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

JUNE 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:

Prof. Dr. I am the Chair
Computer Engineering Department, METU _____

Prof. Dr. Ömür Uğur
Scientific Computing, METU _____

Assoc. Prof. Dr. I may be Co-Supervisor
Computer Engineering Department, METU _____

Assoc. Prof. Dr. A Member with High Title
Computer Engineering Department, METU _____

Assist. Prof. Dr. A Member with Low Title
Computer Engineering Department, Hacettepe University _____

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

June 2021, 51 pages

Deep Reinforcement Learning methods on mechanical control are successful on many environments and used instead of traditional optimal and adaptive control methods on some complex cases. However, Deep Reinforcement Learning algorithms do still have challenges. One is control on partially observable environments. When an agent is not informed well about the environment, it must recover information from past observations. In this thesis, walking of Bipedal Walker (OpenAI GYM) environment is studied by continious actor-critic reinforcement learning algorithm Twin Delayed Deep Determinstic Policy Gradient. Environment is partially observable because walker is not able to see behind. Several neural architectures are implemented. First one is feed-forward neural network with residual connection under the observable environment assumption, while second and third ones are Long Short Term Memory and Transformer using observation history as input to recover hidden state since environment is assumed to be partially observable.

Keywords: deep reinforcement learning, partial observability, robot control, actor-critic methods, etc.

ÖZ

PEKİŞTİRMELİ ÖĞRENME YÖNTEMLERİYLE KISMİ GÖZLENEBİLİR ORTAMDA ÇİFT BACAKLI ROBOTUN YÜRÜTÜLMESİ

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

Haziran 2021, 51 sayfa

Mekanik kontrol üzerine Derin Pekiştirmeli Öğrenme yöntemleri birçok ortamda başarılıdır ve bazı karmaşık durumlarda geleneksel optimal ve uyarlanabilir kontrol yöntemleri yerine kullanılır. Ancak, Derin Pekiştirmeli Öğrenme algoritmalarının hala zorlukları vardır. Bunlardan biri kısmen gözlemlenebilir ortamlarda kontroldür. Bir özne ortam hakkında yeterince bilgilendirilmediğinde, geçmiş gözlemleri anlık gözlemlere ek olarak kullanmalıdır. Bu tezde, Bipedal Walker (Çift bacaklı yürüyen robot, OpenAI GYM) ortamının yürüyüşü, sürekli aktör-eleştirmen pekiştirmeli öğrenme algoritması İkiz Gecikmeli Derin Belirleyici Poliçe Gradyanı ile incelenmiştir. Robot arkasını göremediği için çevre kısmen gözlemlenebilirdir. Çalışmada birkaç sinir mimarisi uygulanmıştır. Birincisi, gözlemlenebilir ortam varsayımı altında artık bağlantıya sahip ileri beslemeli sinir ağı iken, ikinci ve üçüncü olanlar, ortamın kısmen gözlemlenebilir olduğu varsayıldığından girdi olarak gözlem geçmişini kullanan Uzun Kısa Süreli Bellek (LSTM) ve Dönüştürücüdür (Transformer).

Anahtar Kelimeler: pekiştirmeli derin öğrenme, kısmi gözlemlenebilirlik, robot kontrolü, aktör-eleştirmen metodları, vs.

For anyone who made me able to write this thesis.

ACKNOWLEDGMENTS

I would like to express my very great appreciation to my thesis supervisor Assoc. Prof. Dr. Ömür Uğur for his patient guidance, enthusiastic encouragement and valuable advices during the development and preparation of this thesis. His willingness to give his time and to share his experiences has brightened my path. !!!!!

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 Learnin
DRL	Deep Reinforcement Learning
FFNN	Feed Forward Neural Network
RFFNN	Residual Feed Forward Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory

CHAPTER 1

INTRODUCTION

Artificial intelligence (AI) is the ability of a computer program or a machine to think and learn like natural intelligence performed by humans and animals. One way is to create an intellgent agent is using some methods to detect patterns on data and use it to make predictions on unseen data. This approach is called Machine Learning.

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 some objective. A major challenge to machine learning is creating agents that will act more natural and humanlike. As a subfield of Machine Learning, Reinforcement Learning allows an agent to learn how to control (act) itself in different situations. It models environment to give reward or punishment to agent according to environmental state and agent actions, and 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 observation. 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 which has huge state-space (like visual observation, many kinematic sensors etc.) and missing observation. Along with recent developments in DNNs, Deep Reinforcement Learning (DRL) algorithms allows an agent to interact with environment in more complex way. DRL is based on neural networks which are function approximators. The problem with DRL is selection of a

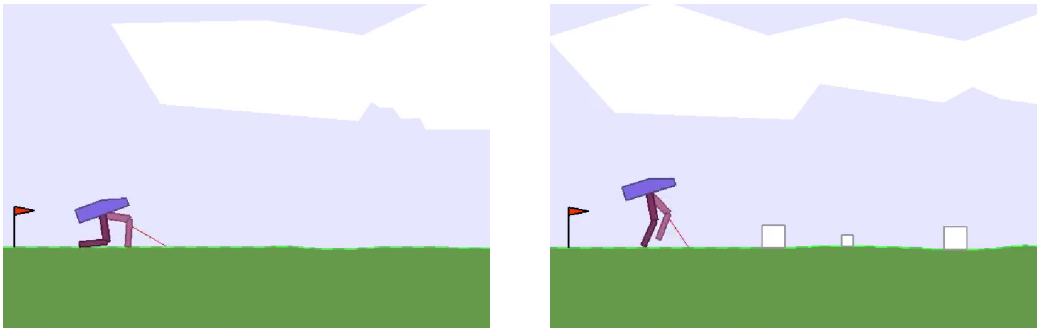
correct neural network, but there is still no analytical method to design a neural network for a particular task. Therefore, neural design is commonly based on trial-error. Since its discovery, robots have been crucial devices for the human race, whether smart or not. Intelligent humanoid and animaloid robots have been in continuous development since early 1980s. This type of robots has legs unlike driving robots. Since most of world terrain is unpaved, this type of robots are good alternative to driving robots. Locomotion is 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 so, design of walking controller using traditional methods is difficult due to same reasons. Therefore, for bipedal walking, Deep Reinforcement Learning (DRL) approach is an easier choice if simulation environment is available.
In this thesis, Bipedal Locomotion Deep Reinforcement Learning (DRL) by is investigated through *Bipedal-Walker-v3* [1] and *Bipedal-Walker-Hardcore-v3* [2] environment of open source GYM library [6]. Different neural architectures are used and results are compared.

1.1 Problem Statement: Bipedal Walker Robot Control

1.1.1 OpenAI Gym and Bipedal-Walker Environment

OpenAI Gym [6] is opensource framework, containing many environments to service reinforcement learning algorithms.
Bipedal-Walker environments [1] [2] is part of Gym environment library. One of them is classical version where the terrain is relatively smooth, while other one is hardcore version which contains ladders, stumps and pitfalls in terrain.
For both settings, the task is to move forward the robot as much as possible. It has continious action and observation space.
Locomotion of the Bipedal Walker is difficult control problem due to following reasons.

- **Nonlinearity**: The dynamics is nonlinear, unstable and multimodal. Dynamical



(a) Bipedal-Walker-v3 Snapshot (b) Bipedal-Walker-Hardcore-v3 Snapshot

Figure 1.1: Bipedal Walkers Snapshots

behavior of robot changes for different situations like ground contact, single leg contact and double leg concact.

- **Uncertainity**: The terrain where robot walks may vary. Designing a controller for all types of terrain is difficult.

- **Partially Observability**: The robot observes ahead of it with lidar measurements and cannot observe behind.

These difficulties make hard to implement analytical methods for control task. RL approach is better to tackle first 2 one. For partial observability problem, more elegant solution is required. This is done by creating a belief state from past observations to inform agent. Agent uses its belief state and observations to choose how to act. However, relying on observations is also possible.

1.1.2 Deep Learning Library: PyTorch

PyTorch is an open source library developed by Facebook's AI Research lab (FAIR) [29]. It is based on Torch library [7] and has Python and C++ interface. It is an automatic differentation library with accelerated math operations backed by graphical processing units (GPUs). This is what a deep learning library requires. And the clean pythonic syntax made it most famous deep learning tool among researches.

1.2 Proposed Methods and Contribution

Partially observable environments are always a hardwork for reinforcement learning algorithms. In this work, the walker environment assumed as fully observable environment at first. A feed-forward neural network architecture is proposed to control the robot. Then, the environment is assumed to be partially observable. In order to recover latent states, Long Short Term Memory (LSTM) and Transformer neural networks are proposed.

LSTMs are used in many deep learning applications which includes sequential data. It is a variant of Recurrent Neural Networks (RNN). It is a good candidate for RL algorithms which solves partially observable environments.

