Sample Complexity Analysis of Transfer Learning for Deep Reinforcement Learning Models

Malek Ben Alaya

Bachelor’s thesis

Sample Complexity Analysis of Transfer Learning for Deep Reinforcement Learning Models

Malek Ben Alaya

30. September 2021

Chair of Data Processing
Technische Universität München

Malek Ben Alaya. *Sample Complexity Analysis of Transfer Learning for Deep Reinforcement Learning Models.* Bachelor's thesis, Technische Universität München, Munich, Germany, 2022.

Supervised by Prof. Dr.-Ing. Klaus Diepold and Luca Saccheto, M.Sc; submitted on 30. September 2021 to the Department of Electrical and Computer Engineering of the Technische Universität München.

Chair of Data Processing, Technische Universität München, 80290 München, Germany, `http://www.ldv.ei.tum.de/`.

Abstract

Deep Reinforcement Learning is a powerful area of Machine Learning that can be used to solve extremely complex control tasks. In context of this bachelor thesis, we implement different deep Reinforcement Learning models on a drone to perform a hovering task in a Transfer Learning setting; first we pre-train the Reinforcement Learning models on the differential equations of the drone where an amount of knowledge is acquired, then we transfer this knowledge by post-training the models on a 3D simulation environment. By means of commonly used metrics, we evaluate the benefits of Transfer Learning to the learning process of the desired task compared to the case where no knowledge is transferred. Furthermore and most importantly, we analyze the sample complexity of the post-training of the implemented deep Reinforcement Learning algorithms. This allows us to draw conclusions about which models deliver the overall best performance and are most appropriate for the combination of deep Reinforcement Learning with Transfer Learning in this specific use case.

Contents

1 Introduction

Reinforcement Learning (RL) is a sub-field of Machine Learning (ML) that relies on learning in a trial and error approach. In recent years, this powerful method has been combined with Deep Learning [13], which gave birth to the field of deep Reinforcement Learning. Through the power of deep neural networks, Reinforcement Learning could be extended to solve more complex tasks. To name just a few domains of applications, deep RL has been extensively used in finance and trading [11], natural language processing [25], video games, where it has surpassed human capabilities [28], and robotic tasks [1].
In this work, we analyze the sample complexity of various deep RL methods applied to control a quadcopter to achieve a hovering task on a 3D simulation environment. However, deep RL methods are known to be data-hungry methods, i.e., they require a bigger amount of data compared to other machine learning techniques in order to achieve comparable results [10]. In the aim of speeding up the convergence of such algorithms, we make use of Transfer Learning (TL), which is a ML approach that aims to re-use knowledge acquired in solving a certain task to solve a different but related task [3]. This method compares to how humans learn: e.g., a person that knows how to ride bicycles would probably benefit from that experience to learn how to ride bikes, instead of learning it from scratch.
Up until now, research on applying RL to simulations has shown very promising results, where great successes in numerous fields were achieved. On the other hand, attempts to apply RL to the real world have shown less promising results, illustrating how unreliable RL could be. This is due to it generally requiring a lot of environment interactions to be able to learn [27]. For an expensive physical system, it is not sustainable to crash countless times in order to learn a certain task. This motivates our efforts to contribute to a safer application of RL in the real world: our goal is to show that with the help of TL, it is possible to use RL to train physical systems in the real world in a safe and controlled manner.
What we do is we test TL for RL in a sim-to-sim setting. A successful application of TL in the sim-to-sim setting would lay the ground for its successful application in a sim-to-real setting, which would benefit a safer application of RL in the real world.
For real-world drone applications, one would pre-train an RL model on simulation and then transfer the model and fine-tune it on real-world data. Because of the complications and expenses that arise in the real world, in this work we use the analogous approach to transfer from a simpler to a more sophisticated simulation: we pre-train deep RL models on a simple simulation environment and then transfer the models and

fine-tune them on the more complex 3D simulation environment. For each of the used deep RL algorithms, we investigate the effects of TL on the learning performance in the 3D simulation environment, compared to the case where no TL is used. We then proceed with the algorithms that benefit from TL to analyze their sample complexity in solving the hovering task in the 3D simulation environment.

2 State of the art

In 1999 Sutton and Barto [21] laid the foundations of RL by introducing its basic concepts and algorithms.

More Recently, the combination of RL with Deep Learning [13] gave birth to the field of deep RL. In 2013 Mnih et al. [16] kickstarted deep RL by introducing the algorithm "Deep Q network" (DQN), which is capable of human level performance in many Atari video games using raw pixel inputs. To achieve that, deep neural networks were used as function approximators to estimate the value function.

However, while DQN is suited to environments with high dimensional input spaces, it can only handle environments with discrete and low dimensional action spaces. In 2015 Lillicrap et al. [14] solve this problem introducing the algorithm "Deep deterministic policy gradient"(DDPG), which can be thought of as the extension of DQN to continuous action spaces. Through this algorithm, applications of deep RL could be extended to continuous physical control tasks.

While DDPG can be successful in certain tasks, its major failure mode is the overestimation of the value-function, which often leads to a sub-optimal policy. Twin delayed DDPG (TD3) [7], which is an improvement of DDPG, addresses this problem by taking the minimum value between two value function approximators and delaying policy updates compared to DDPG.

Concurrently to TD3, Haarnoja et al. [8] introduce soft actor-critic (SAC), which is an off-policy algorithm that aims to maximize the expected rewards while also maximizing entropy. This means succeeding at the task while behaving in the most random way possible. SAC was able to outperform the previous state-of-the-art on-policy and off-policy algorithms on continuous-control benchmark tasks while showing great stability with respect to the choice of the hyperparameters.

Proximal policy optimization (PPO) [19] is a simpler to implement version of trust region policy optimization (TRPO) [18], which offers the same qualities of the latter in terms of stability and reliability. PPO is an on-policy algorithm that directly optimizes a policy to solve a control problem. It makes use of a trust region strategy [20], which aims to keep the new and old policy close to each other when performing a policy update, thus resulting in a stable learning process. PPO is able to reach a better performance level compared to other policy search algorithms, both on continuous control domains such as humanoid running and steering in simulation environments as well as on discrete domains such as the Atari domain [19].

Furthermore, a real-world application of PPO is presented in [1], where the algorithm was used to learn dexterous in-hand manipulation on a physical five-fingered hand.

Because normally deep RL algorithms require millions of training samples to achieve good results [1], training directly on the physical system was not practical. Instead, a sim-to-real TL approach was adopted. In this approach, the training of a policy does not occur on the physical system but rather on a distribution over many simulations of the physical system.

A formalization of the transfer problem in general and in the context of RL is presented in [12]. Therein, a framework that is used to categorize different types of TL settings in the context of RL is presented. Additionally, the work evokes a categorization of the types of knowledge that can be transferred. Furthermore, Metrics that measure the benefits of TL on the performance of an RL algorithm are presented in [22, 23].

3 Theory and Methods

3.1 Reinforcement Learning

RL designates a family of ML algorithms, that learn in a trial and error manner [21].

An RL agent learns solving a task by interacting with the environment it lives in. At every step of interaction, the RL agent takes an action according to its current policy. Based on that action, the environment transits to a new state and sends a reward signal characterizing the quality of the (new-state)-action pair to the RL agent. With the goal of maximizing its expected cumulative reward, the agent then updates its policy using RL methods that will be discussed later in this work. Fig. 3.1 shows the principle of working of an RL agent.

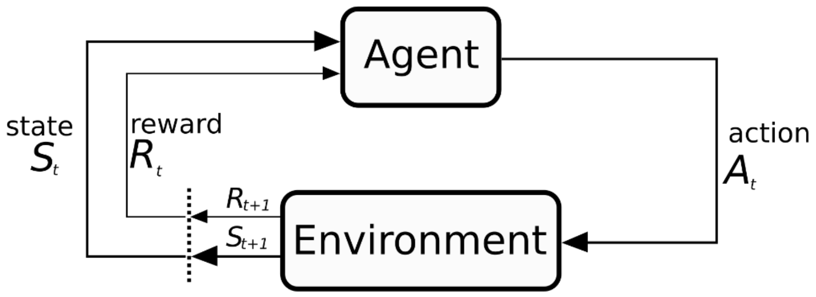


Figure 3.1: RL: at each time step: Agent chooses action A_t, Environment returns reward R_{t+1}, and the new state S_{t+1}.

