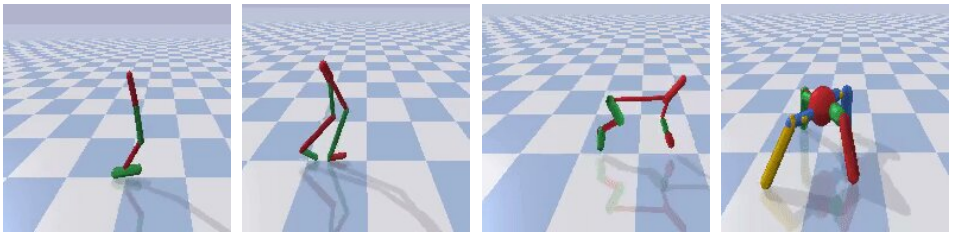
Evaluating and Improving Robustness in Reinforcement Learning using Self-Supervised Representation Learning

Master of Science Thesis

Khushdeep Singh Mann



eit Digital MASTER SCHOOL

berlin

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Evaluating and Improving Robustness in Reinforcement Learning using Self-Supervised Representation Learning

by

Khushdeep Singh Mann

A thesis submitted in partial fulfillment for the
degree of Master of Science (M.Sc.)

in

ICT Innovation - Autonomous Systems

First advisor - Prof. Dr. Matthias Bethge (University of Tübingen)
Second advisor - Prof. Dr. Klaus Robert Müller (Technische Universität Berlin)

June 2021

Declaration of Authorship

I hereby declare that the work conducted during the course of this thesis was in correspondence with the regulations of Technische Universität Berlin. This thesis is the result of my original work and the used external sources have been indicated and acknowledged as references wherever applicable. This thesis has not been submitted to any other academic institution for any degree or qualification. I, hereby, acknowledge and abide by the academic rules and regulations for undertaking post graduate studies at Technische Universität Berlin.

Name of Student: **Khushdeep Singh Beant Singh Mann**

Degree course: **ICT Innovation (Autonomous Systems)**

Faculty: **Fakultät IV Elektrotechnik und Informatik Technische Universität Berlin**

Khushdeep Singh Mann

June 2021

Abstract

Dynamically changing constraints in robotics demand the ability to learn, adapt, and reproduce tasks. The robotic workspace is sometimes unpredictable and high dimensional, limiting the scalability of supervised and Reinforcement Learning (RL). In order to tackle these constraints, we undertake Self-Supervised Learning (SSL) approach for inferring and analyzing the internal dynamics within model-free reinforcement learning algorithms.

The thesis investigates the behavior of RL agents under morphological distribution shifts. We train the policies for different RL agents and test transfer the learnt models over several perturbed environments. The perturbed environments being generated by changing the length and mass of agent limbs. Later, we compare the performance of RL policies with and without integrated SSL representations, allowing the agents to adapt across the environments with perturbed parameters. We find that out-of-distribution performance of self-supervised models is correlated to degradation in agent reward. This work has been accepted at the 'Self-Supervision for Reinforcement Learning Workshop - ICLR 2021' and the short paper is available here.

Abstract

Robotische Systeme müssen in der Lage sein, sich an Änderungen ihrer Dynamik und wechselnden Umweltbedingungen flexibel anzupassen. Diese Änderungen sind manchmal schwer vorhersagbar und limitieren die Skalierung von Methoden des verstärkenden Lernens (Reinforcement Learning, RL) in der realen Welt. In dieser Arbeit betrachten und analysieren wir Methoden des selbstüberwachten Lernens (Self-Supervised Learning, SSL), um RL Modelle an ändernde Dynamiken anzupassen.

Wir betrachten dabei Veränderungen des Körperplan verschiedener RL Agenten und untersuchen, ob mittels Adaptation per SSL eine Verbesserung des erwarteten Gewinns erreicht werden kann. Während wir eine Korrelation zwischen der Verringerung des Gewinns und des erwarteten Fehlers eines SSL Modells unter einer Dynamikänderung feststellen, finden wir noch keinen ausreichenden Beleg für einen kausalen Zusammenhang. Teil dieser Arbeit erschien bereits als kurzer Beitrag auf dem Self-Supervision for Reinforcement Learning Workshop der ICLR2021 link.

Acknowledgements

At first, I would like to thank my supervisor - PhD candidate Steffen Schneider and Prof. Matthias Bethge from the University of Tübingen for giving me an opportunity to work on this thesis and for the constant support. Steffen always provided feedback on my progress and appreciated my efforts. He was always available to discuss ideas and help me with the codebase.

Next, I would like to thank Prof. Georg Martius from Autonomous Learning group at Max Planck Institute for Intelligent Systems for providing valuable feedback during the mid-presentation of my thesis. I highly appreciate the efforts made by Jin Hwa Lee from Technische Universität Munich (TUM) for fine tuning the wav2vec model and generating trained models for various reinforcement learning agents.

I also thank Prof. Dr. Klaus Robert Müller and Dr. Grégoire Montavon from Technische Universität Berlin (TUB) for administrative supervision of this thesis.

I would like to thank EIT Digital Master School for providing me an opportunity to study ICT Innovation in Autonomous Systems at two European universities: KTH Royal Institute of Technology in Sweden and Technische Universität Berlin in Germany.

At last, I thank my parents and my sister for their support throughout my life.

Technische Universität Berlin — Khushdeep Singh Mann

June 2021

Contents

List of Figures

List of Tables

I dedicate this work to my parents and my sister.

Chapter 1

Introduction

Humans and animals have the ability to adapt to the changes in their environments. Their behavior is highly adaptive when being deployed on different terrain, grabbing different objects, changes to their bodies during the injury or during the growth process. This kind of adaptive behavior is necessary for functioning in the real world.

On the other hand, robots and artificial agents are less versatile and are typically deployed with fixed behaviors. This leads to robots and agents succeeding in specific tasks while failing in others such as changes in the environment, experiencing system malfunctioning, or encountering a difficult terrain. While a potential research direction to learn adaptive control policies for robots by imitating animals have been discussed in [9, 10].

1.1 Research goal

The goal of this thesis is to investigate how self-supervised learning can improve the robustness of state-of-art reinforcement learning (RL) policies for continuous control agents subjects to morphological changes (perturbations) in their physical structure. Hence, the following research questions are addressed:

- How robust are current state-of-art RL policies against perturbations in an agent's physical structure?
- Can representations learnt through self-supervised models improve the robustness of RL policies tested on perturbed environments?

1.2 Approach

The first part of the thesis includes changing the morphological structure of continuous control RL agents. Next, we evaluate the robustness of state-of-art RL policies over these different agent structures in order to investigate how sensitive are these policies subject to the changes in the physical model. Then, we filter over all the collected state-space data to train the Self-Supervised Learning (SSL) model to extract useful representations.

The second part involves fine-tuning the SSL model and integrating the learnt state-space representations with state-of-art RL policies. Training and re-evaluating the robustness over agent's morphology. Later, providing experimental evidence for a correlation between the generalization of the agent's internal model and the obtained rewards. Finally, providing the conclusion and directions for the future work.

1.3 Contribution

The main contributions of this thesis are:

- Development of an new scheme for investigating and benchmarking the performance of continuous control RL agents subject to morphological changes.
- Exploring the adaptiveness and robustness of state-of-art RL policies with integrated SSL models across morphological domain shifts in agent's physical structure.

1.4 Thesis Structure

The remaining thesis is structured as follows:

Chapter 2 presents the related work in representation learning and self-supervised learning in context of reinforcement learning.

Chapter 3 discusses an formal introduction to reinforcement learning by providing relevant mathematical background. The chapter also discusses traditional RL approaches, model-free and model-based methods.

Chapter 4 introduces the basics of deep neural networks along with their architecture and some training techniques.

Chapter 5 focuses on Deep Reinforcement Learning (DRL), and provides details on the DRL algorithms considered for the scope of this thesis. In the later part, this chapter introduces the physics engine and elaborates on continuous control agents included in the experimental setup of this thesis.

Chapter 6 presents the self-supervised learning model, its architecture, mathematical background and model parameters used for extracting representation from RL data during the thesis.

Chapter 7 describes the various steps involved in the experimental design process.

Chapter 8 evaluates the performance for different experimental steps and discusses the results.

Chapter 9 presents the conclusion of the undertaken work in the thesis and provides directions for future work.

Chapter 2

Related Work

2.1 Robustness in Reinforcement Learning

Robustness and generalization are the key challenges in reinforcement learning especially when applied to robotic applications, wherein a policy trained in simulation is transferred and should be adaptive to different real-world conditions. As the standard reinforcement learning policies are developed for learning on environments in simulation, these policies are inefficient once transferred to real system as analyzed in [11].

Various attempts have been made by proposing different techniques towards gaining robustness in reinforcement learning policies. Meta-learning has been proven successful for adapting to continuous control tasks on simulated as well as real-world agents [12]. This framework demonstrated an agent's ability to quickly adapt to a missing leg, adjustments on slopes and novel terrains, and compensating for pulling payloads. Yet another framework for accessing generalization in RL by changing parameters such as power, density, friction for continuous control agents have been proposed in [13]. The authors introduced an new algorithm in [14] to learn robust policies for domain adaptation using an ensemble of simulated source domains.

Moreover, a recent framework to incorporate robustness to perturbations into continuous control reinforcement learning algorithms is implemented in [15]. A study of attaining generalization via exploring the morphological structure of the agent has been demonstrated in [16]. Additionally, authors in [17] investigated the existence of a single global policy to have generalization across wide range of agent morphologies. [18] presents a new method for learning the parametric controller for changes in morphological structure of physics-based characters in simulation.

Improving the robustness of RL algorithms has also being considered through data augmentation [19, 20]. The authors in [19] enable robust learning from pixels using an data augmentation scheme that can be applied to model-free RL algorithms.

2.2 Representations in Reinforcement Learning

A new approach to teach robots and artificial agents about their environments is through representation learning. Extracting representations using unsupervised learning approach with contrastive predictive coding is presented in [21]. [22] illustrates the theoretical analysis of contrastive unsupervised representation learning.

A representation learning framework from raw pixels and extracting high-level features [23] show promising results for RL. Yet another approach to learn task-relevant robust latent representations is observed in [24]. The authors study how the learnt invariant representations can accelerate reinforcement learning. In regards to scalable reinforcement learning from pixels, [25] presents learning behaviors by latent imagination. Representation learning have also been used for learning object attributes from unlabelled videos [26] so as to generalize to unseen environments.

2.3 Self-Supervised Learning in Reinforcement Learning

Self-supervised learning is an powerful tool to learn representations from unlabelled data sets. An approach to attain generalization in reinforcement learning by exploring the use of self supervision is [27]. Here, the the policy is allowed to train even after deployment using self-supervision and this approach improves generalization across several tasks. The work in [28] investigates the task-agnostic representations for continuous control tasks and show that such representations enable the learning of continuous control policies.

Complex tasks can be tackled through RL, but the learning tends to be task-specific. The researchers in [29] undertake a new approach using self-supervised exploration that leads to fast adaptation to new tasks. Self-supervised learning have also being deployed for unsupervised domain adaptation [30] having unlabelled data on target domain, the representations of the source and target domains were aligned by performing self-supervised tasks on both domains simultaneously.

Self-supervised approach for learning representations of motion attributes is proposed in [31]. This approach accounts for learning behaviors from unlabelled video recordings

and study how this can be useful in robotic imitation settings. [32] focuses on acquiring object-centric representations for self-supervised robotic grasping tasks with an idea that these representations can be refined continuously as the robot collects more experience by interacting with the environment.

Chapter 3

Reinforcement learning Problem

3.1 Introduction

Reinforcement Learning (RL) is a branch of machine learning that deals with making a sequence of decisions. The RL problem consists of a decision maker called agent. Everything else outside the agent is called environment. The agent interacts with the environment and in return receives new state and a scalar reward. The obtained reward signal indicates how good or bad the agent's action was at that particular instant. The goal of an RL agent is to maximize the cumulative sum of rewards. In order to achieve this goal, the agent tries to learn behaviors by following a certain policy- a rule to decide what actions to take. Accordingly, a policy maps the agent states to action and the task of an RL agent is to learn an optimal policy. RL is different from supervised and unsupervised learning in a way that RL agent needs to explore the environment and learn based on trial and error. The formalization of RL problem is to optimize a Markov Decision Process (MDP). This chapter further provides formal definitions of RL components including Policy, Value function and Bellman equations. The chapter also elaborates traditional RL methods, Model-based and Model-free RL. The chapter is based on Reinforcement Learning by Sutton and Barto [2], lecture series by David Silver [33], educational resource on deep reinforcement learning produced by OpenAI [34] and a master thesis [35] . The interaction loop between agent and environment is depicted in (3.1).

3.2 Markov decision process

A Markov Decision Process (MDP) is defined by the tuple $M = (S, A, P, R, \gamma)$, where the symbols denote:

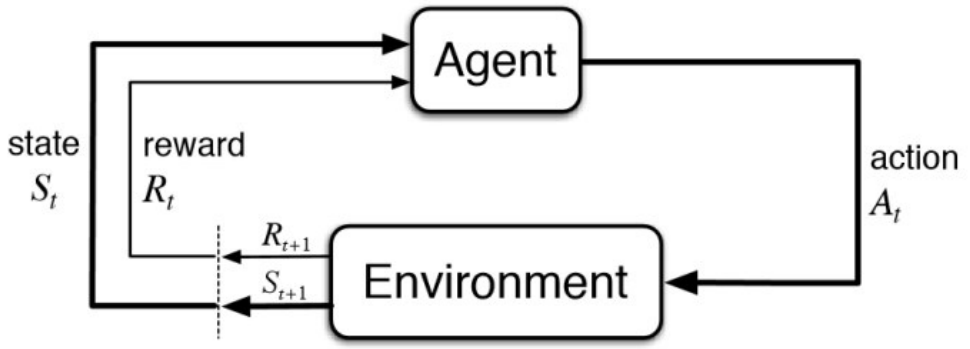


FIGURE 3.1: The fundamental interaction between an RL agent and environment. The agent interacts with action $\boldsymbol{a_t}$ and in return obtains next state $\boldsymbol{s_{t+1}}$ along with reward $\boldsymbol{r_t}$:= r($\boldsymbol{s_t}$, $\boldsymbol{a_t}$). This process occurs at each time step.

- A set of states, $s \in S$, where S is state space
- A set of actions, $a \in A$, where A is action space
- Transitional probability distribution P : S × A × S → [0, 1]

  $$P(\text{s'}|\ \text{s},\ \text{a}) = Pr(s_{t+1} = \text{s'}\ |\ s_t = \text{s},\ a_t = \text{a})$$

  which symbolizes that given a state s_t and action $\boldsymbol{a_t}$, how likely the agent ends up in state $\boldsymbol{s_{t+1}}$.
- Reward function R : S × A × S → $\mathbb{R}$

  $$R(\text{s},\ \text{a},\ \text{s'}) = \mathbb{E}(\boldsymbol{r_t}\ |\ \boldsymbol{s_t} = \text{s},\ \boldsymbol{a_t} = \text{a},\ \boldsymbol{s_{t+1}} = \text{s'})$$

  is the instantaneous reward received after the transition from state s to s' due to action a.
- The discount factor $\boldsymbol{\gamma}$, which is used to generate discounted reward.

The most vital aspects of an MDP are the transitional probabilities and reward function. In general, both of these aspects are considered unknown for an reinforcement learning setting.

3.3 Policy

The motive of the MDP is to find a policy that can train an RL agent and maximize the cumulative rewards obtained after taking a series of actions. If the policy only depends on the state, π(s), it is termed as deterministic, else stochastic π(a|s), which outputs a probability distribution over actions, given a state s.

3.4 Return

The goal of an RL agent in an MDP is to maximize its cumulative rewards. The sum of rewards obtained from the environment is termed as *return*, G_t, which is defined at time t as follows:

$$G_t = r_t + r_{t+1} + r_{t+2} + ... + r_T$$

where, T is the final step.

For continuing tasks, the final step T would be infinite. Thus, we define the *return* in terms of discounted rate, γ as:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

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

3.5 Value functions

It is important for an RL agent to determine how good it is to be in an particular state, or to examine what action to take at a particular time instant. An entity that measures the *goodness* of a state or an state-action pair is termed as *value function*. It is expressed in terms of expected return. Now, the reward obtained is dependent on the actions undertaken in a given state. As the agent's actions are based on the policy it follows, the value functions are defined with respect to the policies. Consider an agent following a policy π from a state s. The *value function* denoted by V_π(s) for the policy π is given by

$$V_\pi(\mathrm{s}) = \mathrm{E}_\pi\ (G_t \mid s_t = s) = E_\pi(\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s)$$

Similarly, we define an *action-value function*, Q_π(s,a), that determines how good it is for an agent to take a particular action a from state s following a policy π. This *action-value function* is also known as *Q-function* and is mathematically expressed as

$$Q_\pi(s, a) = \mathbb{E}_\pi(G_t | s_t = s, a_t = a) = \mathbb{E}_\pi(\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, a_t = a) \tag{3.1}$$

3.6 Bellman Equations

Bellman equations form the central element of many RL algorithms. It decomposes the value function into two parts, the immediate reward and the discounted future values. Bellman equations represent the value function as a recursive relationship between the current value of state and its successive states. The equations in this section are based on [36]. Mathematically, the Bellman expectation equation for state-value function is defined as:

$$V_\pi(s) = \mathbb{E}_\pi(r_{t+1} + \gamma V_\pi(s_{t+1})|s_t = s) \tag{3.2}$$

Similarly, the state-action value function is expressed as:

$$Q_\pi(s) = \mathbb{E}_\pi(r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1})|s_t = s, a_t = a) \tag{3.3}$$

While solving an MDP, the RL agent is trying to find an optimal value function. The optimal *value function*, V_*(s) can be expressed as,

$$V_*(\mathrm{s}) = \max_\pi V_\pi(\mathrm{s}),\ \forall\ s \in S$$

Similarly, the optimal *action-value function*, Q_*(s,a) can be expressed as,

$$Q_*(\mathrm{s}) = \max_\pi Q_\pi(\mathrm{s}),\ \forall\ s \in S,\ a \in A$$

As value functions are dependent on policies, the agent is trying to find an optimal policy. An policy π is said to be better than another policy π' if the expected return of that policy is better than π' for all $s \in S$. This means, $V^\pi(\mathrm{s}) \geq V^{\pi\,\prime}(\mathrm{s})$ for all $s \in S$. For an optimal policy, the following equation holds,

$$V_*(s) = \max_{a \in A(s)} Q_\pi*(s, a) \tag{3.4}$$

Expanding equation (3.1) with (3.4) we get,

$$\begin{aligned} V_*(s) &= \max_a \mathbb{E}_\pi*(G_t|s_t = s, a_t = a) \\ &= \max_a \mathbb{E}_\pi*(\sum_{k=0}^{\infty} \gamma^k r_{t+k}|s_t = s, a_t = a) \\ &= \max_a \sum_{s'} p(s'|s, a)[R(s, a, s') + \gamma V_*(s')] \end{aligned} \tag{3.5}$$

Equation (3.5) is known as Bellman optimality equation for state-value function, V_*(s'). Similarly, the Bellman optimality equation for action-value function, Q_* is given by,

$$\begin{aligned} Q_*(s,a) &= \mathbb{E}_\pi(r_t + \gamma \max_{a'} Q_*(s_{t+1}, a_\prime)|s_t = s, a_t = a) \\ &= \sum_{s'} p(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q_*(s',a')] \end{aligned} \tag{3.6}$$

An important aspect of RL algorithms is whether the agent can learn a model of its environment. This means learning a function that predicts the state transition probabilities and rewards. If learning such a function is feasible, the algorithm can solve the Bellman optimality equations in an iterative process. Such methods are termed as *model-based* algorithms, discussed in section (3.7).

In case, the transition probabilities are not known, the algorithm estimates the value function and policy by performing rollouts on the system. Such methods are termed as *model-free* algorithms, discussed in section (3.8).

3.7 Model Free methods

In *model-free* methods the RL agent learns directly from the interactions with the environment, but does not learn a model. Model-free methods are either value-based or policy-based. The value-based methods perform iterative updates of the perceived state to eventually learn an optimal policy. While, policy-based methods directly update the policy without storing the states [37].

Model-free algorithms can also be categorized into *off-policy* or *on-policy*. Off-policy algorithms evaluate and update a policy which is different from policy being used to generate actions. While On-policy algorithms evaluate and update the same policy that is being used to generate actions [38]. The following illustrates more on value-based methods and policy-based methods.

3.7.1 Value Based Methods

As value-functions are state-action pair functions estimating the value of a specific action in a particular state, or the return for that action. Considering V_π(s) - the value of a state s under a policy π and Q_π(s,a) - the value of taking action a in state s under the policy π, the challenge is estimating these value functions for a particular policy. Estimating these value would result in accurately selecting an action that would provide the best possible total reward [39].

3.7.1.1 Monte Carlo Methods

Monte Carlo methods rely on sampling states, actions and rewards from a given environment. This eliminates the need of knowing the state transitional probability distributions or having a model of the environment. Monte Carlo methods follow Generalized Policy Iteration (GPI) framework which is an two step process. The first step *policy evaluation* attempts to have an approximation of value function based on the current policy. The second step *policy improvement* improves the policy with respect to the current value function. The pictorial representation is visible in figure (3.2).

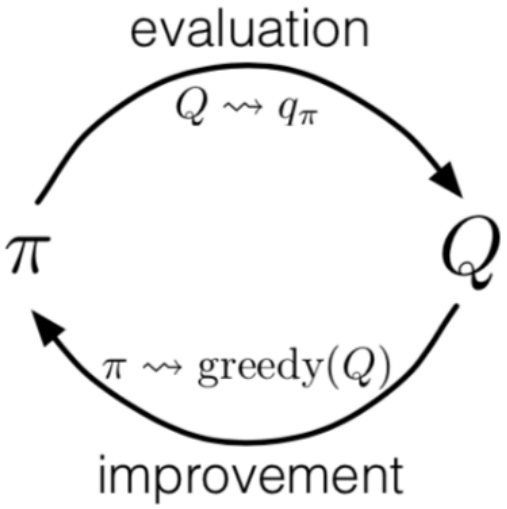


FIGURE 3.2: Two steps in Generalized Policy Iteration (GPI) framework [1]

Using this two step process the Monte Carlo algorithm converges to the optimal policy and value function but need sufficient exploration and large number of iterations.