Transformers are developed to handle sequential data as RNN models does. However, it processes the whole sequence at same time, while RNNs process the sequence in order. It replaced RNNs in Natural Language Processing (NLP) tasks over RNNs due to its major improvements. However, this is not the case for Reinforcement Learning, yet. As RL algorithm, Twin Delayed Deep Deterministic Policy Gradient (TD3) is used. It is improved version of Deep Deterministic Policy Gradient (DDPG) Algorithm.

1.3 Related Work

Reinforcement Learning methods are used in many mechanical control tasks such as autonomus driving [27] [37] [35] [45] and autonomus flight [20] [3] [36].

Rastogi [31] 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 walker robot since it requires long time for convergence. Kumar et al. [21] also used DDPG to perform robot walking in 2D simulation environment. Their agent converged in approximately 25k episodes. Song et al. [39] pointed out the partial observability problem of bipedal walker, using Recurrent Deep Deterministic Policy Gradient (RDPG) [15] algorithm and acquired better results than original Deep Deterministic Policy Gradient (DDPG) algorithm.

Fris [10] used Twin Delayed DDPG (TD3) using Long Short Term Memory (LSTM)

for their quadrocopter landing task. Fu et al. [11] used vanilla RNN with attention mechanism using TD3 for car driving task, but not explicit Transformer. They reported that their method outperformed 7 baselines. Upadhyay et al. [42] used all of feed forward network, LSTM an original Transformer architectures for balancing pole on a cart from Cartpole environment of Gym, and Transformer yield worst results among three architectures.

1.4 Outline of the Thesis

This thesis is consisted of 5 main chapters. In Chapter 2; Reinforcement Learning Theory and used methods are presented. In Chapter 3, Neural Networks and Deep Learning explained. In Chapter 4, Bipedal Walker problem is discussed, methods are presented and results are summarized. In last Chapter, thesis is concluded by discussing obtained results and possible future works.

CHAPTER 2

REINFORCEMENT LEARNING

Machine Learning is ability of computer program which allows adaptation to new situations through experience, without explicitly programmed [24]. There exist three main paradigm.

Supervised Learning: It is 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 $i \in 1, 2, ..., N$ by minimizing error between predicted and target output.

Input x can be thought as state of an agent. That makes y correct action at state x. For supervised learning, both x and y should be available, where the correct action are provided by a friendly supervisor [34].

Unsupervised Learning: Unsupervised learning is discovering structure on input examples without any label on it. Based on N example input pairs (x_i), it is discovering function $f: X \rightarrow Y$, $f(x_i) = y_i$, for all $i \in 1, 2, ..., N$, where y_i is discovered output. This discovery is motivated by predefined objective but this is not target error as in Supervised Learning.

Again, input x can be thought as 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 [34].

Reinforcement Learning: This is one of three main machine learning paradigm along with Supervised and Unsupervised Learning. It is 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 act in an environment [40]. Each action of agent is either rewarded or punished according to reward criteria. Therefore, reward function is mathematical representation

of what to teach agent. The agent explores environment by taking various actions in different states to get experience, based on trial-and-error. Then it exploits experiences to get highest reward from environment considering instant and future rewards over time.

Formally, Reinforcement Learning is learning a policy function $\pi\colon \mathcal{S} \rightarrow \mathcal{A}$ which maps inputs (states) $s \in \mathcal{S}$ to outputs (actions) $a \in \mathcal{A}$. Learning is done by maximizing value function $Q^{\pi}(s, a)$ (cumulative reward) for all possible states, which depends on policy π. Being so, this is similar to unsupervised learning. However, the difference is that value function $Q^{\pi}(s, a)$ is not defined exactly. It is also learned by interacting with environment by taking actions.

Reinforcement Learning is different than Supervised Learning because correct actions are not provided. Meanwhile, it is also different than unsupervised learning because the agent is forced to learn a behaviour. The agent is evaluated at each time step without supervision.

2.1 Reinforcement Learning and Optimal Control

Optimal control is a field of mathematical optimization, concerned by finding control policy of a dynamical system (environment) for given objective. For example, objective might be total reveune for a company as system, minimal fuel burn for a car as system, or total production for a factory.

Reinforcement Learning is kind of naive subfield of Optimal Control. However, RL algorithms find policy (controller) by error minimization of objective from experience, while Optimal Control methods are concerned exact analytical optimal solutions based on dynamic model of environment and agent.

Optimal Control methods are efficient and robust when mathematical model of environment is available, accurate enough and solvable for optimal controller. However, some real world problems usually do not exhibit all of these conditions. In such cases, reinforcement Learning is an easier way 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 [8] [40].

Exploration Explotation Dilemma: A RL agent is supposed to maximize rewards (knowledge exploration) by observing the environment (exploration of environment). This gives rise to the exploration-exploitation dilemma that is the inevitable trade-off between them. Exploration is taking a range of acts to benefit about the consequences. Typically results in low immediate rewards and high rewards for the future. Explotation is taking action that has been learned. Typically results in high immediate rewards and low rewards in the future.

Generalization and Curse of Dimensionality: A RL agent should also be able to generalize experiences to act on unseen situation before. This issue arises when state space and action space is high dimensional since experiencing all possibilities is impractical. This is solved by introducing function approximator. Deep Reinforcement Learning uses neural network as function approximator.

Delayed Consequences: A RL agent should be aware reason of reward or punishment. Once it gets reward or punishment, it should be able to discriminate whether reward is caused by instant actions or past actions.

Partial Observability: Partial observability is absence of all required observation to infer instant state. For instance, a driver does not 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 absence of rear view mirror or side mirror. In real world, most of systems are partially observable. This problem is usually tackled by incorprating observation history from agents memory in acting.

Safety of Agent: Mechanical agents can kill or degrade themselves and their surroundings. This safety problem is important on both exploration stage and full operation. Simulation of environment is a good way to train agent with safety but causes incomplete learning due to inaccuracy compared to real environment.

2.3 Sequential Decision Making

RL is stochastic control process in discrete time [40]. At time t, the agent starts with state s_t and observes o_t, then takes action a_t according to its policy π and obtains reward r_t at time t. Then state transition to s_{t+1} as a consequence of action and agent observes next observation o_{t+1}. History is set of past actions observations and rewards, $h_t = \{a_0, o_0, r_0, ...a_t, o_t, r_t\}$. State s_t is function of the history, $s_t = f(h_t)$, which represents situtaion of environment as much as possible.

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 past, so it is memoryless. In MDP setting, the system is fully observable which means states can be derived from instant observations; i.e., $s_t = f(o_t)$. Therefore, agent can decide action based on only instant observation o_t instead of what happened on previous time [9].

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

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

Model T: Function of how environment evolves through time, representing transition probabilities $T(s', s, a) = p(s'|s, a)$ where $s' \in \mathcal{S}$ is next state, $s \in \mathcal{S}$ is instant state and $a \in \mathcal{A}$ is taken action.

Reward Function R: Function of rewards coming from environment. At each state transition $s_t \rightarrow s_{t+1}$, a reward r_t is given to agent. Rewards can be either deterministic or stochastic. Reward function R is expected value of reward at given state s and taken action a. Therefore, it is defined as function of state and action, R: $\mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$.

$$R(s, a) = \mathbb{E}[r_t|s_t = s, a_t = a] \, \forall t = 0, 1, ... \tag{2.1}$$

Discount Factor γ: Discount factor $\gamma \in [0, 1)$ is measure of importance of rewards in the future for value function.

2.5 Partially Observed Markov Decision Process

Sometimes, observation space is not enough to represent all information (state) about environment. 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 [9].
Observation Space $\mathcal{O}$: A set of all possible observations of agent.
Observation Model O: Function of how observations are related to states, representing observation probabilities $O(o, s) = p(o|s)$ where $s \in \mathcal{S}$ is instant state and $o \in \mathcal{O}$ is observation.

2.6 Policy

Policy is logic of how agent acts according to state of environment. It can be either deterministic or stochastic.

- Deterministic policy μ is a mapping from states to actions; $\mu \colon \mathcal{S} \mapsto \mathcal{A}$.
- Stochastic policy π is a mapping from state-action pair to a probability; $\pi \colon \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$.

2.7 Return, Value Functions and Policy Learning

Return and Discount Factor: At time t, G_t is return which is cumulative sum of future rewards, scaled by discount factor γ.

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

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

State Value Function: State Value Function V^{π} is expected return when policy π is followed in future.

$$V^{\pi}(s) = \mathbb{E}[G_t | s_t = s, \pi] \quad \forall t = 0, 1, ... \tag{2.3}$$

Optimal value function should return maximum expected return. The behavior is controlled by policy.

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

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

$$Q^{\pi}(s, a) = \mathbb{E}[G_t | s_t = s, a_t = a, \pi] \quad \forall t = 0, 1, ... \tag{2.5}$$

Optimal state-action value function should yield maximum expected return for each state-action pair. It is defined as follows.

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

Similarly, the optimal policy π^{*} can be obtained by $Q^{*}(s, a)$. For stochastic policy, it is defined as follows.

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

For deterministic policy, it is defined as follows.

$$\mu^{*}(s) = \text{argmax}_a \, Q^{*}(s, a) \tag{2.8}$$

2.8 Bellman Equation

Bellman proved that optimal value function should satisfy following conditions [5].

