then A2C and finally REINFORCE. Actually, TRPO, NPG and PPO are high performing and deliver very similar best scores. The lowest standard deviations were achieved by PPO, then TRPO, then NPG, then A2C and finally REINFORCE. The lower the standard deviation, the less variance and more accurate the methods are. Actually, PPO and TRPO deliver similar standard deviations, with NPG a close third.

Best scores for off-policy learning methods were delivered by REDQ, then SAC, then TD3 and finally DDPG. Actually, REDQ and SAC are both high performing and deliver very similar best scores. The lowest standard deviations were achieved by REDQ, then SAC, then TD3 and finally DDPG. The lower the standard deviation, the less variance and more accurate the methods are. Actually, REDQ and SAC deliver similar standard deviations, with TD3 a close third.

Let’s discuss total running time. For on-policy learning methods, in total over all environments, PPO took longest to train with 12:09:07, followed by TRPO with 3:29:22, A2C with 3:27:20, NPG with 3:25:11 and finally REINFORCE with 1:49:03. For off-policy learning methods, in total over all environments, SAC took longest to train with 2d 8:20:18, followed by REDQ with 1d 22:35:57, TD3 with 1d 6:30:05 and finally DDPG with 1d 5:37:12. As we can see, off-policy learning methods take considerably more time to execute than on-policy learning methods.

Let’s discuss average speed of training. For on-policy learning methods, on average over all environments, the fastest on-policy learning method was REINFORCE with 7676 step/s, followed by NPG with 3532 step/s, TRPO with 3435 step/s, A2C with 3209 step/s and finally PPO with 818 step/s. For off-policy learning methods, on average over all environments, the fastest was DDPG with 108 step/s, followed by TD3 with 95 step/s, REDQ with 68 step/s and finally SAC with 56 step/s.

Chapter 4

Off-policy learning in robotics simulations

4.1 Introduction

Robotics simulations are a very important tool for training robotic systems on particular tasks. Robotic simulations require a lot of training samples. Off-policy learning methods are the most sample efficient methods in deep RL, hence are one of the most capable for solving robotics simulations.

A weakness of on-policy learning methods is calculating expectations only under the current, known trajectory of interest. Each time we have a new policy, we need to use new samples. With increasing task complexity, this becomes expensive. The number of gradient steps and samples per step needed to learn an effective policy increases [16]. Moreover, training a neural network changes θ only by a little, but the computational overhead for that change is large. These arguments emphasize the need for sample efficient off-policy learning methods.

We use off-policy learning methods together with Hindsight Experience Replay buffer. HER can learn with extremely sparse rewards. It also performs better with sparse rewards than with dense rewards. Robotics tasks usually have sparse reward structures, so using HER is mandatory.

In this chapter, I present the results of training off-policy learning methods with HER in Fetch mobile manipulator, a 7-DoF robotic arm with a two-fingered parallel gripper attached to it.

4.2 Software components

This experiment requires all software components from the previous chapter:

- MuJoCo
- Tianshou
- Gymnasium
- Jupyter Lab
- Tensorboard

The following components are also necessary.

Gymnasium-Robotics

Gymnasium-Robotics is a collection of robotics simulation environments based on Gymnasium. Originally, it was called OpenAI Gym [37]. The robotic manipulation tasks are more difficult than continuous control problems available in Gymnasium. It provides simulations for real-life robots, each with a different degree of freedom (DoF). The physics engine for Gymnasium-Robotics is MuJoCo.

4.3 Fetch mobile manipulator

Fetch mobile manipulator [38] is a 7-DoF robotic arm with a two-fingered parallel gripper attached to it. The robot is controlled by small displacements of the gripper in Cartesian coordinates and the inverse kinematics are computed internally by the MuJoCo framework.

The control frequency of the robot is $f = 25\ Hz$. This is achieved by applying the same action in 20 subsequent simulator step (with a time step of dt = 0.002 s) before returning the control to the robot. The tasks in Fetch mobile manipulator are continuing, which means that the robot has to maintain the puck in the target position for an indefinite period of time.

Fetch can be trained to execute the following tasks.

Reach

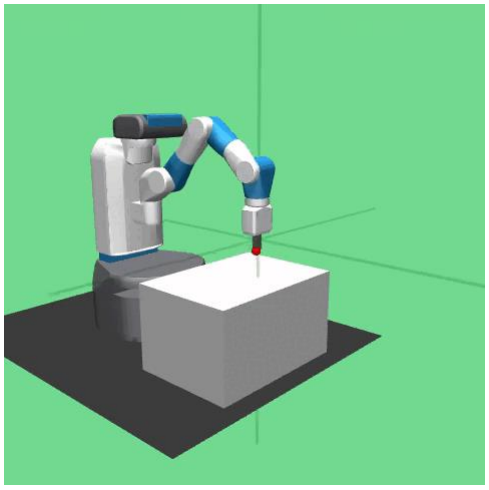


Figure 4-1 Reach task in Fetch mobile manipulator

Task: Move the end effector of the manipulator to a randomly selected position in the workspace.

Action Space: Action space is a vector of size 4. It represents the Cartesian displacement dx, dy, and dz of the end effector.

Push

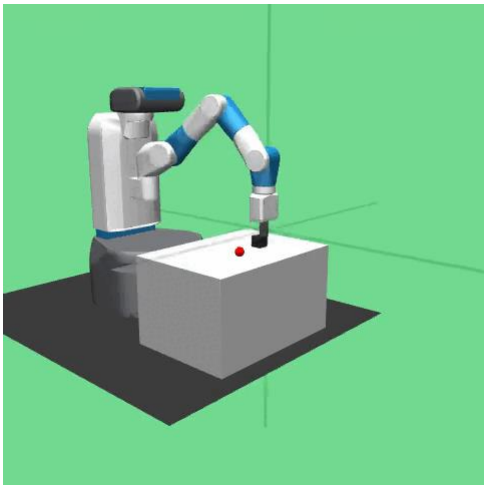


Figure 4-2 Push task in Fetch mobile manipulator

Task: Move a block with the help of the manipulator to a randomly selected target position in the workspace by pushing with its gripper. The gripper is locked in a closed configuration in order to perform the push task.

Action Space: Action space is a vector of size 4. It represents the Cartesian displacement dx, dy, and dz of the end effector.

Pick and place

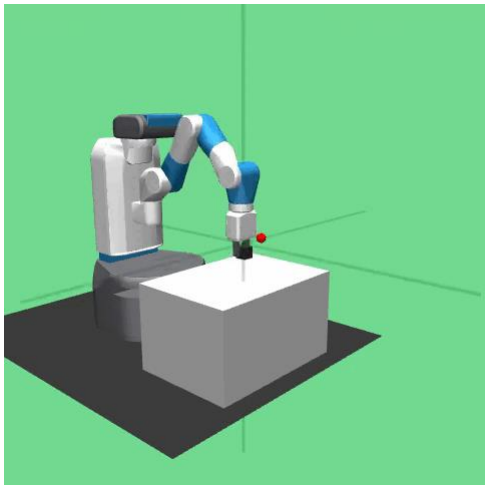


Figure 4-3 Pick and Place task in Fetch mobile manipulator

Task: Move a block with the help of the manipulator to a randomly selected target position in the workspace in mid-air by picking and placing with its gripper. The gripper can be opened or closed in order to perform the grasping operation of pick and place.

Action Space: Action space is a vector of size 4. It represents the Cartesian displacement dx, dy, and dz of the end effector.

Slide

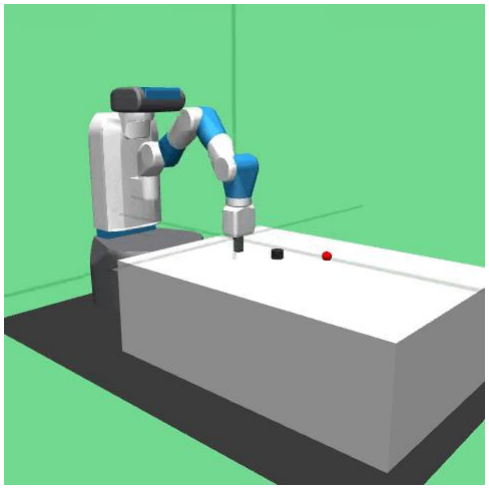


Figure 4-4 Slide task in Fetch mobile manipulator

Task: Hit a puck with the help of the manipulator in order to slide it until it reaches a randomly selected target position in a long and slippery workspace. The workspace has a low friction coefficient, which makes it slipper for the puck to slide and reach the target position. The target position is outside of the robot's workspace. The gripper is locked in a closed configuration. The puck doesn't need to be grasped.

Action space: Action space is a vector of size 4. It represents the Cartesian displacement dx, dy, and dz of the end effector.

4.4 Hardware

To speed up training, we trained on a machine with GPU acceleration. GPU acceleration substantially speeds up training, because of hardware-level parallelization and faster I/O rates.

AMD Ryzen Threadripper 3970X

- Base speed: 3.70 GHz
- 32 Cores and 64 Logical processors
- L1 cache: 2.0 MB, L2 cache: 16.0 MB, L3 cache: 128 MB

NVIDIA RTX A6000

- Shared GPU memory: 64 GB
- NVIDIA Ampere Architecture Based CUDA Cores, Second-Generation RT Cores, Third-Generation Tensor Cores
- 38.7 TFLOPs of FP32 performance

4.5 Methodology

In this chapter, the methodology follows the general principles from [39]. We solve tasks Reach, Push and Pick and Place with the help of off-policy learning methods. We were not able to solve task Slide, which is also the most complicated task. Each episode for all tasks has a length of 50 timesteps. Each timestep the agent takes as a feedback the value -1 from the environment, if it has not reached the asked destination. That being said, it is clear that the final reward for an episode is within $[-50,0]$. We solve Reach with the help of off-policy learning methods: DDPG, SAC, TD-3 and REDQ.

For task Reach, we train for 10 epochs with 5000 steps per epoch, for a total of 50000 timesteps. Hindsight Experience Replay buffer size is kept at 100000. The discount factor gamma γ of future rewards is kept at 0.98. Both actor and critic networks are two-layer neural networks with

sizes [256,256]. For SAC and TD-3, two identic critic networks are used. We train the actor and critic networks with gradient descent using Adam optimizer. The learning rate α is kept at 10^{-3} for the actor and $3 * 10^{-3}$ for the critic.