3.7.1.2 Temporal Difference Learning

Temporal difference (TD) learning methods can be used to estimate these value functions. If estimation technique is not used, the agent needs to wait until the final reward has been received so as to update the state-action pair values. Instead, the TD methods calculate the temporal error- the difference between new and old estimate of the value function. Thus, final reward is estimated at each state and state-action pair is updated. This approach reduces the variance but increases bias in the value-function estimation. The update equation is formally expressed as:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma\ V(s_{t+1})\ -\ V(s_t)\ ]$$

where, s_t is the state at time t, r_{t+1} is the reward at time t+1 and α is the learning rate. As the value is updated partially using an estimate technique, TD method is

termed as "bootstrapping" method. Two widely used TD algorithms are *Q-Learning* and *SARSA (State-Action-Reward-State-Action)*. Q-learning is an off-policy TD algorithm introduced by Chris Watkins [40]. The actions are chosen based on ϵ-greedy policy, and the algorithm converges close to the approximation of action-value function. Q-learning learns an optimal policy irrespective if the actions where chosen from an exploratory or any random policy. The update equation is given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \max_a \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{3.7}$$

The Q-learning algorithm is summarized as:

Algorithm 1: Q-learning (off-policy)

```
Initialize Q(s,a) arbitrarily;
repeat
    Initialize s;
    for each step of episode do
        Select action a from state s using policy derived from Q (e.g, ε-greedy);
        Perform action a, observe reward r and next-state s';
        Update Q using equation (3.7)
    end for
until terminated;
```

SARSA is an on-policy TD algorithm. It is different from Q-learning in a way that it learns an action-value function instead of value function. The maximum reward for the next state may not be used for updating the Q-values, instead, a new action is selected based on the same policy that determined the current action. The algorithm is

summarized as:

Algorithm 2: SARSA (on-policy)

Initialize Q(s,a) arbitrarily; **repeat**
  Initialize s;
  Select action a from state s using policy derived from Q (e.g, ϵ-greedy);
  for *each step of episode* **do**
    Perform action a, observe reward r and next-state s';
    Select next-action a' using policy derived from Q (e.g, ϵ-greedy);
    Update Q using

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{3.8}$$

  end for
until *terminated*;

Monte Carlo methods have a lot of variance but are not biased. While TD methods are biased while having reduced variance. An TD(λ) is a generic reinforcement learning method that unifies Monte Carlo and TD methods. TD(λ) algorithm considers the returns after every certain number of steps and takes an weighted average of those returns.

In TD(λ) algorithm an additional parameter called *eligibility trace* e_t(s) is associated with each state. The eligibility trace is updated for each state as per the following equation:

$$e_t(\mathrm{s}) = \begin{cases} \text{if } s \neq \mathrm{s}_t \\ \text{if } s = s_t \end{cases}$$

where λ is the *trace-decay* parameter.

The *eligibility trace* combines two things - how *frequent* and how *recent* a state is. It is used as a scaling factor for the TD error [41].

3.7.1.3 Function Approximators

A technique for estimating an unknown function based on historical information or available observations is termed as Function Approximation. As the state space dimensionality increases it becomes impossible to keep track of all information. So, representing

the value function in the form of look up table is not possible. Function approximators can be used to estimate the value functions based on environmental observations.

Consider a value function $V(s;\theta)$, where θ is the parameter vector $\theta = (\theta_1, \theta_1, ..., \theta_n)^T$. A way of mapping θ to the space of value function can be accomplished with function approximators. In this way, only the model parameters are needed to be stored and function approximators can better generalize the training samples.

Many types of function approximators are found in literature. Like linear and radial basis functions, or recently the neural networks. Currently neural networks are the most widely used function approximators due to their ability of representing complex value functions with less parameters. However, attaining convergence for neural networks is tricky when applied to reinforcement learning [42].

3.7.2 Policy Based Methods

Policy based methods are a type of reinforcement learning techniques that optimize the parameterized policies with respect to the cumulative rewards with the help of gradient descent. For an parameterized policy π_θ, with θ being the parameter vector, these methods optimize the policy by fine tuning in the parameter space $\theta \in \Theta$. The policy parameters are then updated using the gradient descent in the direction of increasing expected return. The equation for updating the policy parameters is given by:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J \tag{3.9}$$

where J is the expected return given by:

$$J = \mathbb{E}_\pi(\sum_{k=0}^{\infty} \gamma^k r_k) \tag{3.10}$$

Properties of policy based methods include more stability (better convergence) and well performance in high dimensional or continuous action spaces. Also these methods can learn stochastic policies as compared to whereas value based approaches [43]. Drawback of this technique is that the convergence may occur on local optimum and not the global optimum. This is due to large variance in the policy evaluation step.

3.7.3 Actor Critic Methods

Actor critic methods are the TD methods that have separate memory allocation for explicitly representing the policy, independent of the value function. *Actor* is the policy

structure used to select actions and *critic* is the estimated value function. The *critic* criticizes the actions undertaken by the *actor*. The *critic* evaluates the policy based on the temporal difference error and then the policy is updated. A functional representation of actor-critic architecture is shown in figure (3.3).

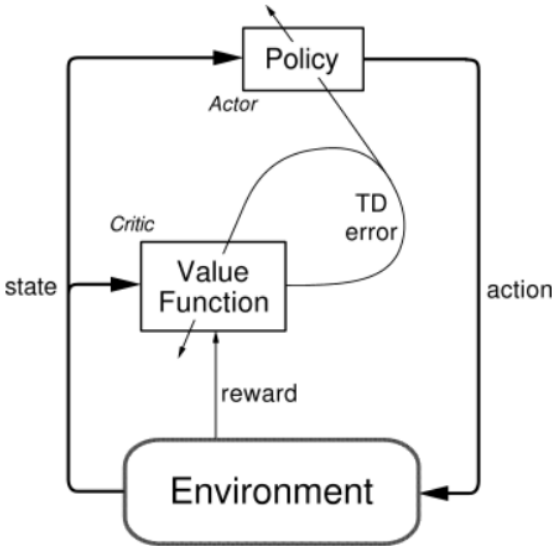


FIGURE 3.3: The actor-critic architecture [2]

The actor-critic methods are mostly on-policy, but off-policy methods have been introduced as well [44]. Further, these methods may have better convergence properties as compared to only critic based methods, as long as these methods are gradient-based. Actor critic methods have faster convergence due to variance reduction [45].

For the mathematical representation of actor critic method we consider [46]. The state value function in a policy π is V_π(s) and can be estimated by ${V_\pi}^V$(s) using neural networks as function approximators. Similarly, action value function Q_π(s,a) can be estimated by ${Q_\pi}^U$(s,a). Next, we substitute the state value function estimator into the Bellman equation (3.2) to get:

$$ {V_\pi}^V(s_t) \cong r + \gamma {V_\pi}^V(s_{t+1}) \tag{3.11} $$

We approximately get an error by using equation (3.11). This error is known as temporal difference error (TD error) represented by the symbol δ.

$$ \delta = {V_\pi}^V(s_t) - r - \gamma {V_\pi}^V(s_{t+1}) \tag{3.12} $$

This error δ is considered as loss function and being driven to zero using stochastic gradient ascent/descent. We consider policy function π with parameter vectors θ and J as expected return from state V_π(s). J is optimized according to *policy gradient theorem*

as [47]:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta(log\pi_\theta(s,a)) * Q_\pi(s,a)] \tag{3.13}$$

where π_θ is the estimate of policy π. These algorithms are termed as *stochastic actor-critic* algorithms.

Although the gradient of J has also been defined subject to deterministic policy by deterministic policy gradient (DPG) theorem [48]. In this case J is obtained as:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta(log\pi_\theta(s,a)) * \nabla_a Q_\pi(s,a)] \tag{3.14}$$

More efficiency is obtained while computing DPG gradients and these algorithms perform better than the stochastic counterparts.

3.8 Model Based methods

The interactions of an RL agent with its environment can be used to learn a model of the environment. In *model-based* methods, the agent tries to learn the model over time in order to simulate transitions and increase the sample efficiency. While model-free RL emphasizes learning, model-based RL emphasizes planning. A flow diagram of model-based RL is presented in figure (3.4).

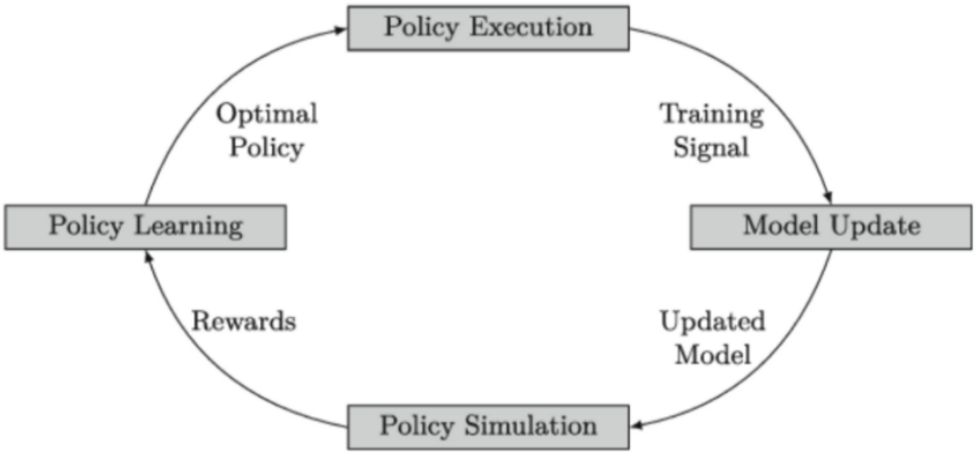


FIGURE 3.4: Flow diagram of model-based RL [3]

Model-based RL methods has higher sample efficiency, implying that it requires less data to learn a policy[49]. These methods can plan better by even simulating sequence of actions rather than performing them in actual world. The virtue of the modeling process enables the transfer of these methods to other tasks.

Model-based RL methods demand learning both policy and a model, thus increasing the degree of involved error. It might happen that the learnt model is not accurate, then the learnt policy would not be optimal or would just fail. For same reason, these methods are computationally demanding than model-free methods [3].

3.9 Summary

The chapter introduced the general reinforcement learning problem along with related concepts. Mathematical formulation of RL problem will later help in understanding the involved algorithms for this thesis (sections 5.3, 5.4, 5.5). Traditional RL approaches along with model-free and model-based methods were discussed. Model-free methods learn directly by interacting with the environment in real-world or in simulation, whereas model-based methods use limited interactions with the real environment and constructs a model for simulating the future episodes. The next chapter will discuss the basics of deep neural networks (DNN) and its relevance for RL.

Chapter 4

Deep Neural Networks

4.1 Introduction

Neural networks are a set of algorithms designed to recognize patterns. Modelling of these networks have been inspired by human brain and how it filters information [50, 51]. Neural networks with atleast two layers can be termed as deep neural networks (DNN).

DNNs have led to some of the recent achievements in artificial intelligence and are used to build state-of-art systems. These achievements lie in different domains including robotics [10, 52–54], image recognition [55], handwriting recognition [56], speech recognition [57], and autonomous vehicles [58]. A detailed history of DNNs is presented in [50].

Neural networks are used for approximating multivariate functions involving higher degrees of complexity. It has been proven that such approximations can achieve any degree of accuracy [59]. This chapter provides details on building blocks, common architecture and training of neural networks. Also, the chapter elaborates on a training technique *batch normalization* and a regularization technique *dropout*.

4.2 Building Blocks

Artificial neural networks consists of several interconnected neurons arranged in several layers.

4.2.1 Artificial Neuron

An basic unit of artificial neuron as seen in figure (4.1) takes an input vector x with weights w and calculates the weighted sum for all inputs. This weighted sum is added

to the bias denoted by b and passed through an activation function f to produce the final output represented by the following equation:

$$y_i = f(\sum_{j=0}^{n} x_j w_{i,j} + b_i) \tag{4.1}$$

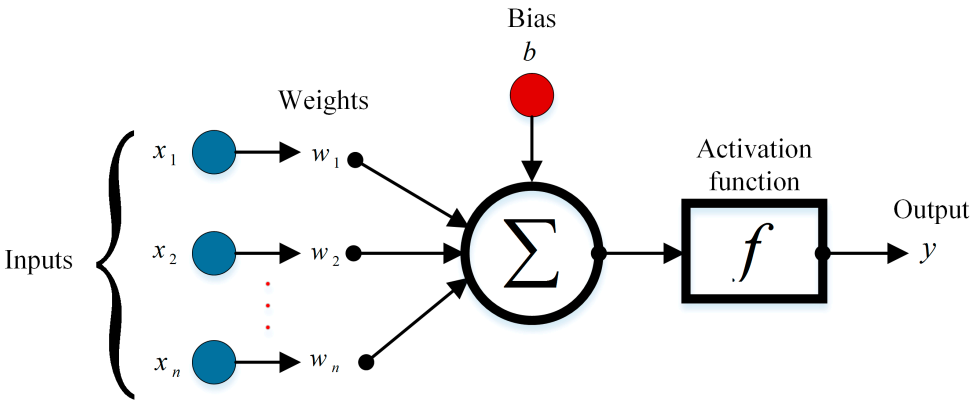


FIGURE 4.1: Basic model of an Artificial Neuron [4]

4.2.2 Activation Functions

Activation functions are mathematical expressions determining the output of an neuron. These functions can be linear or non-linear. Non-linear activation functions help the network to learn complex data, represent almost any function with reasonable accuracy. Many activation functions have been introduced in literature, some of the commonly used activation functions include *rectified linear unit (ReLu)*, *sigmoid* and *hyperbolic tangent*. These functions are mathematically expressed as:

- Rectified Linear Unit (ReLU)[60]:

$$f(x) = max\ (0, x) \tag{4.2}$$

- Sigmoid :

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4.3}$$

- Hyperbolic Tangent :

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{4.4}$$

$ReLU$ is most commonly used activation function and provides best performance over many tasks. Advantages of $ReLU$ over other activation functions include lower run time, sparsity and reduced likelihood of vanishing gradients.

4.3 Feed Forward Neural Networks

4.3.1 Architecture

Feed forward network is most straight forward and common architecture used in DNN. It consists of three layers, viz, *input layer*, *hidden layers* and *output layer*. The flow of information is from input layer towards the output layer through hidden layers as seen in figure 4.2. The 'deepness' of neural networks is based on the number of hidden layers. Complex architectures involving many hidden layers are found in literature [61].

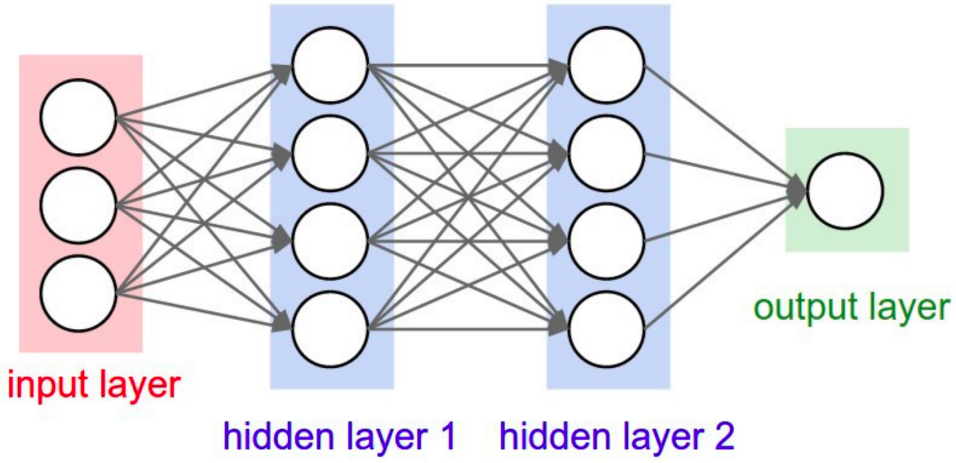


FIGURE 4.2: Feed forward neural network architecture [5]

4.3.2 Training

Initially the weights of the neural network are set randomly. Training a neural networks mean adjusting its weights using a technique called *backpropagation*. Each time an generated output is compared with the desired output and the error is used to update the weights using a gradient descent algorithm. As the weights of output layer are updated at first and then proceeding backward to the input layer, the technique is termed as *backpropagation*.

Recently, *stochastic gradient descent* have gained popularity for training neural networks. However, improved versions of stochastic gradient descent like *AdaGrad* [62] and *Adam* [63] are also available. These methods converge faster and consider only few training samples to update weights, as compared to vanilla gradient descent methods.

4.3.3 Batch normalization

Batch normalization (BN) is an method for training complex DNNs that standardizes the input to a layer for each mini-batch. As the normalization ensures activation values within certain upper and lower bounds, the approach leads to faster learning rates. BN significantly reduces the number of training epochs and stabilizes the learning process. BN method allows each layer to learn independently, improving the speed and accuracy of the training phase [64]. BN technique have been used in RL algorithms for the modelling part of this thesis.

4.3.4 Regularization Methods

The performance of DNNs improve as they are fed with more and more datasets. At the same time, a model can overfit over training dataset. Regularization methods are used to reduce over-fitting and to enhance the robustness of the neural network to unseen data. Different regularization techniques include L1 or L2 regularization, Dropouts[6], Batch normalization[64], Data Augmentation, and Early stopping. A comparison between different regularization techniques is available in [65].

4.3.4.1 Dropout

Dropout is an frequently used regularization technique for large neural networks. Essentially, the technique randomly 'drops' out or turns off certain neurons during the training. This lead to smaller network and other neurons step in and make predictions for missing neurons. Thus, the network learns an independent internal representation and is less sensitive to certain weights of the neurons. Such networks are more robust, eliminating the chances of over-fitting. However, during the testing, none of neurons are dropped.

Visual presentation of dropout technique is shown in figure (4.3). It consists of standard neural network on the left and a the same network after applying dropout on the right. As seen in right network, connections of certain neurons have been removed during the training phase. This removal of the connections is random and neurons to be removed are selected with probability p, which is an hyperparameter of the dropout function. Dropout can be applied to the hidden or input layers.

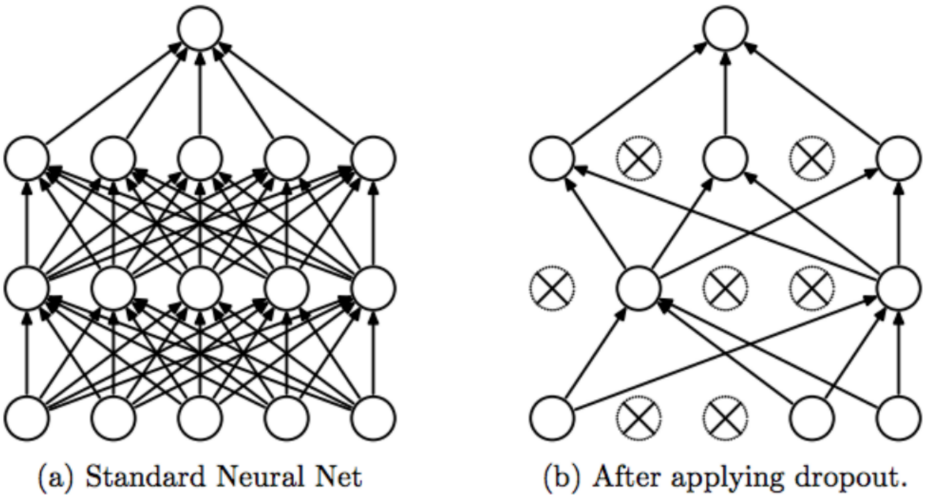


FIGURE 4.3: Neural network with and without dropout [6]

4.4 Summary

DNNs hold potential for fetching representations and learning from large unlabelled datasets. They are general purpose function approximators. DNNs are scalable systems that are being used for high-dimensional and dynamic training environments. DNNs will be used along with RL for training agents to perform specific task in continuous state-space (chapter 7).

Chapter 5

Deep Reinforcement Learning

5.1 Introduction

Deep Reinforcement Learning (DRL) helps intelligent machines to learn through their actions similar to how humans learn through their experience. Recently DRL has gained a lot of attention due to its breakthroughs in AlphaGo[66], solving rubik's cube with robotic hand [67], stratergy emergence in multi-agent systems [68], and Atari games [69]. As DRL algorithms can train an agent in high dimensions it becomes ideal choice for executing locomotive tasks in simulation.

DRL uses Deep Neural Networks as function approximators for policy or value functions in RL. DRL has been successfully applied in Q-learning [69, 70] and actor-critic methods[46]. This chapter provides details on deep Q-learning, elaborates on actor-critic methods, viz, Deep Deterministic Policy Gradient (DDPG), Twin delayed DDPG (TD3) and Soft Actor Critic (SAC). The chapter also provides details on the physics engine used for simulation and the agents involved for experimentation.

5.2 Deep Q Learning

Q-Learning (section 3.7.1.2) is an model-free reinforcement learning algorithm that is widely used. Combining DNN with Q learning proved to be huge success in Atari games [69, 70]. Since then the algorithm has been considered for high dimensional state space.

The paper [70] provides the mathematical background for this section. DNN parameterize the Q-function by a neural network with weights θ and the algorithm is termed

as *Deep Q-Learning* (DQN). The algorithm minimizes the loss function given by:

$$L_i(\theta)_i = \mathbb{E}_{s,a\sim\rho(.)}[(y_i - Q(s,a;\theta))]^2 \tag{5.1}$$

where $y_i = \mathbb{E}_{\ s'\sim\epsilon}[(r+\gamma\ \max_{a'}\ Q(s',a';\theta_{i-1})|\text{s,a}]$ is the target for i^{th} iteration and ρ(s,a) is the probability distribution over sequence s and actions a. Differentiating the loss function with respect to weights we obtain the following gradient:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(.);s'\sim\epsilon}[(r+\gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)] \tag{5.2}$$

It is computationally efficient to optimize the loss function by *stochastic gradient descent*. Weights are updated every time step by single samples obtained from behavior distribution ρ and the expectations from emulator ϵ.

DQN has advantages due to the use of *experience replay*. These include greater data efficiency as each step of experience is used in many weight update and reduced variance due to learning from randomized samples. The randomized samples are sampled from a fixed length representation of history given by the function ϕ. The DQN algorithm is given by:

Algorithm 3: Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for *episode = 1,* M **do**
  Initialise sequence $s_1 = x_1$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for t=*1,* T **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t),\ a;\ \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t,\ a_t,\ x_{t+1}$ and preprocess $\phi_{t+1} = \phi(\text{s}_{t+1})$
    Store transition $(\phi_t,\ a_t,\ r_t,\ \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j,\ a_j,\ r_j,\ \phi_{j+1})$ from D
    if ϕ_{j+1} *is terminal* **then**
      $y_j = r_j$
    else
      $y_j = r_j + \max_{a'} Q(\phi_{j+1},\ a';\ \theta)$
    end if
    Perform a gradient descent step on $(y_j$ - $Q\ (\phi_j,\ a_j;\ \theta))^2$ using eq. 5.2
  end for
end for

DQN cannot be applied to tasks for continuous action spaces due to its need for calculating the maximum over actions. To overcome this issue and to work in continuous action spaces, an actor-critic algorithm has been developed that uses Q-function as critic, updating the policy using deterministic policy gradient as introduced in section (3.7.3).

