ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

DEEP REINFORCEMENT LEARNING FOR AUTONOMOUS AIR COMBAT UNDER NOISY OBSERVATIONS

M.Sc. THESIS

Ahmet Semih TAŞBAŞ

Department of Computer Engineering

Computer Engineering Programme

JULY 2023

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

DEEP REINFORCEMENT LEARNING FOR AUTONOMOUS AIR COMBAT UNDER NOISY OBSERVATIONS

M.Sc. THESIS

Ahmet Semih TAŞBAŞ
(504201547)

Department of Computer Engineering

Computer Engineering Programme

Thesis Advisor: Assoc. Prof. Nazım Kemal ÜRE

JULY 2023

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

GÜRÜLTÜLÜ GÖZLEM ALTINDA OTONOM HAVA MUHAREBESİ İÇİN DERİN PEKİŞTİRMELİ ÖĞRENME

YÜKSEK LİSANS TEZİ

Ahmet Semih TAŞBAŞ
(504201547)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Assoc. Prof. Nazım Kemal ÜRE

TEMMUZ 2023

Ahmet Semih TAŞBAŞ, a M.Sc. student of ITU Graduate School student ID 504201547 successfully defended the thesis entitled "DEEP REINFORCEMENT LEARNING FOR AUTONOMOUS AIR COMBAT UNDER NOISY OBSERVATIONS", which he/she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Assoc. Prof. Nazım Kemal ÜRE** Istanbul Technical University	.....
Jury Members :	**Prof. Dr. Behçet Uğur TÖREYİN** Istanbul Technical University	.....
	Dr. Veysel YÜCESOY ASELSAN	.....
		.....

Date of Submission : **26 July 2023**
Date of Defense : **17 July 2023**

To my grand father,

FOREWORD

I would like to express my deep gratitude to my advisor, Assoc. Prof. Dr. Nazim Kemal Ure, for his magnificent motivation and efforts throughout my M.Sc study. I would like to thank my colleague, Safa Onur Sahin, for his technical and motivational support in my research.

I would like to thank my family for supporting me throughout my life. Finally, I would like to extend my deepest appreciation to Duygu Kurt for her endless support throughout this study.

July 2023

Ahmet Semih TAŞBAŞ
Computer Engineer

TABLE OF CONTENTS

ABBREVIATIONS

RL	**:** Reinforcement Learning
DRL	**:** Deep Reinforcement Learning
ANN	**:** Artificial Neural Network
MDP	**:** Markov Decision Process
TD	**:** Temporal-Difference
DP	**:** Dynamic Programming
MC	**:** Monte Carlo
DL	**:** Deep Learning
DQN	**:** Deep Q-networks
DDQN	**:** Double Deep Q-networks
PPO	**:** Proximal Policy Optimization
TRPO	**:** Trust Region Policy Optimization
A2C	**:** Advantage Actor-Critic
MCTS	**:** Monte Carlo Tree Search
VI	**:** Value Iteration
PER	**:** Prioritized Experience Replay
IS	**:** Importance Sampling
KL	**:** Kullbeck-Leibler
UA	**:** Unmanned Aircraft
ATA	**:** Antenna Train Angle
AA	**:** Aspect Angle
LOS	**:** Line of Sight

SYMBOLS

t	: Discrete time step
S_t	: State at time t
A_t	: Action at time t
R_t	: Reward at time t
T	: Final time step of an episode
π	: Policy
τ	: Trajectory
$P(s, s', a)$	: State transition model
$R(s, a)$	: Reward transition model
G_t	: Return
γ	: Discount factor
$V_\pi(s)$	: State value function
$Q_\pi(s, a)$	: Action value function
δ	: TD-error
$V^*_\pi(s)$	: Optimal state value function
$Q^*_\pi(s, a)$	: Optimal action value function
α	: Learning rate
ε	: ε-greedy
β	: Prioritization in importance sampling
w_i	: Correction bias in importance sampling
$Z(S_t, A_t)$	: Value distribution
V_{MIN}	: Minimum value of the state-value distribution
V_{MAX}	: Maximum value of the state-value distribution
N_{atoms}	: Number of the atoms
b_{noisy}	: Noisy bias
W_{noisy}	: Noisy weight
$J(\theta)$	: Scalar performance measure
$\nabla J(\theta)$	: Partial derivative of $J(\theta)$ with respect to θ
$\nabla_\theta \log \pi_\theta(s, a)$	: Score function
$B(s)$	: Baseline
$A_\pi(s, a)$	: Advantage function
ψ	: Bank angle of aircraft
Δ_ψ	: Bank angle change of aircraft
v	: Velocity
Δ_v	: Velocity change of aircraft
x	: Aircraft position in x-axis
y	: Aircraft position in y-axis
$\hat{x}$	: Noisy aircraft position in x-axis
$\hat{y}$	: Noisy aircraft position in y-axis
$\hat{\psi}$	: Noisy bank angle of aircraft

LIST OF TABLES

LIST OF FIGURES

Page

DEEP REINFORCEMENT LEARNING FOR AUTONOMOUS AIR COMBAT UNDER NOISY OBSERVATIONS

SUMMARY

Artificial intelligence plays an important role in solving many decision-making and autonomy problems. It is divided into sub-branches: supervised learning, unsupervised learning, reinforcement learning, etc. Reinforcement learning (RL) has recently proven itself as a powerful instrument for solving complex problems. It even surpassed human performance in several challenging applications, such as generating efficient matrix multiplication algorithms, large-scale strategy management in complex games, etc.

The problem of autonomous air combat has been studied for many years. It has many advantages. Pilot training in this field is a long and challenging process. In addition, the performance of the trained pilots is restricted by G-load tolerance and human reflexes. The autonomous driving of aircraft eliminates these problems. However, it is a difficult problem to solve. The complexity of air combat arises from aggressive close-range maneuvers and agile enemy behaviors. Reinforcement learning, which has proven to solve many difficult problems, can be used in autonomous air combat.

In this thesis, we study reinforcement learning methods that can be integrated into situations where observation is perceived as noisy in the autonomous air combat problem. First, we develop an air combat simulation consisting of aircraft dynamics. It enables simultaneous action selection by multiple aircraft and is suitable for developing reinforcement learning algorithms. Then, using this simulation, we compare the performance of the state-of-the-art deep reinforcement learning algorithms, which are Deep Q-learning (DQN), Rainbow, Proximal Policy Optimization (PPO), and Advantage Actor-Critic (A2C).

In previous studies, it is assumed that the air combat environment is noiseless. However, there might be uncertainties in real-life scenarios due to sensor errors, which prevent the estimation of the actual position of the enemy. Therefore, in this thesis, we improve the air combat simulation. The new simulation provides noisy observations to the agents, making the air combat problem even more challenging. In the experiments, we observe that noise in the environment causes the performance decrease of reinforcement learning algorithms in proportion to the noise level. In order to increase the performance, we propose the state stacking method. In our extensive experiments, the proposed method significantly outperforms the baseline algorithms regarding the winning ratio, where the performance improvement is even more pronounced in high noise levels. In addition, we incorporate a self-play scheme into our training process by periodically updating the enemy with a frozen copy of the training agent. In this way, the training agent faces smarter enemy strategies which improve the performance and robustness of the agents. In our simulations, we

experimentally demonstrate that the self-play scheme provides important performance gains compared to the classical RL training.

In the training phase, we train reinforcement learning algorithms at different noise levels, and each policy obtains optimal results at its noise level compared to other methods. However, in the test stage, the agent cannot select a policy among them because it does not know the current noise level. Therefore, we study an artificial neural network-based classifier, which determines the current noise level in the environment. Using air combat simulation, we create datasets with different state stacks and noise levels. Then, we train neural network-based classifiers using these datasets.

Finally, we create an architecture that will cover the entire system. The architecture consists of the environment and the agent. The environment includes aircraft dynamics and adding noise to the state space. The environment receives the actions from the agents, passes them through the aircraft dynamics, and calculates the reward value with the new observation. The agent takes the current state, applies the state stacking method, and sends the output to the noise classifier to detect the noise level. Finally, the proper reinforcement learning policy selects an action and sends it to the environment.

GÜRÜLTÜLÜ GÖZLEM ALTINDA OTONOM HAVA MUHAREBESİ İÇİN DERİN PEKİŞTİRMELİ ÖĞRENME

ÖZET

Yapay zeka bir çok karar verme ve otonomi probleminin çözümünde önemli bir rol oynamaktadır. Yapay zeka, denetimli öğrenme, denetimsiz öğrenme, pekiştirmeli öğrenme (RL) vb. alt dallara ayrılır. Pekiştirmeli öğrenme, son zamanlarda karmaşık sorunları çözmek için güçlü bir araç olduğunu kanıtlamış ve hatta birkaç zorlu uygulamada insan performansını geride bırakmıştır. Bu öğrenme yöntemini diğer yapay zeka yöntemlerinden ayıran en önemli fark verinin elde ediliş biçimidir. Denetimli ve denetimsiz öğrenmede veri seti eğitimden önce oluşturulur ve bu veri seti kullanılarak eğitim gerçekleştirilir. RL'de bunun aksine eğitim aşamasında simülasyondan veya gerçek donanım üzerinden veri elde edilir. Bu sebeple sıralı karar verme ve/veya simülasyona dayalı problemleri çözmek için RL kullanılabilir.

Otonom hava muharebesi problemine uzun yıllardır çalışılmaktadır. Hava muharebesinin otonom olarak gerçekleşmesinin bir çok avantajı vardır. Öncelikle, bu alanda pilot eğitimi uzun ve zorlu bir süreçtir. Buna ek olarak, eğitilen pilotların performansı G toleransı, insan refleksleri vb. koşullara bağlı olarak değişmektedir. Ayrıca otonom uçakların muharebe dışında da bir çok kullanım alanı vardır. Yüzey keşif, elektrik hattı izleme, görüntüleme vb. görevler bu alanlara örnek olarak verilebilir.

Muharebe uçaklarının agresif yakın mesafe manevra gerçekleştirmeleri ve çevik hareket tarzı sergilemeleri ortamın hızlı değişen bir yapıda olmasına sebep olur. Ayrıca bu ortam durağan olmayan bir yapıya sahiptir. Çünkü herhangi bir t anında birden çok uçak ortama aksiyon uygulamaktadır. Bu da şuanki ve bir sonraki durum arasında kurulan bağlantıyı azaltmaktadır. Bu nedenlerden dolayı hava muharebesi, çözülmesi zor bir problemdir. Son yıllarda bir çok sıralı karar verme probleminde başarılı sonuçlar veren pekiştirmeli öğrenme algoritmaları bu problemin çözümünde kullanılabilir. Hava muharebesinde bir çok sıralı karar sınırlı bir geri bildirim altında verilmektedir. Örneğin bir muharebenin kazanma veya kaybetme durumunda, ajan sadece oyun sonucuna göre bir geri bildirim almaktadır. Ara durumlarda uyguladığı aksiyonların değerlendirilmesi zorlaşmaktadır.

Bu tezde, otonom hava muharebesi probleminde, gözlemin gürültülü algılandığı durumlara entegre olabilecek pekiştirmeli öğrenme yöntemleri üzerine çalışılmıştır.

İlk olarak, pekiştirmeli öğrenme algoritmalarını geliştirmeye uygun, uçak dinamiklerinin olduğu ve aynı anda birden fazla uçağın aksiyon alabildiği bir hava muharebesi simülasyonu geliştirilmiştir. Bu simülasyon, uçakların birbirlerine göre konum ve durum bilgilerini hesaplayarak pekiştirmeli öğrenme algoritmalarının girdi olarak

kullanacağı gözlem bilgisini sağlar. Uçaklar bu gözleme göre hızını ve burun açısını değiştirerek düşman uçağa karşı avantajlı konuma geçmeye çalışır.

Son yıllarda öne çıkan Deep Q-learning (DQN), Rainbow, Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C) gibi ayrık aksiyon uzayında çalışabilen derin pekiştirmeli öğrenme algoritmalarının performansı, oluşturulan simülasyonda karşılaştırılmıştır. Gerçekleştirilen deneylerde DQN algoritmasının diğer algoritmalara göre daha iyi sonuçlar elde ettiği görülmüştür.

Bundan önce gerçekleştirilen çalışmalarda muharebe ortamının gürültüsüz olduğu varsayılmıştır. Fakat gerçek bir hava muharebesinde çeşitli nedenlerle ortamda gürültü oluşabilir ve düşman uçağın gerçek konumu tam olarak hesaplanamayabilir. Bu tezde, bu varsayımı ortadan kaldırmak ve problemin karmaşıklığını arttırmak için rakip uçağın konumuna bir dağılımdan bağımsız örneklemler alınarak gürültü eklenmiştir. Böylece eğitilen uçak, düşman uçağın gerçek konumunu bilemez ve onun konumunu ayrık bir şekilde algılar. Gerçekleştirilen deneylerde, ortamda gürültünün olması, pekiştirmeli öğrenme algoritmalarının performansının gürültü seviyesi ile orantılı olarak düşmesine neden olduğu görülmüştür. Bu performansı arttırmak için durum istifleme yöntemi önerilmiştir. Bu yöntem ile yüksek gürültü seviyesine sahip ortamlarda referans ajana kıyasla %138'e kadar performans artışı elde edildiği yapılan deneylerde gösterilmiştir.

Bu tezde odaklanılan bir diğer nokta ise düşman ajanın stratejisini geliştirme üzerine olmuştur. Çok ajanlı pekiştirmeli öğrenme problemlerinde, eğitilen ajanın performansı rakip ajanlara bağlı olmaktadır. Çünkü rakibin daha karmaşık hamleler uygulaması eğitilen ajanı zorlar ve daha iyi stratejiler geliştirmesini sağlar. Bundan önceki çalışmalarda kullanılan rakip ajan stratejilerinin aksine, kendi kendine oynama yöntemi, hava muharebesi probleminde uygulanmıştır. Belirli zaman periyotlarında düşman uçağın stratejisi, o ana kadar eğitilmiş olan ajanın stratejisi ile değiştirilmiştir. Gerçekleştirilen simülasyonlarda, kendi kendine oynama yöntemi ile eğitilen ajanın, bu yöntem ile eğitilmeyen ajana karşı %88'lik kazanma oranı elde ettiği görülmüştür. Ayrıca, kendi kendine oynama yöntemi eğitim aşamasında iki farklı avantaj sağlamıştır. İlk olarak, düşman politikası için kural tabanlı bir ajan yazma zorunluluğu ortadan kalkmıştır. İkinci olarak, düşman ajanın politikası belirli periyotlarda güncellendiği için eğitilen ajan basitten zora doğru bir eğitim stratejisi doğal olarak ortaya çıkmıştır.

Bu bölüme kadar oluşturulan yapıda, eğitim aşamasında, pekiştirmeli öğrenme politikaları farklı gürültü seviyelerinde eğitilmiştir ve her bir politika kendi gürültü seviyesinde diğer modellere kıyasla daha iyi bir sonuç ortaya koymaktadır. Fakat test ortamında, ajan içinde bulunduğu gürültü seviyesini bilmediği için bu politikalar arasında bir seçim gerçekleştiremez. Bu sebeple bu tezde çalışılan bir diğer konu ise yapay sinir ağı tabanlı gürültü sınıflandırıcı olmuştur. Hava muharebesi simülasyonu kullanılarak farklı istif sayılarında ve farklı gürültü seviyelerinde veri setleri oluşturulmuştur. Bu veri setleri oluşturulurken dikkat edilen bir nokta, pekiştirmeli öğrenme politikaları ve sınıflandırıcıların girdilerinin aynı olmasıdır. Böylece pekiştirmeli öğrenme döngüsü değiştirilmeden sınıflandırıcı bu döngüye entegre edilebilmiştir. Bu veri setleri kullanılarak yapay sinir ağı tabanlı sınıflandırıcılar

eğitilmiştir. Eğitim sonucunda istif sayısının yüksek olduğu sınıflandırıcılar yaklaşık %90'lık bir validasyon başarı oranı sağlamıştır.

Son olarak, oluşturulan sistemi uçtan uca kapsayacak bir mimari oluşturulmuştur. Bu mimari çevre ve ajandan oluşmaktadır. Çevre bölümünde, uçak dinamikleri ve durum uzayına gürültü eklenmesi bulunmaktadır. Çevre ajanlardan gelen aksiyonları alır, uçak dinamiklerinden geçirir ve yeni gözlem ile ödül değerini hesaplar. Son olarak, mevcut gözleme gürültü ekler, gürültülü gözlem ve ödül değerlerini ajana gönderir. Ajan kısmında ise, durum istifleme, gürültü sınıflandırma ve farklı gürültü seviyelerinde eğitilmiş pekiştirmeli öğrenme politikaları bulunmaktadır. Ajan, çevreden gelen durumları alır ve bunları belirli zaman aralıklarında istifler. Bu istifleri çevredeki gürültü oranını belirlemek için gürültü sınıflandırıcıya iletir. Gürültü sınıflandırıcı, çevrenin gürültü seviyesine karar verir ve aksiyon almak üzere gürültü seviyesine uygun olan pekiştirmeli öğrenme politikasını seçer.

