BIPEDAL ROBOT WALKING BY REINFORCEMENT LEARNING IN PARTIALLY OBSERVED ENVIRONMENT

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

UĞURCAN ÖZALP

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
SCIENTIFIC COMPUTING

AUGUST 2021

Approval of the thesis:

BIPEDAL ROBOT WALKING BY REINFORCEMENT LEARNING IN PARTIALLY OBSERVED ENVIRONMENT

submitted by **UĞURCAN ÖZALP** in partial fulfillment of the requirements for the degree of **Master of Science in Scientific Computing Department, Middle East Technical University** by,

Prof. Dr. A. Sevtap Selçuk-Kestel _____
Director, Graduate School of **Applied Mathematics**

Prof. Dr. Hamdullah Yücel _____
Head of Department, **Scientific Computing**

Prof. Dr. Ömür Uğur _____
Supervisor, **Scientific Computing, METU**

Examining Committee Members:

Assoc. Prof. Dr. Ümit Aksoy
Mathematics, Atılım University _____

Prof. Dr. Ömür Uğur
Scientific Computing, METU _____

Assist. Prof. Dr. Önder Türk
Scientific Computing, METU _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: UĞURCAN ÖZALP

Signature :

ABSTRACT

BIPEDAL ROBOT WALKING BY REINFORCEMENT LEARNING IN PARTIALLY OBSERVED ENVIRONMENT

ÖZALP, UĞURCAN
M.S., Department of Scientific Computing
Supervisor : Prof. Dr. Ömür Uğur

August 2021, 56 pages

Deep Reinforcement Learning methods on mechanical control have been successfully applied in many environments and used instead of traditional optimal and adaptive control methods for some complex problems. However, Deep Reinforcement Learning algorithms do still have some challenges. One is to control on partially observable environments. When an agent is not informed well of the environment, it must recover information from the past observations. In this thesis, walking of Bipedal Walker Hardcore (OpenAI GYM) environment, which is partially observable, is studied by two continuous actor-critic reinforcement learning algorithms; Twin Delayed Deep Determinstic Policy Gradient and Soft Actor-Critic. Several neural architectures are implemented. The first one is Residual Feed Forward Neural Network under the observable environment assumption, while the second and the third ones are Long Short Term Memory and Transformer using observation history as input to recover the hidden information due to the partially observable environment.

Keywords: deep reinforcement learning, partial observability, robot control, actor-critic methods, long short term memory, transformer

ÖZ

PEKİŞTİRMELİ ÖĞRENME YÖNTEMLERİYLE KISMİ GÖZLENEBİLİR ORTAMDA ÇİFT BACAKLI ROBOTUN YÜRÜTÜLMESİ

ÖZALP, UĞURCAN
Yüksek Lisans, Bilimsel Hesaplama Bölümü
Tez Yöneticisi : Prof. Dr. Ömür Uğur

Ağustos 2021, 56 sayfa

Mekanik kontrol üzerine Derin Pekiştirmeli Öğrenme yöntemleri birçok ortamda başarıyla uygulanmış ve bazı karmaşık problemler için geleneksel optimal ve uyarlanabilir kontrol yöntemleri yerine kullanılmıştır. Bununla birlikte, Derin Pekiştirmeli Öğrenme algoritmalarının hala bazı zorlukları vardır. Bunlardan bir tanesi, kısmen gözlemlenebilir ortamlarda özneyi kontrol etmektir. Bir özne ortam hakkında yeterince bilgilendirilmediğinde, geçmiş gözlemleri anlık gözlemlere ek olarak kullanmalıdır. Bu tezde kısmen gözlemlenebilir olan Bipedal Walker Hardcore (OpenAI GYM) ortamında yürüme kontrolü, iki sürekli aktör-eleştirmen pekiştirmeli öğrenme algoritması tarafından incelenmiştir; İkiz Gecikmeli Derin Belirleyici Poliçe Gradyanı (Twin Delayed Deep Determinstic Policy Gradient) ve Hafif Aktör Eleştirmen (Soft Actor-Critic). Birkaç sinir mimarisi uygulanmıştır. Birincisi, gözlemlenebilir ortam varsayımına göre Artık Bağlantılı İleri Beslemeli Sinir Ağı iken ikincisi ve üçüncüsü, ortamın kısmen gözlemlenebilir olduğu varsayıldığından, gizli durumu kurtarmak için girdi olarak gözlem geçmişini kullanan Uzun Kısa Süreli Bellek (LSTM) ve Transformatördür (Transformer).

Anahtar Kelimeler: pekiştirmeli derin öğrenme, kısmi gözlemlenebilirlik, robot kontrolü, aktör-eleştirmen metodları, uzun kısa süreli bellek, transformatör

For anyone who is curious to read.

ACKNOWLEDGMENTS

I would like to thank my thesis supervisor Prof. Dr. Ömur Uğur whose insightful comments and suggestions were of inestimable value for my study. His willingness to give his time and share his expertise has paved the way for me.

Special thanks also go to my friend Mehmet Gökçay Kabataş whose opinions and information have helped me very much throughout the production of this study.

I would also like to express my gratitude to my family for their moral support and warm encouragements. Especially, I would like to show my greatest appreciation to Serpil Sökmen, who provides me to come these days.

Lastly, I would like to thank my partner Dilara Bayram for supporting me in long study days.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
RL	Reinforcement Learning
MDP	Markov Decision Process
POMDP	Partially Observable Markov Decision Process
DRL	Deep Reinforcement Learning
DNN	Deep Neural Network
FFNN	Feed Forward Neural Network
RFFNN	Residual Feed Forward Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory
DQN	Deep Q Network
DDQN	Double Deep Q Network
DDPG	Deep Deterministic Policy Gradient
TD3	Twin Delayed Deep Deterministic Policy Gradient
SAC	Soft Actor-Critic

CHAPTER 1

INTRODUCTION

Humans and animals exhibit several different behaviours in terms of interaction with environment, such as utterance and movement. Their behavior is based on past experience, the situation they are in and their objective. Like humans and animals, an intelligent agent is expected to take action according to its perception based on some objective. A major challenge in Machine Learning (ML) to create agents that will act more natural and humanlike. As a subfield of ML, Reinforcement Learning (RL) allows an agent to learn how to control (or act) itself in different situations by interacting with the environment. In RL, environment is modeled to give reward (or punishment) to agent according to environmental state and agent actions, and agent focuses on learning to predict what actions will lead to highest reward (or lowest punishment, based on its objective) in the future using past experience.

Traditional RL algorithms need feature engineering from observations. For complex problems, the way to extract features is ambiguous or observations are not enough to create a good model. As a newer technique, Deep Neural Networks (DNNs) allows to extract high level features from data with large state-space (like pixelwise visual, lidar scan, multiple kinematic sensors etc.) and missing observations. Along with recent developments in DNNs, Deep Reinforcement Learning (DRL) allows an agent to interact with the environment in a more complex way. The problem with DRL is the selection of a correct neural network, however, there is still no analytical way to design a neural network for all tasks. Therefore, neural design is commonly based on trial-error experiments for the particular problems at hand.

Since its discovery, robots have been crucial devices for the human race, whether

smart or not. Intelligent humanoid and animaloid robots have been developed since early 1980s. This type of robots has legs unlike driving robots. Since most of the world terrain is unpaved, this type of robots are good alternative to driving robots. Locomotion is a major task for such robots. Stable bipedal (2 legged) walking is one of the most challenging problem among the control problems. It is hard to create accurate model due to high order of dynamics, friction and discontinuities. Even further, the design of walking controller using traditional methods is difficult due to the same reasons. Therefore, for bipedal walking, DRL approach is an easier choice if a simulation environment is available.

In this thesis, Bipedal Locomotion is investigated through *BipedalWalker-Hardcore-v3* [3] environment of open source GYM library [8] by DRL. Our contributions to the related literature may be summarized as follows:

- Sequential neural networks are used (LSTM and Transformer) for solution, along with Residual Feed Forward Neural Network.
- Twin Delayed Deep Deterministic Policy Gradient (TD3) and Soft Actor-Critic (SAC) algorithms are used and compared.
- Reward shaping and small modifications are applied on environment to hinder possible problems.

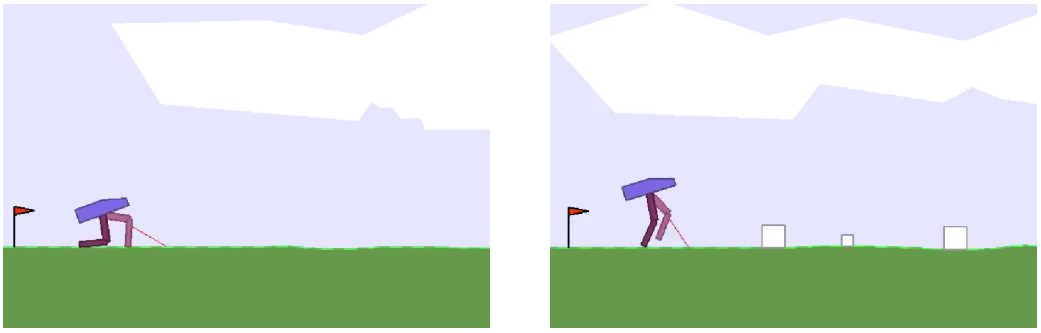
1.1 Problem Statement: Bipedal Walker Robot Control

In this work, we attempted to solve *BipedalWalkerHardcore-v3* [3] from OpenAI's open source Gym [8] library using deep learning framework PyTorch [34].

1.1.1 OpenAI Gym and Bipedal Walker Environment

OpenAI Gym [8] is an open source framework, containing many environments to service the development of reinforcement learning algorithms.

BipedalWalker environments [2, 3] are parts of Gym environment library. One of them is classical where the terrain is relatively smooth, while other is a hardcore



(a) BipedalWalker-v3 Snapshot [2] (b) BipedalWalkerHardcore-v3 Snapshot [3]

Figure 1.1: Bipedal Walkers Snapshots

version containing ladders, stumps and pitfalls in terrain. Robot has deterministic dynamics but terrain is randomly generated at the beginning of each episode. Those environments have continuous action and observation space. For both settings, the task is to move the robot forward as much as possible. Snapshots for both environments are depicted in Figure 1.1.

Locomotion of the Bipedal Walker is a difficult control problem due to following reasons:

Nonlinearity The dynamics are nonlinear, unstable and multimodal. Dynamical behavior of robot changes for different situations like ground contact, single leg contact and double leg contact.

Uncertainity The terrain where the robot walks also varies. Designing a controller for all types of terrain is difficult.

Reward Sparsity Overcoming some obstacles requires a specific maneuver, which is hard to explore sometimes.

Partially Observability The robot observes ahead with lidar measurements and cannot observe behind. In addition, it lacks of acceleration sensors.

These reasons make it also hard to implement analytical methods for control tasks. However, DRL approach can easily overcome nonlinearity and uncertainity problems.

On the other hand, reward sparsity problem brings local minimums to objective function of optimal control. However, this can be challenged by a good exploration strategy and reward shaping.

For the partial observability problem, more elegant solution is required. This is, in general, achieved by creating a belief state from the past observations to inform the agent. Agent uses this belief state to choose how to act. If the belief state is evaluated sufficiently, this increases performance of the agent. Relying on instant observations is also possible, and this may be enough sometimes if advanced type of control is not required.

1.1.2 Deep Learning Library: PyTorch

PyTorch is an open source library developed by Facebook's AI Research Lab (FAIR) [34]. It is based on Torch library [9] and has Python and C++ interface. It is an automatic differentation library with accelerated mathematical operations backed by graphical processing units (GPUs). The clean pythonic syntax made it one of the most famous deep learning tool among researches.

1.2 Proposed Methods and Contribution

Partially observable environments are always a hard work for reinforcement learning algorithms. In this work, the walker environment is assumed to be fully observable environment at first. A Residual Feed-Forward Neural Network architecture is proposed to control the robot under fully observability assumption due to the fact that no memory is used during decision making. Then, the environment is assumed to be partially observable. In order to recover belief states, Long Short Term Memory (LSTM) and Transformer neural networks are proposed using fixed number of past observations (6 and 12 in our case) during decision making.

LSTM is used in many deep learning applications including sequential data. It is a variant of Recurrent Neural Networks (RNN) and a good candidate for RL algorithms to be applied in partially observable environments.

Transformer is developed to handle sequential data as RNN models do. However, it processes the whole sequence at the same time, while RNN processes the sequence in order. Transformers are commonly used in Natural Language Processing (NLP)

thanks to major performance improvements over RNN variants, but this is not the case for Reinforcement Learning, yet.

In order to handle the reward sparsity problem, reward function is redesigned in this thesis. Also, an exploration strategy is formed so that the agent both explores and learns sufficiently well.

In this thesis, Twin Delayed Deep Deterministic Policy Gradient (TD3) and Soft Actor Critic (SAC) are used to solve our environment as RL algorithm. TD3 is a deterministic RL method with additive exploration noise, and it is improved version of Deep Deterministic Policy Gradient (DDPG). SAC is a stochastic type of RL method with adaptive exploration. It adjusts how much to explore depending on observations and rewards.

1.3 Related Work in Literature

Reinforcement Learning methods are used in many mechanical control tasks such as autonomus driving [32, 42, 40, 50] and autonomus flight [24, 5, 41], where conventional control methods are difficult to implement.

Rastogi [36] used Deep Deterministic Policy Gradient (DDPG) algorithm to walk their physical bipedal walker robot along with simulation environment. They concluded that DDPG is infeasible to control the walker robot since it requires long time for convergence. Kumar et al. [25] also used DDPG to perform robot walking in 2D simulation environment. Their agent achieved desired score in approximately 25,000 episodes. Song et al. [44] pointed out the partial observability problem of bipedal walker, using Recurrent Deep Deterministic Policy Gradient (RDDPG) [19] algorithm and acquired better results than the original DDPG algorithm. Haarnoja et al. [16] used maximum entropy learning for gaiting of real robot and achieved stable gait in 2 hours.

