LONDON'S GLOBAL UNIVERSITY

Multiagent Reinforcement Learning for Cooperation and Coordination under Noisy Communication

Candidate Number: YLHN4

BSc Computer Science

University College London

Supervisor: Mirco Musolesi

Submission Date: May 12, 2023

Abstract

This report provides a comprehensive exploration of Reinforcement Learning (RL) and Multiagent Reinforcement Learning (MARL), starting with a solid foundation in RL principles. The complexities of MARL are then examined, including the significant challenges and prevailing paradigms in the field. Moreover, the report thoroughly surveys the sub-field of communication-based MARL problems, encompassing state-of-the-art algorithms in this domain.

Building on this knowledge, a novel MARL algorithm is introduced to tackle the Guide and Scout Grid world problem, emphasizing effective cooperation and coordination among Scout agents. To assess the algorithm's effectiveness, rigorous testing and analysis are conducted in two fully cooperative instances of the problem: Finding-Treat and Spread both using Binary Symmetric Channel with parameter $\mathcal{p}$. The resulting outcomes are meticulously examined to demonstrate the concept's viability, with careful attention given to refining the associated hyperparameters. The algorithm demonstrates comparable performance to models operating in noise-free conditions when $\mathcal{p} \leq 0.2$. However, its results are less optimal when $\mathcal{p}$ exceeds 0.2.

Contents

1 Introduction 1

2 Reinforcement Learning Foundation 3

2.1 Markov Decision Processes 3

2.2 About Reinforcement Learning 5

2.2.1 Problem Setting 5

2.2.2 Bellman's Equations 7

2.3 Reinforcement Learning Algorithms 7

2.3.1 Value-Based algorithms 8

2.3.2 Model-Free Value-Based algorithms 10

2.3.3 Value-Function Approximation 12

2.3.4 Policy-Based Algorithms 14

3 Multiagent Reinforcement Learning 15

3.1 MARL Problem Formulation 15

3.2 Challenges in MARL 16

3.3 Modern MARL approaches 17

4 Literature Review 19

4.1 Communication problem 19

4.2 Related Works in Communication MARL 20

5 Problem Formulation 24

5.1 Guide and Scout 24

5.2 Finding-Treats 25

5.3 Spread 26

6 Proposed Solution 27

6.1 Preliminaries 27

6.1.1 Deep Q Networks (DQN) 27

6.1.2 Error Detection 30

6.2 Algorithmic Solution ... 32
6.2.1 Training Phase ... 32
6.2.2 Deployment Phase ... 33
7 Numerical Results and Evaluation ... 36
7.1 General Experiment Settings ... 36
7.2 Finding-Treats ... 37
7.3 Spread ... 39
7.4 Summative Analysis ... 41
8 Conclusion ... 43
8.1 Summary of Achievements ... 43
8.2 Evaluation & Future Work ... 43
9 Bibliography ... 45
10 Appendix ... 50
10.1 Project Plan ... 50
10.2 Interim Report ... 53
10.3 Finding-Treats Full Hyperparameter Tuning Results ... 55
10.4 Spread Full Hyperparameter Tuning Results ... 55
10.5 Code Listing ... 56
10.5.1 Runner.py ... 56
10.5.2 DQN.py ... 58
10.5.3 CommAgent.py ... 60
10.5.4 GuideScout.py ... 62
10.5.5 MessageRecoverer.py ... 64
10.5.6 CommChannel.py ... 66
10.5.7 CommGridEnv.py ... 67
10.5.8 Spread.py ... 69
10.5.9 FindingTreat.py ... 70
10.5.10 ErrorDetector.py ... 71

10.5.11 CRC.py 72

10.5.12 AdditiveChecksum.py 73

10.5.13 EvalRunner.py 74

10.5.14 Evaluator.py 75

10.5.15 Seeder.py 76

1 Introduction

Reinforcement Learning (RL) is an extensively studied form of Machine Learning, in which a single intelligent agent learns optimal action policies by interacting with an environment and receiving reward signals. Initially, RL algorithms relied on tabular methods, which had limitations in representing complex state and action spaces. However, the introduction of Deep Learning addressed this issue. Consequently, RL algorithms incorporating Deep Neural Networks gained popularity, particularly Deep Q Networks and the DDPG algorithm. This surge in Deep RL also shed light on Multiagent Reinforcement Learning (MARL), where multiple intelligent agents interact within the same environment. MARL offers various interaction types such as cooperation, competition, or a blend of both, making it applicable to a wide range of compelling applications.