1. INTRODUCTION

1.1 Motivation of the Study

Modern aircraft such as F-35 and F-16 are produced in agile structures that can make sudden maneuvers to avoid enemy aircraft. The military pilots experience a long and tough training process to use these aircraft. Although they show a decent maneuvering performance on these aircraft, their performance is inherently restricted by the human reflexes and G-load tolerance.

On the other hand, the autonomous control of the unmanned aircraft (UA) can enable to utilize the full maneuvering capacity of these aircraft and also prevent the possible damages to the human pilots [1]. In addition, UA has many civilian use cases [2] such as surface reconnaissance, power-line monitoring, law enforcement, etc.

As stated by Yang et al. [3], the design of an autonomously maneuvering aircraft is a highly challenging problem since these aircraft fight each other with aggressive maneuvers at close range. In addition, unpredictable actions of the enemy aircraft increases the problem complexity. As a remedy, the deep learning methods and the optimization algorithms can be used to tackle this problem.

As one of the main branches of artificial intelligence, the RL techniques demonstrate superior results in several decision-making problems. For example, trajectory generation in drone racing [4], large-scale strategy management in Dota 2 game [5], generating efficient matrix multiplication algorithms (Alpha Tensor) [6], and auto curricula training in contentious multi-agent environment [7] are the specific illustrative RL based solutions. These examples prove that RL can be used in various domains.

Similar to these problems, the air combat problem requires sequentially taking reasonable actions with only a limited feedback from the environment such as the

win/loss of the fight, therefore, the RL based solutions are highly suitable for this problem.

This thesis is based on the [8]. We used and extended these ideas.

1.2 Literature Review

The air combat problem is extensively studied in the literature.

McGrew et al. [1] present a method based on approximate dynamic programming. They pointed out that the success of their approach depends on feature development, reward shaping, and trajectory sampling.

Isci et al. [9] emphasize that it is important to protect the energy of the aircraft along with defeating the enemy aircraft in air combat. They define an energy condition for aircraft and penalized the agent in reward function based on the energy consumption.

Unlike the other studies, Yang et al. [3] define the probability of shooting the enemy aircraft as being directly proportional to the proximity of the enemy. In addition, they also divide the training phase into two parts. In the first part, the enemy takes simple actions. In the second part, the agent plays against a more sophisticated enemy with a stochastic policy.

Pope et al. [10] employ a hierarchical reinforcement learning methodology. They develop their approach on DARPA's AlphaDogFight Trials environment. This simulator provides low-level controller input. Therefore, they separate the control structure into different layers and trained different agents for specific tasks.

Hu et al. [11] propose an autonomous maneuver model based on deep reinforcement learning algorithms. They design rules based on expert knowledge and use them in the training phase. Thus, by reducing the search space with rule-based actions, they reduce the time that the agent loses by taking random action.

Another issue emphasized in this thesis is the strategy of the enemy agent. In previous studies, the strategy of the enemy agent was generally chosen as follows: predefined maneuvers [3,9,10,12], Minimax [13,14] and heuristic search algorithms (Monte Carlo Tree Search) [13].

Tasbas et al. [15] present a method based on Double Deep Q-networks for 2D air combat problem. They focus on how the enemy agent affects the training performance. They demonstrate that the trained agent can achieve higher returns when the enemy policy is more sophisticated.

1.3 Contributions

In this thesis, we implement basic aircraft dynamics and define state and action space for reinforcement learning algorithms. In previous studies, discrete reward functions are designed [1,13] according to winning or losing an episode. In this thesis, on the contrary, we design a continuous reward function and provide its pseudo-code We also compare state-of-the-art deep reinforcement learning algorithms in the air combat environment, including Deep Q-networks [16], Rainbow [17], Proximal Policy Optimization [18], and Advantage Actor-Critic [19].

Previous works focus on different fields in air combat. They implement air combat environments and propose different approaches for high-level strategy management. In all of these studies, the environment is assumed to be noise-free, i.e., the exact position of the enemy aircraft is known. However, this may not be achievable in a real-life scenario. For example, the sensor errors or medium conditions may cause the observation to differ from the reality.

In this thesis, contrary to previous studies, we develop a simulation that provides noisy observations to the agents. We design the noise characteristic to be adjustable through its parameters. Hence, we can evaluate the performance of the trained agents under a wide range of noise levels.

Since the agent perceives its enemy as dispersed, the air combat problem becomes more complex. Thus, the RL algorithm's performance decreases as the noise level increases. We propose a state-stacking technique, which reduces the noise and increases overall performance.

Contrary to previous studies, we aim to create a smarter enemy behavior using competitive self-play techniques. Therefore, the agent encounters smarter enemy strategies, and the problem gets harder progressively during training. We create

different self-play structures to demonstrate the effect of the enemy difficulty performance of the training agent. Furthermore, we empirically show that the agent trained with self-play scheme outperforms the agent trained without self-play in a noisy test environment.

We train different reinforcement learning policies in different noise-level environments. However, the environment's noise level is unknown in the test stage. Therefore, we propose a classifier that measures the noise level in the environment and leads to selecting the right RL policy for the current noise.

Finally, we summarize the overall system in an end-to-end architecture. This architecture consists of two sub-systems: agent and environment. The environment takes action and returns the reward and noisy states. The agent stacks states, sends them to the noise classifier, and then selects actions from an RL policy based on the environment's noise.

1.4 The Outline of the Thesis

The rest of the organization of the thesis is as follows.

In Chapter 2, we introduce the background of the reinforcement learning concept, including tabular solution methods (Dynamic Programming, Monte Carlo, Temporal Difference Learning), value-based DRL algorithms (Deep Q-networks, Rainbow), policy gradients, and actor-critic based DRL algorithms (Advantage Actor-Critic, Proximal Policy Optimization).

In Chapter 3, we explain the aircraft dynamics, air combat geometry, simulation environment settings (state and action space), and reward design of the dog fight. We also share baseline results of the experiments based on deep reinforcement learning algorithms.

In Chapter 4, we explain how we generate a noisy state in an air combat environment and reduce noise using state stacking. We also clarify the enemy strategy in air combat (including self-play) and the results of the proposed methods.

In Chapter 5, we design a noise classifier based on an artificial neural network. This classifier takes stacked observation as input and predicts the environment's noise level. We also summarize the overall system in an end-to-end noisy air combat architecture.

Finally, Chapter 6 concludes the thesis with final remarks.

2. BACKGROUND

2.1 Reinforcement Learning

Reinforcement learning is a sub-branch of machine learning that maps situations to actions in order to maximize reward signal. The learner, the agent, is not provided with direct information about which action gives more reward in which situation. Instead, the agent tries to increase the reward signal by discovering the environment through trial and error.

Reinforcement learning differs in many aspects from supervised learning and unsupervised learning, which are other sub-branches of machine learning. Firstly, there is no supervisor in reinforcement learning unlike supervised learning. The only feedback that the agent gets is the reward signal. Secondly, almost all of the machine learning algorithms based on independent and identically distributed samples. On the other hand, in reinforcement learning, the data distribution changes over time during the training phase. Finally, the agent is in a sequential decision-making process. Hence, it's actions affects the subsequent data distribution it obtains [20].

2.2 Key elements of reinforcement learning

There are two main characters in reinforcement learning: agent and environment. The environment represents a problem that has to be solved by an agent. The agent is the decision maker or learner algorithm. It interacts with the environment and takes actions according to the states it is in. After this action, the environment returns a new state and a reward signal that the agent has to maximize over time.

This agent-environment interaction loop is demonstrated in Figure 2.1. In this figure, there are some concepts which belongs to reinforcement learning terminology: state S_t, reward R_t, action A_t.

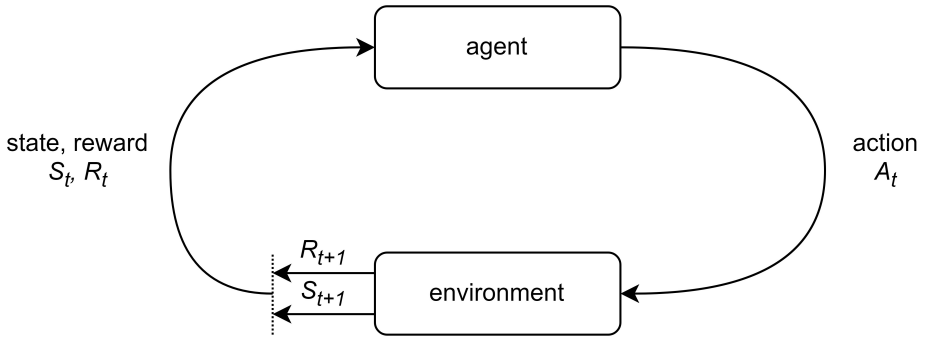


Figure 2.1 : The agent-environment interaction in reinforcement learning.

A state is a description of the current situation. It is represented as a vector, matrix or higher-order tensor. For chess bot that learns to play chess game, the state is the positions of the chess pieces in the board. Another example would be gaming bot that plays arcade games via visual observation. The state of the bot is the RGB matrix of the game pixels.

A reward is a scalar feedback signal that indicates how well the agent make decisions in the environment. The main objective of the reinforcement learning agent is discovering the environment and improving its policy in order to maximize cumulative reward.

An action is the interaction method of the agent. A set of actions create the action space of the environment. There are two different action spaces: discrete and continuous. Discrete action space is limited with a finite number of moves. Some environments, like Atari [21], use discrete action space, where the actions are *"move right", "move left", "fire",* etc. In continuous action spaces, actions are real values. For instance, Mujoco Simulator [22] use continuous action space, where the actions are the inputs of robot joints.

Formally, in Figure 2.1, at each time step t, the state S_t defines the current state of the environment. The agent selects an action A_t according to the current state. One time step later, it receives a reward R_{t+1} and next state S_{t+1}. This interaction continues until the final step of an episode T in a finite environment. This interaction creates a trajectory τ:

$$\tau = S_0, A_0, R_1, S_1, A_1, ..., A_{T-1}, S_T, R_T, \tag{2.1}$$

where S_0 is the initial state of the environment.

Policy

Policy π is a mapping function that determines the behavior of the agent in the environment. It decides what action to take in the current state. There are two types policies in reinforcement learning: deterministic and stochastic. Deterministic policy selects action with 1 probability in a certain state:

$$a = \pi(s). \tag{2.2}$$

On the other hand, output of a stochastic policy is an action distribution. Each action has a probability and they sampled with a randomness:

$$\pi(a|s) = \mathbb{P}\left[A_t = a|S_t = s\right]. \tag{2.3}$$

2.3 Markov decision process

Almost all of the reinforcement learning problems are defined as Markov Decision Process (MDP). It formulates the interaction between the agent and the environment mathematically. Formally, MDP is a tuple $\langle S, A, P, R, \gamma \rangle$, where S is the state space, A is the action space, P is the state transition model:

$$P^a_{ss'} = P(s, s', a) = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a], \tag{2.4}$$

R is the reward model:

$$R^a_s = R(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a], \tag{2.5}$$

and $\gamma \in [0, 1]$ is the discount factor [23].

2.4 Value Functions

Although reward signal demonstrates the immediate feedback, it does not provide information about future rewards. Therefore, a discounted reward called *return* G_t is calculated:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.6}$$

where discount factor $\gamma \in [0, 1]$ emphasis on future reward.

Value function is the expected return of a state s. It estimates how good the agent to be in a given state. There are two different value functions: state value function (2.7) and action value function (2.8). State value function is the expected return starting from state s and following policy π:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right]. \tag{2.7}$$

Action value function is the expected return starting from state s, taking action a, and following policy π:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right]. \tag{2.8}$$

The optimal value function gives the maximum return:

$$V^*(s) = \max_\pi V_\pi(s), \tag{2.9}$$

$$Q^*(s,a) = \max_\pi Q_\pi(s,a). \tag{2.10}$$

following optimal policy π_*.

2.5 Bellman Equation

Value function has a recursive structure. It can be decomposed into immediate reward R_{t+1} and discounted value of next state $\gamma V(S_{t+1})$. The obtained equation is called as *Bellman Equation*. Bellman equation in terms of the state value function is:

$$\begin{aligned} V(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1}, +\gamma R_{t+2} + \gamma^2 R_{t+3}...|S_t = s] \\ &= \mathbb{E}[R_{t+1}, +(\gamma R_{t+2} + \gamma^2 R_{t+3}...)|S_t = s] \\ &= \mathbb{E}[R_{t+1}, +\gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1}, +\gamma V(S_{t+1})|S_t = s]. \end{aligned} \tag{2.11}$$

Bellman Equation in terms of the action value function is derived similar to equation (2.11) as:

$$Q(s,a) = \mathbb{E}[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})|S_t = s, A_t = a]. \tag{2.12}$$

The Bellman equation of the optimal action value function follows the intuition: if the optimal value $Q^*(s',a')$ is known under next states s' and actions a', then the optimal strategy is to select a', which maximizes expected value of $R(s,a)+\gamma Q^*(s',a')$.

The Bellman equations of the optimal value functions are:

$$V^*(s) = \max_a \mathop{\mathbb{E}}_{s'\sim P}[R(s,a)+\gamma V^*(s')], \tag{2.13}$$

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s'\sim P}[R(s,a)+\gamma \max_{a'} Q^*(s',a')], \tag{2.14}$$

where s' is the next state, a' is the next action.

2.6 Dynamic Programming

DP uses the bellman equations to evaluate value functions, when the model of the environment is fully known. One of the common approach of the DP is Value Iteration. It is a version of Bellman optimality equation (2.14) with an update rule. It performs policy evaluation for each state $s \in S$ until convergence [20]:

$$V(s) = \max_a \left(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a V(s') \right). \tag{2.15}$$

After value updates, we obtain a policy $\pi \approx \pi^*$, which acts greedily:

$$\pi(s) = \operatorname*{argmax}_a \left(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a V(s') \right). \tag{2.16}$$

The main drawback of DP is that it needs full knowledge of transition dynamics P and reward dynamics R. Therefore, it is a *model-based* algorithm.

2.7 Monte Carlo

Monte Carlo methods learn from experience without using model of the environment. Therefore, it is a *model-free* algorithm. The main idea of MC methods is to use empirical mean to calculate value function. Therefore, MC learns from the complete episodes: $S_1,A_1,R_2,S_2,A_2,...,S_T,R_T$ to compute return $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$. For this reason, all episodes must terminate eventually. The overall MC prediction is given in Algorithm 1 [20]. As in DP, after value updates, policy act greedily for control.

Algorithm 1 First-visit MC prediction

A policy π, $V(s)$ for all $s \in S$, $Returns(s)$ for all $s \in S$.

For each episode:
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, ..., R_T$
    $G \leftarrow 0$
    For each time step of episode, $t = T-1, T-2, ...0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless S_t appears in $S_0, S_1, ..., S_{t-1}$:
            Append G to $Returns(S_t)$
            $V(S_t) \leftarrow$ avg($Returns(S_t)$)

2.8 Temporal Difference Learning

Temporal difference (TD) is one of the central idea of the reinforcement learning. It combines the advantages of Dynamic Programming (DP) and Monte Carlo (MC) methods [20].

TD learning uses bootstrapping to update estimates of values like DP. Thus, it does not wait for a final outcome like MC. It is also a model-free algorithm. Therefore, it does not use the model of the state and reward dynamics. We arrange these three method's backup functions:

$$V(S_t) \leftarrow \mathbb{E}_\pi[R_{t+1} + \gamma V(S_{t+1})], \tag{2.17}$$

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)), \tag{2.18}$$

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)), \tag{2.19}$$

where (2.17) is DP backup function, (2.18) is MC backup function and (2.19) is TD backup function. TD learning is similar to MC method, however, it replaces G_t with dynamic programming's bootstrapping update $R_{t+1} + \gamma V(S_{t+1})$. This results in a quantity called the *TD Error*:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{2.20}$$

TD Error is the difference between the current estimate $V(S_t)$ and it's better estimate $R_{t+1} + \gamma V(S_{t+1})$. The agent updates its value functions with a better estimate and optimize its policy in each time step. Notice that TD error uses next reward R_{t+1} and next state S_{t+1}. Therefore, it is necessary to take a step for the update.

Q-learning is one of the RL algorithm, which uses TD updates as a value function estimator. It does not use the model of the environment and learn from samples, so it is model-free.

The overall Q-learning algorithm is given in Algorithm 2 [20]. In this algorithm, firstly, all the $Q(s,a)$ values are initialized arbitrarily for each state and action pairs.

In each episode, environment provides initial state S. Then, each step of episode, algorithm chooses an action A from Q policy using state S. In practice, all actions are not taken from the Q policy to ensure exploration of new states. Instead, it is selected from Q policy with $1-\varepsilon$ probability and random policy with ε probability. This method is called ε-greedy. Note that ε-greedy approach is not compulsory for Q-learning. It is just an exploration technique.

Environment takes selected action A and returns reward R and new state S'. $Q(s,a)$ updates its value using TD error. Then, next state S' is assigned as current state S. This loop continuous until the terminal state S.

Algorithm 2 Q-learning (off-policy TD control)

Initialize $Q(s,a), \forall s \in S, \forall a \in A(s)$ arbitrarily.

