THE CHINESE UNIVERSITY OF HONG KONG

FINAL YEAR PROJECT REPORT (TERM 1)

---

# Applying Modern Reinforcement Learning to Play Video Games

---

*Author:*
Man Ho LEUNG

*Supervisor:*
Prof. LYU Rung Tsong Michael

LYU1701
Department of Computer Science and Engineering

November 29, 2017

# Contents

**Abstract** **vii**

**Acknowledgements** **ix**

**1 Introduction** **1**
1.1 Motivation . . . 1
1.2 Background . . . 1
1.2.1 Reinforcement Learning . . . 1
1.2.2 Challenges of Reinforcement Learning . . . 2
Learning with an unknown model . . . 2
Credit Assignment Problem . . . 2
Exploitation vs Exploration . . . 3
1.3 Objectives . . . 3
1.4 Research Importance . . . 3

**2 Literature Review** **5**
2.1 Efficient Reinforcement Learning through Evolving Neural Network Topologies 5
2.1.1 Genetic Encoding . . . 5
2.1.2 Speciation . . . 7
2.1.3 Incremental Growth . . . 7
2.2 Human-level control through deep reinforcement learning . . . 7
2.2.1 Q-learning . . . 8
2.2.2 Approximate Q-function . . . 9
2.2.3 Experience Replay . . . 9
2.2.4 Target Network . . . 10
2.2.5 Preprocessing . . . 10
2.2.6 Network Structure . . . 11
2.3 Learning from Demonstrations for Real World Reinforcement Learning . . . 12
2.3.1 Deep Q-Learning from Demonstrations (DQfD) . . . 12
Loss function . . . 12
Pre-training phase . . . 13
Training phase . . . 13
2.4 Playing FPS Games with Deep Reinforcement Learning . . . 13
2.4.1 Game Feature Network Augmentation . . . 14
2.5 Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation . . . 15
2.5.1 Policy Gradient and Actor Critic Method . . . 15
2.5.2 Actor Critic using Kronecker-Factored Trust Region (ACKTR) . . . 16

**3 Study of Methodology & Technology** **19**
3.1 NEAT . . . 19
3.1.1 SharpNEAT . . . 19
3.1.2 NEAT-Python . . . 19

3.2 Deep Q-Network . . . 19
3.2.1 Double Q-learning . . . 19
3.2.2 Prioritized Replay . . . 20
3.2.3 Dueling Network . . . 21
3.3 OpenAI Baselines . . . 22
3.4 Self play . . . 23

**4 Game** **25**
4.1 Past Trials . . . 25
4.1.1 Classic Nokia game: Space Impact . . . 25
4.1.2 Popular multiplayer online game: Slither.io . . . 27
4.2 Final Choice: Little Fighter 2 (LF2) . . . 29
4.2.1 Introduction . . . 29
Objective . . . 30
Game Setting . . . 30
Characters . . . 30
Control . . . 30
Other details . . . 31
4.2.2 Selection Criteria . . . 31
4.2.3 Project F . . . 31

**5 Implementation** **33**
5.1 Game abstraction layer . . . 33
5.1.1 Environment Interface . . . 33
Constructor . . . 33
Reset . . . 33
Step . . . 34
Render . . . 34
5.1.2 Game Core Modifications . . . 35
5.1.3 Networking layer . . . 36
Socket.IO . . . 36
Game State Encoding . . . 36
5.1.4 Error Recovery . . . 36
5.2 Feature Extraction . . . 37
5.2.1 Pixel Map . . . 37
5.2.2 Sprite-based Map . . . 38
5.2.3 Handcrafted feature . . . 39
5.2.4 Feature Scaling . . . 40
5.3 Reward Shaping . . . 40
5.3.1 Basic Reward . . . 41
5.3.2 Distance Reward . . . 42
5.3.3 Reward Clipping . . . 42
5.4 Game Setup . . . 42
5.4.1 Map . . . 42
5.4.2 Game Mode . . . 43
5.4.3 Render Speed . . . 43
5.4.4 Character Selection . . . 43
5.4.5 Opponent Type . . . 43
In-game AI 0 . . . 44
In-game AI 1 . . . 44
In-game AI 2 . . . 44

Static Agent . . . 45
RL Agent . . . 45
5.4.6 Operation mode . . . 45
Fixed Mode . . . 45
Random Mode . . . 45
Round Robin Mode . . . 46

**6 Result and Analysis** **47**
6.1 Phase 1: Static Agent Task . . . 47
6.1.1 Description . . . 47
6.1.2 NEAT . . . 47
6.1.3 CNN-based DQN . . . 50
Baseline DQN . . . 50
DQfD . . . 52
DQN with game feature augmentation . . . 53
6.1.4 CNN-based ACKTR . . . 53
6.1.5 MLP-based DQN . . . 54
6.1.6 MLP-based ACKTR . . . 56
6.1.7 Brief Summary . . . 57
6.2 Phase 2: In-game AI . . . 58
6.2.1 Fixed Opponent Mode . . . 58
MLP-based DQN . . . 58
MLP-based ACKTR . . . 61
6.2.2 Random Opponent Mode . . . 63
MLP-based DQN . . . 63
MLP-based ACKTR . . . 64
6.2.3 Brief Summary . . . 65
6.3 Phase 3: Self play . . . 65
6.3.1 MLP-based DQN . . . 66
6.3.2 MLP-based ACKTR . . . 67

**7 Conclusion** **69**
7.1 Summary of key effort and contributions . . . 69
7.2 Future Work . . . 69
7.2.1 Self play . . . 69
7.2.2 Diversify play style . . . 70
7.2.3 Online AI evaluation platform . . . 70

**Bibliography** **71**

THE CHINESE UNIVERSITY OF HONG KONG

# *Abstract*

Faculty of Engineering
Department of Computer Science and Engineering

Bachelor of Science

**Applying Modern Reinforcement Learning to Play Video Games**

by Man Ho LEUNG

In this report, we would apply a wide range of modern reinforcement learning frameworks (e.g. evolutionary method, value iteration method, actor-critic method, imitation learning method) that take advantage of deep learning, in order to train an intelligent agent on a challenging video game environment without using prior knowledge.
Unlike most related work which focus on exploring experimental setups which improves the results on well-established platform such as Atari 2600 [1] and other researched-oriented benchmarks, we focus on solving a modern multi-agent competitive video game environment that is highly dynamic and was never explored before.
In this term, we developed an abstraction layer that separates the details of game emulation from agent design, designed and analyzed a set of experimental setups that significantly improved the performance of existing algorithms. Through exploiting these setups, we trained competitive agents that not only outperformed scripted agents that make use of handcrafted domain knowledge, but also demonstrated a dynamic playing style that surprised human players.
We believed with further exploration, our results could not only test the boundary of latest reinforcement learning, but also be important for the video game industry.

[1] The arcade learning environment (ALE) that allows AI researchers to develop general agents to solve games from the Atari 2600 emulator Stella. It has become a popular benchmark on reinforcement learning algorithm.

# *Acknowledgements*

We would like to express our gratitude to our supervisor Prof. Michael Lyu for providing guidance, feedback and resources. Without his help, we would not have made to this far.

We would like to thank Mr. Edward Yau and Jichuan for the technical support.

Finally, we are grateful to everyone who read this report or supported me in this project.

# List of Abbreviations

| | |
|---|---|
| **ACKTR** | **A**ctor **C**ritic using **K**ronecker-factored **T**rust **R**egion |
| **AI** | **A**rtificial **I**ntelligence |
| **CNN** | **C**onvolutional **N**eural **N**etwork |
| **DQfD** | **D**eep **Q**-Learning **f**rom **D**emonstration |
| **DQN** | **D**eep **Q** **N**etwork |
| **FPS** | **F**irst **P**erson **S**hooting |
| **HP** | **H**ealth **P**oint |
| **LF2** | **L**ittle **F**ighter **2** |
| **MDP** | **M**arkov **D**ecision **P**rocesses |
| **MLP** | **M**ultilayer **P**erceptron |
| **MP** | **M**ana **P**oint |
| **NE** | **N**euro**E**volution |
| **NEAT** | **N**euro**E**volution of **A**ugmenting **T**opologies |
| **POMDP** | **P**artially **O**bservable **M**arkov **D**ecision **P**rocesses |
| **RL** | **R**einforcement **L**earning |
| **TD** | **T**emporal **D**ifference |

# Chapter 1

# Introduction

## 1.1 Motivation

In recent years, Artificial Intelligence has been experiencing a lot of breakthroughs. Thanks to the computational power advancement and research breakthroughs, Artificial Intelligence, especially machine learning, is employed in a wide range of tasks. In some domains, such as image recognition, medical diagnosis, finance prediction, AI has demonstrated its "superhuman" capability under certain settings.
Despite the huge success, many of them are actually attributable to pattern recognition and supervised learning. Reinforcement learning was still generally perceived as the hardest machine learning domain that received relatively little attention. [1]
In 2016, Google DeepMind's AlphaGo, a game AI that plays Go, defeated one of the world best Go players, Lee Sedol. It astonished the whole world and the research community as Go had been an open problem for decades. After a year, even though AlphaGo was challenged by proclaimed stronger players, it still remain undefeated.
In DeepMind's paper [2], it was revealed that the superhuman performance was due to a two phase training that first learned from expert replay, and refined itself using reinforcement learning techniques. In Oct 2017, DeepMind published AlphaGo Zero [3], an enhanced version which reached superhuman level within only 3 days of training. It was further revealed that better performance can be achieved without any supervised data.
DeepMind's publications not only showed the potential of reinforcement learning, but also reshaped people perception towards it. It was generally perceived that since reinforcement learning does not utilize domain knowledge, it can hardly reach human level in complicated domains. Many researchers are eager to test its boundary, by applying it into harder domains.
At the moment of writing, RL frontiers are focusing on using RL techniques to solve video games. Video games are excellent testbeds for RL algorithms as they have much larger state and action spaces, complex objectives and scenarios. In a recent technical report published by DeepMind [4], it was revealed that with their latest algorithms, their agents can hardly win a single game against the weakest built-in AI in the popular real-time-strategy (RTS) game StarCraft II. It demonstrated how much harder the video game domain is compared to traditional board game.
The recent developments and challenges in RL have inspired us to explore its boundary and capability. Therefore in this term, we would like to apply different RL algorithms on a challenging video game environment.

## 1.2 Background

### 1.2.1 Reinforcement Learning

Reinforcement learning is a category of machine learning which aims to optimize the cumulative reward by interacting with the environment and learning from reward signal. Different

from supervised learning, a RL agent was not told which actions are more desirable under a particular state. A typical RL agent will try to discover good actions that yielded reward and reproduce them.
RL environment can be modeled as Markov Decision Process (MDP), which can be represented by the following parameters:

- $S$: a finite set of states.
- $A$: a finite set of actions.
- $T(s'|s,a)$: a transition model that outputs the probability of reaching state $s' \in S$ after committing action $a \in A$ under state $s \in S$.
- $R_a(s,s')$: a reward function that outputs the immediate reward for committing action $a \in A$ under state $s \in S$ and transit to new state $s' \in S$.
- $\gamma \in [0,1]$: the discounted factor, indicating the relative importance of future rewards compare to present rewards.

Formally, the goal of RL is to find a policy function $\pi(s)$ that takes in any state $s$ and output an action such that the following term is maximized:

$$\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1}),$$

where $a_t = \pi(s_t)$.
If all of $S, A, R, T$ are known, the problem becomes a planning problem. However, in a typical RL problem, $R$ and $T$ are not known. In other words, given a state, the agent do not know what new state it will end up after committing an action, or what immediate reward it will get.

### 1.2.2 Challenges of Reinforcement Learning

RL is often referred as the hardest machine learning domain. In this section we will describe a few infamous open challenges RL algorithms face to give a sense of how difficult it is.

#### Learning with an unknown model

As mentioned in last section, the transition and reward function are unknowns in RL problems, which means RL agents have to obtain a "worldview" purely by interacting with the environment, without any explicit guidance. It has to figure out how to evaluate the state, the relation between states and actions simply by trial and error.
To make the matter worse, the state spaces in most models are huge. For example, the number of possible board positions of Go is around $10^{172}$, and the state space of most video games could be even larger. It is simply impractical for RL algorithm to sample all states, not to mention all possible transitions.
In short, RL algorithm would need to approximate the model (or some value functions), and use them to make decision under great uncertainty, which is inherently hard.

#### Credit Assignment Problem

Most RL environments suffer from a delayed reward issue. In many real world RL domains, reward signals are not immediate. The reward signal the agent received in time step $t$ might be a collaborative result of many decisions happened before time step $t - 1000$. In the case of Go, all decisions made in one game yield a single reward (win/lose).

Given an environment with such sparse reward, it is hard for agent to assign credits to actions that led to a particular reward. It complicated the task of deciding which actions are desirable under different states.

**Exploitation vs Exploration**

After the agent interacted with the environment and learned some policies that yielded certain reward, it will try to exploit the strategy over and over again. However, when the agent only explored a tiny fraction of the state space and transition, the policies discovered are likely to be suboptimal.
To encourage the agent to find the optimal strategy, it must perform some stochastic actions once in while. However, if the agent perform stochastic exploration too often, it has no time to refine the strategy it just learned.
The trade-off between these two RL objectives are referred as Exploitation vs Exploration. An ideal RL algorithm will ensure the agent to strike a balance between these two goals, but it is often hard to balance both goals optimally.

## 1.3 Objectives

Following are the objectives we planned for the first term:

- Experiment with different video game environments, and identify the most suitable choice for exploring the capabilities of modern reinforcement learning.
- Design and implement an abstraction layer over the selected video game environment, such that different RL algorithms can interact with the game with an unify interface, without concerning about the low level details (graphics, networking, game setup, etc). The layer should be designed to be highly configurable and provides all necessary game information to allow efficient training and benchmarking.
- Broadly experiment with multiple contemporary RL algorithms on the game, under different experimental settings (e.g. model parameters, feature representation, reward model, gaming mode, heuristics), to determine which class of algorithms and configurations perform best under this game environment.
- Focus on the most promising class of algorithms and configurations, then apply heuristics to boost the training furthermore and make it reach reasonable level.

## 1.4 Research Importance

RL has experienced a burst of developments during past few years. Recent researches such as AlphaGo give the world a glimpse of what RL can achieve, its true potential is yet to be uncovered. Therefore, any attempts to test its boundary are inherently meaningful.
In addition, for video game industry, our research is an exploration of a new way to create agents in competitive game genres.
At the moment, most video game AIs are scripted with heavily handcrafted logic. As agent behaviors are explicitly programmed, sometimes it can be easily exploited by players and make the game less amusing. To make the game AI more dynamic, game developers often need to hardwire complicated logic flow and duck tape fixes into the game, which increase the implementation costs.
In the past, game industry refrained from using RL due to practical reasons, RL agent was often hard to train to match the level of scripted agent. However, with the recent enhancements in

computational power and RL researches, whether these assumptions still hold are remain to be seen.
Therefore, our research can potentially be a proof of concept, showing that without using any game domain knowledge, RL agents can not only surpass the level of scripted AI, but also demonstrate a more dynamic play style than scripted AI, which makes it more enjoyable to play with.

# Chapter 2

# Literature Review

## 2.1 Efficient Reinforcement Learning through Evolving Neural Network Topologies

In 2002, a new RL paradigm NeuroEvolution of Augmenting Topologies (NEAT)[5] was proposed as the future of Neuroevolution (NE) method. NE is a class of RL algorithms that use evolutionary algorithms to generate neural network parameters, topology as the policy network [1]. During that time, NE was considered as the strongest method on many RL tasks such as pole-balancing. However, Conventional Neuroevolution (CNE) method is constrained to evolve the weights on fixed topology network.
NEAT was claimed to be the first attempt that evolve both network weights and topology, and learn significantly faster than CNE.

### 2.1.1 Genetic Encoding

Just like other genetic algorithms, NEAT requires a scheme to encode the solution (network) such that mutation and crossover is defined and computationally easy. .

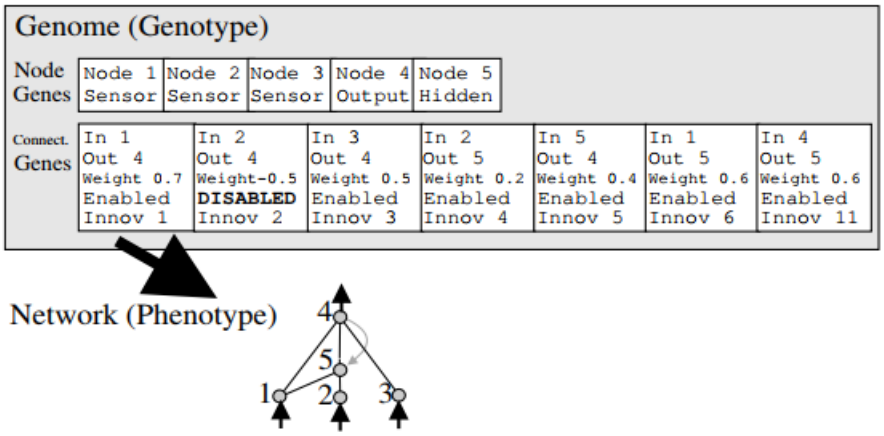


FIGURE 2.1: An example NEAT genome

The genome consists of genes that specify the information about the nodes and connections. For each connection, the involving nodes, weight, an enabled bit (whether the connection is active), and an innovation number are encoded.
In NEAT system, two types of mutation are possible, namely connection weight and structural change (adding connections and nodes).

[1] A network that takes in a state and output the action.

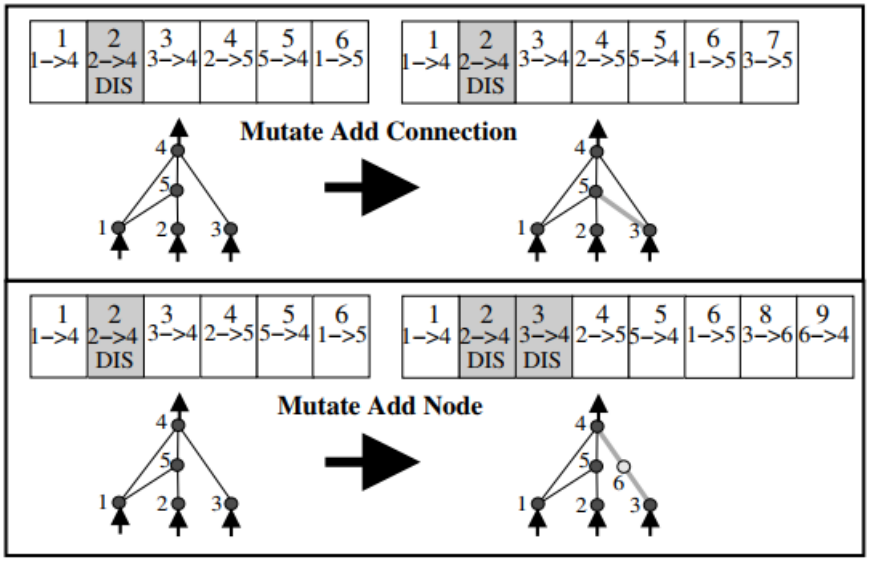


FIGURE 2.2: Illustration of types of NEAT mutation. The number written on top of each gene represent the innovation number generated during mutation.

For adding connection, a new gene specifying the change will be appended to the genome.
For adding node, the old connection will be disabled, while two connection genes will be appended to the genome.
It is worth noting that every time a mutation gene is generated, it is attached with an globally incremented innovation number.
For crossover, the two parents will first align their genes according to the innovation number.

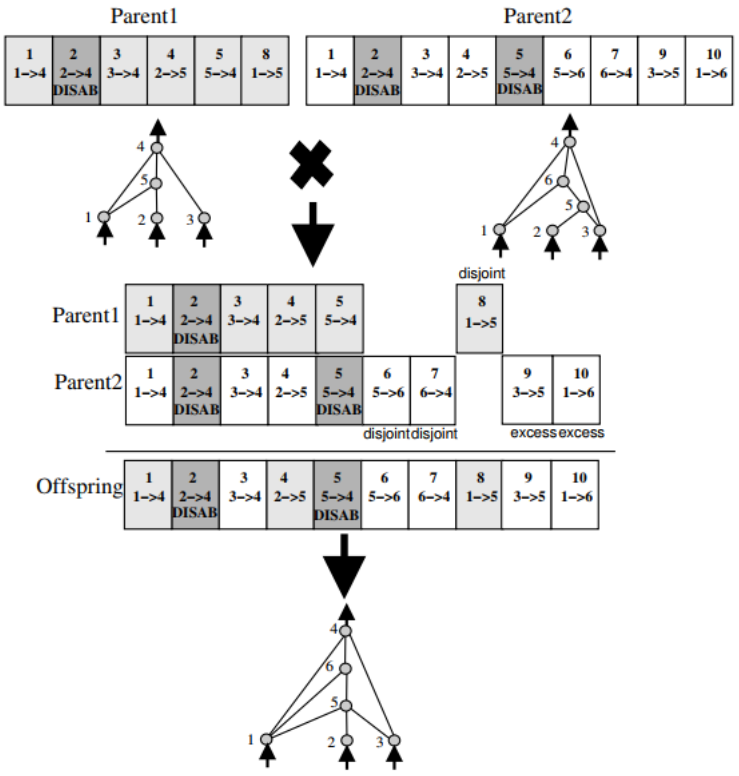


FIGURE 2.3: Illustration of NEAT crossover

Genes that match will get randomly inherited from either parent in the offspring.
For the genes that does not match each other, only the genes from the fitter parent will be inherited. In case of a draw, all mismatched genes from both parents will get inherited.