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

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

2.9 Model Free Reinforcement Learning

Model based methods are based on solving bellman equation 2.92.10 with given model T. On the other hand, Model Free Reinforcement Learning is suitable if environment model is not available but agent can experience environment by consequences of its actions. There are 3 main types.

Value Based Learning: Value functions are learned, then policy arises naturally from value function as shown in 2.7 2.8. Since argmax operation is used, this type of learning is suitable for problems where action space is discrete.

Policy Based Learning: Policy is learned directly, return values are used instead of learning a value function. Unlike value based methods, it is suitable for continious action spaces.

Actor Critic Learning: Both policy (actor) and value (critic) functions are learned simulatenously. It is also suitable for continious action spaces.

2.9.1 Q Learning

Q Learning is a value based type of learning. It is based on optimizing Q function using bellman equation 2.10 [46].

Assume that Q function is parametrized by θ. Target Q value is estimated by bootstrapping estimate itself, as shown in Eq. 2.11.

$$Y_t^Q = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) \tag{2.11}$$

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 target value and estimated value with respect to θ using numerical optimization methods.

$$\mathcal{L}_t(\theta) = \big(Y_t - Q(s_t, a_t; \theta)\big)^2 \tag{2.12}$$

2.9.1.1 Deep Q Learning

Deep Q Learning overcomes instability of Q Learning. When a nonlinear approximator is used, learning is unstable. Deep Q Learning introduces Target Network and Experience Replay [26, 25].

Target Network: Target network is parametrized θ^-. It is used to evaluate target value and not updated by loss minimization. It is updated at each fixed number of update step C by Q network parameter θ. Target value is obtained by using θ^-.

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

Experience Replay: Experience tuples are stored in dataset $\mathcal{D}$ as queue with fixed buffer size N_{replay}. At each iteration i, θ is updated by experiences uniformly subsampled from experience replay. It allows agent to learn from experiences multiple times. More importantly, sampled experiences are close to be independent and identically distributed if buffer size is high enough. This makes learning process more stable.

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim U(\mathcal{D})}\Big[\big(Y^{DQN} - Q(s, a; \theta_i)\big)^2\Big] \tag{2.14}$$

Epsilon Greedy Exploration: As stated in Chapter 2.2, exploration is an important step for reinforcement learning algorithms. In discrete action space (finite action space $\mathcal{A}$), simplest exploration strategy is epsilon-greedy approach. During learning, a random action is selected by probability ϵ or greedy action (maximizing Q value) by $1 - \epsilon$ as shown in Eq. 2.15.

$$\pi(a|s) = \begin{cases} 1-\epsilon, & \text{if } a = \text{argmax}_a\, Q(s,a) \\ \frac{\epsilon}{|\mathcal{A}|-1}, & \text{otherwise} \end{cases} \tag{2.15}$$

Algorithm is summarized in Algorithm 1.

Algorithm 1: Deep Q Learning with Experience Replay

Initialize: Replay memory $\mathcal{D}$ with capacity N_{replay}

Action value function parameters θ

Target action value function parameters $\theta^- \leftarrow \theta$

Epsilon parameter for exploration ϵ, Update delay parameter d

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

  Recieve initial state s_1;

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

    Select random action a_t with probability ϵ, otherwise greedy action $a_t = \text{argmax}_a\, Q(s_t, a; \theta)$;

    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 $\mathcal{D}_r$;

    Set $Y_j^{DQN} =$

$$\begin{cases} r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^-) & \text{if } s_{j+1} \text{ not terminal} \\ r_j & \text{if } s_{j+1} \text{ terminal} \end{cases} \quad \forall e_j \in \mathcal{D}_r;$$

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

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

  end

end

2.9.1.2 Double Deep Q Learning

In DQN, max operator is used to select and evaluate action on the same network 2.13. This yields overestimated value estimations in noisy environments. Therefore, action selection and value estimation is decoupled in target evaluation to overcome Q function overestimation [43].

$$Y_t^{DDQN} = r_t + \gamma Q(s_{t+1}, \text{argmax}_{a'} Q(s_{t+1}, a'; \theta_i); \theta^-) \tag{2.16}$$

Learning process is same with DQN except target value.

2.9.2 Deterministic Actor Critic Learning

Value based methods are not suitable for continious action spaces. Therefore, policy is explicitly defined instead of maximizing Q function. Policy function can be either stochastic or deterministic. Deterministic actor-critic is kind of learning which uses deterministic policy [38]. It can be thought as continious version of Q Learning. Value function is called critic while policy function is called actor.
In discrete action space, policy is naturally obtained by argmax operation on Q function as in 2.8. In DPG, policy μ is parametrized by another set of parameters θ^μ while value function Q is parametrized by θ^Q.
The ultimate goal is to maximize value function V. It is done by selecting action which maximizes Q value as policy Eq. 2.8. Therefore, value function is the criteria to be maximized by solving parameters θ^μ given θ^Q.

$$\theta^\mu = \text{argmax}_{\theta^\mu} Q(s_t, \mu(s_t; \theta^\mu); \theta^Q) \tag{2.17}$$

In order to learn policy, Q function should also be learned simulatenously. For Q function approximation, target value is parametrized by θ^Q and θ^μ Eq. 2.18. And this target is used to learn Q function by minimizing least squares loss Eq. 2.19.

$$Y_t^{DPG} = r_t + \gamma Q(s_{t+1}, \mu(s_{t+1}; \theta^\mu); \theta^Q) \tag{2.18}$$

$$\mathcal{L}_t(\theta^Q) = \left(Y_t^{DPG} - Q(s_t, a_t; \theta^Q)\right)^2 \tag{2.19}$$

Note that both θ^Q and θ^μ should be learned at the same time. Therefore, parameters are updated simulatenously during learning iterations.

2.9.2.1 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient is continuous compliment of Deep Q learning using deterministic policy [22]. It uses experience replay and target networks.
Similar to Deep Q learning, there are target networks parametrized by θ^{μ^-} and θ^{Q^-} along with main networks parametrized by θ^{μ} and θ^{Q}. While target networks are updated in fixed number of steps in DQN, DDPG updates target network parameters at each step with polyak averaging as follows.

$$\theta^- \leftarrow \tau\theta + (1-\tau)\theta^- \tag{2.20}$$

The τ is an hyperparameter indicating how slow the target network is updated and usually close to zero.
Policy network parameters are learned by maximizing resulting expected value 2.21. Note that value network parameters are assumed to be learned.

$$\theta^{\mu} = \text{argmax}_{\theta^{\mu}}\, \mathbb{E}_{s\sim U(\mathcal{D})}\Big[Q(s, \mu(s_t;\theta^{\mu});\theta^{Q})\Big] \tag{2.21}$$

Target networks are used to predict targe value Eq. 2.22. This target is used to learn Q function by minimizing least squares loss Eq. 2.23 in each iteration.

$$Y_t^{DDPG} = r_t + \gamma Q(s_{t+1}, \mu(s_{t+1};\theta^{\mu^-});\theta^{Q^-}) \tag{2.22}$$

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{s,a,r,s'\sim U(\mathcal{D})}\Big[\big(Y^{DDPG} - Q(s,a;\theta_i^Q)\big)^2\Big] \tag{2.23}$$

In DDPG, value and policy network parameters are learned simultaneously. During learning, exploration noise is added to each selection. In original paper [22], authors proposed to use Ornstein Uhlenbeck Noise [41] which has temporal correlation for efficiency. However, simple gaussian noise or another noise is also possible. Algorithm is summarized in Algorithm 2.

Algorithm 2: Deep Deterministic Policy Gradient

Initialize: Replay memory $\mathcal{D}$ with capacity N_{replay}

Policy and action value function parameters θ^{μ}, θ^{Q}

Target policy and action value function parameters $\theta^{\mu^-} \leftarrow \theta^{\mu}$, $\theta^{Q^-} \leftarrow \theta^{Q}$

Random process $\mathcal{N}$ as exploration noise

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 $\epsilon \sim \mathcal{N}$;

    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 $\mathcal{D}_r$;

    Set $Y_j^{DDPG} =$

$$\begin{cases} r_j + \gamma Q(s_{j+1}, \mu(s_{j+1}; \theta^{\mu^-}); \theta^{Q^-}) & \text{if } s_{j+1} \text{ not terminal} \\ r_j & \text{if } s_{j+1} \text{ terminal} \end{cases} \quad \forall e_j \in \mathcal{D}_r;$$

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

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

    Update target networks

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

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

  end

end

Ornstein-Uhlenbeck Process: It is continious analogue of discrete AR(1) process [41]. The process x is defined by a stochastic differential equation Eq. 2.24

$$\frac{dx}{dt} = -\theta x + \sigma\eta(t) \tag{2.24}$$

where $\eta(t)$ is white noise. Its standard deviation in time is equal to $\frac{\sigma}{\sqrt{2\theta}}$. This process is commonly used as exploration noise in physical environments since it has momentum in time. A sample with different parameters is visualized in Figure 2.1.