Fris [12] used Twin Delayed Deep Deterministic Policy Gradient (TD3) using LSTM for their quadrocopter landing task in sparse reward setting. They stated that LSTM is used to infer possible high order dynamics from observations. Fu et al. [13] used

vanilla RNN with attention mechanism using TD3 for car driving task, but not explicit Transformer. They reported that their method outperformed seven baselines. Upadhyay et al. [47] used Feed Forward Neural Network, LSTM and vanilla Transformer architectures for balancing pole on a cart from Cartpole environment of Gym, and Transformer yield worst results among three architectures. Parisotto et al. [33] pointed out order of layer normalization with respect to attention and feed-forward layers dramatically changes performance on RL tasks.

1.4 Outline of the Thesis

This thesis consists of five chapters. In Chapter 2, we discuss the theory of Reinforcement Learning and introduce the methods used in this thesis. In Chapter 3, we explain the theory of Neural Networks and Deep Learning along with architectures which are designed to process sequential data. In Chapter 4, Bipedal Walker environments are presented, neural networks and RL algoritmhs are proposed, results are summarized and discussed. In the last chapter, thesis is concluded by discussing obtained results and possible future work is outlined along with the future of RL.

CHAPTER 2

REINFORCEMENT LEARNING

Machine Learning is the ability of a computer program that allows adaptation to new situations through experience, without explicitly programmed [28]. As shown in Figure 2.1, there exist three main paradigms:

Supervised Learning is the task of learning a function $f: X \rightarrow Y$ that maps an input to an output based on N example input-output pairs (x_i, y_i) such that $f(x_i) \approx y_i$, for all $i = 1, 2, ..., N$ by minimizing error between predicted and target output. Input x can be thought as the state of an agent, and y is the correct action at state x. For supervised learning, both x and y should be available, where the correct actions are provided by a friendly supervisor [39].

Unsupervised Learning discovers the structure on input examples without any label on them. Based on N example input (x_i), it discovers function $f: X \rightarrow Y$, $f(x_i) = y_i$, for all $i = 1, 2, ..., N$, where y_i is discovered output. This discovery is motivated by predefined objective. This objective is maximization of a value which represents compactness of output representations. Again, input x can be thought as the state of an agent. However, correct action is not available and there is no given hint in this case. It can learn relations among states but it does not know what to do since there is no target or utility [39].

Reinforcement Learning is one of the three main machine learning paradigms along with Supervised and Unsupervised Learning. It is the closest kind of learning demonstrated by humans and animals since it is grounded by biological learning systems. It is based on maximizing cumulative reward over time to make agent learn how to

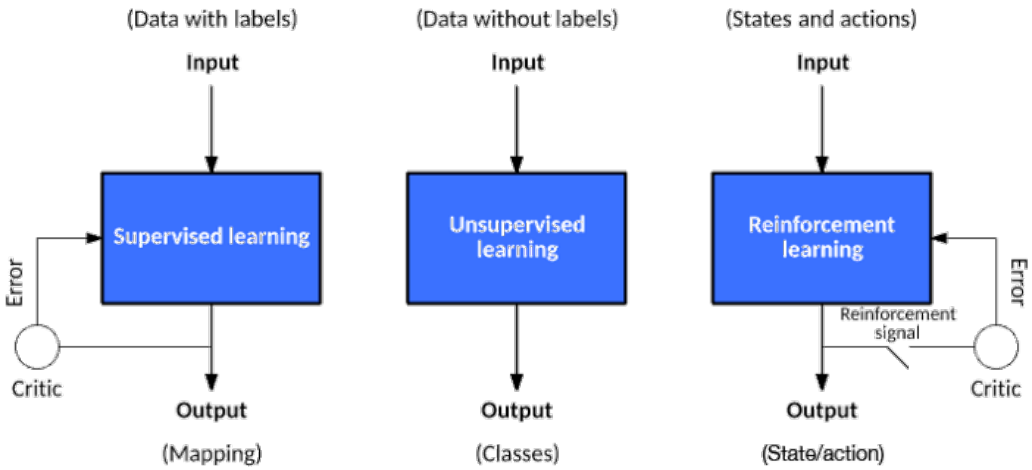


Figure 2.1: Main paradigms of ML [1]

act in an environment [45]. Each action of the agent is either rewarded or punished according to a reward function. Therefore, reward function is representation of what to teach the agent. The agent explores environment by taking various actions in different states to gain experience, based on trial-and-error. Then it exploits experiences to get the highest reward from the environment considering instant and future rewards over time.

Formally, Reinforcement Learning is learning a policy function (strategy of the agent) $\pi : \mathcal{S} \rightarrow \mathcal{A}$ which maps inputs (states) $s \in \mathcal{S}$ to outputs (actions) $a \in \mathcal{A}$. Learning is achieved by maximization of the value function $V^{\pi}(s)$ (cumulative reward) for all possible states, which depends on policy π. In this sense, it is similar to unsupervised learning. However, the difference is that the value function $V^{\pi}(s)$ is not defined exactly unlike unsupervised learning setting. It is also learned by interacting with the environment by taking all possible actions in all possible states.

In short, Reinforcement Learning is different from Supervised Learning because the correct actions are not provided. It is also different from Unsupervised Learning because the agent is forced to learn a specific behaviour and evaluated at each time step without explicit supervision.

2.1 Reinforcement Learning and Optimal Control

Optimal control is a field of mathematical optimization, to find the control policy of a dynamical system (environment) for a given objective. For example, objective might be the total revenue of a company, minimal fuel burn for a car or total production of a factory.

RL may be considered as a naive subfield of optimal control. However, RL algorithms find policy (controller) by error minimization of objective from experience, while traditional optimal control methods are concerned of exact analytical optimal solutions based on dynamic model of the environment and the agent.

Traditional optimal control methods are efficient and robust when mathematical model of dynamics is available, accurate enough and solvable for optimal controller in practice. However, many real world problems usually do not exhibit all of these conditions. In such cases, reinforcement learning is an easier alternative to derive a control policy.

2.2 Challenges

The reinforcement learning environment poses a variety of obstacles that we need to address and potentially make trade-offs among them [10, 45].

2.2.1 Exploration Exploitation Dilemma

In RL, an agent is supposed to maximize rewards (exploitation of knowledge) by observing the environment (exploration of environment). This gives rise to the exploration-exploitation dilemma which is an inevitable trade-off. Exploration means taking a range of acts to benefit from the consequences, which typically results in low immediate rewards but high rewards for the future. Exploitation means taking action that has been learned, and typically results in high immediate rewards but low rewards in the future.

2.2.2 Generalization and Curse of Dimensionality

In RL, an agent should also be able to generalize experiences to act on previously unseen situations. This issue arises when state space and action space is high-dimensional since experiencing all possibilities is impractical. However, this is solved by introducing function approximators. In Deep Reinforcement Learning, neural networks are used as function approximators.

2.2.3 Delayed Consequences

In RL, an agent should be aware of the reason of reward or punishment. Sometimes, a wrong action may cause punishment later. For example, once a self-driving car enters a dead end way, it is not punished unless the way ends. Therefore, once an agent gets a reward or punishment, it should be able to discriminate whether reward is caused by instant or past actions.

2.2.4 Partial Observability

Partial observability is the absence of all required observations to infer the instant state. For instance, a driver may not need to know engine temperature or rotational speed of gears. Although driver is able to drive in that case, s/he would not be able to drive well on traffic in the absence of rear view mirror or side mirrors. In real world, most systems are partially observable. This problem is usually tackled by incorporating observation history from the agents memory for action selection.

2.2.5 Safety of Agent

Mechanical agents can kill or degrade themselves and their surroundings during the learning process. This safety problem is important on both exploration stage and full operation. In addition, it is difficult to analyze an agent's policy, that is, it is uncertain what the agent do in unseen situation. Simulation of environment is a good way to train the agent with safety, however, this causes an incomplete learning due to

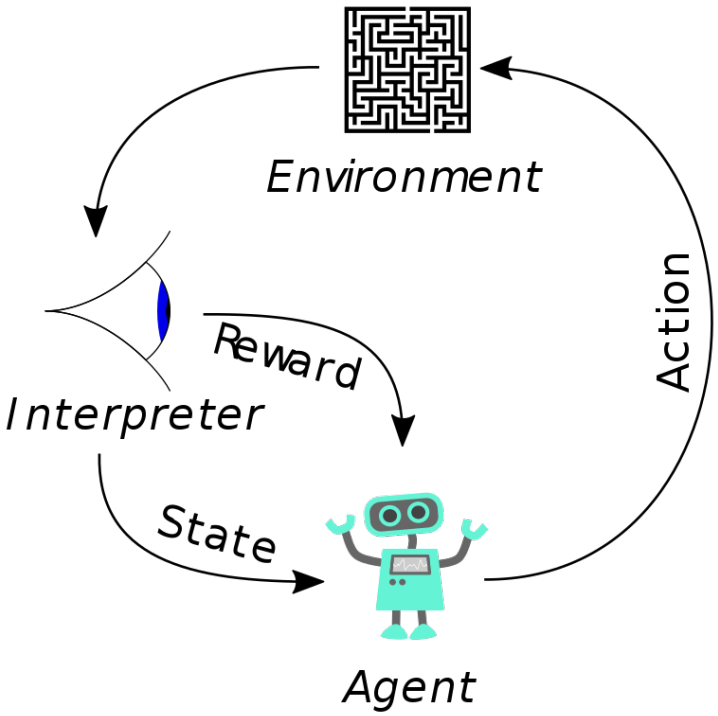


Figure 2.2: Reinforcement Learning Diagram

inaccuracy compared to the real environment.

2.3 Sequential Decision Making

RL may also be considered as a stochastic control process in discrete time setting [45]. At time t, the agent starts with state s_t and observes o_t, then it takes an action a_t according to its policy π and obtains a reward r_t at time t. Hence, a state transition to s_{t+1} occurs as a consequence of the action and the agent gets the next observation o_{t+1}. History is, therefore, the ordered set of past actions, observations and rewards: $h_t = \{a_0, o_0, r_0, ...a_t, o_t, r_t\}$. The state s_t is a function of the history, i.e., $s_t = f(h_t)$, which represents the characteristics of environment at time t as much as possible. The RL diagram is visualized in Figure 2.2.

2.4 Markov Decision Process

Markov Decision Process (MDP) is a sequential decision making process with Markov property. It is represented as a tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$. Markov property means that the

conditional probability distribution of the future state depends only on the instant state and action instead of the entire state/action history, so it is regarded as memoryless. In MDP setting, the system is fully observable which means that the states can be derived from instant observations; i.e., $s_t = f(o_t)$. Therefore, agent can decide an action based on only instant observation o_t instead of what happened at previous times [11]. MDP consists of the following:

State Space $\mathcal{S}$ A set of all possible configurations of the system.

Action Space $\mathcal{A}$ A set of all possible actions of the agent.

Model $T\colon \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ A function of how environment evolves through time, representing transition probabilities as $T(s'|s, a) = p(s'|s, a)$ where $s' \in \mathcal{S}$ is the next state, $s \in \mathcal{S}$ is the instant state and $a \in \mathcal{A}$ is the action taken.

Reward Function $R\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ A function of rewards obtained from the environment. At each state transition $s_t \to s_{t+1}$, a reward r_t is given to the agent. Rewards may be either deterministic or stochastic. Reward function is the expected value of reward given the state s and the action taken a, defined by:

$$R(s, a) = \mathbb{E}[r_t|s_t = s, a_t = a]. \tag{2.1}$$

Discount Factor $\gamma \in [0, 1]$ A measure of the importance of rewards in the future for the value function.

2.5 Partially Observed Markov Decision Process

In MDP, agent can recover full state from observations, i.e., $s_t = f(o_t)$. However, observation space is not enough to represent all information (states) about the environment sometimes. That means one needs more observations from the history, i.e., $s_t = f(o_t, o_{t-1}, o_{t-2}, ...)$. In such cases, past and instant observations are used to filter out a belief state. It is represented as a tuple $(\mathcal{S}, \mathcal{A}, T, R, \mathcal{O}, O, \gamma)$. In addition to MDP, it introduces observation space $\mathcal{O}$ and observation model O [11]:

Observation Space $\mathcal{O}$ A set of all possible observations of the agent.

Observation Model $O\colon \mathcal{O} \times \mathcal{S} \to [0, 1]$ A function of how observations are related to the states, representing observation probabilities as $O(o|s) = p(o|s)$ where $s \in \mathcal{S}$ is the instant state and $o \in \mathcal{O}$ is the observation.

Since states are not observed directly, the agent needs to use the observations while deriving a control policy.

2.6 Policy and Control

2.6.1 Policy

A policy defines how the agent acts according to the state of the environment. It may be either deterministic or stochastic:

Deterministic Policy $\mu\colon \mathcal{S} \to \mathcal{A}$ A mapping from states to actions.

Stochastic Policy $\pi\colon \mathcal{S} \times \mathcal{A} \to [0, 1]$ A mapping from state-action pair to a probability value.

2.6.2 Return

At time t, return G_t is a cumulative sum of the future rewards scaled by the discount factor γ:

$$G_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i = r_t + \gamma G_{t+1}. \tag{2.2}$$

Since the return depends on future rewards, it also depends on the policy of the agent as it affects the future rewards.