For each episode:
    Initialize S
    For each time step of episode:
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        $Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max_a Q(s',a) - Q(s,a)]$
        $S \leftarrow S'$
    until S is terminal

Q-learning is an off-policy algorithm. In other words, the value function updates and the collected transition samples come from different policies.

2.9 Deep Q-Networks (DQN)

One of the sub-branch of machine learning is deep learning, which uses artificial neural networks. DL has an extensive usage in many fields: computer vision [24,25], natural language processing [26] etc. These methods use different ANN architectures,

including convolutional neural networks, and recurrent neural networks to use in both supervised and unsupervised problems.

Mnih et al. [27] use the advantageous aspects of deep learning in reinforcement learning, and compose a deep reinforcement learning algorithm called *Deep Q-Networks*. Their main objective is to train control policies based on convolutional neural networks from raw image data in sophisticated RL environments. They combine Q-learning algorithm with stochastic gradient descent optimizer, and train Atari policies.

Deep learning has several challenges from the RL perspective. Firstly, deep learning algorithms need a large amount of labeled data. The input of the network and target output is explicit. RL algorithms, on the other hand, learn from scalar, delayed, and noisy reward signals. The algorithms meet with a reward thousands of time steps later. This makes it difficult to establish the relationship between intermediate steps and rewards. Secondly, DL algorithms suppose the data samples are independent. RL algorithms, on the other hand, encounter highly correlated states. Finally, DL assumes the data distribution is the same overall as the training. However, data distribution changes over time in RL environments.

2.10 Rainbow

After DQN algorithm is published, many extensions are offered to improve this algorithm. Hessel et al. [17] combined six extensions in one structure and called it the Rainbow Algorithm:

- Double Q-learning,
- Prioritized replay,
- Dueling networks,
- Multi-step learning,
- Distributional RL,
- Noisy nets.

Algorithm 3 Deep Q-learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$

For episode=1, M:
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For t=1, T:
        With probability ε select a random action a_t
        otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect
        to the network parameters θ
        Every C steps reset $\hat{Q} = Q$
    End For
End For

Double Q-learning

The first idea is the Double Q-learning [28]. The DQN algorithm has an overestimation bias problem because of using the max term in temporal difference updates. The DQN optimizes the neural networks by minimizing the following loss:

$$(R_{t+1} + \gamma_{t+1} \max_{a'} q_{\overline{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t))^2, \tag{2.21}$$

where the θ is the online network parameters and the $\overline{\theta}$ is the target network parameters. The online network selects actions and is updated through back-propagation. The target network, on the other hand, is not directly optimized, it's parameters are updated every n step by the online network. Target network use increases the stability of the training.

Double Q-learning uses two different networks to decouple the max term. The first network selects actions, and the second network estimates the value of these actions, using the loss:

$$(R_{t+1} + \gamma_{t+1} q_{\overline{\theta}}(S_{t+1}, \arg\max_{a'} q_\theta(S_{t+1}, a')) - q_\theta(S_t, A_t))^2. \tag{2.22}$$

This change decreases overestimations of the Q values and improves performance [17,28].

Prioritized experience replay

Another fruitful extension is Prioritized Experience Replay (PER) [29]. Deep Q-learning samples data from the replay buffer uniformly. Thus, each sample has an equal selection probability in the training phase. Prioritized experience replay, on the other hand, assigns a probability p_t for each transition. The main idea is the select valuable transitions more frequently.

The selection probability p_t is calculated by absolute temporal difference error:

$$p_t \propto |R_{t+1} + \gamma_{t+1} \max_{a'} q_{\bar{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t)|^\omega, \tag{2.23}$$

where ω is the importance exponent, which adjusts the probability of selecting a transition.

The prioritized experience replay generates bias because it changes the sampling distribution. Therefore, the authors propose prioritization importance sampling weights to correct bias:

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta, \tag{2.24}$$

where β is the prioritization importance sampling parameter. β is between 0 and 1. With $\beta = 1$, the sampling bias is fully compensated, and the distribution is uniform. Lastly, this method inserts new transitions to the replay buffer with the highest probability to increase new samples selection probability.

Dueling networks

Another improvement on the DQN algorithm is Dueling Networks [30]. The dueling network decomposes the Q function $Q(s,a)$ into two quantities: the state-value function $V(s)$ and the state-dependent action advantage function $A(s,a)$. The state-value function $V(s)$ is the expected return of a state s. The action advantage function is a delta value, which demonstrates how much extra reward is obtained by selecting a particular action. The action advantage value could be positive, negative or zero.

The main idea of the using dueling architecture is to show that it is unnecessary to estimate the value of each action in every state, e.g., in the Atari game Enduro [30], the agent should select an action only when it encounters an obstacle. If there is no obstacle, then Q values should be close to the state-value in this scenario.

The network architecture of action values [17]:

$$A_\theta(s,a) = V_\eta(f_\xi(s)) + \left(A_\psi(f_\xi(s),a) - \frac{\sum_{a'} A_\psi(f_\xi(s),a')}{N_{actions}} \right), \tag{2.25}$$

where ξ is the parameters of the shared encoder f_ξ, η is the parameters of the value stream f_η, ψ is the parameters of the advantage stream f_ψ, and the overall network is $\theta = (\xi, \eta, \psi)$. To keep stability of the optimization, the mean of the advantage values are subtracted from the each advantage value.

Multi-step learning

Vanilla Q-learning uses one step learning, i.e., it receives a single reward and select maximum Q-value at time t:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t) \right]. \tag{2.26}$$

Sutton et al. [31] propose forward-view n-step targets, which leads faster learning. In this method, we define n-step return:

$$R_t^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t^k R_{t+k+1}, \tag{2.27}$$

and multi-step DQN loss:

$$(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} Q_{\overline{\theta}}(S_{t+n},a') - Q_\theta(S_t,A_t))^2. \tag{2.28}$$

Distributional reinforcement learning

Q-learning uses the expected return as a value function. Distributional reinforcement learning [32], on the other hand, proposes a value distribution instead of the expected return, i.e., it replaces action-value Q with a value distribution Z:

$$Z(S_t,A_t) = R(S_{t+1},A_{t+1}) + \gamma Z(S_{t+1},A_{t+1}). \tag{2.29}$$

and the expectation of the Z is the Q:

$$Q(S,A) = \mathbb{E}Z(S,A). \tag{2.30}$$

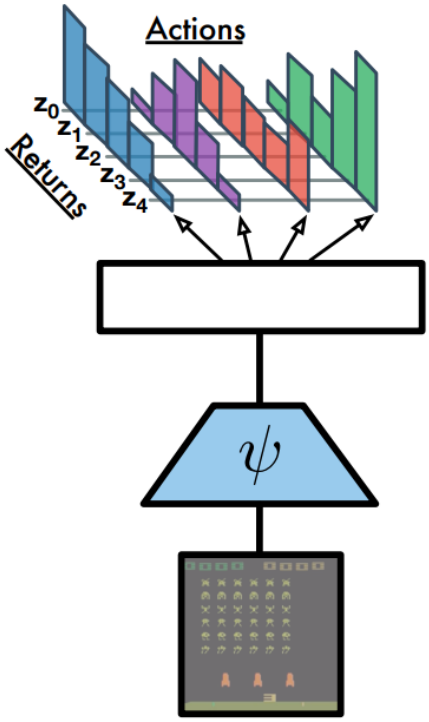


Figure 2.2 : Distributional reinforcement learning network architecture [33].

Bellemare et al. [32] model value distribution d_t using a discrete distribution. The distribution d_t at time t is defined with a probability mass $p_\theta(s_t, a_t)$ and a support z, i.e., $d_t = (z, p_\theta(s_t, a_t))$. The main goal is to bring close the estimated distribution to real distribution by changing network parameter θ.

The support z is the set of atoms:

$$z_i = V_{MIN} + (i-1)\Delta z, \Delta z := \frac{V_{MAX} - V_{MIN}}{N_{atoms} - 1}, \tag{2.31}$$

where $i \in \{1, ..., N_{atoms}\}$ and z_i is the i^{th} atom, , $N \in \mathbb{N}^+$ is the number of the atoms, V_{MIN} and V_{MAX} refer to the minimum and maximum values of the state-value distribution, respectively. Figure 2.2 demonstrates an example of the distributional reinforcement learning network architecture. In this figure, there are four different actions, and each action has five atoms.

And the probability mass function p is:

$$p_i(s,a) = \frac{e^{\theta_i(s,a)}}{\sum_j e^{\theta_j(s,a)}}. \tag{2.32}$$

The training of the probability masses uses a similar approach to the Bellman's update. The distribution d_t is bring close to the target distribution d'_t using a statistical distance

measurement, e.g., Kullbeck-Leibler divergence. The target distribution d'_t is:

$$d'_t \equiv (R_{t+1} + \gamma_{t+1} z, p_{\overline{\theta}}(S_{t+1}, \overline{a}^*_{t+1})), \tag{2.33}$$

where R_{t+1} is the reward, which shifts the value distribution, z is the fixed support, and $\overline{a}^*_{t+1}$ is the greedy action in state S_{t+1}.

Greedy action selection consists of two steps:

$$Q(s_{t+1}, a) := \sum_i z_i p_i(s_{t+1}, a), \tag{2.34}$$

$$a* \leftarrow \arg\max_a Q(s_{t+1}, a), \tag{2.35}$$

where the first step is the calculating return distribution's sum for each Q value, and the second step is the selecting Q value greedily.

Finally, the loss is:

$$D_{KL}(\Phi_z d'_t || d_t), \tag{2.36}$$

where, Φ_z is the L2-projection of the target distribution onto z.

Noisy nets

Last but not least, the noisy net is another improvement over deep Q learning. DQN algorithm uses ε greedy approach for exploration. At the beginning of the training, it takes random actions, then it decays ε, and the agent selects the best action by looking at current information. Fortunato et al. [34], on the other hand, propose another approach for the exploration problem, Noisy Nets.

Noisy Nets add noise to the neural network's weights and biases in the training stage. It leads to discovering unseen states and action regions. The standard linear layer is defined as, $y = b + Wx$, where b is the bias, W is the weights, and x is the input of the layer. The noisy linear layer combines the linear layer and a random variable from Gaussian noise as,

$$y = (b + Wx) + (b_{noisy} \odot \varepsilon^b + (W_{noisy} \odot \varepsilon^w)x), \tag{2.37}$$

where, b_{noisy} is the noisy bias, W_{noisy} is the noisy weight, ε^b and ε^w are random variables, and $\odot$ is the element-wise product.

2.11 Policy Gradients

Value-based methods learn an action-value function and select actions according to the method. So they do not have an explicit policy without action-value estimates. Policy gradient methods, on the other hand, learn a parameterized policy that selects actions without using value functions.

Fundamentally, policy gradient methods learn a policy based on the gradient of a scalar performance measure $J(\theta)$. These methods update parameters to maximize performance using gradient ascend:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}, \tag{2.38}$$

where $\nabla J(\theta)$ is the partial derivative of $J(\theta)$ with respect to θ. All methods that follow this approach are called policy gradient methods.

The notation of probability of action a, in state s with parameter θ at time t is:

$$\pi(a|s,\theta) = Pr\{A_t = a|S_t = s, \theta_t = \theta\}. \tag{2.39}$$

The Policy Gradient Theorem

A natural question is how we can estimate the performance gradient with respect to the policy parameter. The policy gradient theorem has a theoretical answer to this question. Sutton et al. prove the policy gradient theorem using elementary calculus and re-arranging terms for the episodic case in [20].

Before going into the details of the theorem, we describe the score function. Because the underlying idea of the policy gradient theory uses this funtion. Score function describes the direction in which we should change the parameter θ to maximize the likelihood. Suppose a policy π_θ is differentiable, and we have the information of the gradient of the policy $\nabla_\theta \pi_\theta(s,a)$. Rearranging it with $\pi_\theta(s,a)$ and using the log trick derivative we obtain:

$$\begin{aligned} \nabla_\theta \pi_\theta(s,a) &= \pi_\theta(s,a)\frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} \\ &= \pi_\theta(s,a)\nabla_\theta \log \pi_\theta(s,a), \end{aligned} \tag{2.40}$$

where $\nabla_\theta \log \pi_\theta(s,a)$ is the score function. And it describes the direction of the optimization.

Back to the policy gradient theorem, consider a one-step MDP, performance measure $J(\theta)$ is equal to the expectation of the reward r under policy π_θ:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta}[r] \\ &= \sum_s d(s) \sum_a \pi_\theta(s,a) \mathcal{R}_{s,a}, \end{aligned} \tag{2.41}$$

where r is the one time-step reward, $d(s)$ is the distribution over states. And the derivative of the performance measure J_θ is $\nabla_\theta J(\theta)$,

$$\begin{aligned} \nabla_\theta J(\theta) &= \sum_s d(s) \sum_a \nabla_\theta \pi_\theta(s,a) \mathcal{R}_{s,a} \\ &= \sum_s d(s) \sum_a \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) \mathcal{R}_{s,a} \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) r], \end{aligned} \tag{2.42}$$

where $\nabla_\theta \log \pi_\theta(s,a)$ is the score, and r is the instantaneous reward. In the first line of the equation, we arrange $\nabla_\theta \pi_\theta(s,a)$ with $\pi_\theta(s,a)$ as in Equation 2.40. After that, we obtain the second line of the equation using log trick derivative. Finally, taking an expectation over states and actions results in the final line of the equation. The score $\nabla_\theta \log \pi_\theta(s,a)$ tells us how we should change the optimization parameters (weights for neural networks) in order to increase or decrease the log probability of selecting action a_t at state s_t. The reward r is the scoring function, if it is high it will rise the probabilities of the state-action combination, otherwise, if it is low it will descend the probabilities of the state-action combinations. So, the score and the reward works in cooperation to obtain better policy.

Reward r is one-step return, but we can replace it with long-term value $Q^\pi(s,a)$ in order to make equation to multi-step MDP:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)], \tag{2.43}$$

REINFORCE: Monte Carlo Policy Gradient

REINFORCE is an implementation of the policy gradient theorem as an algorithm. It updates parameters by stochastic gradient descent. And it uses return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t,a_t)$:

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t,a_t) v_t. \tag{2.44}$$

Algorithm 4 REINFORCE: Monte-Carlo Policy Gradient

Initialize θ arbitrarily

for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do:
    for $t = 1$ to $T-1$ do:
        $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
end for

The overall algorithm of the REINFORCE is given in Algorithm 4 [23].

2.12 Actor-Critic Methods

REINFORCE has a high variance problem, which slows down learning. One idea to reduce variance is to use actor-critic architecture. The actor is the policy, which takes action in the environment. And it updates its parameters by policy gradients. The critic, on the other hand, does not select any action. It estimates the action-value function $Q(s,a)$ and sends it to the actor. In other words, the actor picks the actions, and the critic evaluates these actions.

Formally, the critic-network parameters w are updated by $TD(0)$:

$$w \leftarrow w + \beta \delta \phi(s,a), \tag{2.45}$$

where δ is the TD-error:

$$\delta = r + \gamma Q_w(s', a') - Q_w(s,a). \tag{2.46}$$

The actor-network parameters $|\theta$ are updated by policy gradient:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a|s) Q_w(s,a). \tag{2.47}$$

Notice that $Q_w(s,a)$ is parameterized by the critic-network weights w, so its just a number, and its derivatives is equal to 0 with respect to actor-network parameters θ.

2.13 Advantage Actor-Critic (A2C)

Another approach to reducing variance in policy gradient methods is subtracting a baseline function from the scoring function:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a|s)(Q_w(s,a) - B(s)), \tag{2.48}$$

where $B(s)$ is a baseline. And it may decrease variance without changing the expectation, because the subtracted value is zero [20]:

$$\sum_a B(s)\nabla\pi(a|s,\theta) = B(s)\nabla\sum_a \pi(a|s,|\theta) = B(s)\nabla 1 = 0. \tag{2.49}$$

A natural and good baseline can be state-value function $B(s) = V^{\pi_\theta}(s)$. So, we obtain advantage function:

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s). \tag{2.50}$$

And we can update policy gradient using the advantage function:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) A^{\pi_\theta}(s,a)\right]. \tag{2.51}$$

We need to estimate both state-value $V_v(s)$, and action-value $Q_w(s,a)$ to calculate advantage function. There is more than one way to do this.

For example, *Advantage Actor-Critic (A2C)* algorithm [19] uses a shared network to estimate both $Q_w(s,a)$ and $V_v(s)$. The TD error δ^{π_θ} is:

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s), \tag{2.52}$$

is an unbiased estimate of the advantage function:

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\delta^{\pi_\theta}|s,a\right] &= \mathbb{E}_{\pi_\theta}\left[r + \gamma V^{\pi_\theta}|s,a\right] - V^{\pi_\theta}(s,a) \\ &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\ &= A^{\pi_\theta}(s,a)\end{aligned} \tag{2.53}$$

So we can update the policy gradient with the TD-error:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\delta^{\pi_\theta}\right]. \tag{2.54}$$

2.14 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [18] is another algorithm based on the policy gradient theorem. Vanilla policy gradients are sample inefficient and have stability problems. In order to solve these problems, Schulman et al. propose an algorithm called Trust Region Policy Optimization [35]. However, this algorithm is complicated and incompatible with some architectures like parameter sharing. PPO simplifies the

TRPO algorithm without losing performance (experimentally). It introduces a simpler, more general, and more compatible algorithm.