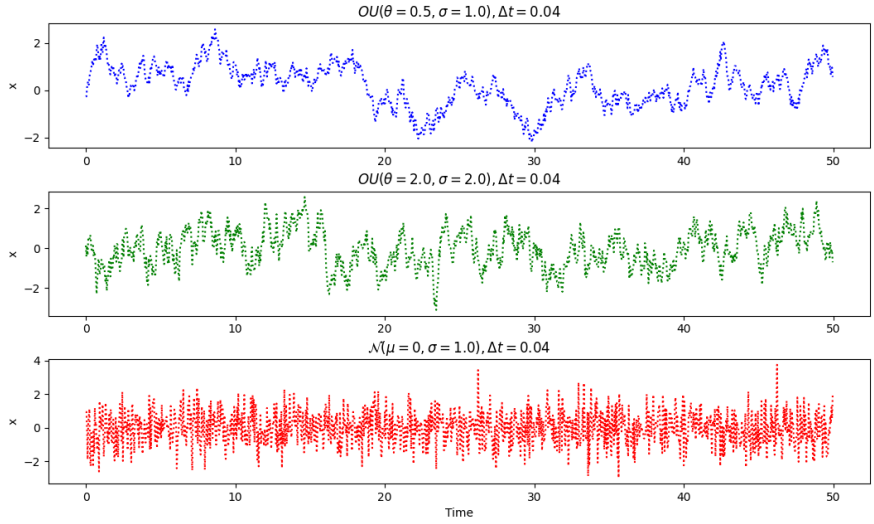


Figure 2.1: Ornstein-Uhlenbeck Process and Gaussian Process comparison

2.9.2.2 Twin Delayed Deep Deterministic Policy Gradient

Twin Delayed Deep Deterministic Policy Gradient [12] is improved version of DDPG with higher stability and efficiency. There are three main tricks.

Target Policy Smoothing: For target value assesing, actions are obtained from target policy network in DDPG, while a clipped zero centered gaussian noise is added to actions in TD3 Eq. 2.25. This regularizes learning process by smoothing effects of actions on value.

$$a'(s') = \mu(s'; \theta^{\mu^-}) + \text{clip}(\epsilon, -c, c), \quad \epsilon \sim \mathcal{N}(0, \sigma) \tag{2.25}$$

Clipped Double Q Learning: There are two different Q networks with their targets. During learning, both of them are learned from single target value. This value is assessed by using whichever of two networks give smaller of it. This allows to escape from overestimation of values.

$$Y_t^{TD3} = r_t + \gamma \min_{k \in \{1,2\}} Q(s_{t+1}, ; a'(s_{t+1}); \theta^{Q_k^-}) \tag{2.26}$$

Policy is learned by maximizing output of first value network Eq. 2.27.

$$\theta^\mu = \text{argmax}_{\theta^\mu} \mathbb{E}_{s \sim U(\mathcal{D})} \Big[Q(s, \mu(s_t; \theta^\mu); \theta^{Q_1}) \Big] \tag{2.27}$$

Delayed Policy Updates: During learning, policy network and target networks are updated less frequently (at each fixed number of step) than value network. Since policy network parameters are learned by maximizing value network, it should be learned slower. Algorithm is summarized in Algorithm 3.

Algorithm 3: Twin Delayed Deep Deterministic Policy Gradient

Initialize: Replay memory $\mathcal{D}$ with capacity N_{replay}

Policy and action value function parameters θ^{μ}, θ_1^Q, θ_2^Q

Target policy and action value function parameters $\theta^{\mu^-} \leftarrow \theta^{\mu}$, $\theta_1^{Q^-} \leftarrow \theta_1^Q$, $\theta_2^{Q^-} \leftarrow \theta_2^Q$

Random process $\mathcal{N}$ as exploration noise

Target policy smoothing parameters σ, c, Update delay parameter d

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 $\epsilon \sim \mathcal{N}$ 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 $\mathcal{D}_r$;

    Evaluate target actions for value target

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

    $\forall k \in \{1, 2\}$, set $Y_{jk}^{TD3} =$

    $$\begin{cases} r_j + \gamma Q(s_{j+1}, a'(s_{j+1}); \theta_k^{Q^-}) & \text{if } s_{j+1} \text{ not terminal} \\ r_j & \text{if } s_{j+1} \text{ terminal} \end{cases} \quad \forall e_j \in \mathcal{D}_r;$$

    Set $Y_j^{TD3} = \min(Y_{j1}^{TD3}, Y_{j2}^{TD3})$;

    Update θ_1^Q θ_2^Q by seperately minimizing $\frac{1}{N} \sum_{e_j \in \mathcal{D}_r} \left(Y_j^{TD3} - Q(s_j, a_j; \theta_k^Q)\right)^2 \quad \forall k \in \{1, 2\}$ with a single optimization step;

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

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

      Update target networks

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

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

  end

end

CHAPTER 3

NEURAL NETWORKS AND DEEP LEARNING

Through increasing computing power ability of computers, deep neural networks dominated machine learning because much more data can be handled in this way. As a subfield of machine learning, the term deep learning emerged which is 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 of tons of variants, a neural network is defined as parametrized function approximator which is inspried by biological neurons. The first models of neural network developed by a neurophysiologist Warren McCulloch and a mathematician Walter Pitts in 1943 [23]. However, the idea of neural network known today arised after development of a simple binary classifier called perceptron invented by Rosenblast et al. [32]. It 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 designed as complex as required. Parameters are updated by backpropagation algorithm to minimize loss between its outputs and desired outputs.

3.1 Backpropagation and Numerical Optimization

Neural networks are composed of weight parameters. Learning is process of updating weights to give desired behavior. This is represented in a loss function. Learning is

nothing but minimizing it 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 computed by chain rule. Therefore, gradient information propagates backward, and this process is called backpropagation.

3.1.1 Stochastic Gradient Descent Optimization

Gradient Descent minimizes loss function $\mathcal{L}$ by updating weight parameters θ to inverse of gradient direction with a 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 Eq. 3.2. Stochastic Gradient Descent approximate gradient of loss function by sample losses and updates parameters accordingly Eq. 3.3.

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

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

However, in practice, mini-batches are used to estimate loss gradient. In that case, batches with size N_b are sampled from instances (Eq. 3.4) and updates are performed accordingly (Eq. 3.5).

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

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

3.1.2 Adam Optimization

Adam [19] (short for Adaptive Moment Estimation) is a variant of stocastic gradient descent as improvement of RMSProp [17] 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 learning rate based on training data to overcome issues arised due to stochastic updates to accelerate training and make it robust to learning rate. It is summarized in 4.

Algorithm 4: 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

3.2.1 Perceptron

Perceptron is a binary classifier model. In order to allocate input x into a class, feature vector $\phi(x) \in \mathbb{R}^{1 \times d_k}$ is generated by a fixed nonlinear function. Then, a linear model is generated with linear transformation weights $W \in \mathbb{R}^{d_k \times 1}$ in the following form 3.6.

$$y = f(\phi(x)W) \tag{3.6}$$

where f is called activation function. For perceptron, it is defined as step function 3.7 while other functions like sigmoid, tanh can also be defined.

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

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

3.2.2 Activation Functions

As in Eq. 3.7, step function is used in perceptron. However, any other nonlinear function can be used instead. This nonlinearity allows a model to capture nonlinearity in data. There are tons of activation function in use today. Commonly used activations are *sigmoid*, *hyperbolic tangent (Tanh)*, *rectified linear unit (ReLU)*, *gaussian error linear unit (GELU)*.

Sigmoid Function: Sigmoid (σ) 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.

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

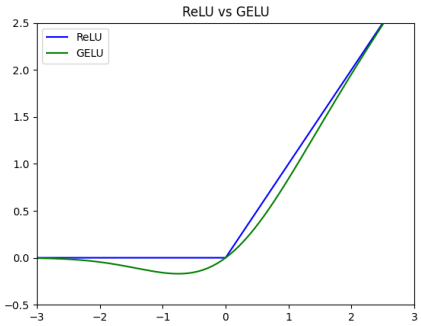
Hyperbolic Tangent: Tanh is used when an output is required to be in $[-1, 1]$. It has similar behavior with sigmoid function except it is zero centered. Their difference is visualized in Figure 3.1b.

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

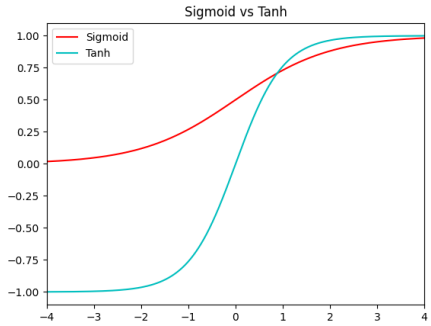
ReLU: ReLU is a simple function mapping negative values to zero while passing positive values as it is. It is computationally cheap and allows to train deep and complex networks [13].

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

GELU: GELU is smoothed version of ReLU function. It is continious but non-convex and has several advantages [16]. ReLU and GELU are visualized in Figure 3.1a.