For task Push, we train for 300 epochs with 5000 steps per epoch, for a total of 1.5 million timesteps. Hindsight Experience Replay buffer size is kept at 100000. The discount factor gamma γ of future rewards is kept at 0.98. Both actor and critic networks are three-layer neural networks with sizes [256,256,256]. For SAC and TD-3, two identic critic networks are used. We train the actor and critic networks with gradient descent using Adam optimizer. The learning rate α is kept at 10^{-4} for the actor and $3 * 10^{-4}$ for the critic.

For task Pick and Place, we train for 600 epochs with 5000 steps per epoch, for a total of 3 million timesteps. Hindsight Experience Replay buffer size is kept at 100000. The discount factor gamma γ of future rewards is kept at 0.98. Both actor and critic networks are three-layer neural networks with sizes [256,256,256]. For SAC and TD-3, two identic critic networks are used. We train the actor and critic networks with gradient descent using Adam optimizer. The learning rate α is kept at 10^{-5} for the actor and $3 * 10^{-5}$ for the critic.

For task Slide, we train for 900 epochs with 5000 steps per epoch, for a total of 4.5 million timesteps. Hindsight Experience Replay buffer size is kept at 50, equal to the length of a training episode. The discount factor gamma γ of future rewards is kept at 0.98. Both actor and critic networks are three-layer neural networks with sizes [256,256,256]. For SAC and TD-3, two identic critic networks are used. We train the actor and critic networks with gradient descent using Adam optimizer. The learning rate α is kept at 10^{-6} for the actor and $3 * 10^{-6}$ for the critic.

Loss curves for tasks Reach, Push and Pick and Place are shown in the appendix. Notice that we were not able to solve task Slide with any off-policy learning method, hence we restrain ourselves from showing results.

4.5.1 Hindsight Experience Replay Buffer

Hindsight Experience Replay buffer (HER) [40] is a form of implicit curriculum. The goals used for replay naturally shift from ones which are simple to achieve even by a random agent, to more difficult ones. First, HER experiences some episode $s_0, s_1, s_2 \dots s_n$. Next, it stores in the replay buffer every transition $s_t \neq s_{t+1}$, not only with the original goal used for this episode, but also with a subset of other goals. Finally, we replay each trajectory with an arbitrary goal. HER only works with an off-policy RL algorithm.

The set of additional goals used for replay has to be chosen. We could choose no additional goals, hence replay each trajectory with the goal $m(s_T)$, which is achieved in the final state of the episode. If we were to choose additional goals, we experimentally compare different types and quantities of goals for replay. In all cases, we also replay each trajectory with the original goal pursued in the episode.

HER does not require having any control over the distribution of initial states. Furthermore, HER can learn with extremely sparse rewards. It also performs better with sparse rewards than with dense rewards.

4.6 Experimental results

4.6.1 FetchReach-v3

Off-policy learning methods

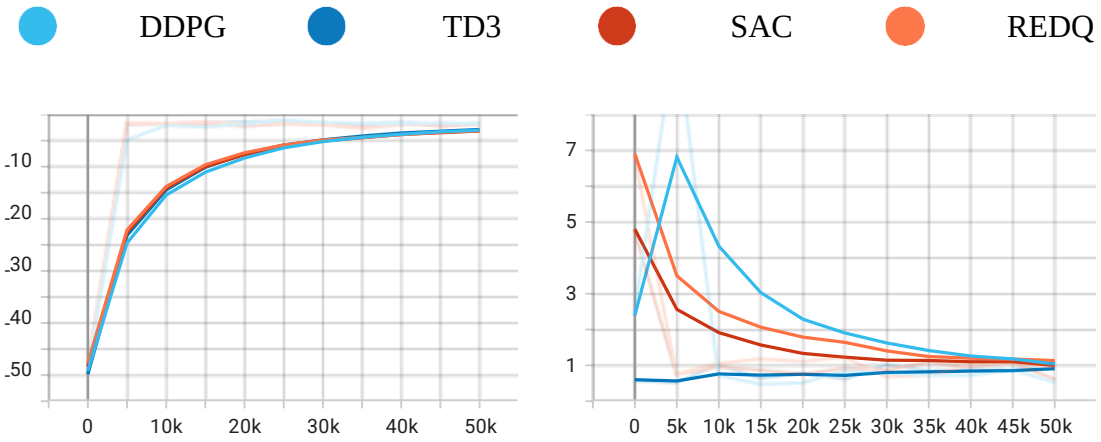


Figure 4-5 Mean rewards (testing) during timesteps

Figure 4-6 Std. deviation of rewards (testing) during timesteps

4.6.2 FetchPush-v2

Off-policy learning methods

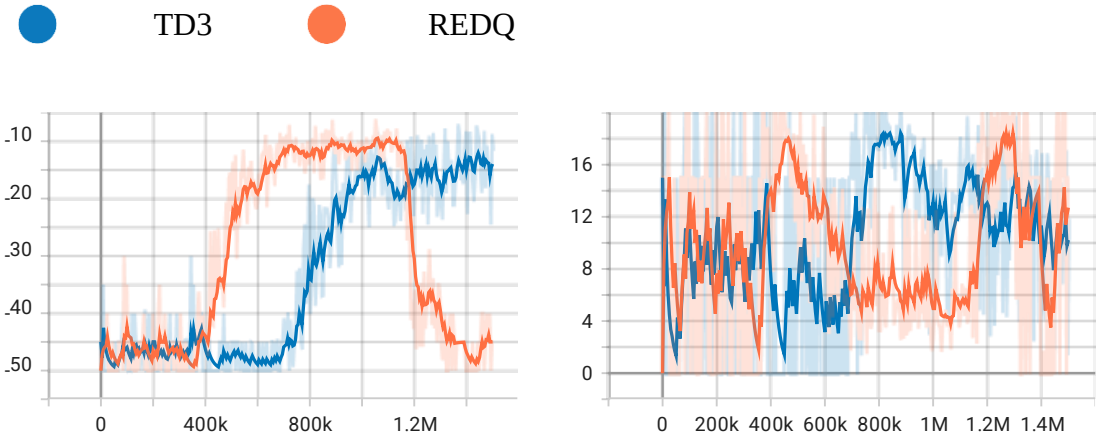


Figure 4-7 Mean rewards (testing) during timesteps

Figure 4-8 Std. deviation of rewards (testing) during timesteps

4.6.3 FetchPickAndPlace-v2

Off-policy learning methods

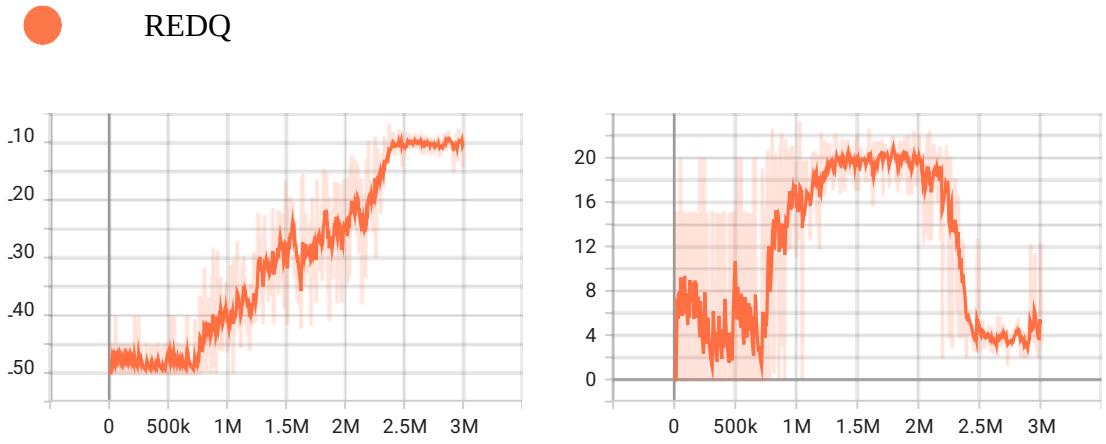


Figure 4-9 Mean rewards (testing) during timesteps

Figure 4-10 Std. deviation of rewards (testing) during timesteps

4.6.4 Best scores

Off-policy learning methods

Table 4-1 Best scores achieved by off-policy learning methods in Fetch Mobile Manipulator

	DDPG	TD3	SAC	REDQ
FetchReach-v3	-1.20 ± 0.87	-1.30 ± 0.64	-1.80 ± 0.98	-1.50 ± 1.20
FetchPush-v2	Diverged	-7.00 ± 4.36	Diverged	-6.20 ± 3.66
FetchPickAndPlace-v2	Diverged	Diverged	Diverged	-6.80 ± 5.38
FetchSlide-v2	Diverged	Diverged	Diverged	Diverged