5.3 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) [53] learns a Q-function and a policy for continuous action spaces. Both the actor and Q-function being used as critic are approximated by two separate neural networks. DDPG is similar to DQN for continuous action spaces. It is an off policy algorithm. An overview of the algorithm is presented in figure (5.1).

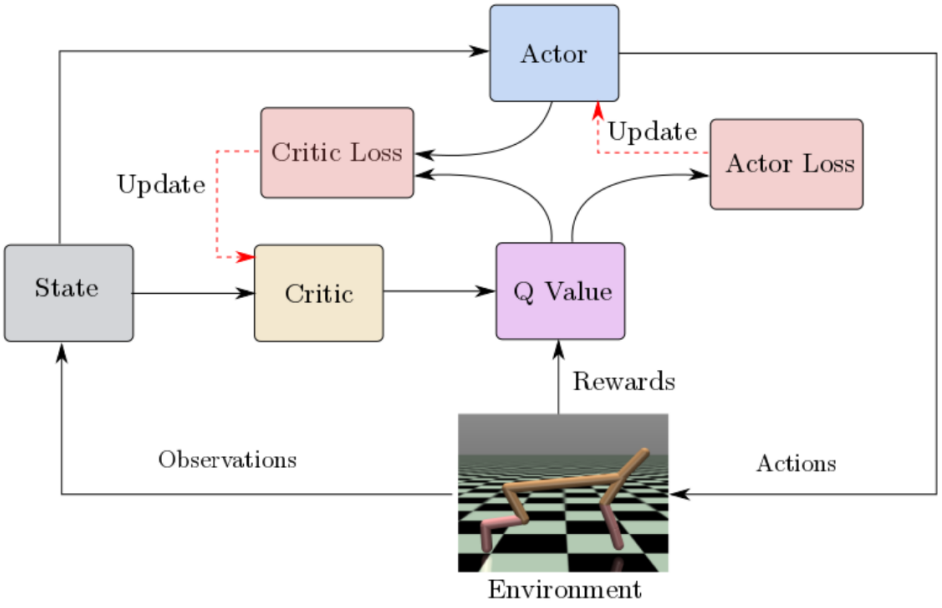


FIGURE 5.1: An overview of DDPG algorithm [7]

Updating the actor-critic parameters using equation (5.2) may not result in the learning to converge. DDPG algorithm includes a couple of tricks to traditional actor-critic approach with neural networks. The first trick is to use *experience replay* which was used in DQN. The other trick is using *target networks*. Target networks are separate neural networks that are copies of actor and critic networks. The weights of these target networks are updated slowly to track the learned networks. This constraint improves the stability of the learning. Target networks are used to calculate the TD error y as expressed below [35]:

$$y = \mathbb{E}_{s' \sim \epsilon}[r + \gamma Q'(s', \mu'(s'; \theta^{\mu'}); \theta^{Q'})] \tag{5.3}$$

where Q'(s,a;$\theta^{Q'}$) and $\mu(s;\theta^{\mu'})$ are the target networks for critic and actor with weights $\theta^{Q'}$ and $\theta^{\mu'}$ respectively.

The loss function is given by:

$$L = \mathbb{E}_{s'\sim\rho,a\sim\beta}(y - Q(s,a;\theta^{Q'}))^2 \tag{5.4}$$

where β is behaviour policy and Q(s,a;θ^Q) is the critic network with weights θ^Q.

Policy updates are obtained from deterministic policy gradient theorem[48] as:

$$\nabla_{\theta^\mu} J = \mathbb{E}_{s\sim\rho_\beta}[\nabla_a Q(s,a;\theta^Q)|_{s,a=\mu(s;\theta^\mu)} \nabla_{\theta^Q}\mu(s;\theta^\mu)|s] \tag{5.5}$$

where $\mu(s;\theta^\mu)$ is the actor network with weights θ^μ.

The DDPG algorithm is presented in algorithm (4) and DDPG hyperparameters used for this thesis are expressed in table (5.1).

Algorithm 4: Deep Deterministic Policy Gradient

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize critic network Q(s,a|θ^Q) and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize the target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
for *episode = 1,* M **do**
  Get initial state s
  Initialize a random process $\mathcal{N}$ for action exploration
  for t=*1,* T **do**
    Select action $a_t = \mu(s_t;\theta^\mu) + \mathcal{N}$
    Perform action a_t, observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Calculate the target using (5.3) for each transition
    Update the critic by minimizing the loss in (5.4)
    Update the actor network using (5.5)
    Update target networks using:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1 - \tau)\theta^{\mu'}$$

  end for
end for

TABLE 5.1: Hyperparameters for DDPG algorithm

Hyperparameters	Value
Discount factor γ	9.9×10^{-1}
Soft target network update parameter τ	10^{-3}
Batch size	64
Actor learning rate	3×10^{-4}
Critic learning rate	3×10^{-3}
Training steps	10^6
Exploration steps	10^3
Maximum steps for each episode	10^3
Replay buffer size	10^5
Dimensions of first hidden layer for actor and critic	400 units
Dimensions of second hidden layer for actor and critic	300 units
Exploration noise	0.1
Policy noise	0.2
Noise clip	0.5
Nonlinearity	ReLU[60]
Optimizer	Adam [63]

5.4 Twin Delayed DDPG (TD3)

DDPG achieves good performance sometimes, but requires hyperparameter tuning. DDPG fails when the learned Q-function begins to overestimate the Q-values dramatically, this exploits the error in Q-function and eventually the policy breaks. Twin Delayed DDPG (TD3) [71] addresses these issues by considering three tricks [34]:

- Firstly, TD3 learns two Q-functions instead of one, using the smaller of two Q-values as the target in Bellman error loss function. This trick is termed as 'Clipped Double Q-learning'. The corresponding target in Bellman error loss function is:

$$y(r, s') = r + \gamma \min_{i=1,2} Q_{\phi_{i,targ}}(s', a'(s')) \tag{5.6}$$

  where Q_{ϕ_1} and Q_{ϕ_2} are two Q-functions with weights ϕ_1 and ϕ_2 respectively.

- Secondly, less frequent updates of the policy and the target networks than the Q-function. This trick is known as 'Delayed Policy Updates'.

- Thirdly, exploitation of Q-function errors is made harder for the policy by adding noise to the target action. This trick is termed as 'Target Policy Smoothing'. The clipped target action is expressed as:

$$a'(s') = clip\Big(\mu_{\theta_{targ}}(s') + clip(\epsilon, -c, c), a_{low}, a_{high}\Big), \quad \epsilon \sim \mathcal{N}(0, \sigma) \tag{5.7}$$

where $\mu_{\theta_{targ}}$ is the target policy, ϵ is the added noise, a_{low} and a_{high} are the lower and higher bounds for all valid actions a.

These three tricks altogether improves the performance over baseline DDPG. The loss functions for two Q-functions are:

$$L(\phi_i, \mathcal{D}) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(y(r,s') - Q_{\phi_i}(s,a)\right)^2\right] \tag{5.8}$$

The TD3 algorithm can be viewed in algorithm (5) and the hyperparameters used for this thesis are presented in table (6.1):

Algorithm 5: Twin Delayed DDPG

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize critic networks Q_{ϕ_1}, Q_{ϕ_2} with weights ϕ_1, ϕ_2 and actor μ_θ with weights θ
Set target parameters to main parameters $\theta_{targ} \leftarrow \theta, \phi_{targ,1} \leftarrow \phi_1$, $\phi_{targ,2} \leftarrow \phi_2$
for *episode = 1,* M **do**
  Get initial state s_t
  for t=*1,* T **do**
    Select action a_t using (5.7)
    Perform action a_t, observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    if *it's time to update* **then**
      for *j in range(how many updates)* **do**
        Randomly sample a batch $\mathcal{B}$ of transitions from $\mathcal{D}$
        Compute target actions *a'*(s') using (5.7)
        Compute targets using (5.6)
        Update Q-functions by one step of gradient descent using (5.5)
        if *j mod policydecay = 0* **then**
          Update the actor network using

$$\nabla_\theta \frac{1}{|\mathcal{B}|} \sum_{s \in \mathcal{B}} Q_{\phi_1}(s, \mu_\theta(s))$$

          Update target networks using:

$$\phi_{targ,i} \leftarrow \tau \phi_{targ,i} + (1 - \tau)\phi_i, \quad \text{for } i = 1, 2$$
$$\theta_{targ} \leftarrow \tau \theta_{targ} + (1 - \tau)\theta$$

        end if
      end for
    end if
  end for
end for

TABLE 5.2: Hyperparameters for TD3 algorithm

Hyperparameters	Value
Discount factor γ	9.9×10^{-1}
Soft target network update parameter τ	5×10^{-3}
Batch size	256
Actor learning rate	3×10^{-4}
Critic learning rate	3×10^{-4}
Training steps	10^6
Exploration steps	10^3
Maximum steps for each episode	10^3
Replay buffer size	10^5
Dimensions of first hidden layer for actor and critic	400 units
Dimensions of second hidden layer for actor and critic	300 units
Exploration noise	0.1
Policy noise	0.2
Noise clip	0.5
Nonlinearity	ReLU[60]
Optimizer	Adam [63]

5.5 Soft Actor Critic (SAC)

Soft Actor Critic (SAC) algorithm [72] is a bridge between stochastic policy optimization and DDPG-based approaches. It optimizes stochastic policy in off-policy way. Similar to TD3, it uses clipped double-Q trick and also benefits from something similar to target policy smoothing.

Entropy regularization is the central feature of SAC. Entropy is a measure of randomness in the policy. The actor network or the policy is trained to maximize the expected reward along with maximizing the entropy. This aligns with exploration-exploitation strategy, increase in entropy imply more exploration [34].

The Bellman equation for the entropy regularized Q^π is approximated by:

$$Q^\pi(s,a) \approx r + \gamma\Big(Q^\pi(s',\tilde{a}') - \alpha\, log\pi(\tilde{a}'|s')\Big), \quad \tilde{a}' \sim \pi(\cdot|s') \tag{5.9}$$

where
π_θ is the policy with parameter θ,
$\tilde{a}$' are the next actions to be sampled from from the policy and not the replay buffer, whereas r and s' are sampled from the replay buffer,
$\alpha > 0$ is the entropy regularization coefficient,
log$\pi(\cdot|$s') represents the entropy term for policy π

SAC also learns two Q-functions similar to TD3, using the smaller one as the target in Bellman error loss function. The loss function is given by equation (5.8). The target is given by:

$$y(r, s') = r + \gamma\left(\min_{i=1,2} Q_{\phi_{targ,i}}(s', \tilde{a}') - \alpha\, log\pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s') \tag{5.10}$$

The SAC algorithm is presented in algorithm (6) and the hyperparameters used for this thesis are listed in table (5.3).

Algorithm 6: Soft Actor Critic

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize critic networks Q_{ϕ_1}, Q_{ϕ_2} with weights ϕ_1, ϕ_2 and actor μ_θ with weights θ
Set target parameters to main parameters $\phi_{targ,1} \leftarrow \phi_1$, $\phi_{targ,2} \leftarrow \phi_2$
for *episode = 1,* M **do**
  Get initial state s_t
  for t=*1,* T **do**
    Select action $a_t \sim \pi_\theta(\cdot|s)$
    Perform action a_t, observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    if *it's time to update* **then**
      for *j in range(how many updates)* **do**
        Randomly sample a batch $\mathcal{B}$ of transitions from $\mathcal{D}$
        Compute targets for the Q-functions using (5.10)
        Update Q-functions by one step of gradient descent using (5.8)
        Update policy by one step of gradient descent using

$$\nabla_\theta \frac{1}{|\mathcal{B}|} \sum_{s\in\mathcal{B}} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha\, log\pi_\theta(\tilde{a}_\theta(s)|s)\right)$$

        Update target networks using:

$$\phi_{targ,i} \leftarrow \tau\phi_{targ,i} + (1 - \tau)\phi_i \quad \text{for i=1,2}$$

      end for
    end if
  end for
end for

TABLE 5.3: Hyperparameters for the SAC algorithm.

Hyperparameters	Value
Discount factor γ	9.9×10^{-1}
Soft target network update parameter τ	5×10^{-3}
Alpha α	$[0.01, 0.033, 0.05, 0.1, 0.2, 0.33]$
Batch size	256
Actor learning rate	3×10^{-4}
Critic learning rate	3×10^{-4}
Policy learning rate	3×10^{-4}
Training steps	10^6
Exploration steps	10^3
Maximum steps for each episode	10^3
Replay buffer size	10^5
Dimensions of first hidden layer for actor and critic	400 units
Dimensions of second hidden layer for actor and critic	300 units
Gradient steps (updates per step)	1
Target update interval	1
Automatic entropy tuning	False
Log sig max	2
Log sig min	-20
Epsilon ϵ	10^{-6}
Nonlinearity	ReLU[60]
Optimizer	Adam([63])

SAC is particularly sensitive to the entropy regularization coefficient α. By choosing the right value of the parameter, the model is able to balance between exploration and exploitation. This leads to faster learning and better performance. However, the optimal value of α varies between different agents and needs to be tuned separately. Four different continuous control agents were considered for the course of this thesis, as visible in section (5.7). The experimentation design (chapter 7) describes the training of these agents corresponding to six different values of α in section 7.3.1 inspired by the values found in [72].

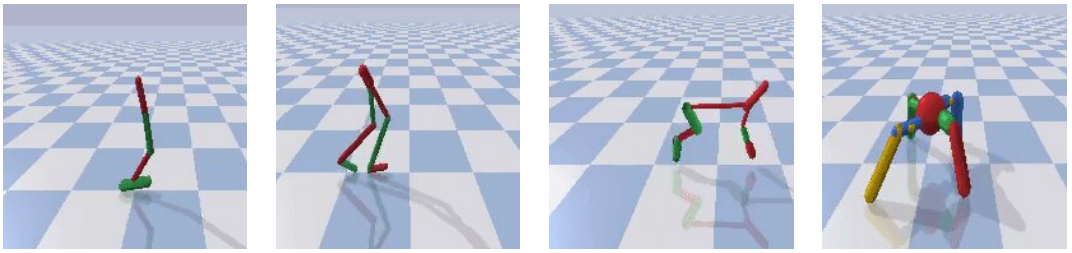
So far we have introduced and elaborated on different reinforcement learning algorithms, the proceeding sections will describe the physics engine used for reinforcement learning simulations and details on four agents.

5.6 Pybullet Physics Engine

PyBullet[73] is an simple python module to Bullet[74]. The Bullet is a free and open source physics engine. Previously, the MuJoCo[75] physics engine was used for many of the standard reinforcement learning environments. An interesting comparison between different simulation tools designed for robotics can be viewed in [76].

5.7 Continuous Control Agents

The thesis considers four continuous control agents that are used for experiments as shown in figure (5.2). The goal for each of these agents is to run as fast as possible by performing locomotion. The observational state space and the action space for each agent is presented alongside. The reward obtained for these agents is the delta in distance covered and small penalty proportional to magnitude of performed action. The state space consists of angular positions and velocities of the joints. The reward contains the delta in distance covered as well as a small penalty proportional to the magnitude of the action.



(a) Hopper (b) Walker2d (c) HalfCheetah (d) Ant

FIGURE 5.2: Continuous control agents in PyBullet physics engine

- **Hopper** ($\mathcal{S} \in \mathrm{R}^{15}$, $\mathcal{A} \in \mathrm{R}^{3}$) is an 3 DoF two-dimensional one-legged agent. The goal to run forward is achieved by hopping (thus the name). The environment is reset when the hopper model falls on the ground. The default values of different hopper limbs in the pybullet simulator are listed in table (5.4).

TABLE 5.4: Default values of Hopper model

Model parameters	Width	Length
Torso	0.05	1.45
Thigh	0.05	1.05
Leg	0.04	0.6
Foot	0.06	0.26

- **Walker2D** ($\mathcal{S} \in \mathrm{R}^{22}$, $\mathcal{A} \in \mathrm{R}^{6}$) is a 6-DoF two-legged agent resembling two connected instances of the Hopper model. The goal to move forward is achieved by walking motion. The environment is reset when the model falls over. Default values of walker2d limbs for pybullet simulator are listed in table (5.5).

Table 5.5: Default values of Walker model

Model parameters	Width	Length
Torso	0.05	1.45
Thigh	0.05	1.05
Leg	0.04	0.6
Foot	0.06	0.2

- **HalfCheetah** ($\mathcal{S} \in R^{26}$, $\mathcal{A} \in R^{6}$) is an 6 DoF two-dimensional model. The goal to move forward is achieved by running. The default values for this agent used in simulation are listed in table (5.6).

Table 5.6: Default values of Half-Cheetah model

Model parameters	Width	Length
Torso	0.046	0.5
Head	0.046	0.15
Front thigh	0.046	0.145
Front shin	0.046	0.15
Front foot	0.046	0.094
Front thigh	0.046	0.133
Front shin	0.046	0.106
Front foot	0.046	0.07

- **Ant** ($\mathcal{S} \in R^{28}$, $\mathcal{A} \in R^{8}$) is an 8 DoF three-dimensional four-legged agent. The agent uses alternate legs at a time for performing forward locomotion. The default limbs and their values for this agent are listed in table (5.7).

Table 5.7: Default values of Ant model

Model parameters	Width	Length
Torso	0.25	0.25
Front left leg joint1	0.08	0.2
Front left leg joint2	0.08	0.2
Front left leg foot	0.08	0.4
Front right leg joint1	0.08	0.2
Front right leg joint2	0.08	0.2
Front right leg foot	0.08	0.4
Left back leg joint1	0.08	0.2
Left back leg joint2	0.08	0.2
Left back leg foot	0.08	0.4
Right back leg joint1	0.08	0.2
Right back leg joint2	0.08	0.2
Right back leg foot	0.08	0.4

5.8 Summary

We described three algorithms that are suitable for training four continuous action spaces reinforcement learning agents. Further, we provide details on the self-supervised learning model used for extracting feature representations from perturbed environment data in chapter (6).

Chapter 6

Self Supervised Learning

6.1 Introduction

Generating labels for a dataset is expensive, while unlabelled data can be generated all the time. The motivation for Self-Supervised Learning (SSL) is to consider unlabelled dataset and exploit a variety of labels. This would help in gaining supervision from the dataset itself. In the *self-supervised task* we are interested in the intermediate learned representations, with an assumption that representation model can learn characteristics or the high-quality latent variables of the unlabelled dataset for real-world tasks [77].

SSL technique for learning representation models have been employed in reinforcement learning[78, 79], machine learning[80], computer vision [81–83], natural language processing[84], and robotics[31, 32]. For the course of thesis, an SSL model, wav2vec[8] which is developed for speech recognition will be integrated with the state-of-art RL algorithms from chapter (5).

6.2 Self supervised learning model

6.2.1 The wav2vec model

The wav2vec model was originally designed to have raw audio signal as input, and then passed through two convolutional networks. The first network 'encoder network' embeds the input in a latent space and the second 'context' network generates contextualized representations by combining multiple time-steps of the encoder network as seen in figure (6.1). Both networks are used to compute the contrastive loss function in equation (6.1).

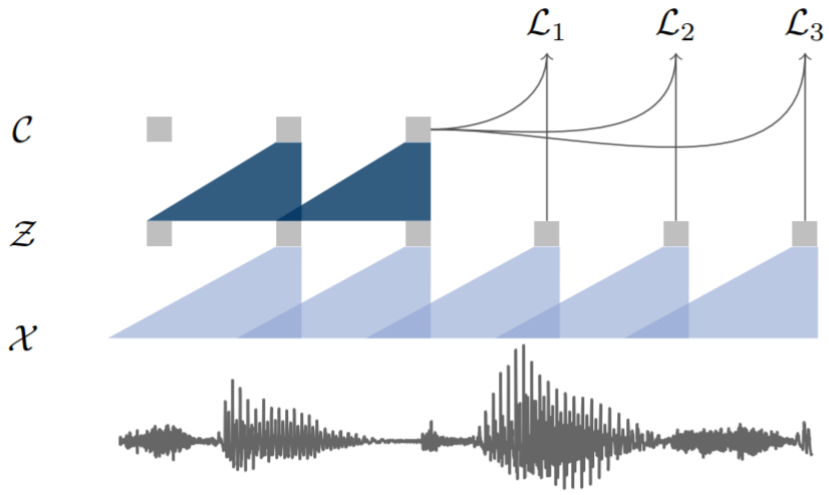


FIGURE 6.1: Pre-training of audio data $\mathcal{X}$, encoded with two convolutional neural networks. The model is optimized to predict the next time step task [8]

The hyperparameters used to train the wav2vec model for this thesis are listed in table (6.1).

$\mathcal{X}$ represents the raw audio samples, the *encoder network* is given by f: $\mathcal{X} \mapsto \mathcal{Z}$ and the *context network* given by g: $\mathcal{Z} \mapsto \mathcal{C}$. For a given receptive field size v, *context network* combines multiple latent representations z_i ... z_{i-v} into single contextualized tensor $c_i = g(z_i...z_{i-v})$.

TABLE 6.1: Hyperparameters for Wav2vec model

Hyperparameters	Value
Receptive field	16
Number of workers	6
Max. update	4 $\times 10^5$
Save interval	1
Log interval	10
Architecture	wav2vec base
Learning rate scheduler	cosine
Learning rate	1e-6
Minimum learning rate	1e-9
Maximum learning rate	5 $\times 10^{-4}$
Convolutional feature layers	(64,128,128)
Convolutional aggregator layers	(128, 128)
Prediction steps	1-16
Offset	auto
Residual scale	0.5
Warmup updates	500
Warmup init learning rate	1e-07
criterion	binary cross entropy
Number of negatives	10
Sample size	10^3
Max. tokens	10^4
Optimizer	Adam[63]

6.2.2 Contrastive loss function

The model is trained to distinguish a sample z_{i+k} which is k future steps from distractor samples $\tilde{z}$ drawn from proposal distribution p_n, by minimizing the contrastive loss given by[8]:

$$\mathcal{L}_k = -\sum_{i=1}^{T-k} \left(log\ \sigma\big(\mathbf{z}_{i+k}^T h_k(\mathbf{c}_i)\big) + \lambda \mathop{\mathbb{E}}_{\tilde{\mathbf{z}}\sim p_n} \left[log\ \sigma(-\tilde{\mathbf{z}}^T\ h_k(\mathbf{c}_i)) \right] \right) \tag{6.1}$$

where $\mathcal{L}_k$ is computed for every k = 1,....,K. T is the sequence length and λ was set to number of negative samples. The sigmoid is denoted by σ(x)= 1/1+exp(-x) and the probability of z_{i+k} being the true sample is given by $\sigma\big(z_{i+k}^T h_k(c_i)\big)$. The $h_k(c_i) = W_k c_i + b_k$ is a step-specific affine transformation for each step k, applied to c_i [85]. The loss $\mathcal{L} = \sum_{k=1}^{K} \mathcal{L}_k$, is optimized over different step sizes.