(a) ReLU and GELU functions



(b) Sigmoid and Tanh functions

Figure 3.1: Activation Functions

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

3.2.3 Layer Normalization

Layer normalization is a layer to overcome unstablity and divergence during learning [4]. Given an input $x \in \mathbb{R}^K$, mean and variance statistics are evaluated along the dimensions Eq. 3.12.

$$\begin{aligned} \mu &= \frac{1}{K}\sum_{n=1}^{K} x_k \\ \sigma^2 &= \frac{1}{K}\sum_{n=1}^{K}(x_k - \mu)^2 \end{aligned} \tag{3.12}$$

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 by $\beta \in \mathbb{R}^K$ where these are learnable parameters.

$$\text{LN}(x) = \frac{x - \mu}{\sigma + \epsilon} * \gamma + \beta \tag{3.13}$$

3.3 Neural Network Types

3.3.1 Feed Forward Neural Networks (Multilayer Perceptron)

Structure of perceptron make a way for feed forward neural layers. Unlike stated above, 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 necessariliy step function. It can be any nonlinear function like sigmoid, tanh, relu etc. Feed Forward Neural Networks are generalization of perceptron algorithm to approximate any function f^*. Neural layers are stacked to construct deep feed forward neural network. It defines a nonlinear mapping $y = f(x;\theta)$ between input x and output y, parametrized by parameters $\theta = \{W\}_n, n = 1,\ldots,N$..
Assuming input signal is x (output of previous layer), activation value of the layer (h) is evaluated as by linear transformation followed by nonlinear activation f Eq. 3.14 applied elementwise.

$$net = xW + b \quad \text{and} \quad h = f(net) \tag{3.14}$$

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 [14]. For a fixed number of stacked layers (usually 2), 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.

3.3.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) [33] are type of neural network to process sequential data. It is specialized for data having sequential topology. It is used com-

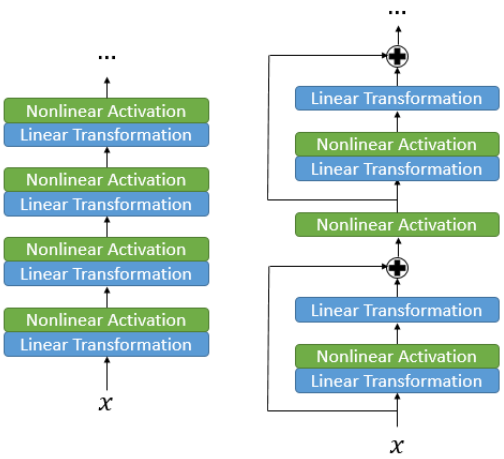


Figure 3.2: Deep Feed Forward (left) and Deep Residual Feed Forward Network (right)

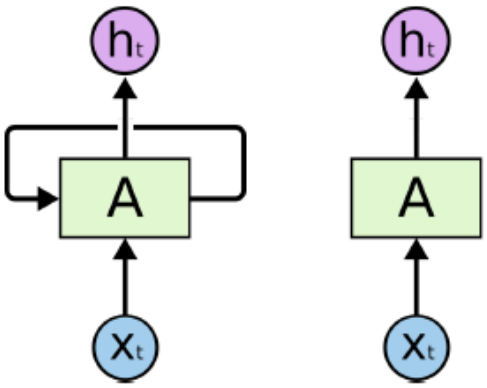


Figure 3.3: Recurrent Layer (left) and Feed Forward Layer (right) illustration.

monly used for sequence based applications.

Sequential data can be inferred by Recurrent Neural Networks. In Feed Forward Layers, output only depends on its input, while Recurrent Layer output is dependent 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 MLP layer 3.14, except input is concatenation of output feedback and input itself 3.15.

Given input sequence $x \in \mathbb{R}^{T \times d_x}$, output sequence $h \in \mathbb{R}^{T \times d_h}$ is evaluated recursively. Initial output h_0 can be either parametrized or assigned as zeros vector. Again, nonlinear activation f 3.14 applied element-wise.

$$net_t = h_{t-1}\tilde{W} + x_t W + b \quad \text{and} \quad h_t = f(net_t) \tag{3.15}$$

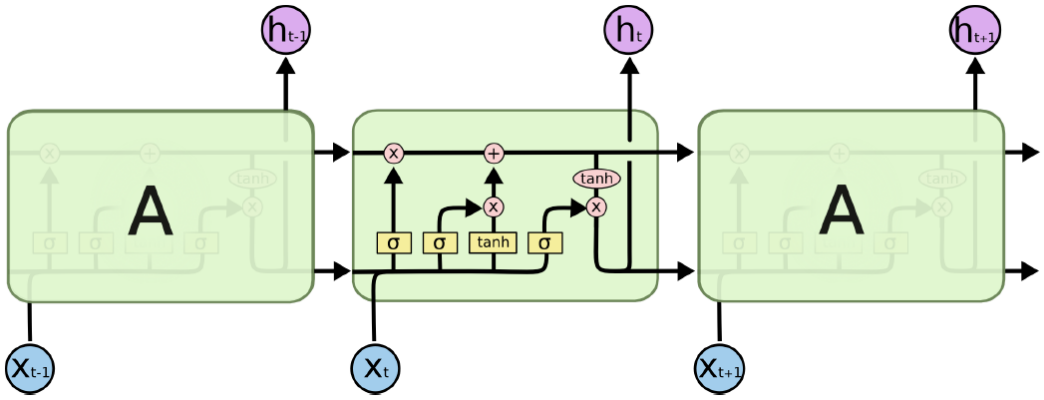


Figure 3.4: LSTM Cell.

3.3.3.1 Long Term Dependence Problem of Vanilla RNNs

Conventional RNNs have problem with vanishing/exploding gradient problem. As the sequence gets longer, effect of initial inputs in sequence decreases. This causes long term dependence problem. If information from initial inputs required, gradients either vanish or explode.
In order to overcome this problem another architecture is developed called Long Short Term Memory (LSTM) [18].

3.3.3.2 Long Short Term Memory

LSTM is a special type of RNN. It is explicitly designed to allow learning long-term dependencies. A single LSTM cell has 4 neural layer while vanilla RNN layer has only one neural layer. In addition to hidden state h_t , there is another state called cell state C_t. Information flow is controlled by 3 gates.
Forget Gate: Forget gate controls past memory. According to input, past memory is either kept or forgotten. Sigmoid function (σ) is used as activation function, applied elementwise.

$$f_t = \sigma([h_{t-1}; x_t]W_f + b_f) \tag{3.16}$$

Input Gate: Input gate controls contribution from input to cell state (memory). Hyperbolic tangent layer creates new candidate of cell state from input.

$$i_t = \sigma([h_{t-1}; x_t]W_i + b_i) \tag{3.17}$$

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

Cell State Update: Once what are to be forget and added are decided, cell state is updated.

$$C_t = f_t \odot C_{t-1} + i_t \odot \hat{C}_t \tag{3.19}$$

Output Gate: Sigmoid layer decides what part of new cell state to be output. Cell state is filtered by tanh to push values to be in $(-1, 1)$.

$$o_t = \sigma([h_{t-1}; x_t]W_o + b_o) \tag{3.20}$$

$$h_t = o_t \odot \tanh(C_t) \tag{3.21}$$

3.3.4 Attention Mechanism

As stated earlier, recurrent neural networks are prone to forget long term dependencies. LSTM and other variants are invented to overcome this problem. Although they reduced this problem, they cannot attend specific parts of the input. Therefore, people came with the idea of weighted avearing all states through time where weights depends on both input and output.

Assume that input sequence $X \in \mathbb{R}^{T \times d_X}$ is encoded to $H \in \mathbb{R}^{T \times d_H}$. The context vector is calculated using weight vector $\alpha \in \mathbb{R}^{1 \times T}$.

Calculation of weight vector depends on the task. For each time step, a score function is calculated between hidden state $H \in \mathbb{R}^{T \times d_H}$ and query q (which may be many things depending on task). Also, score function is also depends on choice. Then, attention score is $\alpha \in \mathbb{R}^T$ is calculated using arbitrary function f depending on choice.

$$\begin{aligned} \alpha &= f(q, H) \\ \text{Attention}(q, H) &= \sum_{\tau=1}^{T} \alpha_\tau h_\tau \end{aligned} \tag{3.22}$$

3.3.4.1 Transformer

The Transformer was proposed in the paper Attention is All You Need [44] . Unlike recurrent networks, this architecture is solely builded 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: An attention layer is a mapping from 3 vectors called query $Q \in \mathbb{R}^{T \times d_k}$, key $K \in \mathbb{R}^{T \times d_k}$ and value $V \in \mathbb{R}^{T \times d_v}$ to output, where T is time length, d_k and d_v are embedding dimensions. Output is weighted sum of values V 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. This approach is called Scaled Dot-product Attention.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{Q^T K}{\sqrt{d_k}})V \tag{3.23}$$

Multi-Head Attention: Instead of performing single attention; keys, queries and values are linearly projected from d_m dimensional vector space to h different spaces using projection matrices. Then, attention is done 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, $W^O \in \mathbb{R}^{hd_v \times d_m}$.

$$\begin{aligned} \text{MHA}(Q, K, V) &= \text{Concat}(a_1, a_2, ...a_h)W^O \\ a_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{3.24}$$

Feed Forward Layer: Both encoder and decoder contains feed forward layer, containing two linear transformations with ReLU activation.