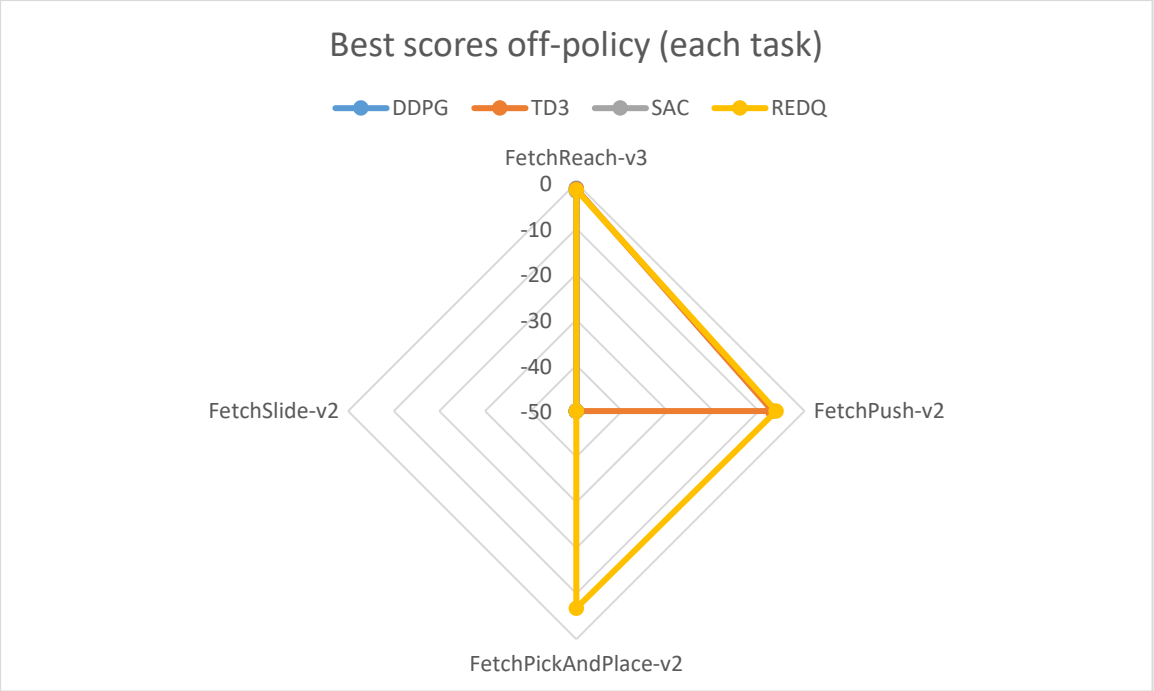


Figure 4-11 Best scores of off-policy learning methods in each task in Fetch mobile manipulator

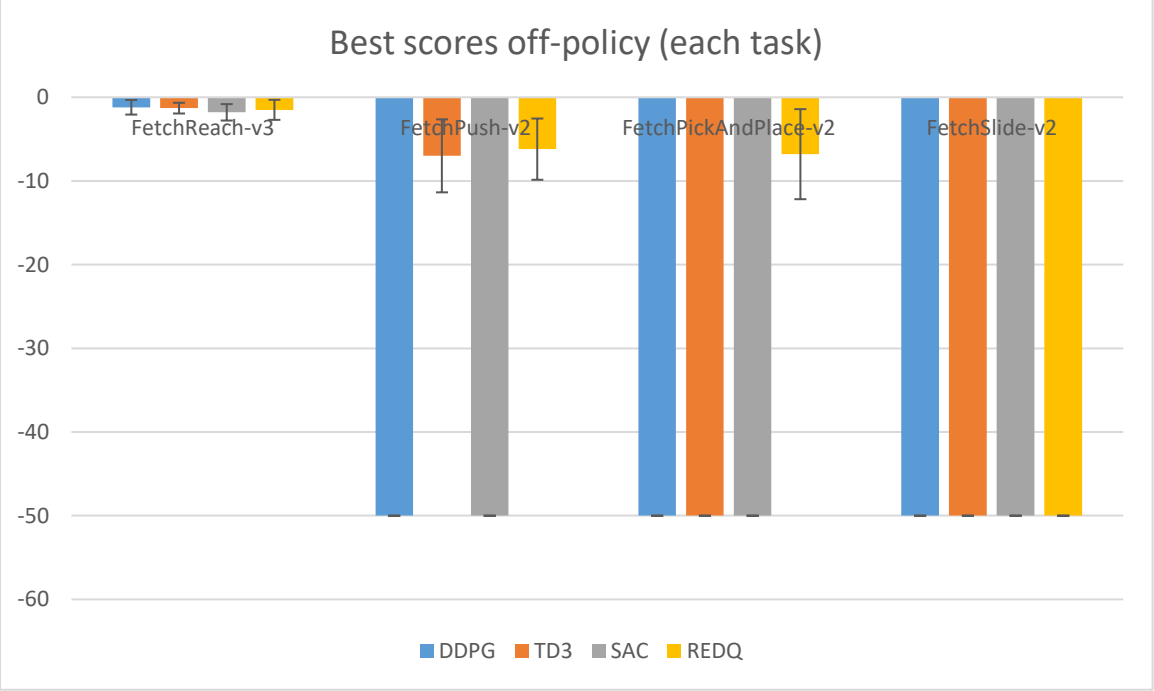


Figure 4-12 Best scores of off-policy learning methods in each task in Fetch mobile manipulator, with standard deviations as confidence bounds (values close to 0 have learned the task, values close to -50 have not learned the task)

4.7 Summary

Off-policy learning methods are very sample efficient, a crucial criteria for solving robotics tasks in physics-based simulations. Moreover, Hindsight Experience Replay buffer is a form of implicit curriculum, where the goals used for replay naturally shift from ones which are simple to more difficult ones. HER can learn with extremely sparse rewards. It also performs better with sparse rewards than with dense rewards. It is to be noted that the rewards from Fetch mobile manipulator are sparse rewards, hence using HER is mandatory. Furthermore, HER works only with off-policy learning methods, not with on-policy learning methods, which emphasizes the need for off-policy learning methods for solving robotics tasks.

Let's discuss best scores and their standard deviations. The lower the standard deviation of scores delivered by a method, the more accurate those methods are, the lower the variance they have. Reach task is the easiest task from Fetch mobile manipulator. We were able to solve Reach with all off-policy learning methods. DDPG achieved a score of -1.20 ± 0.87, TD3 a score of -1.30 ± 0.64, SAC a score of -1.80 ± 0.98 and REDQ a score of -1.50 ± 1.20. Push task is more complicated than Reach. We were able to solve Push task with TD3 and REDQ. TD3 achieved a score of -7.00 ± 4.36 and REDQ achieved a score of -6.20 ± 3.66. Pick and Place is more complicated than Push. We were able to solve Pick and Place with REDQ. REDQ achieved a score of -6.80 ± 5.38. Finally, Slide is the most complicated task from Fetch mobile manipulator. Being the most complicated task, we were not able to solve it with any off-policy learning method.

In more complicated tasks, best scores decrease and standard deviations increase. Essentially, accuracy drops with more complicated tasks, although slightly. In Fetch mobile manipulator, accuracy drops slightly as we increase the complexity of tasks.

Conclusions

In this thesis, we show that in physics-based simulation, robots or other hypothetical agents can be controlled to perform amazing tasks. On the downside, real world applications are often unkind to simulation-derived control policies [1]. This is also known as the simulation-to-reality gap. Nevertheless, up-to-date there are no better alternatives to physics-based simulations. We could say that physics-based simulations are “doomed to succeed”, because trial-and-error approaches in the real world can be expensive, dangerous or sometimes impossible. A prime example is testing a Mars rover. Striving to steadily improve the simulation-to-reality transfer rate is a worthwhile endeavor. In order to derive better control policies in simulation that can be transferred in real world applications, we could increase the fidelity of the simulation.

To replace or augment physics-based simulations, statistical learning techniques and data driven simulations have been proposed. Through statistical learning, we could construct oracles that predict the next system state given its current one. This effectively bypasses simulation-specific hurdles such as model generation and calibration. Through systematic dimensional reduction, data-driven simulation could be used to methodically reduce model complexity. However, such approaches for highly nonlinear and non-smooth systems are in their infancy.

In our experiments, we test the performance of 9 on-policy and off-policy Deep Reinforcement Learning algorithms in various physics-based simulations with MuJoCo. Off-policy learning methods are significantly more accurate, sample efficient and have less variance than on-policy learning methods. On the downside, off-policy learning methods take longer to execute than on-policy learning methods.

Best scores for on-policy learning methods were achieved by TRPO, then NPG, then PPO, then A2C and finally REINFORCE. Actually, TRPO, NPG and PPO are high performing and deliver very similar best scores. The lowest standard deviations were achieved by PPO, then TRPO, then NPG, then A2C and finally REINFORCE. The lower the standard deviation, the less variance and more accurate the methods are. Actually, PPO and TRPO deliver similar standard deviations, with NPG a close third.

Best scores for off-policy learning methods were delivered by REDQ, then SAC, then TD3 and finally DDPG. Actually, REDQ and SAC are both high performing and deliver very similar best scores. The lowest standard deviations were achieved by REDQ, then SAC, then TD3 and finally DDPG. The lower the standard deviation, the less variance and more accurate the methods are. Actually, REDQ and SAC deliver similar standard deviations, with TD3 a close third.

Let's discuss average speed of training. We trained in a machine with CPU @3.70 GHz and GPU acceleration of 38.7 TFLOPs. The actual value for the speed of training is specific to the hardware we are using. Nevertheless, we expect the rankings and ratios of the speed of training between different methods to remain the same under a different set of hardware. For on-policy learning methods, on average over all environments, the fastest on-policy learning method was REINFORCE with 7676 step/s, followed by NPG with 3532 step/s, TRPO with 3435 step/s, A2C with 3209 step/s and finally PPO with 818 step/s. For off-policy learning methods, on average over all environments, the fastest was DDPG with 108 step/s, followed by TD3 with 95 step/s, REDQ with 68 step/s and finally SAC with 56 step/s.

Finally, in chapter 4 we test the performance of off-policy learning methods in robotics simulations with Fetch mobile manipulator [38], a 7-DoF robotic arm with a two-fingered parallel gripper attached to it. Fetch exists and is used in real-world scenarios. Fetch can be trained to execute four different tasks: Reach, Push, Pick and Place and Slide. We make use of Hindsight Experience Replay buffer (HER) [40], a form of implicit curriculum, where the goals used for replay naturally shift from ones which are simple to achieve even by a random agent, to more difficult ones. Reach task was solved by DDPG, TD3, SAC and REDQ. Push task was solved by TD3 and REDQ. Pick and Place was solved only by REDQ. Slide is the most complicated task, we were not able to solve it with any off-policy learning method.

Bibliography

[1] K. Liu and D. Negrut, “The Role of Physics-Based Simulators in Robotics,” in *Annual Review of Control, Robotics, and Autonomous Systems*, 2021.

[2] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Pearson, 2020.

[3] M. van Otterlo and M. Wiering, “Reinforcement Learning and Markov Decision Processes,” in *Reinforcement learning: State-of-the-art*, Springer, 2012, pp. 3-42.

[4] Y. Mansour and S. Singh, “On the Complexity of Policy Iteration,” 23 January 2013.