The wav2vec model outperformed the best character-based system in speech recognition literature, while using less labeled training data.

6.3 Summary

The chapter introduced the basic understanding of self supervised learning and it's applications in different domains. Functional concepts of an SSL technique - wav2vec model were elaborated along with the involved contrastive loss function. A motivation to consider wav2vec model for the course of this thesis was stated in this chapter. The proceeding chapters will describes the experimental setups and the results for different RL agents considering deep RL algorithms integrated with and without wav2vec model.

Chapter 7

Experimental Design

7.1 Introduction

The previous chapters provided theoretical background for the reinforcement learning algorithms (policies) and self-supervised learning model. This chapter demonstrates the experimental design steps for testing the state-of-art RL policies over the morphological changes of an agent's body (perturbed environments). We illustrate how these perturbed environments are generated, an overview of the research hypothesis.

7.2 Generating novel morphological perturbed environments

We consider the continuous control agents from section (5.7) and make morphological perturbations according to an gaussian normal distribution in two parameters, viz, 'length' and 'width' for each individual limbs. The mean of the normal distribution is 'default value' of length and width (as shown for different agents in section (5.7)), and three different values of standard deviation - 0.05, 0.1 and 0.2 are considered for sampling. Thus, the two parameters for individual limbs are sampled according to the criteria shown in figure (7.2). 200 perturbed environments are generated corresponding to each value of standard deviation, resulting in total 600 perturbed environments. An comparison between baseline and morphological perturbations for all agents is presented in figure (7.1).

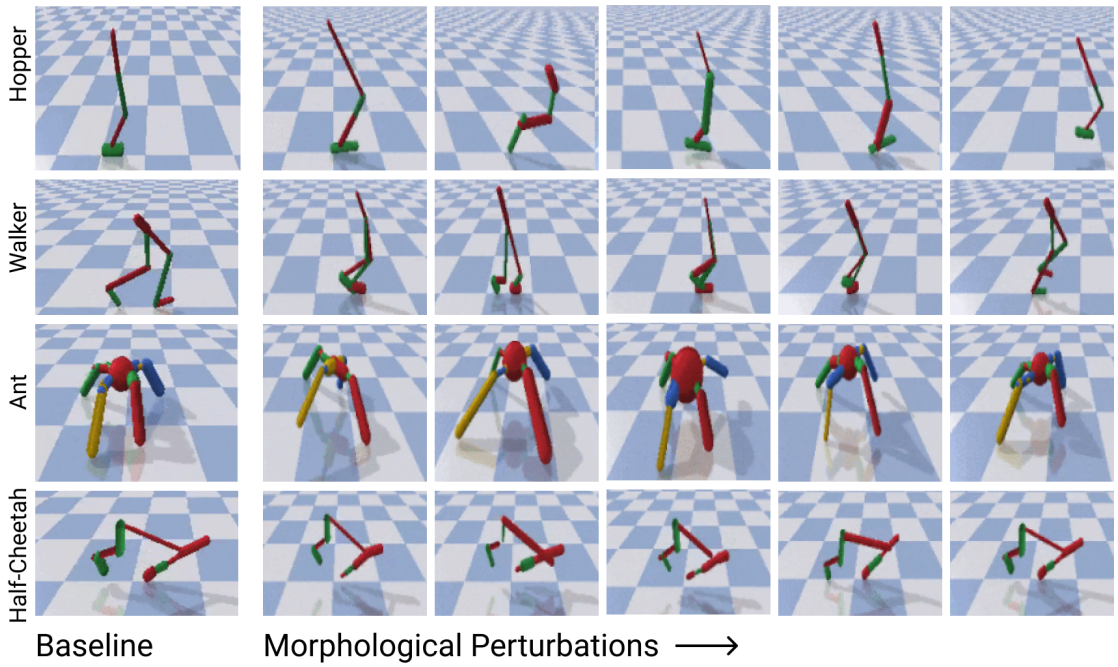


FIGURE 7.1: Benchmarking robustness to morphological perturbations in PyBullet. The modified morphology of Hopper, Walker2D, Half-Cheetah and Ant is a challenging test for the policies trained on baseline.

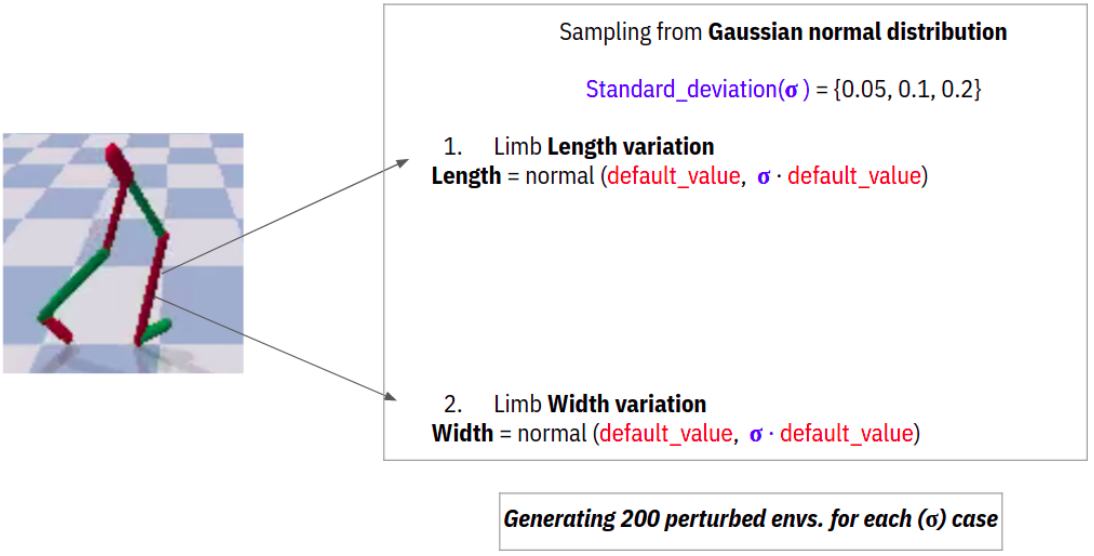


FIGURE 7.2: Generating novel perturbed environments according to gaussian normal distribution with three different standard deviation values

7.3 Overview of the research hypothesis and experimental setup

We propose an hypothesis that SSL representations improve the robustness of RL policies. Accordingly, an visual presentation of the experimental setup can be viewed in figure (7.3). At first, the reinforcement learning algorithms (policies) are trained on

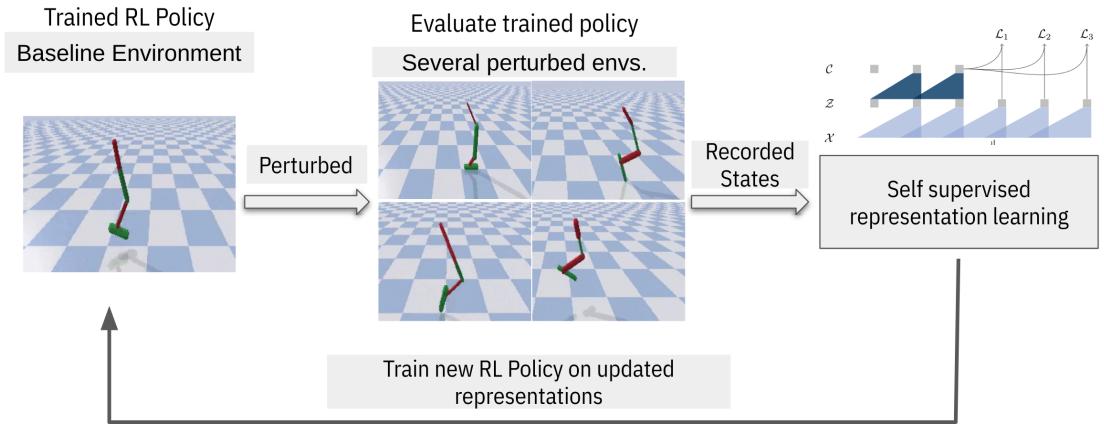


FIGURE 7.3: Overview of the research hypothesis and experimental setup involving - training RL policy on baseline envs., evaluating on perturbed envs., training SSL model, and integrating representations with RL policy

the baseline environment for different agents. Next, these trained policies are tested on different novel perturbed environments (old) and agent states are recorded. These recorded states are fed into the self supervised learning model to gain representations of the data. Then, these representations are integrated with RL policies. These RL policies are again trained with updated representations over baseline environment and finally tested on new and old perturbed environments.

7.3.1 Step 1 - Training RL policies over baseline environments

We train one million time-steps for different policies consisting of several episodes. Each episode is limited to maximum 1000 time-steps. We train multiple seeds of RL policies for each continuous control agent over their baseline environments in PyBullet simulator. For all the agents, we have trained 50 seeds for DDPG, 50 seeds of TD3 and 60 seeds of SAC. While training for SAC, we considered six different values of α and trained 10 seeds corresponding to each α value.

7.3.2 Step 2 - Evaluating trained RL policies on novel perturbed environments

The multiple seeds of trained RL policies - DDPG, TD3 and SAC are evaluated on novel perturbed environments generated according to section (7.2). Since we considered 200 perturbed environments for each value of standard deviation, we use this same 200 envs. for evaluating the performance of three RL policies in order to have fair comparison. We compare the performance of policies based on the rewards obtained on perturbed envs. This evaluation step is shown in figure (7.4).

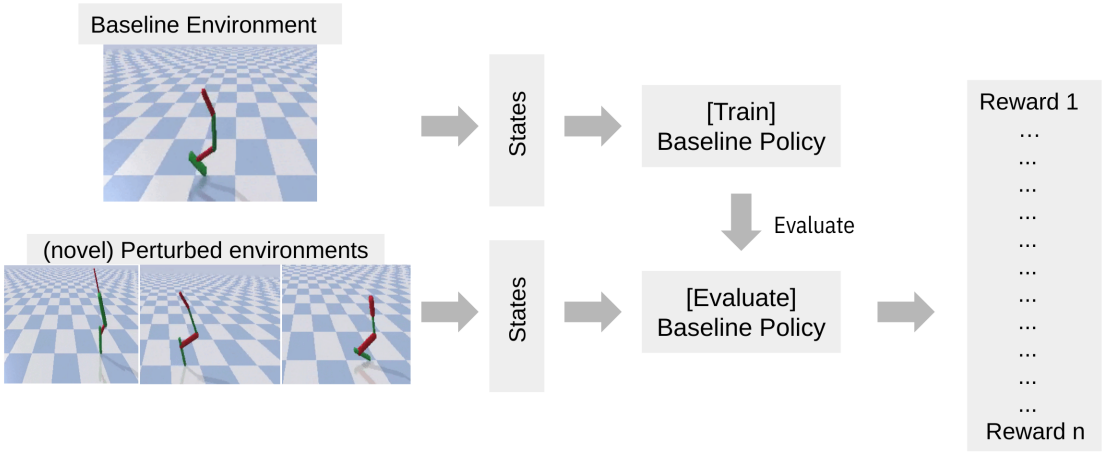


FIGURE 7.4: Step 2 - Evaluating several trained RL policies (DDPG, TD3, SAC) on novel perturbed environments

7.3.3 Step 3 - Training the SSL model using the data from N most suitable environments

In the previous step we reported the rewards obtained by evaluating RL policies over several perturbed envs. In this step we consider the data from all 600 perturbed envs. (200 envs. for each of 3 different standard deviation values) for all RL algorithms. The rewards obtained and reported are the 'median' of an specific experiment.

The collected data is split into training and validation sets with 80% for the former and 20% for later. Importantly, we perform this split based on the perturbed envs., meaning all data from randomly chosen 160 envs. (80% of 200) is considered for training set, whereas remaining data from 40 envs. is considered for validation set. Similarly, the same procedure is followed for data corresponding to different standard deviation values for all agents.

Next, we choose the N most suitable envs. by having a median reward threshold above a value of 750. This choice of reward threshold is sufficiently enough for an RL agent to solve the original task of moving/running in forward direction.

We consider the recorded states data from these N most suitable envs. to train the wav2vec SSL model. Note that this process of selecting N most suitable envs. is done separately for each agent and SSL model is trained separately for each agent, while the condition of reward threshold being 750 for all the agents. This experimental step is visible in figure (7.5).

We train two variants of wav2vec model for Hopper with receptive fields 16 and 32. While for other three agents, we train the wav2vec model only with receptive field value of 16.

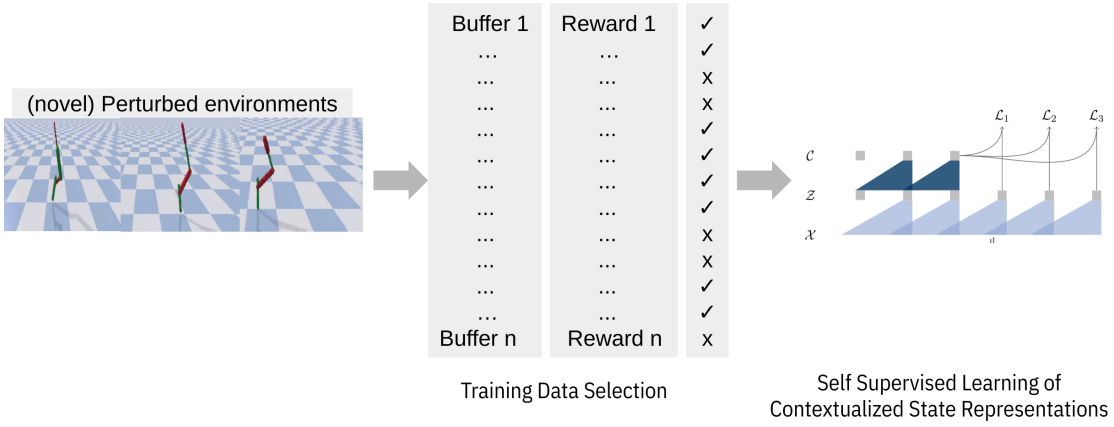


FIGURE 7.5: Step 3 - Pre-training contextualized representations jointly on the N most suitable environments

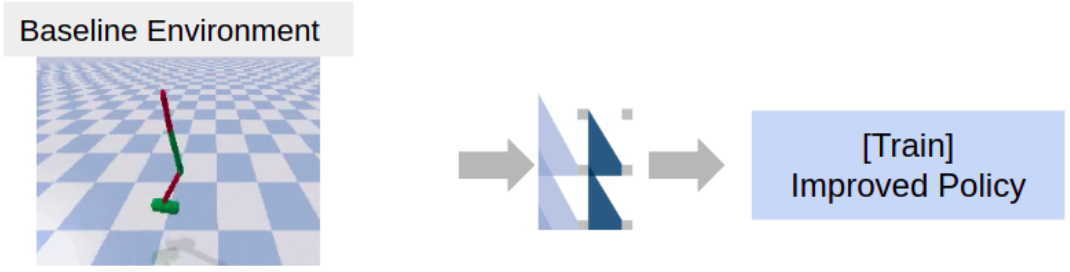


FIGURE 7.6: Step 4 - Integrating the (non-invariant) state representation and re-training the RL policy

7.3.4 Step 4 - Integrating the non-invariant state representations and re-training the RL policy

This step involves integrating the non-invariant state representations from SSL model and re-training the baseline RL policy as presented in figure (7.6). The hypothesis is that the state representations improve the performance of baseline policy. In this step we train various models, viz, training 10 seeds on just SSL representations, and integrated SSL representations with the last state vector (obtained by default from the simulator)

By default, the RL policies are designed to train on only the last state vector, further in order to understand how increasing the memory affects the training process we provide results from training on last 8 state vectors, and last 16 state vectors

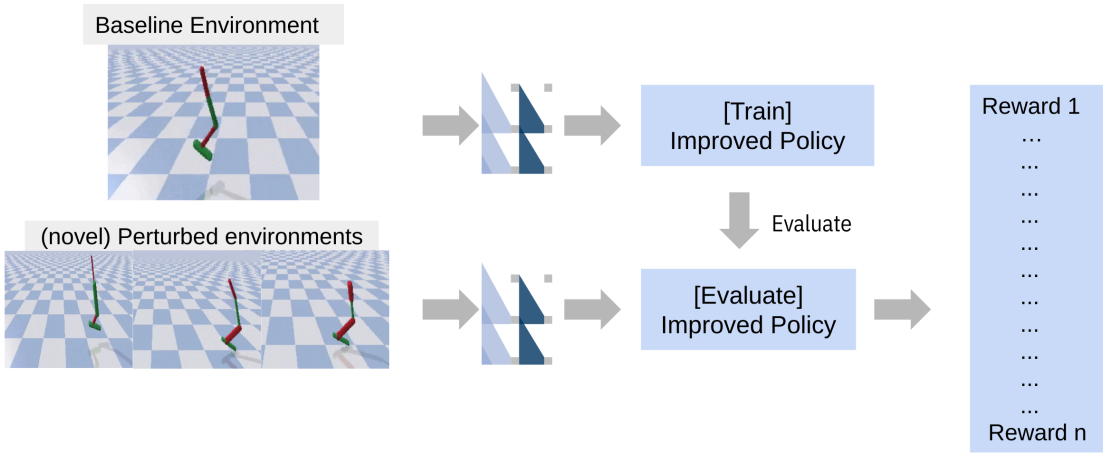


FIGURE 7.7: Step 5 - Evaluating the re-trained RL policy over perturbed environments

7.3.5 Step 5 - Evaluating the re-trained RL policy over new perturbed environments

The step includes evaluating the re-trained RL policy (trained by integrating the SSL representations) over new perturbed environments as shown in figure (7.7). Here, we again sample the perturbed envs. according to the same criteria as described in section (7.2) and term these as 'new' perturbed envs. Also, for curiosity we evaluated the re-trained RL policy over previous perturbed envs. that were generated during step 2 and call them as 'old' perturbed envs. For this step we only consider TD3 policy and for the Hopper agent.

7.3.6 Step 6 - Comparing the baseline RL policy performance over perturbed environments with and without integrated SSL representations

In this step we compare the performance of RL policy, specifically TD3 on the perturbed environments before and after integrating the SSL representations as shown in figure (7.8). This step compares the results from step 2 and step 5. We support or falsify our initial hypothesis that SSL representations improve the robustness of RL policies.

7.4 Summary

The chapter described about the research hypothesis in general and the breakdown of experimental design into several steps in order to support or falsify the hypothesis. For experimental step 3, I would like to thank Jin Hwa Lee from Technische Universität

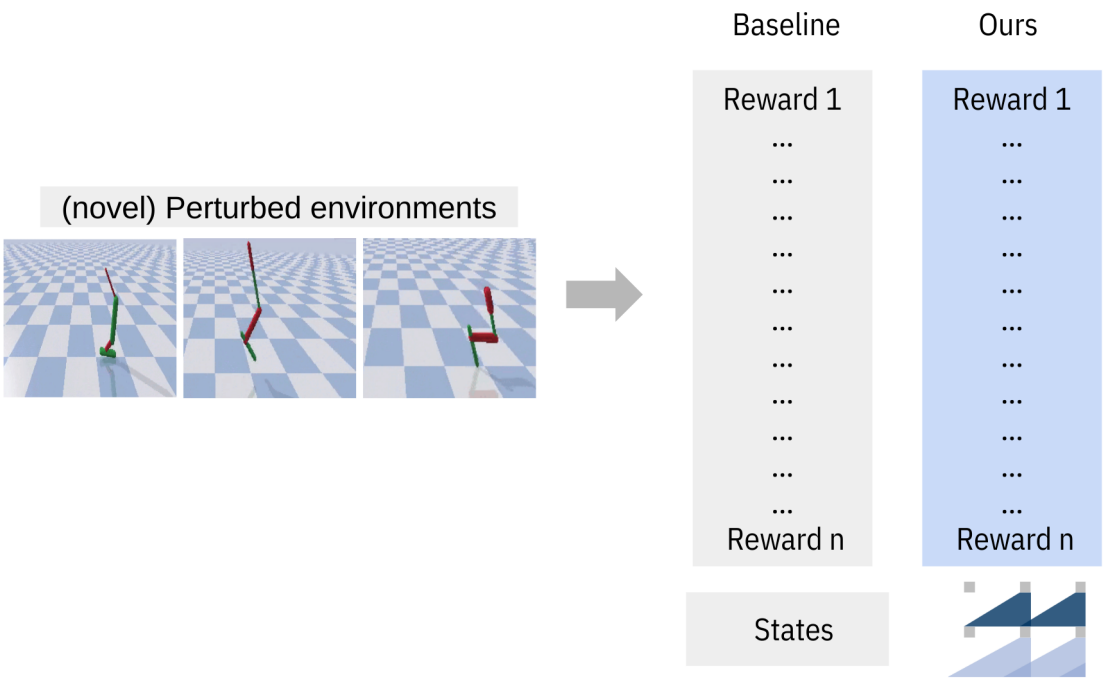


FIGURE 7.8: Step 6 - Comparing the performance of baseline RL policy over perturbed environments with and without integrated SSL representations

Munich (TUM) for fine tuning and generating all trained wav2vec models for various RL agents. Next chapter provides the results and analysis for different experimental design steps.

Chapter 8

Experimental results

8.1 Introduction

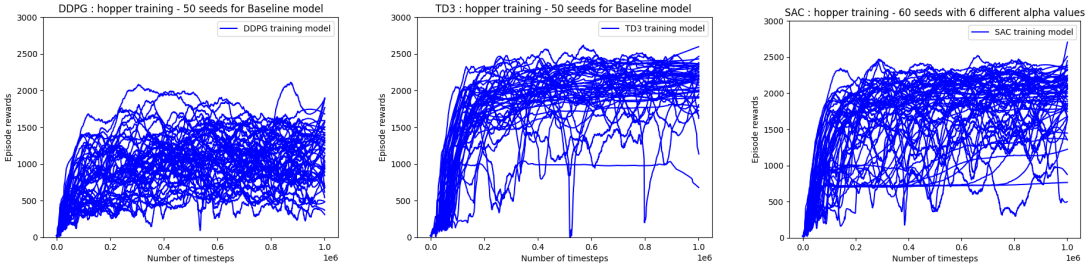
This chapter provides the results for different experimental steps described in the previous chapter. We show that a correlation exists between the degradation in rewards with the difference in environments. The chapter also provides description and analysis on the obtained results.