Trust Region Methods

Policy gradient methods have a training stability problem. One reason for this problem is differences in policy updates. If we have large policy updates, we may face the "off the cliff" problem. Moreover, we may not recover the old policy again. In addition, smaller policy updates provide more stable optimization, and we may obtain an optimal solution more likely.

In this context, TRPO maximizes an objective function (surrogate objective):

$$\text{maximize } \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right], \tag{2.55}$$

where $r_t(\theta)$ is the ratio function:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}. \tag{2.56}$$

If $r_t(\theta) > 1$, the probability of taking action a_t at state s_t is more likely in current policy π_θ than old policy $\pi_{\theta_{old}}$. Otherwise, if $0 < r_t(\theta) < 1$, the probability of taking action a_t at state s_t is less likely in current policy π_θ than old policy $\pi_{\theta_{old}}$.

TRPO uses a hard constraint to ensure that the new policy is not far from the old policy. The constraint is:

$$\hat{\mathbb{E}}_t \left[KL \left[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t) \right] \right] \leq \delta, \tag{2.57}$$

where, KL is the KL-divergence.

Clipped Surrogate Objective

The TRPO objective $\hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right]$ may lead to excessively large policy updates. Therefore, it uses a constraint as seen in Equation 2.57. However, this objective function with constraint leads to the second-order derivative. Therefore, PPO algorithm simplifies the TRPO's objective, and uses the following objective:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t) \right], \tag{2.58}$$

where epsilon ε is a hyper-parameter. With this objective, it is guaranteed that the current policy does not far from the previous policy, and this update requires only first-order derivation.

If the policy and the value function networks share network parameters, then we have to define a loss function that combines the objectives of these functions. PPO combines these terms as follows:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right], \tag{2.59}$$

where c_1, c_2 are coefficients, L_t^{VF} is value function loss $(V_\theta(s_t) - V_t^{\text{targ}})^2$ and S is the entropy bonus.

3. AUTONOMOUS MANEUVER DECISION BASED ON REINFORCEMENT LEARNING

3.1 Aircraft Dynamics

We design and construct a simulator, which models the motion dynamics of the aircraft and performs a close-range one-to-one air combat scenario. Since the deep reinforcement learning based algorithms usually require a large number of interactions with the environment, we assume that the inner loop controllers of the aircraft are well-tuned. We use a simplified model for the state of the aircraft.

The state-transition model is inspired by previous studies [1,13] with the following equations:

$$\psi = \psi + a_{\psi}\Delta_{\psi}, \tag{3.1}$$

$$v = \text{clip}(v + a_v\Delta_v, v_{min}, v_{max}), \tag{3.2}$$

$$x = x + v_t \cos(\psi \frac{\pi}{180^{\circ}}), \tag{3.3}$$

$$y = y + v_t \sin(\psi \frac{\pi}{180^{\circ}}). \tag{3.4}$$

In (3.1) and (3.2), the bank angle change Δ_{ψ} and the velocity change Δ_v selected as 10° and 0.1 m/s, respectively. We also limit the minimum and maximum velocity of the aircraft as $v_{min} = 4$ m/s and $v_{max} = 8$ m/s.

3.2 Air Combat Geometry

We describe a close-range air combat scenario to clarify the problem, where the black aircraft is the agent and the red aircraft is the enemy in Figure 3.1. Here, LOS is the line of sight, ATA is the antenna train angle, and AA is the aspect angle. LOS refers to the line passing through the center of gravity of two aircraft. ATA refers to the angle between the agent's heading and the enemy aircraft's position. AA denotes the angle between the enemy aircraft's tail and LOS. When both ATA and AA values get closer to zero, the agent gets the position, where it can shoot the enemy.

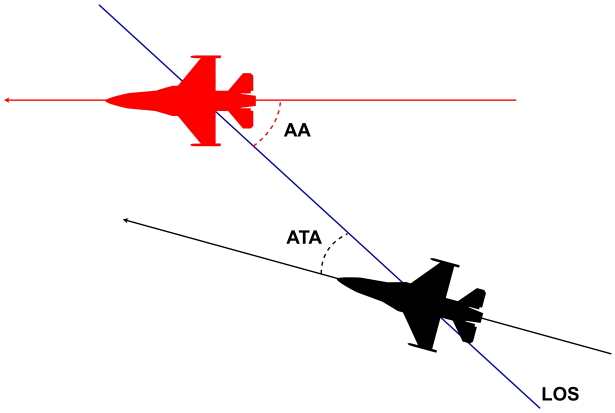


Figure 3.1 : Representation of a close-range one-to-one air combat scenario, where we control the black aircraft and the red aircraft is the enemy. It also illustrates antenna train angle (ATA), aspect angle (AA), and line of sight (LOS).

3.3 State Space

State space consists of 7 arguments:

$$s = [R, ATA_a, ATA_e, \psi_a, \psi_e, v_a, v_e], \tag{3.5}$$

where R is the distance, ATA_a is the agent's antenna train angle, ATA_e is the enemy's antenna train angle, ψ_a is the agent's bank angle, ψ_e is the enemy's bank angle, v_a is the agent's velocity and v_e is the enemy's velocity.

The ATA values and the distance R provide information about the relative positions of aircraft to each other. In addition, aircraft can predict the enemy's next position by taking other arguments, which are bank angles (ψ_a, ψ_e) and velocities (v_a, v_e).

3.4 Action Space

Aircraft take two different control inputs as an action; velocity v and angle ψ:

$$a = [v, \psi]. \tag{3.6}$$

These two parameters can be ascent, descend or remain same in discrete space. Therefore, the combination of these two parameters composes nine discrete actions:

$$\begin{bmatrix} v_t - \Delta_v, \psi_t + \Delta_\psi & v_t, \psi_t + \Delta_\psi & v_t + \Delta_v, \psi_t + \Delta_\psi \\ v_t - \Delta_v, \psi_t & v_t, \psi_t & v_t + \Delta_v, \psi_t \\ v_t - \Delta_v, \psi_t - \Delta_\psi & v_t, \psi_t - \Delta_\psi & v_t + \Delta_v, \psi_t - \Delta_\psi \end{bmatrix}. \tag{3.7}$$

3.5 Reward Design

Reward function design is an important step in the RL. The designed reward function should maximize the long-term utility. The aim of the air combat is to get behind the opponent aircraft and follow it as much as possible. It is also to prevent the enemy agent from following the training agent. In previous studies, discrete reward functions are designed [1,13] according to winning or losing an episode. In this study, on the contrary, we use a continuous reward function and provide its pseudo-code in Algorithm 5. In this reward function, when the agent gets behind the enemy aircraft, the smaller the angle between the direction of the agent aircraft and the enemy's position, the closer the reward value to 1. On the contrary, it gets closer to -1 when the enemy aircraft is behind us. If there is a crash between two aircraft, the reward is -1.

Algorithm 5 Reward Function

Input: $ATA_a, ATA_e, AA_a, AA_e, R$ = distance

if $(\text{crash}, 0 \text{ m} <= R < 10 \text{ m})$:
    $r = -1$
else if $(10 \text{ m} <= R <= 100 \text{ m})$:
    if $(\text{advantage}, |ATA_a| <= 30^\circ \quad \text{and} \quad |AA_a| < 60^\circ)$:
        $r = 1 - \frac{ATA_a}{30}$
    else if $(\text{disadvantage}, |ATA_e| <= 30^\circ \quad \text{and} \quad |AA_e| < 60^\circ)$:
        $r = \frac{ATA_e}{30} - 1$
else:
    $r = 0$

Output: r

3.6 Experiments

We model the environment as a closed box as shown in Figure 4.1. Therefore, the agents can not go beyond a certain distance. As seen in this figure, x and y length is 300m. We initialize the positions and angle values of the agent and the enemy randomly during the training phase in the environment. In this way, we force the agent to explore new states.

We normalize observations (Equation (3.5)) between 0 and 1 since we use neural networks as a function approximator. The episode length of the game is 200 timesteps.

We train various policies using four different algorithms (DQN, Rainbow, A2C, and PPO) and evaluate them using return G_t. Each algorithm has its hyper-parameters. While some of the hyper-parameters are common, some of them are algorithm-specific. We tune each algorithm's hyper-parameters using grid-search and summarize best policies in Table 3.1. In this table, we categorize algorithms by their architecture, value-based and actor-critic.

Table 3.1 : Deep reinforcement learning algorithm's training hyper-parameters.

		Algorithms			
Types of RL	**Hyper-parameters**	**DQN**	**Rainbow**	**A2C**	**PPO**
Both	learning rate	6.25e-5	6.25e-5	1e-4	3e-4
	gamma	0.99	0.99	0.99	0.99
	batch size	32	128		64
Value-based	memory size	100.000	100.000		
	learning starts	50.000	50.000		
	target update	10.000	10.000		
	alpha		0.5		
	beta		0.4		
	v_min		-150		
	v_max		150		
	n_steps		3		
Actor-critic	number of envs.			4	4
	n_epochs				10
	n_steps			5	2048
	gae_lambda			0.95	0.95
	clip_range				0.2

Learning rate, gamma, and batch size are common hyper-parameters for all the algorithms. We use Adam [36] as a neural network optimizer. We use similar architectures for both value-based and actor-critic methods as shown in Figure 3.2.

In the value-based architecture, there are two hidden layers with 128 neurons. The input and output sizes are equal to the state space and action space sizes, respectively. DQN and Rainbow algorithms use this architecture. However, it should be noted that the Rainbow algorithm's output is different because of the distributional atoms. It uses the distributional network architecture, as shown in Figure 2.2.

On the other hand, actor-critic network architecture has one shared network, and its output is divided into two branches: actor and critic networks. We use only one

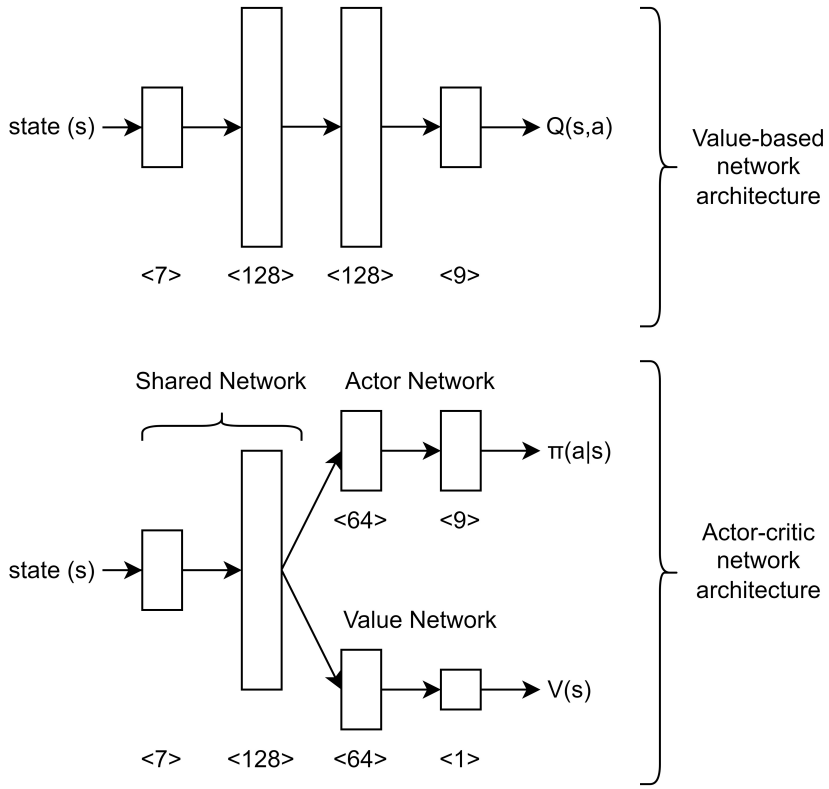


Figure 3.2 : Visualization of the two different network architectures: value-based and actor-critic.

network for this architecture. However, some instead create two separate networks. A2C and PPO algorithms use actor-critic network architecture.

We trained four algorithms (DQN, Rainbow, A2C, PPO) with 5 million timesteps and demonstrate results in Figure 3.3. In addition, we added a random policy, which selects a random action in each timestep.

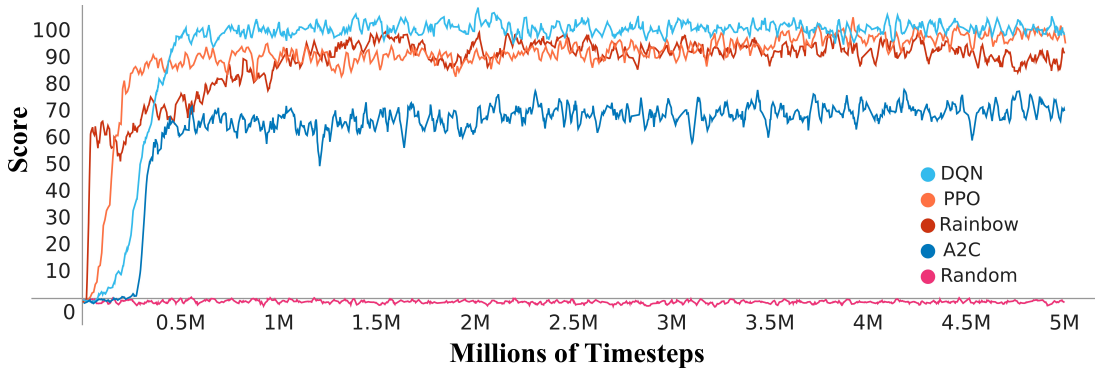


Figure 3.3 : The performance comparison of the reinforcement learning algorithms on the air-combat environment. The vertical axis is the score, demonstrating an episode's return value. The horizontal axis is the timesteps, indicating the number of actions taken during the training.

This figure demonstrates that the random policy gets around 0 returns in an episode. It might obtain positive rewards in some steps. However, it also faces negative rewards because of the policy's randomness, so the episode score converges to around 0.

We compared results with respect to episode return and sample efficiency. Nature DQN outperformed all algorithms in episode return. It achieved more than 100 scores in an episode. PPO and Rainbow algorithms obtained around 90 episode returns. However, the A2C algorithm could not converge well as the other algorithms. It achieved around 70 episode returns.

On the other hand, when we compare sample efficiency, the Rainbow algorithm obtained meaningful results faster than the algorithms. It reached 60 episode return in 30.000 timesteps. However, its convergence to the global optimal point is as long as the other algorithms.

4. REINFORCEMENT LEARNING UNDER NOISY OBSERVATIONS

4.1 Introduction

As stated in the Chapter 1.2, the previous studies assume that, the environment is assumed to be noise-free. Thus, the exact position of the enemy aircraft is known. However, this may not be achievable in a real-life scenario. For example, the sensor errors and medium conditions may cause the observation to differ from the reality. In this study, contrary to previous studies, we first develop a simulation that provides noisy observations to the agents. We also emphasize that the noise characteristics are adjustable through its parameters, hence, we can evaluate the performance of the trained agents under a wide range of noise level. Since the agents fights against the enemy aircraft by using the inaccurate observations, the air combat problem becomes more complex, hence, the performance of the RL algorithms usually decreases as the noise level increases. In our algorithm, we stack the consecutive observations to address deteriorating effect of the corrupted observations on the performance of the RL algorithms.

4.2 Noisy State Generation

The state transition model generated from aircraft dynamics is specified in the Section 3.1. It is assumed that the enemy aircraft's position is absolutely correct in these dynamics. However, it is usually impossible to obtain noise-free data in real life. In order to overcome this problem, we suggest to add noise to the state space during the training and testing phases. The noise-free state space is mentioned in the Equation (3.5).

Applying different noise to each of the parameters in state space may lead to inconsistencies. Hence, we insert the noise into the enemy aircraft's parameters before calculating these values. The details of the noisy state generation are shown

in Algorithm 6. The noise values are derived from the Gaussian distribution. The position noise $n_{x,y} \sim \mathcal{N}(\mu, \sigma^2_{x,y})$ is inserted to the enemy aircraft's position x_e and y_e, whereas the angle noise $n_\psi \sim \mathcal{N}(\mu, \sigma^2_\psi)$ is inserted to the enemy aircraft's angle ψ_e.

Since neural networks are used as a function approximator, we normalize values in state space between 0 and 1 to ensure stability during the training stage.

Algorithm 6 Noisy State Generation

Input: agent status = $\{x_a, y_a, \psi_a, v_a\}$, enemy status = $\{x_e, y_e, \psi_e, v_e\}$, $n_x, n_y \sim \mathcal{N}(\mu, \sigma^2_{x,y})$, $n_\psi \sim \mathcal{N}(\mu, \sigma^2_\psi)$.