[5] J. A. Boyan and A. W. Moore, “Generalization in reinforcement learning: Safely approximating the value function,” 1998.

[6] N. Metropolis and S. Ulam, “The monte carlo method,” 1949.

[7] G. Tesauro, “Temporal Difference Learning and TD-Gammon,” *ACM*, 1995.

[8] G. Rummery and M. Niranjan, “On-line Q-Learning using connectionist systems”.

[9] C. J. Watkins and P. Dayan, “Q-Learning,” *Kluwer Academic Publishers*, 1992.

[10] H. Hasselt, “Double Q-learning,” in *Advances in Neural Information Processing Systems 23*, 2010.

[11] M. Roderick, J. MacGlashan and S. Tellex, “Implementing the Deep Q-Network,” 20 November 2017.

[12] H. v. Hasselt, A. Guez and D. Silver, “Deep Reinforcement Learning with Double Q-Learning,” *DeepMind*, 2 March 2016.

[13] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra and M. Riedmiller, “Deterministic Policy Gradient Algorithms,” in *Proceedings of the 31st International Conference on Machine Learning*, 2014.

[14] R. J. Williams, “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning,” in *The Springer International Series in Engineering and Computer Science*, 1992.

[15] S. M. Kakade, “A Natural Policy Gradient,” in *Advances in Neural Information Processing Systems 14*, 2001.

[16] T. Haarnoja, A. Zhou, P. Abbeel and S. Levine, “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor,” in *Proceedings of the 35th International Conference on Machine Learning*.

[17] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons and J. Kautz, “Reinforcement Learning through Asynchronous Advantage Actor-Critic on a GPU,” 2016.

[18] J. Schulman, S. Levine, P. Abbeel, M. Jordan and P. Moritz, “Trust Region Policy Optimization,” in *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, “Proximal Policy Optimization Algorithms,” July 2017.

[20] S. Mayer, T. Classen and C. Endisch, “Modular production control using deep reinforcement learning: proximal policy optimization,” in *Journal of Intelligent Manufacturing*, May 2021.

[21] P. T. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra, “Continuous control with deep reinforcement learning,” in *International Conference on Learning Representations*, 2016.

[22] A. Kumar, N. Paul and S. N. Omkar, “Bipedal Walking Robot using Deep Deterministic Policy Gradient,” in *IEEE Symposium Series on Computational Intelligence(SSCI)*, 2018.

[23] S. Dankwa and W. Zheng, “Twin-Delayed DDPG: A Deep Reinforcement Learning Technique to Model a Continuous Movement of an Intelligent Robot Agent,” in *Proceedings of the 3rd International Conference on Vision, Image and Signal Processing*, August 2019.

[24] X. Chen, C. Wang, Z. Zhou and K. Ross, “Randomized Ensembled Double Q-Learning: Learning Fast Without a Model,” in *International Conference on Learning Representations*, 2021.

[25] Y. Li, Reinforcement learning applications, arXiv, 2019.

[26] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato and P. Wang, “Applications of Deep Reinforcement Learning in Communications and Networking: A Survey,” in *IEEE Communications Surveys & Tutorials*, 2019.

[27] S. Sicular, J. Hare and K. Brant, “Hype Cycle for Artificial Intelligence,” *Gartner Inc*, 25 July 2019.

[28] E. Todorov, T. Erez and Y. Tassa, “MuJoCo: A physics engine for model-based control,” in *International Workshop on Intelligent Robots and Systems*, 2012.

[29] J. Weng, H. Chen, D. Yan, K. You, A. Duburcq, M. Zhang, Y. Su, H. Su and J. Zhu, “Tianshou: A highly modularized deep reinforcement learning library,” in *Journal of Machine Learning Research 23*, 2021.

[30] B. M. Randles, I. V. Pasquetto, M. S. Golshan and C. L. Borgman, “Using the Jupyter Notebook as a Tool for Open Science: An Empirical Study,” 16 April 2018.

[31] J. Schulman, P. Moritz, S. Levine, M. Jordan and P. Abbeel, “High-Dimensional Continuous Control Using Generalized Advantage Estimation,” June 2015.

[32] P. Wawrzyński, “A Cat-Like Robot Real-Time Learning to Run,” in *Adaptive and Natural Computing Algorithms,* 2009.

[33] T. Erez, Y. Tassa and E. Todorov, “Infinite-horizon model predictive control for periodic tasks with contacts,” in *Robotics: Science and systems VII,* 2012.

[34] Y. Tassa, T. Erez and E. Todorov, “Synthesis and stabilization of complex behaviors through online trajectory optimization,” in *International Workshop on Intelligent Robots and Systems (IROS)* , 2012.

[35] A. G. Barto, R. S. Sutton and C. W. Anderson, “Neuronlike adaptive elements that can solve difficult learning control problems,” in *Transactions on Systems, Man, and Cybernetics,* 1983.

[36] R. Coulom, “Reinforcement Learning Using Neural Networks, with Applications to Motor Control,” 2002.

[37] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba, “Openai gym,” 2016.

[38] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew and B. Baker, “Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research,” February 2018.

[39] A. Rzayev and V. T. Aghaei, “Off-Policy Deep Reinforcement Learning Algorithms for Handling Various Robotic Manipulator Tasks,” *arXiv.*

[40] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel and W. Zaremba, “Hindsight Experience Replay,” in *Advances in Neural Information Processing Systems,* 2017.

Appendix

Chapter 3: Losses from experimental results

Ant-v4

Losses of off-policy learning methods

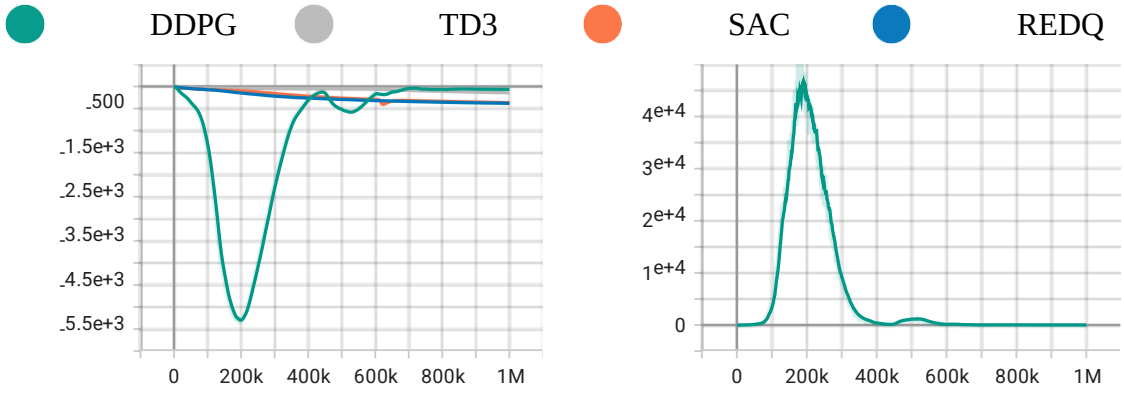


Figure 7-1 Actor loss during timesteps (training)

Figure 7-2 Critic loss during timesteps (training)

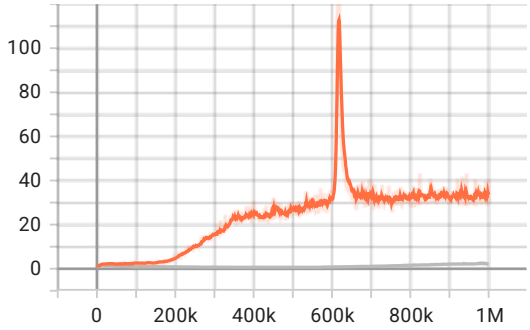


Figure 7-3 Critic 1 loss during timesteps (training)

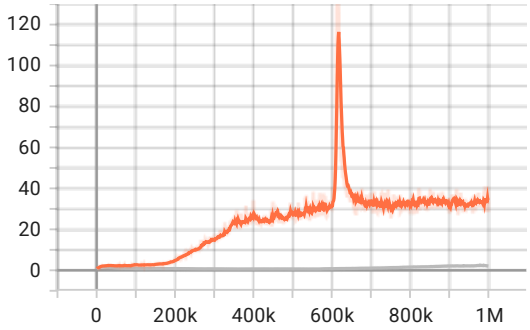


Figure 7-4 Critic 2 loss during timesteps (training)

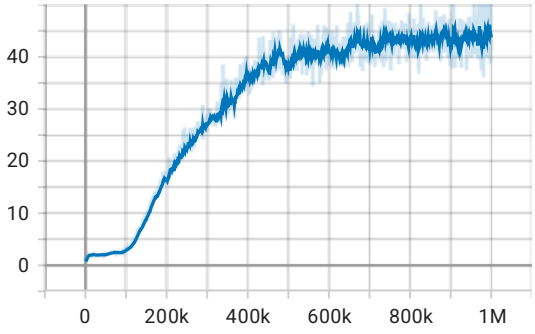


Figure 7-5 REDQ critics ensemble loss during timesteps (training)

HalfCheetah-v4

Losses of off-policy learning methods

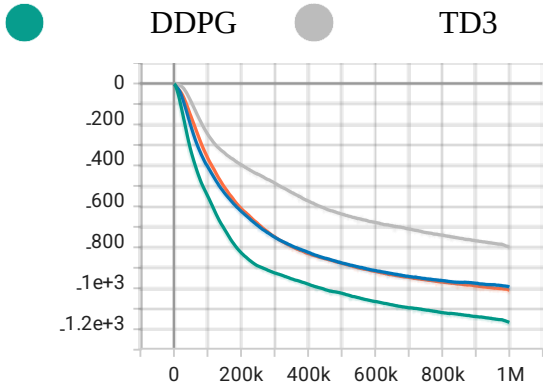


Figure 7-6 Actor loss during timesteps (training)