However, MARL introduces additional challenges, which complicates the direct application of existing Deep RL concepts and algorithms to multiagent scenarios. Various approaches have been proposed to tackle these challenges, one of which involves enabling communication among agents. Communication-based approaches in MARL encompass diverse problem dimensions, with one relatively unexplored dimension being the noised communication problem. This problem arises when multiple agents need to achieve their objectives while communicating over a shared noisy communication channel.

Therefore, the goal of this project is to delve into the emerging field of MARL and develop a Multiagent Reinforcement Learning algorithm using Reinforcement Learning based approach. The algorithm aims to enable multiple agents to learn cooperation and coordination while communicating over a discrete binary symmetric channel.

The structure of this report is as follows: Firstly, it discusses the fundamental concepts of Reinforcement Learning. It thoroughly formulates the problem that these learning methods aim to solve and explores a wide range of related RL algorithms. This knowledge is then extended to the multiagent scenario, where it examines the significant challenges in MARL and existing state-of-the-art MARL algorithms.

The subsequent literature review focuses on Communication MARL and highlights a limited set of problem dimensions found in relevant works. After the literature review, the report introduces the

Guide and Scout problem setting, specifically the two instances called Finding-Treats and Spread, which will be used for training and testing.

Once the algorithmic solution to the problem is established, a series of experiments are conducted on the designated Guide and Scout instances. The experiments include suitable hyperparameter tuning, and the results are analysed and presented in numerical figures. To conclude the report, a summary of the contributions made by this project is provided, along with a critical evaluation of the achievements and suggestions for future work.

2 Reinforcement Learning Foundation

The objective of this chapter is to provide a comprehensive introduction to the fundamental principles of Reinforcement Learning problems. These foundational concepts will be further extended to multi-agent scenarios. The chapter will delve into a detailed explanation of notable algorithms, with a primary emphasis on value-based approaches. However, policy-based methods will also be briefly discussed.

2.1 Markov Decision Processes

Markov Decision Processes (MDPs) are well-established models that tackle the challenge of sequential decision-making by abstracting goal-directed learning from interactions (Howard, 1960). Comprising two fundamental elements, MDPs encompass the agent, responsible for making decisions, and the environment, where the agent engages in interactions. It is made up with a 5-tuple $< \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma >$:

- $\mathcal{S}$: The set of states.
- $\mathcal{A}$: The set of actions available to the agent.
- $\mathcal{R}: \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$: The reward function.
- $\mathcal{P}: \mathcal{S} \times \mathcal{A} \mapsto [0,1]$: State transition function.
- $\gamma \in [0,1]$: The discount factor.

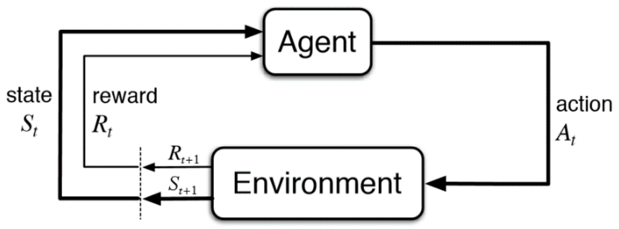


Figure 1. MDP Flow diagram (R. Sutton & Barto, 2018, p. 48)

At each timestep $t \in \mathbb{N}$, the agent would receive a representation of the environment's current state $s_t \in \mathcal{S}$ and a reward $r_t \in \mathbb{R}$. Based on the current state, the agent selects an action $a_t \in \mathcal{A}(s_t)$ where the set of available actions to choose from $\mathcal{A}(s_t)$ depends on the current state s_t. In response to the action, at the next time step $t + 1$, the agent would receive a reward $r_{t+1} \in \mathbb{R}$ and transition to a new state $s_{t+1} \in \mathcal{S}$. There is a probability of those values occurring at time $t + 1$, given values of current state and action:

$$p(s', r|s, a) = \Pr(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) \tag{1}$$

This function characterizes the dynamics of the environment and also satisfies the law of total probability for each choice of $s \in \mathcal{S}$ and $a \in \mathcal{A}$:

$$\sum_{s' \in \mathcal{S}} \sum_{r} p(s', r|s, a) = 1$$

This dynamics can be used to express $\mathcal{P}$, the state transition function $p(s'|s, a)$, which describes the probability for the agent to transition to state s' at timestep $t + 1$ given the state and action at timestep t are s and a respectively:

$$p(s'|s, a) = \Pr(S_{t+1} = s'|S_t = s, A_t = a) = \textstyle\sum_r p(s', r|s, a)$$ [1]