Typically, a RL problem can be formalized as an agent acting on an environment which is modelled as a Markov Decision Process (MDP) [24]. An MDP can be represented as a tuple $M = (S, A, T, R, \gamma)$ in which:

- S is the state space.
- A is the action space
- $T : S \times A \times S \rightarrow R$ is the transition probability distribution, where $T(s'|s, a)$ specifies the probability of the state transitioning to s' upon taking action a from state s.

- $R : S \times A \times S \longrightarrow R$ is the reward function , where $R(s, a, s')$ is the reward an agent can get by taking action a from state s with the next state being s' .
- γ is a discount factor, with $\gamma \in [0, 1]$.

Both the state and action spaces of an MDP can be either discrete or continuous. This distinction is important as there are RL methods that only apply to the one case or to the other.

In this work, the terms MDP, domain and task can be used interchangeably. Next, We start by defining key concepts in RL.

Policies: An RL agent acts in an MDP M according to its policy. A policy $\pi : S \longrightarrow A$ is a mapping from states to actions (deterministic case) or from states to distributions over actions (stochastic case). For stochastic policies and discrete actions spaces $\pi(a \mid s)$ denotes the probability that the agent takes action a from state s.

Trajectories: A trajectory τ is a sequence of states and actions. $\tau = (s_0, a_0, s_1, \ a_1, s_2, \ldots)$, Where the first state is sampled from the initial-state distribution of the MDP M and the actions are chosen according to the policy of the agent.

Reward and return: The reward function R is a focal point when defining an MDP. It basically defines the task required to be solved from an RL agent and needs to be designed carefully.
Let $r_t = R(s_t, a_t, s_{t+1})$ be the reward that an agent receives at time step t by taking action a_t in state s_t, and where the next state transits to s_{t+1}.

The goal of the agent is to maximize the expected cumulative reward, also known as the return, over a trajectory. We distinguish two types of return in RL. The first one is the finite-horizon undiscounted return defined as:

$$R(\tau) = \sum_{t=0}^{T} r_t, \tag{3.1}$$

which is the sum of rewards over a finite number of timesteps T.
The second type of return is the infinite horizon discounted return, defined as:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t, \tag{3.2}$$

which is the sum of all rewards ever obtained by the agent, discounted in time.
Normally, an infinite sum of rewards would keep growing in time, which results in an unbounded return. Dealing with unbounded mathematical terms is computationally intractable. Therefore, a solution to this problem is the discount factor γ. It scales down

the future rewards increasingly in time to keep the return term bounded. Intuitively, the discount factor γ determines how far in time future rewards should be taken into account.

Value functions: Given an MDP M and a policy π, the value function of a state s is the expected cumulative reward the agent can get, if the agent starts in that state and then acts according to policy π in the environment forever after. It is defined over the state-space as:

$$V^{\pi}(s) = \mathbb{E}_{\tau \sim \pi}[R(\tau) \mid s_0 = s] \tag{3.3}$$

The value function is used to measure the quality of being in state s. Similar to the value function, a policy π has an action-value function. It is defined over the state and action-space as:

$$Q^{\pi}(s, a) = \mathbb{E}_{\tau \sim \pi}[R(\tau) \mid s_0 = s, a_0 = a] \tag{3.4}$$

The action-value function of a state-action pair (a, s) gives the expected cumulative reward an agent can get, if it takes an action a in state s and then acts according to policy π in the environment forever after. It characterizes the quality of taking action a from state s.

The main goal of an RL agent is to maximize the expected cumulative rewards by learning an optimal policy π^* , so that:

$$\forall s \in \mathbb{S}, \pi^*(s) = \arg\max_{a \in A} Q^*(s, a), \tag{3.5}$$

where

$$Q^*(s, a) = \sup_{\pi} Q^{\pi}(s, a) \tag{3.6}$$

On-policy vs off-policy RL: RL algorithms generally try to iteratively improve a policy until it converges to an optimal policy. In order to compute an optimal policy, on-policy algorithms try to improve a policy π_1 by means of episodes generated using that same policy π_1. On the other side, off-policy algorithms try to improve a policy π_1 by means of episodes generated using other policies than π_1. On-policy methods tend to be more stable during training compared to off-policy methods but on the other hand they are generally less sample efficient.

3.2 Deep Reinforcement Learning

In order to learn an optimal policy, RL methods often need to estimate value functions and action-value functions. Traditional RL methods use tabular representations and linear function approximators for that matter. With a growing state-space, using tabular methods may become impractical. For a continuous state space it even becomes impossible. Furthermore, for many MDP's, linear function approximators may show poor results [17]. Therefore, non-linear function approximators such as neural networks are often used in this case. Using neural networks as function approximators for action-value function lies under the field of deep RL. Moreover, in deep RL, policies can be directly parameterized through neural networks.

3.3 Transfer Learning

TL [3] is a technique used to leverage knowledge learned on a source task to improve the learning performance on a target task. In general, TL can be effective at speeding up learning or at achieving better final (asymptotic) learning results. Fig. 3.2 shows a comparison between the learning process in traditional ML and in TL.

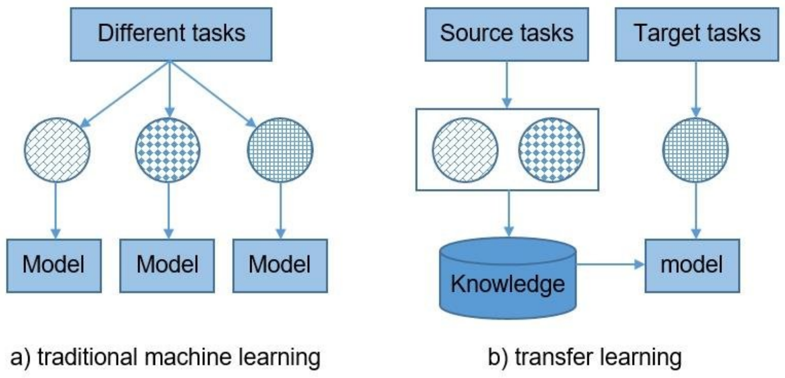


Figure 3.2: Difference of learning process of (a) traditional ML and (b) TL [6]

.

Transfer Learning in the context of Reinforcement Learning:

Let $G_s = \{M_s|, M_s \in G_s\}$ be a set of source domains, which provides prior knowledge D_s that is accessible by the target domain M_t . Normally, by utilizing the information from D_s, the target agent learns better in the target domain M_t, compared with not utilizing it. We use $M_s \in G_s$ to refer to a single source domain. a more concrete description of TL from the RL perspective is the following [29]: Given are G_s as defined above and a target domain M_t. The objective of Transfer Learning in RL is to learn an optimal policy π^* for the target domain, by utilizing both exterior information D_s from M_s and interior information D_t from M_t, s.t.:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{s\sim\mu_0^t, a\sim\pi}[Q^\pi(s,a)],$$

where μ_0^t is the distribution of the initial states of the MDP in question and $\pi = \phi(D_s \sim M_s, D_t \sim M_t) : S_t \to A_t$ is a policy which maps states to actions for the target domain

M_t, learned based on information from both D_t and D_s.
In the above definition, $\phi(D)$ denotes a learned policy based on information D.

Transfer Learning settings:

Transfer from source task to target task with fixed state-action space [12]: in this setting, only one source task and one target task are involved in transfer and they have the same state-action space $S{\times}A$. The difference between both tasks can be the transition dynamics T and/or the reward function R. In this work we make use of this setting.

Transfer Learning knowledge (transferred knowledge):

Knowledge transfer approaches can be classified into three categories: instance transfer, parameter transfer and representation transfer [12]. In this work, we are only interested in parameter transfer. Thus, we give an explanation of it in the following:
RL algorithms are characterized by a set of input parameters that define the initialization of the learning process and the behavior of the learning algorithm itself. Some TL approaches adapt these parameters based on experience learnt on source tasks. Particularly, in transfer settings where only one source task is used, it is common to transfer initial solutions (i.e., policies or value functions) and use them to initialize the learning algorithm on the target task.

Transfer Learning metrics:

Many metrics to measure the benefits of TL on the performance of a learning algorithm already exist, which were first presented in [22]. Fig. 3.3 shows a graphical interpretation of these metrics on the learning curve of an RL algorithm.