8.2 Experimental setup results

8.2.1 Step 1 - Training RL policies over baseline environments

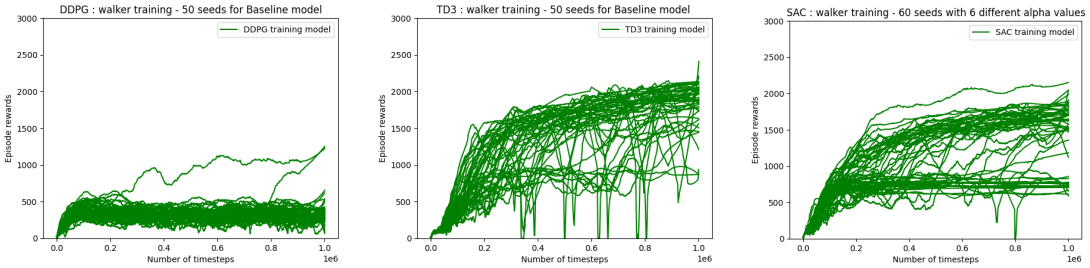
Results of training multiple seeds for different RL policies corresponding to four different agents are presented in figure (8.1, 8.2, 8.3, 8.4). Training results includes 50 seeds for DDPG and TD3, while 60 seeds for SAC involving six different α values. Benchmarking results for different agents on PyBullet environments for TD3 and DDPG policies can be viewed in [86].

Considering the training results for different agents, it can be concluded that multiple seeds of DDPG policy does not train well, while TD3 and SAC works well for all agents. We observe variance in training with SAC due to different α values suggesting that the agent is difficult to train with certain α.



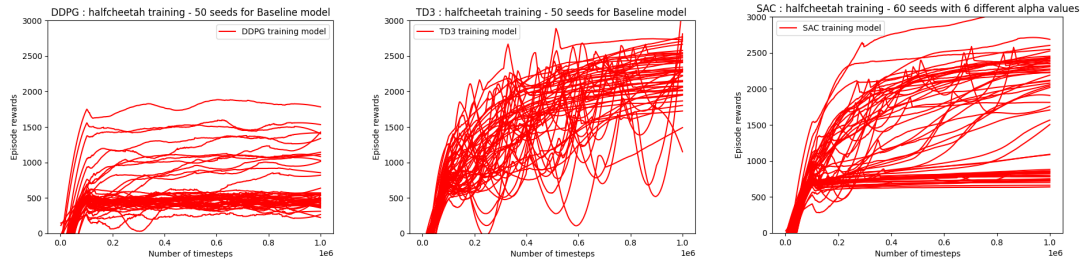
(a) Training with DDPG- 50 seeds (b) Training with TD3- 50 seeds (c) Training with SAC- 60 seeds

FIGURE 8.1: Hopper training results with different RL policies



(a) Training with DDPG- 50 seeds (b) Training with TD3- 50 seeds (c) Training with SAC- 60 seeds

FIGURE 8.2: Walker training results with different RL policies



(a) Training with DDPG- 50 seeds (b) Training with TD3- 50 seeds (c) Training with SAC- 60 seeds

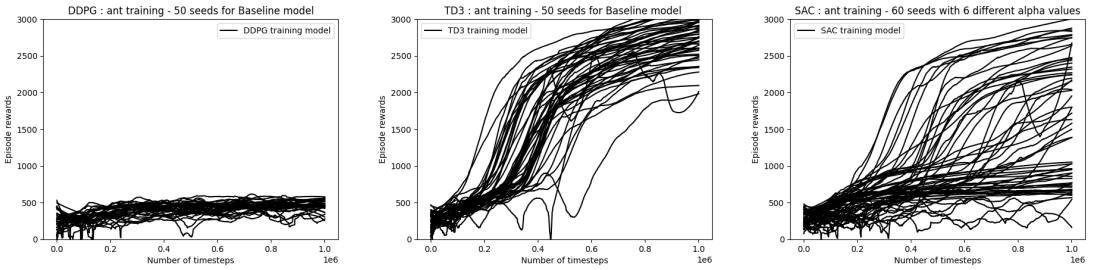
FIGURE 8.3: Halfcheetah training results with different RL policies

8.2.2 Step 2 - Evaluation of trained RL policies on novel perturbed environments

Hopper

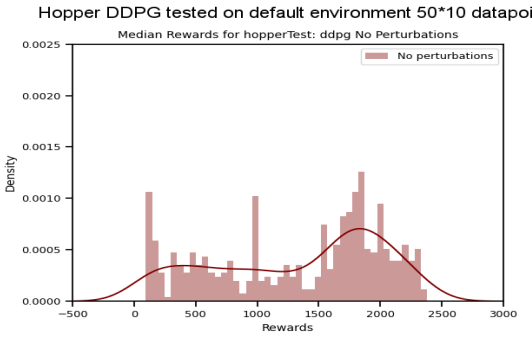
Several trained models of RL policies starting with DDPG in figure (8.5) are evaluated on non-perturbed (the baseline env. for reference) and novel perturbed environments. We evaluate on 200 perturbed envs. for each value of standard deviation. Results corresponding to TD3 and SAC are visible in figures (8.6) and (8.7) respectively.

In general, the performance of RL policies degrade with increase in standard deviation value. The performance with DDPG is not good on perturbed envs. (figure (8.5)) as

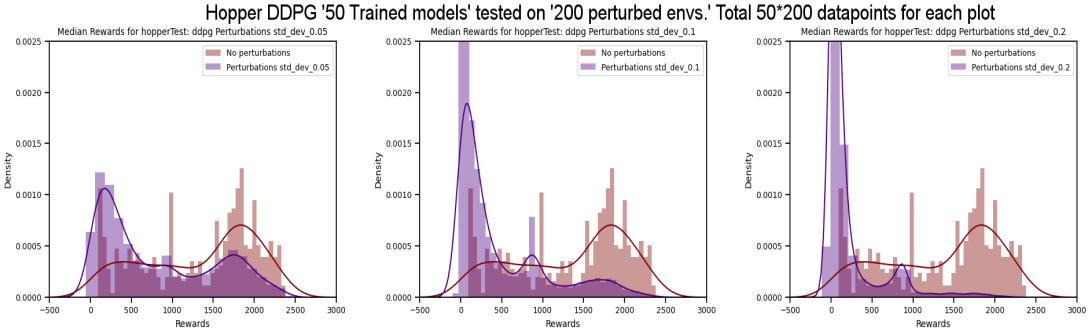


(a) Training with DDPG- 50 seeds (b) Training with TD3- 50 seeds (c) Training with SAC- 60 seeds

FIGURE 8.4: Ant training results with different RL policies



(a) Testing on non-perturbed hopper parameters (baseline)

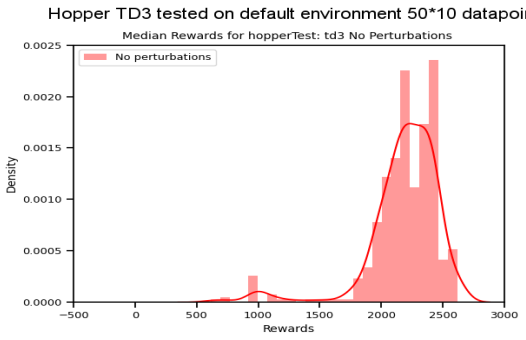


(b) Testing on perturbed hopper parameters with standard deviations of 0.05, 0.1 and 0.2 (with baseline as reference)

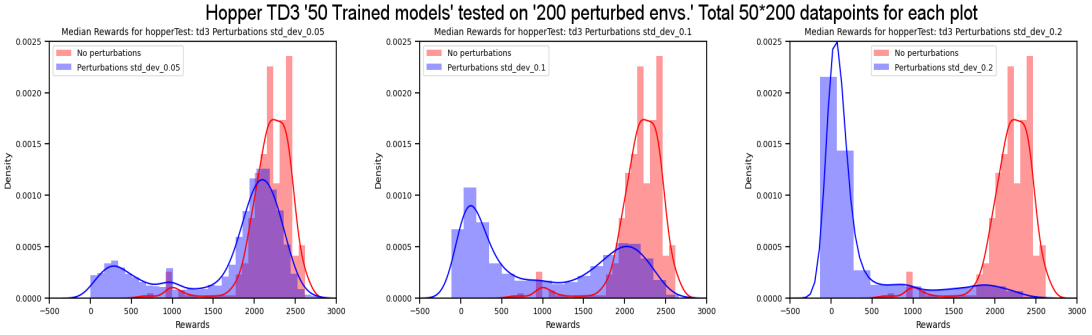
FIGURE 8.5: Testing trained DDPG models on non-perturbed and perturbed Hopper parameters

compared on TD3 and SAC (figures (8.6) and (8.7)), this is due to poor training of DDPG models over baseline hopper environment as seen in figure (8.1). This means that data generated from DDPG evaluation cannot be used for training SSL model for step 3. Thus, for the other three agents - Walker2D, HalfCheetah and Ant, we do not evaluate their DDPG models on perturbed envs. and instead only consider TD3 and SAC evaluation.

The SAC evaluation on perturbed envs. in figure (8.7) is a combined result of six α values i.e 0.01, 0.033, 0.05, 0.1, 0.2, and 0.33. Figure (8.8) shows the performance of specific α over three different standard deviation values. As observed in figure (8.8), all



(a) Testing on non-perturbed hopper parameters (baseline)



(b) Testing on perturbed hopper parameters with standard deviations of 0.05, 0.1 and 0.2 (with baseline as reference)

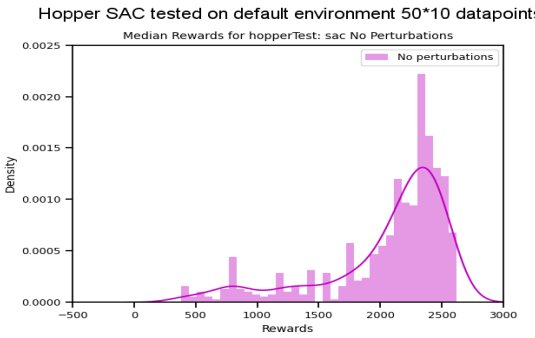
FIGURE 8.6: Testing trained TD3 models on non-perturbed and perturbed Hopper parameters

α parameters follow similar performance pattern for each case of standard deviation. No drastic difference in performance is visible for any specific value of α. This imply that SAC policy in the case of Hopper is not affected much by changing the α value.

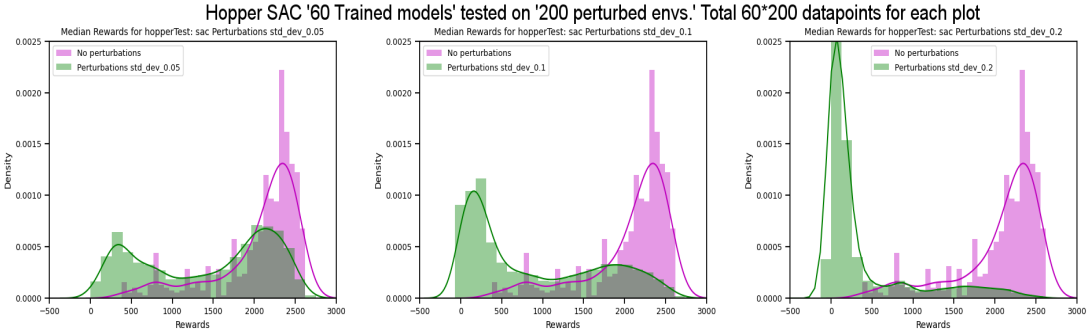
Walker2D

Trained models of TD3 and SAC policies are evaluated on non-perturbed (the baseline env. for reference) and novel perturbed environments as seen in figures (A.1) and (A.2) respectively. While the performance of TD3 is quite good, some SAC models lacked performance typically because those were trained with some α values not suitable for Walker2D. In general the data is sufficiently good enough to be used for SSL training.

The SAC evaluation results on perturbed envs. consisting of various α values are presented in figure (A.3). As observed, four α parameters follow similar pattern in performance, except for $\alpha = 0.2$ and $\alpha = 0.33$ those illustrate poor performance. Also, it can be noted that $\alpha = 0.2$ and $\alpha = 0.33$ limit the performance within a certain reward range. Thus, for the Walker2D we observe a drastic difference in performance for two values of α.



(a) Testing on non-perturbed hopper parameters (baseline)



(b) Testing on perturbed hopper parameters with standard deviations of 0.05, 0.1 and 0.2 (with baseline as reference)

FIGURE 8.7: Testing trained SAC models on non-perturbed and perturbed Hopper parameters

Halfcheetah

TD3 and SAC trained models are evaluated on non-perturbed and novel perturbed environments as seen in figures (A.4) and (A.5) respectively. TD3 performance is quite good, while some SAC models do not perform well, we infer that training with some α values was not suitable for HalfCheetah. Generally, the generated data is good enough to be considered for SSL training.

Figure (A.6) presents SAC evaluations corresponding to different α values on perturbed environments. It can be noted that $\alpha = 0.01, 0.033, 0.05$ follow similar pattern in performance, while $\alpha = 0.2, 0.33$ illustrate poor performance in a certain range. However, the results for $\alpha = 0.1$ do not vary much across different standard deviation values. Thus, for the walker2D agent we observe a drastic performance difference for three α values.

Ant

Figures (A.7) and (A.8) represent the evaluation of TD3 and SAC trained models on non-perturbed and novel perturbed environments. Performance of TD3 policy is quite

good, while we observe variance in SAC performance. This variance is likely due to training with certain α values that are not suitable for Ant. Moreover, the amount of generated data can considered for SSL training.

SAC evaluations corresponding to different α values on perturbed environments can be viewed in figure (A.9). For $\alpha = 0.2, 0.33$ the reward values are confined to particular range across all plots. However, $\alpha = 0.01, 0.033, 0.05$ seems to follow regular pattern in performance, while $\alpha = 0.1$ does not seems to have a regular pattern. Thus, a drastic difference in Ant performance can be observed across plots for different α values.

8.2.3 Step 3 - Training the SSL model using the data from N most suitable environments

The two trained variants of wav2vec model with receptive fields 16 and 32 in the case of Hopper are presented in figure (A.10). In general, the loss function drastically decreases and saturates at a low value against the training steps for both the training and validation datasets. This imply appropriate tuning of hyperparameters for the model.

Also, figure (A.10) shows the accuracy of the model to predict latents in the future from 1 to 14 latent steps for Hopper. Similar result for Walker2D and HalfCheetah can be viewed in figure (A.11). These plots demonstrate that the objective is neither impossible nor trivial. As expected the prediction accuracy falls with the number of latent steps implying that prediction task becomes harder as the target is further ahead.

8.2.4 Step 4 - Integrating the non-invariant state representations and re-training the RL policy

The non-invariant state representations obtained from the SSL model are integrated back to the baseline TD3 policy. We perform re-training of the TD3 policy in two cases, firstly, re-training only over the obtained SSL representations, and secondly, re-training over the integrated representations along with the last state vector of the agent.

The results of training 10 seeds for these two cases for Hopper (for two variants of receptive fields) are presented in figure (A.12) and for all agents are presented in figure (A.13). In general, re-training just over the obtained SSL representations do not yield expected results, this can be due to limited training of the SSL model. While, the performance of integrated representations and one state model is good enough.

Figure (A.14) presents the comparisons between 10 seeds of different models for Hopper agent with TD3 policy. Here, the 1 state model is the baseline model wherein we use

one state vector (obtained from simulator) for training the TD3 policy. While we also considered training with last 8 states and 16 states in order to evaluate the role of training with larger memory of state vectors. The other models considered are the ones from figure (A.12). It can be seen that the performance of 8 state and 16 state models are bad as compared to baseline 1 state model. Thus, having more state vectors in memory does not result in good performance. A potential reason is the need of larger neural network architecture when we consider more state vectors. Of all the considered models, the best performance has been obtained by integrated representations and 1 state model. It is worth noticing that this model also reduces variance in training as compared to the baseline 1 state model.

8.2.5 Step 5 - Evaluating the re-trained RL policy over new perturbed environments

In the previous step, the results of re-trained TD3 policy models with integrated SSL representations for Hopper agent were presented. In this step we evaluate these trained models over the newly sampled perturbed environments according to the criteria mentioned in section (7.2). While we also evaluate these models over the same perturbed envs. that were considered to generate the training data in step 3, these envs. are termed as 'old' perturbed envs. for reference. The evaluation results over these 'old' and 'new' perturbed envs. are presented in figure (A.15). It is surprising that the performance is similar in both cases and thus we conclude that their is little to no robustness gain with this evaluation step.

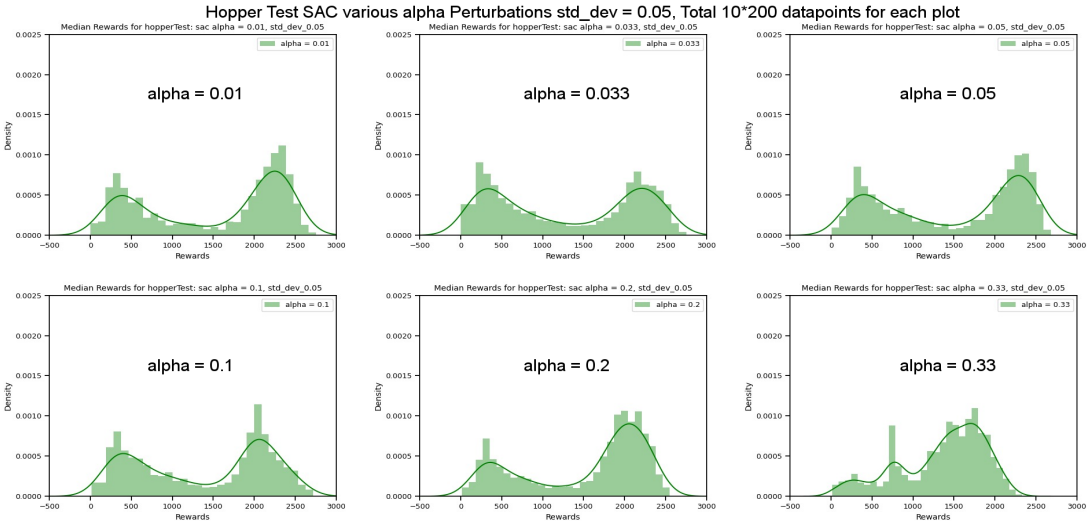
8.2.6 Step 6 - Comparing the baseline RL policy performance over perturbed environments with and without integrated SSL representations

In step 5 we introduced the 'new' perturbed envs. and evaluated the performance of re-trained TD3 policy models with integrated SSL representations for Hopper agent. In this step we compare the evaluation performance of 1 state and 8 state models from step 4 over these 'new' envs. The comparison results are presented in figure (A.16). Due to poor training with 8 state models in step 4, their evaluation performance is bad (A.16(b)) compared to the other models. While, their is no significant difference when we compare evaluations from 1 state model (A.16(a)) and integrated representations and 1 state model (A.16(c)). Thus, we conclude by saying that their is little to no robustness gain before and after integration of the representations in this evaluation step.

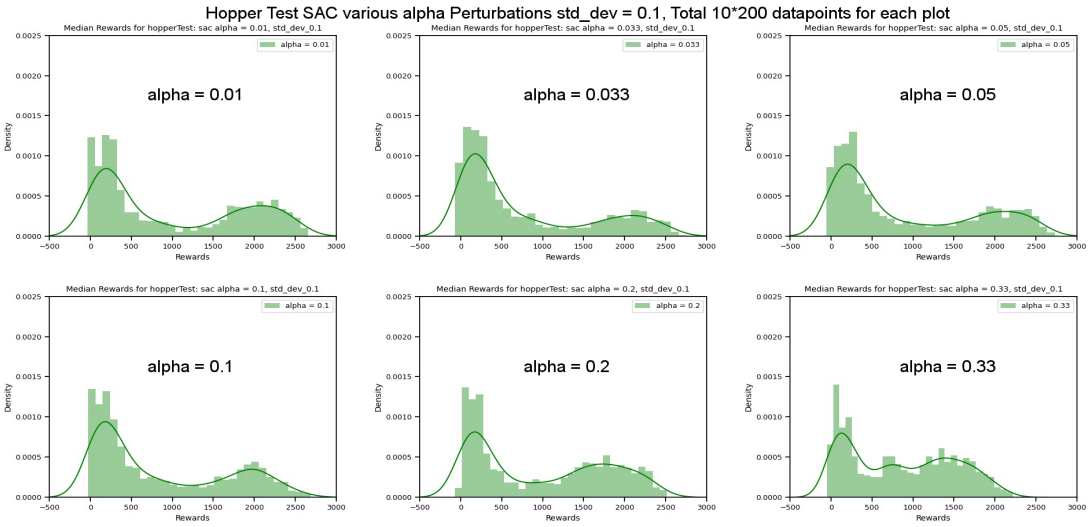
8.3 Summary

The chapter provided results for different experimental steps. Based on the results, it is visible that increasing the level of perturbations (by increasing the standard deviation while sampling) correlates with the degradation in rewards. For the SAC policy, the α value affects the rewards degradation over perturbed envs. to limited extent for Hopper and Walker2D as they are unstable envs. with respect to the morphological structure, while to an large extent in Half-cheetah and Ant as these are stable environments.

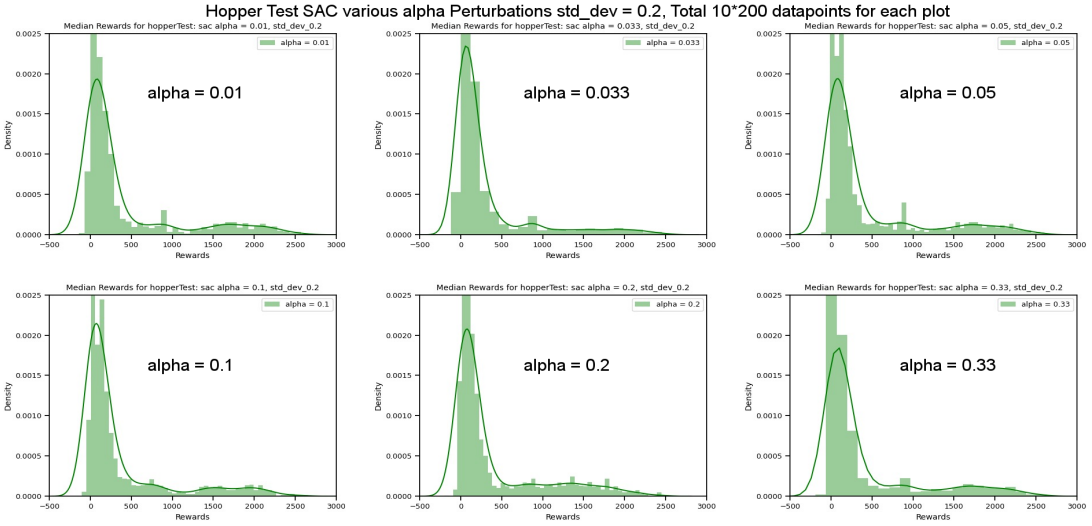
The SSL model was trained well due to a drastic drop in the model loss function. It was observed that the integrated representations helped in reducing the variance during the re-training of RL policy models. While there was no significant difference when evaluating the trained models over several perturbed envs. with and without integrated SSL representations. Thus, we acquire no to little robustness gain during this experimental design process.