As well as $\mathcal{R}$, the reward function $r(s, a)$, which is the expected reward received given the current state-action pair. This defines the goal which the agent is trying the achieve:

$$r(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_r r \sum_{s' \in \mathcal{S}} p(s', r|s, a)$$

By examining the dynamics of an environment, we can deduce that the probability of specific values for r_{t+1} and s_{t+1} is solely reliant on the current state and action taken by the agent in the preceding time step. This implies that the Markov Property holds, which states that the future state of the environment is solely dependent on the current state and action, without any dependence on the past. Any scenario adhering to this property is deemed Markovian, as it satisfies the fundamental principle that the present fully determines the future without any need for historical information.

An MDP may have a finite number of timesteps, which would end in a terminal state and are called episodic. It may on the other hand have an infinite number of timesteps and goes on without a limit, and these are known as continuous.

[1] The $\sum_r$ notation indicates a summation over all possible reward values in the reward set R, this is omitted to avoid confusion with the reward function notation $\mathcal{R}$

2.2 About Reinforcement Learning

2.2.1 Problem Setting

The planning problem, within the context of a problem involving an underlying MDP, refers to the task of identifying the optimal policy that maximizes the expected return value. This problem can be effectively addressed by utilizing Reinforcement Learning (RL) techniques.

Return is denoted as G_t which is the cumulative sum of rewards. More precisely, we are maximizing the expected value of the discounted return with discount factor γ to consider continuous tasks which may have infinite t.

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots = \sum_{i=0}^{\infty} \gamma^i R_{t+i}$$

$$G_t = R_t + \gamma G_{t+1}$$

(2)

The expectation of return is represented as state-value functions, which define the expected cumulative reward that an agent could receive at a particular state while following certain agent behaviours governed by policies. Policies are functions that map from state s to the probability of selecting each action $a \in \mathcal{A}(s)$, denoted as $\pi(s|a)$. Thus the state-value function at state s while following policy π has the following definition:

$$v_\pi(s) = \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} \gamma^i R_{t+i} \,|S_t = s\right]$$

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

(3)

The value of a state can also be considered in conjunction with a specific action instead of an expectation over all possible actions, called action-value functions, which are more commonly used in many RL algorithms and can also be used to express state-value functions:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s)\, q_\pi(s,a)$$

In the case where the policy is deterministic, only one of the actions would have $\pi(a|s) = 1$ and others would have 0 probability. Thus we could write $\pi(s)$ which represents the action being yielded by the policy at state s. Using this definition, state-value functions can be expressed more easily using action-value functions:

$$v_{\pi}(s) = q_{\pi}(s, \pi(s))$$

For simplicity of explanation, whenever a policy is mentioned in the following sections, it is assumed to be deterministic, though the same principles apply to non-deterministic policies as well.

Since the goal is to maximize the expected return, RL can be formulated as searching for a policy that is better than any other policy. More formally, whether one policy is better than the other is defined as follows:

$$\pi' \geq \pi \text{ } if \text{ } and \text{ } only \text{ } if \text{ } v_{\pi'}(s) \geq v_{\pi}(s) \text{ } for \text{ } all \text{ } s \in \mathcal{S}$$

It can be stated that a policy π' is superior to another policy π if and only if the state-value function resulting from following π' produces a value that is greater than or equal to the state-value function obtained from following π across all states.

This definition can then be extended and slightly rearranged to obtain the policy improvement theorem:

$$If \text{ } for \text{ } all \text{ } s \in \mathcal{S} \text{ } q_{\pi}\big(s, \pi'(s)\big) \geq v_{\pi}(s),$$

$$then \text{ } v_{\pi'}(s) \geq v_{\pi}(s) \text{ } for \text{ } all \text{ } s \in \mathcal{S}, \text{ } thus \text{ } \pi' \geq \pi$$

(4)

There may be more than one optimal policy, all of which are denoted as π_* and share the same optimal state-value function v_*:

$$v_*(s) = \max_{\pi} v_{\pi}(s) \text{ } for \text{ } all \text{ } s \in \mathcal{S}$$

They also share the same optimal action-value function q_*:

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a) \text{ } for \text{ } all \text{ } s \in \mathcal{S}, \mathrm{a} \in \mathcal{A}$$

Which can be rearranged to express v_*:

$$v_*(s) = \max_{a \in \mathcal{A}(\mathrm{s})} q_*(s, a)$$

(5)

Any greedy policy with respect to these optimal value functions is the optimal policy.