- 1) Jumpstart: The initial performance of an agent in a target task may be improved by a transfer of knowledge from a source task. This metric answers the question whether TL increases the initial performance of an agent relative to the performance of an initial (random) policy. [23]

- 2) Asymptotic Performance: The final learned performance of an agent in the target task may be improved via TL. This metric serves as a comparison metric between the final performance of an agent both with and without transfer.

- 3) Total Reward: The total reward accumulated by an agent (i.e., the area under the learning curve) may be improved if it uses transfer, compared to learning without transfer.

- 4) Transfer Ratio: The ratio of the total reward accumulated by the transfer learner and the total reward accumulated by the non-transfer learner. It is defined as:

$$r = \frac{area\ under\ curve\ with\ transfer - area\ under\ curve\ without\ transfer}{area\ under\ curve\ without\ transfer}$$

This metric is most appropriate if both transfer and non-transfer learners achieve the same learning performance or if the task has to be learnt in a pre-defined amount of time. Otherwise the ratio will depend on for how much time the agents act on the environment.

- 5) Time to Threshold: The learning time needed by the agent to achieve a pre-specified performance level may be reduced via knowledge transfer. For this metric, choosing a (potentially arbitrary) performance the agent must achieve can be tricky.

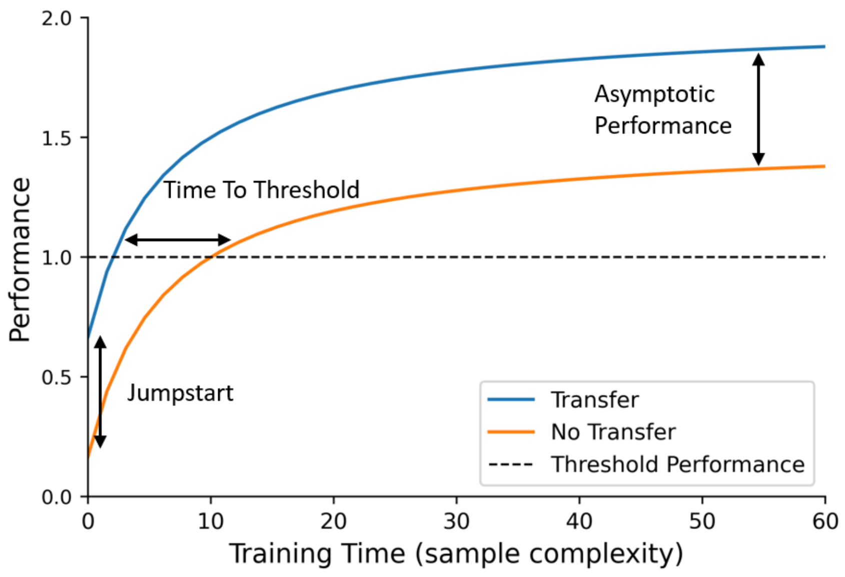


Figure 3.3: Different metrics to evaluate TL benefits to a learning algorithm exist. This graph shows benefits to the jumpstart, time to threshold, asymptotic performance and total reward (total area under the learning curve).

3.4 Methods and Algorithms

To compute an optimal policy of an RL agent, two classes of RL methods exist: methods that rely on learning value functions and then inferring an optimal policy, and methods that rely on directly parameterizing a policy and optimizing it, i.e., policy search methods. There is also an approach that combines both methods, which is the actor-critic approach. In the following we will briefly explain how such methods work.

Q-learning: Among methods that rely on learning value functions to compute an optimal policy, we count Q-learning [4]. This method relies on the following rule to learn a Q function:

$$Q^{\pi}(s_t, a_t) \leftarrow Q^{\pi}(s_t, a_t) + \alpha\delta, \tag{3.7}$$

where α is the learning rate and $\delta = y - Q^{\pi}(s_t, a_t)$ is referred to as the temporal difference (TD) error, where $y = r_t + \gamma \max_a Q^{\pi}(s_{t+1}, a)$ is a target that directly approximates Q^* . Q-learning is off-policy, which means the transitions $(s_t,\ a_t,\ s_{t+1})$ that it uses for the policy update are not necessarily generated by the derived policy [2].

Policy search: Policy search methods directly parameterize a policy through a set of parameters θ and try to update them such that the expected return is maximized. [2] In this work this optimization process is done using exclusively gradient-based update rules. That is because they are more sample efficient compared to gradient-free methods for the case when policies possess a large number of parameters, which is the case in this work as we parameterize policies through neural networks.

Actor-critic methods: In actor-critic methods, a critic is used to estimate the value function of each state which gives a feedback that guides the actor towards choosing an action [2]. Fig. 3.4 shows a block diagram of the architecture of the actor-critic approach.

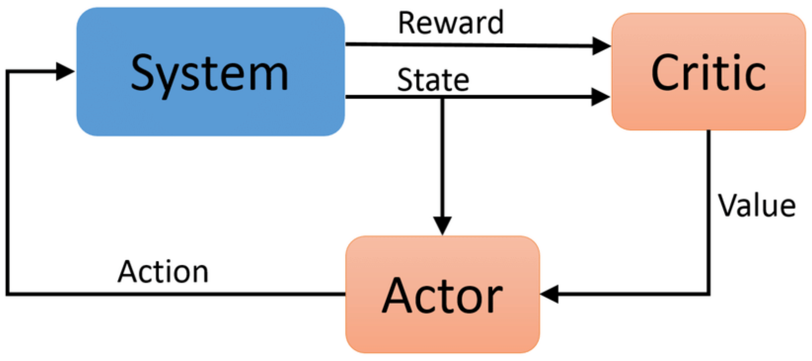


Figure 3.4: A high-level block diagram of the architecture of the actor-critic approach [26].

3.4.1 PPO [19]

PPO is a stochastic on-policy algorithm that computes the optimal policy of an RL agent by directly parameterizing its policy and optimizing it. The idea behind PPO is to take the biggest improvement step in each policy update while keeping the old and new policy close to each other. In each policy update, PPO finds the policy that maximizes a sample mean estimate of the clipped surrogate objective function:

$$L(s,a,\theta,\theta_k) = min(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s,a), clip(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}, 1-\epsilon, 1+\epsilon) A^{\pi_{\theta_k}}(s,a)), \tag{3.8}$$

where $A^{\pi_{\theta_k}}(s,a) = Q^{\pi_{\theta_k}}(s,a) - V^{\pi_{\theta_k}}(s)$ is an advantage estimate, which tells how much better is taking action a from state s, over taking a random action according to π_{θ_k}.
Therefore, the policy parameter update looks like the following:

$$\theta_{k+1} = \arg\max_\theta \mathbb{E}_{s,a\sim\pi_{\theta_k}}[L(s,a,\theta,\theta_k] \tag{3.9}$$

The function $L(s,a,\theta,\theta_k)$ measures how much better policy π_θ performs compared to the old policy π_{θ_k}. The clipping embodies the core idea behind PPO: it is a technique used to keep the new policy close to the old policy.

3.4.2 DDPG [14]

Straightforwardly applying Q-learning to continuous action spaces is computationally intractable, because an optimization process needs to occur whenever an action is to be taken in the environment; this optimization is too slow for the case of large function approximators like neural networks and nontrivial action spaces [14]. Therefore, by using an actor-critic approach, DDPG computes the optimal policy of an RL agent by combining Q learning [4] and policy optimization methods. The key idea behind DDPG is to interleave learning an approximator to Q^* with learning an approximator to π^*. Instead of running a computationally expensive optimization subroutine each time $\max_a Q^\pi(s_{t+1},a)$ needs to be computed when updating the Q function according to Eq. 3.7, $\max_a Q^\pi(s_{t+1},a)$ is approximated through $\max_a Q^\pi(s_{t+1},a) \approx Q^\pi(s_{t+1},\mu(s_{t+1}))$, where μ is an approximator to π^* which is learnt through a gradient-based learning rule.
The main characteristics of DDPG are that it is off-policy and works on continuous action spaces.

3.4.3 TD3 [7]

TD3 is a deterministic off-policy algorithm that is used for continuous action spaces. It is an improvement of DDPG as it tries to solve the over-estimation bias of value functions that DDPG is known for. For that matter, it uses three main tricks. The first one is using two Q-functions instead of one and it uses the minimum of the two functions to build the target in the Q-function update. The second trick is that it delays the policy network update compared to DDPG, in order to reduce the per-update error. More specifically, DDPG applies a policy update for every Q-function update whereas the original paper of TD3 applies a policy update for every two Q-functions updates. The last trick is that TD3 adds noise to the target action in order to prevent the policy from exploiting the over-estimation errors of the Q-function.