2.6.3 State Value Function

State Value Function V^π is the expected return when policy π is followed in the future and is defined by

$$V^\pi(s) = \mathbb{E}[G_t | s_t = s, \pi]. \tag{2.3}$$

Optimal value function should return maximum expected return, where the behavior is controlled by the policy. In other words,

$$V^*(s) = \max_{\pi} V^{\pi}(s). \tag{2.4}$$

2.6.4 State-Action Value Function

State-Action Value Function Q^{π} is again the expected return when policy π is followed in the future, however, any action taken at the instant step:

$$Q^{\pi}(s,a) = \mathbb{E}[G_t | s_t = s, a_t = a, \pi] = R(s,a) + \gamma V^{\pi}(s'), \tag{2.5}$$

where s' is the next state resulting from action a. Optimal state-action value function should yield maximum expected return for each state-action pair. Hence,

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a). \tag{2.6}$$

The optimal policy π^* can be obtained by $Q^*(s,a)$. For stochastic policy, it is defined as

$$\pi^*(a|s) = \begin{cases} 1, & \text{if } a = \arg\max_a Q^*(s,a), \\ 0, & \text{otherwise .} \end{cases} \tag{2.7}$$

For deterministic policy, it is

$$\mu^*(s) = \arg\max_{a} Q^*(s,a). \tag{2.8}$$

2.6.5 Bellman Equation

Bellman proved that optimal value function, for a model T, must satisfy following conditions [7]:

$$V^*(s) = \max_{a} \Big\{ R(s,a) + \gamma \sum_{s'} T(s'|s,a) V^*(s') \Big\} \tag{2.9}$$

$$Q^*(s,a) = R(s,a) + \gamma \max_{a'} \Big\{ \sum_{s'} T(s'|s,a) Q^*(s',a') \Big\} \tag{2.10}$$

These equations can simply be derived from (2.4) and (2.6). Most of RL methods are build upon solving (2.10), since there exist a direct relation between Q and π as in (2.8) and (2.7).

2.7 Model-Free Reinforcement Learning

Model based methods are based on solving Bellman equation (2.9), (2.10) with a given model T. On the other hand, Model-Free Reinforcement Learning is suitable if environment model is not available but the agent can experience environment by the consequences of its actions. There are three main categories in model-free RL:

Value-Based Learning The value functions are learned, hence the policy arises naturally from the value function using (2.7) and (2.8). Since argmax operation is used, this type of learning is suitable for problems where action space is discrete.

Policy-Based Learning The policy is learned directly, and return values are used instead of learning a value function. Unlike value-based methods, it is suitable when continuous action spaces are available in the environment.

Actor-Critic Learning Both policy (actor) and value (critic) functions are learned simulatenously. Thus, it is also suitable for continuous action spaces.

2.7.1 Q Learning

Q Learning is a value-based type of learning. It is based on optimizing Q function using Bellman Equation (2.10) [51].

In optimal policy context, state-value function maximizes itself adjusting policy. Therefore, there is a direct relation between it and state-action value function,

$$V^*(s) = \max_a Q^*(s, a). \tag{2.11}$$

Then, plugging this definition into (2.5) for next state-action pairs by assuming we have optimal policy, we obtain,

$$Q^*(s, a) = R(s, a) + \gamma \max_{a'} Q^*(s', a'). \tag{2.12}$$

In this learning strategy, Q function is assumed to be parametrized by θ. Target Q value (Y^Q) is estimated by bootstrapping current model (θ) for each time t,

$$Y_t^Q = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta). \tag{2.13}$$

At time t, with state, action, reward, next state tuples (s_t, a_t, r_t, s_{t+1}), Q values are updated by minimizing difference between the target value and the estimated value. Hence the loss

$$\mathcal{L}_t(\theta) = \big(Y_t - Q(s_t, a_t; \theta)\big)^2, \tag{2.14}$$

is to be minimized with respect to θ using numerical optimization methods.

2.7.1.1 Deep Q Learning

When a nonlinear approximator is used for Q estimation, learning is unstable due to the correlation among recent observations, hence correlations between updated Q function and observations. Deep Q Learning solves this problem by introducing Target Network and Experience Replay [30, 29] along with using deep neural networks.

Target Network is parametrized by θ^-. It is used to evaluate target value, but it is not updated by the loss minimization. It is updated at each fixed number of update step by Q network parameter θ. In this way, correlation between target value and observations are reduced. The target value is obtained by using θ^- as

$$Y_t^{DQN} = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-). \tag{2.15}$$

Experience Replay stores experience tuples in the replay memory $\mathcal{D}$ as a queue with fixed buffer size N_{replay}. At each iteration i, θ is updated by experiences $(s, a, r, s') \sim U(\mathcal{D})$ uniformly subsampled from experience replay by minimizing the expected loss

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}\Big[\big(Y^{DQN} - Q(s, a; \theta_i)\big)^2\Big]. \tag{2.16}$$

It allows agent to learn from experiences multiple times at different stages of learning. More importantly, sampled experiences are close to be independent and identically distributed if buffer size is large enough. Again, this reduces correlation between recent observations and updated Q value. This makes learning process more stable.

Epsilon Greedy Exploration is used to let agent explore environment. In discrete action space (finite action space $\mathcal{A}$), this is the simplest exploration strategy used in RL algorithms. During the learning process, a random action with probability ϵ is

selected or greedy action (maximizing Q value) with probability $1-\epsilon$. In order to construct policy:

$$\pi(a|s) = \begin{cases} 1-\epsilon, & \text{if } a = \arg\max_a Q(s,a), \\ \dfrac{\epsilon}{|\mathcal{A}|-1}, & \text{otherwise,} \end{cases} \tag{2.17}$$

where $|\mathcal{A}|$ denotes cardinality of $\mathcal{A}$. In Algorithm 1, we summarize the Deep Q Learning with Experience Replay.

Algorithm 1: Deep Q Learning with Experience Replay

Initialize: $\mathcal{D}$, N, N_{replay}, θ, ϵ, d

$\theta^- \leftarrow \theta$

for *episode* $= 1, M$ **do**

  Recieve initial state s_1;

  for $t = 1, T$ **do**

    Randomly select action a_t with probability ϵ, otherwise $a_t = \arg\max_a Q(s_t, a; \theta)$ greedily;

    Execute action a_t and recieve reward r_t and next state s_{t+1};

    Store experience tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ to $\mathcal{D}$;

    Sample random batch $\mathcal{D}_r$ ($|\mathcal{D}_r| = N$) from $\mathcal{D}$;

    $$Y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ terminal} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases} \qquad \forall e_j \in \mathcal{D}_r$$

    Update θ by minimizing $\frac{1}{N}\sum_{e_j \in \mathcal{D}_r} \left[Y_j - Q(s_j, a_j; \theta)\right]^2$ for a single step;

    if $t \mod d$ **then** Update target network: $\theta^- \leftarrow \theta$;

  end

end

2.7.1.2 Double Deep Q Learning

In DQN, maximum operator is used to both select and evaluate action on the same network as in (2.15). This yields overestimation of Q function in noisy environments. Therefore, action selection and value estimation is decoupled in target evaluation to overcome Q function overestimation [48] using Double Deep Q Network (DDQN). The target value in this approach can therefore be written as follows:

$$Y_t^{DDQN} = r_t + \gamma Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta_i); \theta^-). \tag{2.18}$$

Except the target value, the learning process is the same as in DQN.

2.7.2 Actor-Critic Learning

Value-based methods are not suitable for continuous action spaces. Therefore, policy should be explicitly defined instead of maximizing Q function. For such problems, actor-critic type learning methods use both value and policy models seperately.

In actor-critic learning, there exists two components [43]: First one is the actor, which is the policy function, either stochastic or deterministic, parametrized by θ^π or θ^μ, respectively. For policy learning, an estimate of the value function is used instead of the true value function since it is already unknown. Second one is the critic, which is an estimator of the Q value, parametrized by θ^Q. Critic learning is achieved by minimizing the error obtained by either Monte Carlo sampling or Temporal Difference bootstrapping. Critic is what policy uses for value estimation.

The ultimate goal is to learn a policy maximizing the value function V by selecting action which maximizes Q value in (2.8). Therefore, value function is the criteria to be maximized by solving parameters θ^π (or θ^μ) given θ^Q. At time t, the loss function for the policy is negative, that is,

$$\mathcal{L}_t(\theta^\pi) = -Q(s_t, \widetilde{a}_t; \theta^Q), \quad \widetilde{a}_t \sim \pi(\cdot|s_t; \theta^\pi). \tag{2.19}$$

In order to learn policy π, Q function should also be learned simultaneously. For Q function approximation, the target value is parametrized by θ^Q and θ^π,

$$Y_t^{AC} = r_t + \gamma Q(s_{t+1}, \widetilde{a}_{t+1}; \theta^Q), \quad \widetilde{a}_{t+1} \sim \pi(\cdot|s_{t+1}; \theta^\pi), \tag{2.20}$$

and this target is used to learn Q function by minimizing the least squares loss (in general),

$$\mathcal{L}_t(\theta^Q) = \left[Y_t^{AC} - Q(s_t, a_t; \theta^Q)\right]^2. \tag{2.21}$$

Note that both the actor and the critic should be learned at the same time. Therefore, parameters are updated simultaneously during the iterations in the learning process.

2.7.2.1 Deep Deterministic Policy Gradient

DDPG is continuous complement of DQN using a deterministic policy [26]. It also uses experience replay and target networks. Similar to Deep Q Learning, there are tar-

get networks parametrized by θ^{μ^-} and θ^{Q^-} along with the main networks parametrized by θ^{μ} and θ^{Q}.

While target networks are updated in a fixed number of steps in DQN, DDPG updates target network parameters at each step with Polyak averaging,

$$\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-. \tag{2.22}$$

The τ is an hyperparameter indicating how fast the target network is updated and it is usually close to zero.

Policy network parameters are learned by maximizing resulting expected value, or minimizing its negative,

$$\mathcal{L}_i(\theta_i^\mu) = -\mathbb{E}_{s\sim U(\mathcal{D})}\Big[Q(s, \mu(s\theta_i^\mu); \theta^Q)\Big]. \tag{2.23}$$

Note that value network parameters are also assumed to be learned.

In addition, target networks are used to predict target value and the target is defined by

$$Y_t^{DDPG} = r_t + \gamma Q(s_{t+1}, \mu(s_{t+1}; \theta^{\mu^-}); \theta^{Q^-}). \tag{2.24}$$

In each iteration, this target is used to learn Q function by minimizing the least squares loss

$$\mathcal{L}_i(\theta_i^Q) = \mathbb{E}_{s,a,r,s'\sim U(\mathcal{D})}\Big[\big(Y^{DDPG} - Q(s, a; \theta_i^Q)\big)^2\Big]. \tag{2.25}$$

In DDPG, value and policy network parameters are learned simultaneously. During the learning process, exploration noise is added to each selected action, sampled by a process $\mathcal{X}$. In [26], authors proposed to use Ornstein-Uhlenbeck Noise [46] in order to have temporal correlation for efficiency. However, a simple Gaussian white noise or any other one is also possible.

DDPG is summarized in Algorithm 2.

Algorithm 2: Deep Deterministic Policy Gradient

Initialize: $\mathcal{D}$, N, N_{replay}, θ^{μ}, θ^{Q}, $\mathcal{X}$

$\theta^{\mu^-} \leftarrow \theta^{\mu}$, $\theta^{Q^-} \leftarrow \theta^{Q}$

for *episode* $= 1, M$ **do**

  Recieve initial state s_1;

  for $t = 1, T$ **do**

    Select action $a_t = \mu(s_t; \theta^{\mu}) + \epsilon$ where $\epsilon \sim \mathcal{X}$;

    Execute action a_t and recieve reward r_t and next state s_{t+1};

    Store experience tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ to $\mathcal{D}$;

    Sample random batch $\mathcal{D}_r$ ($|\mathcal{D}_r| = N$) from $\mathcal{D}$;

    $$Y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ terminal} \\ r_j + \gamma Q(s_{j+1}, \mu(s_{j+1}; \theta^{\mu^-}); \theta^{Q^-}) & \text{otherwise} \end{cases} \qquad \forall e_j \in \mathcal{D}_r$$

    Update θ^{Q} by minimizing $\frac{1}{N} \sum_{e_j \in \mathcal{D}_r} \left(Y_j - Q(s_j, a_j; \theta^{Q})\right)^2$ for a single step;

    Update θ^{μ} by maximizing $\frac{1}{N} \sum_{e_j \in \mathcal{D}_r} Q(s_j, a_j; \theta^{Q})$ for a single step;

    Update target networks:

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

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

  end

end

Ornstein-Uhlenbeck Process is the continuous analogue of the discrete first order autoregressive (AR(1)) process [46]. The process x is defined by a stochastic differential equation,

$$\frac{dx}{dt} = -\theta x + \sigma\eta(t), \tag{2.26}$$

where $\eta(t)$ is a white noise. Its standard deviation in time is equal to $\frac{\sigma}{\sqrt{2\theta}}$. This process is commonly used as an exploration noise in physical environments since it has temporal correlation. Two sample paths of the process are shown in Figure 2.3 in order to compare with the Gaussian noise.