2.2.2 Bellman's Equations

Bellman's Equation is used to express $v_\pi(s)$ mathematically (R. Sutton & Barto, 2018, p. 59):

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\
&= \mathbb{E}_\pi[R_t + \gamma G_{t+1} | S_t = s] && \textit{From (2)} \\
&= \mathbb{E}_\pi[R_t | S_t = s] + \mathbb{E}_\pi[\gamma G_{t+1} | S_t = s] \\
&= [\textstyle\sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) * r] + [\sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) * (\gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'])] \\
&= \textstyle\sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) * (r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']) \\
&= \textstyle\sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) * [r + \gamma v_\pi(s')] && \textit{From (3)}
\end{aligned}
\tag{6}
$$

Notably, v_* being a state-value function of optimal policies must satisfy this condition as well. In addition, it can be expressed without reference to any policy, and this equation is known as Bellman's optimality equation (R. Sutton & Barto, 2018, p. 63):

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_*(s, a) && \textit{From (5)} \\
&= \max_a \mathbb{E}_*[G_t | S_t = s, A_t = a] && max_a \textit{ shorthand for } max_{a \in \mathcal{A}(s)} \\
&= \max_a \mathbb{E}_*[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] && \textit{From (2)} \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] && \textit{Optimal Expected return } \mathbb{E}_*[G_{t+1}] = \mathbb{E}[v_*(S_{t+1})] \\
&= \max_a \textstyle\sum_{s', r} p(s', r|s, a)[r + \gamma v_*(s')]
\end{aligned}
\tag{7}
$$

Finding the optimal policy by solving the optimality equation can be one approach, but it is seldom practical in real-world scenarios. This is because solving the optimality equation involves an exhaustive search across all possibilities, demanding significant computational resources. Additionally, in practice, the agent typically lacks knowledge about the dynamics of the environment from the outset, as will be elaborated in the subsequent sections.

2.3 Reinforcement Learning Algorithms

A planning problem can be divided into two sub-tasks: Prediction and Control. In the context of reinforcement learning, one way to address the prediction task is to compute the value function for a given policy, while the control task involves enhancing policies and approximating optimal policies. Algorithms that fall into the Value-Based category aim to accomplish these objectives. On the other hand, Policy-Based algorithms directly search within the policy space to find the optimal policy.

Reinforcement learning algorithms can generally be categorized into two paradigms: Model-Based and Model-Free. However, the primary focus of this dissertation will be on Model-Free algorithms, which are designed to learn and improve policies without relying on a pre-defined model of the environment.

2.3.1 Value-Based algorithms

Dynamic programming has proven to be an effective approach for efficiently searching for an optimal policy. By utilizing dynamic programming, value functions can be computed, allowing for a more organized and systematic exploration of improved policies. This iterative process continues until no further enhancements can be made, at which point the optimal policy is discovered. Two well-known value-based algorithms are Policy Iteration and Value Iteration, both of which requires knowledge of the dynamics of the environment (1) and follows the structure of Generalized Policy Iteration (GPI):

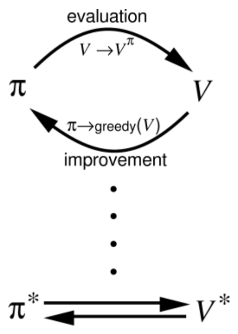


Figure 2. Generalized Policy Iteration (R. Sutton & Barto, 2018, p. 86)

The iterations involve 2 steps, Policy Evaluation and Policy Improvement, which correspond to the Prediction and Control tasks for the planning problem. To compute the state value function, the idea is by turning Bellman's equation into update rules to compute the value function given a policy, thus utilizing dynamic programming methods:

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) * [r + \gamma V(s')]$$ *From (6) transformed for Policy Iteration*

$$V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$$ *From (7) transformed for Value Iteration*

These updates are iterated until convergence, which is when the maximal difference in values between the newly computed value function and the previous value function falls below a certain threshold.