1: $\hat{x}_e = x_e + n_x$, where $n_x \sim \mathcal{N}(\mu, \sigma^2_{x,y})$
2: $\hat{y}_e = y_e + n_y$, where $n_y \sim \mathcal{N}(\mu, \sigma^2_{x,y})$
3: $\hat{\psi}_e = \psi_e + n_\psi$, where $n_\psi \sim \mathcal{N}(\mu, \sigma^2_\psi)$
4: $\hat{R} = \sqrt{(x_a - \hat{x}_e)^2 + (y_a - \hat{y}_e)^2}$
5: $\hat{ATA}_a = ATA(x_a, y_a, \hat{x}_e, \hat{y}_e, \psi_a)$
6: $\hat{ATA}_e = ATA(\hat{x}_e, \hat{y}_e, x_a, y_a, \hat{\psi}_e)$

Output: $s_{t+1} = [\hat{R}, \hat{ATA}_a, \hat{ATA}_e, \psi_a, \hat{\psi}_e, v_a, v_e]$

Figure 4.1 demonstrates the actual and the noisy positions of the enemy aircraft. In this figure, the black plane is the training agent, the red plane is the enemy agent, and the orange plane is how the training agent sees the enemy agent. In part (a) of this figure, the enemy's position noise value is $n_{x,y} \sim \mathcal{N}(0,5)$ and the angle noise value is $n_\psi \sim \mathcal{N}(0,1)$. In part (b) of this figure, the enemy's position noise value is $n_{x,y} \sim \mathcal{N}(0,20)$ and the angle noise value is $n_\psi \sim \mathcal{N}(0,1)$. Since the noise in part (b) is high, the training agent perceives the opponent as more dispersed.

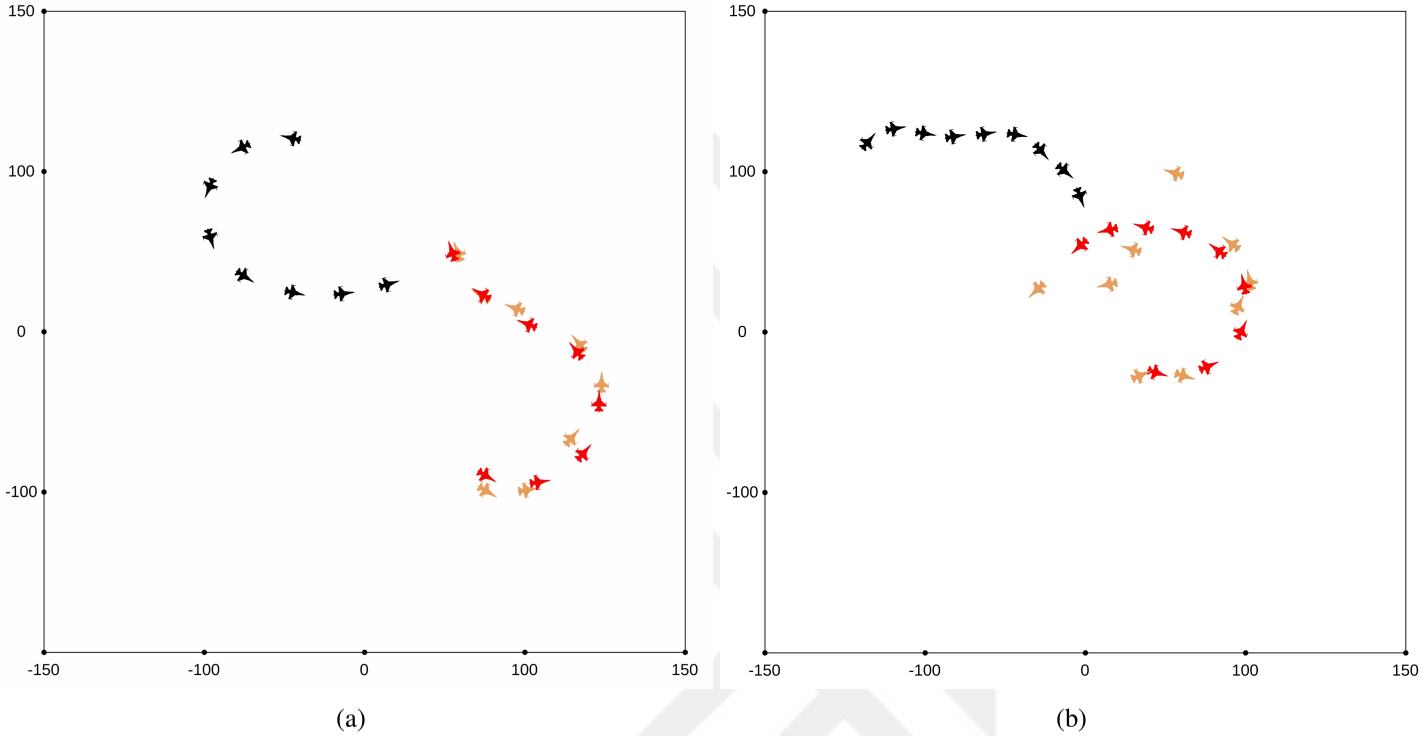


Figure 4.1 : Visualization of the effect of the noisy state space on the environment. The black plane is the training agent, the red plane is the enemy agent, and the orange plane is how the training agent sees the enemy agent. (a) The position noise value is $n_{x,y} \sim \mathcal{N}(0,5)$, and the angle noise value is $n_{\psi} \sim \mathcal{N}(0,1)$ (b) The position noise value is $n_{x,y} \sim \mathcal{N}(0,20)$, and the angle noise value is $n_{\psi} \sim \mathcal{N}(0,1)$.

4.3 Noise Reduction with State Stacking

The perceived position of the enemy aircraft by the agent changes depending on the variance of the noise as shown in Figure 4.1. The agent can generalize and ignore the noise in low variance cases. However, after the noise exceeds a certain threshold, the agent's performance begins to degrade.

There is a correlation between states, since the positions of the agent are sequential. Therefore, we propose a state stacking technique in order to reduce noise and increase performance. The agent sees the state space at time t is as follows:

$$S_t = \{S_t, S_{t-1}, ..., S_{t-n+1}\}, \tag{4.1}$$

where the n is the stack number. The trained model associates between states using this technique.

4.4 Enemy Strategy and Self-play

Another subject emphasized in this study is the strategy of the enemy agent. As stated by Bansal et al. [37], the complexity of the trained RL agent depends on the complexity of the environment. In other words, smarter agents can be trained in more complex environments. They also pointed out that self-play can produce substantially more complex behaviors than the environment itself. In addition, self-play also provides many advantages:

- Any supervision or human expertise is no longer required.
- Agent can learn the policy by playing with itself.
- It ensures that the environment difficulty is at the right level to improve agent strategy.

To show the effect of the self-play, we enforce the enemy agent to select random action, i.e., the enemy agent chooses the action from the network with probability $1-\lambda$ and

take random action from uniform distribution with probabililty λ. For the specisif case $\lambda = 0$, the enemy agent always use the actions from the network.

4.5 Experiments

We model the environment as a closed box as shown in Figure 4.1. Therefore, the agents can not go beyond a certain distance. We initialize the positions and angle values of the agent and the enemy randomly during the training phase in the environment. In this way, the agent is forced to explore new states.

We conducted a hyper-parameter search and the selected hyper-parameter set is given in Table 4.1, where we use the same hyper-parameter set for all experiments.

Table 4.1 : Reinforcement learning policies hyper-parameters.

Hyper-parameter	Value
State space size	7
Action space size	9
Learning rate, lr	1e-4
Discount factor, γ	0.99
Replay memory size	20.000
Mini batch size	32
Target update frequency, C	10.000
Initial exploration rate	1
Final exploration rate	0.05
Final exploration time-steps	100.000
Episode length	200

4.5.1 State stacking on different noise levels

We carry out different experiments on the state stacking approach without using self-play. Two different training graphs with different state noise are given in Figure 4.2 and Figure 4.3. In the all of the training processes, the stack number value are chosen as 1, 2, 4, and 8.

Firstly, in Figure 4.2, the enemy's position noise value is $n_{x,y} \sim \mathcal{N}(0,5)$ and the enemy's angle noise value is $n_{\psi} \sim \mathcal{N}(0,1)$. In this figure, when the state stack is not used, the reward value is around 90. On the contrary, the agent's performance increases around 10% when stack number is greater than 1. As a result of this experiment,

although the use of stacked states provides performance gains, the number of stacks has only a marginal improvement.

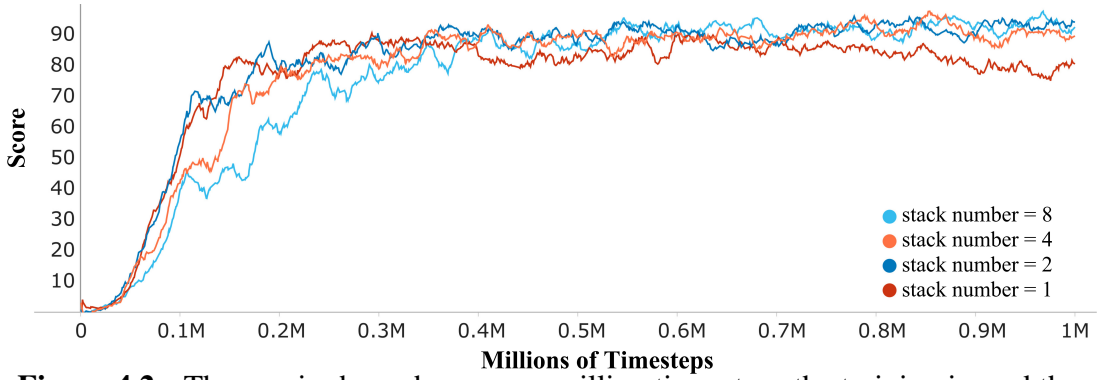


Figure 4.2 : The x-axis shows how many million time-steps the training is, and the y-axis shows the obtained score for a certain moment t. Self-play is not used in this experiment. The position noise value is $n_{x,y} \sim \mathcal{N}(0,5)$ and the angle noise value is $n_{\psi} \sim \mathcal{N}(0,1)$. It is observed that the performance increases around %10 as the state stack is increased.

Secondly, in Figure 4.3, enemy's position variance is increased to $\sigma_{x,y} = 20$, but its angle variance is remained as $\sigma_{\psi} = 1$. In this case, the environment is quite noisy, and the episode score is around 44 when the stack number is 1. On the other hand, the score increases to around 70, when the number of state stack is 8. In other words, state stacking increases the performance around %59 compared to the situation where no stacking is used. In addition, it is clearly shown in this graph that the score increases in proportion to the number of stacks. This gives information about how robust the proposed method is.

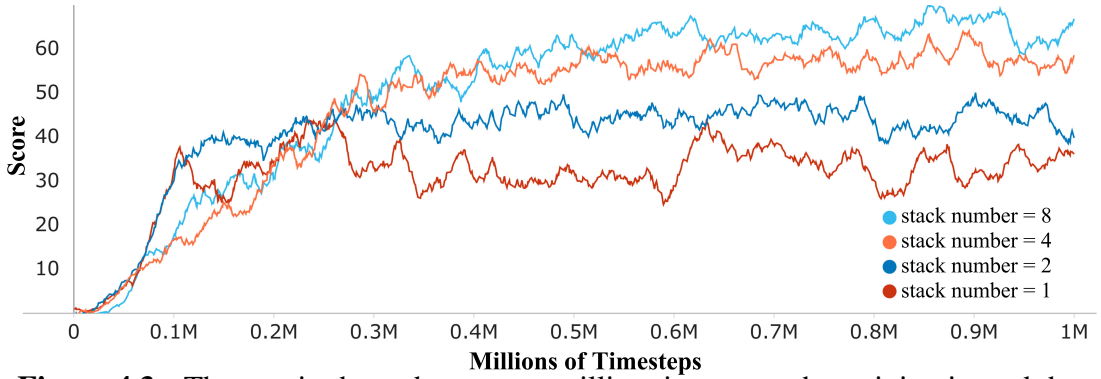


Figure 4.3 : The x-axis shows how many million time-steps the training is, and the y-axis shows the obtained score for a certain moment t. Self-play is not used in this experiment. The position noise value is $n_{x,y} \sim \mathcal{N}(0,20)$ and the angle noise value is $n_{\psi} \sim \mathcal{N}(0,1)$. It can be observed that the performance increases around %59 as the state stack is increased.

Table 4.2 : Highest scores in different variation of noise and state stack values. Normalized stacking scores is calculated as: 100 x (highest stacking score / # of stack=1).

Environment noise	# of stack = 1	# of stack = 2	# of stack = 4	# of stack = 8	Normalized stacking score
$n_{x,y} \sim \mathcal{N}(0,1)$	**107.3**	105.1	106.9	104.6	% 99,62
$n_{x,y} \sim \mathcal{N}(0,3)$	**102.8**	102.1	101.6	**102.8**	%100
$n_{x,y} \sim \mathcal{N}(0,5)$	90.82	96.11	97.65	**98.25**	%108.18
$n_{x,y} \sim \mathcal{N}(0,10)$	68.2	80.92	85.79	**88.89**	%130.33
$n_{x,y} \sim \mathcal{N}(0,20)$	44.17	50.11	64.1	**70.47**	%159.54
$n_{x,y} \sim \mathcal{N}(0,40)$	19.11	34.39	39.95	**45.49**	%238.04

In addition, we train policies with different combinations of state stack and noise values and summarize it in Table 4.2. This table shows that as the noise value increases, obtained highest score decreases inversely. Stack number does not affect the performance where the position's noise variance is low ($\sigma^2_{x,y} = 1$ and $\sigma^2_{x,y} = 3$). However, when noise exceeds these values, stacking starts to compensate the noise.

4.5.2 Training with self-play

There are two sides in the air combat problem. Better strategies may emerge if there is competition between these sides during training. One of the purpose of the self-play is to employ this contest. Therefore, we design a training process that includes self-play. In this experimental setup, enemy policy π^t_e is replaced with the agent policy π^t_a at every n time-steps. The parameter n is selected as 50.000 by using grid search. In addition, the enemy is forced to choose random action with the probability λ and action from its own policy with the probability $1 - \lambda$.

In this experiment, we compare the performance of self-play agent with different exploration parameter λ. Thus, we demonstrate the effect of self-play. The results of the self-play are shown in Table 4.3. In this table, the enemy agent is the one, which is trained without self-play. Both the agent and the enemy are trained in the same environment conditions, where the noise variances are $\sigma_{x,y} = 10$ and $\sigma_{\psi} = 1$. These two agents also did not encounter each other during the training.

We run 1000 Monte Carlo simulation. At the beginning of the each simulation, we initialize both agents at random positions with random angles. Win, lose and tie conditions are designed according to the reward function. When any agent achieves the strike angle within striking distance, then it is the winner. The maximum episode length is set as 200 time-steps. If both agents are alive at the end of the episode, then tie condition occurs.

As seen in Table 4.3, it is obvious that self-play agents outperformed each match regardless of the parameter λ. However, for the lower values of the parameter λ, which correspond to the more sophisticated enemy agents during training, the training agent achieved better performances in the test stage.

Table 4.3 : Agents trained with self-play and without self-play launched from random locations and faced 1000 times. Noise value's variances are $\sigma_{x,y} = 10$ and $\sigma_{\psi} = 1$. The win probability is calculated as: win/(win+lose).

Agent	Enemy	Win	Lose	Tie	Win Probability
Self-play $\lambda = 0$	$n_{x,y} \sim \mathcal{N}(0,10)$	726	99	175	0.88
Self-play $\lambda = 0.2$	$n_{x,y} \sim \mathcal{N}(0,10)$	753	102	145	0.88
Self-play $\lambda = 0.5$	$n_{x,y} \sim \mathcal{N}(0,10)$	758	109	133	0.87
Self-play $\lambda = 0.9$	$n_{x,y} \sim \mathcal{N}(0,10)$	574	309	117	0.65

5. CLASSIFYING NOISY OBSERVATIONS IN REINFORCEMENT LEARNING SIMULATIONS

5.1 Introduction

In Chapter 3, we create a baseline air combat environment and compare state-of-the-art deep reinforcement learning algorithms. Chapter 4 proposes a noise generation method for air combat environments. Since the noise makes it difficult to predict the actual location of the enemy, the RL algorithm's performance decreases. Therefore, we propose state-stacking and self-play methods to lower noise and increase performance. We train different RL policies in different noise-level environments using these methods. Experiments demonstrate that the proposed method outperforms baseline DRL algorithms. In this structure, we assume that the environmental noise level is known. However, the aircraft does not know the environment's noise level in a real-life test scenario.

In this chapter, we propose a feed-forward neural network-based noise classifier to determine noise-level. Our main idea is to integrate a box that takes observations as input and predict noise level as outputs. Since the input of this box is the same as the RL algorithm, it does not affect the RL loop.

5.2 Noise Classifier

We collect three datasets using the simulation, which generates noisy observation. Datasets contain five classes: noise's standard deviations are 0, 4, 8, 16, and 32. Each dataset has 50.000 samples. Their input size is different and depends on the stack sizes. Therefore, we train three different classifiers for each stack size.