3.4.4 SAC [8]

SAC is a stochastic off-policy actor-critic algorithm that alternates between learning a policy π and a Q-value function . It tries to maximize the expected return while also maximizing the entropy of the policy:

$$J(\pi) = \mathbb{E}_{(s_t,a_t,s_{t+1})\sim\pi}[\sum_t R(s_t, a_t, s_{t+1}) - \alpha log(\pi(a_t \mid s_t))] \tag{3.10}$$

Here The term $-log(\pi(a_t \mid s_t))$ corresponds to the entropy of the policy π and α is a hyper-parameter that is used to balance between exploitation and exploration. In this work, we use an updated version of SAC where α is learnt automatically by the algorithm.

It is important to note that for the stochastic policies (SAC/PPO) and continuous action spaces, the policy network takes in the state as input and outputs the mean and standard deviation of an action probability distribution.

4 Task and System overview

In this work, the main idea is to apply different deep RL algorithms on a Transfer Learning setting: we first pre-train different models of such algorithms on a source task where an amount of knowledge is acquired. In a subsequent step, for each pre-trained model, we transfer the acquired knowledge gained during pre-training and we further build on it by post-training the model on the target task. In a final step, we analyze the sample complexity of the different models in the target task.
We first begin by explaining the source task and the target task that we consider in this work. We consider the task of drone hovering; starting from an initial position with a strictly positive altitude value, the goal is to control the drone to stay at that same position. The source and target tasks that we consider in this work both consist in the drone having to hover in place. However, the transition dynamics of the the tasks are different. The transition dynamics of the source task are based on the differential equations of the drone. On the other hand, the transition dynamics of the target task are based on the Pybullet simulation environment [5]. Therefore, both domains share the same state space, action space and reward function but their transition dynamics are different.

Sec. 4.1 and Sec. 4.2, respectively, present the similarities and differences between both tasks in more details.

4.1 Source and target domains similarities

The following characteristics are shared between the source and target tasks:

- **States** : The state is 20-dimensional and consists of the 3-dimensional drone position and orientation, the drone velocity and angular velocity, the 4-dimensional last action and a 4-dimensional quaternion. Each dimension of the state can take continuous values, which means the state space is continuous.
- **Observations**: The observation is 16-dimensional. It is the same as the state except for the 4-dimensional quaternion which is not included in the observation.
- **Observation noise**: Sensors that measure the states of the drone have measurement uncertainties. We use an uncorrelated gaussian noise which is sampled at every timestep to account for such uncertainties. Table. 4.2 shows the standard deviations of the observation noises.

- **Actions**: Actions are 4-dimensional and correspond to the input of a PID controller, which converts them to PWM signals that act on the 4 motors of the drone. More specifically, the actions correspond to the desired total thrust of the motors and the desired roll, pitch and yaw velocities. Each dimension of the Actions can take continuous values, which means the action space is continuous.
- **Rewards**: The reward given at timestep t is $r_t = 100 * (d_t - d_{t+1}) - p_t$, where p_t is a penalty term and d_t and d_{t+1} are the Euclidean distance between the current and the desired drone position before and after the transition, respectively. The penalty term penalizes the velocity, the angular velocity, the action and the action rate of the drone. Additionally, a penalty of 10 is computed if an episode is cut short due to constraint violation.
- **Timing**: Each environment step corresponds to 2 ms of real time. The maximum episode length is set to 500 environment steps, which corresponds to 1 s of real time.
- **Target position**: The goal position for both domains is set to $[x\,y\,z]^T = [0\,0\,1]^T$.
- **Initial position**: The initial position is equal to the target position $[x\ y\ z]^T = [0\ 0\ 1]^T$.
- **System parameters**: In both domains, the physical parameters of the drone, e.g., the mass or the inertia matrix, are the physical parameters of the Bitcraze Crazyflie 2.13 quadrotor.[1]
- **Constraints**: An episode is cut short if at least one of the constraints is violated. Such constraints include speed, position and orientation limits. These are presented in Table. 4.1.

[1] https://www.bitcraze.io/products/crazyflie-2-1/

4.2 Source and target domains differences

Differential equations of the drone

In the source domain, the dynamics of the drone are derived from its differential equations. Such differential equations are a simple theoretical representation of the drone's dynamics, as they omit many physical effects.

$$\begin{bmatrix}\ddot{r}\\ \ddot{p}\\ \ddot{q}\end{bmatrix} = J^{-1}\left(\begin{bmatrix}\frac{L}{\sqrt{2}}(-F_1 - F_2 + F_3 + F_4)\\ \frac{L}{\sqrt{2}}(-F_1 + F_2 + F_3 - F_4)\\ \kappa(-F_1 + F_2 - F_3 + F_4)\end{bmatrix} - \begin{bmatrix}\dot{r}\\ \dot{p}\\ \dot{q}\end{bmatrix} \times J \begin{bmatrix}\dot{r}\\ \dot{p}\\ \dot{q}\end{bmatrix}\right) \tag{4.1}$$

$$\begin{bmatrix}\ddot{x}\\ \ddot{y}\\ \ddot{z}\end{bmatrix} = \begin{bmatrix}0\\ 0\\ \frac{F_1+F_2+F_3+F_4}{m}\end{bmatrix} - \begin{bmatrix}0\\ 0\\ g\end{bmatrix} \tag{4.2}$$

The matrix J denotes the inertia matrix of the drone, m denotes the mass of the drone, L denotes the distance from the center of gravity of the drone to the center of each motor, g denotes the gravitational acceleration, κ denotes the rotors' torque constant, F_i denotes the thrust of the i-th rotor with $i \in \{1, 2, 3, 4\}$, r denotes the roll angle, p denotes the pitch angle and q denotes the yaw angle.
In Eq. 4.1, terms that include angular acceleration as well as gyroscopic moments have been neglected as they tend to be small and thus their contribution in the equation is also small. [15]
In Eq. 4.2, R denotes a rotation matrix of the body-fixed frame of the drone w.r.t the world frame.

Pybullet simulation environment

In the target domain, the dynamics of the drone are based on the Pybullet simulation environment [5]. In this environment, the representation of the dynamics of the drone is more complex and closer to the real world, because it takes into account physical effects which are omitted in the differential equations. Among these effects, we count the drag effect, which is a type of friction also known as air resistance, and the downwash, which is an effect that reduces lift force generated by the rotors. These effects add up to make a dynamic system more unstable. Therefore, taking them into account makes the control of the drone on Pybullet more challenging than on the differential equations. Fig. 4.1 shows a rendering of the Pybullet simulation environment.

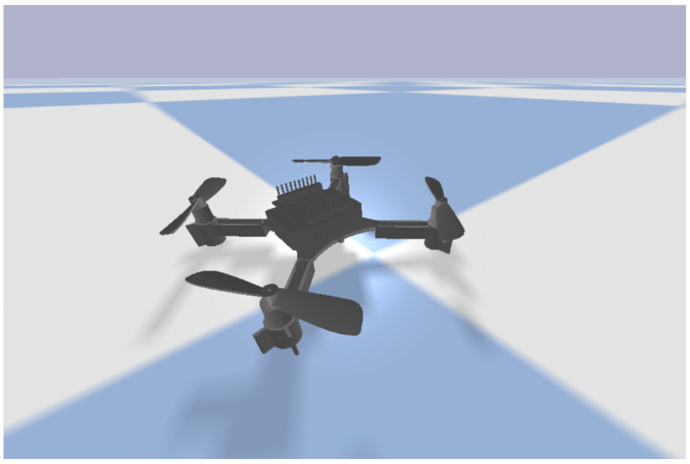


Figure 4.1: A rendering of the Pybullet simulation environment of the Drone.

$\dot{x}$ / $\dot{y}$ / $\dot{z}$	$\dot{r}$/ $\dot{p}$ / $\dot{q}$	r / p	x	y	z
0.25 m/s	100 deg/s	10 deg	0.15 m	0.15 m	1.2 m

Table 4.1: Speed, position and orientation limits that should not be exceeded in the source domain as well as in the target domain.