(a) Testing different alpha models on perturbed Hopper parameters with standard deviations of 0.05



(b) Testing different alpha models on perturbed Hopper parameters with standard deviations of 0.1



(c) Testing different alpha models on perturbed Hopper parameters with standard deviations of 0.2

FIGURE 8.8: Testing SAC models with different alpha values on perturbed Hopper parameters

Chapter 9

Conclusion

9.1 Discussion

Robots and artificial agents need to adapt to the intrinsic or extrinsic changes while performing different tasks in their respective environments. Reinforcement learning (RL) can be used to develop adaptive controller for an agent subjected to these changes. Accordingly, the first goal of this thesis was to investigate the robustness of RL policies against intrinsic changes in agent's morphological structure. The second goal was to identify whether the self-supervised representation learning can improve the robustness of considered RL policies.

In the beginning, thesis provides an theoretical background for understanding RL in general, defining the necessary terminologies with the mathematical equations and introducing model free and model based methods. Next, the basic building blocks of Deep Neural Networks (DNN) were presented alongside with their architecture and training techniques. Later, the relevant deep RL policies were described in detail with respect to theory, mathematical expressions, pseudo codes and model hyperparameters. The thesis also provides details on the physics engine used for simulation and the continuous state space agents used for experimentation. Lastly, Self-Supervised Learning (SSL) model was introduced with its theory, model parameters and mathematical loss function.

9.2 Conclusion

In order to achieve thesis goals, a step by step experimental pipeline is proposed in chapter (7). The results corresponding to first goal are presented in sections (8.2.1) and (8.2.2). Wherein we find that the performance of baseline RL policies degrade as

we increase the distribution shift in agent's physical morphology. Next, to achieve the second thesis goal, we trained the SSL model for extracting the feature representations from agent's state space dataset. While we did not find evidence of a performance improvement when using SSL representations, some initial evidence points towards a reduced variance in the obtained reward as described in section (8.2.4).

Later, we investigated for any robustness gain after integrating the representations with RL policies in section (8.2.5). We find that there was not much difference when evaluating RL policies over newly sampled perturbed environments with and without integrated representations. Thus, we conclude that there was hardly any robustness gain achieved via experimental design step 5. Lastly, in section (8.2.6) we compared performance of different trained models over new perturbed envs. and hardly find any robustness gain in this step.

In summary, the thesis examined the performance of actor-critic methods DDPG, TD3 and SAC for different RL agents subjected to morphological changes in body structure. We find an correlation between obtained rewards and agent's morphology. Further, with SSL representations we find an improvement in training RL policies while we do not achieve robustness over perturbed envs. for these policies. At last, we expect that exploring self-supervision for developing adaptive RL mechanisms is an exiting research direction.

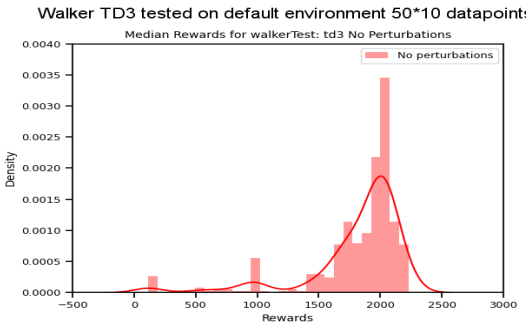
9.3 Directions for the future work

The thesis presented an approach to integrate self-supervision with RL. The extensions of this work can be the implemented actor-critic methods - DDPG, TD3 and SAC can be fine tuned further with optimal hyperparameters. Also, new experiments can be undertaken with increased capacity of these baseline networks. The SSL model- wav2vec used for the course of this thesis was originally developed for speech processing, it might be interesting to test using the latest developments in SSL. Robustness evaluation of other baseline RL policies such as PPO[87], ACER[88], A2C[89], TRPO[90], GAIL[91] can be undertaken along with integration of SSL. Experiments can be done with more complex continuous state space RL agents such as Humanoid, Quadruped and Dog from dm control suite[92] or with real mobile robots such as bipeds, quadrupeds or hexapods.

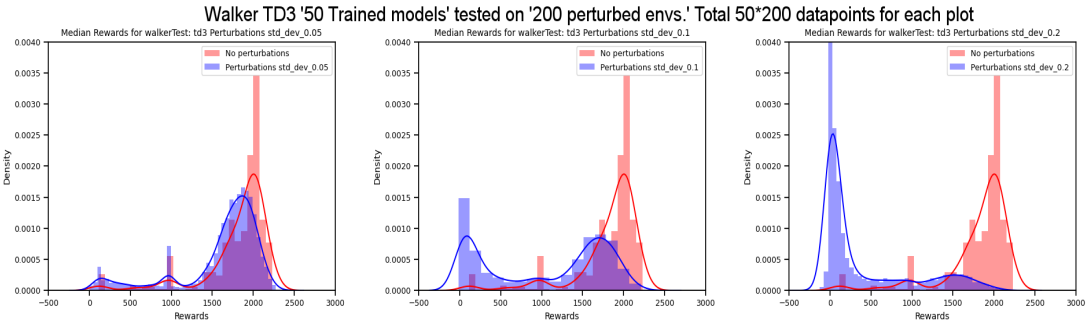
Appendix A

Supplementary results

This section includes additional figures in correspondence to chapter 8.

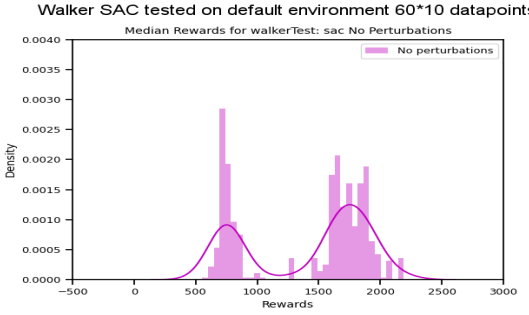


(a) Testing on non-perturbed Walker2D parameters (baseline)

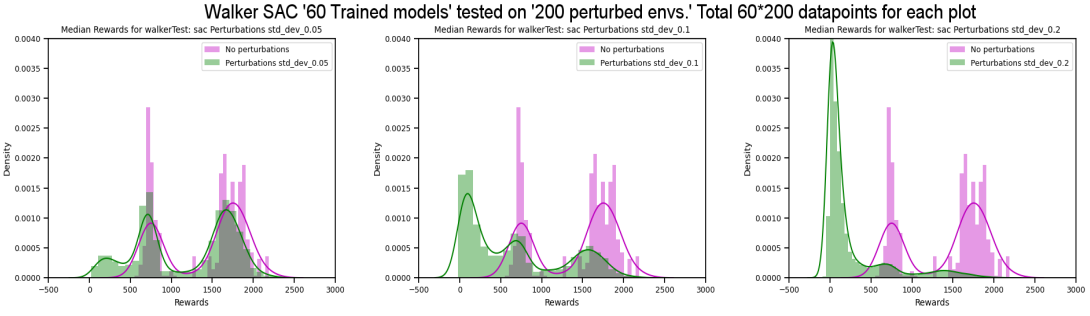


(b) Testing on perturbed Walker2D parameters with standard deviations of 0.05, 0.1 and 0.2 (with baseline as reference)

FIGURE A.1: Testing trained TD3 models on non-perturbed and perturbed Walker2D parameters

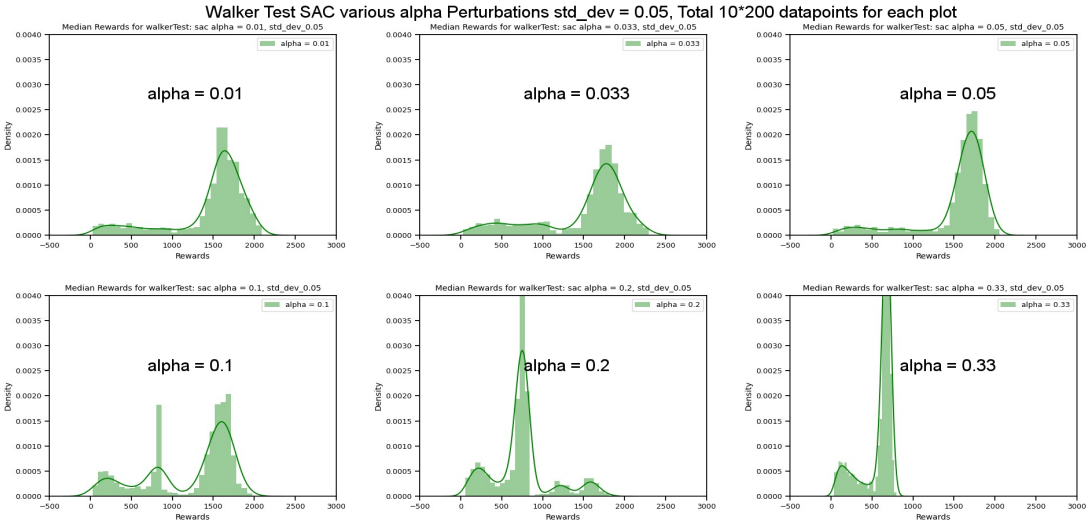


(a) Testing on non-perturbed Walker2D parameters (baseline)

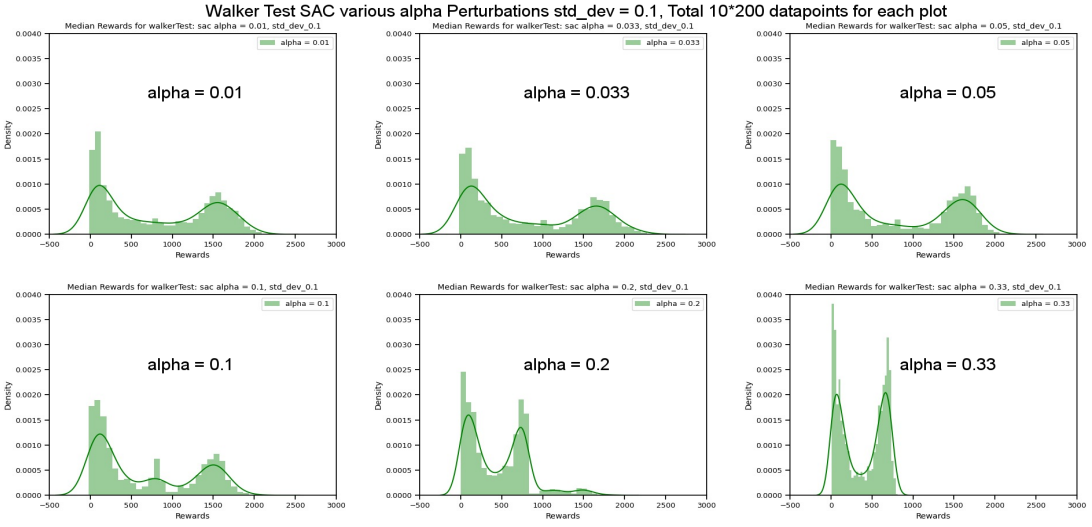


(b) Testing on perturbed Walker2D parameters with standard deviations of 0.05, 0.1 and 0.2 (with baseline as reference)

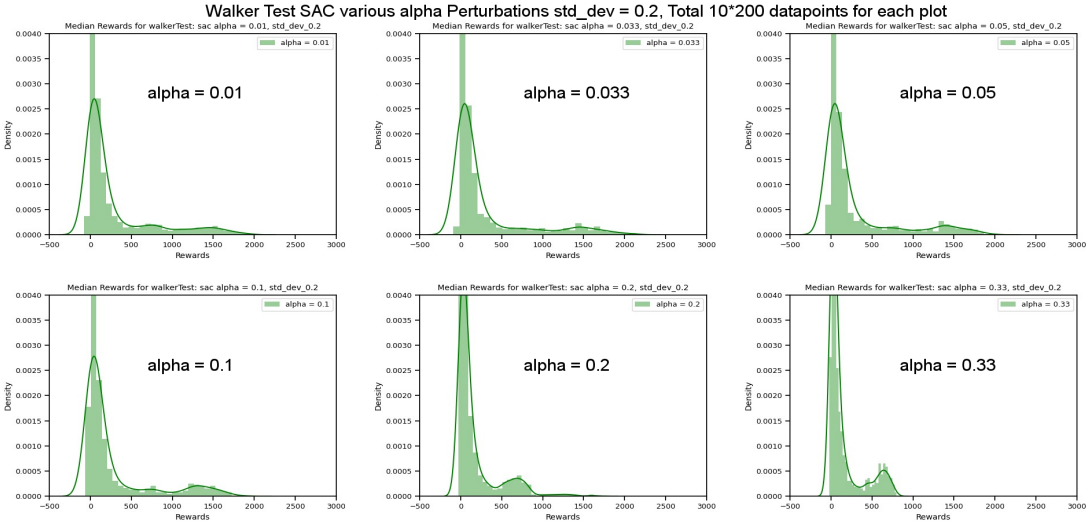
FIGURE A.2: Testing trained SAC models with different alpha values on perturbed Walker2D parameters



(a) Testing different alpha models on perturbed Walker2D parameters with standard deviations of 0.05

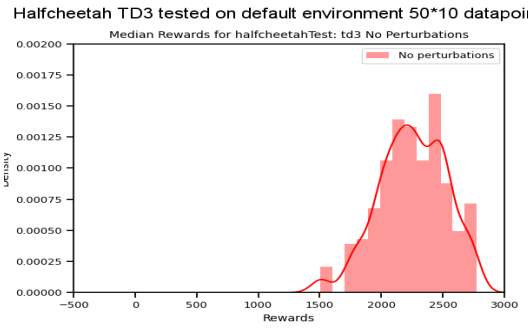


(b) Testing different alpha models on perturbed Walker2D parameters with standard deviations of 0.1

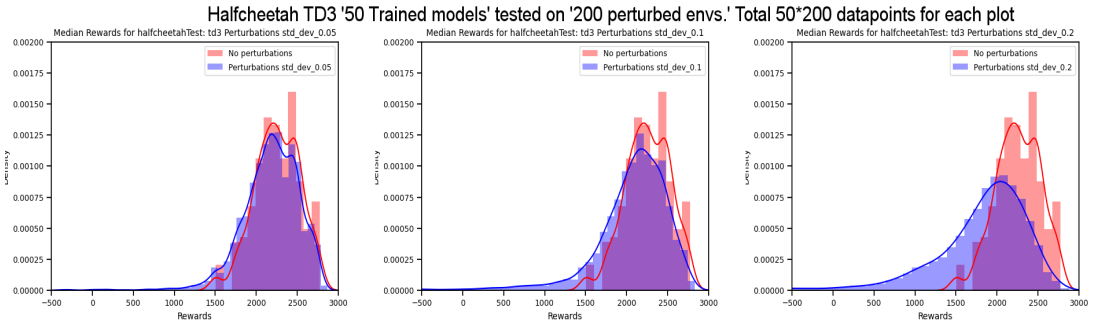


(c) Testing different alpha models on perturbed Walker2D parameters with standard deviations of 0.2

FIGURE A.3: Testing SAC models with different alpha values on perturbed Walker2D parameters

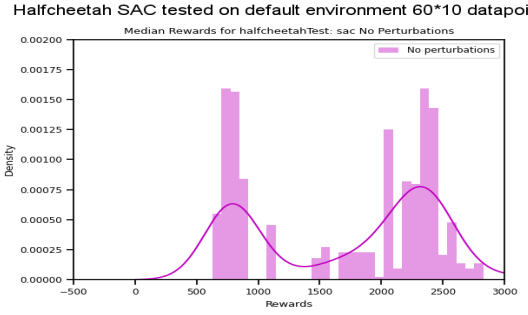


(a) Testing on non-perturbed Halfcheetah parameters (baseline)

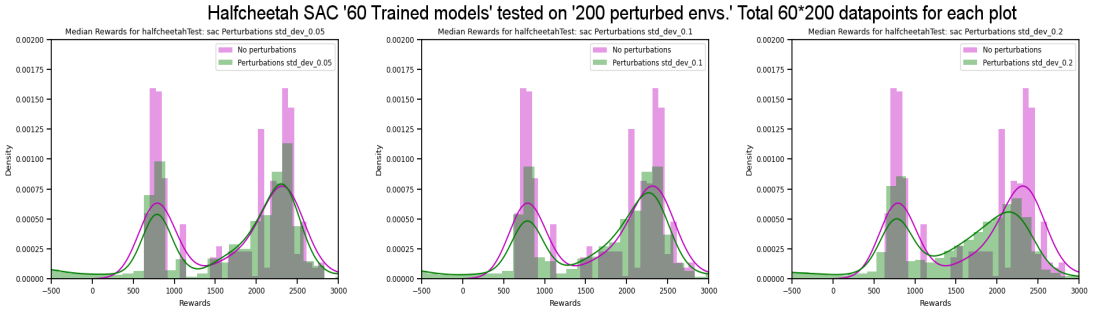


(b) Testing on perturbed Halfcheetah parameters with standard deviations of 0.05, 0.1 and 0.2 (with baseline as reference)

FIGURE A.4: Testing trained TD3 models on non-perturbed and perturbed Halfcheetah parameters

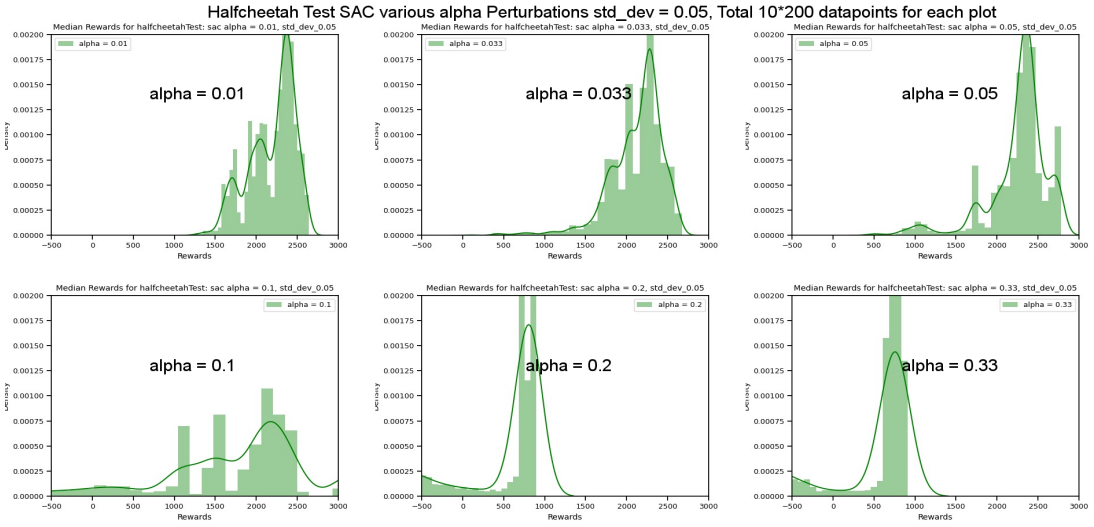


(a) Testing on non-perturbed Halfcheetah parameters (baseline)

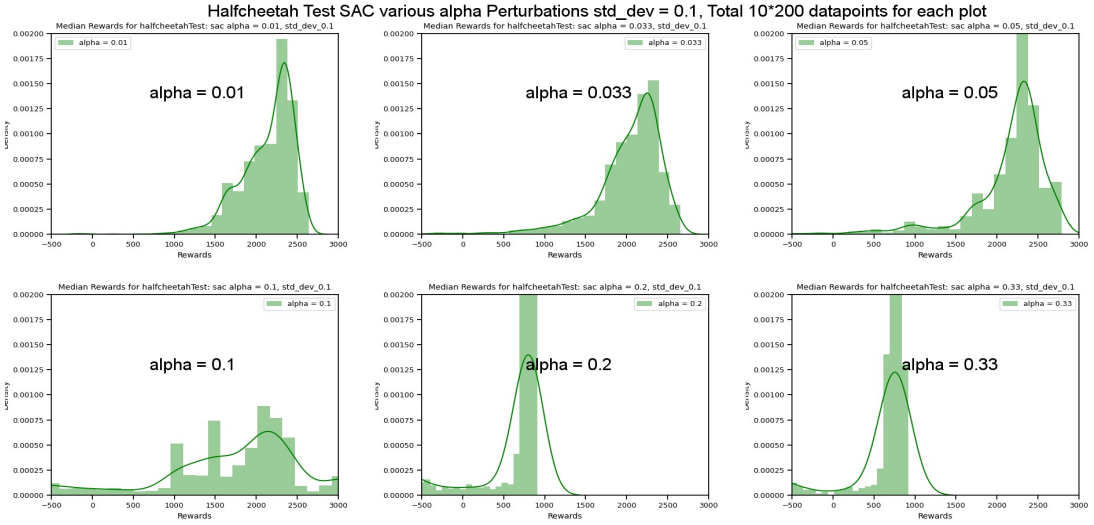


(b) Testing on perturbed Halfcheetah parameters with standard deviations of 0.05, 0.1 and 0.2 (with baseline as reference)

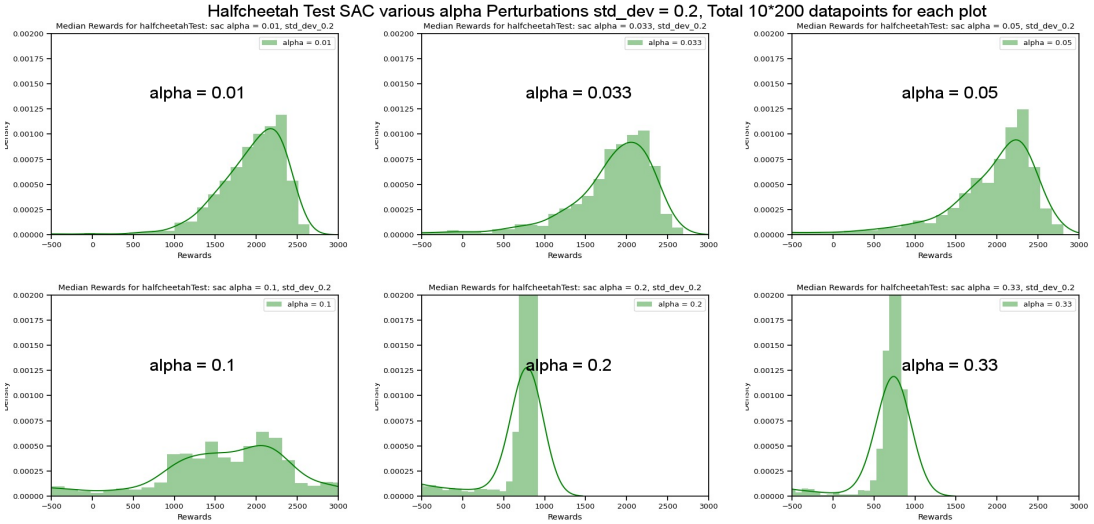
FIGURE A.5: Testing trained SAC models on non-perturbed and perturbed Halfcheetah parameters