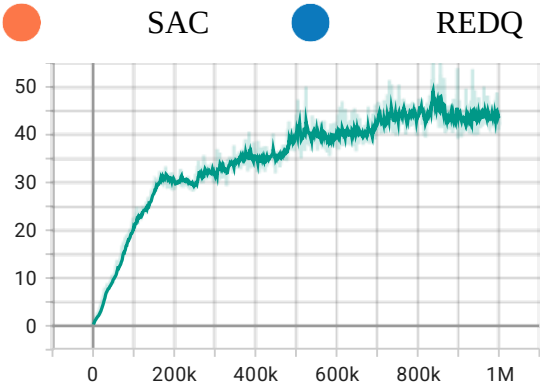


Figure 7-7 Critic loss during timesteps (training)

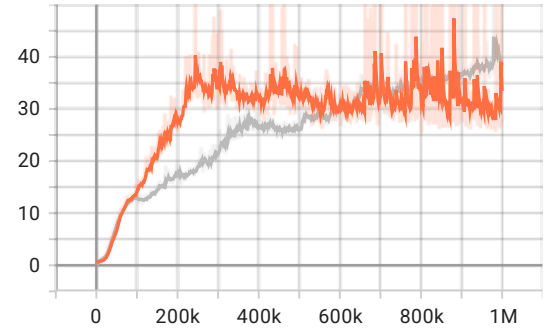


Figure 7-8 Critic 1 loss during timesteps (training)

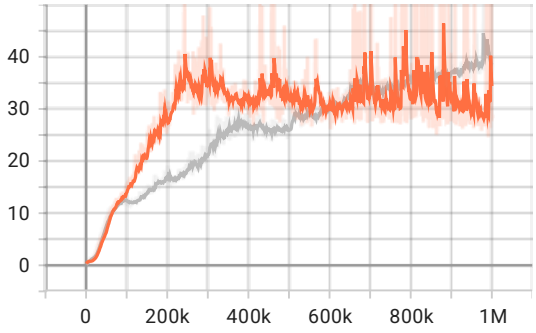


Figure 7-9 Critic 2 loss during timesteps (training)

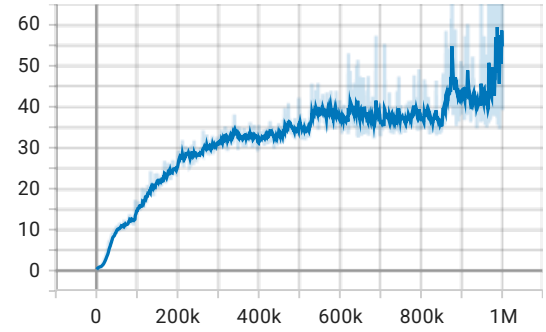


Figure 7-10 REDQ critics ensemble loss during timesteps (training)

Hopper-v4

Losses of off-policy learning methods

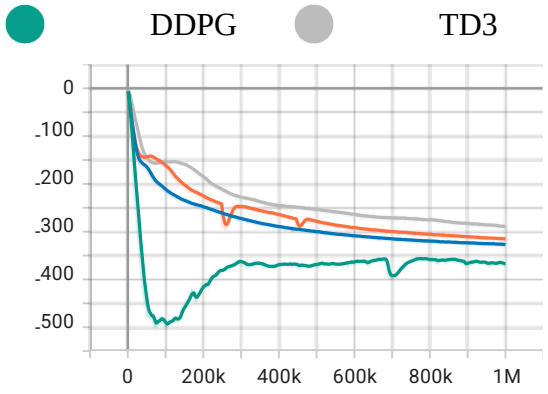


Figure 7-11 Actor loss during timesteps (training)

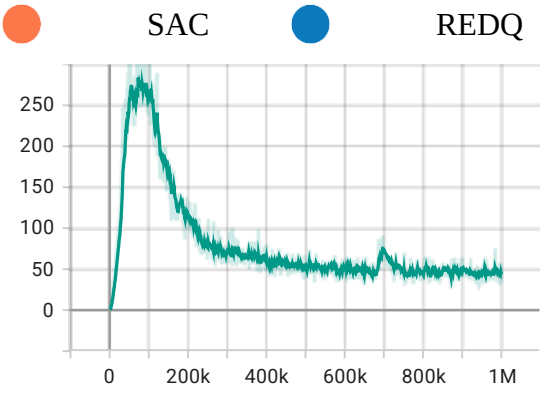


Figure 7-12 Critic loss during timesteps (training)

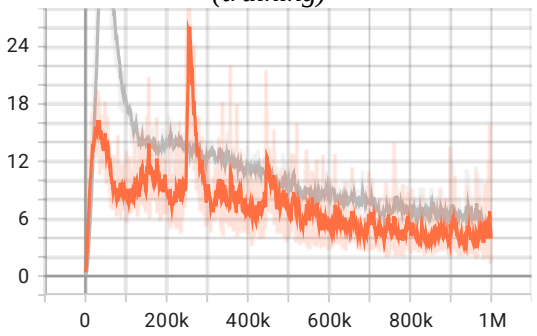


Figure 7-13 Critic 1 loss during timesteps (training)

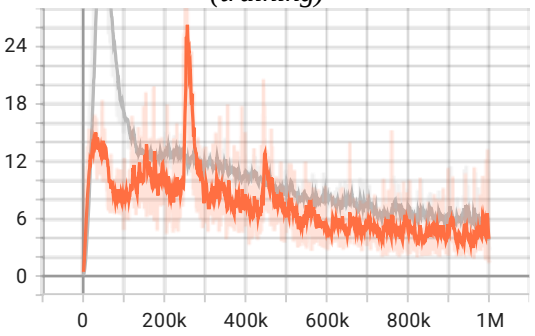


Figure 7-14 Critic 2 loss during timesteps (training)

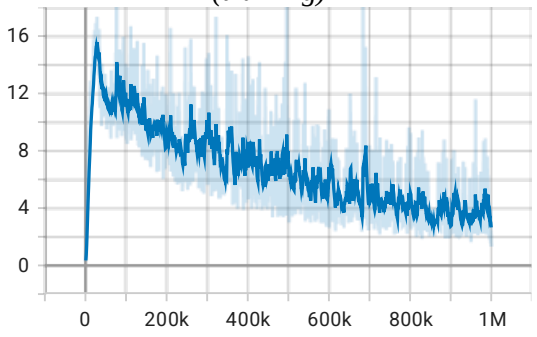


Figure 7-15 REDQ critics ensemble loss during timesteps (training)

HumanoidStandup-v4

Losses of off-policy learning methods

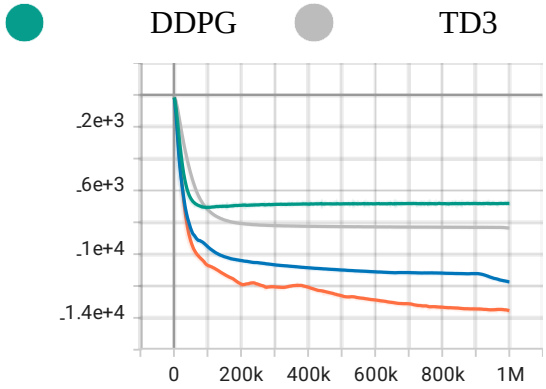


Figure 7-16 Actor loss during timesteps (training)

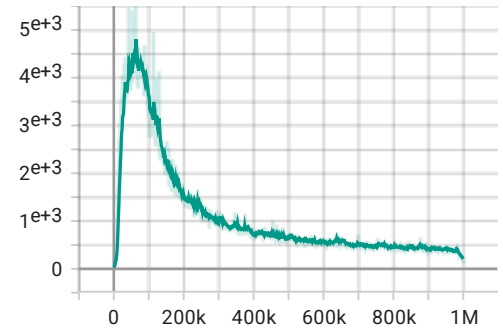


Figure 7-17 Critic loss during timesteps (training)

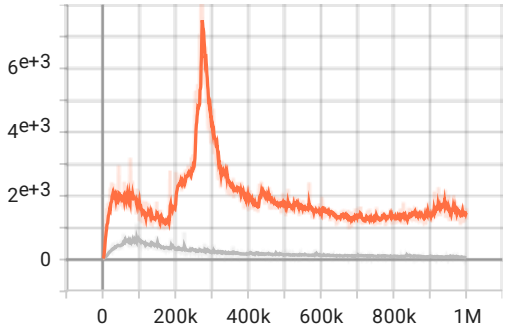


Figure 7-18 Critic 1 loss during timesteps (training)

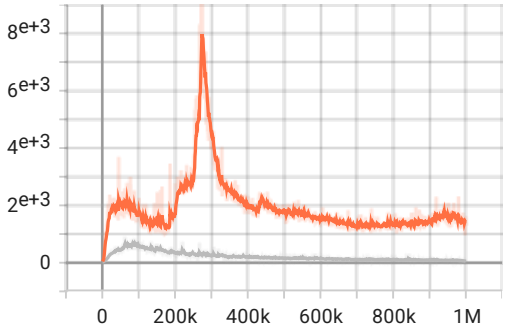


Figure 7-19 Critic 2 loss during timesteps (training)

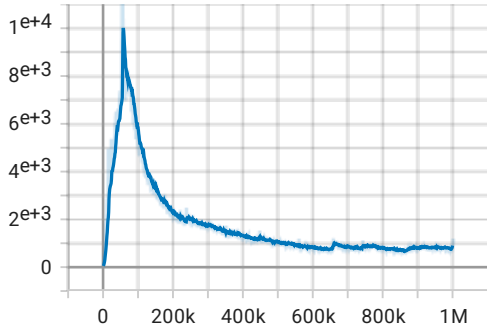


Figure 7-20 REDQ critics ensemble loss during timesteps (training)

Humanoid-v4

Losses of off-policy learning methods

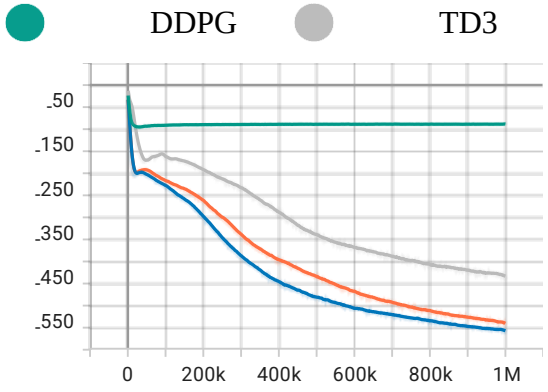


Figure 7-21 Actor loss during timesteps (training)