2.7.2.2 Twin Delayed Deep Deterministic Policy Gradient

TD3 [14] is an improved version of DDPG with higher stability and efficiency. There are three main tricks upon DDPG:

Target Policy Smoothing is to regularize the learning process by smoothing effects

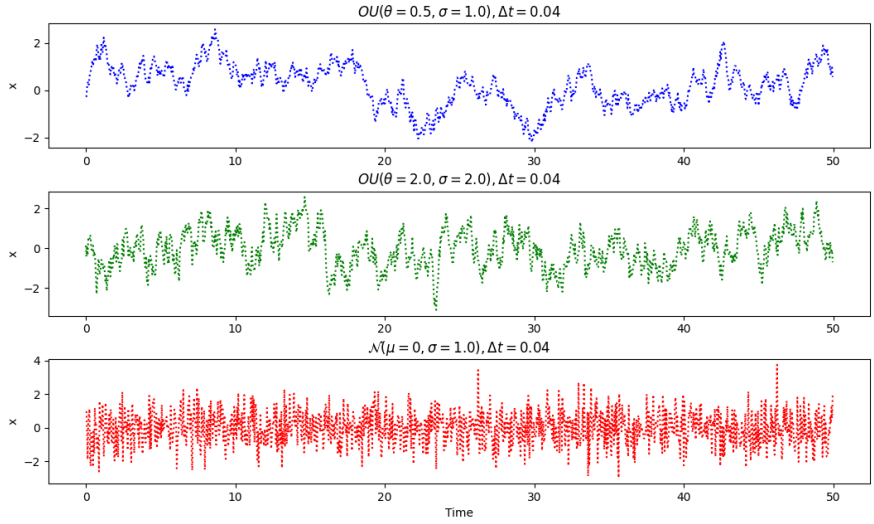


Figure 2.3: Ornstein-Uhlenbeck Process and Gaussian Process comparison

of actions on value. For target value assessing, actions are obtained from the target policy network in DDPG, while a clipped zero-centered Gaussian noise is added to actions in TD3 as follows:

$$\widetilde{a}' = \mu(s'; \theta^{\mu^-}) + \text{clip}(\epsilon, -c, c), \quad \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{2.27}$$

Clipped Double Q Learning is to escape from Q value overestimation. There are two different value networks with their targets. During learning, both networks are trained by single target value assessed by using whichever of the two networks give smaller. In other words,

$$Y_t^{TD3} = r_t + \gamma \min_{k \in \{1,2\}} Q(s_{t+1}, ; \widetilde{a}_{j+1}; \theta^{Q_k^-}). \tag{2.28}$$

On the other hand, the policy is learned by maximizing the output of the first value network, or minimizing its negative,

$$\mathcal{L}_i(\theta_i^\mu) = -\mathbb{E}_{s \sim U(\mathcal{D})} \Big[Q(s, \mu(s; \theta_i^\mu); \theta^{Q_1}) \Big]. \tag{2.29}$$

Delayed Policy Updates is used for stable training. During learning, policy network and target networks are updated less frequently (at each fixed number of steps) than the value network. Since the policy network parameters are learned by maximizing the value network, they are learned slower.

TD3 is summarized in Algorithm 3.

Algorithm 3: Twin Delayed Deep Deterministic Policy Gradient

Initialize: $\mathcal{D}$, N, N_{replay}, θ^{μ}, θ_1^Q, θ_2^Q, $\mathcal{X}$, σ, c, d

$\theta^{\mu^-} \leftarrow \theta^{\mu}$, $\theta_1^{Q^-} \leftarrow \theta_1^Q$, $\theta_2^{Q^-} \leftarrow \theta_2^Q$

for *episode* $= 1, M$ **do**

  Recieve initial state s_1;

  for $t = 1, T$ **do**

    Select action $a_t = \mu(s_t; \theta^{\mu}) + \epsilon$ where $\epsilon \sim \mathcal{X}$ Execute action a_t and recieve reward r_t and next state s_{t+1};

    Store experience tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ to $\mathcal{D}$;

    Sample random batch $\mathcal{D}_r$ ($|\mathcal{D}_r| = N$) from $\mathcal{D}$;

    Sample target actions for value target,

    $\widetilde{a}_{j+1} = \mu(s_{j+1}; \theta^{\mu^-}) + \text{clip}(\epsilon, -c, c), \quad \epsilon \sim \mathcal{N}(0, \sigma^2) \quad \forall e_j \in \mathcal{D}_r$;

    $$Y_{jk} = \begin{cases} r_j & \text{if } s_{j+1} \text{ terminal} \\ r_j + \gamma Q(s_{j+1}, \widetilde{a}_{j+1}); \theta_k^{Q^-}) & \text{otherwise} \end{cases} \quad \forall e_j \in \mathcal{D}_r,$$

    $\forall k \in \{1, 2\}$

    $Y_j = \min(Y_{j1}, Y_{j2})$;

    Update θ_1^Q θ_2^Q by seperately minimizing

    $\frac{1}{N} \sum_{e_j \in \mathcal{D}_r} \left(Y_j - Q(s_j, a_j; \theta_k^Q)\right)^2 \quad \forall k \in \{1, 2\}$ for a single step;

    if $t \mod d$ **then**

      Update θ^{μ} by maximizing $\frac{1}{N} \sum_{e_j \in \mathcal{D}_r} Q(s_j, a_j; \theta_1^Q)$ for a single step;

      Update target networks:

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

      $\theta_k^{Q^-} \leftarrow \tau\theta_k^Q + (1 - \tau)\theta_k^{Q^-} \quad \forall k \in \{1, 2\}$;

  end

end

2.7.2.3 Soft Actor-Critic

SAC [17] is a stochastic actor-critic method and many characteristics are similar to TD3 and DDPG. It is an entropy-regularized RL method. Such methods give bonus reward to the agent propotional to the policy entropy: given state s, the entropy of a policy π is defined by

$$H(\pi(\cdot|s)) = \mathbb{E}_{\widetilde{a} \sim \pi(\cdot|s)}[-\log(\pi(\widetilde{a}|s))]. \tag{2.30}$$

Given the entropy coefficient α, definition of state-action value function is redefined

as follows:

$$Q^{\pi}(s,a) = \mathbb{E}_{\substack{s'\sim T(\cdot|s,a) \\ \widetilde{a}'\sim\pi(\cdot|s')}}\Big[r + \gamma\Big(Q^{\pi}(s',\widetilde{a}') - \alpha\log(\pi(\widetilde{a}'|s')\Big)\Big]. \tag{2.31}$$

This modification changes Q value target definition,

$$Y_t^{SAC} = r_t + \gamma\Big(\min_{k\in\{1,2\}} Q(s_{t+1},;\widetilde{a}_{t+1};\theta^{Q_k^-}) - \alpha\log(\pi(\widetilde{a}_{t+1}|s_{t+1}))\Big), \tag{2.32}$$

where $\widetilde{a}_{t+1} \sim \pi(\cdot|s_{t+1};\theta^{\pi})$.

While the policy is updated according to first value network by (2.29) in TD3, the policy is updated according to minimum value of both networks output along with the entropy regularization,

$$\mathcal{L}_i(\theta_i^{\pi}) = -\mathbb{E}_{\substack{s\sim U(\mathcal{D}) \\ \widetilde{a}\sim\pi(\cdot|s)}}\Big[\min_{k\in\{1,2\}} Q(s,\widetilde{a};\theta^{Q_k^-}) - \alpha\log(\pi(\widetilde{a}|s;\theta_i^{\pi}))\Big]. \tag{2.33}$$

The policy function is stochastic in SAC, and in practice, it is a parametrized probability disribution. Most common one is Squashed Gaussian policy to squash actions in the range $(-1,1)$ by $\tanh$ function. This is parametrized by its mean and standart deviation, i.e., $\theta^{\pi} = (\theta^{\mu},\theta^{\sigma})$. Actions are then sampled as follows:

$$a(s) = \tanh(\mu(s;\theta^{\mu}) + \eta\odot\sigma(s;\theta^{\sigma})), \quad \eta\sim\mathcal{N}(0,I), \tag{2.34}$$

where $\odot$ is elementwise multiplication.

Finally, we note that SAC method does not include policy delay, target policy smoothing and target policy network.

SAC is summarized in Algorithm 4.

Algorithm 4: Soft Actor-Critic

Initialize: $\mathcal{D}$, N, N_{replay}, θ^{π}, θ_1^Q, θ_2^Q, α

$\theta_1^{Q^-} \leftarrow \theta_1^Q$, $\theta_2^{Q^-} \leftarrow \theta_2^Q$

for *episode* $= 1, M$ **do**

Recieve initial state s_1;

for $t = 1, T$ **do**

Select action $a_t \sim \pi(\cdot|s_t; \theta^{\pi})$ Execute action a_t and recieve reward r_t and next state s_{t+1};

Store experience tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ to $\mathcal{D}$;

Sample random batch with N transitions from $\mathcal{D}$ as $\mathcal{D}_r$;

Sample next actions for value target $\widetilde{a}_{j+1} \sim \pi(\cdot|s_{j+1}; \theta^{\pi}) \quad \forall e_j \in \mathcal{D}_r$;

$$Y_{jk} = \begin{cases} r_j & \text{if } s_{j+1} \text{ terminal} \\ r_j + \gamma\Big(Q(s_{j+1}, \widetilde{a}_{j+1}; \theta_k^{Q^-}) - \alpha \log(\widetilde{a}_{j+1}|s_{j+1}; \theta^{\pi})\Big) & \text{otherwise} \end{cases}$$

$\forall e_j \in \mathcal{D}_r \; \forall k \in \{1, 2\}$

$Y_j = \min(Y_{j1}, Y_{j2})$;

Update θ_1^Q θ_2^Q by seperately minimizing

$\frac{1}{N} \sum_{e_j \in \mathcal{D}_r} \left(Y_j - Q(s_j, a_j; \theta_k^Q)\right)^2 \quad \forall k \in \{1, 2\}$ for a single step;

Sample actions for policy update, $\widetilde{a}_j \sim \pi(\cdot|s_j; \theta^{\pi}) \quad \forall e_j \in \mathcal{D}_r$;

Update θ^{π} by maximizing

$\frac{1}{N} \sum_{e_j \in \mathcal{D}_r} [\min_{k \in \{1,2\}} Q(s_j, \widetilde{a}_j; \theta^{Q_k}) - \alpha \log(\pi(\widetilde{a}_j|s_j; \theta^{\pi}))]$ for a single step;

Update target networks:

$\theta_k^{Q^-} \leftarrow \tau\theta_k^Q + (1 - \tau)\theta_k^{Q^-} \quad \forall k \in \{1, 2\}$;

end

end

CHAPTER 3

NEURAL NETWORKS AND DEEP LEARNING

Through increasing computing power, deep neural networks dominated machine learning since much more data can be handled in this way. As a subfield of machine learning, the term deep learning emerged from the idea of machine learning using deep neural networks. The recent success in computer vision, natural language processing, reinforcement learning etc. was possible thanks to deep neural networks.

Despite tons of variants, a neural network is defined as a parametrized function approximator inspried by biological neurons. The first models of neural network developed by a neurophysiologist Warren McCulloch and a mathematician Walter Pitts in 1943 [27]. However, the idea of neural network known today arised after development of a simple binary classifier called perceptron invented by Rosenblatt et al. [37]. Perceptron is a learning framework inspired by human brain. Although there are many types of neural networks, they are commonly based on linear transformations and nonlinear activations.

Neural networks can approximate any nonlinear function if they are designed as complex as required. In deep learning, parameters are updated by backpropagation algorithm to minimize the loss predefined by designer, while loss function depends on output of the network.

3.1 Backpropagation and Numerical Optimization

Neural networks are composed of weight parameters. Learning is the process of updating weights to give desired behavior. This behavior is represented in a loss function. Thus, learning is nothing but minimization of loss by numerical optimization methods.

In order to minimize a loss function, its gradient with respect to weight parameters needs to be calculated. These gradients are obtained by chain rule of basic calculus. Therefore, gradient information propagates backward, and this process is called backpropagation.

3.1.1 Stochastic Gradient Descent Optimization

Gradient descent minimizes the loss function $\mathcal{L}$ by updating the weight parameters, say, θ to opposide of gradient direction with a predefined learning rate η,

$$\theta \leftarrow \theta - \eta \nabla \mathcal{L}(\theta). \tag{3.1}$$

In machine learning problems, loss functions have usually summed form of sample losses $\mathcal{L}_i$:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i(\theta). \tag{3.2}$$

Stochastic Gradient Descent approximate gradient of loss function by sample losses and updates parameters accordingly,

$$\theta \leftarrow \theta - \eta \nabla \mathcal{L}_i(\theta) \quad \forall i \in \{1, 2, \cdots N\}. \tag{3.3}$$

However, in practice, mini-batches are used to estimate loss gradient. In that case, batches with size N_b are sampled from instances,

$$\mathcal{L}_j(\theta) = \frac{1}{N_b} \sum_{i=1+(j-1)N_b}^{jN_b} \mathcal{L}_i(\theta), \tag{3.4}$$

and updates are performed accordingly,

$$\theta \leftarrow \theta - \eta \nabla \mathcal{L}_j(\theta) \quad \forall j \in \{1, 2, \cdots \left\lfloor \frac{N}{N_b} \right\rfloor\}. \tag{3.5}$$

3.1.2 Adam Optimization

Adam [23] (short for Adaptive Moment Estimation) is a variant of stochastic gradient descent as improvement of RMSProp [21] algorithm. It scales the learning rate using second moment of gradients as in RMSprop and uses momentum estimation for both first and second moment of gradients.

It is one of mostly used optimization method in deep learning nowadays. Adam adjusts step length based on training data to overcome issues arised due to stochastic updates in order to accelerate training and make it robust. It is summarized in Algortihm 5. Note that $\odot$ and $\oslash$ are elementwise multiplication and division respectively.

Algorithm 5: Adam Optimization Algorithm

Initialize: Learning Rate η, Moving average parameters β_1, β_2

Initial Model parameters θ_0

Initial first and second moment of gradients $m \leftarrow 0$, $v \leftarrow 0$

Initial step $j \leftarrow 0$

while θ_j *not converged* **do**

- $j \leftarrow j + 1$
- $g_j \leftarrow \nabla \mathcal{L}_j(\theta)$ (Obtain gradient)
- $m_j \leftarrow \beta_1 m_{j-1} + (1 - \beta_1) g_j$ (Update first moment estimate)
- $v_j \leftarrow \beta_2 v_{j-1} + (1 - \beta_2) g_j \odot g_j$ (Update second moment estimate)
- $\hat{m}_j \leftarrow \frac{m_j}{1-\beta_1^j}$ (First moment bias correction)
- $\hat{v}_j \leftarrow \frac{v_j}{1-\beta_2^j}$ (Second moment bias correction)
- $\theta_j \leftarrow \theta_{j-1} - \eta \hat{m}_j \oslash (\hat{v}_j + \epsilon)$ (Update parameters)

end

3.2 Building Units of Neural Networks

3.2.1 Perceptron

Perceptron is a binary classifier model. In order to allocate input $x \in \mathbb{R}^{1 \times d_x}$ into a class, a linear model is generated with linear transformation weights $W \in \mathbb{R}^{d_x}$ and bias $b \in \mathbb{R}$ as

$$y = \phi(xW + b). \tag{3.6}$$

where ϕ is called activation function. For perceptron, it is defined as step function,

$$\phi(a) = \begin{cases} 1, & \text{if } a \geq 0, \\ 0, & \text{otherwise}, \end{cases} \tag{3.7}$$

while other functions like sigmoid, hyperbolic tangent or ReLU can also be defined.