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b)W_2 + b_2 \tag{3.25}$$

Encoder Layer: Encoder Layer starts with a residual self attention layer. Self attention means that query, key and value are same vectors. Then it is followed by feed forward neural layer, Both sublayers are employed with resudial connection with

layer normalization, i.e summation of layer input and output is passed through layer normalization.

$$\begin{aligned} att =& \mathrm{LN}(x + \mathrm{MHA}(x, x, x)) \\ y =& \mathrm{LN}(att + \mathrm{FFN}(att)) \end{aligned} \tag{3.26}$$

Decoder Layer: Similar to encoder layer, decoder layer has also self-attention and feed forward layers. In addition, there is another attention layer which is over encoder outputs. Same as encoder, all sublayers have resudial connection with layer normalization. Let's call encoded sequence $e \in \mathbb{R}^{T \times d_m}$ and decoded sequence $d \in \mathbb{R}^{T \times d_m}$ (masked). Assume that the sequence decoded up to tth point in sequence. Then, d_{t+1} is calculated as follows.

$$\begin{aligned} att =& \mathrm{LN}(d_{1:t} + \mathrm{MHA}(d_{1:t}, d_{1:t}, d_{1:t})) \\ dec =& \mathrm{LN}(att + \mathrm{MHA}(att, e, e)) \\ d_{t+1} =& \mathrm{LN}(dec + \mathrm{FFN}(dec)) \end{aligned} \tag{3.27}$$

Positional encoding: Since there are no recurrent or convolutional architecture in the model, sequential information needs to be embedded. Positional encodings has same dimension, so that input embeddings can be added to at the beginning of encoder or decoder stacks. For position pos, $2i$ or $2i + 1$th dimension has following values ($i \in \mathbb{N}$), as proposed in the original paper.

$$\begin{aligned} \mathrm{PE}_{pos,2i} &= \sin(pos/10000^{2i/d_m}) \\ \mathrm{PE}_{pos,2i+1} &= \cos(pos/10000^{2i/d_m}) \end{aligned} \tag{3.28}$$

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 gradients of output layers are high, so Pre-Layer Normalized transformer is proposed by [47]. Moreover, Parisotto et al. Xiong et al. [28]. proposed gated transformer which also includes layer normalizations before attention and feedforward layer. They also stated that although

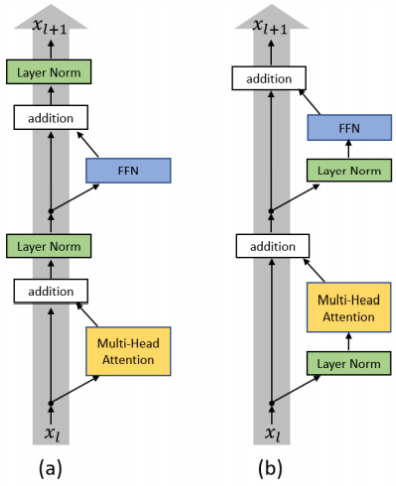


Figure 3.5: (a) Post-LN Transformer layer, (b) Pre-LN Transformer layer.

gated architecture improves many RL tasks drastically, non-gated pre-layer normalized transformer are good enough.

Encoder equations are as follows.

$$\begin{aligned} att &= x + \mathrm{MHA}(\mathrm{LN}(x), \mathrm{LN}(x), \mathrm{LN}(x)) \\ y &= att + \mathrm{FFN}(\mathrm{LN}(att)) \end{aligned} \tag{3.29}$$

CHAPTER 4

BIPEDAL WALKING BY TWIN DELAYED DEEP DETERMINISTIC POLICY GRADIENTS

4.1 Details of the Environment

Bipedal-Walker-v3 and *Bipedal-Walker-Hardocore-v3* are 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. It is modeled as Markov Decision Process with deterministic dynamics.
Observation Space: Hull angle, hull angular velocity, translational velocity on two dimension, joint positions, joint angular speeds, leg ground concats and 10 lidar rangefinder measurements. Details are summarized at Table 4.1.
Action Space: The robot has 2 legs with 2 joints at knee and hip. Torque provided to knee and pelvis joints of both legs. Details are presented in Table 4.2.
Rewarding: The robot should run fast with little energy while should not stumble and fall to ground. Therefore reward is shaped accordingly. Directly proportional to distance traveled forward, +300 points given if agent reaches end of path. -10 points (-100 points in original version) if agent falls, and small amount of negative reward proportional to applied motor torque (preventing applying unnecessary torque). Lastly, the robots gets negative reward propotional to absolut value of hull angle for reinforcing to keep hull straigth.

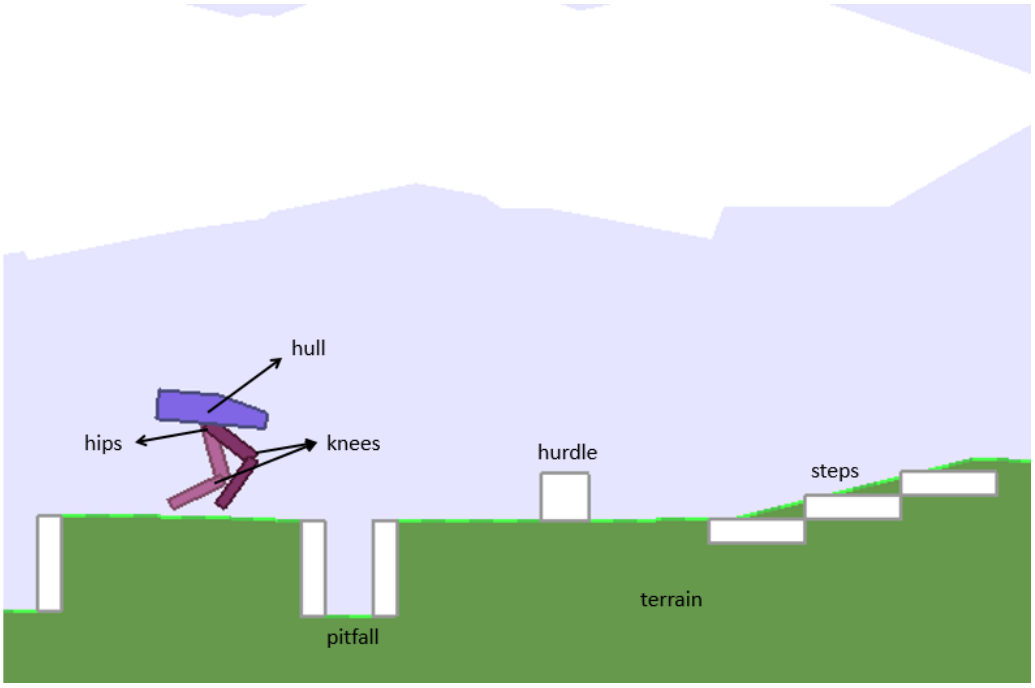


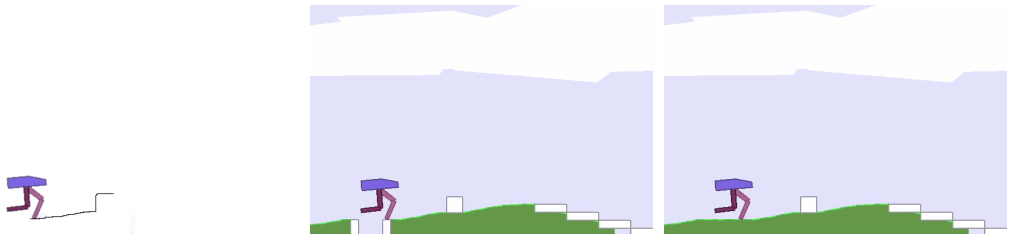
Figure 4.1: Bipedal Walker Hardcore Components

Num	Observation	Interval
0	Hull Angle	$[-\pi, \pi]$
1	Hull Angular Speed	$[-\infty, \infty]$
2	Velocity x	$[-1, 1]$
3	Velocity y	$[-1, 1]$
4	Hip 1 Joint Angle	$[-\infty, \infty]$
5	Hip 1 Joint Speed	$[-\infty, \infty]$
6	Knee 1 Joint Angle	$[-\infty, \infty]$
7	Knee 1 Joint Speed	$[-\infty, \infty]$
8	Leg 1 Ground Contact Flag	$\{0, 1\}$
9	Hip 2 Joint Angle	$[-\infty, \infty]$
10	Hip 2 Joint Speed	$[-\infty, \infty]$
11	Knee 2 Joint Angle	$[-\infty, \infty]$
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

4.1.1 Partial Observability

The environment is partially observable due to following reasons.