### 2.1.2 Speciation

The authors discovered that, when performing structural mutation on an established network, its fitness is likely to drop. As a result, topology innovations are likely to be extinct within one generation if the traditional genetic algorithm is applied. Therefore, they proposed a new concept called Speciation.
They first designed a metric measuring the differences between two genomes, based on number of genes with mismatched innovation number. Recall that innovation number is coupled with a specific feature inside a network. Two genomes with many mismatched genes are unlikely to share similar structures.
With this metric, all genomes can be separated into species, such that same species have relatively similar structure. The fitness score of each genome inside a species is normalized by the size of the species, to avoid any species to dominate the whole generation.
The species size in next generation is determined by:

$$\frac{\sum_{i=1}^{N_j} f_{ij}}{\overline{f}},$$

where $\overline{f}$ is the average normalized fitness of current generation, $f_{ij}$ is the normalized fitness of the $j$ genome in species $i$. After the species size is determined, the best performing genomes of each species will randomly crossover to generate that number of offspring.
With this setting, even a new genome does not perform well compare to the whole population, the main competition will be held among genomes with similar structures (same species), different topologies are still preserved.

### 2.1.3 Incremental Growth

The authors also proposed that by starting with a uniform population of networks without hidden nodes, and incrementally introduce structural mutations, it gives NEAT a performance advantage compared to other approaches. They argued and experimented that under such method, NEAT will only evolve complex network if the need is justified.

## 2.2 Human-level control through deep reinforcement learning

In 2015, Google DeepMind published a variant of Q-learning (DQN) that learn successful policies directly from high-dimensional sensory inputs[6]. They tested the agent on the classic Atari 2600 games, using only screen pixels and the game score as inputs, it surpass the performance of all previous algorithms and achieve human level across 49 games.

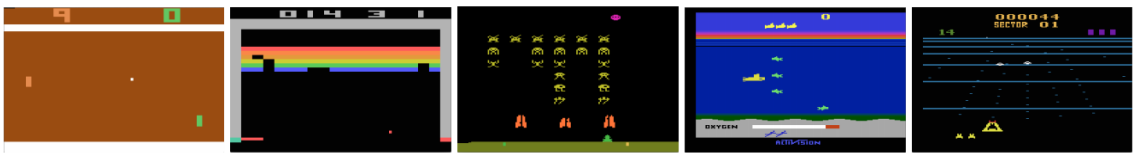

FIGURE 2.4: 5 games from Atari 2600 environment.

The high level idea of DQN is to use a deep convolutional neural network to approximate the Q-function. DeepMind was not the first to use non-linear function approximator to approximate the Q-function. However, many[7] have pointed out that such approaches are highly unstable. Many of the past success are either restricted to low-dimensional domains, or inefficient to use with large neural network. In this paper, DeepMind addressed these instabilities and obtained results that are unmatched by other attempts:

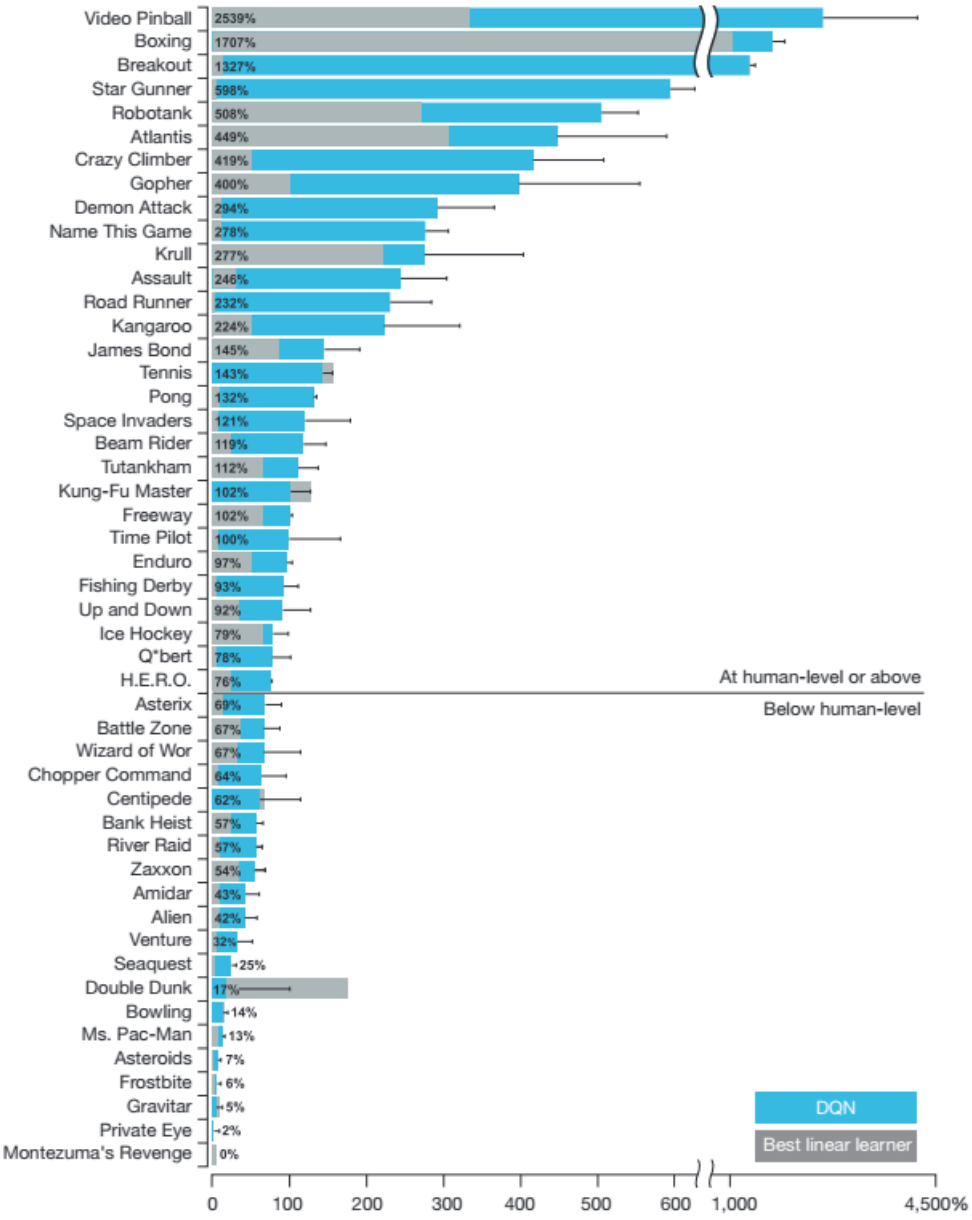


FIGURE 2.5: Normalized score achieved by DQN in different Atari 2600 games.

As DQN is a fundamental component in our project, we will study its principles in detail in this section.

### 2.2.1 Q-learning

In Q-learning, Q-function $Q(s, a)$ is defined as the maximum discounted reward the optimal policy can get by committing action $a$ under state $s$. The key idea of Q-learning is that the

optimal Q-function fulfill bellman equation, that is, assuming after committing $a$ on state $s$, the agent received reward $r$ and end up in state $s'$:

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a'),$$

where $\gamma$ is the discounted factor, denoting how important future reward is, $0 \leq \gamma \leq 1$.
In traditional Q-learning, we iteratively explore the RL environment, and update Q table value until they converged. In the $i^{th}$ iteration, assume we encounter a transition tuple $(s,a,r,s')$. The update would be the following:

$$Q_i(s,a) = (1-\alpha)Q_{i-1}(s,a) + \alpha(r + \gamma \max_{a'} Q_{i-1}(s',a'))$$

where $\alpha$ is the learning rate, $0 \leq \alpha \leq 1$.
Another way to interpret this updates is:

$$Q_i(s,a) = Q_{i-1}(s,a) + \alpha \underbrace{(r + \gamma \max_{a'} Q_{i-1}(s',a') - Q_{i-1}(s,a))}_{\text{TD-error}},$$

where the second term are often referred as Temporal Difference (TD) error of Q-learning.
In order to learn the Q-values, the agent has to explore a wide range of transitions. Usually, Q-learning follow a $\epsilon$-greedy exploration, where the agent will commit the action with highest Q-values on its state with probability $1-\epsilon$, and commit random action with probability $\epsilon$.
In practice, $\epsilon$ is usually a high number and will be annealing to a low number during training. The reason that such schedule is being used is that in early phase, the policies are crudely formed and are likely to be a suboptimal solution. Therefore we need to explore actions that lead to states that we have not seen before. In later phase, the current policies need to be refined so it is better to follow the policy.

### 2.2.2 Approximate Q-function

Traditional Q-learning has been proven to converged to optimal policy given iterations [8]. However, one could observe that in order for $Q(s,a)$ to be updated, the agent must have explored the exact state and action pair at least once. In most domains, the state spaces alone is large enough such that it is impossible to visit them all. Therefore, traditional Q-learning is impractical in complex domains.
In reality, states space are often highly correlated – similar states might have similar Q-values. Motivated by this, many have proposed to use function approximator such as neural network to generalized a Q function based on a limited amount of transitions.
If we express the Q function approximated by a network with weight $\theta$ as $Q(s,a;\theta)$, the loss function can be defined by:

$$L(\theta) = E_{s,a,r,s'}[((r + \gamma \max_{a'} Q(s',a';\theta')) - Q(s,a;\theta))^2],$$

where $\theta'$ is network weights from some previous iterations. One can notice the resemblance between this error form with the TD-error of traditional Q-learning 2.2.1.
If we hold $\theta'$ as a constant when calculating the gradient of the $L(\theta)$ with respect to $\theta$, one could reduce the task into a well defined optimization problem.

### 2.2.3 Experience Replay

Although the above loss function is similar to the one we see in typical supervised learning and seems can be solved using the same techniques (e.g. gradient descent). If we follow traditional

Q-learning and always use the latest transition to train the value network, several side effects will happen:

- Consecutive transitions are highly correlated, while optimization techniques assume the data comes from independent and identically distributed sampling.
- The data distribution used to train the current network are affected by policy emitted by the network itself, which cause the network parameter to diverge catastrophically.

To tackle these issues, DeepMind proposed Experience Replay. Instead of learning directly from the most recent transitions, DQN stores $P$ latest transitions into a replay buffer. In addition, it perform stochastic gradient descent on a randomly sampled minibatch. If $P$ is large enough, the randomness in sampling can break the correlation across transitions. In addition, the data distribution contributed by different network parameters will be averaged out.

### 2.2.4 Target Network

Recall the DQN TD-loss function:

$$L(\theta) = E_{s,a,r,s'}[\underbrace{((r + \gamma \max_{a'} Q(s', a'; \theta'))}_{\text{target}} - \underbrace{Q(s, a; \theta))}_{\text{prediction}})^2],$$

where $\theta'$ is network weights from some previous iterations.
DeepMind pointed out that, the first term $(r + \gamma \max_{a'} Q(s', a'; \theta'))$ is actually correlated to $Q(s, a; \theta)$ since $\theta'$ is just an older version of $\theta$. If we update $\theta'$ to the latest version too often, the changes applied to the $\theta$ will affect $\theta'$ in a similar way, which is likely to cause the network parameters to oscillate or diverge. An analogous way to describe this issue is that our network is training towards a moving target.
To reduce this effect, DeepMind proposed to update $\theta'$ with a sufficiently long interval, which turns out to improve the performance a lot.

### 2.2.5 Preprocessing

Before feeding the high-dimensional input to DQN, a sequence of preprocessing is required:

- To reduce the dimensionality, the Atari game frame of size ($210 \times 160 \times 3$) will convert to grayscale and downsampled to $110 \times 84$.
- Since convolution neural network requires the input to be a square, the frame will be cropped into $84 \times 84$.
- To enable DQN to distinguish movement from static frames, 4 consecutive frames are stacked as if an image with 4 channels. The final input will contain dimension of $4 \times 84 \times 84$.

### 2.2.6 Network Structure

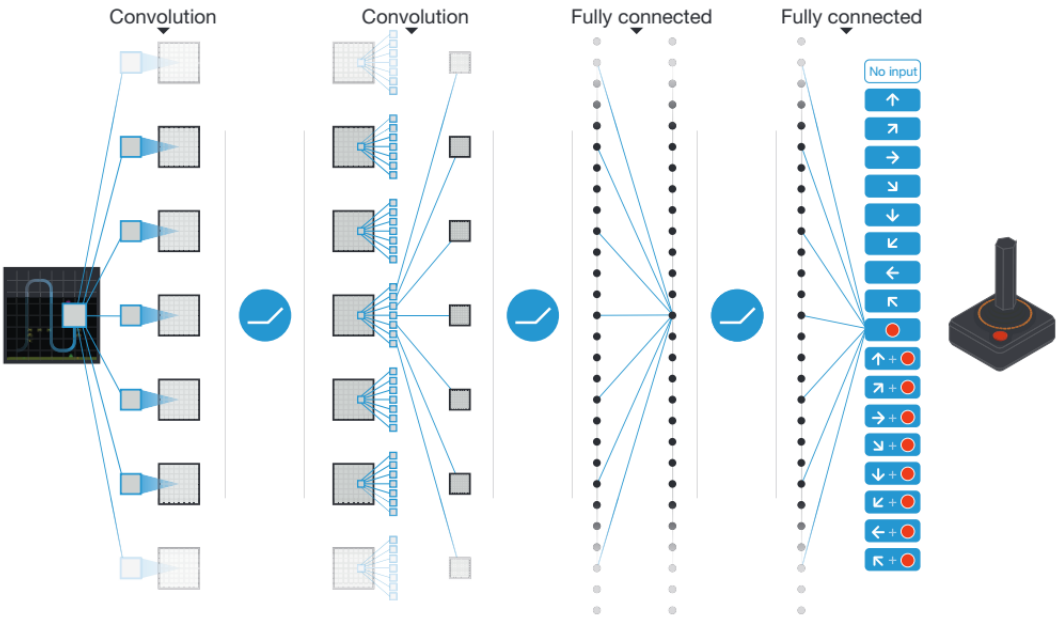


FIGURE 2.6: A high level description of general DQN structure.

The input layer will take in a stacked frame of size $4 \times 84 \times 84$ as state representation.
Then the input will go through three layers of convolution layer, a fully connected layer and an output layer.

| Layer | Num filters | Filter size | Stride | Activation | Output |
|---|---|---|---|---|---|
| conv1 | 32 | 8x8 | 4 | ReLU | $20 \times 20 \times 32$ |
| conv2 | 64 | 4x4 | 2 | ReLU | $9 \times 9 \times 64$ |
| conv3 | 64 | 3x3 | 1 | ReLU | $7 \times 7 \times 64$ |
| fc1 | | | | ReLU | 512 |
| output | | | | Linear | Number of actions |

TABLE 2.1: The actual network structure used in the paper.

The output layer will output a Q-value for each action.

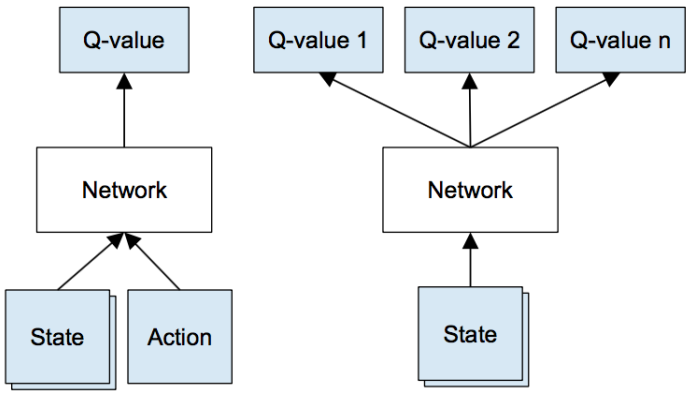


FIGURE 2.7: Two possible ways to approximate a Q-function. The right way is used in the paper.

The reason why the neural network does not take an extra input $a$ and output a single Q-value $Q(s, a)$ is that the current setup can obtain Q-value of all actions with only one forward pass, which saves computation proportionate to the number of actions.

## 2.3 Learning from Demonstrations for Real World Reinforcement Learning

In many published experiments, it can be observed that the initial performance of DQN (during first few millions frames) is usually extremely poor. For some difficult domain, DQN might fail to learn anything at all.
For these domains, it is natural to think if we could somehow guide our RL agent to learn by providing expert demonstrations. In research field, imitation learning is a domain that aims to leverage expert data to boost the performance of RL agent. However, these approaches usually require huge amount of expert data for the agent to behave well. Such expert data are usually expensive to obtain and require longer time to train. In addition, the trained agent are not capable of improving the policy exhibited by the expert. As a result, the learned policy might not be self-consistent.
In April 2017, DeepMind published a new algorithm Deep Q-Learning from Demonstrations (DQfD) [9] that is based on DQN. The algorithm leverages a small set of expert demonstration data (5,574 to 75,472 transitions), learn from their behavior and also try to refine it to make it self-consistent.

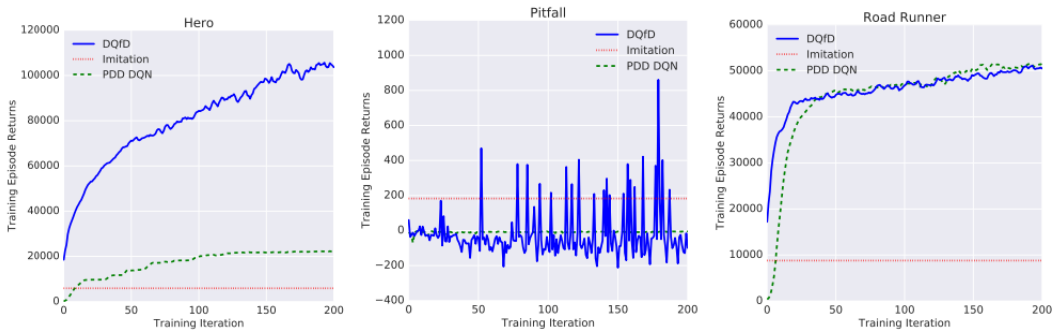


FIGURE 2.8: DQfD performance compared to pure imitation learning and DQN in three Atari games.

The algorithm was once again tested on the Atari game set, and the result from 2.12 shows that it not only outperform the performance of human demonstration, but also surpass algorithms that use pure DQN and pure imitation learning in some of the hardest games.

### 2.3.1 Deep Q-Learning from Demonstrations (DQfD)

DQfD is different from the original DQN in many ways, in this section, we will explain briefly key changes.

#### Loss function

DeepMind proposed the following loss function $J(Q)$ for DQfD:

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q),$$

where $J_{DQ}(Q)$ is the original one step TD-loss from DQN, $J_n(Q)$ is the $n$ step TD-loss (details omitted here), $J_E(Q)$ is the expert supervised loss, $J_{L2}(Q)$ is the L2 regularization loss. $\lambda$ are simply the weights of individual error terms.
The L2 regularization loss prevent the network to overfit on the relatively small expert data set. The expert supervised loss is defined as:

$$J_E(Q) = \max_{a \in A}[Q(s,a) + l(a_E, a)] - Q(s, a_E),$$

where $l(a_E, a)$ margin function that evaluates to zero when $a_E = a$, positive otherwise.
Since we are minimizing the loss function, such construct forces the network to push down the Q-values for actions that does not match expert choice under same state $s$. In short, this will make expert actions more likely to be chosen.

#### Pre-training phase

Before any interaction with the environment, it will first run gradient descent solely on several batch of demonstration data.

#### Training phase

During training phase, as usual, it will interact with the environment and store transitions into experience replay buffer. However, when performing gradient updates, it will not only sample from the replay buffer, but also from the expert transitions. The mix ratio is determined by a mechanism explained in the paper which we will not elaborate here.
When training from the replay buffer (self-generated experience), the term $\lambda_2 J_E(Q)$ is undefined so we will set $\lambda_2 = 0$. When training from the expert data, we will use the full error form $J(Q)$.
Under such training, the algorithm can learn to follow expert action while also learn a value function that minimize the TD-loss.

## 2.4 Playing FPS Games with Deep Reinforcement Learning

In this paper [10], the authors applied a variant of DQN on a 3D first person shooting (FPS) environment, VizDoom, using high dimensional pixel input. In addition it trained an agent that surpass built-in AI under certain scenario.
VizDoom is a challenging environment that is three dimensional as well as partially observable. The player is required to navigate around the environment to pick up tools and eliminate enemies.


FIGURE 2.9: Actual gameplay of VizDoom.

The baseline that the authors tried to use to tackle the environment is Deep Recurrent Q Network (DRQN), a variant of DQN that aims to solve environment that requires memory.

However, such model failed to learn any reasonable strategy. The best agent trained under this scheme will basically keep firing and expect the enemies to come under its line of fire. If penalty of firing is given, it does not fire at all. In short, the agents failed to navigate around environment and is not responsive to the observation.
They argued that the reason behind the failure is the agent failed to detect the existence of particular game entity. To mitigate this problem, a new architecture is proposed.