A learning algorithm of a perceptron aims determining the weight W and bias b. It is best motivated by error minimization of data samples once a loss function is constructed.

3.2.2 Activation Functions

As in (3.7), step function is used in perceptron. However, any other nonlinearity can be used instead to capture nonlinearity in data. Commonly used activations are *sigmoid*, *hyperbolic tangent (Tanh)*, *rectified linear unit (ReLU)*, *gaussian error linear unit (GELU)*.

Sigmoid Function, defined by

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{3.8}$$

is used when an output is required to be in $(0, 1)$, like probability value. However, it has small derivative values at value near 0 and 1.

Hyperbolic Tangent, defined by

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \tag{3.9}$$

is used when an output is required to be in $(-1, 1)$. It has similar behavior with sigmoid function except it is zero centered. Their difference is visualized in Figure 3.1a.

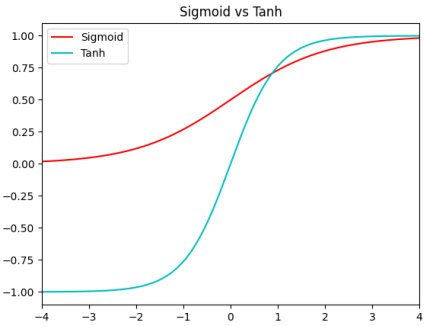
ReLU, defined by

$$\text{ReLU}(x) = \max(0, x), \tag{3.10}$$

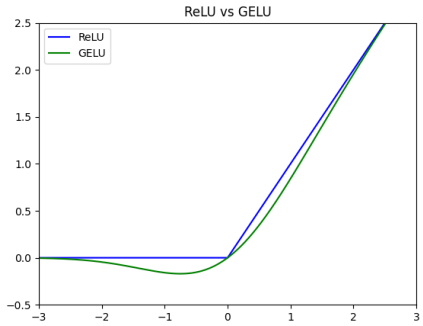
is a simple function mapping negative values to zero while passing positive values as it is. It is computationally cheap and allows to train deep and complex networks [15].

GELU, defined by

$$\text{GELU}(x) = x\Phi(x) = \frac{x}{2}\left[1 + \text{erf}\Big(\frac{x}{\sqrt{2}}\Big)\right], \tag{3.11}$$



(a) Sigmoid and Tanh functions



(b) ReLU and GELU functions

Figure 3.1: Activation Functions

is smoothed version of ReLU function. It is continuous but non-convex and has several advantages [20]. ReLU and GELU are visualized in Figure 3.1b.

3.2.3 Softmax

Softmax function is generalization of the sigmoid function and used for multinomial classifications. Given $x \in \mathbb{R}^K$, softmax assigns probability to each element of x. Resulting vectors elements sum up to 1. It is defined as,

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}. \tag{3.12}$$

3.2.4 Layer Normalization

Layer normalization used to overcome instability and divergence during learning [6]. Given an input $x \in \mathbb{R}^K$, mean and variance statistics are evaluated along the dimensions as

$$\mu = \frac{1}{K}\sum_{n=1}^{K} x_k \quad \text{and} \quad \sigma^2 = \frac{1}{K}\sum_{n=1}^{K}(x_k - \mu)^2. \tag{3.13}$$

Then, the input is first scaled to have zero mean and unity variance along dimensions. The term ϵ is added to prevent division by zero. Optionally, the result is scaled by elementwise multiplication by $\gamma \in \mathbb{R}^K$ and addition of $\beta \in \mathbb{R}^K$ as

$$\text{LN}(x) = \frac{x - \mu}{\sigma + \epsilon}\gamma + \beta, \tag{3.14}$$

where γ and β are learnable (weight) parameters.

3.3 Neural Network Types

3.3.1 Feed Forward Neural Networks (Multilayer Perceptron)

Feed forward neural layers is multidimensional generalization of perceptron. Generally, a neural layer might output multiple values (say $o \in \mathbb{R}^{1\times d_o}$) as vector from input (say $x \in \mathbb{R}^{1\times d_x}$). Such a setting forces parameter $W \in \mathbb{R}^{d_x\times d_o}$ to be a matrix. Moreover, activation function is not necessarily a step function. It can be any nonlinear function like sigmoid, tanh, ReLU etc. Feed Forward Neural Networks are the generalization of perceptron to approximate any function. Neural layers are stacked to construct deep feed forward neural network. It defines a nonlinear mapping $y = \phi(x;\theta)$ between input x and output y, parametrized by θ that includes (learnable) parameters like weights, bias and possibly others if defined.

Assuming input signal is $x \in \mathbb{R}^{1\times d_x}$ (output of previous layer), activation value of the layer ($h \in \mathbb{R}^{1\times d_h}$) is calculated by linear transformation (weight W and bias b) followed by nonlinear activation ϕ,

$$h = \phi(xW + b), \tag{3.15}$$

where ϕ is applied elementwise.

3.3.2 Residual Feed Forward Neural Networks

As Feed Forward Networks becomes deeper, optimizing weights gets difficult. Therefore, people come with the idea of residual connections [18]. For a fixed number of stacked layers (usually 2 which is single skip), input and output of the stack is summed up for next calculations. Replacing feed forward layers with other types yield different kind of residual network. The difference is demonstrated in Figure 3.2, where ϕ corresponds to activation and yellow block means no activation.

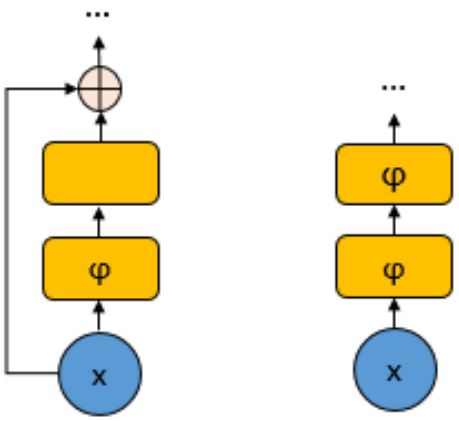


Figure 3.2: Deep Residual Feed Forward Network with single skip connection (left) and Deep Feed Forward Network (right)

3.3.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) [38] are one type of neural network to process sequential data. It is specialized for data having sequential topology.

In FFNN layer, output only depends on its input, while Recurrent Layer output depends on both input at time t and its output in previous time step $t-1$.

RNN can be thought as multiple copies of same network which passes message to its successor through time. A RNN layer is similar to a FFNN layer as in (3.15), except that input is concatenation of the output feedback and input itself.

Given an input sequence $x \in \mathbb{R}^{T \times d_x}$ with time length T, an output sequence $h \in \mathbb{R}^{T \times d_h}$ is evaluated recursively by

$$h_t = \phi(h_{t-1}\tilde{W} + x_t W + b), \tag{3.16}$$

where nonlinear activation ϕ applied elementwise as in (3.15) again. To begin with, the initial output h_0 can be either parametrized or assigned to a zero vector. A comparison between FFNN and RNN layer is visualized in Figure 3.3.

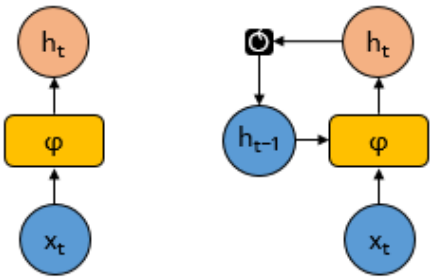


Figure 3.3: Feed Forward Layer (left) and Recurrent Layer (right) illustration [31]

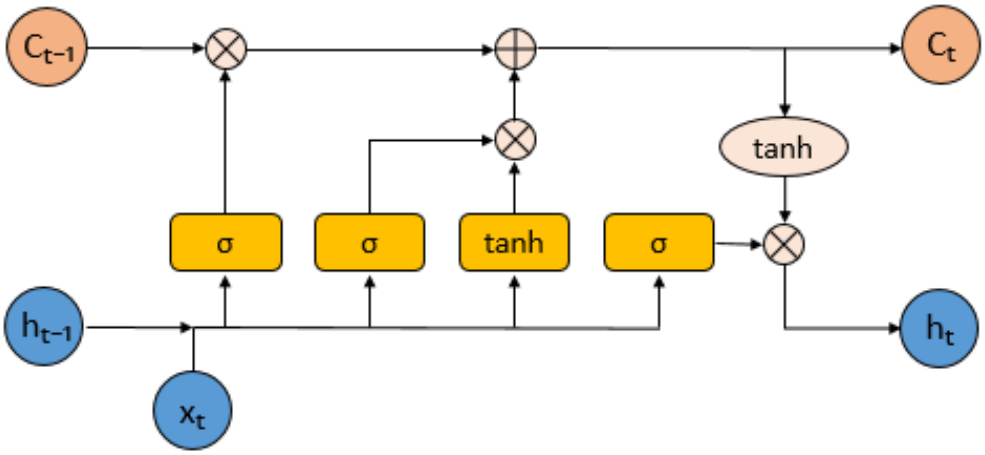


Figure 3.4: LSTM Cell [31]

3.3.3.1 Long Short Term Memory

Conventional RNNs have, in general, the problem with vanishing/exploding gradient [31]. As the sequence gets longer, effect of initial inputs in sequence decreases. This causes a long term dependence problem. Furthermore, if information from initial inputs required, gradients either vanish or explode. In order to overcome this problem another architecture is developed: Long Short Term Memory (LSTM) [22].

LSTM is a special type of RNN. It is explicitly designed to allow learning long-term dependencies. A single LSTM cell has four neural layers while a vanilla RNN layer has only one neural layer. In addition to the hidden state h_t, there is another state called cell state C_t. Information flow is controlled by three gates.

Forget Gate controls past memory. According to input, past memory is either kept

or forgotten. The sigmoid function (σ) is generally used as an activation function:

$$f_t = \sigma([h_{t-1}; x_t]W_f + b_f). \tag{3.17}$$

Hyperbolic tangent layer creates new candidate of cell state from the input:

$$\hat{C}_t = \tanh([h_{t-1}; x_t]W_C + b_C). \tag{3.18}$$

Input Gate controls contribution from input to cell state (memory):

$$i_t = \sigma([h_{t-1}; x_t]W_i + b_i). \tag{3.19}$$

Once what are to be forgotten and added decided, cell state is updated accordingly,

$$C_t = f_t \odot C_{t-1} + i_t \odot \hat{C}_t. \tag{3.20}$$

using input gate and forget gate outputs. Note that $\odot$ is elementwise multiplication.

Output Gate controls which part of new cell state to be output:

$$o_t = \sigma([h_{t-1}; x_t]W_o + b_o). \tag{3.21}$$

Finally, cell state is filtered by hyperbolic tangent to push values to be in $(-1, 1)$ before evaluating the hidden state h_t using output gate outputs:

$$h_t = o_t \odot \tanh(C_t). \tag{3.22}$$

3.3.4 Attention Based Networks

As stated earlier, recurrent neural networks are prone to forget long term dependencies. LSTM and other variants are invented to overcome such problems. However, they cannot attend specific parts of the input. Therefore, researchers come with the idea of weighted averaging all states through time where weights depends on both input and output. Let the input sequence $x \in \mathbb{R}^{T \times d_x}$ with time length T be encoded to $h \in \mathbb{R}^{T \times d_h}$. The context vector is calculated using weight the vector $\alpha \in \mathbb{R}^{1 \times T}$ including a scalar for each time step, then attention output is,

$$\text{Attention}(q, h) = \alpha h. \tag{3.23}$$

Attention weight $\alpha \in \mathbb{R}^T$ is calculated using hidden sequence $h \in \mathbb{R}^{T \times d_H}$ and query $q \in \mathbb{R}^{1 \times d_Q}$ with a predefined a arbitrary function ϕ depending on choice. Query can be hidden state itself (self attention) or any input assumed to help attention weighting. In general form,

$$\alpha = \phi(q, h; \theta_\alpha). \tag{3.24}$$

3.3.4.1 Transformer

The Transformer was proposed in "Attention is All You Need" [49] paper. Unlike recurrent networks, this architecture is solely built on attention layers.

A transformer layer consists of feed-forward and attention layers, which makes the mechanism special. Like RNNs, it can be used as both encoder and decoder. While encoder layers attend to itself, decoder layers attends both itself and encoded input.

Attention Layer is a mapping from 3 vectors: query $Q \in \mathbb{R}^{T_q \times d_k}$, key $K \in \mathbb{R}^{T \times d_k}$ and value $V \in \mathbb{R}^{T \times d_v}$ to output, where T is input time length, T_q time length of query, d_k and d_v are embedding dimensions. Output is weighted sum of values V through time dimension while weights are evaluated by compatibility metric of query Q and key K. In vanilla transformer, compatibility of query and key is evaluated by dot product, normalizing by $\sqrt{d_k}$. For a query, dot product with all keys are evaluated, then softmax function is applied to get weights of values,

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{3.25}$$

This approach is called Scaled Dot-product Attention.

Instead of performing a single attention, **Multi-Head Attention** projects keys, queries and values linearly from d_m dimensional vector space to h different spaces using projection matrices. In this case, attention is performed h times, and results are then concatenated and linearly projected to final values of the layer. Projection matrices are model parameters, $W_i^Q \in \mathbb{R}^{d_m \times d_k}$, $W_i^K \in \mathbb{R}^{d_m \times d_k}$, $W_i^V \in \mathbb{R}^{d_m \times d_v}$ for $i = 1, ..., h$. Also output matrix is used to project multiple values into single one, $W^O \in \mathbb{R}^{hd_v \times d_m}$.