The improvement of policies is based on the policy improvement theorem (4). It is achieved by following a greedy policy π' with respect to the value function v_π computed at the policy evaluation step using policy π, which always selects the action that gives the highest value at the current state:

$$\pi'(s) = argmax_a \sum_{s',r} p(s',r|s,a)\,[r + \gamma v_\pi(s')]$$

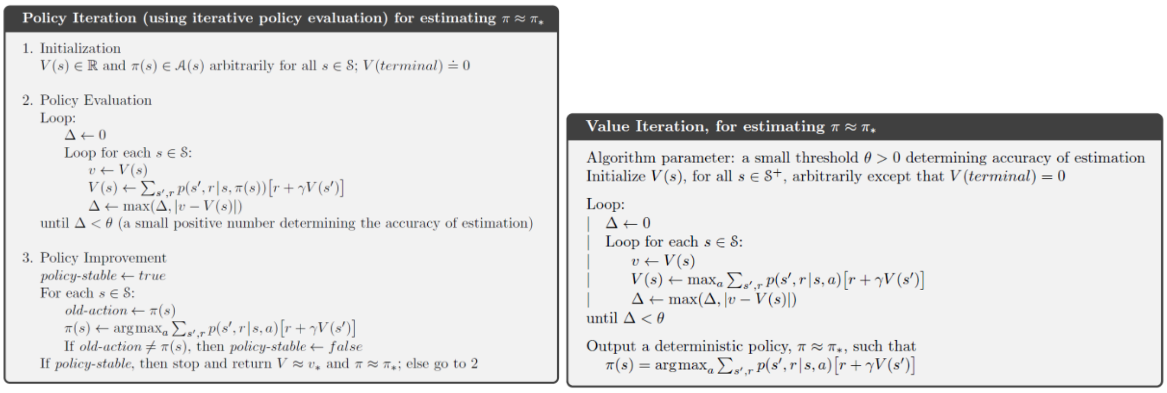
Or equivalently:

$$\pi'(s) = argmax_a q_\pi(s,a) \tag{8}$$

And thus, the action-value function is:

$$q_\pi(s, \pi'(s)) = \max_a q_\pi(s,a)$$

By employing this construction, the updated greedy policy π' is required to possess an equivalent or superior value compared to the initial policy π, as it consistently yields the highest value based on the present value function across all states. Therefore, according to its definition, this newly formulated greedy policy must be at least as advantageous as the original policy, if not superior. In the event that the greedy policy π' is equally beneficial but not superior to the original policy π, it indicates the discovery of an optimal policy, where both π' and π represent optimal policies.



```
Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(terminal) \doteq 0$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))[r + \gamma V(s')]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ true
   For each $s \in \mathcal{S}$:
       old-action $\leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
       If old-action $\neq \pi(s)$, then policy-stable $\leftarrow$ false
   If policy-stable, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