	$x/y/z$	$r/p/q$	$\dot{x}$ / $\dot{y}$ / $\dot{z}$	$\dot{r}$/ $\dot{p}$ / $\dot{q}$
Standard deviation	0.002 m	$0.1°$	0.02 m/s	1 °/s

Table 4.2: Standard deviations of the zero-mean Gaussian noise applied to the observation variables.

5 Experiments and Results

In this chapter, we aim to compare a variety of deep RL algorithms, namely PPO, DDPG TD3 and SAC, which cover all types of model-free RL algorithms such as on-policy, off-policy, deterministic as well as stochastic algorithms. We first investigate the possible benefits TL can have on the learning performance of the considered algorithms in the target domain, compared to the case where no TL is used. Subsequently, we analyze the sample complexity of the considered algorithms in the target domain. This means analyzing how many environment interactions each algorithm needs in order to achieve the hovering task successfully in the target domain.

When pre-training a model in simulation and then post-training it on real-world data, the amount of data that the simulation offers for pre-training is unlimited whereas real-world data is expensive and not so abundant. Analogously, for our setting, the amount of data available in the source domain is unlimited. On the other hand, the amount of data that we consider accessible in the target domain is in the range of 250-300k time-steps. This is because this range is in the order of magnitude of the amount of data that is collectible in a sensible amount of time in the real world. Therefore, satisfactory results during post-training would mean the drone achieving the hovering task successfully in that range of time steps.

General structure

A major problem with deep RL algorithms is that the training results show great variance across runs in which the exact same hyper-parameter configuration is used [9], which may be due to the stochasticity of the environment dynamics or of the policy. In the experiments that we conduct in Sec. 5.1 and Sec. 5.2, we also encounter this problem. Therefore, in order to have more reliable results for each hyper-parameter configuration and for each algorithm, we adopt the following approach: for a fixed hyper-parameter configuration and a fixed algorithm of choice, we make a certain number of independent runs of the algorithm and then compute the average performance and the corresponding standard deviation.

For all models in this work, the majority of the hyper-parameters match those used in the original papers of the corresponding algorithms. Only the architectures of the neural networks required a manual tuning to achieve better results.

5.1 Transfer Learning Evaluation

In order to investigate the effect of TL on the overall learning performance in the target domain, we run the following experiment for each of the algorithms that we consider in this work:
On the one hand: we fix a set of hyper-parameters for which we pre-train 5 instances of the algorithm in the source domain. For each instance, we save the actor and critic networks that achieve the best evaluation performance. Out of the 5 saved actor and critic networks, we only retain the best one of them. After that, we transfer the retained actor and critic networks, together with the hyper-parameters, to the target domain. Therein, we post-train the weights of the actor and critic networks.
On the other hand, we train another 5 instances of the algorithm, with the same hyper-parameter configuration that was fixed in the first part of the experiment, in the target domain. This time, we randomly initialize the weights of the actor and critic networks instead of initializing them with pre-trained weights. Fig. 5.1, Fig. 5.2, Fig. 5.3 and Fig. 5.4 show the training results of the experiment. As performance metrics, we choose to monitor the episode length and episode return : regarding both of these metrics, a higher value corresponds to a better performance. As mentioned earlier in Chap. 4, the best episode length an RL agent can reach during training in the target domain has the value of 500, which corresponds to the agent performing an episode without violating any of the position and speed constraints mentioned in Chap. 4. The return and episode length shown in the plots are running means, which are averaged over the last 100 episodes of training. This is also valid for all training results in the rest of this work.
It is noteworthy that in the experiment, all training runs, on both the source and target domains, occur for 1 million time steps. We also note that each instance of the considered algorithms performs 5 evaluation episodes every 10K timesteps parallelly to its training process.
We henceforth refer to the 'running mean return' and 'running mean episode length' simply as 'return' and 'episode length', respectively.

By analyzing the curves corresponding to the average (over 5 runs) return of SAC 5.1b, we observe an advantage of the curve corresponding to the case where the weights are transferred. We denote by $k : \left[0, 10^6\right] \to \mathbb{R}$ the function corresponding to the average return of the pre-trained SAC and by $d : \left[0, 10^6\right] \to \mathbb{R}$ the function corresponding to the average return of the randomly initialized SAC. According to the curves of these functions, we observe that $k(x) \geqslant d(x)\ \forall t \in \left[0, 10^6\right]$. By referring to the metrics presented in Sec. 5.1, we observe benefits to the initial return (+33.7 increase) and asymptotic return (+1.6 increase). Moreover, it is to be seen that the pre-trained SAC exhibits a generally smaller standard deviation over the 5 runs regarding the return: the mean standard deviation of the pre-trained SAC is 2.3 whereas for the randomly initialized SAC it is 7.82.

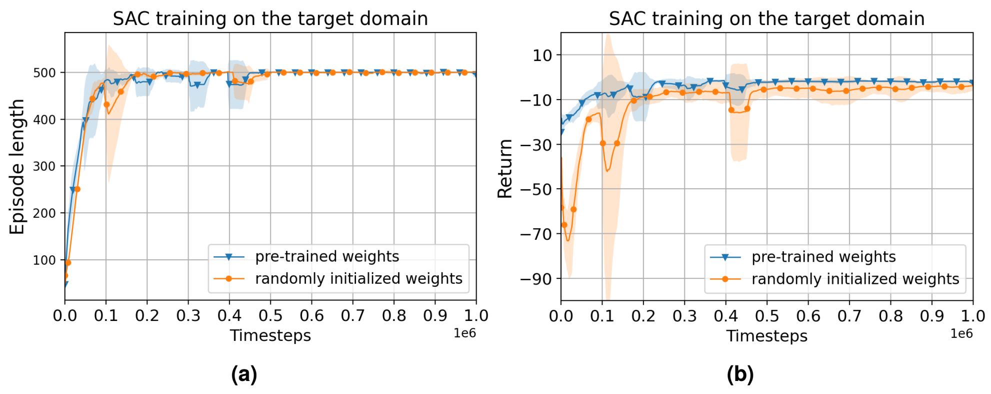


Figure 5.1: Comparison of the episode length 5.1a and return 5.1b of the training of SAC in the target domain with randomly initialized weights and with pre-trained weights. The lines denote the average over 5 runs of 1 million timesteps and the shaded areas correspond to the standard deviations. The blue curves correspond to the pre-trained SAC and the orange curves correspond to the randomly initialized SAC.

By analyzing the curves corresponding to the average (over 5 runs) return 5.2b and episode length 5.2a of PPO, we observe better performances for the case where the weights are transferred. We observe benefits to the initial return (+21.2 increase) and asymptotic return (+1.7 increase). Furthermore, we observe a significant improvement to the learning speed: the pre-trained PPO needs less time to attain the maximum episode length or to converge to the asymptotic return performance. More specifically, the pre-trained PPO needs around 200k time-steps to achieve the maximum episode length and for its return to converge. On the other hand, the randomly initialized PPO needs around 700k time-steps to reach the maximum episode length and around 1 million time-steps for its return to converge, which is a significantly bigger amount of environment interactions than the pre-trained PPO needs. Furthermore, the pre-trained PPO has a slightly better asymptotic episode length value (+10 increase).

By analyzing the curves corresponding to the average (over 5 runs) return 5.3b and episode length 5.3a of DDPG, we observe that the pre-trained DDPG has no advantage compared to the randomly initialized DDPG; regarding both metrics, the asymptotic performance as well as the learning speed are similar for the case where the weights are transferred and for the case where the weights are randomly initialized. Moreover, regarding the episode return, the initial performance is slightly worse for the pre-trained DDPG (-0.78 decrease). Furthermore, both cases show a marginally worse performance regarding the return values compared to PPO and SAC. By rendering the target task during training episodes, we observe that the best learnt policies of both the pre-trained as well as the randomly initialized DDPG are very far from achieving the hovering task.