### 2.4.1 Game Feature Network Augmentation

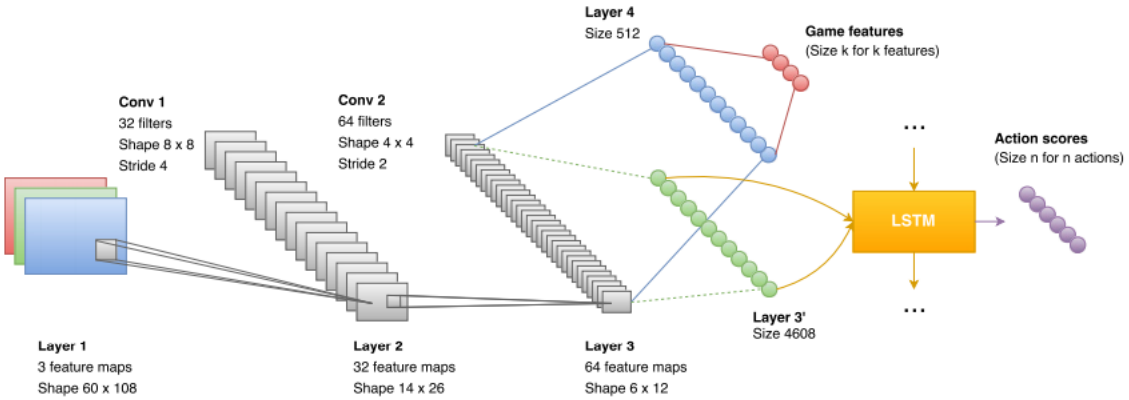


FIGURE 2.10: The network structure augmented with game feature layers.

The new architecture basically split the output of convolution neural network (CNN) layer to two networks, namely the LSTM network and the game feature detection network.
The game feature detection network (from layer 7) will basically output a game feature vector that denote the existence of $k$ different game entity.
The author modify the game engine such that it returns the target game feature vector during training. In addition, the following loss function is used:

$$J(Q) = J_{DQ}(Q) + \lambda J_G(Q),$$

where $J_{DQ}(Q)$ is the original TD-loss of DQN, $J_G(Q)$ is the cross entropy error between the target and output game feature vector. $\lambda$ is the weight of the game feature error.
They showed that under such construct, the network reached an game feature classification accuracy of 90% in a few hours. In addition, with this setup, the learning is improved significantly.

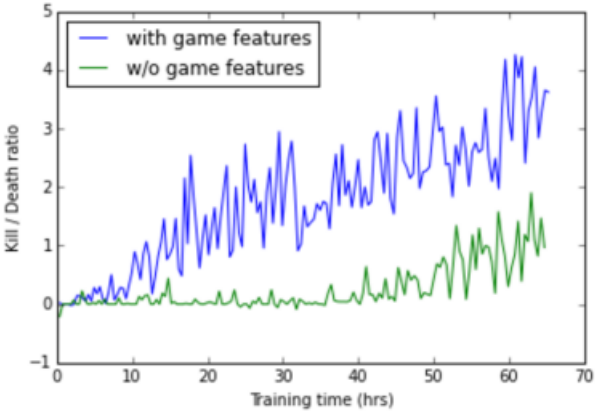


FIGURE 2.11: The kill-death ratio versus training hours of models trained with and without the game feature augmentation layers.

They also experimented with alternative network architecture such as using a separate network to learn about the game features and inject its prediction vector directly into the LSTM layer. However, the result was not as good as the above setup.
This implies the performance boost was in fact attributable to sharing the CNN layers. One high level explanation is that, by sharing the CNN layers, it will guide the CNN filters to extract features that is related to game entity, which are eventually helpful to learn the Q-function.

## 2.5 Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation

Recently, OpenAI has published a new RL approach [11]. This method produced impressive results even compared to an improvement over DQN in terms of training time and data efficiency[2]. In this section, we will study the high level idea of this approach. [3]

### 2.5.1 Policy Gradient and Actor Critic Method

In this section we will describe another framework of RL – Actor Critic methods, a framework that ACKTR uses.
In section 2.2, we described a novel architecture that make used of Q-learning, a value iteration method. Value iteration method basically try to learn the optimal value function that evaluates state $V(s)$ or state-action pair $Q(s, a)$.
For Q-learning, the optimal policy $\pi(s)$ is defined by:

$$\pi(s) = \max_a Q(s, a),$$

it will return the action that return the largest future discounted reward.
There is another class of RL algorithms called Policy Gradient method. This method takes a more direct approach – estimating the optimal policy function $\pi(a|s)$, a probability distribution over actions $a$ under $s$.

[2]The amount of frames required for the model to reach a certain level

[3]The contributions of this paper are mostly mathematical, since it is not the our research focus, the exact mathematical contribution will not be described here.

The objective of this method can be mathematically expressed as finding a parameterized policy $\pi(a|s;\theta)$ such that the following term is maximized:

$$J(\theta) = E_\pi[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)],$$

where $\gamma$ is the future discounted factor, $a_t$ is the action sampled by the policy distribution, $r(s_t, a_t)$ is the reward of taking action $a_t$ under $s_t$. The term is simply the expected $\gamma$-discounted reward.
To maximized this term, one could compute its gradient. It has been proven to be the following form:

$$\nabla_\theta J(\theta) = E_\pi[\sum_{t=0}^{\infty} \psi^t \nabla_\theta \log \pi_\theta(a_t|s_t)],$$

where $\psi^t$ is often chosen to be the advantage function $A^\pi(s_t, a_t)$, expressing how better the value of action $a_t$ compare to other actions on the state $s_t$, $\nabla_\theta \log \pi_\theta(a_t|s_t)]$ is the direction in parameter space which leads to the maximum increase on $\pi_\theta(a_t|s_t)$.
By applying the gradient on the parameters $\theta$, one can improve the expected future reward.
One high level interpretation is that, under state $s_t$, for actions $a_t$ that have a positive advantage (relatively good action), we would apply the gradient of log policy to tune up the probability of $a_t$, vice versa, and this gives us the direction to improve expected reward.
Since $A^\pi(s_t, a_t)$ is defined as the relative "advantage" of an action on a particular state. One could estimate the term by using a variant of the following form: [4]

$$A^\pi(s_t, a_t) = r(s_{t+1}, a_{t+1}) + \gamma V(s_{t+1}) - V(s_t),$$

where $V(s_t)$ is the expected future reward on $s_t$, independent of the action taken.
To conclude, to compute the policy gradient, one could do so by learning both value function $V(s)$[5] (a function that evaluates/criticizes a state) and policy function $\pi(s)$ (a function that gives us an probability distribution of optimal actions). Such method are often referred Actor Critic methods.

### 2.5.2 Actor Critic using Kronecker-Factored Trust Region (ACKTR)

The proposed method ACKTR, is one of the latest Actor Critic methods. Similar to DQN, ACKTR exploit neural network as its function approximator. For discrete control task, the ACKTR uses a single network to output two layers, denoting the value function and policy function.
Unlike most work that rely on first order method like stochastic gradient descent. The author argued that SGD is not the most efficient method to explore the weight space of a large model such as a neural network.
The contribution of this paper was mainly to apply Kronecker-factored approximated curvature (K-FAC) to compute the gradient. They showed K-FAC is a more scalable method in the sense that it has comparable computation cost to SGD, with a much improved data efficiency.
In addition, ACKTR interact with the environment with parallel agents to break the correlation within minibatches. This method was showed by previous work [12] that can achieve the same effect as experience replay. This also makes the algorithm parallizable.

[4] Please note that the form used by different literatures are a lot complicated than the following lower the variance during estimation.

[5] One could simply use the TD-loss function similar to Q-learning for learning the value function.

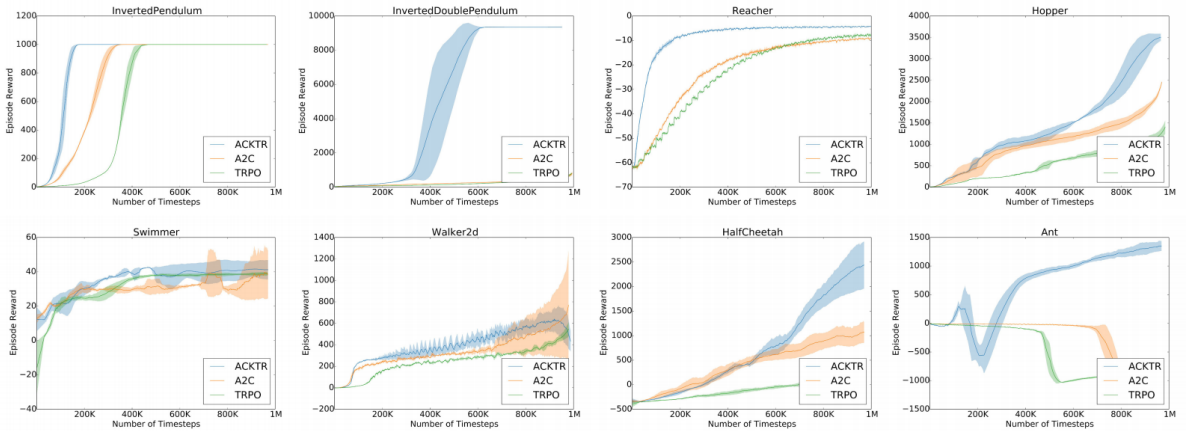


FIGURE 2.12: Performance of ACKTR compared to other state-of-the-art methods on MuJoco benchmarks.

ACKTR beat several state-of-the-art such as A2C (an improvement over DQN) and TRPO (a famous trust region optimizer TRPO) not only in many Atari environments, but also for continuous robotic control benchmark.

# Chapter 3

# Study of Methodology & Technology

## 3.1 NEAT

As mentioned in chapter 2, NEAT is an evolutionary algorithm that search the optimal policy network by evolving both structures and weights. In this project, two different implementations are used.

### 3.1.1 SharpNEAT

SharpNEAT is an opensource implementation of NEAT written in C# and support .NET environment. We mainly used this implementation in early phase of the research, where we are testing NEAT with Space Impact.

### 3.1.2 NEAT-Python

NEAT-Python is an opensource implementation of NEAT written in Python. This environment was mainly used in later phase, when we discovered a majority of resources and code base written in Python. To reduce the number of interprocess call during training, we decided to use a Python implementation of NEAT.

## 3.2 Deep Q-Network

In the section 2.2, we introduced a novel algorithm that was proposed by DeepMind in 2015 that obtain huge success on various domains. Since the paper was published, many techniques are proven effective to solve some of the known DQN issues.
In this section, we will group the techniques proposed by several literatures and briefly cover the difference between this version and the vanilla DQN.

### 3.2.1 Double Q-learning

Recall the loss function of vanilla DQN:

$$L(\theta) = E_{s,a,r,s'}[\underbrace{((r + \gamma \max_{a'} Q(s', a'; \theta'))}_{\text{target}} - \underbrace{Q(s, a; \theta))}_{\text{prediction}})^2],$$

where $\theta'$ is the parameters of the target network (older version of the current network) and $\theta$ is the parameters of the online network (latest).
Some papers [13] provided evidences that the way DQN estimate the target tends to overestimate the Q-value of many state-action pairs. Assume that the true value of $Q(s, a)$ over any action $a$ under any state is 0. Since we are using an approximator $Q(s, a; \theta')$ to learn the true value, with vanilla DQN, it is likely to output values that close to (but not exactly equal to)

0. When we are taking a max operator over the set of all possible actions – $\max_{a'} Q(s', a'; \theta')$, we are likely to get some positive noise. It will in turn overestimate the target, hence, after the gradient update, the Q-value of the current network will be pushed up to match the target.
To mitigate this issue, DeepMind proposed Double Q-learning [14]. It basically try to use one network to pick the action $a$ with highest Q-value and another network to evaluate the corresponding Q-value of that action $a$. In other words, the new loss function becomes:

$$L(\theta) = E_{s,a,r,s'}[\underbrace{((r + \gamma Q(s', \max_{a'}(Q(s', a'; \theta)); \theta'))}_{\text{target}} - \underbrace{Q(s, a; \theta))}_{\text{prediction}})^2]$$

Under this formulation, the noise produced by $Q(*, *; \theta)$ is unlikely to match the noise produced by $Q(*, *; \theta')$. In long run, these noise tends to cancel out each other, which resolves the overestimation issue.
In principle, the two network parameters $(\theta, \theta')$ used to form the target should be trained separately using two experience sets to make them completely uncorrelated. However, they discovered empirically by using parameters of older version of current network as $\theta'$, the benefit is still preserved. Therefore such method is more preferable to minimize the structural changes [1] to the original DQN algorithm.

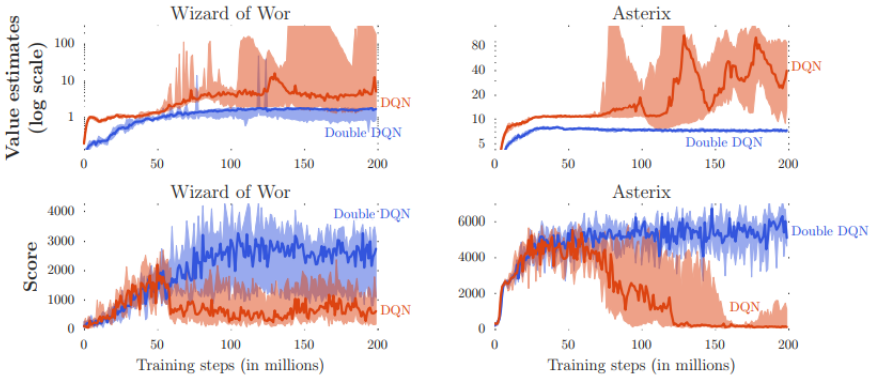


FIGURE 3.1: Averaged Q-values estimation and reward of vanilla DQN and Double DQN during training on two Atari games.

Figure 3.1 shows that it not only resolve the value overestimation problem, but also improve the quality of the trained policy significantly.

### 3.2.2 Prioritized Replay

Recall in vanilla DQN, in each batch update, the transition are sample uniformly random from the replay buffer. However, in reality, not every transition are equally important. Recall the TD-error term of DQN for a single transition tuple $(s, a, r, s')$:

$$L(\theta) = \underbrace{((r + \gamma Q(s', \max_{a'}(Q(s', a'; \theta)); \theta'))}_{\text{target}} - \underbrace{Q(s, a; \theta))}_{\text{prediction}})^2$$

For transition that yield high TD-error, after the backpropagation, the effect to the network will be larger. It is analogous to human learning process, new observations that are not align with our understanding are often more valuable for us to correct our model.
In view of this, DeepMind proposed to use Prioritized Replay [15]. The high level idea is to assign a probability $p$ portion to its TD-error to each transition. Such that when we are

[1] Recall that $\theta'$ is already a network required by vanilla DQN.

sampling a batch of transitions to perform gradient updates, the transition are sampled with their respective probability.
This turned out to improve training progress significantly.

### 3.2.3 Dueling Network

In many games, there are states $s$ that the true Q-value of $Q(s,a)$ is not affected by what action $a$ it take. For example, in a game screen that contains no enemy, no matter what action the agent take, it is unlikely to affect the future discounted reward.
In vanilla DQN setup, the Q-value of every possible action need to be learned separately. In order to learn the action-independent Q-value for all actions, many transitions on similar state have to be observed. This process turns out to be inefficient, especially for games with large action space.
To use this insight, DeepMind separate Q-value into the following terms:

$$Q(s,a) = V(s) + A(s,a),$$

where $V(s)$ is the value function that returns how much discounted future reward the agent can get on state $s$ (independent of action) and $A(s,a)$ is the advantage function that denoting relative measure of the importance of each action.
To realize this, one has to split the output of CNN layer of vanilla DQN as figure 3.2.

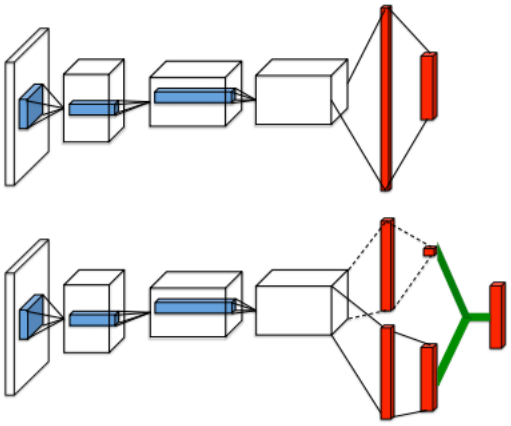

FIGURE 3.2: Comparing the structure of vanilla DQN (upper) and dueling network (lower).

In the lower network, one can see the CNN layer are split into two networks.
The first network estimates the value function $V(s)$, it outputs a scalar evaluation of the state $s$.
The second network estimates the advantage function $A(s,a)$, it outputs the advantage estimate of each possible action.
To recover the $Q(s,a)$, each output of the advantage network will be added to the state network output.
The whole architecture is referred as dueling network [16].
Intuitively, to learn a Q-value that is independent of any actions for a state $s$, the dueling network can learn the appropriate value of that $s$, then the Q-value of all possible actions $a$ are adjusted.
This turned out to be another important optimization that improve learning substantially.

## 3.3 OpenAI Baselines

Implementing modern RL algorithms are challenging. According to a report published by OpenAI [17], among ten samples of popular opensource modern RL algorithm implementation (including DQN), six of them contains flaws. These flaws can either make the agent learn suboptimal strategy, or prevent the algorithm from converging.
Following are some of the well known reasons why implementing high quality modern RL algorithms are so challenging:

- Most modern RL algorithm are usually expensive to run [2]. This makes testing time consuming and expensive.
- In many paper, implementation details and hyperparameters are not explicitly listed. Due to the unstable nature of RL training process, subtle differences in these could have huge effect on the performance. It is inherently difficult to find a decent combination implementation details and parameters.
- Performance of RL algorithm is hard to verify comprehensively. One way of testing the correctness of RL algorithm is to reproduce the result that is on par with the published results. Since not every opensource developer can afford the computational cost, many implementations can only be tested on a few environments.

OpenAI baselines [18] is a set of high quality implementation of modern RL algorithms (which includes DQN) released and maintained since May, 2017. The project is lead by OpenAI research group, which is formed by a lot of researchers and owns a lot of computational resources. The project objective is to provide reliable baseline for other researchers to work on top of. Each of these baselines are heavily tested on many environments, and are on par with the published results.
Currently, the repository contains baselines of the following algorithms:

- Advantage Actor Critic (A2C)
- Actor Critic using Kronecker-factored Trust Region (ACKTR)
- Deep Q-Network (DQN)
- Deep Deterministic Policy Gradients (DDPG)
- Proximal Policy Optimization (PPO)
- Trust Region Policy Optimization (TPRO)

Its DQN implementation support double Q learning, prioritized replay and dueling network, as a result, it served as a baseline for many of our experimentations.
In addition, we have also experimented with their A2C, ACKTR, PPO implementation on our finalized game environment. In the end, ACKTR was selected to be another focus of our research.
These implementations use Python and TensorFlow [3] for defining the network models, loss function and computing gradients.
By using OpenAI Baselines, it really help us to focus on the following work:

- Comprehensive logging to help understanding the training progress more.

[2] Some of the DeepMind DQN experiments require 7 days to run even with top GPU model.

[3] TensorFlow is an open source software library for numerical computation using data flow graphs. It is one of the most popular library for defining machine learning models such as neural network. It is released and maintained by Google Brain Team.

- Creating checkpoint and progress recovery tools to allow us to observe and interact with the agent flexibly.
- Modify network structure, loss function and work flow to implement novel variants of the baselines.
- Interpreting the experimental results.

To conclude, OpenAI Baselines is one of the most important tools for our research.

## 3.4 Self play

In many previous work, self play has been proven useful to boost performance of reinforcement learning algorithm on a competitive environment. TD-Gammon [19] is the first RL agent that reach expert level in backgammon, the learning process is basically using transition from self-play to perform TD-learning. Another famous example is AlphaGo Zero [3], the RL-based Go AI that reach superhuman level purely by self play. [4]
There are various motivations for using self-play to enhance RL algorithm:

- The agent is always competing with the opponent of similar level, which forces the agent to continuously improve itself.
- The RL agent experiences various playing style throughout the training process, which might make it learn more general strategy rather than strategy that is tailer-made for the weakness of a small set of experts. For example, the agent might need to learn how to defend against strategy that is not performed by any expert. This might lead to discovery of interesting policies.

In this term, we focus on a simple form of self play [5], we will always make our RL agent confront with the latest self.

[4]Note that the self play mechanism exploited in AlphaGo Zero, is somehow quite different from TD-Gammon.
[5]This approach is similar to self-play in TD-Gammon.

# Chapter 4

# Game

In this chapter we will cover the decision making process before we selected the final game environment.

## 4.1 Past Trials

In this section we will describe about two games where we tried to apply RL on and explain why they are not chosen as the final video game environment. These pilot tests are conducted for two main reasons:

- To scout for ideal testbeds for modern RL algorithms.
- To explore the traits and limitations different RL algorithms.