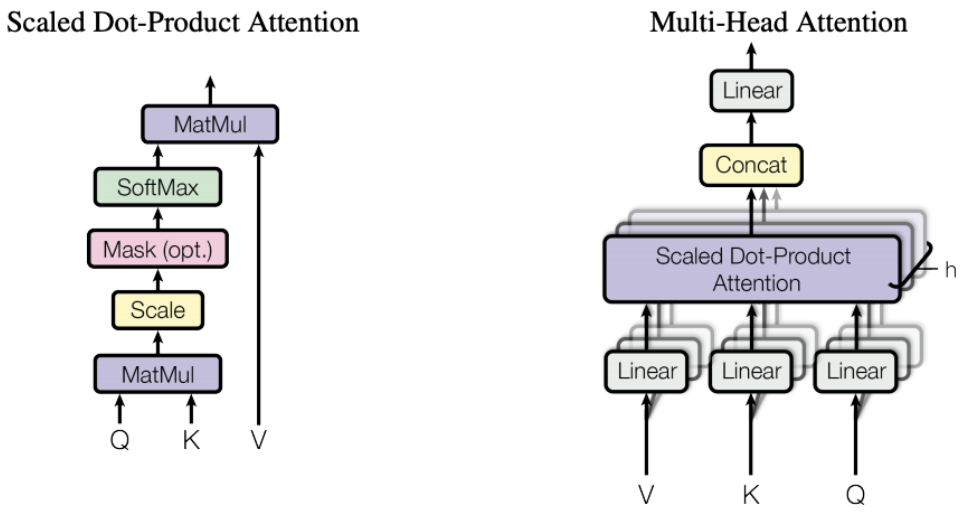


Figure 3.5: Scaled Dot-Product Attention (left) and Multi-Head Attention (right) [49]

Mathematically,

$$\begin{aligned} \mathrm{MHA}(Q,K,V) &= \textbf{Concat}(\mathrm{head}_1, \mathrm{head}_2, ...\mathrm{head}_h)W^O \\ \mathrm{head}_i &= \textbf{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} . \tag{3.26}$$

Scaled dot-product attention and Multi-Head attention are demonstrated in Figure 3.5

A transformer layer contains **Feed Forward Layer**, containing two linear transformations (W_1, b_1 and W_2, b_2) with ReLU-like activation:

$$\mathrm{FFN}(x) = \textbf{ReLU}(xW_1 + b_1)W_2 + b_2. \tag{3.27}$$

Encoder Layer starts with a residual self attention layer. Self attention means that query, key and value are the same vectors. This is followed by feed forward neural layer. Both sublayers are employed with resudial connection with layer normalization. In other words, summation of layer input and output is passed through layer normalization:

$$\begin{aligned} a =& \mathrm{LN}(x + \mathrm{MHA}(x,x,x)), \\ y =& \mathrm{LN}(a + \mathrm{FFN}(a)). \end{aligned} \tag{3.28}$$

Apart from encoder layers; query, key and value inputs may vary depending on design. In general,

$$\begin{aligned} a =& \mathrm{LN}(x + \mathrm{MHA}(q,k,v)), \\ y =& \mathrm{LN}(a + \mathrm{FFN}(a)). \end{aligned} \tag{3.29}$$

Since there are no recurrent or convolutional architecture in the model, sequential information needs to be embedded. For this purpose, **Positional encoding**s are used. They have same dimension with as the input x, so that input embeddings can be added to at the beginning of encoder or decoder stacks. Let the positional encoding $\mathrm{PE} \in \mathbb{R}^{T \times d_m}$. For tth position in time, and ith position in embedding axis ($i \in \mathbb{N}$), PE is defined as follows:

$$\mathrm{PE}_{t,i} = \begin{cases} \sin(t/10000^{i/d_m}), & \text{if } i \equiv 0 \pmod 2, \\ \cos(t/10000^{(i-1)/d_m}), & \text{if } i \equiv 1 \pmod 2, \end{cases} \tag{3.30}$$

as proposed in the original paper [49].

3.3.4.2 Pre-Layer Normalized Transformer

Original transformer architecture includes layer normalization operations after attention and feed-forward layers. It is unstable since the values of gradients of output layers are high. Pre-Layer Normalized Transformer is proposed by [52] by carrying layer normalization operation to in front of attention and feed-forward layers. Moreover, Parisotto et al. Xiong et al. [33] propose gated transformer which also includes layer normalizations before attention and feedforward layer. They also state that although gated architecture improves many reinforcement learning (RL) tasks drastically, and non-gated pre-layer normalized transformer are also much better than vanilla transformer.

In pre-layer normalized transformer, encoder equations are:

$$\begin{aligned} a =& x + \mathrm{MHA}(\mathrm{LN}(x), \mathrm{LN}(x), \mathrm{LN}(x)) \\ y =& a + \mathrm{FFN}(\mathrm{LN}(a)) \end{aligned}. \tag{3.31}$$

A simple pre-layer normalized transformer encoder layer is demonstrated in Figure 3.6. The difference between post-layer and pre-layer transformer encoders are also visualized in Figure 3.7

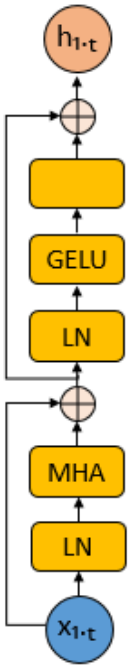


Figure 3.6: Pre-LN Transformer encoder layer with GELU activation

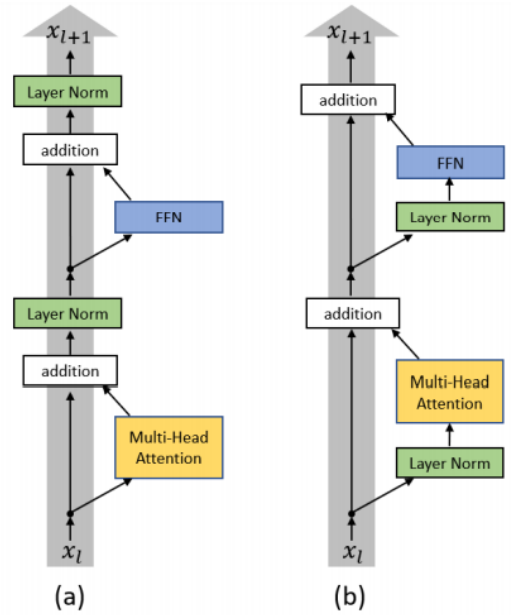


Figure 3.7: (a) Post-LN Transformer layer, (b) Pre-LN Transformer layer [52]

CHAPTER 4

BIPEDAL WALKING BY TWIN DELAYED DEEP DETERMINISTIC POLICY GRADIENTS

4.1 Details of the Environment

BipedalWalker-v3 and *BipedalWalker-Hardcore-v3* are two simulation environments of a bipedal robot, with relatively flat course and obstacle course respectively. Dynamics of the robot are exactly identical in both environments. Our task is to solve hardcore version where the agent is expected to learn to run and walk in different road conditions. Components of the hardcore environment is visualized in Figure 4.1.

The robot has kinematic and lidar sensors, and has deterministic dynamics.

Observation Space contains hull angle, hull angular velocity, hull translational velocities, joint positions, joint angular speeds, leg ground concats and ten lidar rangefinder measurements. Details and their constraints are summarized in Table 4.1.

The robot has two legs with two joints at knee and hip. Torque is provided to knee and pelvis joints of both legs. These four torque values forms the **Action Space**, presented in Table 4.2 with their constraints.

Reward Function is a bit complex in the sense that the robot should run fast with little energy while it should not stumble and fall to ground. Directly proportional to distance traveled forward, +300 points given if agent reaches to end of the path. However, -10 points (-100 points in the original version) if agent falls, and small amount of negative reward proportional to absolute value of applied motor torque τ (to prevent applying unnecessary torque). Lastly, the robots gets negative reward

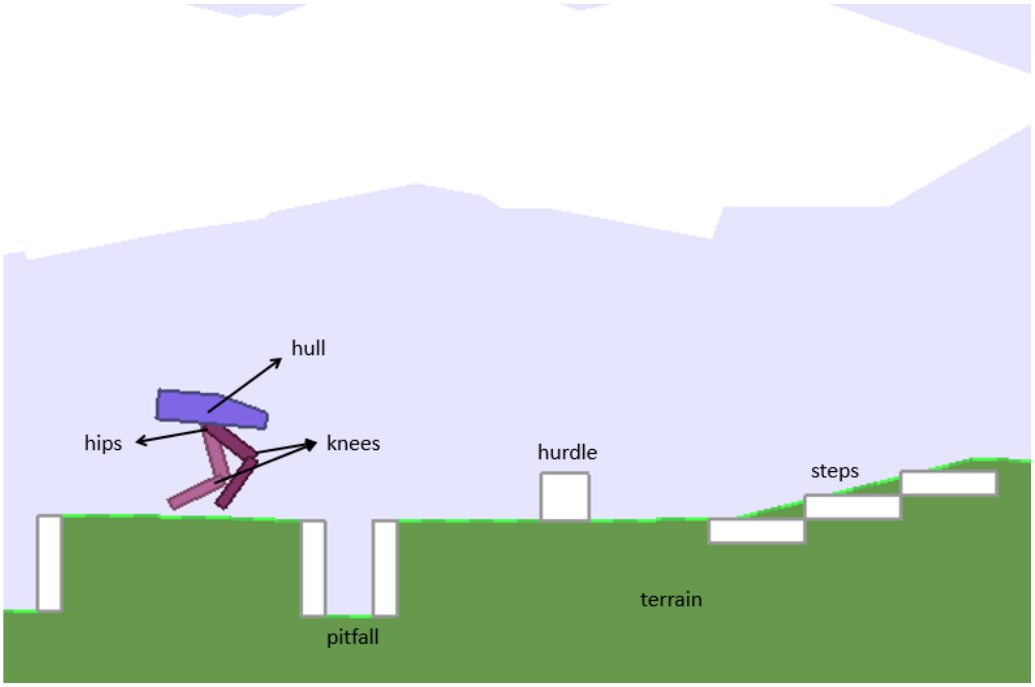


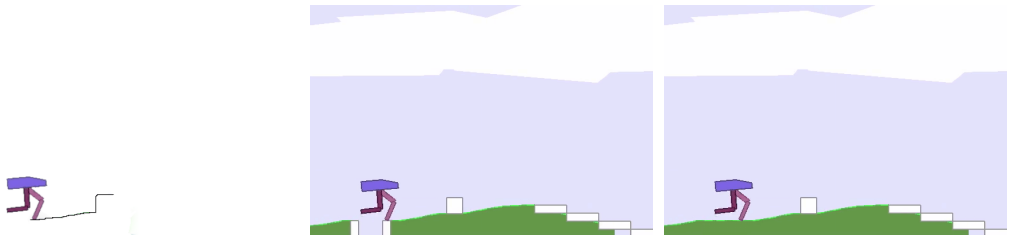
Figure 4.1: Bipedal Walker Hardcore Components [3]

Num	Observation	Interval
0	Hull Angle	$[-\pi, \pi]$
1	Hull Angular Speed	$[-\infty, \infty]$
2	Hull Horizontal Speed	$[-1, 1]$
3	Hull Vertical Speed	$[-1, 1]$
4	Hip 1 Joint Angle	$[-\pi, \pi]$
5	Hip 1 Joint Speed	$[-\infty, \infty]$
6	Knee 1 Joint Angle	$[-\pi, \pi]$
7	Knee 1 Joint Speed	$[-\infty, \infty]$
8	Leg 1 Ground Contact Flag	$\{0, 1\}$
9	Hip 2 Joint Angle	$[-\pi, \pi]$
10	Hip 2 Joint Speed	$[-\infty, \infty]$
11	Knee 2 Joint Angle	$[-\pi, \pi]$
12	Knee 2 Joint Speed	$[-\infty, \infty]$
13	Leg 2 Ground Contact Flag	$\{0, 1\}$
14-23	Lidar measures	$[-\infty, \infty]$

Table 4.1: Observation Space of Bipedal Walker

Num	Observation	Interval
0	Hip 1 Torque	$[-1, 1]$
1	Hip 2 Torque	$[-1, 1]$
2	Knee 1 Torque	$[-1, 1]$
3	Knee 2 Torque	$[-1, 1]$

Table 4.2: Action Space of Bipedal Walker



(a) Perspective (b) Pitfall Behind (c) No Pitfall Behind

Figure 4.2: Perspective of agent and possible realities

propotional to the absolute value of the hull angle θ_{hull} for reinforcing to keep the hull straigth. Mathematically, reward is;

$$R = \begin{cases} -10, & \text{if falls,} \\ 130\Delta x - 5|\theta_{hull}| - 0.00035|\tau|, & \text{otherwise.} \end{cases} \tag{4.1}$$

4.1.1 Partial Observability

The environment is partially observable due to following reasons.

- The agent is not able to track behind with the lidar sensor. Unless it has a memory, it cannot know whether a pitfall or hurdle behind. Illustration is shown in Figure 4.2.
- There is no accelerometer sensor. Therefore, the agent do not know whether it is accelerating or decelerating.

In DRL, partial observability is handled by two ways in literature [10]. The first is incorporating fixed number of last observations while the second way is updating hidden belief state using recurrent-like neural network at each time step. Our approach is somewhere in between. We pass fixed number of past observations into LSTM and Transformer networks for both actor and critic.

4.1.2 Reward Sparsity

Rewards given to the agent is sparse in some circumstances, in the following sense:

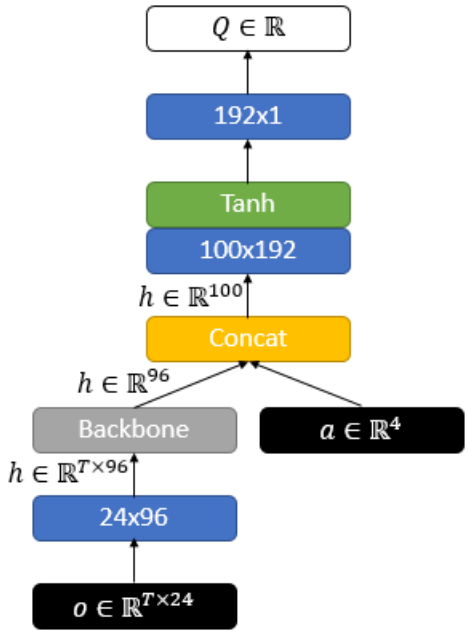
- Overcoming big hurdles requires a very specific move. The agent should explore many actions when faced with a big hurdle.
- Crossing pitfalls also require a specific move but not as complex as big hurdles.