```
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|   $\Delta \leftarrow 0$
|   Loop for each $s \in \mathcal{S}$:
|       $v \leftarrow V(s)$
|       $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
|       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
```

Figure 3. Policy Iteration and Value Iteration algorithm(R. Sutton & Barto, 2018, p. 81,83)

However, these algorithms require knowledge of the actual dynamics of the environment and are rarely useful in practice as this information is usually unknown. Nonetheless, the concepts introduced by these algorithms such as GPI serve as important foundations for many more sophisticated value-based RL algorithms.

2.3.2 Model-Free Value-Based algorithms

For Model-Free algorithms, the goal is to solve the planning problem while the dynamics of the MDP are unknown to the agent. There exist two methods for Model-Free Value-Based algorithms, the Monte Carlo (MC) method, and the Temporal Difference (TD) method, both of which require the agent to interact directly with the environment, and learn directly from sampled episodes of experiences. These methods also follow the GPI, but this time for the Policy Evaluation step, since the dynamic of the MDP is unknown, only empirical estimation of v_π can be computed based on the agent's experiences generated with policy π.

The idea behind MC methods is straightforward, the estimation of v_π is accomplished by calculating the average return obtained at each state within the sampled episodes of experiences. The specific manner in which the averaging is performed varies depending on whether it is First-Visit MC or Every-Visit MC. However, in both cases, this averaging computation is initiated only when the terminal state is reached, and the calculation is carried out by traversing backwards in time through the stored experiences. In actual implementation, q_π is estimated instead, as Policy Improvement with v_π requires knowledge of the dynamics of the MDP (8), which is unknown in model-free scenarios. It is worth mentioning that there are several issues associated with MC methods, with the major ones being it only work for episodic tasks as the average calculation requires a terminal state, and these algorithms have very slow convergence when returns have high variance.

For TD methods, instead of waiting until the termination of an episode before updating, the update is done at every time step, which means that these methods are not restricted to episodic tasks. TD methods use $V(S)$ as a biased estimator for v_π. These methods start with an arbitrary $V(S)$ and attempt to update this function towards the actual return through sampled episodes, where α is the step size for the update:

$$V(S_t) \leftarrow V(S_t) + \alpha(\boldsymbol{G_t} - V(S_t))$$

Since G_t is not known, TD methods use a technique called bootstrapping to generate an estimated return, referred to as the TD target. An illustration of this approach is presented below, which demonstrates the update rule for the TD(0) algorithm. In this case, TD(0) is bootstrapping one step ahead, where the TD target (in bold) is the sum of the actual immediate reward and the discounted estimated value of the next state:

$$V(S_t) \leftarrow V(S_t) + \alpha(\mathbf{R_{t+1}} + \boldsymbol{\gamma}\mathbf{V}(\mathbf{S_{t+1}}) - \mathrm{V}(\mathrm{S_t}))$$

Bootstrapping is not only limited to one step lookahead, such technique can also be applied to multiple steps TD(λ) (R. S. Sutton, 1988, pp. 33–36), which takes information at multiple time steps and is weighted using λ, hence making this method more robust to change in environment and less biased. After the empirical estimation V has been computed, the Policy Improvement step is carried out, as usual, using V instead of v_π.

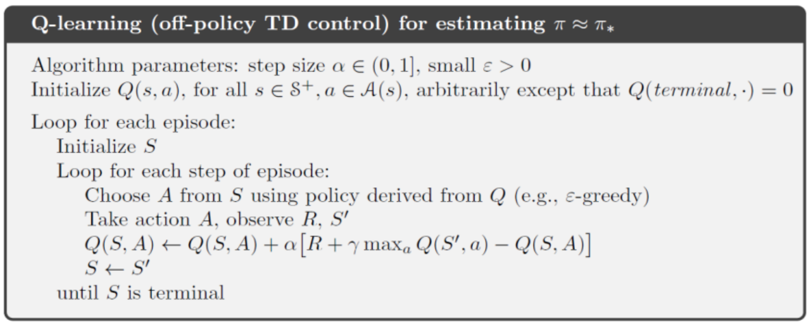
Some notable TD-Based algorithms are SARSA(Rummery & Niranjan, 1994, p. 6) and Q-Learning(Watkins & Dayan, 1992). Once again both algorithms estimate for q_π instead of v_π since the dynamics are unknown, and the estimator is denoted as $Q(S, A)$. The update rules for these algorithms are as follow:

$$\text{SARSA:} \qquad Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \boldsymbol{Q(S_{t+1}, A_{t+1})} - Q(S_t, A_t)]$$

$$\text{Q-Learning:} \qquad Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \boldsymbol{\max_a Q(S_{t+1}, a)} - Q(S_t, A_t)] \tag{9}$$

What differentiates the two TD-Based algorithms is that SARSA is known as an On-Policy algorithm and Q-Learning is Off-Policy. For On-Policy algorithms, the same policy is used for policy evaluation and policy improvement. As for Off-Policy algorithms, two policies are used, a Behaviour Policy for policy evaluation and a Target policy for policy improvement.

Q-Learning is an off-policy algorithm because the action value function is updated with the action of maximal q-value as shown by the bolded section, thus the agent is learning from a behavioural policy that is greedy while the target policy may not be necessarily greedy at all. SARSA is on-policy because its q-value is updated using $Q(S_{t+1}, A_{t+1})$ where A_{t+1} is generated from its current policy π. It is important to mention that if the current policy is also a greedy policy, both algorithms can be considered On-Policy. However, such an algorithm would not learn effectively due to the absence of exploration.



```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

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

Figure 4. Algorithm for Tabular Q-Learning(R. Sutton & Barto, 2018, p. 131)

The major drawback to these algorithms is that they are tabular. Taking Q-Learning as an example, the algorithm requires the storage of q-values for each of the state-action pairs constructing a q-table for direct lookup and storage. However, for problems that have very larger state spaces, such as Go which has approximately 10^{171} legal states (Tromp & Farnebäck, 2007, p. 21) or some robotic tasks which may have continuous state spaces. This is too much to be stored inside the memory as lookup tables.