The end-to-end noisy air combat architecture is given in Figure 5.1. The architecture consists of the agent and the environment. The agent includes three steps. In the first step, the agent receives the current observation and stacks it with the previous $n-1$ states. In the second step, the classifier takes stacked states as input. Then, it

predicts the environment noise level and selects a proper RL policy. Finally, the proper RL policy takes the stacked state and selects an action. Note that the noise classifier and RL policies are trained separately, although they are given together in the figure demonstrating the overall system architecture.

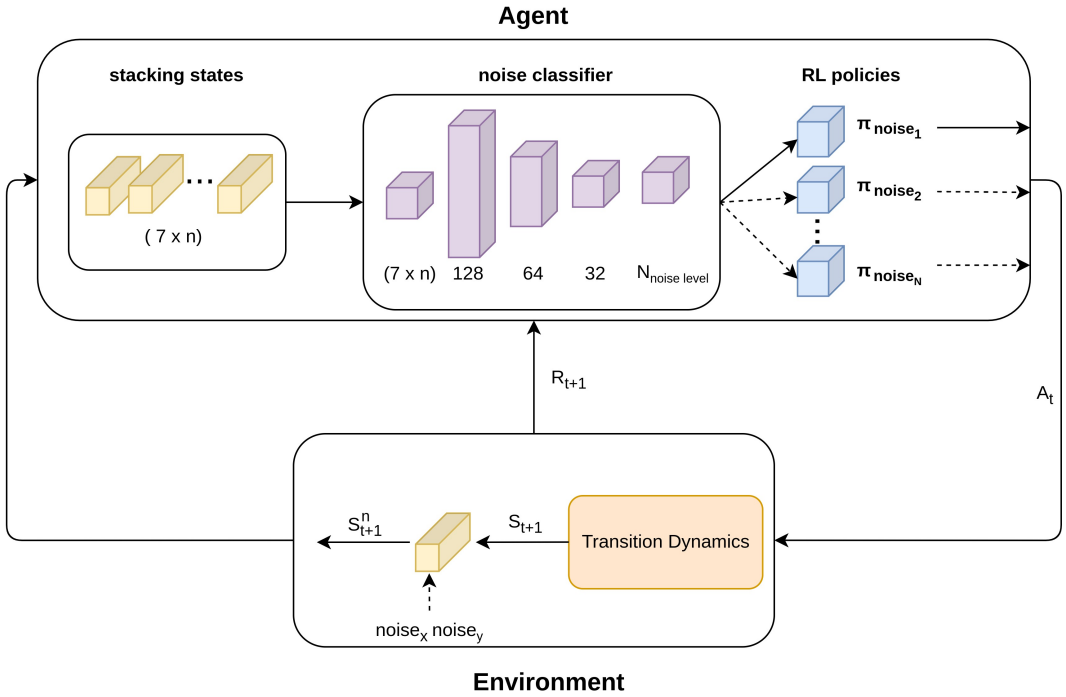


Figure 5.1 : Visualization of reinforcement learning cycle incorporating noise classifier architecture. In this architecture, the agent includes state stacking, noise classification, and reinforcement learning policies. The environment includes transition dynamics, and the addition of noise to the situation.

5.2.1 Training setup

As mentioned in the previous section, we train three classifiers based on the stack sizes: 2, 4, and 8. We use the feed-forward neural network as shown in Figure 5.1. The input size of the neural network $(7 \text{ x } n)$ depends on the stack size n. The output size is the same and five for each classifier: 0 (noiseless), 4, 8, 16, and 32.

Table 5.1 : Noise classifier hyper-parameters.

Hyper-parameter	Values
Learning rates	[0.01, 0.001, 0.0001 0.03 0.003 0.0003 0.07 0.007 0.0007]
Weight decay	[True, False]
Step sizes	$[1, 100, 200]$

We tune the classifiers using the hyperparameters in Table 5.1. In this table, the learning rate is the step size used when updating the neural network weights. Weight decay is a regularization method. It reduces the over-fitting of training. We select the weight decay parameter as 0.0001. Learning rate decay is another regularization method used in this study. We decay the learning rate at each step size. For example, when the step size is 1, the learning rate is multiplied by 0.99 at the end of each epoch; when the step size is 100 and 200, the learning rate is multiplied by 0.1. We divide the data set into %80 training set and %20 validation set. We use Adam as a neural network optimizer. We use cross entropy as a loss function.

5.3 Experiments

The training and validation accuracy rates are shown in Figure 5.2 and Figure 5.3. The training is more stable when the number of stacks is 4. When the number of stacks is 2, the classifier reaches about %81 validation because the inference from sequential data is less than other methods. Figure 5.3 illustrates that learning rate decay increases the performance when the stack number is 4 and 2, whereas it decreases the performance when the stack number is 8.

Another evaluation method is the loss function. Training and validation loss values are given respectively in Figure 5.5 and Figure 5.6. In these figures, we only compare the best results of the accuracy figures. When the number of stacks is 4, it is obvious that the classifier reaches lower loss values more quickly.

We train each classifier for 600 epochs. However, as seen in the loss function figures, we terminated the training with an early stop at the points where the training loss continued to decrease, but the validation loss remained constant or started to oscillate. We indicate the points by the red dots.

We can use confusion matrixes to evaluate the trained models in more detail based on classes. Confusion matrixes of the classifiers are given in Figure 5.7, Figure 5.8 and Figure 5.9. When the stack number is 2, the classifier obtains the lowest results for each class in proportion to the validation accuracy. We observe the highest error rate in classes with noise values of 16 and 32. When the stack number is 4, the classifier

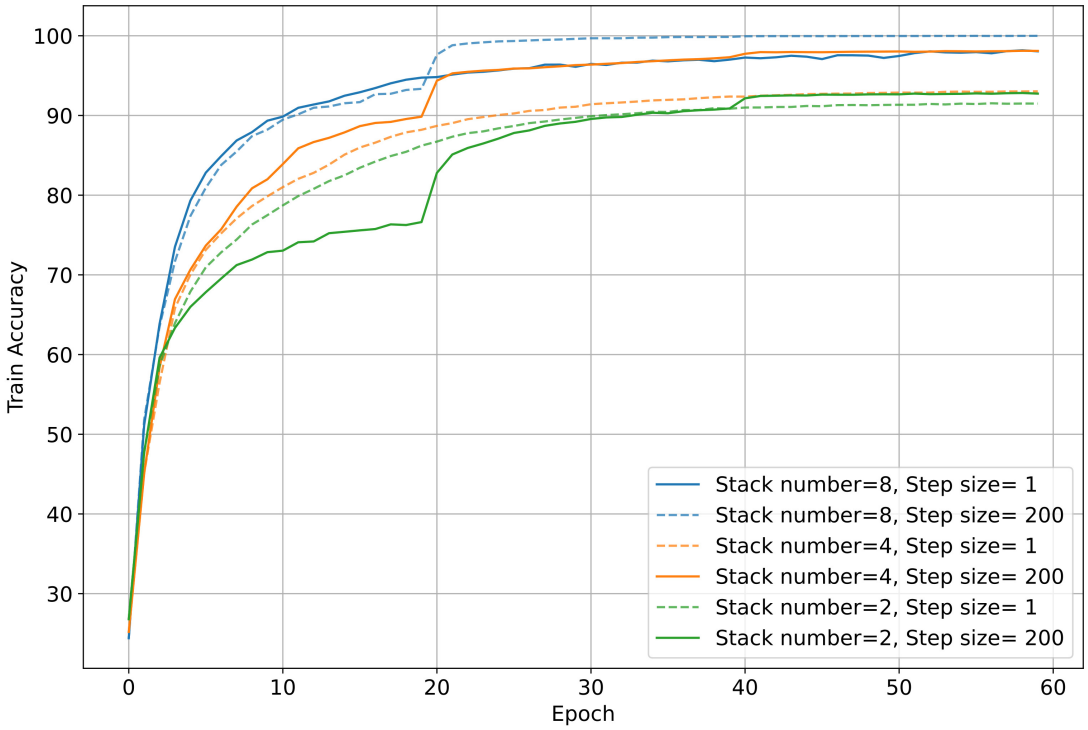


Figure 5.2 : Train accuracy of the noise classifier. The stack number is the last n states, a noise reduction technique. When the stack number is 2 and 4, we decay the learning rate in epoch 200 and 400.

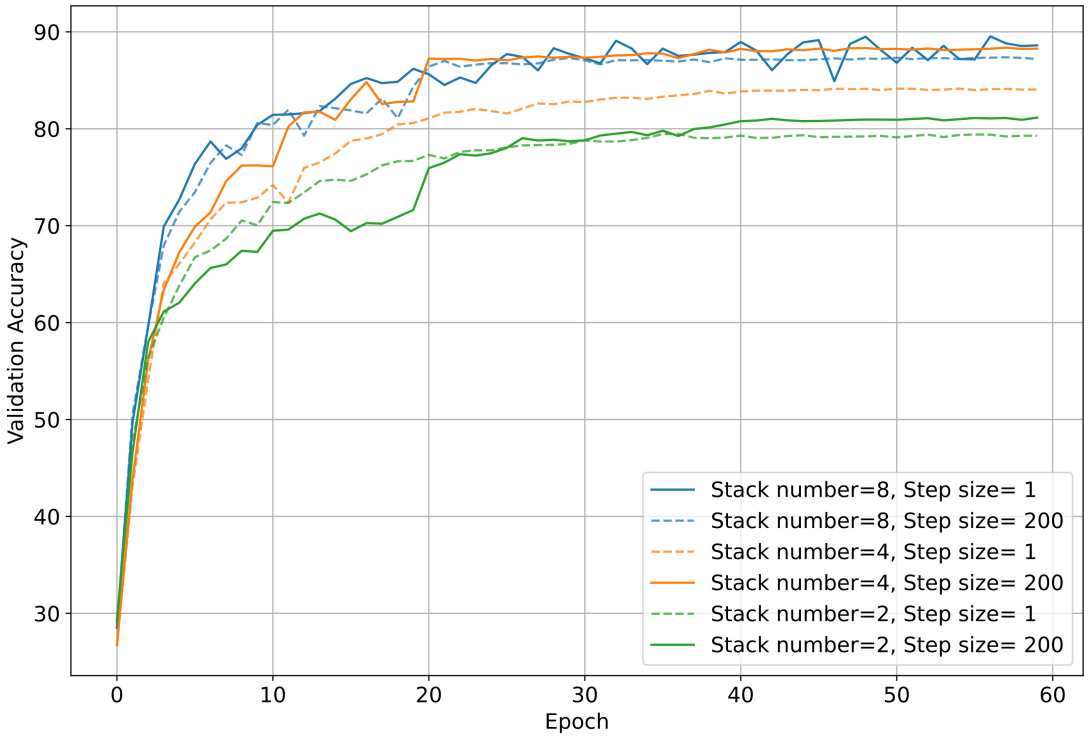


Figure 5.3 : Validation accuracy of the noise classifier. The stack number is the last n states, a noise reduction technique. When the stack number is 2 and 4, we decay the learning rate in epoch 200 and 400.

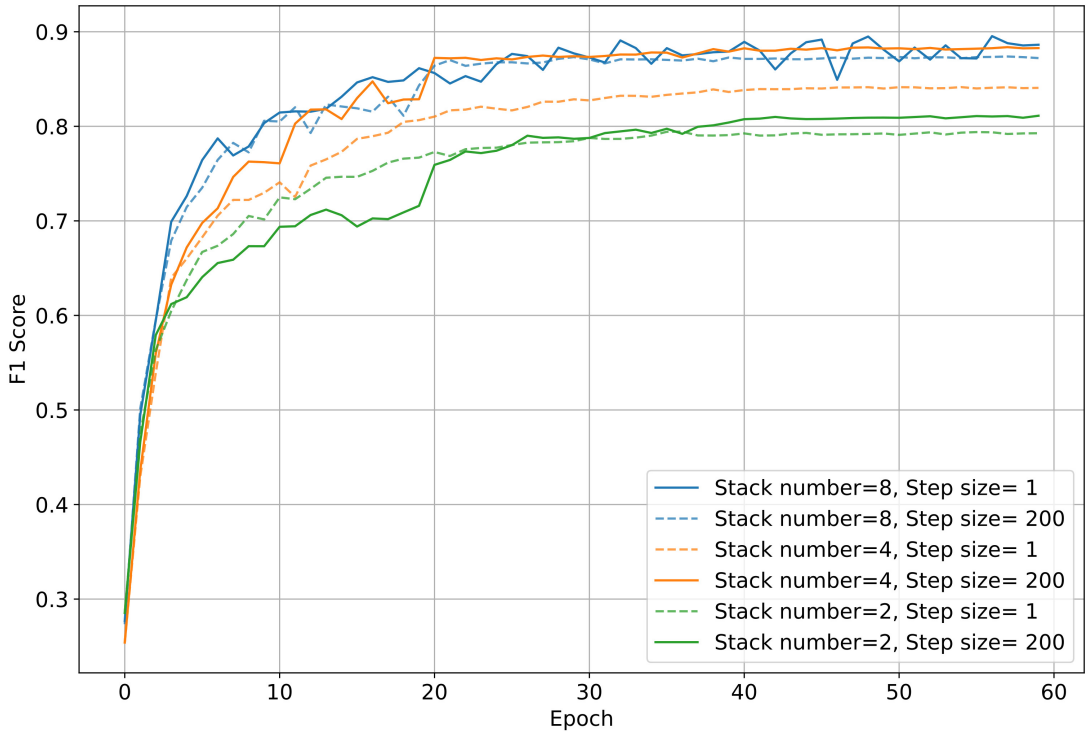


Figure 5.4 : F1 scores of the noise classifier.

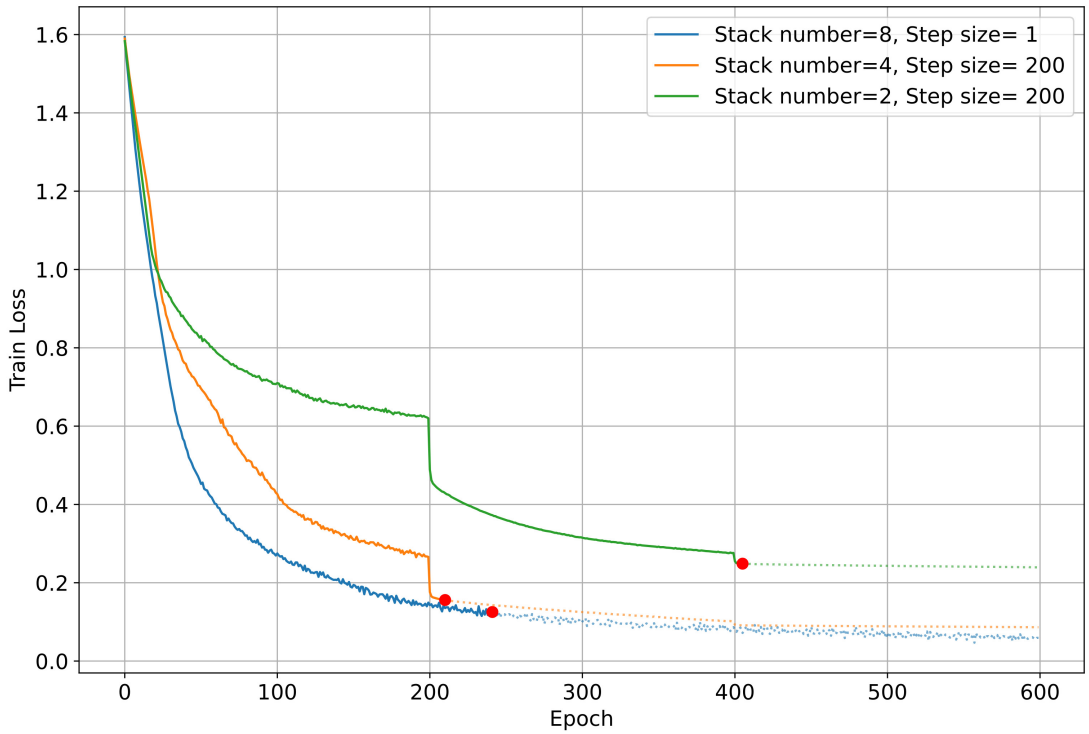


Figure 5.5 : Train loss of the noise classifier. The stack number is the last n states, a noise reduction technique. Red points demonstrate the early stop, where we stop the training and prevent overfitting.

predicts the true 16 noise level with the lowest accuracy. When the number of stacks is 8, we obtain better results on a class basis than the other two classifiers.