In this section we will study these trials in brief to see

### 4.1.1 Classic Nokia game: Space Impact

Our first trial was conducted on the famous Nokia game, it is a simple mobile game that is pre-installed in older versions of Nokia phones. The game objective is to control the movement (4 directions) of a space ship and its fire and survive as many levels as possible.
In each level there will be enemies with different shapes, move pattern and firing pattern. Some enemies will follow a fixed trajectory while some will trace the space ship.
Reward will be given after the spaceship destroyed an enemy or survived for a level. The space ship will be destroyed whenever it is in contact with an enemy or its fire.

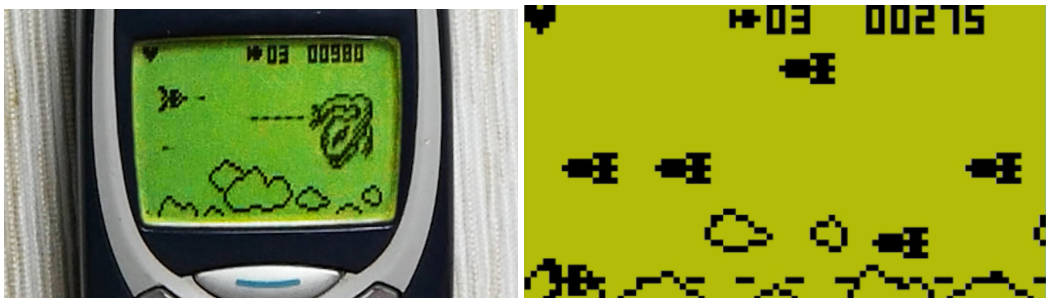


FIGURE 4.1: Space Impact on an old Nokia mobile and its pixel-perfect clone

The trial was made possible as its opensource pixel-perfect clone was released in 2016 under educational license [20]. The clone was written in C with Simple DirectMedia Layer (SDL).
The RL algorithm we experimented with this environment is NEAT, we utilized the Sharp-NEAT (C#) implementation. After modifying the game core and establish a socket connection between the NEAT fitness evaluator and the game, the game can continuously send its state

and receive action commands. And the NEAT fitness evaluator will receive the fitness (game score) in the final frame.
After running the NEAT with setups in table 4.1, we managed to obtain satisfying result.

| Parameter | Value |
|---|---|
| Population size | 300 |
| Initial connected portion [1] | 0.5 |
| Number of species | 10 |
| Activation | ReLU |

TABLE 4.1: NEAT parameters used to train to play Space Impact.

Given only one life [2], our best genome managed to solve 4 out of 6 levels. Furthermore, from the gameplay, it demonstrated several skillful traits.

- From the trajectory of our agent, it clearly knows how to move away from malicious objects such as enemies in different shapes and their fire. Besides, we discovered that it knows the difference between powerups and enemies. For example, it always move away from the enemies if possible, but if a powerup is approaching, it will stay there.
- The agent knows to occupy positions that intersect with least enemies' trajectories. Then it will use that position as center and keep oscillating and firing to eliminate incoming enemies that is currently not intersect with the line of fire, but will eventually collide. This strategy helps the space ship clear up a narrow safe zone.

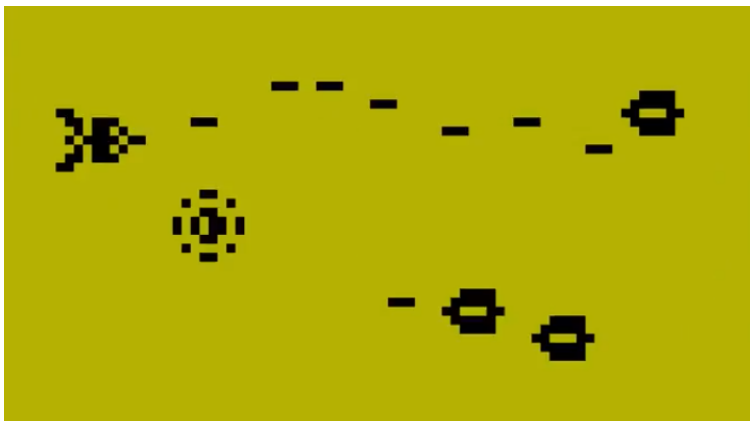

FIGURE 4.2: NEAT agent learned to oscillate and fire from a position to eliminate a range of enemies.

The above behaviors show that not only did our agent obtain the basic game sense, but it also know how to exploit the game element to make smart moves.
However, NEAT does not work directly out of the box. To make the training run smoother, the following techniques are employed:

- To test NEAT ability to handle high dimensional feature, we initially use whole pixels (4032) as the state. However, we discovered no matter how many generations we run the algorithm, the resulting policy is simple. The agent will only fly to the lowest row and keep firing, which will help it get some points but still it is a suboptimal strategy. The neural structure failed to complexify (zero hidden neurons). We later switched to

[2] default gameplay have three lives.

reduce the dimensionality (to 252 pixels) by averaging pixels, which solved the issues. The algorithm started to evolve more complex policy and neural topology.

- We observed that even with the dimensionality reduction technique, the neural structure NEAT evolved are generally simple (less than 10 hidden neurons). It is believed that its ability to distinguish game objects are limited. Therefore we decided to remove game objects that are irrelevant to gameplay, such as the background and game statistics to allow the agent focus on the game object that is relevant to gameplay.
- We observed that the design of the fitness function can drastically affect the training progress. Initially, we defined the fitness as the score starting from running game from the first level until it dies (just like the default game flow). However, we observed that the evolution is progressing very slowly (fitness score of generation champion remain unchanged for more than 50 generations). Consider the case when a few genomes were stuck on a level 2 (say, encounter a new boss), when they try to mutate some more general skills to overcome the challenge, even if the new skills are useful to level 3 or above, as long as it does not help the genome get better score on level 2, these differences will not get reflected in the fitness score. And those promising genomes will not get enough chance to develop. In other words, NEAT will flavor genomes that overfit level 1-2 rather than a more general genome that can get better result in long run.

  We argue that by changing the fitness score to the summation score of that genome in all levels, it will encourage NEAT to develop general genome and hence boost training. We ran the NEAT with the new setting for 34 generations, and the resulting champion already demonstrate the skill comparable to old setting that is ran for 500 generations. This result show us that the design of fitness can impact the training progress significantly.

Although there are still some tasks we could explore on this Space Impact environment, we decided that it is not a good testbed for other modern RL frameworks.

### 4.1.2 Popular multiplayer online game: Slither.io

The second environment we explored is Slither.io, a massively multiplayer browser game developed by Steve Howse in 2016. The concept of this game is similar to the classic game Snake, but in a competitive setting. This game was ranked by Alexa as one of the 1000 most visited sites by July 2016, which shows little about its popularity.

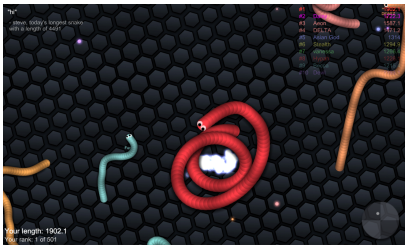

FIGURE 4.3: Gameplay of Slither.io.

In this game, player first starts with a little snake, and need to control it to consume as many powerups as possible to make it grow larger. Whenever a snake crashes into the body of other snakes, it will disseminate into powerups proportionate to its size (the snake being crashed will still be alive). The challenge of this game is to avoid crashing into other snakes, while consume as many powerups as possible.

The direction of the snake is controlled via mouse, or left/right arrow (turning clockwise/anti-clockwise). In addition, the snake can consume its own powerups to move faster (charging) to get away from danger, or make other snakes crash into its body.
We thought that Slither.io is a better testbed for modern RL algorithms compared to Space Impact for the following reasons:

- Even the game settings are incredibly simple, since it is a competitive setup, the agent always need to evolve to match the level of its opponents in order to get more reward. Just as AlphaGo, the policy learned by RL is actually more complex than the Go rules. A competitive environment provides more room for us to explore compare to environment that can be solved entirely (e.g. Atari, Space Impact).

- From observing human gameplay, we learned that a wide range of skills are required to excel in this game. One can charge into the expected trajectory of other snakes, in order to make them crashes into your snake body. One can exploit the advantage of its size, and try to surround other smaller snakes by making a huge circle, narrowing down the circle diameter until all surrounded snakes eventually crashes due to lack of space.

  These behaviors make us wonder whether they are discoverable by RL.

At the moment, Slither.io is not opensourced. However, thanks to an initiative lead by OpenAI – OpenAI Universe, applying AI algorithms in this environment is possible.
OpenAI Universe aims to provide a simple interface over games, websites and other applications, without any access to their source code. Universe achieve this by packaging an application's dependencies into a Docker container, and provide the screen pixels from the running container to the agent. As for reward signal, Universe use techniques like OCR to obtain such information from the screen pixels.
Currently, Universe contains around 1000 environments, including Slither.io. This allows us to run a RL environment without taking care of low level details. After setting up the environment, we ran the Asynchronous Advantage Actor-Critic Agents (A3C) [3] algorithm [12] with 8 parallel agents for more than 1M total frames, however the results are disappointing.

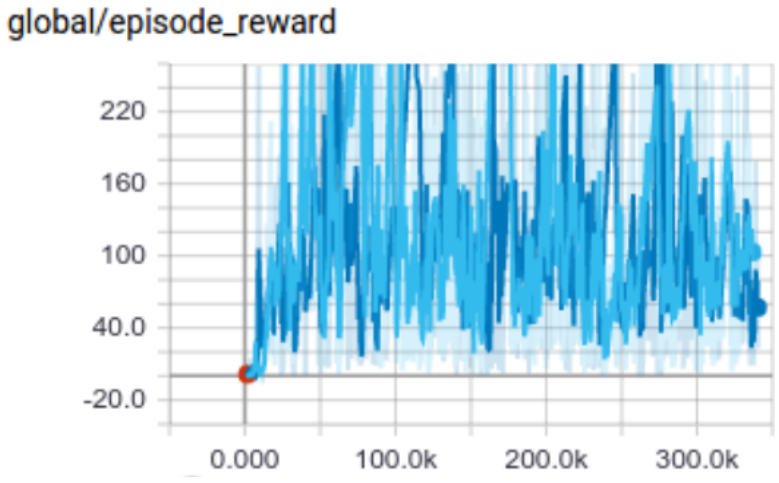


FIGURE 4.4: Training curve of A3C on slither.io. The two lines represent the episodic reward of two agents, other agents are removed for clarity.

[3] We used a baseline implementation provide by OpenAI.

As one could see in the chart, the training process are extremely unstable, and the max size our snakes reach is below 300 [4]. While inspecting the gameplay, all 8 agents are not skillful at all and will only move to a fixed direction.
We argued that the reason that A3C does not work well under this environment are as follows:

- As Slither.io is an internet game running in real time, one cannot pause the environment while the RL algorithm is running. While the RL algorithm is making an action, multiple frames with be dropped. Since the running speed of the RL algorithm can be easily affected by many factors, it makes the number of frames dropped unstable. By the time the RL agent emit an action, the current frame might be a lot different than the frame the agent used to make that decision. We argued that this is partly responsible for the unstable training.

- The game environment is dependent on a lot of factors.

  Firstly, since every snake in the environment is controlled by actual human and the game server assigned based on geolocation, there will be time when the server is filled with less people.

  Secondly, the game world is a large circle, where most players are centered in the middle region. Depends on where the snake is spawned initially, the game environment can be quite different. The snake can move for 5 minutes in sparse region without seeing other snakes, but got instantly surrounded by snakes in the dense region.

  While we know that little randomness in the environment can help the RL algorithm learn more general strategy, we argued that the current inconsistencies between runs actually make it impossible to learn reasonable strategy. Imagine that our RL agent just managed to learn some basic moves agent other snakes, but for the next 10 rounds it get tossed into a corner where it hardly see any other snakes. Most of our RL models, like other neural network algorithm, are suffered from catastrophic forgetting, it is unlikely to refine the features it learned long time ago, explaining the unstable training process.

Since these issues are tightly coupled with the nature of Slither.io, it is unlikely that we can circumvent these limits. In addition, while exploring Slither.io along with OpenAI Universe, we discovered that the latter is actually a complicated machinery to work with. Sometimes we encountered issues such as a failed Docker container, but it is non-trivial to investigate which components failed. This motivated us to find a better environment.

## 4.2 Final Choice: Little Fighter 2 (LF2)

### 4.2.1 Introduction

Little Fighter 2 (LF2) is a Hong Kong free fighting game that was created by Marti Wong and Starsky Wong in 1999 [5]. After its release, it went viral in Hong Kong for years and become a collective memory of 80s and 90s generation. According to Marti Wong [21], it was once one of the top 10 game search in Yahoo for three years. The game's success was not constrained in Hong Kong, it has also received worldwide recognition. Even till now, active LF2 communities can be found over the world, holding regular tournament.

[4] For reference, amateur player can easily reach 2000+

[5] One interesting fact is that the prequel of LF2 was actually created as a Final Year Project in CUHK CSE by Marti Wong under the supervision of Prof. Michael Lyu.

FIGURE 4.5: Gameplay of LF2.

### Objective

In the combat mode of LF2, player will control a selected character to fight against players in other teams. The objective of combat mode is to eliminate them.

### Game Setting

The game world is a 3D environment projected into a 2D scene, the exact scene size depends on different maps.
Each player has a fixed amount of health point (hp), when attacked by other players, hp value will be decreased, and when it reaches zero, the character will be dead. If the hp is not full, it will regenerate slowly.
Each player has a fixed amount of mana point (mp) as well, it is required for launching special attacks. Similar to hp, if mp is not full, it will regenerate as well.
There will be special game objects scatter around the scene/dropping from the sky randomly that can be used as weapons, it allows players to launch attack from a greater distance.

### Characters

Each player can selected one of 11 basic characters, each character has a unique set of special attacks. Some characters have mostly close range attacks, while others will have mostly long range attack.

### Control

LF2 have 7 operations, namely {up, down, left, right, attack, jump, defense}.

- {up, down, left, right} are basic movement keys that enable the player to move to the respective direction. If the horizontal movement keys are activated twice, it will trigger the running operation.
- "attack" key will launch a punch or projectile attack (arrow), depending on the character design. If the attack key is trigger during different states (e.g. idling, running, jumping), it will produce attacks that causes different damages.
- "jump" key will trigger the character to jump. Depending the original state of the player, it will jump to different directions.

- "defense" key will trigger the character to enter a defense state. If the character is facing the incoming attack. It can reduce the amount of damage taken.

Aside from the these basic operations, character's special attack can be launched by executing different key sequence (e.g. defense->down->jump->attack). These special attacks are generally more powerful than basic attack and it will consume various amount of mp depends on the attack strength.

**Other details**

After character took several consecutive attacks, or suffered from a great hit, it will enter a coma mode for a few frames. During coma mode, no enemy can attack it further and the character cannot move freely.
After reviving from coma mode, the character will first enter transparent mode for a few frames, during which no enemy can attack it, but the character can move freely (but its attack cannot hurt anyone). Using this short interval, the suppressed character can therefore move to a better position for counterattack.
This setting make the game more balanced and allow more chance for the disadvantaged side to come back.

### 4.2.2 Selection Criteria

Gathering the lessons we learned from the last two attempts, we have generalized a few criteria for picking a suitable environment. In this section we will use these criteria to justify why LF2 is most suitable.

- Just as we mentioned in last section (Slither.io), a competitive environment allows more room for exploration. LF2 is a multiagent competitive environment, which is desirable in this sense. In the future, we could even explore the cooperative behavior of RL agents in team match (e.g. 2v2, 3v3).

- LF2 is unlike some other fighting game that feature merely fast reaction and action-per-minute (APM). It is well known for its incredibly balanced game setting in terms of the characters and skills. In order to excel in the game, one has dodge and deflect others' attack, surprise opponent by attacking from different angles, exploit game objects and location. It is meaningful to see if modern RL algorithms is capable for reproducing these behavior.

- LF2 comes with several in-game AIs that are considered challenging to mid-level players. These scripted AIs are very helpful in our research as it provide a baseline for us to compare our RL-powered AI. The win rate of running these AIs against RL agents, is one of the key performance index (KPI) of this research.

- Most importantly, a copyright-free opensource implementation of LF2 is available on github. We are able to establish full control on the game core. This allows us to extract features from the game directly, which is a cleaner approach compare to OpenAI Universe.

  The details of it will be described briefly in next section.

### 4.2.3 Project F

Since the source code of LF2 is not publicly available, Project F [22] is an attempt to reverse engineer it such that everyone could modify the game without any restriction. In order to

make it copyright-free, the author follows a strict clean room implementation process. The source code is available via github.
The game is written in HTML5 and Javascript, as a result, it can be run on most platform with web browser (e.g. Linux, Windows, Mobile). Although it does not contain all features of the original LF2, the game dynamics in the combat mode are highly similar to the original.
Just like the original LF2, it contains several scriptable and selectable in-game AI, although the design and behavior are different from the original, it is still challenging to mid-level players.
We were once face a dilemma of whether to use we should use the original game as environment or Project F. If we were to use the original game, we would need to create a Docker image that package all LF2 dependency, extract the game screen in the running Docker container via VNC viewer, and perform OCR to get the internal state of the game (e.g. hp bar of each player). While these are still technically possible to implement, it will be increasingly challenging if we decide to use feature other than screen pixels, or refine the reward mechanism later.
Due to above concerns, we have decided to use Project F as our codebase to create a RL environment.

# Chapter 5

# Implementation

## 5.1 Game abstraction layer

As mentioned in section 4.2.3, Project F is a web application that runs on browser. To allow any algorithm establish control on the game and receive game states, one has to manage miscellaneous tasks such as networking, preprocessing, benchmarking. To separate the concerns and allow the algorithm implementation to focus on RL itself, an abstraction layer is created. Under this abstraction layer, different RL implementations can adapt to a simple interface without worrying about the low level details.
The abstraction layer consists of two major components, namely the web application and the Python program that provides the interface. The choice of Python is naturally motivated by the abundance of Python Machine Learning resources.

### 5.1.1 Environment Interface

Our Python RL environment interface [1] contains mainly four methods, namely `constructor`, `reset`, `step`, `render`, we will now explain in details what these methods accomplish.

#### Constructor

It is the method (`__init__`) that takes in all user defined options (e.g. feature types, game setting) and responsible for tailor-made the actual game environment that is specified by the parameters.
It will first create a web server in a separate thread, the server is responsible for all the communications between the game core and Python program.
After that, it will start another thread and use Selenium to launch a headless browser [2], we will drive the browser to the URL that is encoded with configurations. The configurations include game configuration options and the port number of the running web server. The port number is selected in a way that it will never collide with other game instances or applications to ensure multiple game instances can run concurrently.
After starting two threads, the main thread of the program will be sleeping until the web server received acknowledgement from the game core, denoting that the game is ready to run.

#### Reset

This method allows the user to reset the environment without reconstructing the whole interface, which is expensive.

[1] The interface design is heavily influenced by the design of OpenAI Gym environment.
[2] We decided to chrome with chromedriver as we discovered it is uses less resources compared to Firefox for running a single tab application.

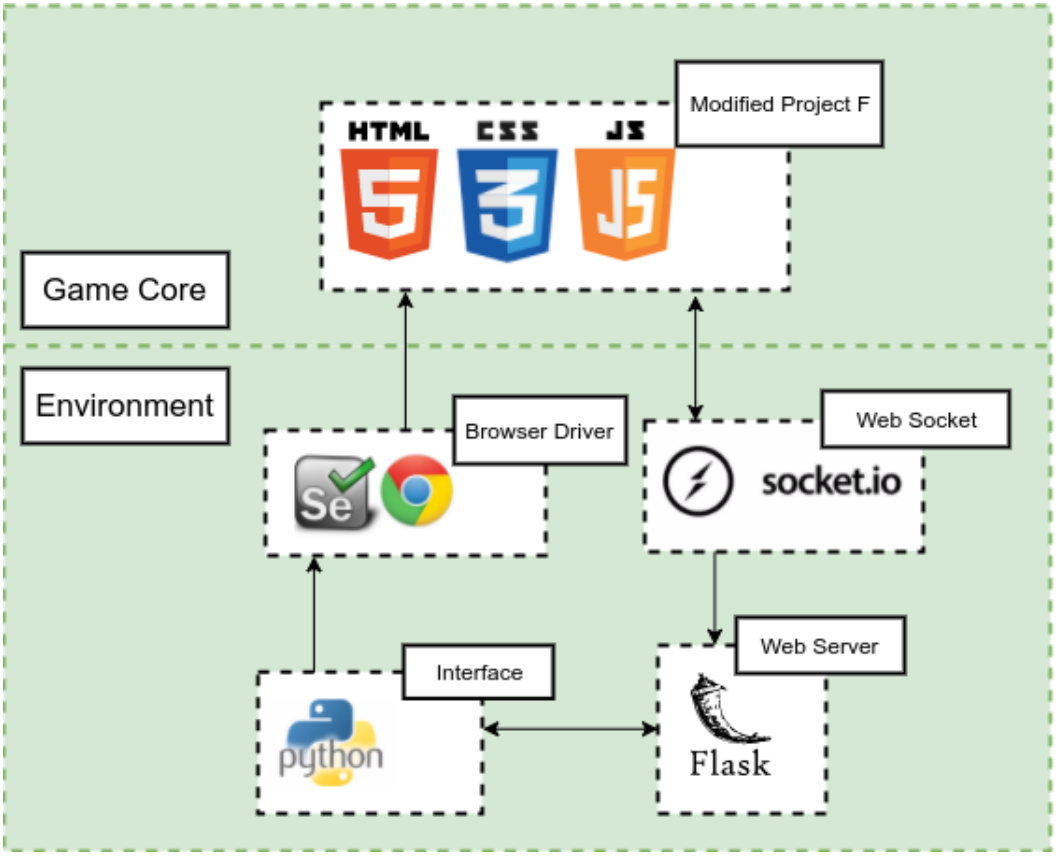


FIGURE 5.1: System design diagram

It will emit a reset signal to the game core, meanwhile, it will be sleeping until a feedback signal is received.
The feedback signal denotes that the game core has been properly reset, and it will contain the first game state, which this method will return.