4.1.3 Modifications on Original Envrionment Settings

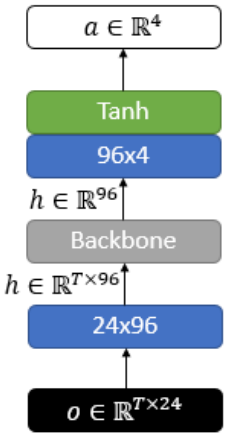
It is difficult to find an optimal policy directly with available setting. There are few studies in the literature demonstrating a solution for *BipedalWalker-Hardcore-v3*. The following modifications are proposed.

- In original version, agent gets -100 points when its hull hits the floor. In order to make the robot more greedy, this is changed to -10 points. Otherwise, agent cannot explore environment since it gets too much punishment when failed and learns to do nothing.
- Time resolution of simulation is halved (from 50 Hz to 25 Hz) by only observing last of each two consecutive frames using a custom wrapper function. Since there is not a high frequency dynamics, this allows nothing but speeding up the learning process.
- In original implementation, an episode has a time limit. Once this limit is reached, simulation stops with terminal state flag. On the other hand, when agent fails before the time limit, the episode ends with terminal state flag too. In the first case, the terminal state flag causes instability since next step's value is not used in value update, since it is not a logical terminal, just time up indicator. The environment changed such that terminal state flag is not given in this case unless agent fails.

Once those modifications are applied on the environment, we observed quite improvement on both convergence and scores.



(a) Critic Architecture



(b) Actor Architecture

Figure 4.3: Neural Architecture Design

4.2 Proposed Neural Networks

Varying neural backbones are used to encode state information from observations for both actor and critic networks. In critic network, actions are concatenated by state information coming from backbones. Then, this concatenated vector is passed through feed forward layer with hyperbolic tangent activation then through a linear layer with single output. Before feeding observations to backbone, they are passed through a linear layer with 96 dimensional output. However, for only LSTM backbones, this layer is not used and observations are passed to LSTM backbone as it is. In actor network, backbone is followed by a single layer with tanh activation for action estimation. Again, observations are passed through a linear layer with 96 dimensional output, and this is not valid for LSTM. Critic and Actor networks are illustrated in Figure 4.3.

As backbones, following networks are proposed.

4.2.1 Residual Feed Forward Network

Incoming vector is passed through 2 layers with 192 dimensional hidden size and 96 dimensional output with single skip connection, where there is GELU activation at first layer.

4.2.2 Long Short Term Memory

Sequence of incoming vectors is passed though single vanilla LSTM layer with 96 dimensional hidden state. Output at last time step is considered as belief state.

4.2.3 Transformer (Pre-layer Normalized)

Sequence of incoming vectors is passed through single pre-layer normalized transformer layer with 192 dimensional feed forward layer with GELU activation after addition of positional encodings. In multi-head attention, 4 head is used and size of queries, keys and values are 8 for each head. During multi-head attention, only the last observation is fed as query so that attentions are calculated only for last state. Its output is considered as belief state. Note that excluding multi-head attention for this network gives our residual feed-forward neural network.

4.2.4 Summary of Networks

The designed networks have close number of parameters to make fair comparison among them. Number of trainable parameters for designed networks are summarized in Table 4.3.

In addition, we designed transformer network to be same as residual feed forward network once multi-head attention and preceeding layer normalization is removed. It can be easily demonstrated by looking at Figure 3.2 (left one, layer normalization is added in the implementation) and Figure 3.6.

Table 4.3: Number of trainable parameters of neural networks

Network	Actor Parameters	Critic Parameters
RFFNN	40512	59328
LSTM	47616	66432
TRSF	52992	71808

4.3 RL Method and Hyperparameters

TD3 and SAC is used as RL algorithm. Hyperparameters are selected by grid search and best performing values are used. Adam optimizer is used for optimization. For all models, agents are trained by 8000 episodes.

In TD3, as exploration noise, Ornstein-Uhlenbeck noise is used, and standart deviation is multiplied by 0.9995 at the end of each episode. For LSTM and Transformer, last 6 observations are used to train agent. More than 6 observations made TD3 algorithm diverge.

In SAC model, squashed gaussian policy is implemented. Therefore, along with layer giving mean actions, an additive layer is implemented for the standart deviation. For LSTM and Transformer, last 6 observations and 12 observations are used to train agent.

All hyperparameters are found after a trial-error process considering the literature. They are summarized in Table 4.4 and Table 4.5. Training sessions are run by multiple times to make comparison fair among models since some training sessions fail to converge and some yield worse results.

4.4 Results

For each episode, episode scores are calculated by summing up rewards of each time step. In Figure 4.4a and Figure 4.4b, a scatter plot is visualized for each model's episode scores. In Figure 4.5a and Figure 4.5b, moving average and standard devia-

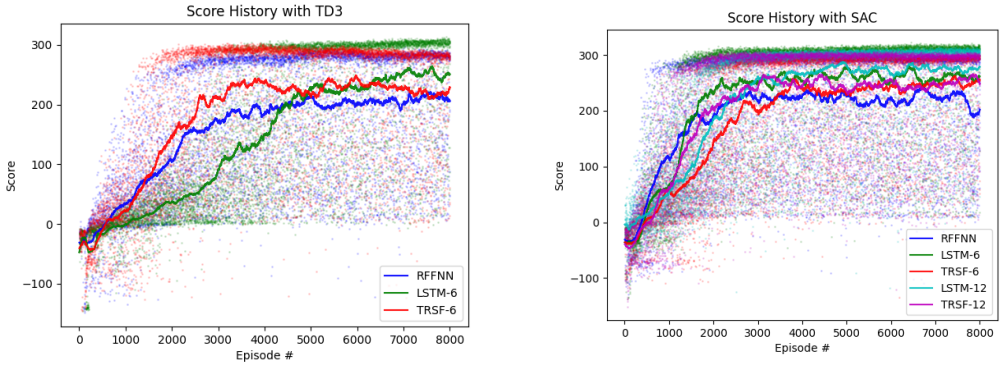
Table 4.4: Hyperparmeters and Exploration of Learning Processes for TD3

Hyperparameter \ Model	RFFNN	LSTM	Transformer
η (Learning Rate)	4.0×10^{-4}		
β (Momentum)	$(0.9, 0.999)$		
γ (Discount Factor)	0.98		
N_{replay} (Replay Buffer Size)	500000		
N (Batch Size)	64		
d (Policy Delay)	2		
σ (Policy smoothing std)	0.2		
τ (Polyak parameter)	0.01		
Exploration	$OU(\theta = 4.0, \sigma = 1.0)$		

Table 4.5: Hyperparmeters and Exploration of Learning Processes for SAC

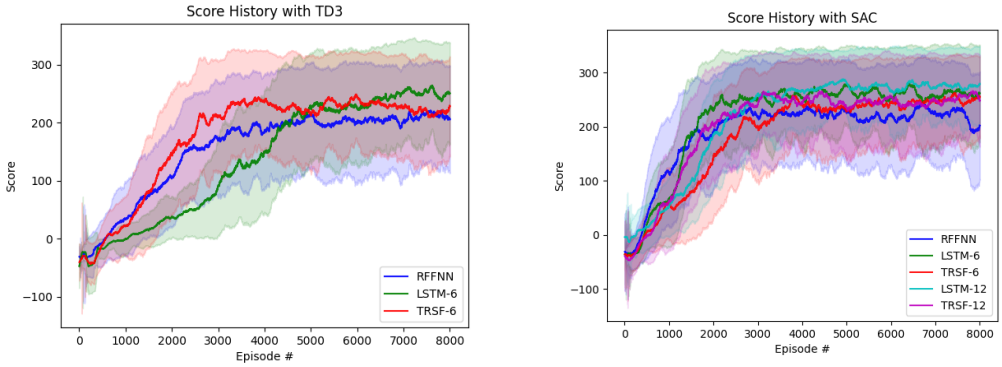
Hyperparameter \ Model	RFFNN	LSTM	Transformer
η (Learning Rate)	4.0×10^{-4}		
β (Momentum)	$(0.9, 0.999)$		
γ (Discount Factor)	0.98		
N_{replay} (Replay Buffer Size)	500000		
N (Batch Size)	64		
τ (Polyak parameter)	0.01		
α (Entropy Temperature)	0.01		

tion is visualized for each model's episode scores.



(a) TD3 (b) SAC

Figure 4.4: Scatter Plot with Moving Average for Episode Scores (Window length: 200 episodes) for SAC



(a) TD3 (b) SAC

Figure 4.5: Moving Average and Standard Deviation for Episode Scores (Window length: 200 episodes) for TD3

RFFNN seems enough to make agent walk, although there exist partial observability in the environment. That model reaches around 221 points in average with TD3 and 245 points with SAC.

Transformer models yield better performance compared to RFFNN. 249 points are reached by TD3 and 262 points reached by SAC model when 6 last observations are fed to the model. 266 points are obtained when last 12 observations are fed to model with SAC.

LSTM model yield the best results by reaching 264 points with TD3 and exceeds 280 points with SAC when 6 last observations are fed to model. 287 points are obtained when last 12 observations are fed to the model with SAC.

Table 4.6: Best checkpoint performances with 100 test simulations

RL Method	Model	Episode	Avg. Score
TD3	RFFNN	6600	207
SAC	RFFNN	7600	219
TD3	TRSF-6	6400	222
SAC	TRSF-6	6800	254
SAC	TRSF-12	6000	270
TD3	LSTM-6	7000	242
SAC	LSTM-6	7600	268
SAC	LSTM-12	7200	298

These scores are maximum obtained average scores of concurrent 200 episodes during training. However, while traning, model changes and agent still tries to explore envrionment. Officially, 300 points in average required in random 100 simulations to say the environment is solved [4]. Therefore, best checkpoints are evaluated for 100 concurrent simulations without exploration. Results are summarized in Table 4.6.

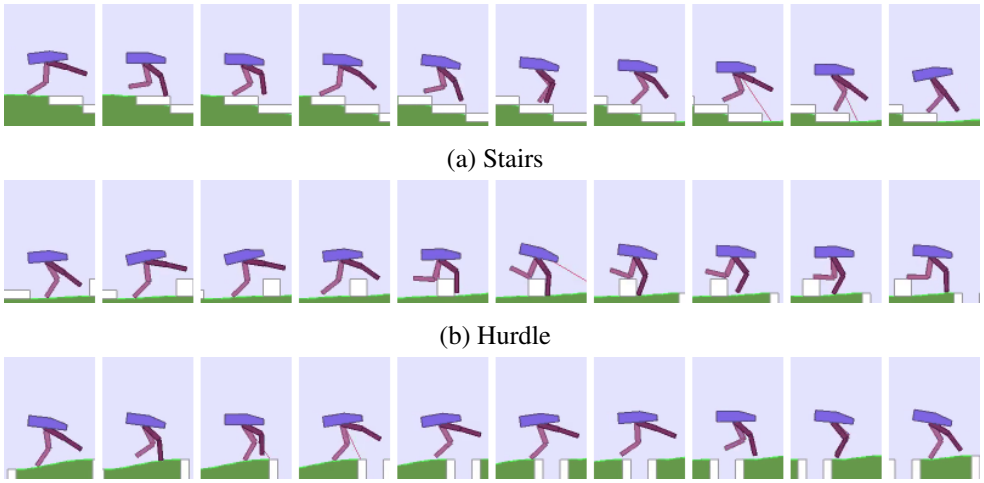
None of our models exceed 300 point limit, but LSTM-12 model with SAC almost reaches to the limit Table 4.6. However, all models partially solved problems by exceeding 200 point limit, while some simulations exceed 300 points with both TD3 (Figure 4.4a) and SAC (Figure 4.4b).

Behaviour of agents are visualized in Figure 4.6, Figure 4.7 and Figure 4.8 for SAC models. All of 3 models exhibit similar walking characteriscis.

SAC model performes better than TD3 in general, as seen in Figure 4.4a and Figure 4.4b. In addition, as seen from the moving average points, SAC agents are superior (Figure 4.5a, Figure 4.5b).