By analyzing the curves corresponding to the average (over 5 runs) return 5.4b and

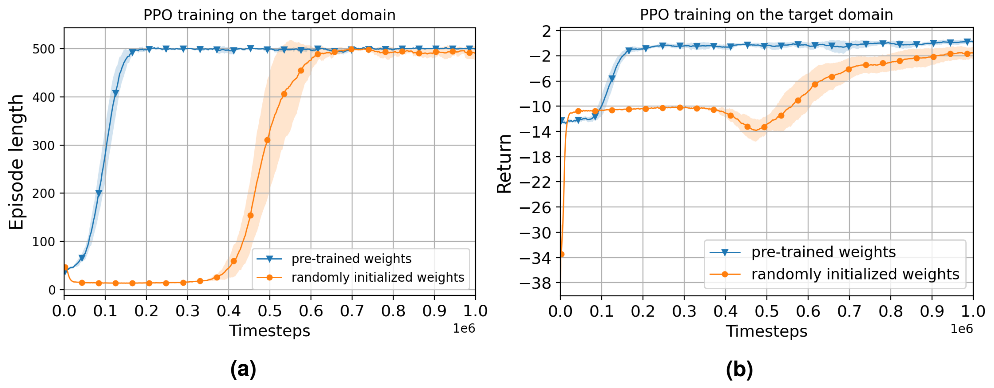


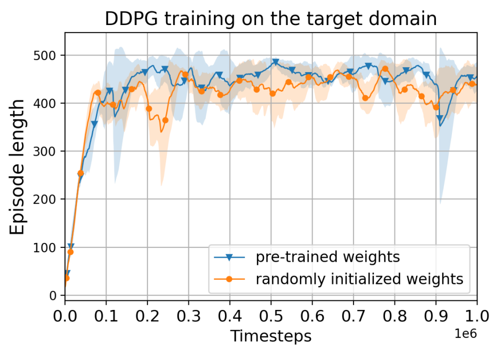
Figure 5.2: Comparison of the episode length 5.2a and return 5.2b of the training of PPO in the target domain with randomly initialized weights and with pre-trained weights. The lines denote the average over 5 runs of 1 million time-steps and the shaded areas correspond to the standard deviations. The blue curves correspond to the pre-trained PPO and the orange curves correspond to the randomly initialized PPO.

	PPO	SAC	DDPG	TD3
Gain in initial return	$\mathbf{+21.2}$	$\mathbf{+33.7}$	-0.78	$\mathbf{+43}$
Gain in asymptotic return	$\mathbf{+1.7}$	$\mathbf{+1.6}$	0	$\mathbf{+1.8}$

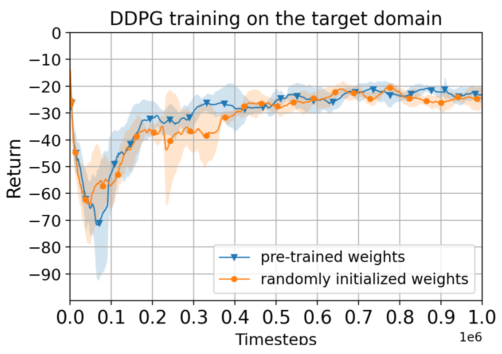
Table 5.1: Summary of results regarding the effect of TL on the learning performance of the algorithms in the target domain, regarding the episode return. A '+' indicates an increase in performance and a '−' indicates a decrease in performance, with respect to the corresponding metric.

episode length 5.4a of TD3, we observe a slight advantage for the case where the weights are pre-trained. The pre-trained TD3 has higher episode length values until approximately 150k time steps and a better initial episode length value ($+60$ increase). Regarding the return, it has higher return values until around 200k time steps. Moreover, it has a better initial return ($+43$ increase) and a slightly better asymptotic return ($+1.8$ increase).

Overall, the transfer of the pre-trained weights of the actor and critic proved to be beneficial to SAC, PPO and TD3. On the other hand, TL has no benefits on the performance of DDPG. In Table. 5.1 and Table. 5.2, we summarize the effects of TL on the considered algorithms regarding the episode return and episode length, respectively. Based on these findings, we proceed to analyze the sample complexity of the post-trained models of SAC, PPO and TD3.



(a)



(b)

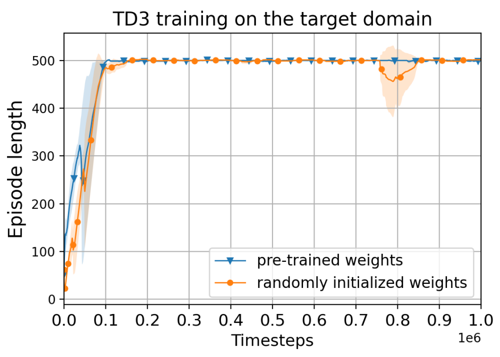
Figure 5.3: Comparison of the episode length 5.3a and return 5.3b of the training of DDPG in the target domain with randomly initialized weights and with pre-trained weights. The lines denote the average over 5 runs of 1 million time-steps and the shaded areas correspond to the standard deviations. The blue curves correspond to the pre-trained DDPG and the orange curves correspond to the randomly initialized DDPG.

	PPO	SAC	DDPG	TD3
Gain in initial episode length	0	0	0	$\mathbf{+60}$
Gain in asymptotic episode length	$\mathbf{+10}$	0	0	0
Gain in time to reach max episode length	$\mathbf{+500}$k	0	0	0

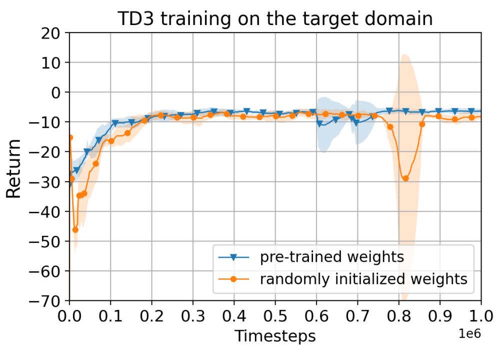
Table 5.2: Summary of results regarding the effect of TL on the learning performance of the algorithms in the target domain, regarding the episode length. A '+' indicates an increase in performance and a '$-$' indicates a decrease in performance, with respect to the corresponding metric.

5.2 Sample complexity analysis

In order to analyze the sample complexity of PPO,SAC and TD3 in the target task, we set goal performances regarding the episode length and episode return, that correspond to a successful achievement of this task.
We choose the goal value for the episode length to be 500, which corresponds to the maximum episode length in the target task. Reaching the maximum episode length means the drone can accomplish the episode without violating any constraints. If this analysis is to be extended to the real world, reaching the maximum episode length is crucial as real-world applications highly need to adhere to safety constraints. In the considered target domain, which is the Pybullet simulation environment, safety constraints are incorporated in speed constraints.
In order to choose a goal return value, we render the Pybullet simulation environment during evaluation episodes of the considered algorithms and observe the performance of the drone. Based on these observations, we empirically set the goal return value to



(a)



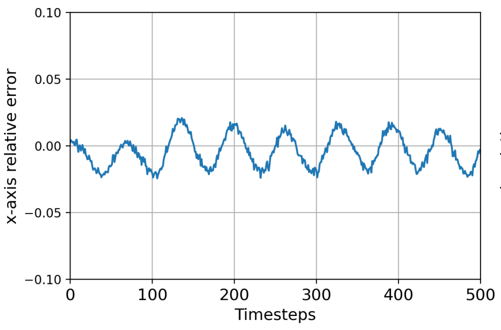
(b)

Figure 5.4: Comparison of the episode length 5.4a and return 5.4b of the training of TD3 in the target domain with randomly initialized weights and with pre-trained weights. The lines denote the average over 5 runs of 1 million time-steps and the shaded areas correspond to the standard deviations. The blue curves correspond to the pre-trained TD3 and the orange curves correspond to the randomly initialized TD3.

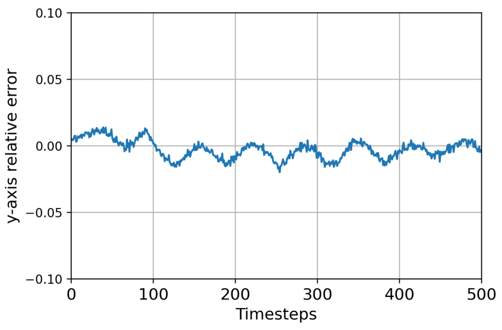
-2.

For an evaluation episode of SAC, that has a length value of 500 and a return value of -2, we plot the relative errors between the target position and the current position of the drone, for the x, y and z-axis, as a function of time. We do that to give insight to what the goal values of the metrics represent in relation to the position of the drone. Fig. 5.5 shows the corresponding results for each of the axes. It is to be seen that for every axis, the magnitude of the relative error is in the range of 0.02, which is equivalent to a 2% error.