### Step

This method allows the user to commit an action and proceed to the next state.
It takes in a single argument – the action id.
It will first send the action signal to the game core, meanwhile, it will be sleeping until a feedback signal is received. The feedback signal contains a tuple of game information after executing the corresponding action. The tuple contains the following four information:

- The new game state.
- The reward obtained since last frame.
- Whether the game is over or not.
- An associative array containing all additional information about the game, such as the opponent AI id.

### Render

This method allows the user to visualize the preprocessed game states (if it is pixel-like) in real time, it is useful for inspecting what the RL algorithms are actually seeing.

Every time a new state is feedback to the program (in `step` and `reset`), the latest game state is saved in the environment object. When `render` is called, it will show that in a simple image viewer.

### 5.1.2 Game Core Modifications

In order to make the game core work with the above Python interface, several modifications to Project F codebase is required. In this section, we will cover the major changes.
Originally, when the game is launched in a browser, it will enter an interactive UI, where the user need to select the play mode, choose the character and in-game AIs and game settings. And eventually start the game. After the game is finished, it will redirect the player back to character selection scene.
To run the game continuously, we would need to automate the above process.

- When the game core is executed on a browser, we will first extract all URL parameters that are passed by the Python program.
- By manipulating the scene manager, we skip directly into the character selection scene.
- According to the options specified in URL parameters, we will select the corresponding map, opponent, opponent type programmatically. In addition, we will create our RL-controlled agent as just another in-game AI.
- By manipulating the match manager, we create and start a game instance along with our customized options.

In order to establish control on a game character, we implement the interface of Project F in-game AI with several modification.
Originally, when an in-game AI is constructed in the game, it will be provided with reference to the current game instance and a buffered controller. The in-game AI interface expect an `update` function that would be called periodically, in which the AI will perform calculation based on the state in the game instance. By registering keystroke using the controller, the corresponding actions will be committed in next frame.
By exploiting this design, we created a new in-game AI class that is controllable by our Python program.

- When our AI object is constructed, we will create the socket client object and connect it to the Python server endpoint specified in the URL parameters.
- In the first invocation of `update`, the match scene has finished rendering, so we will send an acknowledgement signal (alongside the first game state) to server and freeze the timer used by the game core. The signal will release the Python environment from the blocking state in `constructor`.

  The game core will enter pause mode after the timer was frozen.
- When we receive an action signal from the Python program, we will register the specified key into the controller and resume the timer. In next invocation of `update`, we will compute and send the updated game information tuple to the server. Then we will freeze the timer again.
- The above step will be repeated until the game is over.

Other than these modifications, most of the preprocessing of game information are implemented inside the AI script. The details of these would be discussed in later sections.

### 5.1.3 Networking layer

The game simulation speed affect how fast a RL algorithm can learn.

#### Socket.IO

To achieve low latency and overhead between each data exchange, we decided to use Socket.IO as the communication channel. Socket.IO will try to establish a WebSocket connection between server and client if possible, which is more efficient than plain HTTP calls for frequent data exchange.
In order to process Socket.IO connection in our Python program, we use the opensource Flask-SocketIO that allows us to run a simple Flask server that can handle callback in a separate thread.

#### Game State Encoding

In many experiments, we used high dimensional pixels as input. In these cases, our program performance becomes network-blocking. We discovered that by optimizing the way way we pack our message, we can actually reduce the bandwidth by 3-4 times and speed up the training process by 50%.
In principle, Socket.IO will apply gzip to any message being transmitted and that should remove most redundancy. Originally, we transmit the whole json pixel array to server in each update. However, we soon found out that by encoding each 8-bit pixel into a character, the size of the compressed payload is reduced by about 4 times.
We also tried using modern compression algorithms and encoding methods, such that the pixel array is more compactly packed. While it does further shrink the bandwidth (by 20%), it requires a more complicated unfolding phase in the Python side, which in turns did not improve the speed. Therefore we decided to stick to our simple encoding method for transmitting pixel states.

### 5.1.4 Error Recovery

The complete abstraction layer consists of multiple components, namely the Python program, Flask web server, browser driver (chromedriver), browser (chrome), JS game engine.
As RL algorithm usually need to use the layer for more than days, many unexpected situation can arise. In early phase, we encountered miscellaneous intermittent issues:

- Browser killed the game tab due to out-of-memory (OOM) issues.
- Game engine threw JS execution error, hence escaped the expected execution flow.
- Flask web server missed a feedback message from client as the thread was blocked at that moment.

When any of these happen, the Python program will stuck at waiting for feedback. And the experimentation process, that had run for days, was disturbed.
Even with verbose logging, identifying the sources of these hiccups are still challenging due to its intermittent nature. Moreover, some issues are highly dependent on the condition of running server and out of our control.
To prevent such issues, we incorporated the following changes:

- In any situation where our Python server is waiting for feedback from remote client (`constructor, reset, step`), we limit the longest waiting interval to be 20 seconds.

- After certain interval has passed, we will turn on an error flag. In the next invocation of `step` [3], we return an artificial game information tuple stating that the game is over, and reset the error flag.
- In user's point of view, a crashed game is just a game that end prematurely. It will just call `reset` for next game simulation.
- In the end, we will establish a fresh connection with a new socket client.

This implementation turned out to be robust and make training process more smooth and reliable.

## 5.2 Feature Extraction

Feature extraction is the representation of game state to be fed into RL algorithms. It dictates all information the model can use during the training process and has profound effect on the performance.
Throughout term one, different features have been experimented on various models. The result and analysis on how these features affect the model performance will be described in next chapter.
In this section, we described what does these features include and how are they calculated.
Note that to reduce the network bandwidth, instead of transferring the whole game state to Python program and perform preprocessing downstream, we implemented everything in the frontend.

### 5.2.1 Pixel Map

Inspired by the huge success DeepMind and others on training RL agents purely on pixel input, it is natural for us to consider it as one of the feature types.
The original game screen has a size of $550 \times 794 \times 3$, the complete game screen is not only expensive to transmit over network, but also challenging for most existing RL algorithm to learn. To produce the pixels that can be effectively learned, we do the following preprocessing:

- Crop the region that is most related to the game play (battle ground). The hp, mp status of characters, while relevant to the game play, are cropped out since we considered the information are less helpful for training considering the portion of space it occupy.
- Reduce the three channel image into gray scale using luminosity method.
- By averaging pixels, reduce it to size $64 \times 64$. Note that the aspect ratio is not kept during this process. The choice of a square dimension is to allow the image to be learned by model that apply convolution on the input.

[3] for error that occurred during `reset`, the current invocation

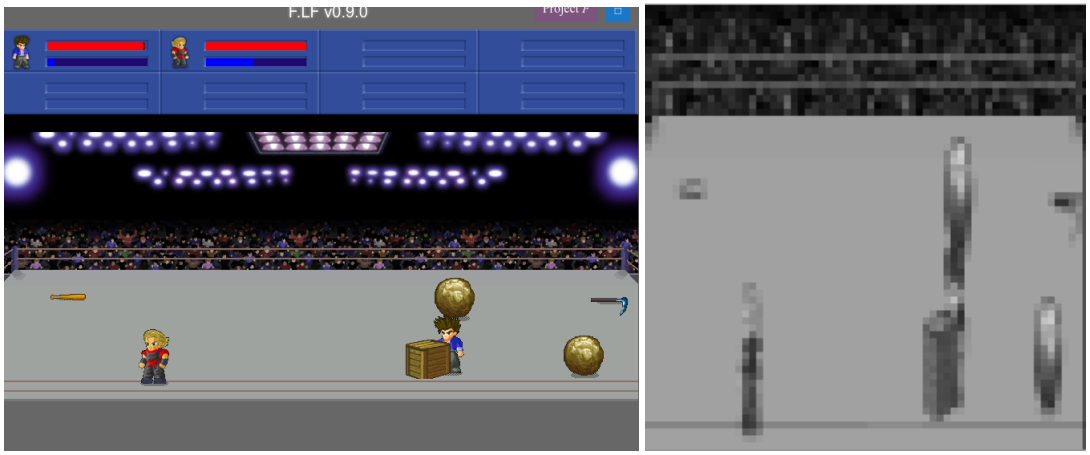

FIGURE 5.2: An illustration of the state transformation.

By using pixel map, we provide a complete representation of the game environment to the RL algorithm.

### 5.2.2 Sprite-based Map

As mentioned in previous attempts, not all RL algorithms are capable of handling high dimensional input such as pixel map. To further reduce the input dimensionality, one can simply further reduce the pixels. However, we argued that such approach will make the observation blurry and noisy, which is undesirable for any RL algorithms.
Through reviewing related work in this domain, we came across a method that is used to resolve such problem and obtained successes. It was also used in the viral video published by developer Seth Bling, whom in the video, explained his approach of using this feature on the game Super Mario in training a NEAT agent:

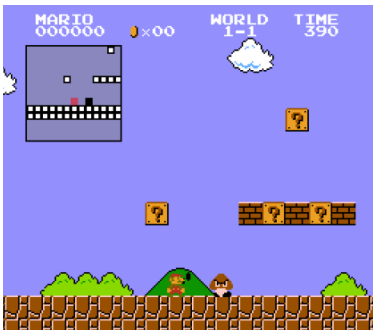


FIGURE 5.3: The state representation used by Seth Bling in his Mario NEAT project.

It represents the game state using a small grid system, in which each cell encodes the corresponding 2D game space. The encoded value is determined by what kind of game sprite it contains.
This feature selection encodes the spatial information of all game sprites, and preserve details such that the agent is able to distinguish game objects. In addition, it achieve such purpose using fewer pixels.
Our sprite-based was designed under following settings:

- The grid system we use has a size of $13 \times 13$, which is the same size as the input Seth Bling used to train his NEAT agents. By picking such size, we want to ensure the dimension is manageable for NEAT.
- If a grid cell contains a sprite object, the cell values are set with a predefined value that corresponds to the sprite type. The values are designed in a way that different game sprite contain significantly different values.
- As a baseline, we encode three game sprites differently: the controlled agent with first value, the opponent with second value, other objects with three value.
- If a grid cell contains multiple sprite of different types, their respective value is added up to that cell.

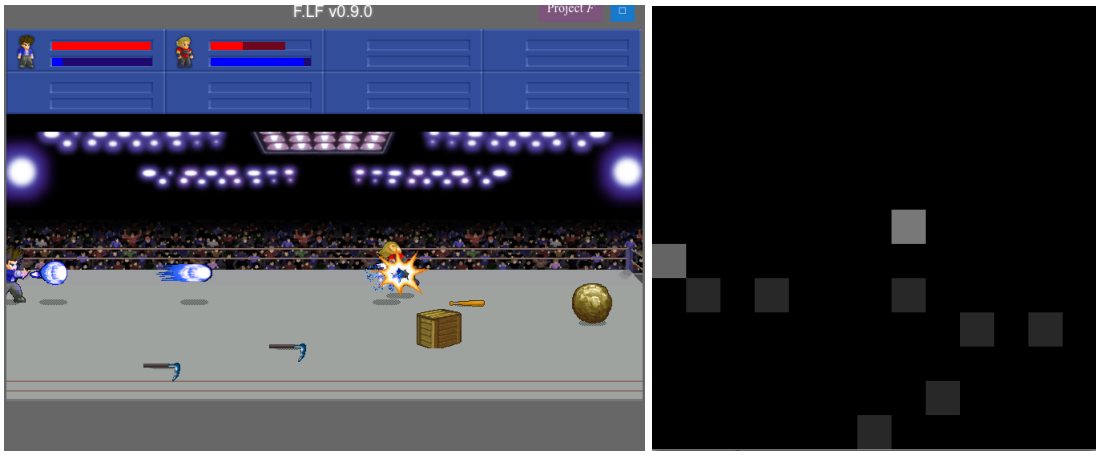


FIGURE 5.4: An illustration of sprite-based map transformation on LF2.

Compare to pixel feature, this feature has the following two inherent limitations:

- Subtle states of the characters are not encoded, the agent cannot know whether a character is attacking, defending, or launching a certain special attack, or facing which direction.
- Many game objects are encoded into the same value, as a result, the agent cannot tell the difference between a rock or a knife. Using more unique values to represent every single kind of game sprite is not a good solution either, since it will make the difference between sprite types blurry.

### 5.2.3 Handcrafted feature

In many RL domains, handcrafted features are used directly. For example, in a recent DeepMind publication [23] that used RL model to solve many Mujoco-based environments, physical states are used directly as the input.
Handcrafted feature is usually a set of compact observations that is considered sufficient for the agent to use to solve the environment. In the humanoid task, a set of complicated features, namely joint angles, angular velocities, velocities, gyroscope readings, torque sensors reading, contact sensors reading, terrain profile, relative position to the target, to name but a few, are used.

FIGURE 5.5: Several Mujoco tasks solved by the recent DeepMind publication.

The motivation of using handcrafted features instead of pixels is to enable the RL agent to focus on the solving the environment, instead of the perception task. This turned out, as we will show in next chapter, can make a huge difference.
In addition, it can provide the agent with extra information that are normally improbable to derive from the screen (e.g. gyroscope reading, torque sensor reading). This might enable the RL agent to show more sophisticated behaviors.
In our task, we encoded the information of each character in the following way:

- 3D coordinates of the character, $(x, y, z)$. [4]
- Velocity of the character, $(v_x, v_y, v_z)$.
- Direction the character is facing, -1 or 1.
- State of character, e.g. attacking, defending, standing, etc.

Afterwards, we will pack the features of different characters into one tuple, where the features of the controlled character will always go first.

### 5.2.4 Feature Scaling

In many machine learning domains, feature scaling is applied before feeding inputs to the model and it is proven to speed up training [24] by improving performance of gradient descent. Since gradient descent is also used in several RL algorithms, we will also apply feature scaling to all three feature types.
For pixel features and sprite-based map, we scale each input pixel to [0, 1] range.
For handcrafted feature, we scale each input to [-1, 1] range.

## 5.3 Reward Shaping

As mentioned in early chapters, reward signal is the only feedback RL algorithm receive from the environment. Reward shaping[5] is of utmost importance for RL algorithm to work well.
An inappropriate reward function will either provide sparse feedback that our agent cannot deduce anything from the interaction, or misguide our agent to perform suboptimal strategies. Given a domain LF2 where the objective is just to win the fight, the challenge is to specify a dense reward function such that the optimal policy is an invariant under this transformation.

[4] Recall that the game scene in LF2 is actually a 3D projection on a 2D plane.
[5] Reward shaping is a method for engineering a reward function in order to provide more frequent feedback on appropriate behaviors.

Imagine if we design the reward function as giving a fixed positive reward every time our agent managed to hit the opponent. While this gives more frequent feedback compared to giving out the reward when the game is finished, an agent will be mislead to hit the opponent with the weakest attack, such that the accumulated number of hits are maximized. Such policy, while being optimal against the proposed reward function, is not something we expect from optimal LF2 policy.
Even if we give a positive reward proportion to the damage done by that attack, it still does not solve all issues. Under this scheme, the optimal agent will hit its opponent, revive the opponent (one LF2 character contains this ability) or wait until opponent's hp regenerate (under LF2 mechanism), and hit it again. By repeating this cycle, the agent can generate infinite reward. It is once again not an optimal policy of the original environment.
Through these examples, one can get a sense of why reward shaping remains to be an open challenge in RL field.
In this section, we will discuss how the current set of reward function is designed and implemented.

### 5.3.1 Basic Reward

In [25], Andrew Ng proposed a form of reward function that resolved some of the mentioned challenges. He first defined potential based reward function as a function that takes in a state transition tuple $(s, a, s')$:

$$F(s, a, s') = \gamma\Phi(s') - \Phi(s),$$

where $\Phi$ is a potential function that takes in a state and return a scalar value, $\gamma$ is the discounted factor of future reward as usual.
He showed that it addressed the positive reward cycle issue, and proved that using such reward formulation is the necessary condition for ensuring policy invariance under reward transformation.
We will design and implement our reward function as such.
For LF2 environment, we define $\Phi(s)$ as a linear combination of the following features:

- *ally_hp*: The summation of health points for characters in same team.
- *enemy_hp*: The summation of health points for characters in other teams.
- *num_ally*: Number of characters in same team that are still alive.
- *num_enemy*: Number of characters in other teams that are still alive.

Potential function $\Phi(s)$ is defined as:

$$\Phi(s) = ally\_hp(s) + k_1 \times num\_ally(s) - enemy\_hp(s) - k_2 \times num\_enemy(s),$$

where $k_1, k_2$ are positive constants.
The reward function $F(s, a, s')$ can be realized as: (assume $\gamma = 1$ for simplicity)

$$\Delta ally\_hp(s, s') + k_1 \times \Delta num\_ally(s, s') - \Delta enemy\_hp(s) - k_2 \times \Delta num\_enemy(s)$$

Under this setting, the agent will be rewarded if enemies' hp are dropping, punished if enemies' hp are reviving. This not only encourages the agent to attack its opponents, but also to finish the match quickly to stop enemies hp from reviving.
Even in cases when enemies are hanging by a thread (with negligible hp), the agent will still be motivated to finish it off to obtain a bonus reward. And when the agent itself is low on hp, the agent will be motivated to stay alive to avoid the extra penalty.

In addition, it is easy to see the reward cycle mentioned in last section can no longer happen [6].

### 5.3.2 Distance Reward

As we will show later in next chapter, simple reward function that only reward goal-relevant information might not be enough for modern RL agent to learn good policy. In fact, such issues are not rare in RL domain and it is common to reward intermediate behaviors, that are not directly related to the goal, to guide the agent to explore the task. Such reward constructs are often referred as exploration reward [26].
In LF2 environment, we decided to include the following term into our potential function $\Phi$:
*dist_enemy*: The summation of Manhattan distance between the controlled agent and the enemies.
$\Phi(s) = ally\_hp(s) + k_1 \times num\_ally(s) - enemy\_hp(s) - k_2 \times num\_enemy(s) - k_3 \times dist\_enemy(s)$, where $k_1, k_2, k_3$ are positive constants.
This formulation encourages the agent to move towards its enemy, which we argued to boost the chance of effective interaction: once the agent learned how to get closer to the enemy, it have more chance to hit the enemy and therefore deduce the game objective.
Note that in order to prevent such reward function to encourage suboptimal behavior (e.g. rushing to an enemy regardless of the situation), $k_3$ is tuned small enough such that it is negligible to the task-relevant reward. Under such setting, after the agent learned the primary game objective, the exploration reward becomes an secondary concern that are unlikely to alter its decision.

### 5.3.3 Reward Clipping

The variance of the reward function were showed to have a huge impact on RL algorithms (including DQN) [27]. In practice, it is considered better to clip the reward into a controlled range.
One typical way [6] to clip the reward is simply map all positive reward to 1 and negative reward to -1. However, such reward clipping will make the agent indifference to the value of different actions in LF2.
More importantly, it triggered the positive reward cycle issue that we were trying to avoid. [7]
To resolve this issue, we decided to use a constant to normalize the reward within reasonable range.

## 5.4 Game Setup

### 5.4.1 Map

Project F provides 10 maps that was adapted from the original LF2 design. For consistent training, we use a fixed map over all experiments. The selected map is called HK Coliseum.
It is the smallest map among all available maps, we decided to use this map to make it easier for the AI to learn as smaller game space increase chance of character interaction.

[6] Assume $\gamma = 1$ for simplicity, $F(s_1, *, s_2) + F(s_2, *, s_1) = \Phi(s_2) - \Phi(s_1) + \Phi(s_1) - \Phi(s_2) = 0$

[7] We discovered that for the distance reward type, such reward clipping lead the agent to learn to repetitively walk slowly towards the enemy and runs backwards. This policy will yield net reward of zero in original scheme but formed a positive reward cycle under such reward clipping.