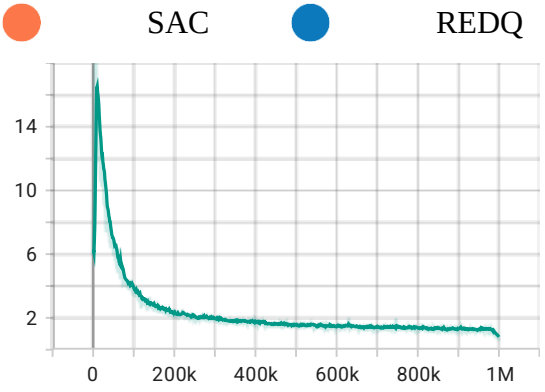


Figure 7-22 Critic loss during timesteps (training)

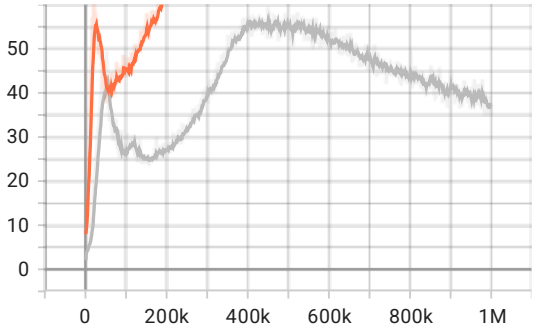


Figure 7-23 Critic 1 loss during timesteps (training)

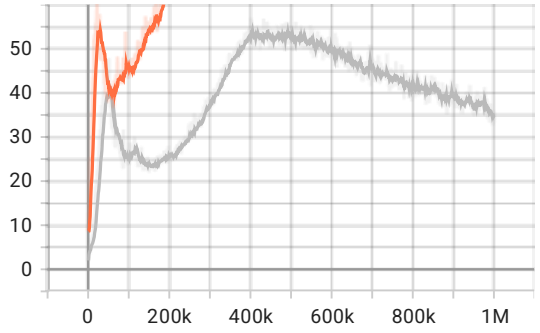


Figure 7-24 Critic 2 loss during timesteps (training)

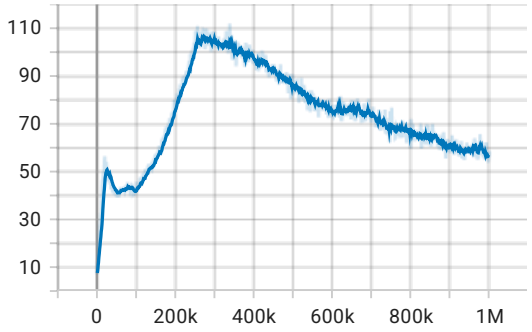


Figure 7-25 REDQ critics ensemble loss during timesteps (training)

InvertedDoublePendulum-v4

Losses of off-policy learning methods

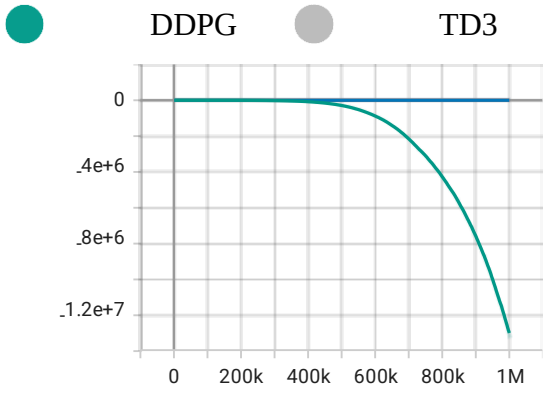


Figure 7-26 Actor loss during timesteps (training)

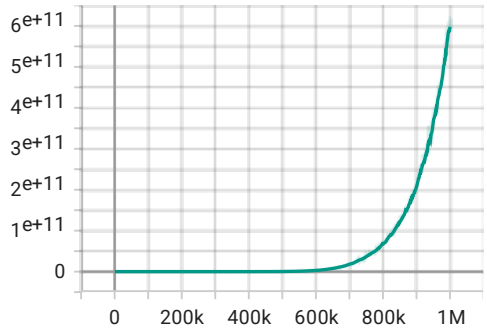


Figure 7-27 Critic loss during timesteps (training)

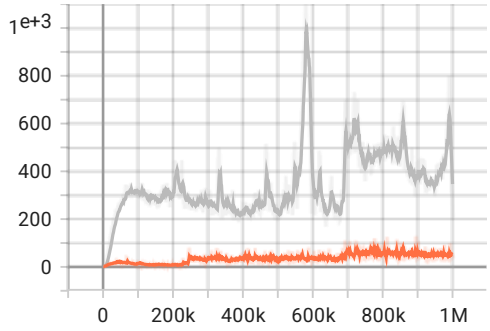


Figure 7-28 Critic 1 loss during timesteps (training)

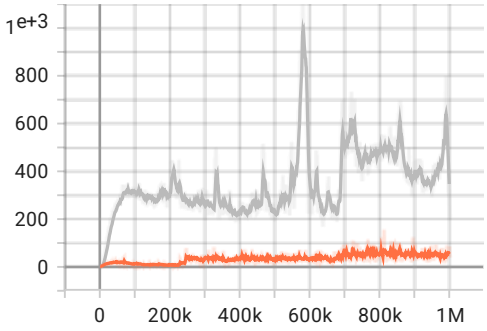


Figure 7-29 Critic 2 loss during timesteps (training)

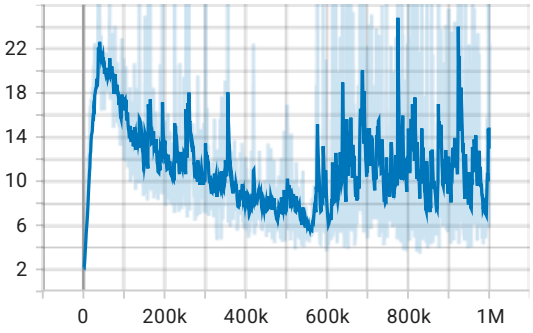


Figure 7-30 REDQ critics ensemble loss during timesteps (training)

InvertedPendulum-v4

Losses of off-policy learning methods

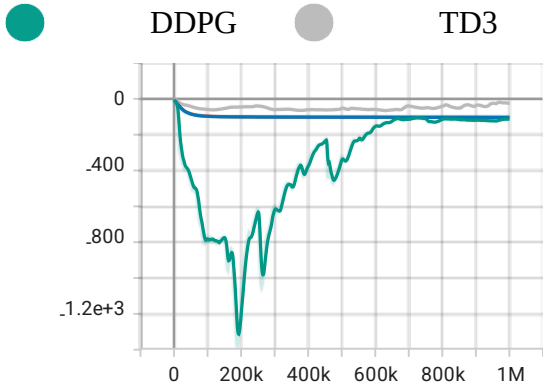


Figure 7-31 Actor loss during timesteps (training)

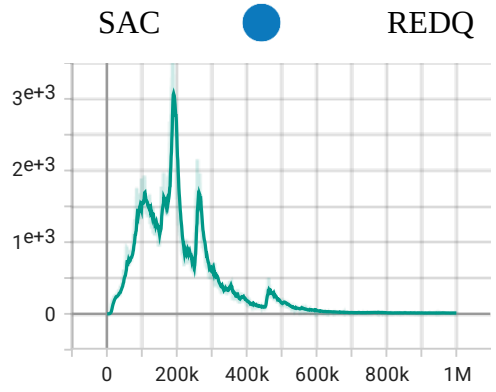


Figure 7-32 Critic loss during timesteps (training)

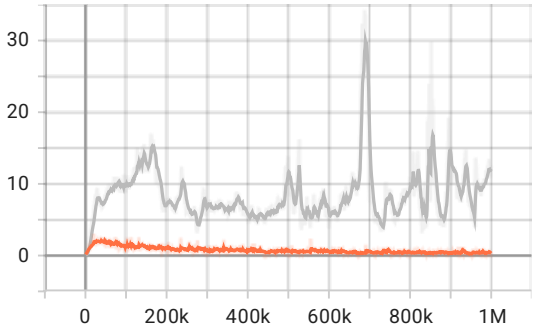


Figure 7-33 Critic 1 loss during timesteps (training)

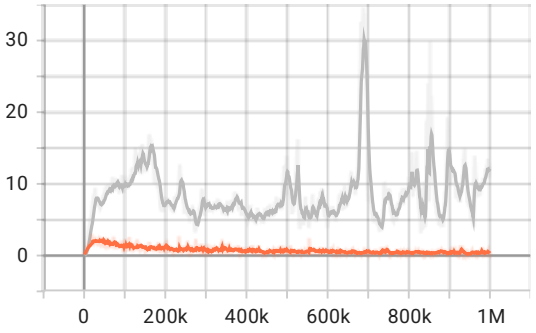


Figure 7-34 Critic 2 loss during timesteps (training)

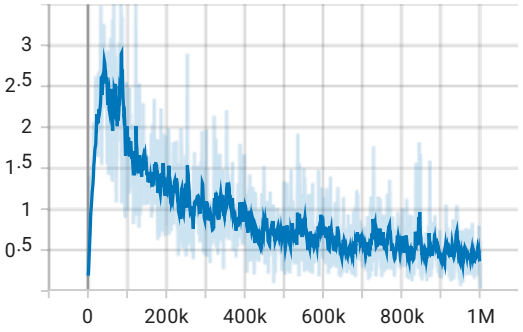


Figure 7-35 REDQ critics ensemble loss during timesteps (training)

Pusher-v4

Losses of off-policy learning methods

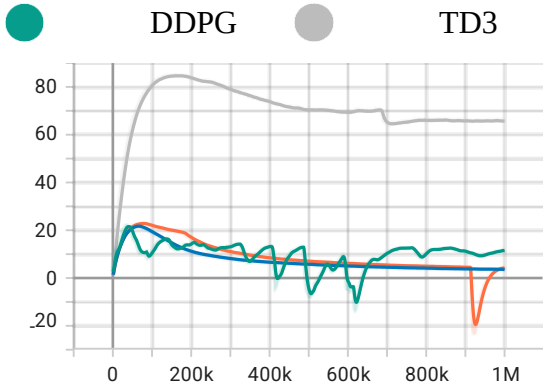


Figure 7-36 Actor loss during timesteps (training)