(a) Testing different alpha models on perturbed Halfcheetah parameters with standard deviations of 0.05

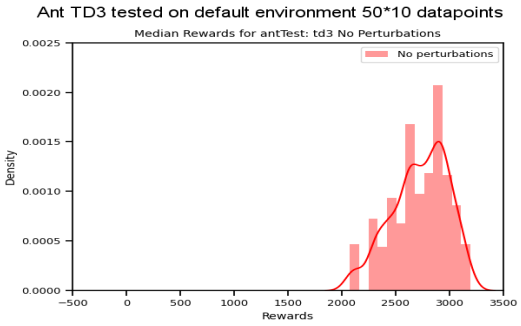


(b) Testing different alpha models on perturbed Halfcheetah parameters with standard deviations of 0.1

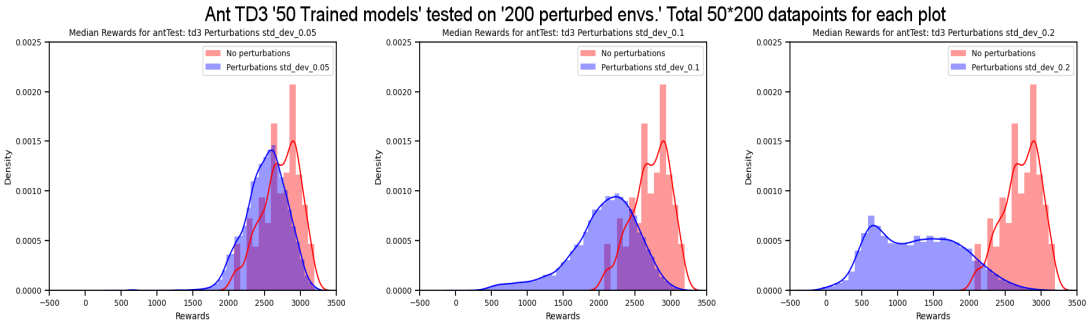


(c) Testing different alpha models on perturbed Halfcheetah parameters with standard deviations of 0.2

FIGURE A.6: Testing SAC models with different alpha values on perturbed Halfcheetah parameters

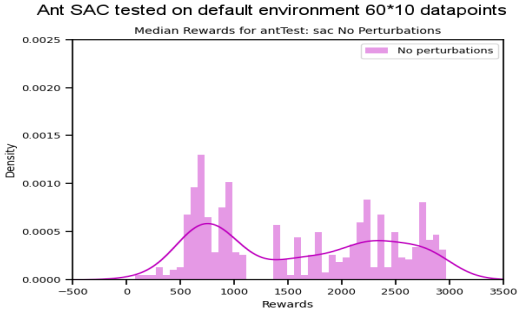


(a) Testing on non-perturbed Ant parameters (baseline)

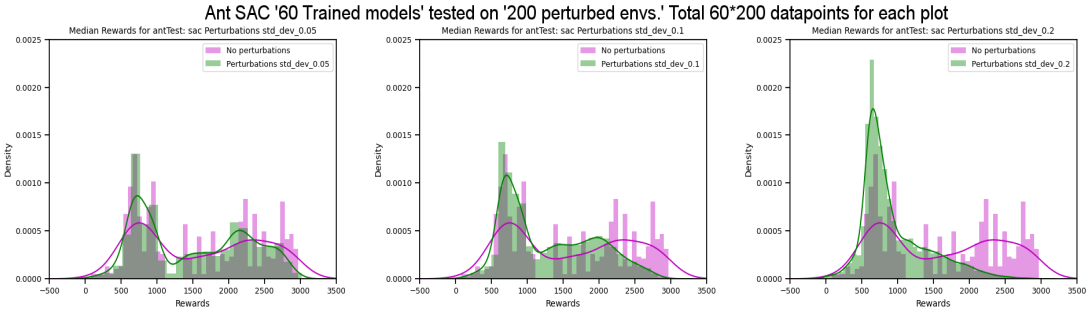


(b) Testing on perturbed Ant parameters with standard deviations of 0.05, 0.1 and 0.2 (with baseline as reference)

FIGURE A.7: Testing trained TD3 models on non-perturbed and perturbed Ant parameters

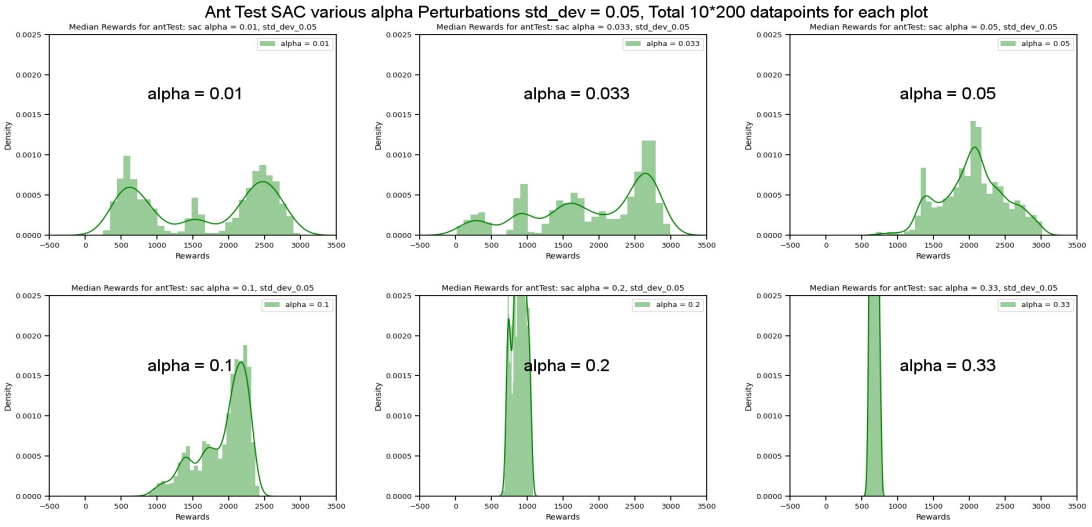


(a) Testing on non-perturbed Ant parameters (baseline)

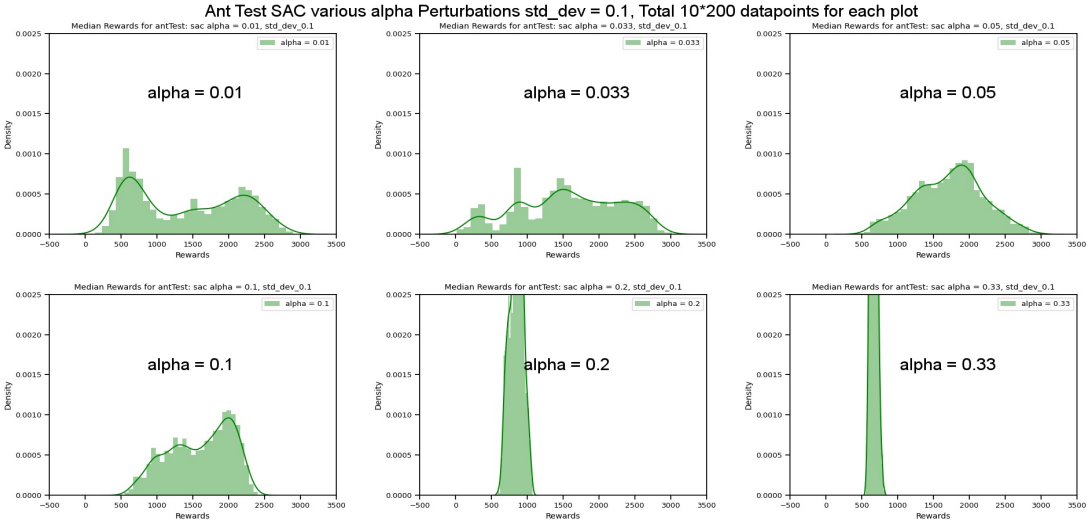


(b) Testing on perturbed Ant parameters with standard deviations of 0.05, 0.1 and 0.2 (with baseline as reference)

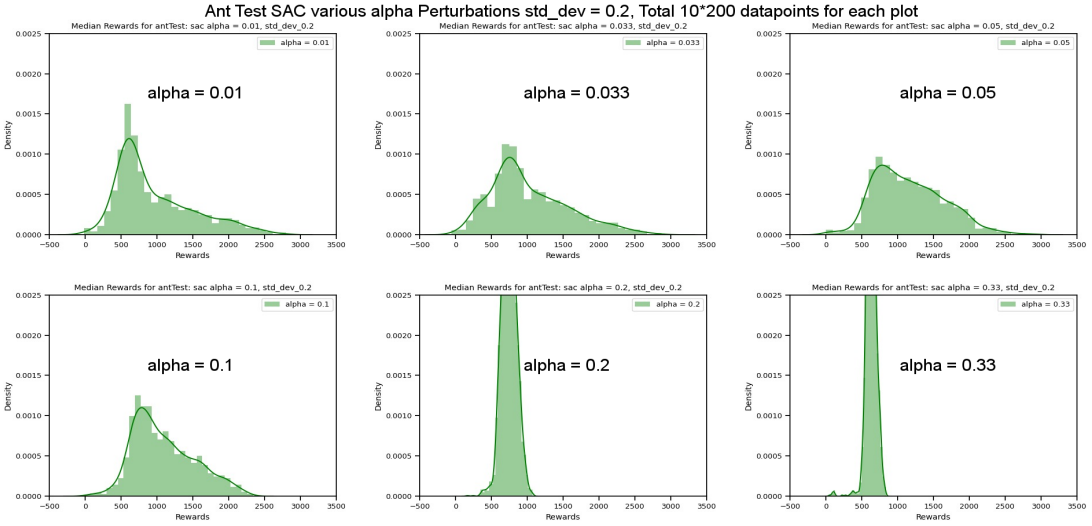
FIGURE A.8: Testing trained SAC models on non-perturbed and perturbed Ant parameters



(a) Testing different alpha models on perturbed Ant parameters with standard deviations of 0.05

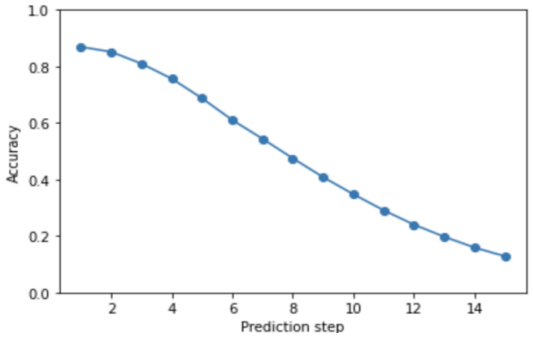


(b) Testing different alpha models on perturbed Ant parameters with standard deviations of 0.1

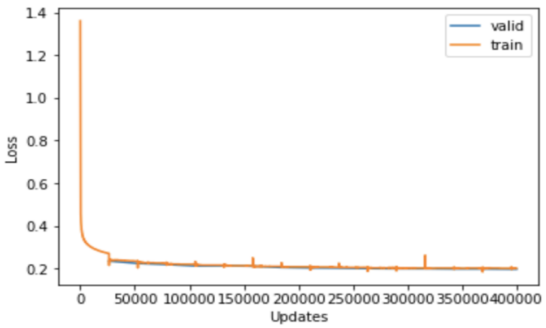


(c) Testing different alpha models on perturbed Ant parameters with standard deviations of 0.2

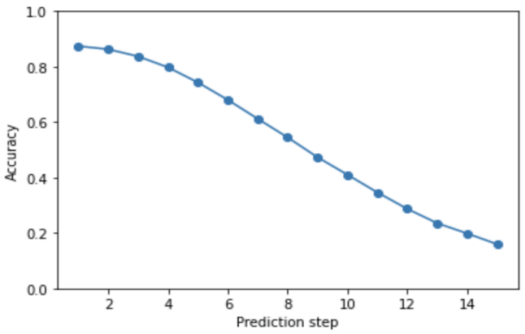
FIGURE A.9: Testing SAC models with different alpha values on perturbed Ant parameters



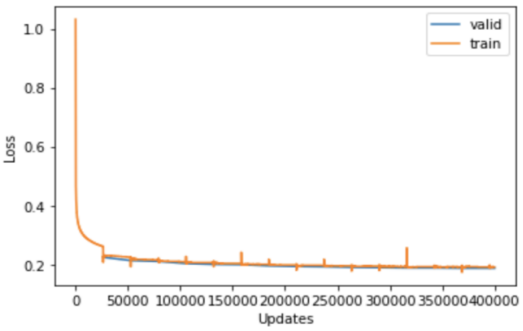
(a) Wav2vec average accuracy to predict the latents in future from 1 to 14 latent steps with receptive field 16



(b) Wav2vec training and validation loss with receptive field 16

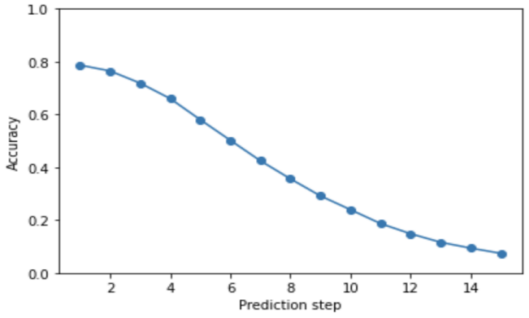


(c) Wav2vec average accuracy to predict the latents in future from 1 to 14 latent steps with receptive field 32

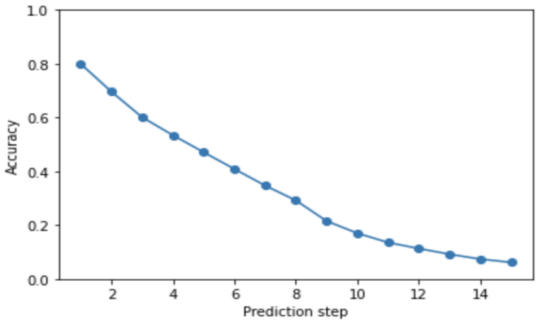


(d) Wav2vec training and validation loss with receptive field 32

FIGURE A.10: Wav2vec model plots for average accuracy of predicting future latents and training, validation loss functions with two receptive fields for Hopper

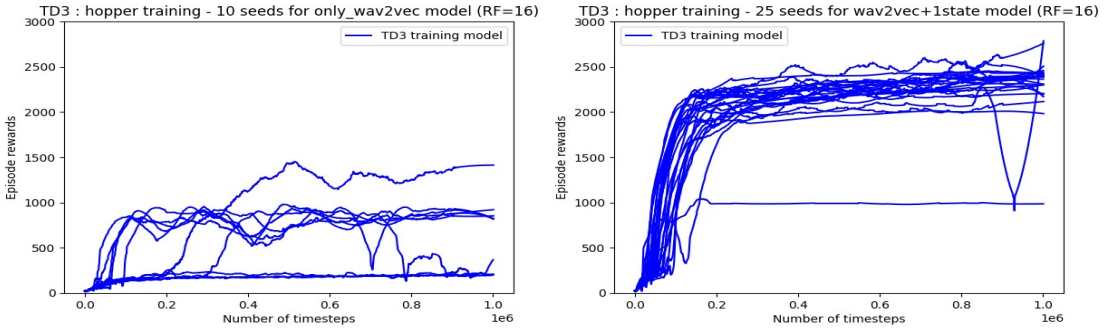


(a) Wav2vec average accuracy to predict the latents in future from 1 to 14 latent steps with receptive field 16 - Walker2D

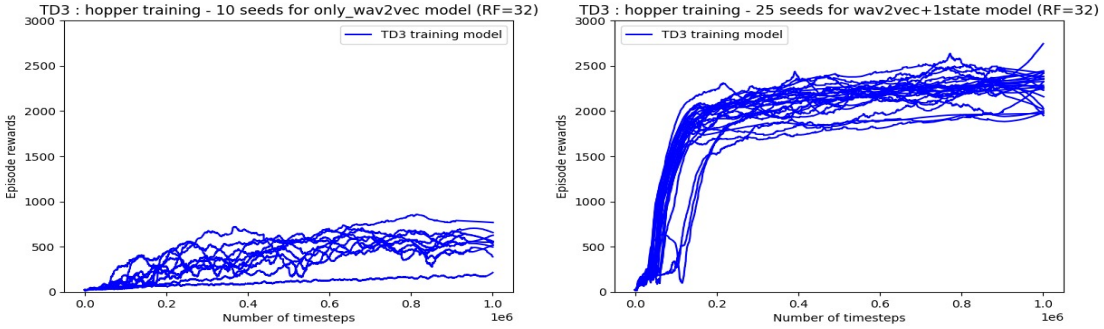


(b) Wav2vec average accuracy to predict the latents in future from 1 to 14 latent steps with receptive fiel 16 - Halfcheetah

FIGURE A.11: Wav2vec model plots for average accuracy of predicting future latents for Walker2D and Halfcheetah

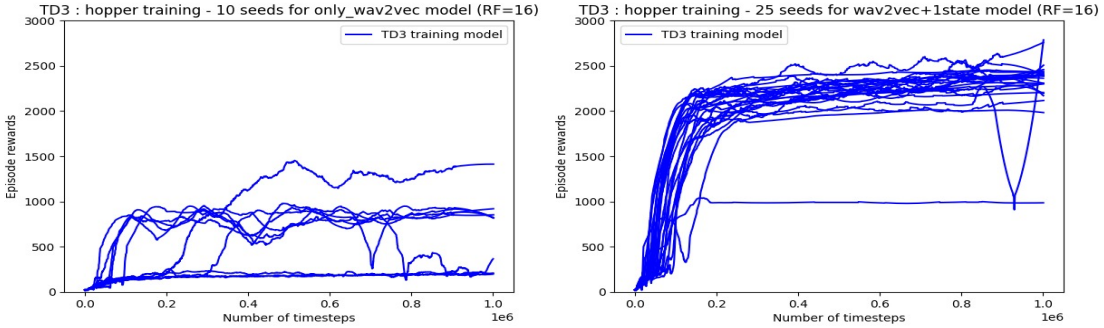


(a) TD3 training with integrated 'only wav2vec' and 'wav2vec+ 1state' with receptive field 16

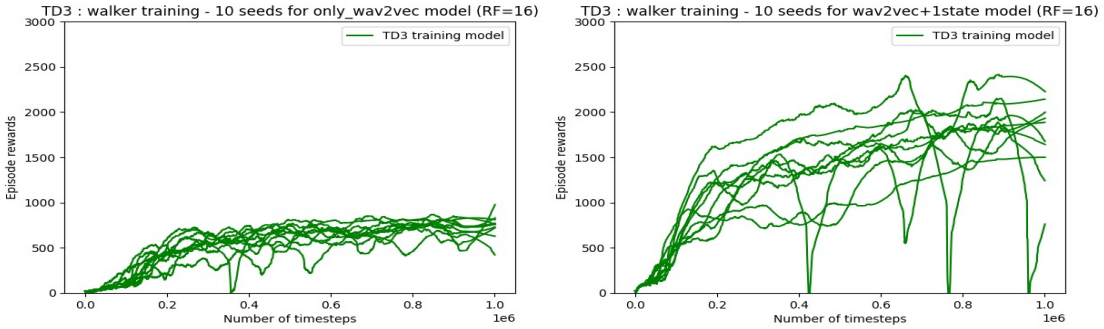


(b) TD3 training with integrated 'only wav2vec' and 'wav2vec+ 1state' with receptive field 32

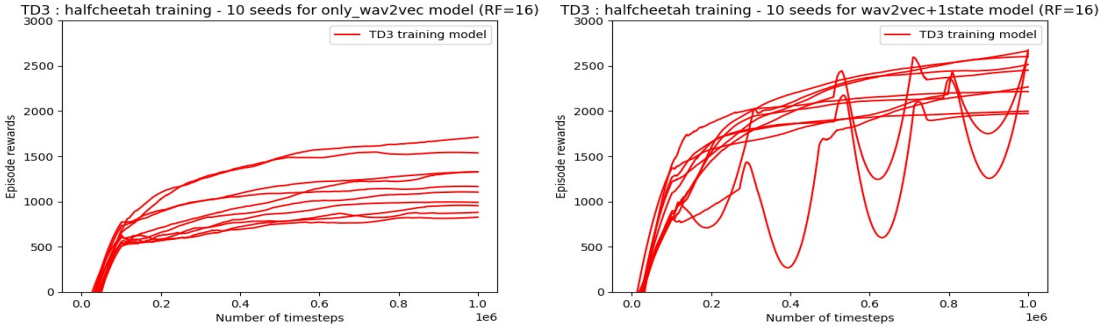
FIGURE A.12: Comparing TD3 training for two receptive fields with integrated 'only wav2vec' and 'wav2vec+ 1 state' models for Hopper



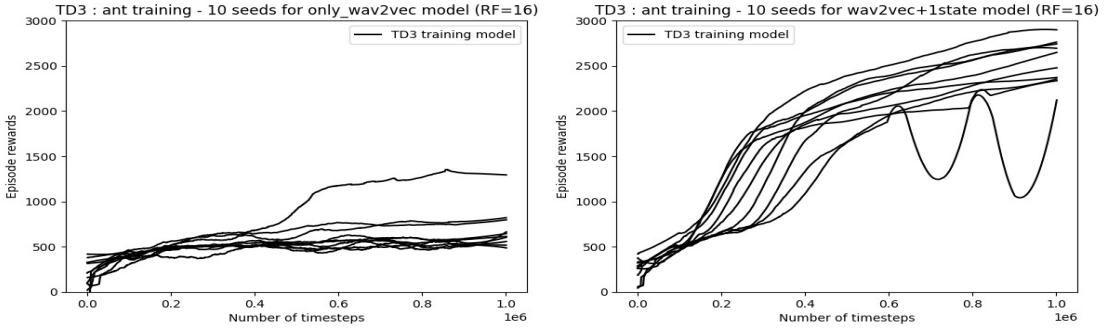
(a) TD3 training with integrated 'only wav2vec' (left) and 'wav2vec+ 1state' (right) for Hopper default parameters