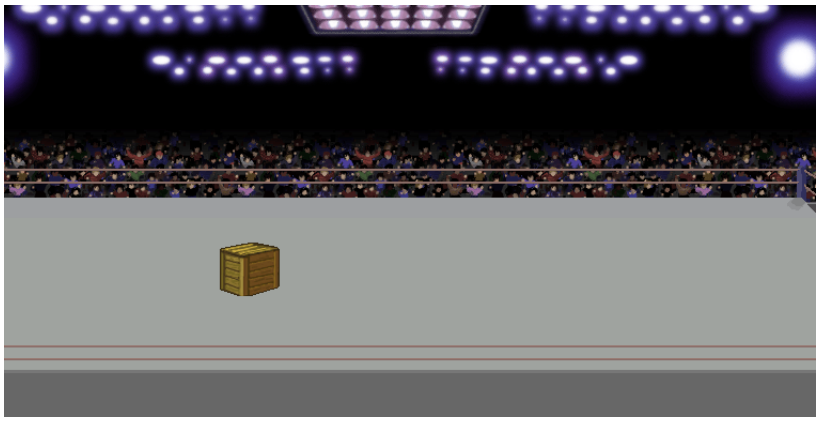

FIGURE 5.6: HK Coliseum – The map we used for most training.

### 5.4.2 Game Mode

LF2 is a multiplayer game that support at most 8 characters in the same match. While the multiagent behavior of AIs is interesting, we only focus on training our RL agent on one versus one game.
That is, during training and test phase, the AI will only encounter one target. The game will end immediately if target or our agent ran out of hp.

### 5.4.3 Render Speed

The original frame rate is fixed to be 30 frame-per-second. However, this frame rate is too low for effective training. Therefore, we tuned up the game frame rate to 250 such that the rendering will not slow down the training [8].

### 5.4.4 Character Selection

Project F provides implementation of 9 LF2 characters.
For consistent training, we used a fixed character selection over all experiments.
For the controlled agent, we always use the Firen character.
For the opponent agent, we always use the Davis character.

FIGURE 5.7: Firen on the left and Davis on the right.

One reason of using distinct characters is to make it easier for our agent to distinguish the opponent from itself.

### 5.4.5 Opponent Type

In our experiments, we will run our RL agent against different kinds of opponents.
As we have mentioned in section 4.2.3, Project F contains three implementations of scripted AI, which we treated as the target in 1v1 game. The in-game AIs have different skill levels, and have a certain playing style.

[8]Note that 250 is not the actual frame rate during training, it is subjected to the processing speed of individual algorithms.

**In-game AI 0**

This is the strongest AI among the three, the AI will launch all kind of special abilities against its opponent.
When it is far away from its target and contain enough mp, it will launch long range attack against its target.

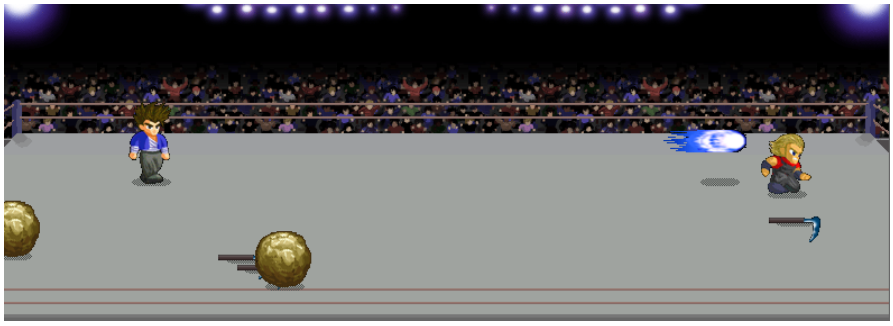

FIGURE 5.8: Davis (in-game AI 0) launched a long distance special attack to Firen.

When it is low on mp or close to its target, it will launch different types of close range attacks. The AI is considered challenging to mid-level human player.

**In-game AI 1**

This AI has a fixed playing style.
It will try to establish a distance from its target until it can make a jump kick towards the target. The jump kick attack can be launched from many different positions, so its target is unlikely to tackle them all. One jump kick is usually powerful enough to bring the enemy to coma mode.

FIGURE 5.9: Davis (in-game AI 1) launching a jump attack to Firen.

In addition, since the AI is always keeping a distance from the target, it is challenging to hit the AI.
Despite the fact that the attack pattern is predictable and it does not exploit any special ability of the player, this AI is still challenging to mid-level human player.

**In-game AI 2**

This AI has a fixed playing style.
When it is high on hp, it will enter aggressive mode. Under this mode the AI will approach its target and launch close range attack.

FIGURE 5.10: Davis (in-game AI 2) is punching Firen.

When it is low on hp, it will enter passive mode. Under this mode the AI will try to move away from its target, waiting the game to restore its hp until it enter aggressive mode again.
Similar to in-game 1, it does not use any special attack. The AI is challenging to low-level human player.

#### Static Agent

Other than in-game AI, we will put our AI against a static agent, which is an agent that will not commit any action.
The position of a static agent is randomized over all possible location by the game engine.

#### RL Agent

To allow the training that uses self-play, and evaluation of different RL agents, we have to make it possible for the opponent agent to be controlled by us as well. To achieve that means, the following changes have to be applied to the default implementation:

- Recall the original Python environment interface, the `step` method is supposed to take in an action of the controlled agent. In this case, both the target and opponent should be controlled by Python program. Therefore that method should take in an action tuple denoting the actions of both agents.
- When the game core receive an action tuple instead of a single action, it will mark the action of the opponent in a buffer that is accessible by any in-game AI script.
- We created a new in-game AI script that commits the action inside the buffer in every frame update call.

### 5.4.6 Operation mode

#### Fixed Mode

In fixed mode, the agent will be repeatedly facing a fixed opponent throughout the whole training process.

#### Random Mode

In random mode, the agent will be facing a randomized opponent from the following pool with uniform probability.

- In-game AI 0

- In-game AI 1
- In-game AI 2
- Static Agent

**Round Robin Mode**

In round robin mode, the agent will be facing an opponent from the above pool in rotation. The purpose of designing such operation mode is to allow our benchmark script to be disturbed at any time (since number of matches against each AI will be uniform at any time).

# Chapter 6

# Result and Analysis

In this chapter, we will explain our experimental results in details.
The experiments are held in three phases, in each phase, we set incrementally harder objective for the AI.

## 6.1 Phase 1: Static Agent Task

### 6.1.1 Description

In this task, we trained our agent against a static opponent that does not perform any action. In earlier phase, we ran our RL agent against some simple in-game AI for training. We discovered that while the agent showed growth during the training progress, the episodic reward [1] becomes misleading. Some learned policy exploited the fact that in-game AIs always approaches its target. It simply kept pressing the attack key and obtained some random hits. By inspecting the agent manually, it could be easily shown that the agent is not responsive to its opponent – it never approached its opponent. In short, the agent either reacts when the enemy is at attack distance, or does not react at all.
To check whether a model is capable of making use of the observation and learning basic game sense, we proposed the static agent baseline. Since the chosen map is reasonable big, it is unlikely the agent can accidentally eliminate the static target without using any observation. Any model that could consistently track the enemy and beat it to death, are clearly having the basic game sense. In contrast, if a model fails to solve this task, it is unlikely to develop high level skills (that must make use of its observation) when training it against harder opponent.
As a result, this task can act as a baseline to tell which model is relatively promising, and allow us to narrow our resources to study them in-depth.

### 6.1.2 NEAT

Due to our past success with NEAT on another domain (Space Impact), it was natural that we conducted experiment with NEAT first. Here we reused the parameter that we previously applied to this domain:

| Parameter | Value |
|---|---|
| Population size | 300 |
| Initial connected portion | 0.5 |
| Number of species | 10 |
| Activation | ReLU |

TABLE 6.1: NEAT parameters used for LF2 training.

[1] The reward obtained in an individual game trial.

We designed the fitness function of each genome to be the average total reward accumulated through three independent game episodes. The reason we need to average the performance over multiple trials is that we want to reduce the factor of luck. Ideally, we should run the game for a larger number (say, 30) of trials for the fitness to bear statistics significance. However, it will slow down the simulation a lot so three is chosen at last.
We ran the following NEAT experiment with different combination of features and reward in table 6.2.

| Setup | A | B | C | D |
|---|---|---|---|---|
| Number of generation | 30 | 30 | 30 | 30 |
| Feature type | Sprite-based Map | Sprite-based Map | Handcrafted | Handcrafted |
| Reward type | Basic | Distance | Basic | Distance |

TABLE 6.2: Design of NEAT experiments.

Based on our prior investigation to NEAT in section 4.1.1, we concluded that NEAT does not perform well with high dimensional input such as pixel feature. This was why we only tested NEAT with these features.
Unfortunately, none of these experiments yielded good result.

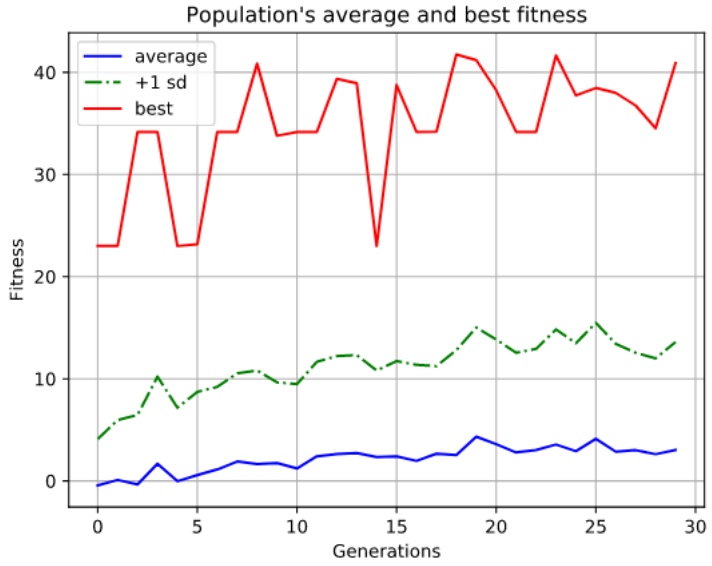


FIGURE 6.1: The maximum, average and standard deviation of fitness for each generation under setup D.

By observing the maximum fitness in figure 6.1, the best genome from each generation seems to be performing reasonably well [2]. When inspecting the resulting agent, all generation champions learned a simple strategy – keep punching forward. This showed that best fitness curve in the diagram is indeed deceiving.
From 300 genome evaluations per generation, there were always some "lucky" genomes that encountered randomized scenes such that both agents stood on the same row for 2-3 consecutive trials. In these cases, the averaged reward of three trials looked good. However, it does not mean that our agent is skillful.

[2]In a game that uses distance reward type, killing the opponent will generate a reward of 50, while the distance exploration reward will typically range from 0 to 10, depending on the initial separation between opponent and agent.

The average fitness curve would be a better metric to see if the generation is improving healthily. However, it can be observed that the average fitness does not show significant sign of improvement since the tenth generation.
To decide whether we should run the experiment longer, we manually inspected 20 sample genomes from the last generation to see whether there were some interesting features developing. This sample should be large enough to contain genomes from multiple species. However, almost all samples were using similar strategy, indicating that even though their underlying neural structures are quite different, they are just evolving towards the same policy.
To improve the results, a naive way would be changing the fitness function to run the average score over more trials and allow the experiment to hold for more generations. However, we were dissatisfied by the data efficiency of NEAT [3] and decided to move on to other branches.
Following are our speculations of why NEAT fail to solve the task, even when given a relatively simple feature representation and dense reward:

- There are many major hyperparameters associated with the NEAT algorithm that we did not touched, namely activation function, number of species, probability of each possible mutation, etc. It is perfectly possible that we did not pick the right parameters for it to learn this task. However, given that our first term goal is to sample various approaches, we cannot investigate too much resources on a single branch.

- One trait about NEAT is that it always searches the solution from the minimal network structure (zero hidden neurons). While this was argued by the original authors to be an advantage of NEAT since it reduce the search space (larger network have more parameters to evolve), this makes NEAT almost impossible to produce complex networks as each mutation of adding connection/node takes several generations to evolve.

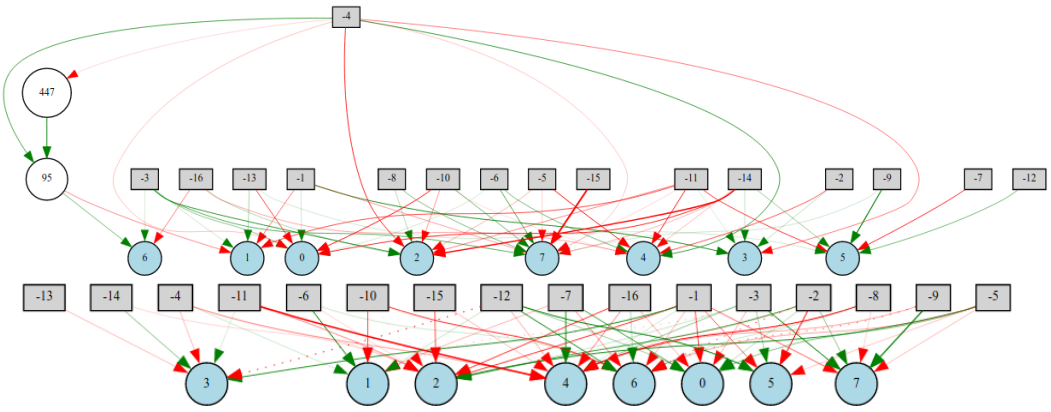


FIGURE 6.2: Two top 30th generation genome sampled from the result of setup D

From 6.3, both genomes from last generation are very simple. We doubted if the optimal solution for static agent task could be expressed by such simple network.

- We later discovered that, NEAT-based method was criticized [28] of relying on finding a deterministic sequence of states that represents a successful exploit. Since the position of characters are randomly initialized in each game, this might explain the poor performance.

[3] In hindsight, about 40M frames were used per experiment under this setup, which is at least 20 times more than how much other methods required to solve the complete task.

### 6.1.3 CNN-based DQN

The DQN agent that utilized convolution neural network has received huge success on the Atari domain using on pixel input. Therefore after the failure of NEAT, we turned to explore such algorithm.

#### Baseline DQN

In this section, we will try to use the DQN implementation provided by OpenAI baselines directly. Other than the classical setup we covered in literature review, it contains the following optimizations:

- Double Q learning
- Prioritized Replay
- Dueling Network

We designed the network setup P as in table 6.3.

| Layer | Num filters | Filter size | Stride | Activation | Size |
|---|---|---|---|---|---|
| conv1 | 32 | 8x8 | 4 | ReLU | |
| conv2 | 64 | 4x4 | 2 | ReLU | |
| conv3 | 64 | 3x3 | 1 | ReLU | |
| fc1 | | | | ReLU | 512 |

TABLE 6.3: Network setup P

The motivation of setup P was to match the setup in the first DQN paper published in Nature. To ensure we had the right setup, before running the algorithm for LF2, we tested the implementation on Atari Breakout and reproduced similar result.
We ran the experiment for two million frames [4].

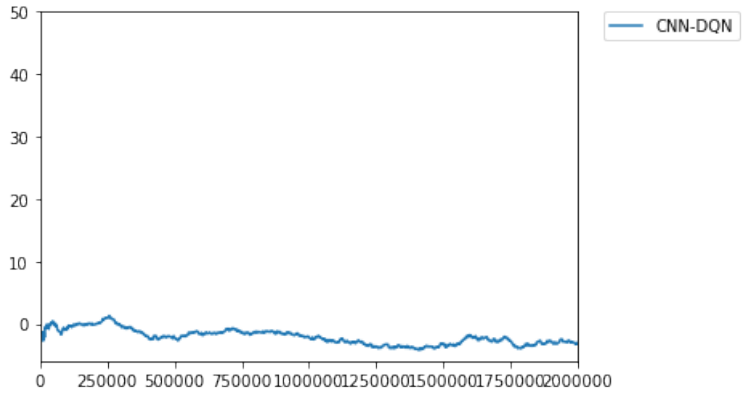


FIGURE 6.3: Mean 100 episode reward of setup P.

However, the result was not satisfying. The eventual reward was roughly around 0 and the training curve was constantly flat.

[4]Note that in many publication, a frame skip of four will be applied to the environment therefore only one fourth of the frame would be effective frames. However in our case, since the frame skip is applied on the game core side, every transmitted frame will be fed into our algorithm.

To ensure the problem was not caused by oversimplified or overcomplicated network architecture, we made modifications to the network structure, and created the following 3 network setups:

| Layer | Num filters | Filter size | Stride | Activation | Size |
|---|---|---|---|---|---|
| conv1 | 32 | 8x8 | 4 | ReLU | |
| conv2 | 64 | 4x4 | 2 | ReLU | |
| conv3 | 64 | 3x3 | 1 | ReLU | |
| fc1 | | | | ReLU | 1024 |

TABLE 6.4: Network setup Q

| Layer | Num filters | Filter size | Stride | Activation | Size |
|---|---|---|---|---|---|
| conv1 | 32 | 8x8 | 4 | ReLU | |
| conv2 | 64 | 4x4 | 2 | ReLU | |
| conv3 | 64 | 3x3 | 1 | ReLU | |
| fc1 | | | | ReLU | 512 |
| fc2 | | | | ReLU | 512 |

TABLE 6.5: Network setup R

| Layer | Num filters | Filter size | Stride | Activation | Size |
|---|---|---|---|---|---|
| conv1 | 16 | 8x8 | 4 | ReLU | |
| conv2 | 32 | 4x4 | 2 | ReLU | |
| fc1 | | | | ReLU | 256 |

TABLE 6.6: Network setup S

| Setup | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Network setup | P | P | Q | Q | R | R | S | S |
| Reward type | Basic | Distance | Basic | Distance | Basic | Distance | Basic | Distance |

TABLE 6.7: Summary of our experimental results, every setup is run for 2M frames.

However, all these experiments produced similar results.
We once argued that the background might be too fuzzy for DQN to identify game objects. To ease the task for our agent, we re-ran setup B with a customized background that is completely black.

FIGURE 6.4: A customized game map.

However, it was proven not to be the cause of the poor performance.

### DQfD

The previous failure inspired us to pursue more complicated setup that have proven to improve DQN. The first candidate we looked at is Deep Q-learning from Demonstrations (DQfD). As mentioned in chapter 2, it modified the DQN algorithm by introducing a pre-training phase when the agent learns from expert transition. During normal training phase, it learns from a hybrid of expert transition and self generated experience. We believed that with the guidance of an expert, DQN might be able to learn reasonable strategy.
To conduct this experiment, we implemented a replay recorder and captured 30000 game transitions. [5] The transitions were extracted from hundreds games played between in-game AI 2 and a static agent. The character of static agent and expert was set to match the training setup such that the expert looked exactly like the RL agent in the game.
For the network setup, we used the network setup P we defined in Baseline DQN.
For the reward type, we used distance reward function in this experiment. [6]
In the pre-training phase, the agent ran 1000 iterations of network update on the expert replay with batch size of 32.
In the training phase, the agent sampled from the self-generated experience replay and expert replay with a ratio of 9:1.
We allowed the experiment to run for 2M frames, however, the result was unexpected.
When the game that was played between in-game AI 2 and a static agent, one could see that the in-game AI will first traverse to the static agent position (behavior A). Afterwards, it would constantly move back and forth around the static agent and punch the agent from both directions (behavior B).
This strategy was partially cloned to our DQfD agent: it was running back and forth and attacking without traversing to the opponent.
We believed that it is caused by the following reasons:

- Behavior A is a relatively short process compare to behavior B. When the transition is sampled from expert replay, the majority of transitions occurred during behavior B. The imbalanced data distribution might explain why it did not learn to trace the opponent.
- In the expert replay, the reward accumulated during behavior A was not comparable with behavior B. This might give a false idea that all credits should be assigned to behavior B.

[5] 30000 is chosen to match the order of magnitude of the expert replay pool size in the original DQfD paper.
[6] The experiment with basic reward function was conducted as well. Yet the result was not included since it was not interesting.

This caused the DQfD agent failing to acknowledge that behavior A is a prerequisite for behavior B to generate any reward.

**DQN with game feature augmentation**

After our failures with baseline DQN and DQfD, we wondered if the fundamental cause was that the features extracted by the convolution neural network filters were not relevant to the game.
We experienced agents which kept punching forward and hoped that it hit the target by chance. Assuming that our CNN filters did not encode any spatial information about the agent and the opponent, this might actually be the only reasonable strategy.
From chapter 2, we reviewed a work which faced a similar problem: the authors tried to apply a variant of DQN on a FPS game – VizDoom, however the trained agent only learn simple strategy that kept firing and expected the enemy to walk into the line of fire. They also attributed the problem to the CNN layers failing to extract information about game feature that is crucial in playing the game, such as the existence of a particular game sprite on the screen. Their solution was to change the network architecture to guide CNN to encode important game feature.
To adapt their solution, we split the output of our last CNN layers to our game feature network and the original MLP layers.
Our game feature network consists of two fully connected layers:

- The first layer consists of 256 ReLU neurons, it is fully connected with the last CNN layers.
- The second layer consists of one output sigmoid neuron, it is fully connected with the first layer. This neuron outputs a number $\in [0, 1]$ signifying whether the enemy is close by the agent (< 200 pixel).

Then we modified our loss function definition in the baseline to:
$J(Q) = J_{DQ}(Q) + \lambda J_G(Q)$, where $J_{DQ}(Q)$ is the original TD-loss of DQN, $J_G(Q)$ is the cross entropy error between the target (whether in that frame, there is enemy close by the agent) and output of the neuron. $\lambda$ is the weighting of the game feature error. In this experiment $\lambda$ is set to be 0.5, to scale the term to match the order of magnitude of $J_{DQ}(Q)$.
Within the first hour [7], the algorithm can already reach an accuracy higher than 85%. After running it for 1.5M frames, the behavior was different from before. It kept jumping around for unknown reason, the mean 100 episodic reward was not improved. In short, we failed to take advantage of this novel architecture that worked for VizDoom.