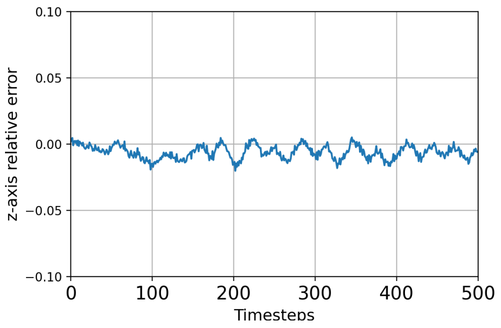
For the case where the policy is stochastic (PPO/SAC), empirical results [8] show that when evaluating an RL agent on a certain environment, it is better to use a deterministic policy; instead of sampling an action from the action probability distribution, the agent chooses the mean action of the probability distribution [8]. In the aim of having better evaluation results, we try this practice for PPO and SAC. To investigate how beneficial it could be, we do the following experiment for each of these algorithms:
We make a post-training run of 1 million time steps on the target domain, while performing 30 evaluation episodes every 100K time steps using both a deterministic variant as well as a stochastic variant of the algorithm. We report the results in Fig. 5.6.
As can be seen from Fig. 5.6a, for SAC, the deterministic version scores better mean return values for all the considered timesteps. According to Fig. 5.6b, for PPO, the deterministic version scores significantly better mean return values for the first time steps (at 100K timesteps). For the subsequent time steps, no version has a clear advantage over the other.
According to the results, this practice seems to be a beneficial one in our use case. Therefore, we use it for both PPO and SAC.



(a)



(b)

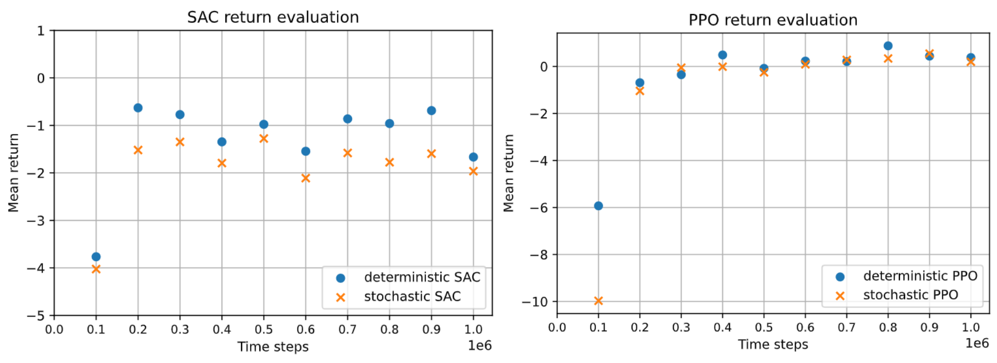


(c)

Figure 5.5: For an evaluation episode of SAC, that has a length value of 500 and a return value of -2: the relative errors between the target position and the current position of the drone, for the x, y and z-axis, as a function of time.

For SAC, TD3 and PPO , we consider the same 5 post-training instances that were realized in the first experiment. As a reminder: parallel to post-training, each instance of these algorithms performs 5 evaluation episodes every 10k time steps.

We first begin with analyzing the sample complexity of the algorithms with respect to the episode length. Fig. 5.7 shows the evaluation results of the algorithms regarding the episode length. From Fig. 5.7, we observe that the learning speed of both SAC and TD3 regarding the episode length is higher than PPO at the beginning of training: On evaluation, SAC and TD3 reach the maximum episode length at 60k and 70k time steps, respectively, and this with a zero standard deviation. PPO is only able to do the same later at 120k time steps. It is to be seen that all the three algorithms exhibit performance drops regarding the episode length; after having already reached the goal episode length (500), the episode length value drops below 500 at subsequent steps. Sudden performance drops affect the reliability of the concerned algorithms, because even if these algorithms reach the goal performance regarding the episode length, they will always be prone to diverge from it at the subsequent evaluation steps.



(a) (b)

Figure 5.6: Parallel to post-training, the SAC and PPO instances perform 30 evaluation episodes every 100k time steps, for both their deterministic as well as their stochastic versions.

	Number of performance drops	Average episode length over these drops
PPO	7	479
SAC	11	473
TD3	13	469

Table 5.3: On evaluation episodes: the number of times the episode length drops below the goal episode length value (500) and the average episode length over these drops.

In order to quantify the magnitude of these drops and their frequency of occurrence for each algorithm, we do the following:
For each algorithm, we count the number of times the episode length drops below 500 and we compute the average over the episode length values at these drops. Table. 5.3 shows the corresponding results.

According to the results, PPO has the smallest number of performance drops and the highest average episode length value over these drops. SAC and TD3 score worse results both regarding the number of performance drops as well as the average episode length over these drops, with TD3 suffering from approximately double the number of performance drops of PPO. We conclude that on evaluation episodes, PPO reaches the goal episode length value in around double the time SAC and TD3 need (60k and 70k for SAC and TD3, respectively, and 120k for PPO), but PPO is more stable than SAC and TD3 staying at this value.

Next, we analyze the sample complexity of the algorithms with respect to the episode return. Fig. 5.8 shows the evaluation results of the algorithms regarding the episode return.
It is to be seen that TD3 never reaches the goal reward: its best reward is at around -6. Therefore, we consider that TD3 fails to solve the target task.
PPO reaches the goal return (-2) at 110k time steps. At 120k time steps, PPO reaches

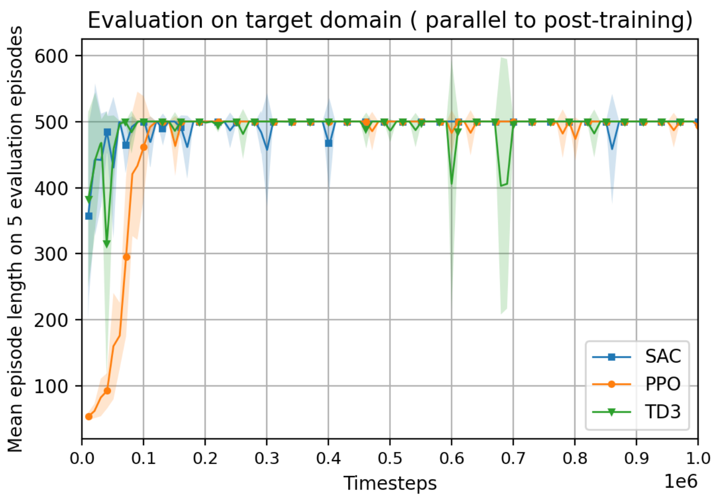


Figure 5.7: Comparison of the episode length of SAC, TD3 and PPO during evaluation. Parallel to post-training, each of the 5 instances of these algorithms performs 5 evaluation episodes every 10k time steps. The solid lines denote the average over 5 runs and the shaded areas correspond to the standard deviations. The blue, orange and green curves correspond to SAC, PPO and TD3, respectively.

both the goal return as well as the goal episode length value. Therefore, we consider that on average (over 5 post-training runs), the first time PPO successfully solves the target task is after 120k time steps of post-training. We also observe that in the subsequent time steps of 120k, PPO does not score below the goal return.
SAC reaches the goal return (-2) at 120K time steps. At that time step, it also scores the goal episode length value (500). Therefore, we consider that on average (over 5 post-training runs), the first time SAC successfully solves the target task is after 120k time steps of post-training. In the subsequent time steps of 120k, the return value drops 14 times below the goal return value. The average return over these drops is -5.5. Some of these drops, e.g., the ones that happen at 170k and 400k timesteps, have a significantly higher magnitude (-19 and -14, respectively) compared to the other drops. Moreover, such drops correlate with the performance drops of the episode length curve in Fig. 5.7, which were discussed earlier. Such huge drops mean SAC fails dramatically to solve the task but only for a while; it needs at max 20k time steps to restore the return levels that were reached before the drops.
We also observe that the asymptotic performance of PPO is better than that of SAC. The asymptotic reward reached by SAC is at around -1, whereas for PPO it is around 1. This means both PPO and SAC fulfill the minimum return required to solve the target

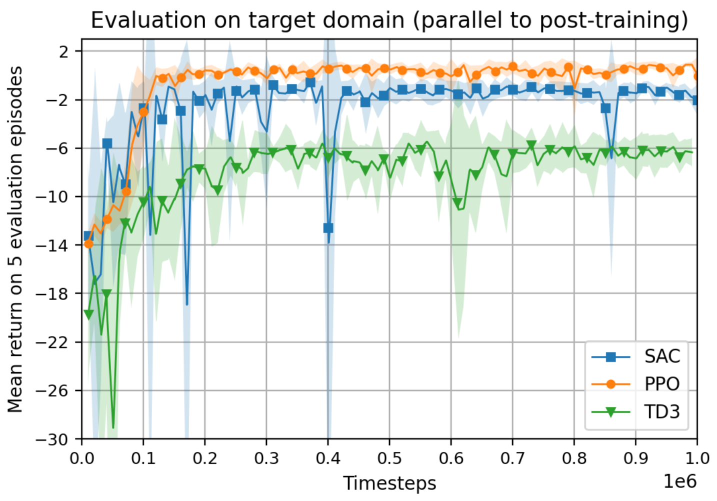


Figure 5.8: Comparison of the episode return of SAC, TD3 and PPO during evaluation. Parallel to post-training, each of the 5 instances of these algorithms performs 5 evaluation episodes every 10k time steps. The solid lines denote the average over 5 runs and the shaded areas correspond to the standard deviations. The blue, orange and green curves correspond to SAC, PPO and TD3, respectively.