(b) TD3 training with integrated 'only wav2vec' (left) and 'wav2vec+ 1state' (right) for Walker2D default parameters

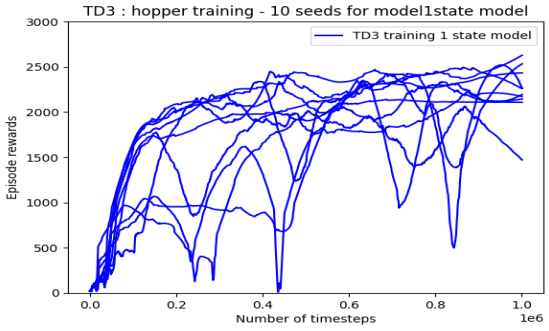


(c) TD3 training with integrated 'only wav2vec' (left) and 'wav2vec+ 1state' (right) for Halfcheetah default parameters

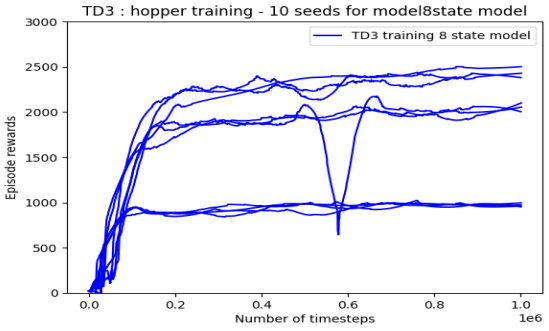


(d) TD3 training with integrated 'only wav2vec' (left) and 'wav2vec+ 1state' (right) for Ant default parameters

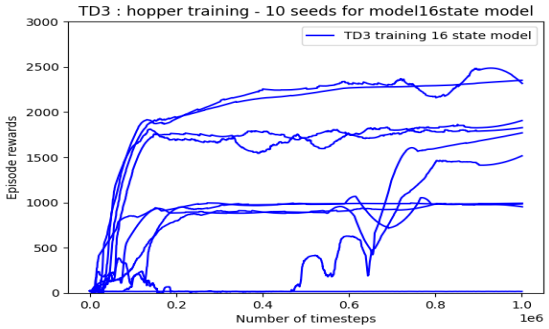
FIGURE A.13: Comparing TD3 training with integrated 'only wav2vec' and 'wav2vec+ 1 state' models for four agents



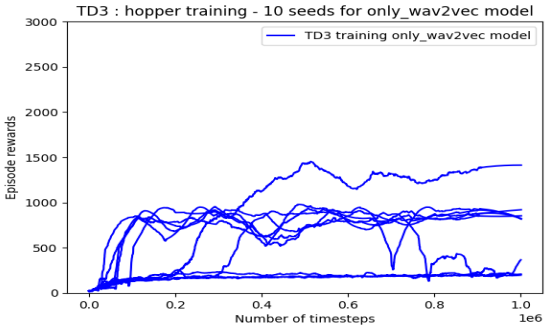
(a) TD3 training with integrated 1 state model



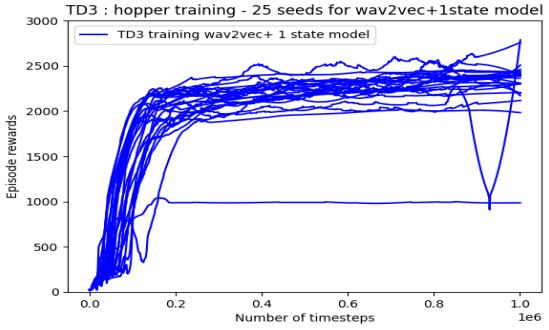
(b) TD3 training with integrated 8 state model



(c) TD3 training with integrated 16 state model

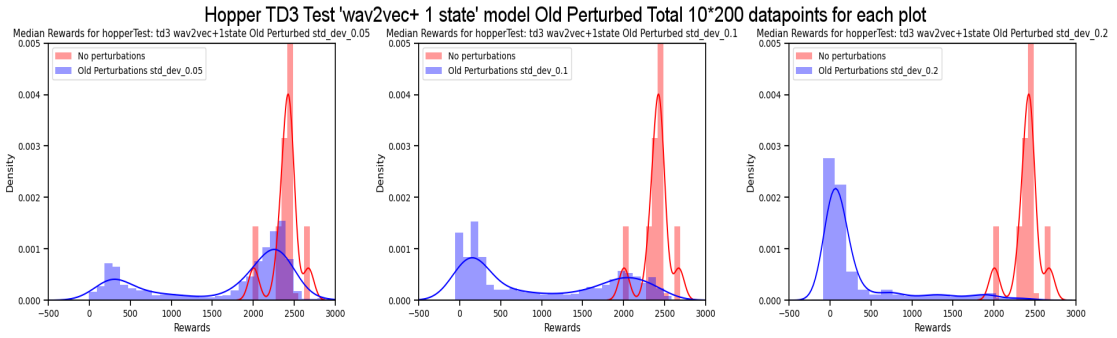


(d) TD3 training with integrated 'wav2vec' model

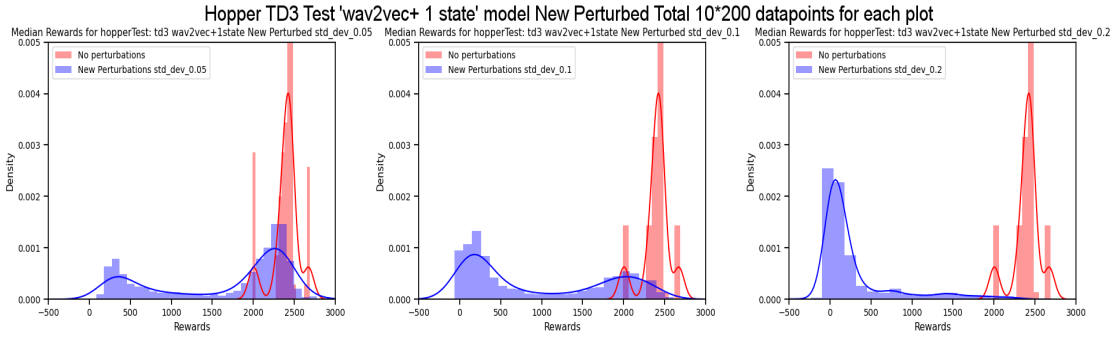


(e) TD3 training with integrated 'wav2vec+ 1 state' model

FIGURE A.14: Comparing TD3 training with different integrated models for Hopper

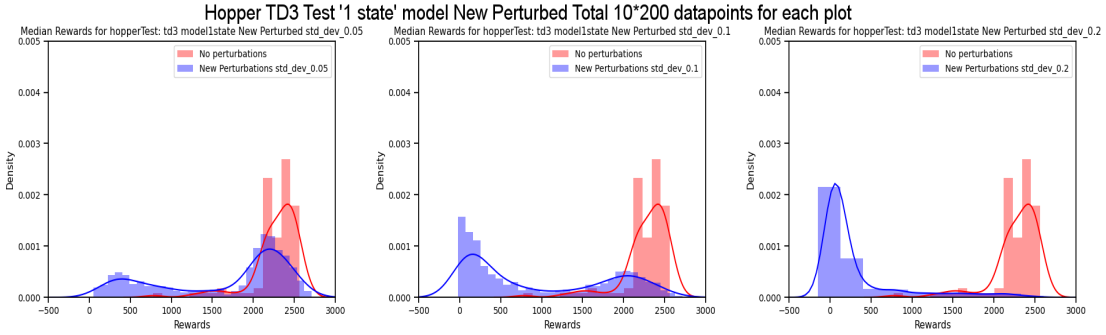


(a) Testing 'wav2vec + 1 state' models on Old perturbed Hopper parameters with standard deviations of 0.05, 0.1, 0.2

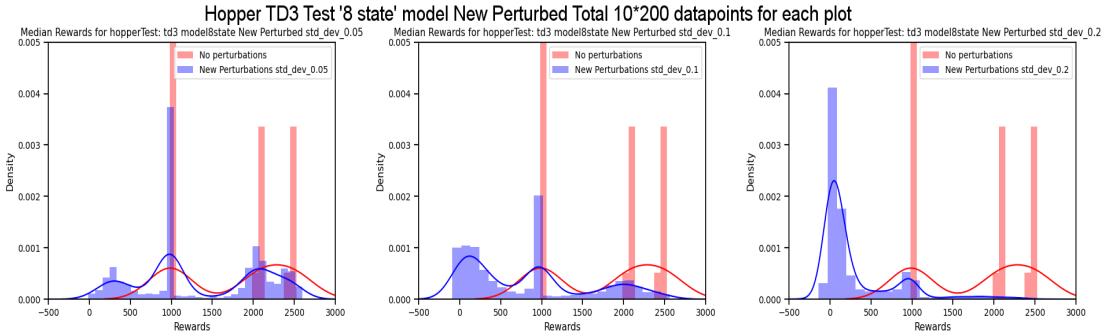


(b) Testing 'wav2vec + 1 state' models on New perturbed Hopper parameters with standard deviations of 0.05, 0.1, 0.2

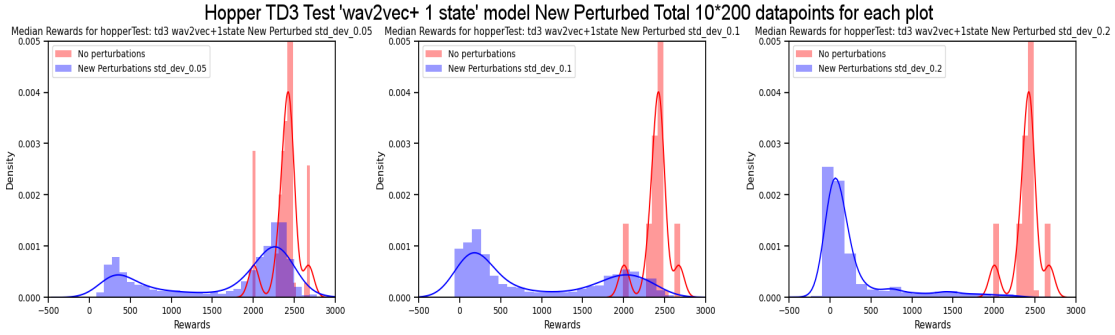
FIGURE A.15: Comparison between wav2vec + 1 state models tested on 'old' and 'new' perturbed Hopper parameters



(a) Testing '1 state' models on perturbed Hopper parameters with standard deviations of 0.05, 0.1, 0.2



(b) Testing '8 state' models on perturbed Hopper parameters with standard deviations of 0.05, 0.1, 0.2



(c) Testing 'wav2vec + 1 state' models on perturbed Hopper parameters with standard deviations of 0.05, 0.1, 0.2

FIGURE A.16: Comparison between three different models tested on perturbed Hopper parameters

Bibliography

[1] Abdulhadi Mohamed. Rl tutorial part 1: Monte carlo methods. https://plusreinforcement.com/2018/07/05/rl-tutorial-part-1-monte-carlo-methods/.

[2] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1st edition, 1998.

[3] integrate.ai. What is model-based reinforcement learning? https://medium.com/the-official-integrate-ai-blog/understanding-reinforcement-learning-93d4e34e5698, 2018.

[4] Mahdi Shariati, Mohammad Saeed Mafipour, Peyman Mehrabi, Alireza Bahadori, Yousef Zandi, Musab NA Salih, Hoang Nguyen, Jie Dou, Xuan Song, and Shek Poi-Ngian. Application of a hybrid artificial neural network-particle swarm optimization (ann-pso) model in behavior prediction of channel shear connectors embedded in normal and high-strength concrete. *Applied Sciences*, 9(24):5534, 2019.

[5] Imad Dabbura. Coding neural network — forward propagation and backpropagtion. https://towardsdatascience.com/coding-neural-network-forward-propagation-and-backpropagtion-ccf8cf369f76.

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[7] Shrinath Deshpande. How to train your cheetah with deep reinforcement learning. https://medium.com/@deshpandeshrinath/how-to-train-your-cheetah-with-deep-reinforcement-learning-14855518f916#:~:text=As%20we%20see%20in%20Half,forward%20but%20in%20flipper%20state., 2018.

[8] Steffen Schneider, Alexei Baevski, Ronan Collobert, and Michael Auli. wav2vec: Unsupervised pre-training for speech recognition. *arXiv preprint arXiv:1904.05862*, 2019.

[9] Xue Bin Peng, Erwin Coumans, Tingnan Zhang, Tsang-Wei Lee, Jie Tan, and Sergey Levine. Learning agile robotic locomotion skills by imitating animals. *arXiv preprint arXiv:2004.00784*, 2020.

[10] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26), 2019.

[11] Sylvain Koos, Jean-Baptiste Mouret, and Stéphane Doncieux. Crossing the reality gap in evolutionary robotics by promoting transferable controllers. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 119–126, 2010.

[12] Anusha Nagabandi, Ignasi Clavera, Simin Liu, Ronald S Fearing, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Learning to adapt in dynamic, real-world environments through meta-reinforcement learning. *arXiv preprint arXiv:1803.11347*, 2018.

[13] Charles Packer, Katelyn Gao, Jernej Kos, Philipp Krähenbühl, Vladlen Koltun, and Dawn Song. Assessing generalization in deep reinforcement learning. 2019.

[14] Aravind Rajeswaran, Sarvjeet Ghotra, Balaraman Ravindran, and Sergey Levine. Epopt: Learning robust neural network policies using model ensembles. *arXiv preprint arXiv:1610.01283*, 2016.

[15] Daniel J Mankowitz, Nir Levine, Rae Jeong, Yuanyuan Shi, Jackie Kay, Abbas Abdolmaleki, Jost Tobias Springenberg, Timothy Mann, Todd Hester, and Martin Riedmiller. Robust reinforcement learning for continuous control with model misspecification. *arXiv preprint arXiv:1906.07516*, 2019.

[16] Deepak Pathak, Chris Lu, Trevor Darrell, Phillip Isola, and Alexei A Efros. Learning to control self-assembling morphologies: a study of generalization via modularity. *arXiv preprint arXiv:1902.05546*, 2019.

[17] Wenlong Huang, Igor Mordatch, and Deepak Pathak. One policy to control them all: Shared modular policies for agent-agnostic control. In *International Conference on Machine Learning*, pages 4455–4464. PMLR, 2020.

[18] Jungdam Won and Jehee Lee. Learning body shape variation in physics-based characters. *ACM Transactions on Graphics (TOG)*, 38(6):1–12, 2019.

[19] Ilya Kostrikov, Denis Yarats, and Rob Fergus. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *arXiv preprint arXiv:2004.13649*, 2020.

[20] Michael Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. Reinforcement learning with augmented data. *arXiv preprint arXiv:2004.14990*, 2020.

[21] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

[22] Sanjeev Arora, Hrishikesh Khandeparkar, Mikhail Khodak, Orestis Plevrakis, and Nikunj Saunshi. A theoretical analysis of contrastive unsupervised representation learning. *arXiv preprint arXiv:1902.09229*, 2019.

[23] Aravind Srinivas, Michael Laskin, and Pieter Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. *arXiv preprint arXiv:2004.04136*, 2020.

[24] Amy Zhang, Rowan McAllister, Roberto Calandra, Yarin Gal, and Sergey Levine. Learning invariant representations for reinforcement learning without reconstruction. *arXiv preprint arXiv:2006.10742*, 2020.

[25] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

[26] Soeren Pirk, Mohi Khansari, Yunfei Bai, Corey Lynch, and Pierre Sermanet. Object-contrastive networks: Unsupervised object representations. 2018.

[27] Nicklas Hansen, Yu Sun, Pieter Abbeel, Alexei A Efros, Lerrel Pinto, and Xiaolong Wang. Self-supervised policy adaptation during deployment. *arXiv preprint arXiv:2007.04309*, 2020.

[28] Debidatta Dwibedi, Jonathan Tompson, Corey Lynch, and Pierre Sermanet. Self-supervised representation learning for continuous control. 2018.

[29] Ramanan Sekar, Oleh Rybkin, Kostas Daniilidis, Pieter Abbeel, Danijar Hafner, and Deepak Pathak. Planning to explore via self-supervised world models. In *International Conference on Machine Learning*, pages 8583–8592. PMLR, 2020.

[30] Yu Sun, Eric Tzeng, Trevor Darrell, and Alexei A Efros. Unsupervised domain adaptation through self-supervision. *arXiv preprint arXiv:1909.11825*, 2019.

[31] Pierre Sermanet, Corey Lynch, Yevgen Chebotar, Jasmine Hsu, Eric Jang, Stefan Schaal, Sergey Levine, and Google Brain. Time-contrastive networks: Self-supervised learning from video. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1134–1141. IEEE, 2018.

[32] Eric Jang, Coline Devin, Vincent Vanhoucke, and Sergey Levine. Grasp2vec: Learning object representations from self-supervised grasping. *arXiv preprint arXiv:1811.06964*, 2018.

[33] David Silver. Reinforcement learning course by david silver. https://www.youtube.com/watch?v=2pWv7GOvuf0&ab_channel=DeepMind.

[34] Spinning up in deep rl. https://spinningup.openai.com/en/latest/user/introduction.html.

[35] Divyam Rastogi. Deep reinforcement learning for bipedal robots. http://resolver.tudelft.nl/uuid:0fac495f-f87a-4a61-a80f-5f901323379a, 2017.

[36] Ayush Singh. Reinforcement learning: Bellman equation and optimality (part 2). https://towardsdatascience.com/reinforcement-learning-markov-decision-process-part-2-96837c936ec3, 2019.

[37] Cellstrat AI Lab. A summary of model-free rl algorithms. https://www.cellstrat.com/2020/04/13/a-summary-of-model-free-rl-algorithms/, 2020.

[38] Abhishek Suran. On-policy v/s off-policy learning. https://towardsdatascience.com/on-policy-v-s-off-policy-learning-75089916bc2f, 2020.

[39] Anthony Knittel Tim Eden and Raphael van Uffelen. Reinforcement learning. https://www.cse.unsw.edu.au/~cs9417ml/RL1/index.html.

[40] Chris Watkins. *Learning from delayed rewards*. PhD thesis, King's College, London, 1989.

[41] Alister Reis. Reinforcement learning: Eligibility traces and td(lambda). https://amreis.github.io/ml/reinf-learn/2017/11/02/reinforcement-learning-eligibility-traces.html, 2017.

[42] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, 1993.

[43] David Silver. Lecture notes on reinforcement learning. https://stdm.github.io/Lecture-notes-on-RL-David_Silver/, 2018.

[44] Thomas Degris, Martha White, and Richard S Sutton. Off-policy actor-critic. *arXiv preprint arXiv:1205.4839*, 2012.

[45] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[46] Andy Steinbach. Rl introduction: simple actor-critic for continuous actions. https://medium.com/@asteinbach/rl-introduction-simple-actor-critic-for-continuous-actions-4e22afb712, 2018.

[47] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[48] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, page pp. 387–395, 2014.

[49] Sebastian Curi, Felix Berkenkamp, and Andreas Krause. Efficient model-based reinforcement learning through optimistic policy search and planning. *arXiv preprint arXiv:2006.08684*, 2020.

[50] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[51] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[52] Hiroyuki Miyamoto, Mitsuo Kawato, Tohru Setoyama, and Ryoji Suzuki. Feedback-error-learning neural network for trajectory control of a robotic manipulator. *Neural networks*, 1(3):251–265, 1988.

[53] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[54] Harry A Pierson and Michael S Gashler. Deep learning in robotics: a review of recent research. *Advanced Robotics*, 31(16):821–835, 2017.

[55] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[56] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31(5):855–868, 2008.

[57] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6): 82–97, 2012.

[58] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[59] D. A. Vaccari and E. Wojciechowski. Neural networks as function approximators: teaching a neural network to multiply. In *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, volume 4, pages 2217–2222 vol.4, 1994. doi: 10.1109/ICNN.1994.374561.

[60] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

[61] Eugenio Culurciello. Neural network architectures. https://towardsdatascience.com/neural-network-architectures-156e5bad51ba.

[62] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[63] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[64] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[65] Ismoilov Nusrat and Sung-Bong Jang. A comparison of regularization techniques in deep neural networks. *Symmetry*, 10(11):648, 2018.

[66] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676): 354–359, 2017.

[67] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

[68] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528*, 2019.

[69] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[70] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[71] Scott Fujimoto, Herke Van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[72] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[73] Yunfei Bai Erwin Coumans. Pybullet quickstart guide. https://docs.google.com/document/d/10sXEhzFRSnvFcl3XxNGhnD4N2SedqwdAvK3dsihxVUA/edit#heading=h.2ye70wns7io3, 2016-2021.

[74] Yunfei Bai Erwin Coumans. Bullet real-time physics simulation. https://pybullet.org/wordpress/index.php/2020/09/24/pybullet-bullet-physics-3-05/, 2016-2021.

[75] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[76] Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *2015 IEEE international conference on robotics and automation (ICRA)*, pages 4397–4404. IEEE, 2015.

[77] Lilian Weng. Self-supervised representation learning. *lilianweng.github.io/lil-log*, 2019. URL https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html.

[78] Ankesh Anand, Evan Racah, Sherjil Ozair, Yoshua Bengio, Marc-Alexandre Côté, and R Devon Hjelm. Unsupervised state representation learning in atari. *arXiv preprint arXiv:1906.08226*, 2019.

[79] Yusuf Aytar, Tobias Pfaff, David Budden, Tom Le Paine, Ziyu Wang, and Nando de Freitas. Playing hard exploration games by watching youtube. *arXiv preprint arXiv:1805.11592*, 2018.

[80] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[81] Ashish Jaiswal, Ashwin Ramesh Babu, Mohammad Zaki Zadeh, Debapriya Banerjee, and Fillia Makedon. A survey on contrastive self-supervised learning. *Technologies*, 9(1):2, 2021.

[82] Zhirong Wu, Yuanjun Xiong, Stella Yu, and Dahua Lin. Unsupervised feature learning via non-parametric instance-level discrimination. *arXiv preprint arXiv:1805.01978*, 2018.

[83] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9729–9738, 2020.

[84] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[85] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

[86] Chris Nota. The autonomous learning library. https://autonomous-learning-library.readthedocs.io/en/stable/guide/benchmark_performance.html#:~:text=tested%20by%20DeepMind.-,PyBullet%20Benchmark,PyBullet%20is%20free%20and%20open., 2020.

[87] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[88] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.

[89] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[90] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[91] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *arXiv preprint arXiv:1606.03476*, 2016.

[92] Yuval Tassa, Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, and Nicolas Heess. $\mathrm{dm}_c ontrol : Softwareandtasksforcontinuouscontrol$, 2020.