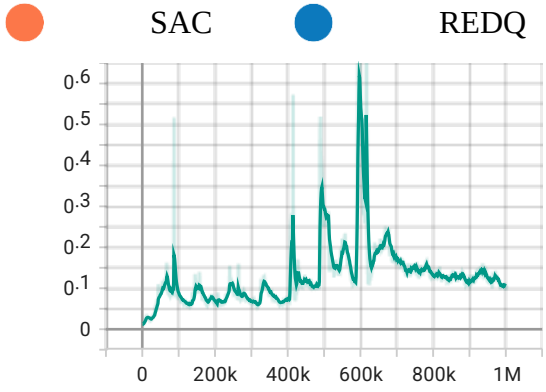


Figure 7-37 Critic loss during timesteps (training)

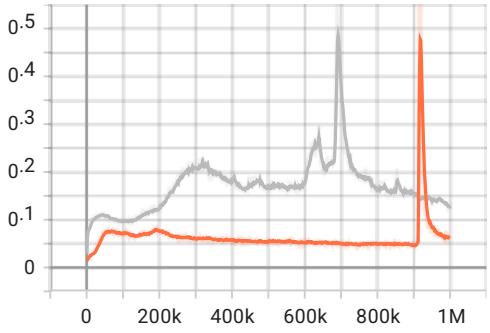


Figure 7-38 Critic 1 loss during timesteps (training)

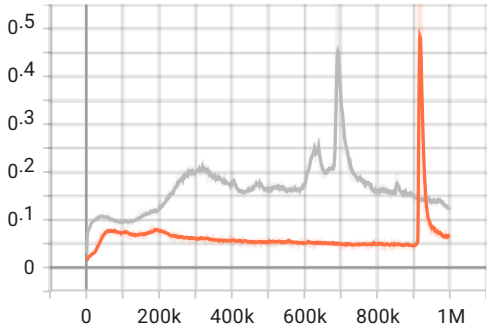


Figure 7-39 Critic 2 loss during timesteps (training)

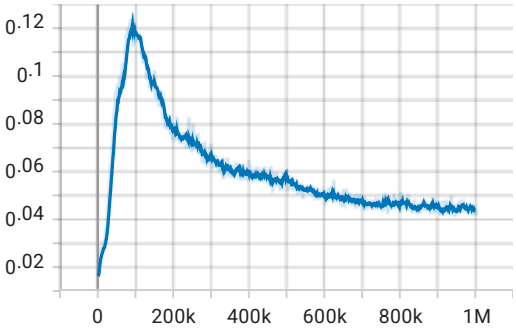


Figure 7-40 REDQ critics ensemble loss during timesteps (training)

Reacher-v4

Losses of off-policy learning methods

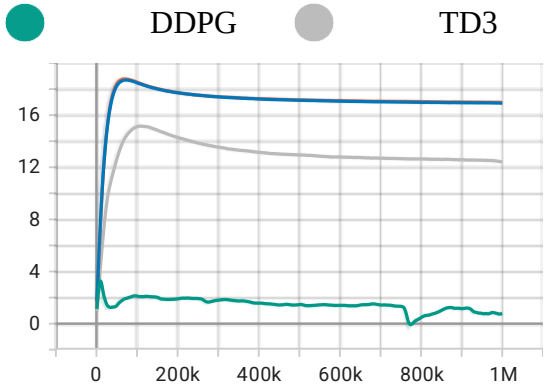


Figure 7-41 Actor loss during timesteps (training)

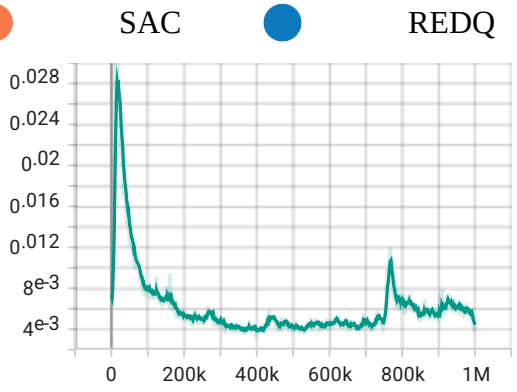


Figure 7-42 Critic loss during timesteps (training)

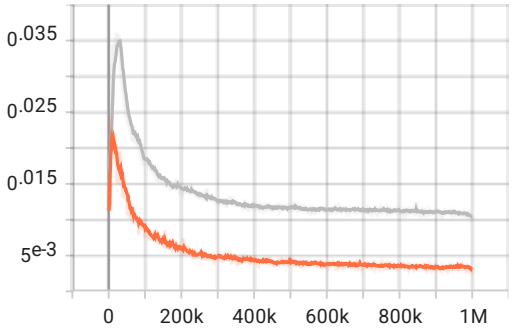


Figure 7-43 Critic 1 loss during timesteps (training)

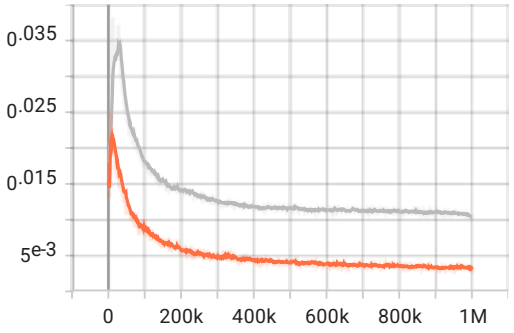


Figure 7-44 Critic 2 loss during timesteps (training)

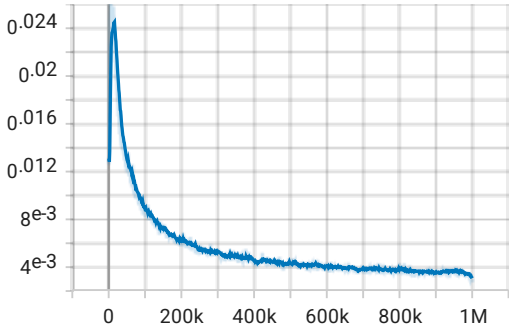


Figure 7-45 REDQ critics ensemble loss during timesteps (training)

Swimmer-v4

Losses of off-policy learning methods

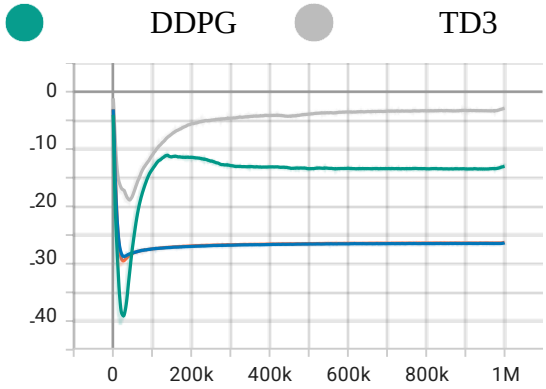


Figure 7-46 Actor loss during timesteps (training)

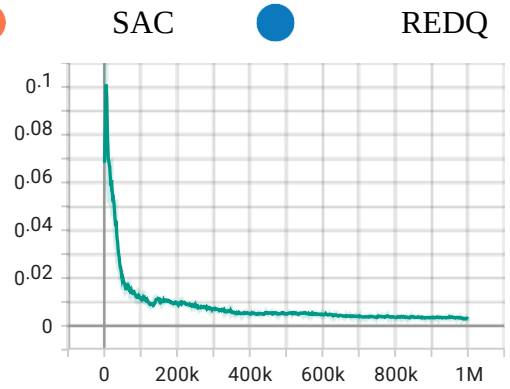


Figure 7-47 Critic loss during timesteps (training)

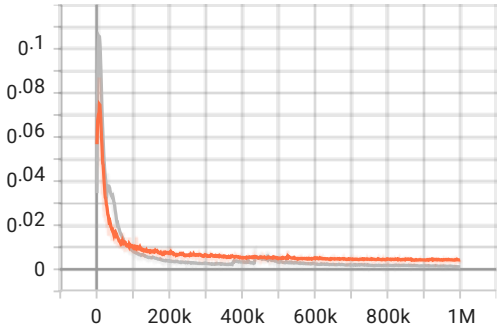


Figure 7-48 Critic 1 loss during timesteps (training)

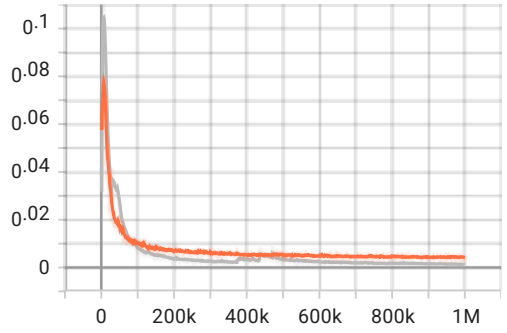


Figure 7-49 Critic 2 loss during timesteps (training)

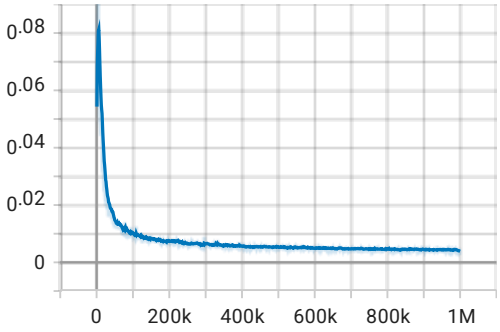


Figure 7-50 REDQ critics ensemble loss during timesteps (training)