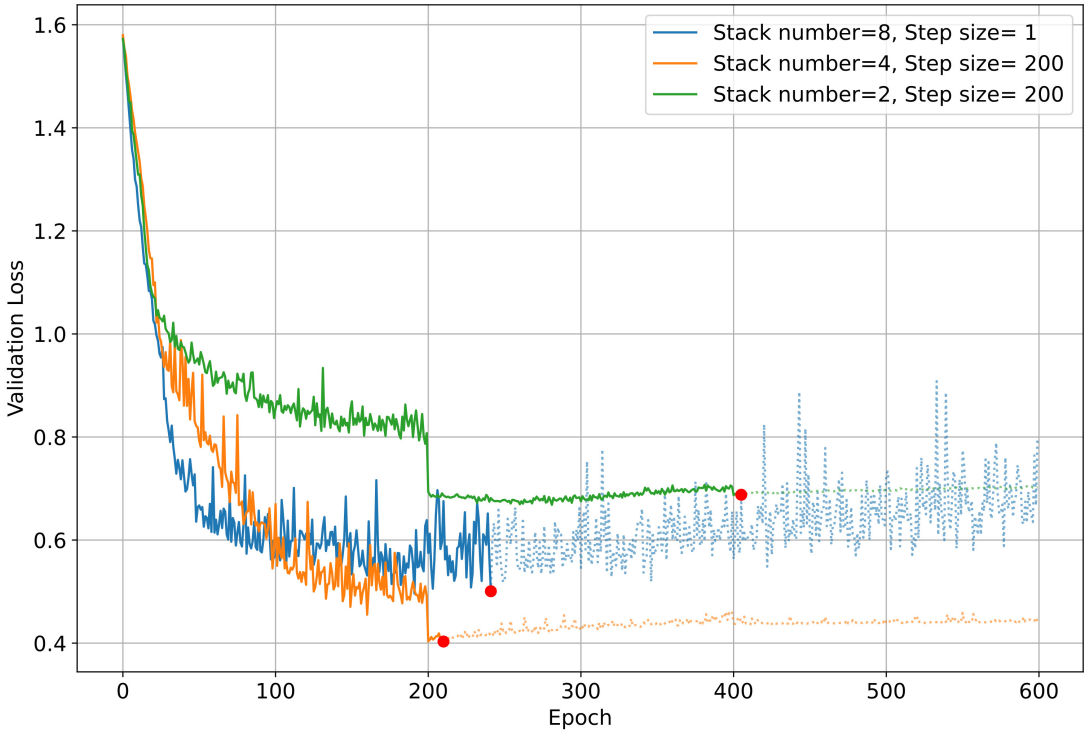


Figure 5.6 : Validation loss of the noise classifier. The stack number is the last n states, a noise reduction technique. Red points demonstrate the early stop, where we stop the training and prevent overfitting.

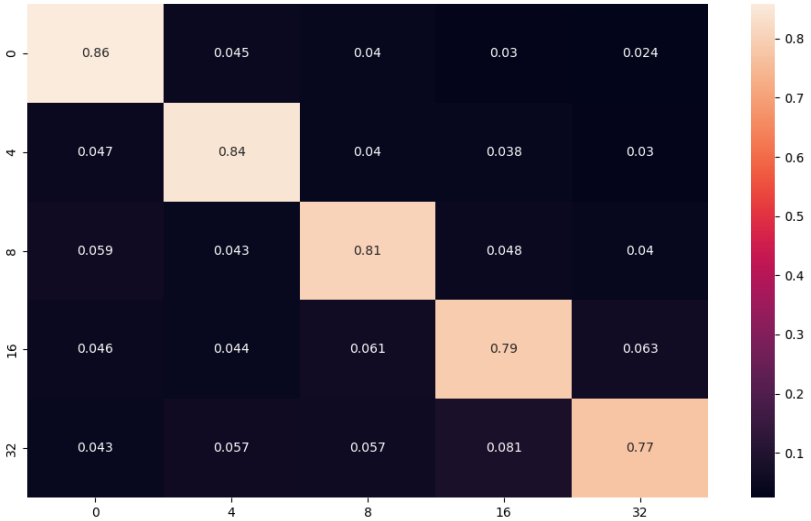


Figure 5.7 : Confusion matrix of the noise classifier when the stack number is 2.

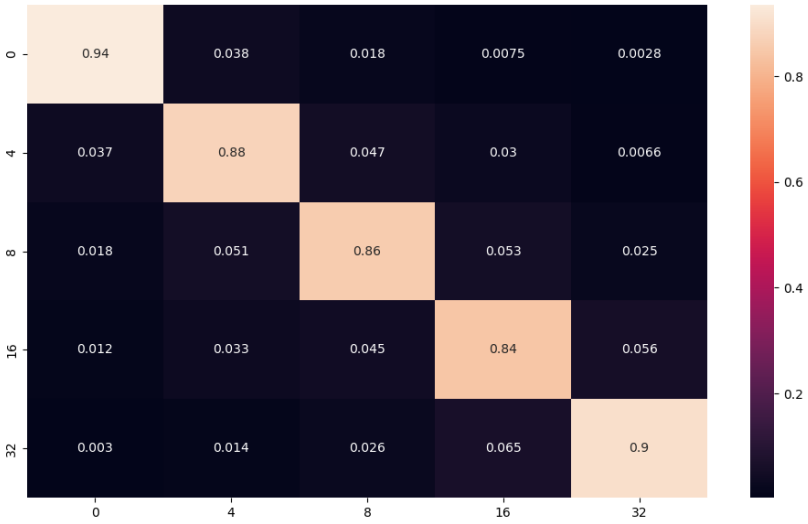


Figure 5.8 : Confusion matrix of the noise classifier when the stack number is 4.

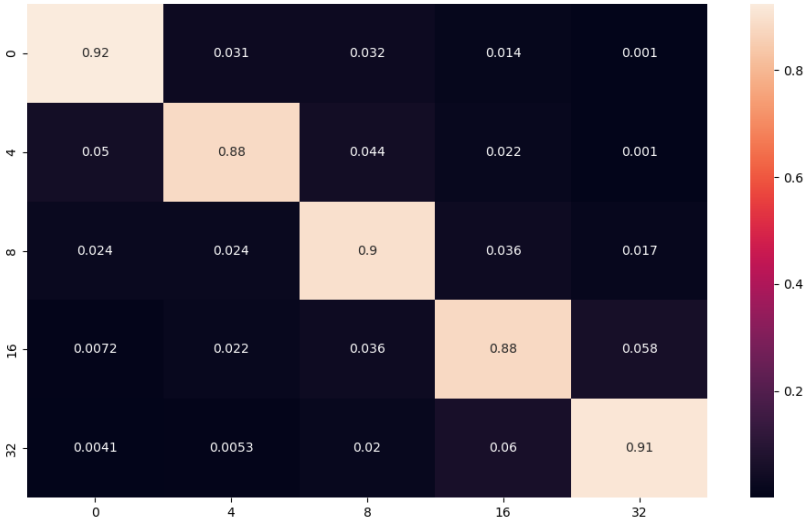


Figure 5.9 : Confusion matrix of the noise classifier when the stack number is 8.

6. CONCLUSIONS

In this thesis, we investigated the air combat problem under noisy conditions. We examined the creation of a multi-agent air combat environment and the addition of noise to this environment. We proposed a novel structure to solve this problem based on state-of-the-art reinforcement learning algorithms.

Firstly, we implemented aircraft dynamics and created an environment that supports multi-agent interactions. Then, we defined state and action space for reinforcement learning algorithms. We also designed a continuous reward function and provided its pseudo-code. We compared state-of-the-art deep reinforcement learning algorithms in the air combat environment, including DQN, Rainbow, PPO, and A2C.

In addition to previous works, we developed a simulation that generates noisy observations to the agents. Since the noise can be adjustable through its parameters, we could evaluate our policies in different noise conditions. As a result of noise addition, the RL agent's performance decreased as the noise level increased. Therefore, we proposed a state-stacking technique, which reduces the noise and increases the trained policy performance. We empirically show that the proposed stacking method increases performance up to %138.04 in high-noise environments.

Secondly, we focused on the create smarter enemy behaviors using competitive self-play techniques. Thus, the agent faced smarter enemy strategies in the training stage and obtained better policies. We also created different self-play structures to demonstrate the effect of the proposed method. We empirically demonstrate that the agent trained with self-play beats the agent trained without self-play by %88 win rate.

Thirdly, we trained different DRL policies in different noise-level environments. However, the environment's noise level is unpredictable in the validation stage. Therefore, we collected data using the air combat simulation. The content of the data is the state and the noise level in that state. We trained a classifier that takes states as

input and selects noise level as output. The classifier achieves around %90 validation accuracy.

Finally, we combined the overall system in an end-to-end architecture. This architecture contains the agent and the environment. The environment takes action as an input. It applies the action in the state and reward transitions, and obtains noisy states and a reward value. The agent takes the noisy state as an input. It applies state stacking and transmits data to the noise classifier. The classifier determines the current noise level and selects the proper deep reinforcement learning policy for action selection.

REFERENCES

[1] **McGrew, J.S.**, **How, J.P.**, **Williams, B. and Roy, N.** (2010). Air-Combat Strategy Using Approximate Dynamic Programming, *Journal of Guidance, Control, and Dynamics*, *33*(5), 1641–1654.

[2] **Chen, H.**, **Wang, X.m. and Li, Y.** (2009). A survey of autonomous control for UAV, *2009 International Conference on Artificial Intelligence and Computational Intelligence*, volume 2, IEEE, pp.267–271.

[3] **Yang, Q.**, **Zhang, J.**, **Shi, G.**, **Hu, J. and Wu, Y.** (2020). Maneuver Decision of UAV in Short-Range Air Combat Based on Deep Reinforcement Learning, *IEEE Access*, *8*, 363–378.

[4] **Song, Y.**, **Steinweg, M.**, **Kaufmann, E. and Scaramuzza, D.** (2021). Autonomous Drone Racing with Deep Reinforcement Learning, 1205–1212.

[5] **Berner, C.**, **Brockman, G.**, **Chan, B.**, **Cheung, V.**, **Dębiak, P.**, **Dennison, C.**, **Farhi, D.**, **Fischer, Q.**, **Hashme, S.**, **Hesse, C.** *et al.* (2019). Dota 2 with large scale deep reinforcement learning, *arXiv preprint arXiv:1912.06680*.

[6] **Fawzi, A.**, **Balog, M.**, **Huang, A.**, **Hubert, T.**, **Romera-Paredes, B.**, **Barekatain, M.**, **Novikov, A.**, **R Ruiz, F.J.**, **Schrittwieser, J.**, **Swirszcz, G.** *et al.* (2022). Discovering faster matrix multiplication algorithms with reinforcement learning, *Nature*, *610*(7930), 47–53.

[7] **Baker, B.**, **Kanitscheider, I.**, **Markov, T.**, **Wu, Y.**, **Powell, G.**, **McGrew, B. and Mordatch, I.** (2019). Emergent tool use from multi-agent autocurricula, *arXiv preprint arXiv:1909.07528*.

[8] **Tasbas, A.S.**, **Sahin, S.O. and Üre, N.K.** (2023). Reinforcement Learning Based Self-play and State Stacking Techniques for Noisy Air Combat Environment, *AIAA SCITECH 2023 Forum*, p.1077.

[9] **Isci, H. and Koyuncu, E.** (2022). Reinforcement Learning Based Autonomous Air Combat with Energy Budgets, *AIAA SCITECH 2022 Forum*, p.0786.

[10] **Pope, A.P.**, **Ide, J.S.**, **Mićović, D.**, **Diaz, H.**, **Rosenbluth, D.**, **Ritholtz, L.**, **Twedt, J.C.**, **Walker, T.T.**, **Alcedo, K. and Javorsek, D.** (2021). Hierarchical Reinforcement Learning for Air-to-Air Combat, *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp.275–284.

[11] **Hu, T.**, **Hu, J.**, **Zhao, C. and Pan, Q.** (2023). Autonomous Decision Making of UAV in Short-Range Air Combat Based on DQN Aided by

Expert Knowledge, *Proceedings of 2022 International Conference on Autonomous Unmanned Systems (ICAUS 2022)*, Springer, pp.1661–1670.

[12] **Hu, D.**, **Yang, R.**, **Zuo, J.**, **Zhang, Z.**, **Wu, J. and Wang, Y.** (2021). Application of Deep Reinforcement Learning in Maneuver Planning of Beyond-Visual-Range Air Combat, *IEEE Access*, *9*, 32282–32297.

[13] **Ma, X.**, **Xia, L. and Zhao, Q.** (2018). Air-Combat Strategy Using Deep Q-Learning, *2018 Chinese Automation Congress (CAC)*, pp.3952–3957.

[14] **Kong, W.**, **Zhou, D.**, **Yang, Z.**, **Zhao, Y. and Zhang, K.** (2020). UAV Autonomous Aerial Combat Maneuver Strategy Generation with Observation Error Based on State-Adversarial Deep Deterministic Policy Gradient and Inverse Reinforcement Learning, *Electronics*, *9*(7), `https://www.mdpi.com/2079-9292/9/7/1121`.

[15] **Taşbaş, A.S. and Aydınlı, S.U.** (2021). 2-D Air Combat Maneuver Decision Using Reinforcement Learning, *2021 International Conference on Engineering and Emerging Technologies (ICEET)*, IEEE, pp.1–6.

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

[17] **Hessel, M.**, **Modayil, J.**, **Van Hasselt, H.**, **Schaul, T.**, **Ostrovski, G.**, **Dabney, W.**, **Horgan, D.**, **Piot, B.**, **Azar, M. and Silver, D.** (2018). Rainbow: Combining improvements in deep reinforcement learning, *Proceedings of the AAAI conference on artificial intelligence*, volume 32.

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

[19] **Mnih, V.**, **Badia, A.P.**, **Mirza, M.**, **Graves, A.**, **Lillicrap, T.**, **Harley, T.**, **Silver, D. and Kavukcuoglu, K.** (2016). Asynchronous methods for deep reinforcement learning, *International conference on machine learning*, PMLR, pp.1928–1937.

[20] **Sutton, R.S. and Barto, A.G.** (2018). *Reinforcement learning: An introduction*, MIT press.

[21] **Bellemare, M.G.**, **Naddaf, Y.**, **Veness, J. and Bowling, M.** (2013). The arcade learning environment: An evaluation platform for general agents, *Journal of Artificial Intelligence Research*, *47*, 253–279.

[22] **Todorov, E.**, **Erez, T. and Tassa, Y.** (2012). Mujoco: A physics engine for model-based control, *2012 IEEE/RSJ international conference on intelligent robots and systems*, IEEE, pp.5026–5033.

[23] **URL-3**. https://www.davidsilver.uk/teaching/, date retrieved: 10.09.2022.

[24] **Krizhevsky, A.**, **Sutskever, I. and Hinton, G.E.** (2012). ImageNet Classification with Deep Convolutional Neural Networks, ***F. Pereira, C. Burges, L. Bottou and K. Weinberger***, *editors, Advances in Neural Information Processing Systems*, volume 25, Curran Associates, Inc., https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[25] **Sermanet, P.**, **Kavukcuoglu, K.**, **Chintala, S. and Lecun, Y.** (2013). Pedestrian Detection with Unsupervised Multi-Stage Feature Learning, *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, IEEE Computer Society, USA, p.3626–3633, https://doi.org/10.1109/CVPR.2013.465.

[26] **Mikolov, T.**, **Chen, K.**, **Corrado, G. and Dean, J.** (2013). Efficient Estimation of Word Representations in Vector Space, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, http://arxiv.org/abs/1301.3781.

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

[28] **Van Hasselt, H.**, **Guez, A. and Silver, D.** (2016). Deep reinforcement learning with double q-learning, *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

[29] **Schaul, T.**, **Quan, J.**, **Antonoglou, I. and Silver, D.** (2015). Prioritized experience replay, *arXiv preprint arXiv:1511.05952*.

[30] **Wang, Z.**, **Schaul, T.**, **Hessel, M.**, **Hasselt, H.**, **Lanctot, M. and Freitas, N.** (2016). Dueling network architectures for deep reinforcement learning, *International conference on machine learning*, PMLR, pp.1995–2003.

[31] **Sutton, R.S.** (1988). Learning to predict by the methods of temporal differences, *Machine learning*, *3*, 9–44.

[32] **Bellemare, M.G.**, **Dabney, W. and Munos, R.** (2017). A distributional perspective on reinforcement learning, *International conference on machine learning*, PMLR, pp.449–458.

[33] **Dabney, W.**, **Ostrovski, G.**, **Silver, D. and Munos, R.** (2018). Implicit quantile networks for distributional reinforcement learning, *International conference on machine learning*, PMLR, pp.1096–1105.

[34] **Fortunato, M.**, **Azar, M.G.**, **Piot, B.**, **Menick, J.**, **Hessel, M.**, **Osband, I.**, **Graves, A.**, **Mnih, V.**, **Munos, R.**, **Hassabis, D.** *et al.*. Noisy Networks For Exploration, *International Conference on Learning Representations*.

[35] **Schulman, J.**, **Levine, S.**, **Abbeel, P.**, **Jordan, M. and Moritz, P.** (2015). Trust region policy optimization, *International conference on machine learning*, PMLR, pp.1889–1897.