### 6.1.4 CNN-based ACKTR

After three unsuccessful trials with DQN, we wondered if the game environment was too challenging for DQN. So we researched other novel candidates that are more robust and advanced. This is when we discovered ACKTR[8], an Actor-Critic method.
Before this experiment, we had high expectation on ACKTR. In some literature, it showed dominating results against other state-of-the-art methods, which includes A2C, a method that said to be better than the best DQN setup[9].

[7] After processing 100000 frames.

[8] In this phase, not only ACKTR was explored, but also Advantage Actor Critic (A2C), Proximal Policy Optimization (PPO). However, since their experimental setup and results are quite similar and neither of those explorations were conducted in-depth, we omitted those results.

[9] Setup with Double Q learning, Prioritized Replay, Dueling Network

| Layer | Previous Layer | Num filters | Filter size | Stride | Activation | Size |
|---|---|---|---|---|---|---|
| conv1 | Input | 32 | 8x8 | 4 | ReLU | |
| conv2 | conv1 | 64 | 4x4 | 2 | ReLU | |
| conv3 | conv2 | 32 | 3x3 | 1 | ReLU | |
| fc1 | conv3 | | | | ReLU | 512 |
| policy | fc1 | | | | Linear | 1 |
| value | fc1 | | | | Linear | number of action |

TABLE 6.8: CNN-ACKTR network setup.

We launched the algorithm with 8 parallel agents for a total of 10M frames. However, like all the other attempts, the training curve was flat and the mean episodic score kept oscillating around 0. The resulting agent was not reacting properly to the position of the opponent.

### 6.1.5 MLP-based DQN

After a sequence of failures on using pixel feature, we concluded that in this game, the task of learning the optimal policy along with visual perception was too challenging for modern RL algorithms. Despite we cannot train with raw pixels, there are other feature types that contain much lower dimensional features, such as handcrafted feature.
The first experiment with handcrafted feature was conducted with DQN. To conduct this experiment, we removed the CNN layers from the network and fed the input into the multilayer perceptron (MLP). The rest of the DQN network and the algorithm were not modified.

| Setup | A | B |
|---|---|---|
| Fully connected layers | [128, 128] | [128, 128] |
| Reward type | Basic | Distance |

TABLE 6.9: MLP-DQN experiment setup.

We allowed the experiment to run for 3M frames.

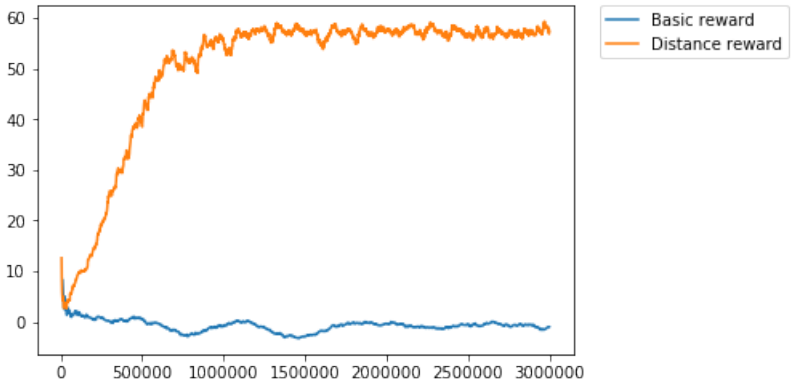


FIGURE 6.5: Mean 100 episodic reward of MLP-based DQN under different reward scheme.

**This was one of our most important experiments, we discovered the first setup (B) that solved the static agent task.** It could locate the agent no matter where it is, and after traversing to the opponent, it would punch it until death. This behavior was not only observed when fighting against a static agent, but also for a human controlled opponent that was consistently

walking away from the RL agent. This showed that our RL agent truly understood the game objective and obtained an optimal policy for the static agent task.
Note that our basic reward function was not suffice for the agent to learn such behavior. This showed that the distance reward function is crucial to provide the initial dense incentive for the agent to explore the game environment, and hence lead it to find the optimal policy. This reward type was therefore used as the default reward type for all subsequent experiments.[10]
To study how different network setup might affect the training progress, we designed the experimental setups in table 6.10:

| Setup | Fully connected layers |
|---|---|
| Double 128 | [128, 128] |
| Double 256 | [256, 256] |
| Double 512 | [512, 512] |
| Triple 256 | [256, 256, 256] |
| Triple 512 | [512, 512, 512] |

TABLE 6.10: Multiple DQN structure used in the static agent experiment.

We ran the above setups with 3M frames each.

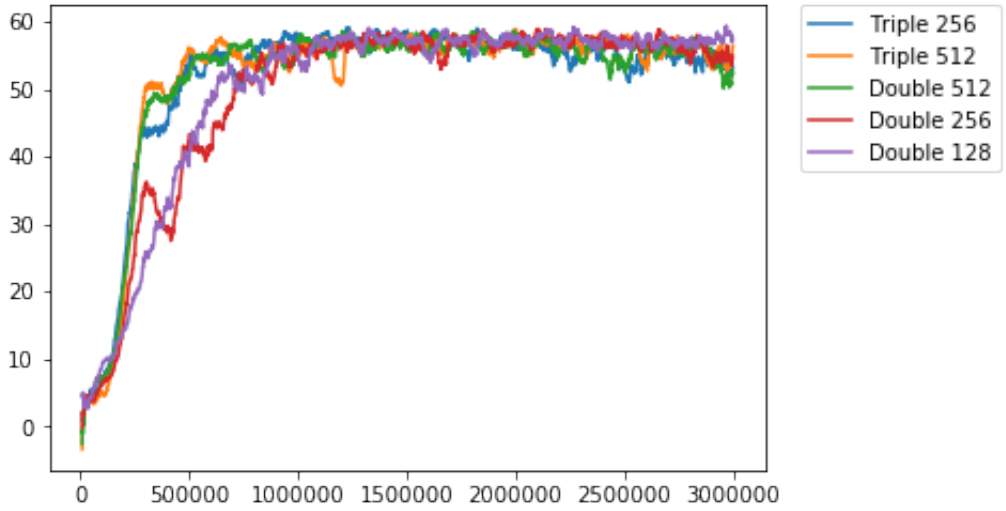


FIGURE 6.6: Mean 100 episodic reward of MLP-based DQN of different structures.

From the figure 6.6, we observed that:

- All network setups were able to solve the static agent task consistently within 1.5M frames.
- The simplest network (Double 128), had the most stable training process and reached the highest final level [11]. The mean reward was steadily increasing throughout the whole process.

[10] As a matter of fact, the setup of distance reward type was proposed in this phase. Knowing that it might significantly improve the learning process of all previously held experiments, we re-ran them all with this reward setup. (As presented in previous sections) However, none of the previous setups (including all CNN-based setup) yielded comparable result. This showed that learning with handcrafted feature is also crucial.

[11] Although the final differences seems negligible on the chart, the reward plotted was actually a moving average over 100 latest game trials. The difference of 5 unit reward indicates a difference of 500 unit reward over 100 trials, which is equivalent to maximal achievable reward in 8 episodes.

- The most complicated network (Triple 256, Triple 512) reached a reasonable level (50) about 0.25M frames earlier than simpler networks. However, their training processes were more unstable and suffered from more hiccups throughout the whole training.

We inspected the resulting agent manually and discovered a minor problem. Even though for most game trials the agent could locate the target at ease. For some positions it struggled to traverse and stuck on a fixed position. This situation happened more frequently for complicated networks, which echoed with our second observation.
This indicated that complex network does not guarantee better performance, sometimes the simplest network can generalize the policy better. We should use the simplest network for the right task as a regularization means.

### 6.1.6 MLP-based ACKTR

After our first success with MLP-based DQN, we wondered how would more advanced method like ACKTR would perform in this task. We modified the original implementation to adapt to the case of MLP-based policy. The MLP network model has the following network structure:

| Layer | Previous Layer | Size |
|---|---|---|
| fc1 | Input | 512 |
| fc2 | fc1 | 512 |
| policy | fc2 | 1 |
| value | fc2 | Number of actions |

TABLE 6.11: ACKTR network structure

We ran the algorithm for 3M frames, with 8 parallel agents.
Following are the mean 100 episodic reward of ACKTR across number of frames it processed, for comparison, the learning curve of some representative DQN setups are plotted as well.

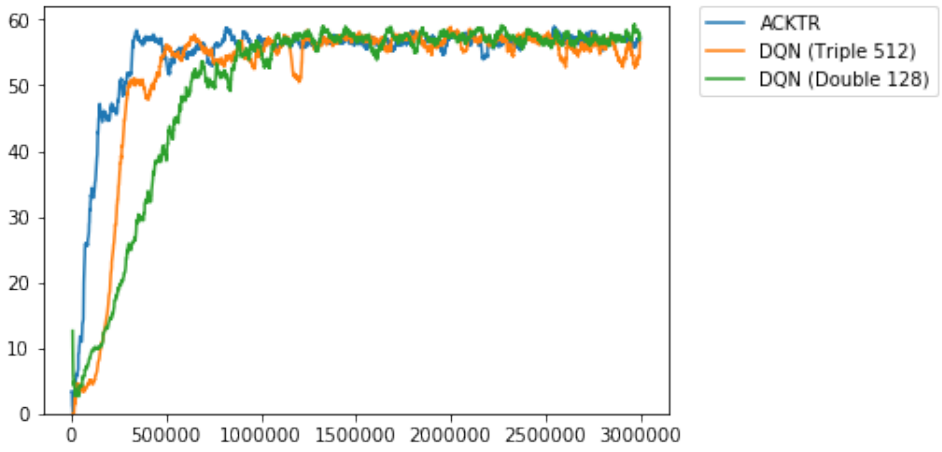


FIGURE 6.7: Mean 100 episodic reward of MLP-based ACKTR compare to MLP-based DQN.

Following is our observations:

- From figure 6.7, ACKTR learned faster than previous DQN setups (DQN Triple 512 was the fastest DQN learner), and demonstrated comparable stability to the most stable DQN setup even when the network structure was more complex. ACKTR (under this task) is

indeed a robust method that provides steady training and better data efficiency, as shown in other literature.

- ACKTR is a parallizable algorithm, by concurrently running 8 agents (using 8 CPU cores), it can gather and process 300000 frames within 20 minutes. We were surprised to see the agent reaching such level under such short time [12].
- The resulting agent did not show the weakness of blind spot we observed from several DQN agents. We attributed it to the fact that DQN is value-based while ACKTR is policy-based.

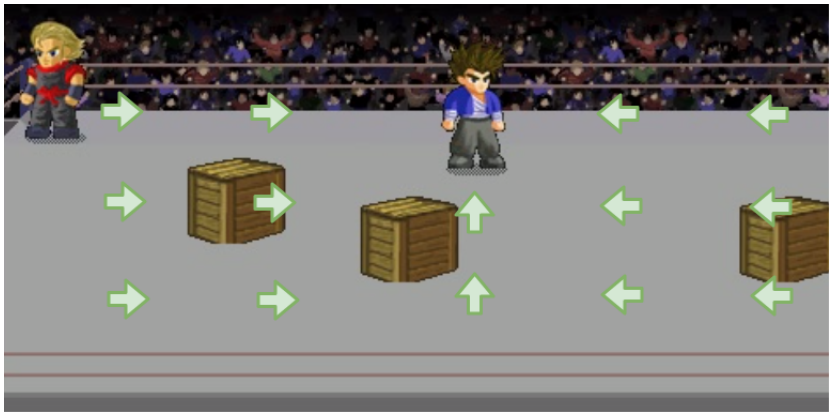

FIGURE 6.8: Green arrows represent the optimal policy when the agent is placed on that position.

From figure 6.8, we knew that if our DQN agent learned the optimal strategy, with random errors that reversed the policy (i.e. right->left) on the current state. It will trap in a nearly deterministic state cycle and stuck forever.

However, for ACKTR, since it learns a probability distribution over any states, the stochasticity in every state makes it easy to escape any similar states cycle.

### 6.1.7 Brief Summary

In phase 1, we explored a wide range of RL algorithms, and combined them with different setups. From these many failures (and some successes), we learned the following points that were helpful for us to narrow down our focus.

- None of our approaches succeeded in learning reasonable strategy by using pixel as features. The only feature format that obtained successes is handcrafted feature type.
- None of our approaches succeeded in using basic reward function, the method of awarding a small reward/penalty for agent reducing/increasing the distance between the opponent proven to dramatically improve training.
- Among all approaches, only MLP-based DQN and ACKTR produced satisfying results. In next phase, these approaches remained to be our focus [13].

[12] All previous attempts took from a few hours to a day to reach good result.

[13] In principle, DQfD and Game Feature Augmentation should be compatible to work with MLP-based DQN. However, compared to policies that are developed with explicit guidance, we were more interested in knowing what policies can be explored by the RL algorithms themselves. Therefore we decided to discontinue our effort on them.

## 6.2 Phase 2: In-game AI

After knowing that several models are capable of solving static agent task under certain setups, we knew that these agents are intelligent in the sense that they understood the game rules and are reactive to the opponent. The next step was to test whether they are also capable of defeating a dynamic target – in-game AIs.
This benchmark is meaningful since these in-game AIs were designed to be challenging, and much domain specific knowledge were hardwired into its logic. On the contrary, our RL models had no zero knowledge about the environment. If our agents are capable of defeating them under an unfair setting, it would be an important achievement.

### 6.2.1 Fixed Opponent Mode

In this section, we explored whether our models can defeat the in-game AI if it is repetitively trained against that AI. In this phase, we expected the agent to exploit every behavior of the opponent as part of the environment. As a result, it might yield policy that is not general. If our models struggle to defeat in-game AI under this setting, there is no meaning of training it under more challenging setting (e.g. Random Opponent Mode). Therefore, this training mode was essential and cannot be skipped. [14]

#### MLP-based DQN

In this part we continued to use the MLP-based DQN that succeeded on the static agent task. Since we discovered that using different network structures on the task could have huge effect, we chose to run the experiments with two different network setups.

- Double 128 layers, [128, 128].
- Triple 256 layers, [256, 256, 256].

As usual, we ran the experiment for 3M frames.

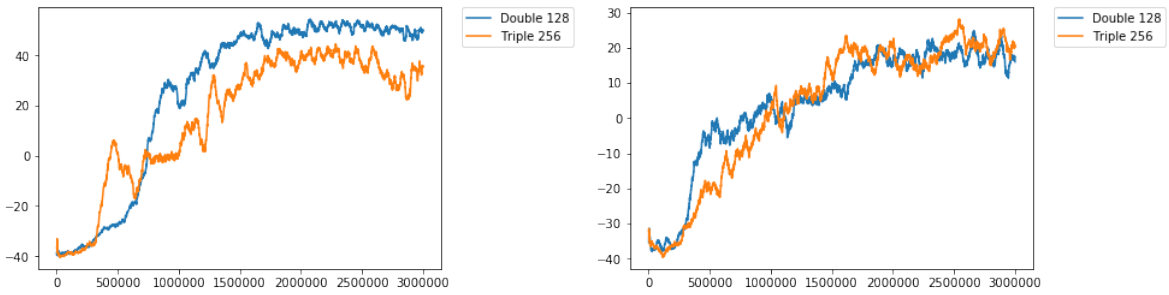


FIGURE 6.9: Mean 100 episodic reward of MLP-based DQN trained against in-game AI 1 and 2.

Win rate:[15]

[14] Notice that as mentioned in previous chapter, Project F provides three in-game AI. However, since we decided on using handcrafted feature as input. All projectile attacks used by in-game AI 0 are invisible to our AI, which makes the training not that meaningful. As a result, we decided not to include those experiments.

[15] The condition of winning is defined as the agent is able to eliminate the opponent under 1500 time steps (a threshold we set before). In case that the game is not finished by then, the enemy with higher hp wins.

| Setup | Network | Training Target | Win Rate (%) |
|---|---|---|---|
| A | Double 128 | In-game AI 1 | 100 |
| B | Triple 256 | In-game AI 1 | 100 |
| C | Double 128 | In-game AI 2 | 85 |
| D | Triple 256 | In-game AI 2 | 75 |

TABLE 6.12: Win rate table.

After this batch of experiments, we concluded the following:

- The agents are capable of learning a dominating strategy against both in-game AIs (under both setups). [16]
- For this task, using simpler networks led to better results in terms of winning rate as well as reward (the agent is able to reach a larger hp difference even with similar win rate).
- By inspecting agents trained against these two AIs, we observed that the policy it learned was indeed highly skillful and did not use trivial strategy exploiting obvious flaws of the AI as we worried. The performance of the agents exceeded our expectation.
- For in-game AI 1, we originally expected this would be a harder AI than in-game AI 2 based on our first hand experience. Recall that this AI will constantly run away from its target and launch jump kick from different directions. However, our agent actually reached a higher win rate and reward against this AI.

  We discovered that it found a strong policy by kept chasing the AI, such that the jump kick attack cannot be launched smoothly since it must be launched from a distance. Then when the opponent stuck in a corner, our agent will block its path and attack it.
- For in-game AI 2, our agent learned a very similar strategy to its opponent, it will try to launch intensive close range attack against its target. The match is balanced until near end-game, when the opponent is trying to getaway to restore its health. Our agent will aggressively track it down and finish it off.
- One trait observed for both agents (more frequently on the agent trained against in-game AI 2) were that, during start game (when the opponents were apart), the agent would get to the same column but different row relative to the opponent and wait the opponent to approach itself. We argued that it is indeed a good tactic because under such Mexican standoff, the passive agent will actually get the upper hand since one cannot launch a punch while moving upward/downward. And in a close fist fight, the side that make the first effective punch gain a huge advantage.

[16] As mentioned before, the agent is blind to any projectile attack. Therefore, we did not include in-game AI 0 for fair comparison.

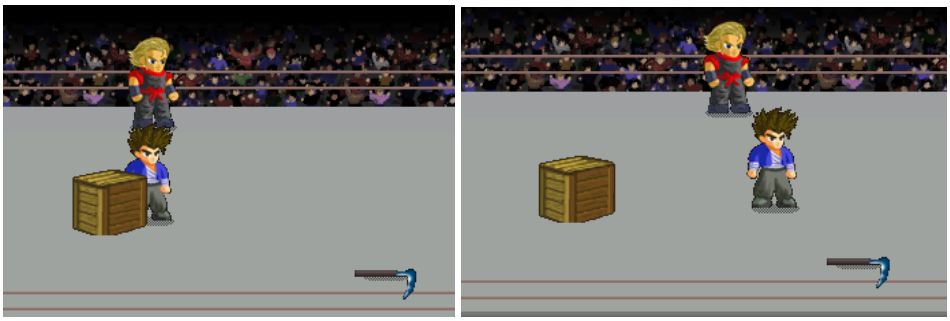

FIGURE 6.10: Agent (red) reaction towards the horizontal movement of its target (blue).

- In most times, both agents showed passive playing style such that it only launch counter attack when it is approached. Such property made it unable to solve the static agent task consistently.
- To further explore how general are the learned policies, we ran more experiment by putting our agents against targets it have not seen during training and recorded their win rate in figure 6.13.

| Network \ Target | In-game AI 1 | In-game AI 2 |
|---|---|---|
| A | 100 | **30** |
| C | **100** | 85 |

TABLE 6.13: Win rate table of DQN agents. Bold numbers represented the unseen setup.

As expected, setup A did not perform very well against an unseen in-game AI 2. However, we were surprised to see that setup C did perform very well against an unseen in-game AI 1.

This result echoed with our hypothesis that, the target used to train against the RL agent can pose a huge effect on its eventual skill level.

**Transfer Learning**
We also designed an experiment that we used the network that solved the static agent task as a base network. Then instead of running DQN on a randomized network, we applied it on the base network.
We reused the Double 128 layers network we trained in static agent task, and ran another DQN trial against the in-game AI 2 for 3M frames: (Setup C – a purely trained DQN in last section are plotted for comparison)

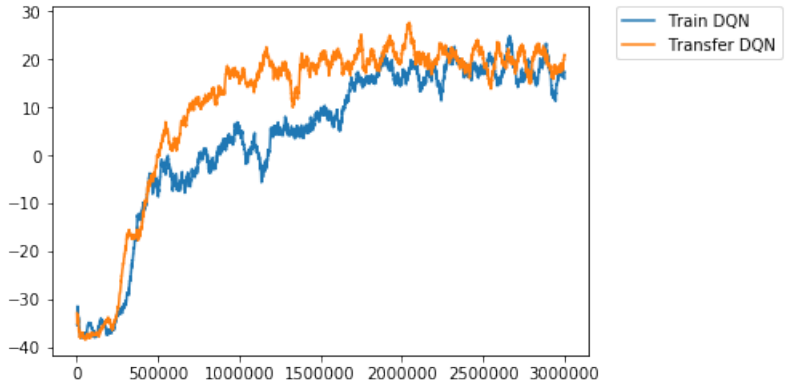


FIGURE 6.11: Transfer learning.