	Does it learn to solve the target task?	Post-training steps needed
PPO	yes	120k
SAC	yes	120k
TD3	no	–

Table 5.4: Summary of results regarding the sample complexity of PPO, SAC and TD3 on the target task.

task, but PPO solves the task comparably better.
In Table. 5.4, we summarize the the results regarding the sample complexity of PPO, SAC and TD3 on the target task.

6 Conclusion

This thesis aimed to analyze the sample complexity of various deep RL algorithms applied to control a quadrotor to achieve a hovering task on Pybullet simulation environment. The considered algorithms covered on-policy, off-policy, deterministic and stochastic algorithms. In order to speed up the convergence of such algorithms, a TL approach was adopted: before being trained on the Pybullet simulation environment, the considered algorithms are pre-trained to achieve the hovering task on another simulation, which is based on simpler transition dynamics, namely the differential equations of the drone. In the considered TL approach, the weights of the actor and/or critic of the algorithms constituted the transferrable knowledge.
Over the course of the thesis, firstly the benefits of TL to each of the considered algorithms were investigated by means of commonly used metrics. It was shown that TL was beneficial to the majority of the algorithms: it benefitted SAC PPO and TD3 but had a rather neutral effect on DDPG. For SAC, PPO and TD3 it improved the initial performance, the asymptotic performance and the learning speed.
Subsequently, the sample complexity of the algorithms to which TL proved to be beneficial, namely SAC, PPO and TD3, was to be analyzed. For that matter, a minimum required goal performance, starting from which the hovering task is considered successfully achieved, was empirically chosen: for an evaluation episode, the goal performance corresponds to the drone concurrently scoring the maximum episode length (500) and an episode return value of -2. This return value means that during an evaluation episode, for the x, y and z positions of the drone, the magnitude of the relative error between the current and the desired position of the drone is in the range of 2%.
Based on empirical results, it was shown that for our specific use case, deterministic variants of SAC and PPO proved to yield better performances than the stochastic variants on evaluation episodes. Therefore, evaluation results of both PPO and SAC were based on their deterministic variants.
Based on evaluation episodes conducted concurrently to the post-training of PPO, SAC and TD3 on Pybullet simulation environment, we observed that on average (over 5 post-training runs), the first time PPO and SAC successfully solve the hovering task on Pybullet is after 120k post-training steps whereas TD3 fails to solve the task over the course of 1 million post-training steps. Moreover, we observed that over the course of these evaluation episodes, PPO has a more stable behavior than SAC: regarding the episode length and the episode return, SAC exhibits more performance drops with a significantly higher magnitude. Furthermore, we observed that PPO has a better asymptotic performance than SAC over the course of these evaluation episodes. From

all these observations, we conclude that among the considered algorithms that benefit from TL, PPO is the most reliable one to learn the hovering task on Pybullet simulation environment.

List of Tables

List of Figures

Bibliography

[1] O. M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, et al. “Learning dexterous in-hand manipulation”. In: *The International Journal of Robotics Research* 39(1) (2020), pp. 3–20.

[2] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. “A brief survey of deep reinforcement learning”. In: *arXiv preprint arXiv:1708.05866* (2017).

[3] S. Bozinovski. “Reminder of the first paper on transfer learning in neural networks, 1976”. In: *Informatica* 44(3) (2020).

[4] J. Christopher. “Watkins and peter dayan”. In: *Q-Learning. Machine Learning* 8(3) (1992), pp. 279–292.

[5] E. Coumans and Y. Bai. “Pybullet, a python module for physics simulation for games, robotics and machine learning”. In: (2016).

[6] N. Duong Trung, T. Ngoc, and H. Huynh. “Automated Pneumonia Detection in X-Ray Images via Depthwise Separable Convolution Based Learning”. In: June 2019. DOI: `10.15625/vap.2019.0005`.

[7] S. Fujimoto, H. Hoof, and D. Meger. “Addressing function approximation error in actor-critic methods”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 1587–1596.

[8] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

[9] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. “Deep Reinforcement Learning that Matters”. In: *CoRR* abs/1709.06560 (2017). arXiv: `1709.06560`. URL: `http://arxiv.org/abs/1709.06560`.

[10] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver. “Rainbow: Combining Improvements in Deep Reinforcement Learning”. In: *CoRR* abs/1710.02298 (2017). arXiv: `1710.02298`. URL: `http://arxiv.org/abs/1710.02298`.

[11] P. N. Kolm and G. Ritter. “Modern perspectives on reinforcement learning in finance”. In: *Modern Perspectives on Reinforcement Learning in Finance (September 6, 2019). The Journal of Machine Learning in Finance* 1(1) (2020).

[12] A. Lazaric. “Transfer in reinforcement learning: a framework and a survey”. In: *Reinforcement Learning*. Springer, 2012, pp. 143–173.

[13] Y. LeCun, Y. Bengio, and G. Hinton. “Deep learning”. In: *nature* 521(7553) (2015), pp. 436–444.

[14] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. “Continuous control with deep reinforcement learning”. In: *arXiv preprint arXiv:1509.02971* (2015).

[15] C. Luis and J. L. Ny. *Design of a Trajectory Tracking Controller for a Nanoquadcopter*. 2016. arXiv: `1608.05786 [cs.SY]`.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. “Human-level control through deep reinforcement learning”. In: *nature* 518(7540) (2015), pp. 529–533.

[18] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. “Trust region policy optimization”. In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897.

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

[20] D. C. Sorensen. “Newton’s method with a model trust region modification”. In: *SIAM Journal on Numerical Analysis* 19(2) (1982), pp. 409–426.

[21] R. S. Sutton, A. G. Barto, et al. “Reinforcement learning”. In: *Journal of Cognitive Neuroscience* 11(1) (1999), pp. 126–134.

[22] M. E. Taylor and P. Stone. “Cross-domain transfer for reinforcement learning”. In: *Proceedings of the 24th international conference on Machine learning*. 2007, pp. 879–886.

[23] M. E. Taylor and P. Stone. “Transfer learning for reinforcement learning domains: A survey.” In: *Journal of Machine Learning Research* 10(7) (2009).

[24] M. Van Otterlo and M. Wiering. “Reinforcement learning and markov decision processes”. In: *Reinforcement learning*. Springer, 2012, pp. 3–42.

[25] W. Y. Wang, J. Li, and X. He. “Deep reinforcement learning for NLP”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*. 2018, pp. 19–21.

[26] Y. Wang, K. Velswamy, and B. Huang. “A Long-Short Term Memory Recurrent Neural Network Based Reinforcement Learning Controller for Office Heating Ventilation and Air Conditioning Systems”. In: *Processes* 5 (Sept. 2017). DOI: `10.3390/pr5030046`.

[27] Y. Yu. “Towards Sample Efficient Reinforcement Learning.” In: *IJCAI*. 2018, pp. 5739–5743.

[28] H. Zhang and T. Yu. “AlphaZero”. In: *Deep Reinforcement Learning*. Springer, 2020, pp. 391–415.

[29] Z. Zhu, K. Lin, and J. Zhou. “Transfer learning in deep reinforcement learning: A survey”. In: *arXiv preprint arXiv:2009.07888* (2020).