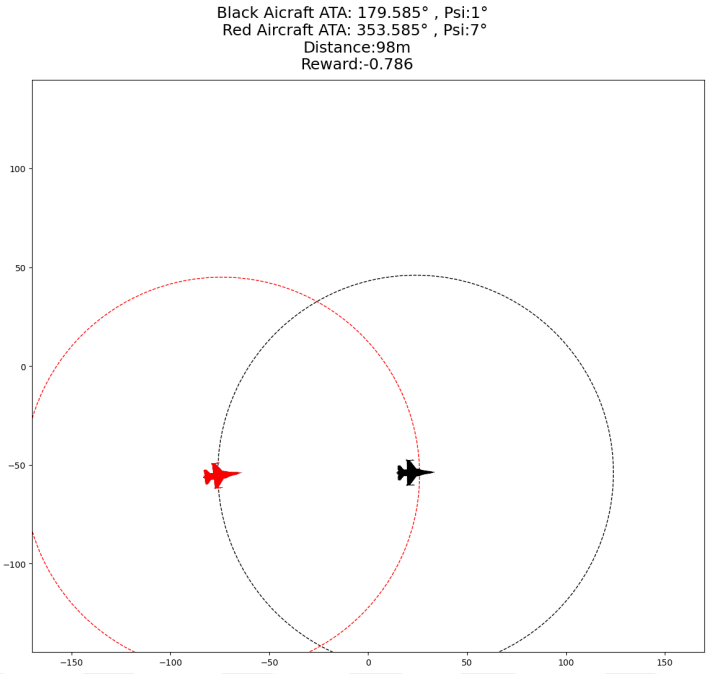
[36] **Kingma, D.P. and Ba, J.** (2014). Adam: A method for stochastic optimization, *arXiv preprint arXiv:1412.6980*.

[37] **Bansal, T.**, **Pachocki, J.**, **Sidor, S.**, **Sutskever, I. and Mordatch, I.** (2018). Emergent Complexity via Multi-Agent Competition, *International Conference on Learning Representations*, `https://openreview.net/forum?id=Sy0GnUxCb`.

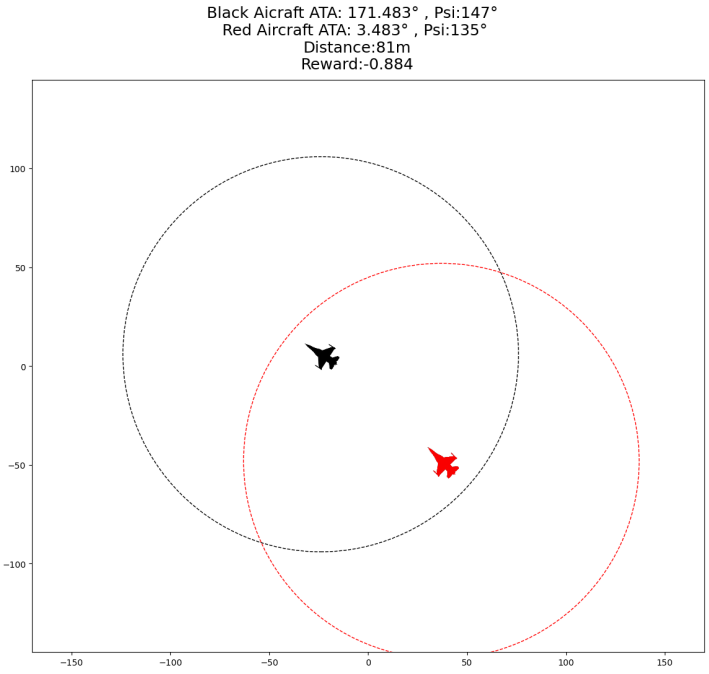
APPENDICES

APPENDIX A : Reward Visualization
APPENDIX B : Air Combat Example Trajectories

APPENDIX A : Reward Visualization

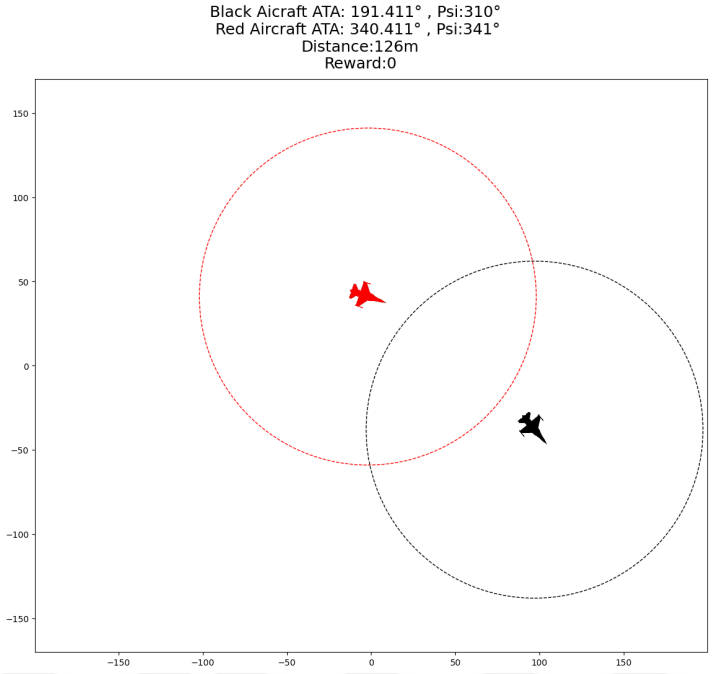


(a)

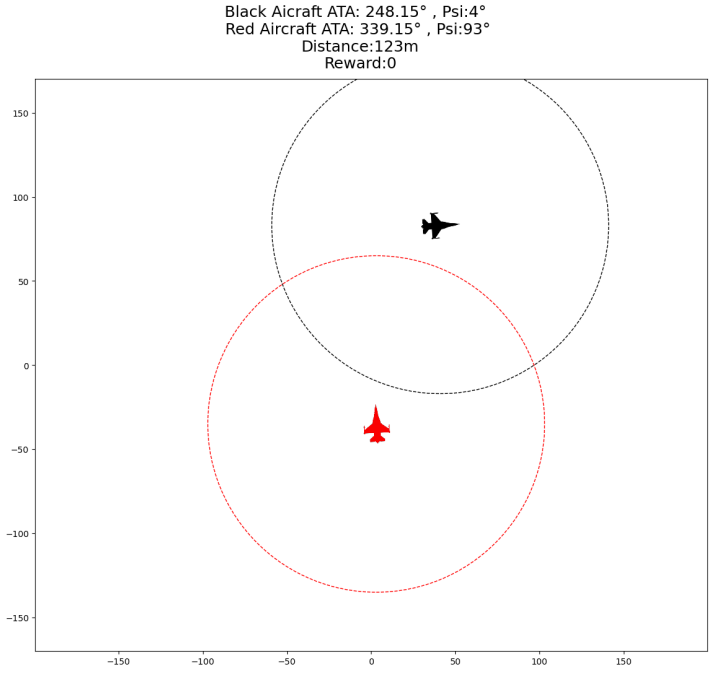


(b)

Figure A.1 : Examples of the negative reward function. Agent receives the negative reward if it is in the enemy shooting range and distance.

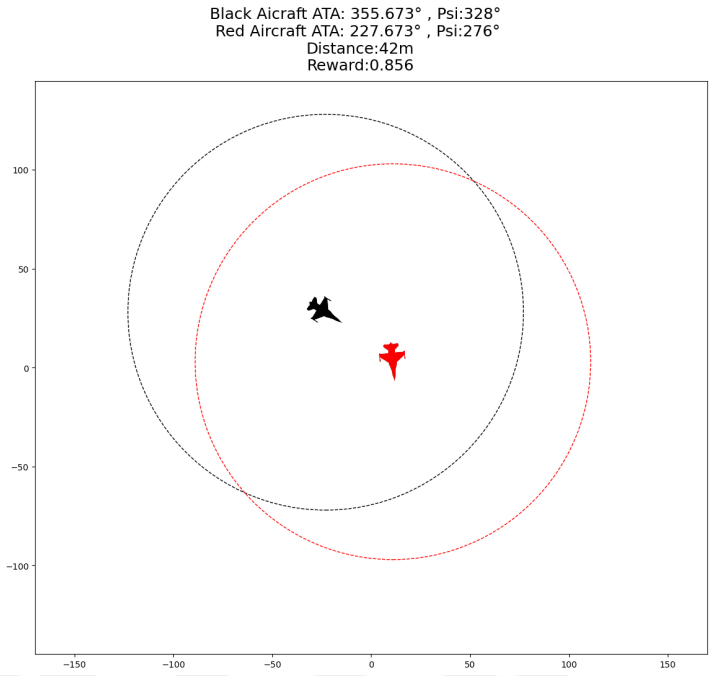


(a)

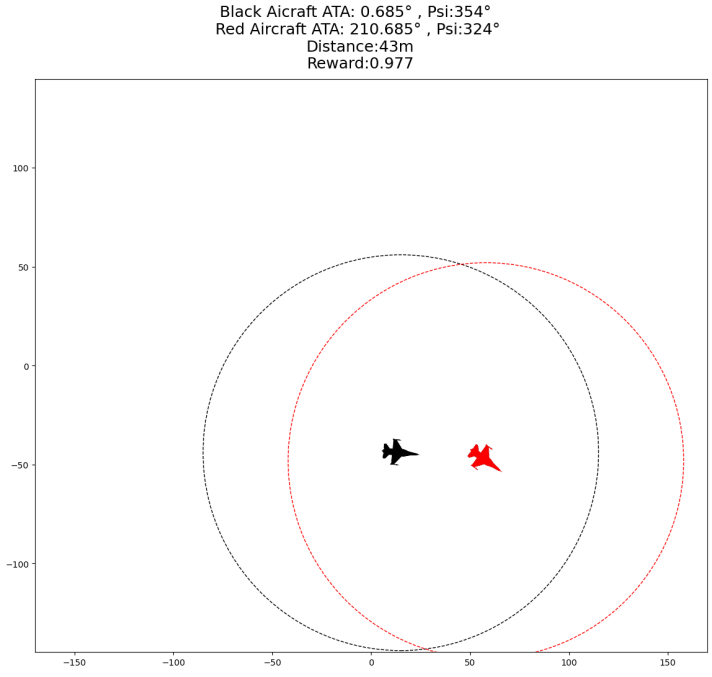


(b)

Figure A.2 : Examples of the zero reward function. In both figures, the agent receives no reward signal because of the shooting distance.



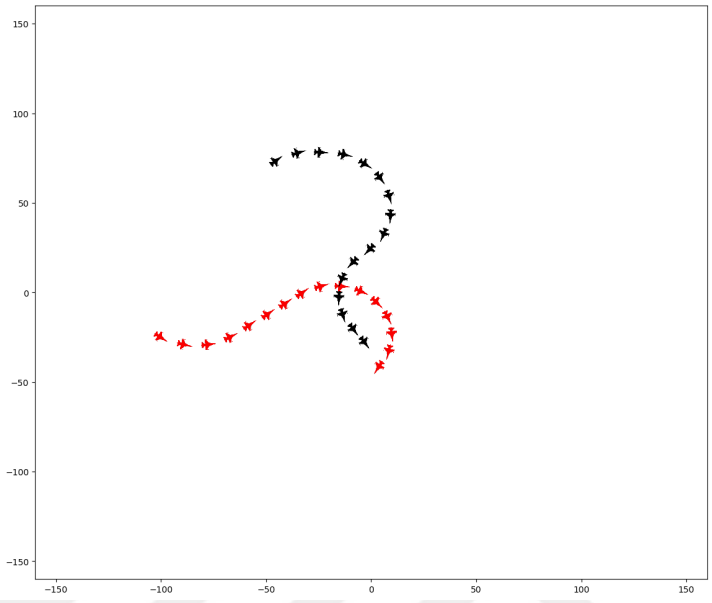
(a)



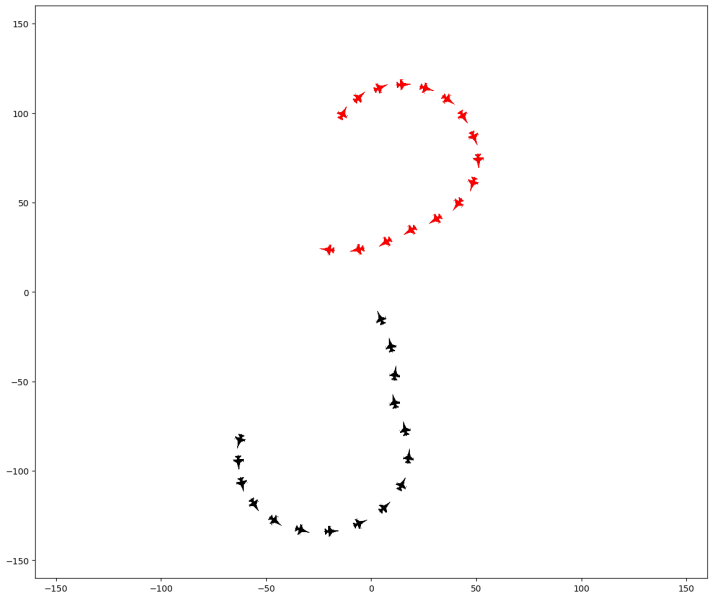
(b)

Figure A.3 : Examples of the positive reward function. The agent receives a positive reward if the enemy is in the agent's shooting range and distance.

APPENDIX B : Air Combat Trajectory Examples

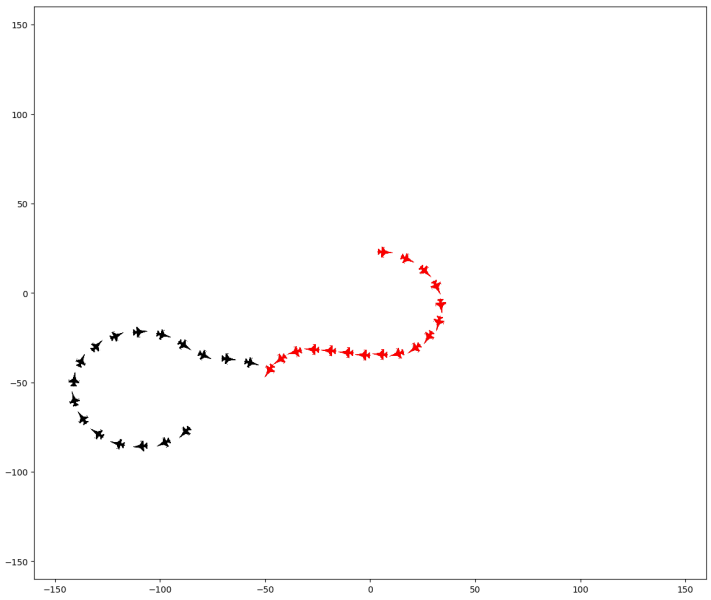


(a)

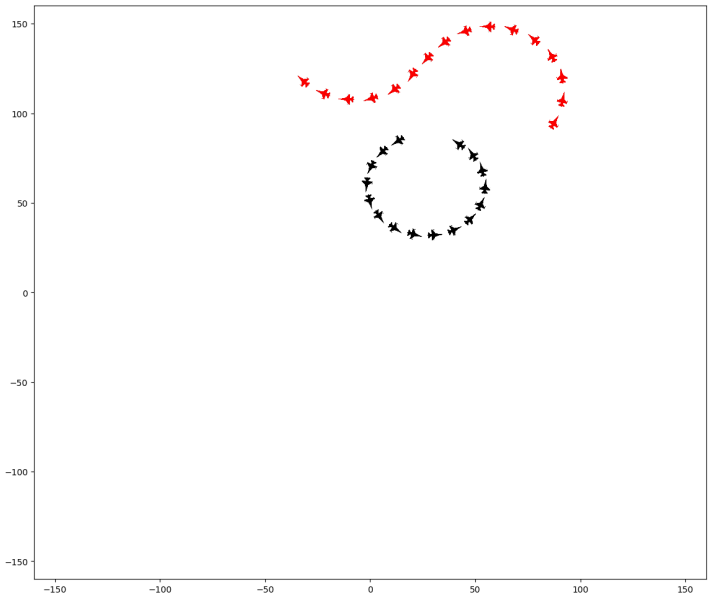


(b)

Figure B.1 : Visualization of the example air combat trajectories 1.



(a)



(b)

Figure B.2 : Visualization of the example air combat trajectories 2.

CURRICULUM VITAE

Name SURNAME: Ahmet Semih TAŞBAŞ

EDUCATION:

- **M.Sc.:** 2023, Istanbul Technical University, Faculty of Computer and Informatics Engineering, Department of Computer Engineering
- **B.Sc.:** 2020, Gazi University, Faculty of Engineering, Department of Computer Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2020-2023 Working at ASELSAN as a research engineer who focuses on reinforcement learning and deep learning.

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

- **Tasbas, A.S.**, Sahin, S.O. and Üre, N.K. (2023). Reinforcement Learning Based Self-play and State Stacking Techniques for Noisy Air Combat Environment, *AIAA SCITECH 2023 Forum*, p.1077.

OTHER PUBLICATIONS, PRESENTATIONS AND PATENTS:

- **Tasbas, A. S.**, & Aydınlı, S. U. (2021, October). 2-D Air Combat Maneuver Decision Using Reinforcement Learning. *International Conference on Engineering and Emerging Technologies (ICEET)* (pp. 1-6). IEEE.
- **Tasbas, A. S.**, Erdal, E., & Özdemir, S. (2019, September). Real-time object and personnel tracking in indoor location. *4th International Conference on Computer Science and Engineering (UBMK)* (pp. 585-590). IEEE.