Walker2d-v4

Losses of off-policy learning methods

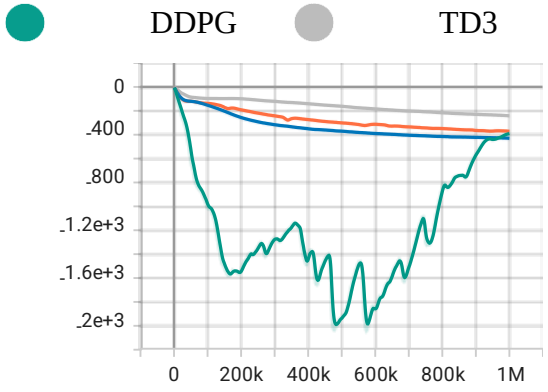


Figure 7-51 Actor loss during timesteps (training)

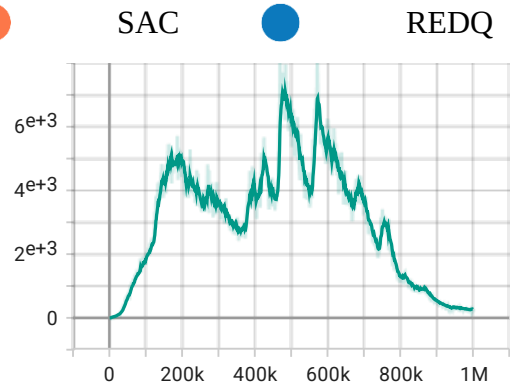


Figure 7-52 Critic loss during timesteps (training)

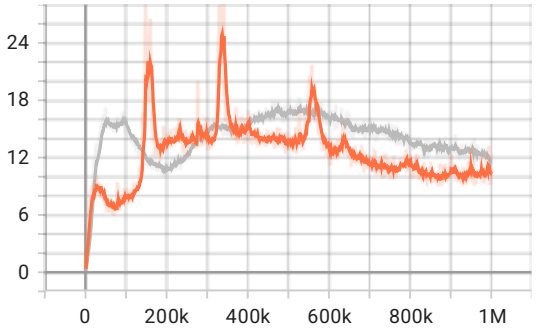


Figure 7-53 Critic 1 loss during timesteps (training)

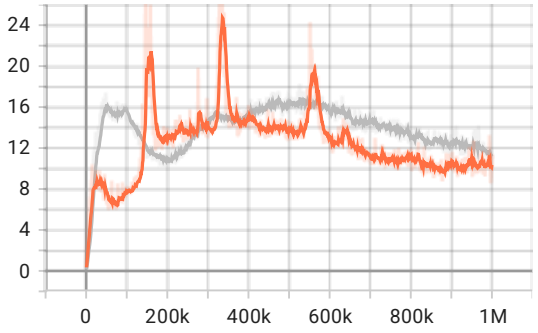


Figure 7-54 Critic 2 loss during timesteps (training)

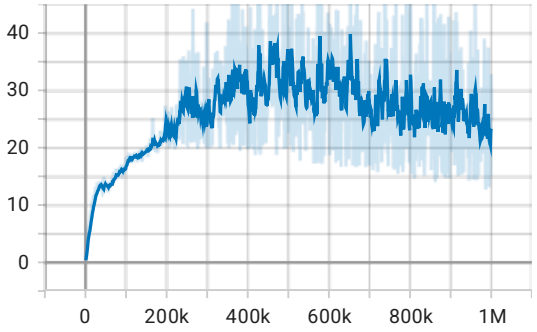


Figure 7-55 REDQ critics ensemble loss during timesteps (training)

Chapter 4: Losses from experimental results

FetchReach-v3

Losses of off-policy learning methods

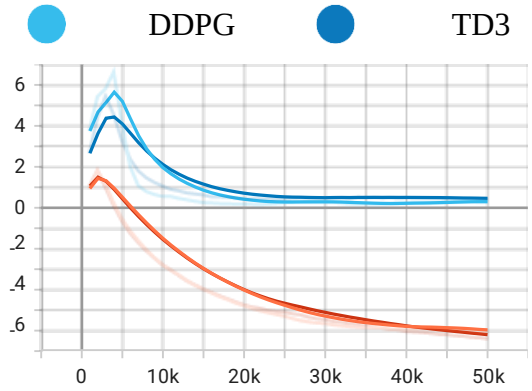


Figure 7-56 Actor loss during timesteps (training)

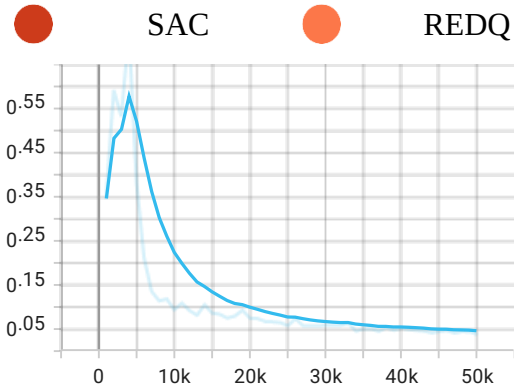


Figure 7-57 Critic loss during timesteps (training)

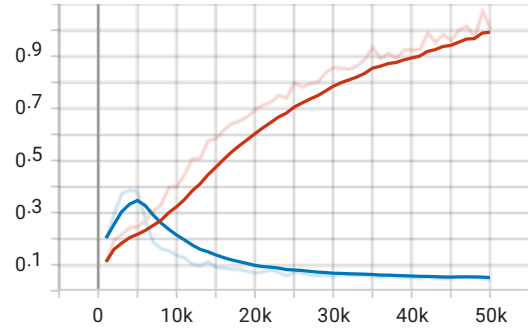


Figure 7-58 Critic 1 loss during timesteps (training)

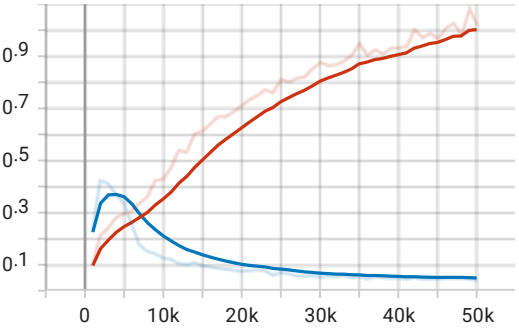


Figure 7-59 Critic 2 loss during timesteps (training)

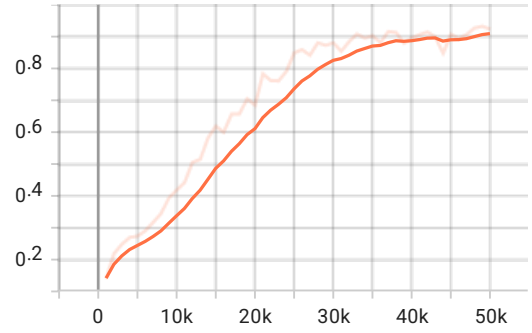


Figure 7-60 REDQ critics ensemble loss during timesteps (training)

FetchPush-v2

Losses of off-policy learning methods

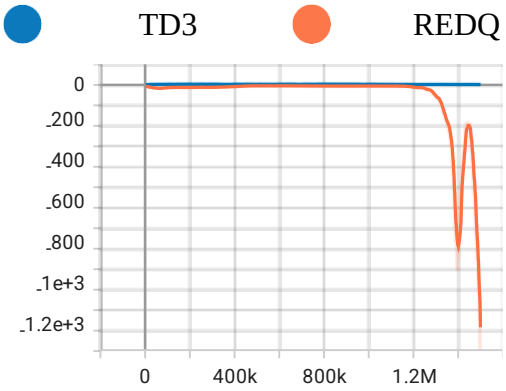


Figure 7-61 Actor loss during timesteps (training)

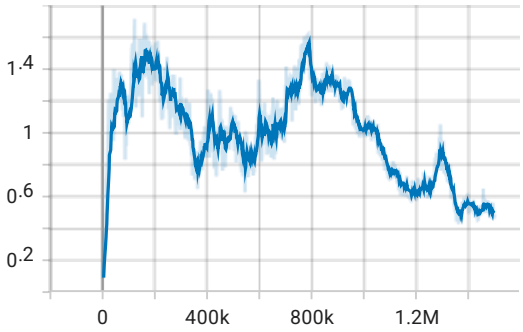


Figure 7-62 Critic 1 loss during timesteps (training)

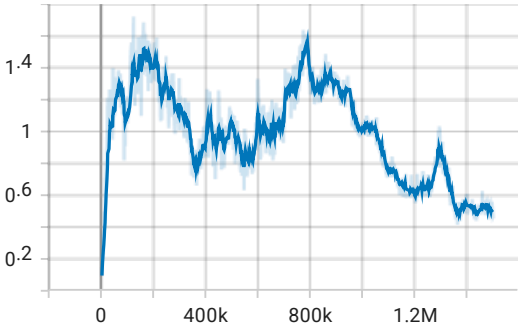


Figure 7-63 Critic 2 loss during timesteps (training)

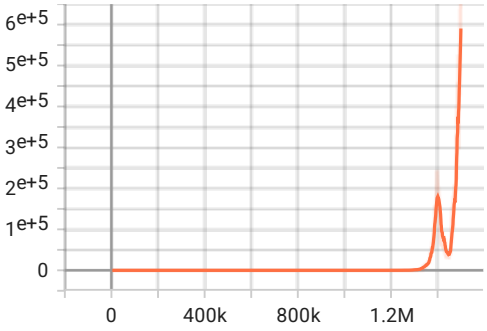


Figure 7-64 REDQ critics ensemble loss during timesteps (training)

FetchPickAndPlace-v2

Losses of off-policy learning methods

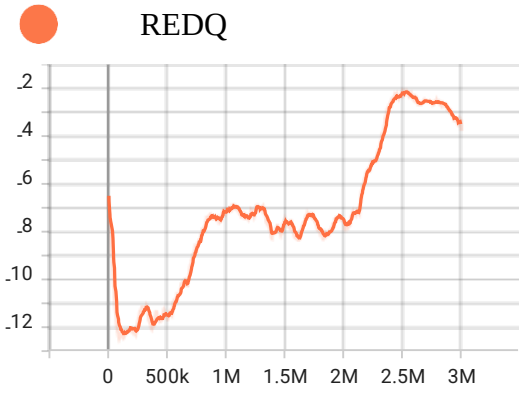


Figure 7-65 Actor loss during timesteps (training)

Figure 7-66 REDQ critics ensemble loss during timesteps (training)