4.5 Discussion

These results are not enough to conclude on a superior neural network for all RL problems, because there are other factors such as DRL algorithm, number of episodes, network size etc. However, networks are designed to have similar sizes and good model requires to converge in less episodes. As a result, LSTM is superior to Transformer

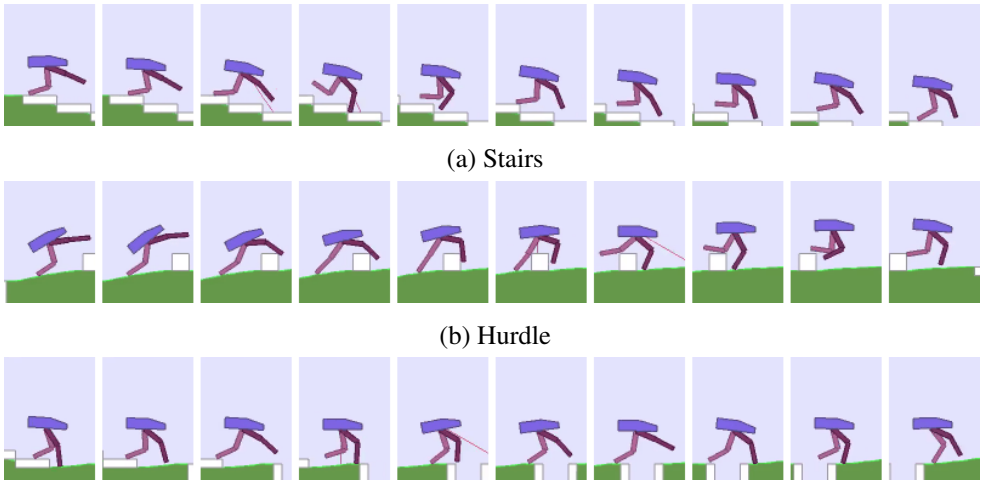


(a) Stairs

(b) Hurdle

(c) Pitfall

Figure 4.6: Walking Simulation of RFFNN model at best version with SAC



(a) Stairs

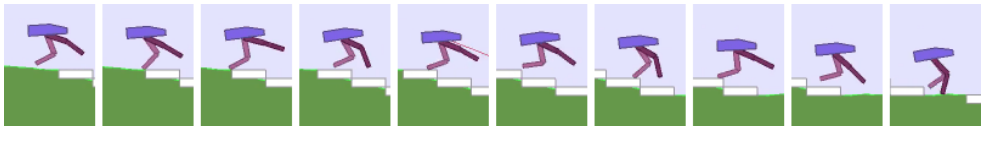
(b) Hurdle

(c) Pitfall

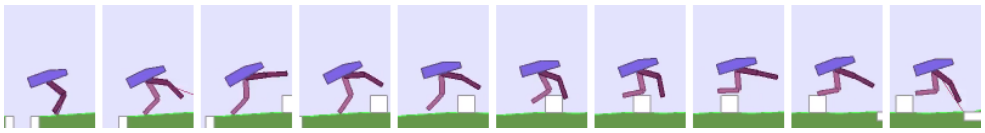
Figure 4.7: Walking Simulation of LSTM-12 model at best version with SAC

for our environment. In addition, it is possible to conclude that Transformers can be an option for partially observed RL problems. Note that this is valid where layer normalization is applied before multihead attention and feed-forward layers [52] as opposed to vanilla transformer proposed in [49].

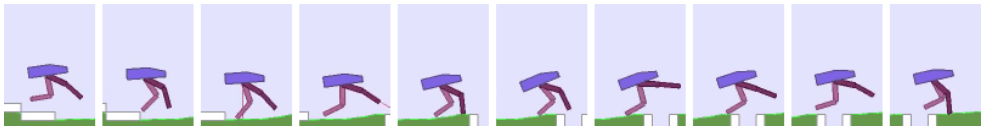
Another result is that incorprating past observations did improve performance significantly since environment is partially observable. Especially increasing history length keeps increasing performance for both LSTM and Transformer models. To address this issue better, we designed Transformer and RFFNN networks to be almost same once multi-head attention layer is removed. We observed significant performance gains as observation history is used as input to controller.



(a) Stairs



(b) Hurdle



(c) Pitfall

Figure 4.8: Walking Simulation of Transformer-12 model at best version with SAC

The environment is a difficult one. There are really few available models with solution [4]. Apart from neural networks, there are other factors affecting performance such as RL algorithm, rewarding, exploration etc. In this work, all of them are adjusted such that the environment becomes solvable. Time frequency is reduced for sample efficiency and speed. Also, the agent is not informed for the terminal state when it reaches time limit. Lastly, punishment of falling reduced, so the agent is allowed to learn by mistakes. Those modifications are probably another source of our high performance.

As RL algorithm, TD3 is selected first, since it is suitable for continuous RL. Ornstein-Uhlenbeck noise is used for better exploration since it has momentum, and variance is reduced by time to make agent learn small actions well in later episodes. However, we cannot feed 12 observation history to sequential models since TD3 failed to learn in that case. In addition, SAC is used for learning along with TD3. Results are better compared to those of TD3. SAC policy maximizes randomness (entropy) if agent cannot get sufficient reward and this allows the agent to decide where/when to explore more or less. This way, SAC handles the sparse rewards from the environment better than TD3.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In RL context, learning in a simulation is an important step for mechanical control. Usually, models are pretrained in simulation environment before learning in reality due to safety and exploration reasons. Today, RL is rarely used in real world applications due to safety and sample inefficiency problems.

In this thesis, bipedal robot walking is investigated by deep reinforcement learning due to complex dynamics in OpenAI Gym's simulation environment. TD3 and SAC algorthims are used since they are robust and well suited for continuous control. Also, environment is slightly modified by reward shaping, halving simulation frequency, cancelling terminal state information at time limit so that learning becomes easier.

The environment was difficult for exploration. Especially handling big hurdles requires very much exploration. In our results, SAC agents performed better than TD3 since it handles exploration problem by learning how much to explore for a particular state.

As stated in previous chapters, most of the real world environments are partially observable. In *BipedalWalker-Hardcore-v3*, the environment is also partially observable since agent cannot observe behind and it lacks of acceleration sensors, which is better to have for controlling mechanical systems. Therefore, we propose to use Long Short Term Memory (LSTM) and Transformer Neural Networks to capture more information from past observations unlike Residual Feed Forward Neural Network (RFFNN) using a single instant observation as input.

RFFNN model performed well thanks to carefully selected hyperparameters and mod-

ifications on the environment. However, sequential models performed much better indicating partial observability is an important issue for our environment. Among sequential models, LSTM performed better compared to Transformer agents. However, note that simple multi-head attention layer added to RFFNN lets Transformer model emerge. Deciding to use observation history with simple attention layer resulted in significant performance gains. This means that input-output design may be more important than neural network design.

Another conclusion is that Transformer model performed well enough to say it can be used in DRL problems. It is surprising because it is not succesfully used in DRL problems in general except recent architectural developments [33]. In natural language processing, this type of attention models completely replace recurrent models recently, and our results seems promising for this in DRL domain.

Today, all subfields of Deep Learning suffers from lack of analytical methods to design neural networks. It is mostly based on mathematical & logical intuition. Until such methods are developed, it seems that we need to try out several neural networks to get better models, which is the case in our work.

There might be a few possible ways to improve this work. First of all, longer observation history can be used to handle partial observability for LSTM and Transformer models. However, this makes learning slower and difficult and requires more stable RL algorithms and fine tuning on hyperparameters. Secondly, different exploration strategies might be followed. Especially parameter space exploration [35] may perform better since it works better for environments with sparse reward like our environment. Lastly, advanced neural networks and RL algorithms (specificly designed for environment) may be designed by incorprating domain knowledge.

REFERENCES

[1] Models for machine learning, https://developer.ibm.com/technologies/artificial-intelligence/articles/cc-models-machine-learning/, Accessed: 2021-09-05, December 2017.

[2] BipedalWalker-v2, https://gym.openai.com/envs/BipedalWalker-v2/, Accessed: 2021-09-05, January 2021.

[3] BipedalWalkerHardcore-v2, https://gym.openai.com/envs/BipedalWalkerHardcore-v2/, Accessed: 2021-09-05, January 2021.

[4] OpenAI Gym Leaderboard-v2, https://github.com/openai/gym/wiki/Leaderboard/, Accessed: 2021-09-05, August 2021.

[5] P. Abbeel, A. Coates, M. Quigley, and A. Ng, An Application of Reinforcement Learning to Aerobatic Helicopter Flight, in *NIPS*, 2006.

[6] J. L. Ba, J. R. Kiros, and G. E. Hinton, Layer Normalization, arXiv:1607.06450 [cs, stat], July 2016.

[7] R. Bellman, *Dynamic Programming*, Dover Publications, Mineola, N.Y, reprint edition edition, March 2003, ISBN 9780486428093.

[8] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, OpenAI Gym, arXiv:1606.01540 [cs], June 2016.

[9] R. Collobert, K. Kavukcuoglu, and C. Farabet, Torch7: A Matlab-like Environment for Machine Learning, 2011.

[10] G. Dulac-Arnold, D. Mankowitz, and T. Hester, Challenges of Real-World Reinforcement Learning, arXiv:1904.12901 [cs, stat], April 2019.

[11] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, An Introduction to Deep Reinforcement Learning, Foundations and Trends® in Machine Learning, 11(3-4), pp. 219–354, 2018, ISSN 1935-8237, 1935-8245.

[12] R. Fris, The Landing of a Quadcopter on Inclined Surfaces using Reinforcement Learning, 2020.

[13] X. Fu, F. Gao, and J. Wu, When Do Drivers Concentrate? Attention-based Driver Behavior Modeling With Deep Reinforcement Learning, arXiv:2002.11385 [cs], June 2020.

[14] S. Fujimoto, H. van Hoof, and D. Meger, Addressing Function Approximation Error in Actor-Critic Methods, arXiv:1802.09477 [cs, stat], October 2018.

[15] X. Glorot, A. Bordes, and Y. Bengio, Deep Sparse Rectifier Neural Networks, in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 315–323, JMLR Workshop and Conference Proceedings, June 2011.

[16] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, and S. Levine, Learning to Walk via Deep Reinforcement Learning, arXiv:1812.11103 [cs, stat], June 2019.

[17] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, arXiv:1801.01290 [cs, stat], August 2018.

[18] K. He, X. Zhang, S. Ren, and J. Sun, Deep Residual Learning for Image Recognition, arXiv:1512.03385 [cs], December 2015.

[19] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, Memory-based control with recurrent neural networks, arXiv:1512.04455 [cs], December 2015.

[20] D. Hendrycks and K. Gimpel, Gaussian Error Linear Units (GELUs), arXiv:1606.08415 [cs], July 2020.

[21] G. Hinton, Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude, `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`, Accessed: 2021-09-05, September 2021.

[22] S. Hochreiter and J. Schmidhuber, Long Short-Term Memory, Neural Computation, 9(8), pp. 1735–1780, November 1997, ISSN 0899-7667.

[23] D. P. Kingma and J. Ba, Adam: A Method for Stochastic Optimization, arXiv:1412.6980 [cs], January 2017.

[24] K. Kopşa and A. T. Kutay, *Reinforcement learning control for autorotation of a simple point-mass helicopter model*, METU, Ankara, 2018.

[25] A. Kumar, N. Paul, and S. N. Omkar, Bipedal Walking Robot using Deep Deterministic Policy Gradient, arXiv:1807.05924 [cs], July 2018.

[26] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, Continuous control with deep reinforcement learning, arXiv:1509.02971 [cs, stat], July 2019.

[27] W. S. McCulloch and W. Pitts, A logical calculus of the ideas immanent in nervous activity, The bulletin of mathematical biophysics, 5(4), pp. 115–133, December 1943, ISSN 1522-9602.

[28] T. M. Mitchell, *Machine Learning*, McGraw-Hill Education, New York, 1st edition edition, March 1997, ISBN 9780070428072.

[29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, Playing Atari with Deep Reinforcement Learning, arXiv:1312.5602 [cs], December 2013.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, Human-level control through deep reinforcement learning, Nature, 518(7540), pp. 529–533, February 2015, ISSN 1476-4687.

[31] C. Olah, Understanding LSTM Networks – colah's blog, `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, Accessed: 2021-09-05, August 2015.

[32] X. Pan, Y. You, Z. Wang, and C. Lu, Virtual to Real Reinforcement Learning for Autonomous Driving, arXiv:1704.03952 [cs], September 2017.

[33] E. Parisotto, H. F. Song, J. W. Rae, R. Pascanu, C. Gulcehre, S. M. Jayakumar, M. Jaderberg, R. L. Kaufman, A. Clark, S. Noury, M. M. Botvinick, N. Heess, and R. Hadsell, Stabilizing Transformers for Reinforcement Learning, arXiv:1910.06764 [cs, stat], October 2019.

[34] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, PyTorch: An Imperative Style, High-Performance Deep Learning Library, Advances in Neural Information Processing Systems, 32, pp. 8026–8037, 2019.

[35] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz, Parameter Space Noise for Exploration, arXiv:1706.01905 [cs, stat], January 2018.

[36] D. Rastogi, Deep Reinforcement Learning for Bipedal Robots, 2017.

[37] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain., Psychological Review, 65, pp. 386–408, 1958.

[38] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning representations by back-propagating errors, Nature, 323(6088), pp. 533–536, October 1986, ISSN 1476-4687.

[39] S. J. Russell and P. Norvig, *Artificial Intelligence A Modern Approach*, Prentice Hall, 3 edition, 1995, ISBN 9780136042594.

[40] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, Deep Reinforcement Learning framework for Autonomous Driving, Electronic Imaging, 2017(19), pp. 70–76, January 2017.

[41] S. R. B. d. Santos, S. N. Givigi, and C. L. N. Júnior, An experimental validation of reinforcement learning applied to the position control of UAVs, in *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 2796–2802, October 2012, iSSN: 1062-922X.

[42] S. Shalev-Shwartz, S. Shammah, and A. Shashua, Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving, arXiv:1610.03295 [cs, stat], October 2016.

[43] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, Deterministic Policy Gradient Algorithms, in *International Conference on Machine Learning*, pp. 387–395, PMLR, January 2014.

[44] D. R. Song, C. Yang, C. McGreavy, and Z. Li, Recurrent Deterministic Policy Gradient Method for Bipedal Locomotion on Rough Terrain Challenge, in *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pp. 311–318, November 2018.

[45] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, A Bradford Book, February 1998.

[46] G. E. Uhlenbeck and L. S. Ornstein, On the Theory of the Brownian Motion, Physical Review, 36(5), pp. 823–841, September 1930.

[47] U. Upadhyay, N. Shah, S. Ravikanti, and M. Medhe, Transformer Based Reinforcement Learning For Games, arXiv:1912.03918 [cs], December 2019.

[48] H. van Hasselt, A. Guez, and D. Silver, Deep Reinforcement Learning with Double Q-learning, arXiv:1509.06461 [cs], December 2015.

[49] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, Attention is All You Need, 2017.

[50] S. Wang, D. Jia, and X. Weng, Deep Reinforcement Learning for Autonomous Driving, arXiv:1811.11329 [cs], May 2019.

[51] C. J. Watkins and P. Dayan, Technical Note: Q-Learning, Machine Learning, 8(3), pp. 279–292, May 1992, ISSN 1573-0565.

[52] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T.-Y. Liu, On Layer Normalization in the Transformer Architecture, arXiv:2002.04745 [cs, stat], June 2020.