Drawing the attention back to the GPI, after having evaluated the policy by computing an empirical estimation of v_π, a soft policy is used in the Policy Improvement step to encourage the exploration of different policies. This is because using a greedy policy may cause improvements to be stuck at local maxima. An example of a soft policy is the ϵ-greedy policy, for which the policy improvement theorem still holds(Silver, 2015, p. 14):

$$\pi_\epsilon(s)\begin{cases} argmax_a q_\pi(s,a) & probability\ 1-\epsilon \\ a\sim\text{Uniform}(\mathcal{A}(\text{s})) & probability\ \epsilon \end{cases}$$

A common technique that is used along with ϵ-greedy policy is known as ϵ-decay, this is a method to schedule ϵ values such that the ϵ value would decrease further in the training, thus focusing on exploration at the start of the training and prioritizing exploitation of discovered policies towards the end of the training. It is defined as follows, where ϵ_{start} and ϵ_{end} indicate the desired starting and ending values for ϵ, λ controls the rate of decay and $epsisode$ indicates the number of training episodes that have occurred.:

$$\epsilon = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) * e^{-\frac{episode}{\lambda}}$$

(10)

2.3.3 Value-Function Approximation

To account for larger state spaces, the idea of value-function approximation is to use a lower-dimensioned parametric approximator as a way of representing states such that a small number of tuneable parameters θ can be used to fit the approximator to the true value of each state. The fitted approximator would also be able to generalize unseen states and can then be used to extrapolate or interpolate over unseen states:

$$\hat{v}_\theta(s) \approx v_\pi(s)$$

$$\hat{q}_\theta(s,a) \approx q_\pi(s,a)$$

The goal is to find parameter vector θ that minimizes the error between the approximator $\hat{v}_\theta(s)$ and the ground truth v_π. For simplicity of explanation, a common metric used to evaluate this error is expected mean squared error (MSE), where the expectation is used to account for the MSE across all states:

$$J(\theta) = \frac{1}{2}\mathbb{E}_\pi\left[\left(\boldsymbol{v_\pi(S)} - \hat{v}_\theta(\mathrm{S})\right)^2\right]$$

$$\theta^* = argmin_\theta \frac{1}{2}\mathbb{E}_\pi\left[\left(\boldsymbol{v_\pi(S)} - \hat{v}_\theta(\mathrm{S})\right)^2\right]$$

The function $J(\theta)$, which depends on the parameter vector θ, is a differentiable function and is commonly referred to as the loss function. The objective is to minimize this function and locate its local minima θ^*. The $\frac{1}{2}$ is a constant added to cancel out the 2 that comes out when differentiating $J(\theta)$.

Gradient descent is used to find $argmin_\theta J(\theta)$. This is a common optimization algorithm that finds optima by iteratively updating θ in the negative direction to the gradient of $J(\theta)$ with respect to θ (denoted as $\nabla_\theta J(\theta)$). We could also use a variation of gradient descent called Stochastic Gradient Descent (SGD) that takes random samples of the gradient to approximate the true gradient of $J(\theta)$ which often converges faster than gradient descent:

$$\Delta\theta = -\alpha\nabla_\theta J(\theta) \qquad \text{α is the step size}$$

$$= -\alpha E_\pi\left[\left(\boldsymbol{v_\pi(S_t)} - \hat{v}_\theta(\mathrm{S_t})\right)\nabla_\theta \hat{v}_\theta(\mathrm{S_t})\right] \qquad \textit{By chain rule}$$

$$= -\alpha(\boldsymbol{v_\pi(S_t)} - \hat{v}_\theta(\mathrm{S_t}))\nabla_\theta \hat{v}_\theta(\mathrm{S_t}) \qquad \textit{From SGD assumption}$$

Here α represents the step size which is a tunable hyperparameter that decides how much θ is being updated at each step. The same principles apply to $\hat{q}_\theta(s,a)$ approximators:

$$J(\theta) = \frac{1}{2}\mathbb{E}_\pi\left[\left(\boldsymbol{q_\pi(S,A)} - \hat{q}_\theta(\mathrm{S,A})\right)^2\right]$$

$$\Delta\theta = -\alpha\left(\boldsymbol{q_\pi(S,A)} - \hat{q}_\theta(\mathrm{S,A})\right)\nabla_\theta \hat{q}_\theta(\mathrm{S,A})$$

Since ground truth $v_\pi(s)$ and $q_\pi(s,a)$ are unknown in model-free scenarios, they can be estimated using model-free methods such as MC and TD as explained in the previous section. Taking Q-Learning as an example, then:

$$J(\theta) = \frac{1}{2}\mathbb{E}_\pi\left[\left(\boldsymbol{R_{t+1} + \gamma \max_a \hat{q}_\theta(S_{t+1}, a)} - \hat{q}_\theta(\mathrm{S_t, A_t})\right)^2\right] \qquad \textit{From (9)}$$

$$\Delta\theta = -\alpha\left(\boldsymbol{R_{t+1}} + \boldsymbol{\gamma}\max_{\boldsymbol{a}}\boldsymbol{\hat{q}_{\theta}(S_{t+1}, a)} - \hat{q}_{\theta}(\mathrm{S_t}, \mathrm{A_t})\right)\nabla_{\theta}\hat{q}_{\theta}(\mathrm{S_t}, \mathrm{A_t})$$