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

In DRL, partial observability is handled by 2 ways in literature [8]. First is incorprating fixed number of last observations while second way is updating hidden belief state using recurrent neural network at each time step. Our approach is using fixed number of past observation into LSTM and Transformer based networks for both actor and critic networks.

4.1.2 Reward Sparsity

Rewards given to the agent is sparse in some circumstances.

- 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.

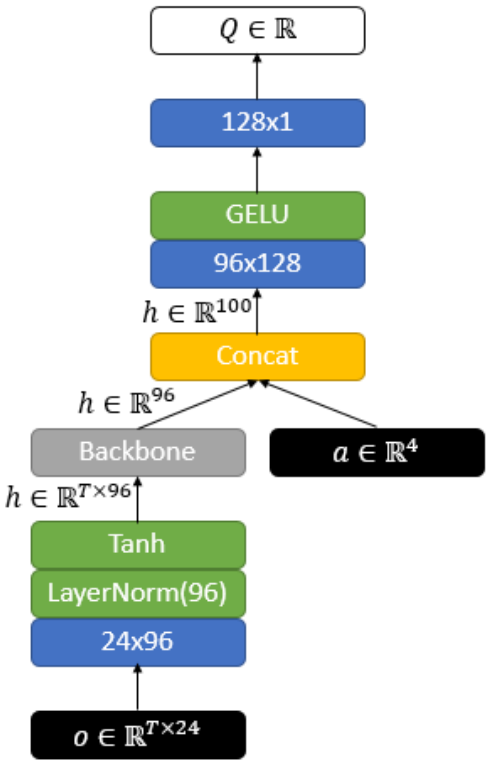
4.1.3 Modifications on Original Envrionment

It is difficult to solve directly available environment as it is. Therefore, there are few works on the literature demonstrating a solution.

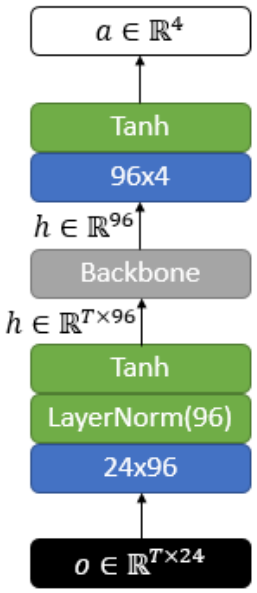
- In original version, agent gets -100 points when its hull hits the floor. In order to make the robot more greedy, this is changed to -10 points. Otherwise, agent cannot explore environment since it gets too much punisment when failed.
- Time frequency of simulation is halved (from 50 Hz to 25 Hz) by 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 time limit. Once this limit is reached, simulation stops with terminal flag. On the other hand, when agent fails before time limit, the episode ends with terminal flag too. In first case, the terminal flag causes instability since next step's value is not used in value update. The environment changed such that terminal flag is not given in this case unless agent fails.

4.2 Proposed Neural Networks

For all networks, varing backbones 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 GELU activation then a linear layer with single output. Before feeding observations to backbone, they are passed through a layer with layer normalization with tanh activation to 96 dimensional output. In actor network,



(a) Critic Architecture



(b) Actor Architecture

Figure 4.3: Neural Architecture Design

backbone is followed by a single layer with tanh activation for action estimation. As backbones, following networks are proposed. Again, observations are passed through a layer with layer normalization with tanh activation to 96 dimensional output before feeding to backbone. Critic and Actor networks are illustrated in Figure 4.3

4.2.1 Residual Feed Forward Network

Incoming vector is passed through 2 layers with 192 dimensional hidden size and 96 dimensional output, where there is GELU activation between 2 layers. This output is summed with initial vector and this is lastly passed through layer normalization.

4.2.2 Long Short Term Memory

Sequence of incoming vectors is passed though vanilla lstm layer with 96 dimensional hidden state. Output at last time step is outputted.

4.2.3 Transformer (Pre-layer Normalized)

Sequence of incoming vectors is passed though pre-layer normalized transformer with 192 dimensional feed forward layer with GELU activation. The output is lastly passed through layer normalization. During multi-head attention, only last state is fed as query so that attentions are calculated for only last state.

4.3 RL Method and hyperparameters

TD3 is used as RL algorithm. Hyperparameters are selected by grid search and best performing values are used. Adam optimizer is used as optimizer.
As exploration noise, Ornstein-Uhlenbeck noise is used and standart deviation is multiplied by 0.999 at the end of each episode. Initially $\theta = 4.0$ and $\sigma = 1.0$ are used.
Other hyperparameters are as follows.

- $\alpha = 7.0 \cdot 10^{-4}$ (For LSTM)
- $\alpha = 1.0 \cdot 10^{-3}$ (For Other networks)
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- $\gamma = 0.98$
- $N_{replay} = 500000$
- $N = 128$

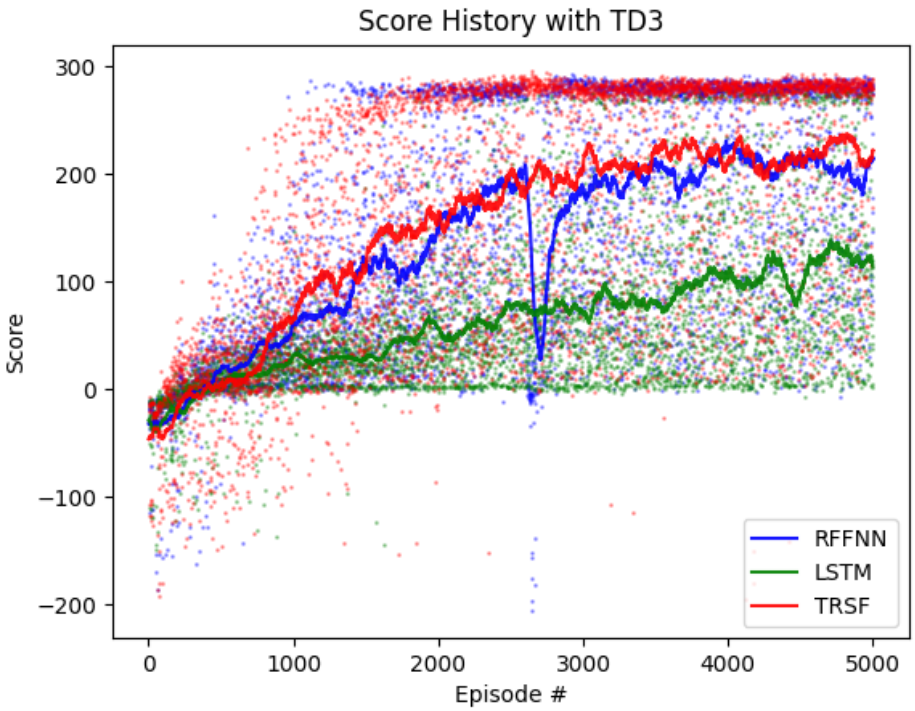


Figure 4.4: Scatter Plot with Moving Average for Episode Scores

4.4 Results

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

First of all, none of our approaches solved the problem since 300 points required in 100 random simulations as solution. However, our methods partially solved problems by exceeding 200 point limit, while some simulations yield around 280 points in all models.

RFFNN seems to be enough for solving the problem, although there exist partial observability in the environment. That model reaches around 220 points in average.

The robot was able to walk by LSTM model but yield worse results and cannot exceed 120 points in average.

Transformer model yield best results by reaching 230 points.

As Transformer and RFFNN are relatively succesfull, their behavior is visualized in Figure 4.6 and Figure 4.7. The main behavior difference is when the agent faces with

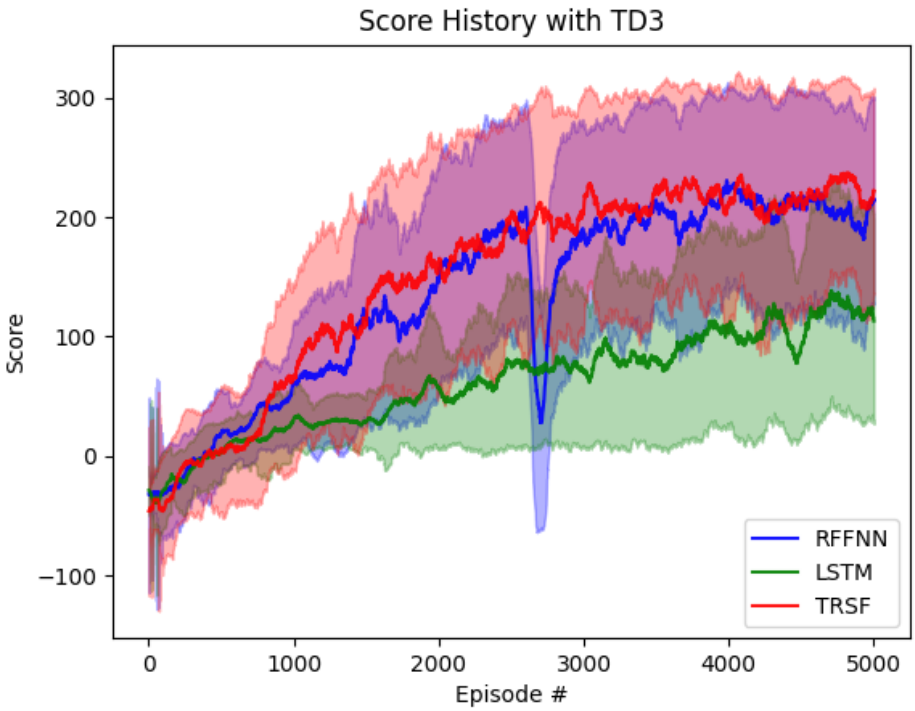
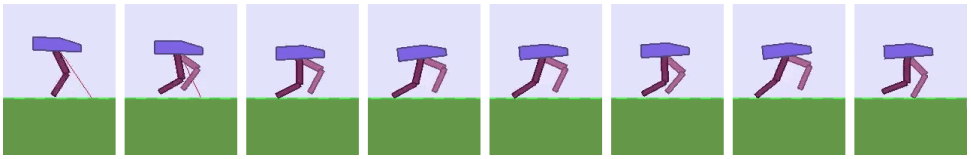