We observed the following interesting results:

- It took setup C an extra 1M frames for it to reach the peak performance of this agent. It showed that although the opponent is different, DQN can quickly adapt its weights to handle another situation. This phenomenon was expected as in the base network, there are some common knowledge that are also helpful in solving the in-game AI task (e.g. how to attack opponent, how to get closer to opponent).

  One implication is that the base network can be used as a means to speed up training.

- Although the base network can consistently solve the static agent task. After 3M frames of training, the agent started to behave passively against a static agent just like the agents we trained last section. Out of curiosity, we ran another identical experiment for only 1M frames. We observed that the behavior of approaching a static agent was still intact (although not as consistent as before). We argued that the behavior was gradually fading as it became less useful in this task, since blindly rushing to the target became dangerous.

  Another perspective is, if we stop the training before its convergence (1M frames), actually we get an agent that solves both the static agent and in-game AI 2 task pretty well.

### MLP-based ACKTR

As usual, we ran the algorithm for 3M frames.

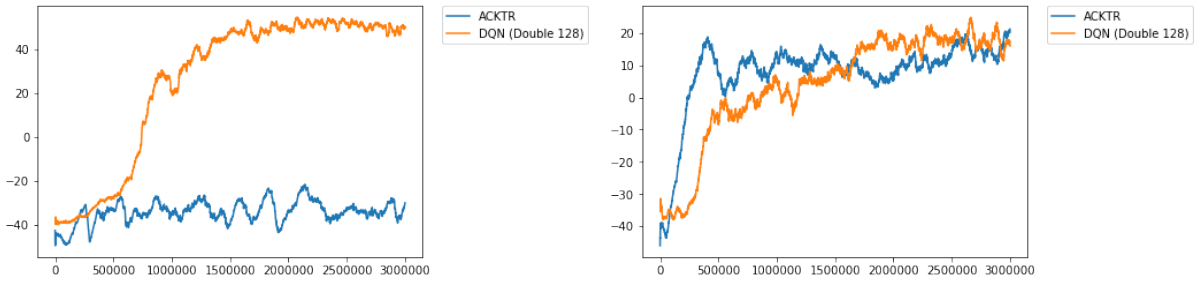


FIGURE 6.12: mean 100 episodic reward of ACKTR against in-game AI 1 and 2.

| Training Target | Win Rate (%) |
|---|---|
| In-game AI 0 | 80 |
| In-game AI 1 | 5 |
| In-game AI 2 | 100 |

TABLE 6.14: ACKTR win rate table against fixed target.

- For in-game AI 1, our ACKTR agent did not learn any reasonable strategy against it. That was an unexpected result so we re-ran the experiment twice. However, the result was still the same. This once again showed that in RL there is no single best algorithm that can dominate other algorithms, performances are often domain specific.
- For in-game AI 2, our ACKTR agent reached a performance that took our best DQN agent an extra 1.5M frames to reach. The final reward level achieved by both agents are similar, however, the ACKTR agent reached a win rate of 100% while DQN reached only 85%.

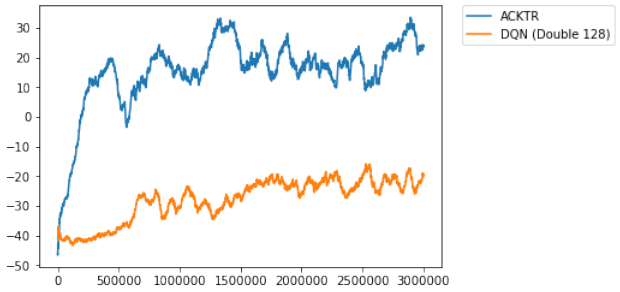


FIGURE 6.13: mean 100 episodic reward of ACKTR against in-game AI 0, comparing to DQN.

The policies discovered by ACKTR and DQN were very different. The ACKTR agent found a policy by keep pressing attack key. In-game AI 2 was designed to approach to the target and launch attack when the target are within a certain range. However, this hardcoded range seems to be shorter than the effective range of a punch. Therefore, if the agent keep punching forward, in-game AI 2 will be disadvantaged. This is clearly exploiting the design flaws of in-game AI 2.

- One surprising finding is that our ACKTR agent learned a highly specific strategy against in-game AI 0 and dominated it, which was not achievable by all other setups. The policy involved running to a corner and repetitively use a fire ball special attack (key sequence: def -> left/right arrow -> attack) to aim at its target. And the in-game AI 0 failed to escape the cycle due to its design flaws.

  When our agent run into the corner, in-game AI 0 is likely to run into the corner and walk to the same row to approach our agent. However, the in-game AI will start punching even they are not in the exact same row. We argued that it is a feature designed to avoid the in-game AI to walk directly within enemy punch range. Supposedly, either parties cannot hit each other with normal punches under such cases. However, the fire ball special attack has a larger attack vertical range than punches, which makes this exploit possible.

FIGURE 6.14: ACKTR specific strategy against in-game AI 0.

This is also the only trial that the agent learned to use a special attack consistently against its enemy.

Unfortunately, the specific policy is useless against other agents.

### 6.2.2 Random Opponent Mode

Even though some of our trained agents in previous section demonstrated good playing skills, some tend to exploit the design flaws of particular in-game AI and developed trivial strategy that does not work generally.
We believed that by running the agent against randomized opponents, in order to obtain good reward all dynamic opponents, the agent will learn more general strategy. Therefore we designed the random opponent mode. As mentioned before, in each episode, all three in-game AIs and the static agent have a uniform probability of being chosen as opponent.

#### MLP-based DQN

As usual, we run the double 128 and triple 256 network for 3M frames:

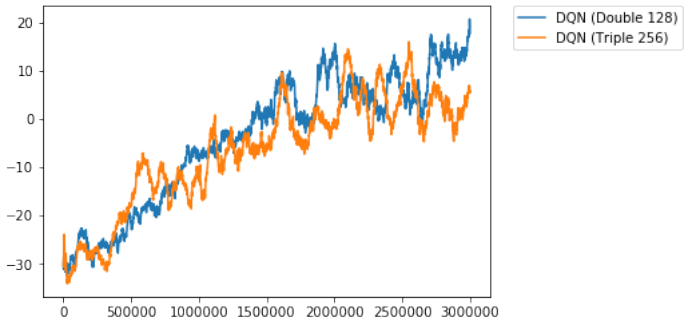


FIGURE 6.15: mean 100 episodic reward of DQN against random opponents.

| Network \ Target | Static Agent | In-game AI 0 | In-game AI 1 | In-game AI 2 |
|---|---|---|---|---|
| Double 128 (Random mode) | 40 | 45 | 90 | 20 |
| Triple 256 (Random mode) | 25 | 5 | 40 | 20 |
| Double 128 (Fixed mode) | 15 | 20 | 100 | 85 |

TABLE 6.15: Win rate table. The last row is agent from last section that is trained against fixed in-game AI 2, added for comparison.

Following are our observations:

- Despite the opponent is constantly changing, both agents managed to learn policy that have average positive reward, denoting that the average hp difference among all run is likely to be positive as well.
- In previous section, it was discovered our DQN agents that were trained against fixed in-game AIs all obtained a passive playing style which makes it unable to approach a static agent. However, we discovered by training it under random opponents, it has a higher probability of approaching a static agent. This is a clear evidence of this training mode is effective in shaping a more general policy.
- In general, the win rate against different targets are lower than those trained against its fixed target. With one exception, the win rate of the agent trained against fixed in-game AI 0 never reached a win rate beyond 20% under DQN. However, the win rate is boosted to 45% when trained against random targets. We argued that when train against a super strong AI, the agent always get eliminated quickly without much time to test its learned policy before an episode end. However, when trained against a random pool of opponents with different levels, the moves can be developed during other runs, which improve its learning.
- By inspecting the game play manually, the strategies are dynamic and non-trivial.
- Simpler network once again yielded better overall result in terms of average reward and win rate.

**MLP-based ACKTR**

We ran the ACKTR algorithm with 3M frames:

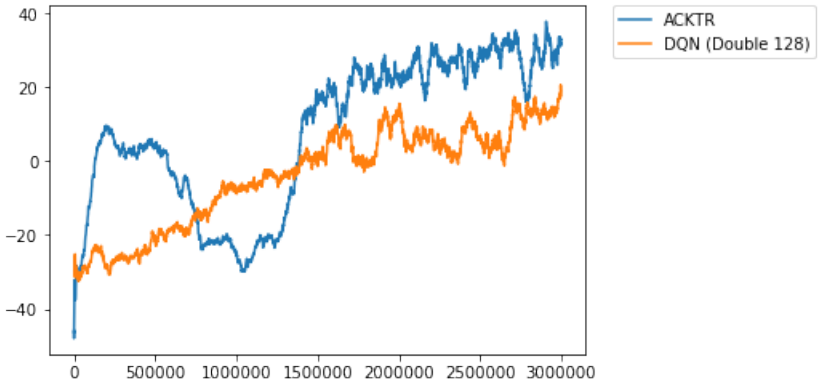


FIGURE 6.16: mean 100 episodic reward of ACKTR against random agent.

| Network \ Target | Static Agent | In-game AI 0 | In-game AI 1 | In-game AI 2 |
|---|---|---|---|---|
| ACKTR (Random mode) | 100 | 15 | 100 | 45 |
| DQN Double 128 (Random mode) | 40 | 45 | 90 | 20 |
| DQN Triple 256 (Random mode) | 25 | 5 | 40 | 20 |

TABLE 6.16: Win rate table.

Following are our observations:

- ACKTR not only learned faster than DQN, but also reached a higher average reward at the end within the given frames.
- ACKTR obtained a win rate that is significantly higher than DQN for all opponents (except in-game AI 0).
- We explained why ACKTR is able to perform well across multiple agents with the following reasons: DQN works by learning Q-function, after training, the policy on a state is mostly deterministic.[17] For a state $s$ in static agent task, the optimal policy is to go forward to the opponent, but for the same state in in-game AIs task, it might be better to do nothing. The deterministic construct of Q-learning do not provide a way to strike a balance between these objectives, which confuses the agent.

  However, since ACKTR learns a policy function that outputs a probability distribution over action spaces. Facing the same dilemma, model can strike a balance of probability to approach the enemy or to stay idle, which eradicate the confusion.

### 6.2.3 Brief Summary

In this section, we conducted experiments and obtained the following results:

- Through training the agent against a fixed opponent, we were able to obtain agents that dominated [18] all three in-game AIs. However, some policies learned are not general against all in-game AI or human players. Under this curriculum, no agents are able to beat all unseen AIs. Also, ACKTR are more likely to learn trivial policies that make use of individual AI's design flaws to win the game.
- We designed an alternative training curriculum, that showed evidence to induce the agent to learn more general strategy.
- In general [19], ACKTR can learn strategy with fewer frames and reach higher final reward and win rate.

## 6.3 Phase 3: Self play

For the approaches mentioned in previous phases, we relied on running on a small pool of predefined opponent AIs. As mentioned in previous chapter, we wondered if such training curriculum will limit the capability of the trained agents.
To mitigate this problem, one solution is to create more in-game AIs, such that the agent will face more different strategies, and force to learn their counter attack and become stronger. However, such approaches are very time consuming and as the agents become stronger, one has to devise more challenging in-game AIs to push it to learn again.
We believed the long term solution for this is to allow the agent to constantly compete with itself, such that as the network evolves, it will continuously face an improved version of itself. This motivated us to design the self play mode, where agent is always competing with the latest version of network.
To experiment self play, the training process has two notable differences:

- Since the agent is learning the control from the perspective of one character (Firen) while the same network requires to control both the agent and opponent. The opponent need

[17] The action during test run is still chosen under $\epsilon$-greedy scheme.
[18] Reach a win rate over 80%.
[19] With the exception of the agent trained against fixed in-game AI 1.

to use the same character (instead of Davis) to make sure the control sequences are valid for both agents.

- Since the agent is competing with an improving self, the average episodic reward will be oscillating around a constant. The reward curve will not be meaningful and will not be shown.

### 6.3.1 MLP-based DQN

We ran the self play curriculum for 3M frames, with multiple network structures:

| Target / Network | Static Agent | In-game AI 0 | In-game AI 1 | In-game AI 2 |
|---|---|---|---|---|
| DQN (Double 128) | 0 | 10 | 75 | 5 |
| DQN (Double 256) | 70 | 5 | 80 | 30 |
| DQN (Triple 256) | 20 | 5 | 95 | 40 |
| DQN (Triple 512) | 5 | 5 | 70 | 60 |

TABLE 6.17: Win rate table.

Following are our observations:

- It first started as completely random agents. Not trained against any in-game AI, it had no idea how normal agent behaves. However, they were still able to win some of the unseen opponents. This showed the potential of self play method.

- In previous DQN attempts, we concluded a general rule that simple networks (Double 128) typically render better agent that are more competent.

  We originally ran this experiment for double 128 layers and triple 256 layers network. However, for the first time, the above rule did not hold. To confirm our findings, we repeated the experiment with Double 256 and Triple 512 network.

  It was discovered that our Double 128 network were actually performing badly in this task. Not only the win rates against in-game AIs were significantly lower, the resulting policies were also problematic. They ran into the corner and waited for the enemy to track it to a beneficial position such that it could corner the opponent and launch punches. [20]

  Compare to other three networks, it did not learn how to track opponent.

- Although we were surprised by the level reached by self play, regarding its training process, we observed a common problem where the agent and its opponent get stuck at some configurations.

  Assume that our network learned that at some point it is better to do nothing under a state and wait for the enemy to reach a better position. Since the opponent is sharing the same network, although the state representation is flipped, it is still possible for that network to make the same decision.

  In such cases, it will cause a deadlock situation. The frames received during this phase will be repetitive and useless.

[20] In the game, normally one can launch 3 to 4 consecutive punches before the enemy fall backwards and enter coma mode. However, in the corner, the character cannot fall backward so the opponent can get more consecutive punches.

### 6.3.2 MLP-based ACKTR

| Network \ Target | Static Agent | In-game AI 0 | In-game AI 1 | In-game AI 2 |
|---|---|---|---|---|
| ACKTR | 85 | 15 | 55 | 70 |
| DQN (Double 128) | 0 | 10 | 75 | 5 |
| DQN (Double 256) | 70 | 5 | 80 | 30 |
| DQN (Triple 256) | 20 | 5 | 95 | 40 |
| DQN (Triple 512) | 5 | 5 | 70 | 60 |

TABLE 6.18: Win rate table.

- ACKTR once again produced better result than DQN, it surpassed the highest win rate achieved by all DQN networks for the static agent and in-game AI 2 targets.
- Out of curiosity, we ran the ACKTR self play agent against all DQN agents trained under same curriculum, and the result shows that ACKTR dominated almost every other DQN agents:

| Network \ Target | DQN (Double 128) | DQN (Double 256) | DQN (Triple 256) | DQN (Triple 512) |
|---|---|---|---|---|
| ACKTR | 90 | 80 | 90 | 50 |

TABLE 6.19: Win rate table.

- Similar to DQN, ACKTR training progress also demonstrated a problem caused by the fact that opponent shared the same network as the agent, at some point they will perform symmetric actions for a long time, which makes the learning process dull and useless afterwards.

  The policy learning will essentially enter an equilibrium state where gradient is zero.

FIGURE 6.17: Symmetric states that ACKTR encountered during training.

# Chapter 7

# Conclusion

## 7.1 Summary of key effort and contributions

- Through trial and errors on three video game environments (Space Impact, Slither.io, LF2), established a set of qualities of ideal RL testbeds and finalized LF2 as the final environment.
- Implemented an efficient, error-tolerance, highly configurable Python interface to allow different agents to interact with LF2 environment at ease.
- Experimented with three modern RL frameworks as well as their variants, and showed that two of them are capable of learning high quality policies.
- Experimented with three feature formats, and showed that only one of them are able to allow agents to learn high quality policies.
- Experimented with two reward shaping function and showed that a small and dense exploration reward (not directly related to true objective) can drastically improved learning without affecting the optimal policies.
- Experimented with different training curriculum against in-game AIs, and showed that for each in-game AI, RL algorithms are able to dominate it without using any domain knowledge. In addition, showed that training against randomized opponent AIs can allow agent learn more general strategy and obtain decent win rate against all in-game AIs.
- Experimented with self play curriculum, and showed that such method can also trained agent that outperformed most in-game AIs, with much potential yet to be optimized.

## 7.2 Future Work

In this section, we provided a list of possible research tracks for the second term.

### 7.2.1 Self play

As mentioned in previous chapter, our current way of self play lead to a few issues during training. We would like to continue explore the possibility of self play learning:

- Instead of always running the agent against the latest network, we can run it against a randomized opponent drawn from the pool of its older versions. We believe it will ensure that the agent to persist features that are developed in early stage but not useful against latest network. In addition, we believe it will resolve the issue we mentioned in last chapter by breaking the symmetry within the policy/value network.

- We could also explore the possibility of hybrid learning from self play and existing in-game AIs to speed up the initial process.

### 7.2.2 Diversify play style

Although our trained agents are highly skillful, they seldom exploit special attacks that takes a specific key sequence to launch. Also, the current handcrafted feature does not encode any information about any incoming special attack.
These factors limited the diversity of strategy it can perform.
Following are few possible future work we could explore:

- Encode more game related features (e.g. hp, mp of all players, game time) into handcrafted features.
- Complexify the current handcrafted rules so that it does not only aware of incoming attack, but also the special attack itself is performing. We hope that by doing so it could induce the agents to discover special attacks combo that are formidable against opponents.
- Make the agent action history a part of the handcrafted features, such that it can discover key sequence for launching particular attacks.
- Experiment the memory-enabled variant of DQN (DRQN) such that the trained agent can adjust its strategy according to its observation to the opponents. We hope that it could make the agent perform differently when faced with different opponents.

### 7.2.3 Online AI evaluation platform

For now the objective metric of the strength of trained agents are the win rate across 4 predefined AIs. However, we have not objectively measured its skill level relative to human.
Therefore, we planned to create an online AI evaluation platform for human users to test against our agents and collect data that is interesting for our research:

- The objective match metrics such as the win rate against human players.
- The actual gameplay recording for us to study how the agents react to human players.
- The human users' subjective opinion on whether the agents are skillful and do they provide a more enjoyable experience.

# Bibliography


[1] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction", 2011.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search", *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge", *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[4] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, *et al.*, "Starcraft ii: A new challenge for reinforcement learning", *ArXiv preprint arXiv:1708.04782*, 2017.

[5] K. O. Stanley and R. Miikkulainen, "Efficient reinforcement learning through evolving neural network topologies", in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, Morgan Kaufmann Publishers Inc., 2002, pp. 569–577.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[7] J. Tsitsiklis and B Van Roy, "An analysis of temporal-difference learning with function approximationtechnical", Report LIDS-P-2322). Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Tech. Rep., 1996.

[8] C. J. Watkins and P. Dayan, "Q-learning", *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[9] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, A. Sendonaris, G. Dulac-Arnold, I. Osband, J. Agapiou, *et al.*, "Learning from demonstrations for real world reinforcement learning", *ArXiv preprint arXiv:1704.03732*, 2017.

[10] G. Lample and D. S. Chaplot, "Playing fps games with deep reinforcement learning.", in *AAAI*, 2017, pp. 2140–2146.

[11] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation", *ArXiv preprint arXiv:1708.05144*, 2017.

[12] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning", in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

[13] H. P. van Hasselt, *Insights in reinforcement learning*. Hado van Hasselt, 2011.

[14] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning.", in *AAAI*, 2016, pp. 2094–2100.

[15] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay", *ArXiv preprint arXiv:1511.05952*, 2015.

[16] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning", *ArXiv preprint arXiv:1511.06581*, 2015.

[17] S. Sidor and J. Schulman. (2017). Openai baselines: Dqn, [Online]. Available: `https://blog.openai.com/openai-baselines-dqn/`.

[18] P. Dhariwal, C. Hesse, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, *Openai baselines*, `https://github.com/openai/baselines`, 2017.

[19] G. Tesauro, "Td-gammon: A self-teaching backgammon program", in *Applications of Neural Networks*, Springer, 1995, pp. 267–285.

[20] B. Sgánetz, *Pixel-perfect clone of nokia's space impact*, `https://github.com/VoidXH/Space-Impact-II`, 2016.

[21] A. Yu, *Hard lessons for creator of little fighter 2 and other would-be hong kong games developers*, `http://www.scmp.com/culture/arts-entertainment/article/1937266/hong-kong-video-game-developers-finally-making-it-bigI`, 2016.

[22] ——, *A clean room implementation of lf2*, `https://github.com/Project-F/F.LF`, 2015.

[23] N. Heess, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, A. Eslami, M. Riedmiller, *et al.*, "Emergence of locomotion behaviours in rich environments", *ArXiv preprint arXiv:1707.02286*, 2017.

[24] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", in *International Conference on Machine Learning*, 2015, pp. 448–456.

[25] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping", in *ICML*, vol. 99, 1999, pp. 278–287.

[26] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch, "Emergent complexity via multi-agent competition", *ArXiv preprint arXiv:1710.03748*, 2017.

[27] T. Zahavy, N. Ben-Zrihem, and S. Mannor, "Graying the black box: Understanding dqns", in *International Conference on Machine Learning*, 2016, pp. 1899–1908.

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning", *ArXiv preprint arXiv:1312.5602*, 2013.