Figure 4.5: Moving Average and Standard Deviation for Episode Scores

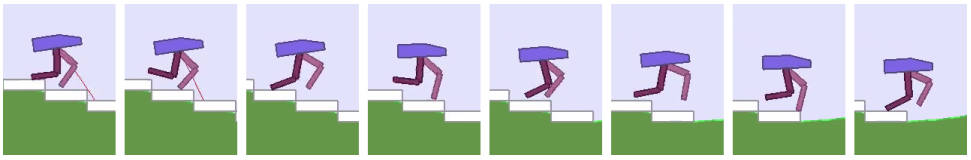
a big hurdle. First model passes it by jumping while other does by taking a very big step.

4.5 Discussion

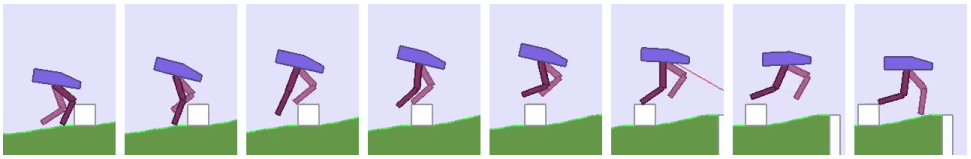
We believe that these results are not enough to say any model is superior to another, because there are other factors such as DRL method, number of episodes, network size etc. However, networks are designed to have similar sizes and good model requires to converge in less episodes. As a result, it is possible to say that transformers are able to surpass performance of LSTMs for partially observed RL problems. Note that this is valid where layer normalization is applied before multihead attention and feed-forward layers [47] as opposed to vanilla transformer proposed in [44].
Another result is that incorprating past observations did not improve performance drastically, although the opposite was expected. We can interpret that either the history length is less or the partial observability is not a big deal for the agent.



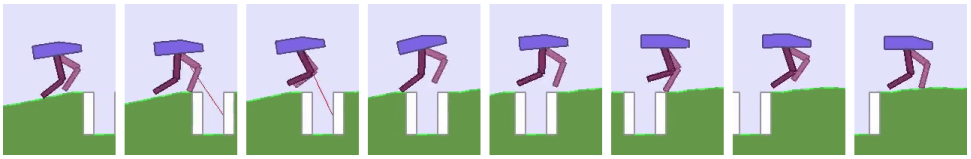
(a) Flat Surface



(b) Stairs

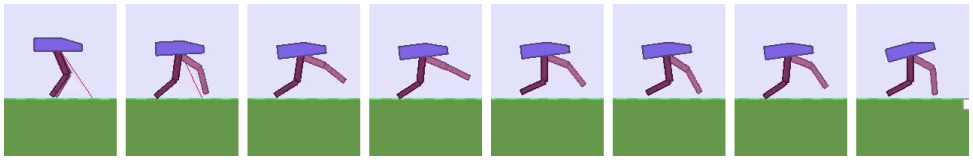


(c) Hurdle

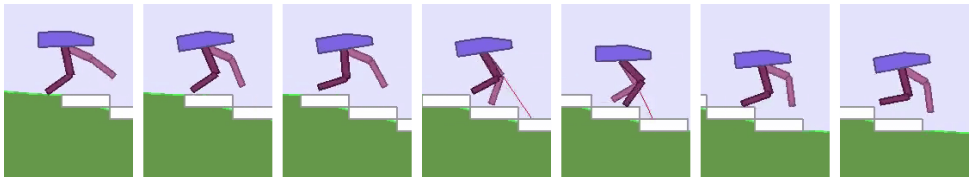


(d) Pitfall

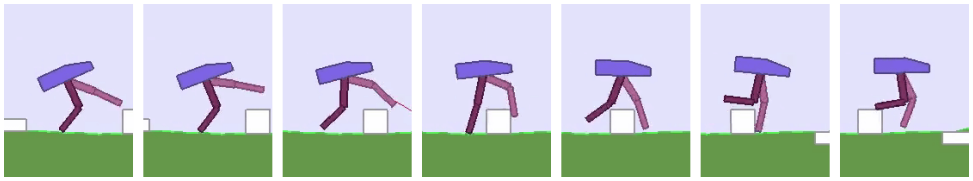
Figure 4.6: Walking Simulation of RFFNN model at best version



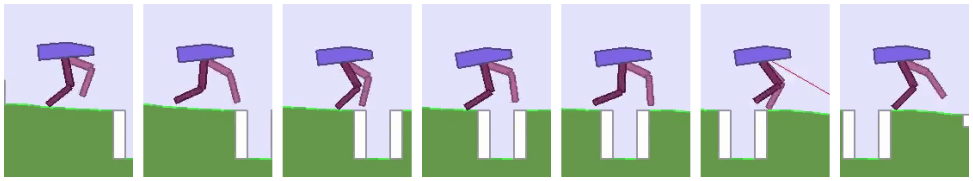
(a) Flat Surface



(b) Stairs



(c) Hurdle



(d) Pitfall

Figure 4.7: Walking Simulation of Transformer model at best version

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, bipedal robot walking is investigated by deep reinforcement learning methods. As stated in previous chapters, most of the real world environments are partially observable. In Bipedal-Walker-Hardcore, the environment is also partially observable since agent cannot observe behind of it lacks of acceleration sensors which is better to have for controlling mechanical systems. Therefore, we used Long Short Term Memory and Transformer Neural Networks to capture more information from past observations compared to single instant observation. Along with them, we also implemented Residually connected Feed Forward Neural Network using single instant observation. Moreover, LSTM and Transformer are compared for partially observed RL problems.

5.2 Future 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 stable RL algorithms.

Secondly, different exploration strategies might be followed. Especially parameter space exploration [30] may perform better since it works better for environments with sparse reward like our environment.

Lastly, another RL algorithms might be applied. Especially, stochastic policy mod-

els might handle exploration problem since they do not require hyperparameters for exploration, learn randomness during iterations.

REFERENCES

[1] BipedalWalker-v2, https://gym.openai.com/envs/BipedalWalker-v2/, January 2021.

[2] BipedalWalkerHardcore-v2, https://gym.openai.com/envs/BipedalWalkerHardcore-v2/, January 2021.

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

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

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

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

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

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

[9] 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, arXiv: 1811.12560.

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

[11] 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, arXiv: 2002.11385.

[12] S. Fujimoto, H. van Hoof, and D. Meger, Addressing Function Approximation Error in Actor-Critic Methods, arXiv:1802.09477 [cs, stat], October 2018, arXiv: 1802.09477 version: 3.

[13] 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.

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

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

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

[17] G. Hinton, Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude.

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

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

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

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

[22] 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, arXiv: 1509.02971.

[23] 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.

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

[25] 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, arXiv: 1312.5602.

[26] 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.

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

[28] 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, arXiv: 1910.06764.

[29] 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.

[30] 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, arXiv: 1706.01905.

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

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

[33] 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.

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

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

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

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

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

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

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

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

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

[47] 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, arXiv: 2002.04745.

APPENDIX A

PROOF OF SOME THEOREM

This is appendix text.

```
%%% Here is the US Census data from 1900 to 2000
%%% Copied from:
%%% https://www.mathworks.com/help/matlab/examples ...
%%%              /predicting-the-us-population.html
%%%

%%% Don't use too long lines

% Time interval
t = (1900:10:2000)';

% Population
p = [75.995 91.972 105.711 123.203 131.669 ...
   150.697 179.323 203.212 226.505 249.633 281.422]';

% Plot
plot(t,p,'bo');
axis([1900 2020 0 400]);
title('Population of the U.S. 1900-2000');
ylabel('Millions');

n = length(t);
s = (t-1950)/50;
A = zeros(n);
A(:,end) = 1;
for j = n-1:-1:1
   A(:,j) = s .* A(:,j+1);
end


```

Listing A.1: The `lintest` function in a floating "listing" environment.