(11)

The values of $\hat{q}_{\theta}(\mathrm{S_t}, \mathrm{A_t})$ and $\nabla_{\theta}\hat{q}_{\theta}(\mathrm{S_t}, \mathrm{A_t})$ are determined by the function approximator used, which can take various forms. With the success of deep neural networks (DNN), methods involving DNN approximators became increasingly popular, though linear model approximators such as Linear MC and Linear TD approximators are also widely used.

After having approximated $q_{\pi}(s, a)$ with $\hat{q}_{\theta}(s, a)$, we then can use a soft policy (e.g., ϵ-greedy) for Policy Improvement and hope for the best to obtain a policy close to the optimal. Unfortunately, function approximation methods do not adhere to the policy improvement theorem, and their convergence is not guaranteed, even in the case of linear approximators. Therefore, additional techniques are necessary to address this issue. One notable approach is the Deep Q Network (see Chapter 6.1) which uses Q-Learning and Deep Neural Networks as the function approximator along with Target Networks and Experience replay to stabilise training.

2.3.4 Policy-Based Algorithms

Since planning problems are about finding optimal policies of an arbitrarily given MDP, another approach would be to search directly in the policy space of that MDP for an optimal policy. Within this set of algorithms, there are gradient-free algorithms such as simulated annealing and evolutionary algorithms and gradient-based algorithms such as REINFORCE and Policy Gradients that rely on gradient-ascent optimization to find local optima within the search space. An important advantage of this method over value-based methods is that they are effective in continuous action space whereas value-based methods are restricted to discrete action space.

However, these policy-gradient methods have been shown in practice to have very slow convergence and incur high variance, therefor actor-critic methods were introduced to tackle these issues. Actor-Critic methods were often described as the TD version of policy gradient methods, and they consist of an actor and a critic. The role of an actor is to decide the best action to be taken given the current state, and the critic would evaluate the actor on how good the action was taking the current into account and returning an action value, of which the actor would adjust against. The actor would learn through gradient-based algorithms whereas the critics would be a value-function approximator learned using TD methods.

3 Multiagent Reinforcement Learning

In this chapter, the main focus is on presenting the objectives of Multiagent Reinforcement Learning (MARL), the key challenges involved in MARL, and providing a broad overview of the major paradigms and their associated state-of-the-art algorithms in MARL.

3.1 MARL Problem Formulation

The MDP formulation in single agent scenario can be extended and generalized in a multi-agent setting known as Markov games (sometimes referred to as Stochastic Games) and is defined with the following tuple (Littman, 1994, pp. 1–2; Yang & Wang, 2021, pp. 16–17), which generalises to MDPs when $\mathcal{N} = 1$:

$$< \mathcal{N}, \mathcal{S}, \{\mathcal{A}^i\}_{i \in \{1,\dots,\mathcal{N}\}}, \mathcal{P}, \{\mathcal{R}^i\}_{i \in \{1,\dots,\mathcal{N}\}}, \gamma >$$

- $\mathcal{N}$: Number of agents in the environment.
- $\mathcal{S}$: Set of states shared across all agents.
- $\mathcal{A}^i$: Set of actions available to agent i, the cartesian product with all action sets is represented as $\mathbb{A} = \mathcal{A}^1 \times \mathcal{A}^2 \times \dots \times \mathcal{A}^N$.
- $\mathcal{P}: \mathcal{S} \times \mathbb{A} \mapsto PD(\mathcal{S})$: The state transition function, where $PD(\mathcal{S})$ is the set of discrete probabilities over the set $\mathcal{S}$.
- $\mathcal{R}^i: \mathcal{S} \times \mathcal{A}^i \mapsto \mathbb{R}$: The reward function of agent i.
- $\gamma \in [0,1]$: The discount factor.

The interactions are rather similar to MDPs. At each timestep $t \in \mathbb{N}$, all agents would receive a shared representation of the environment's current state $s_t \in \mathcal{S}$ and each agent i would receive an individual reward $r_t^i \in \mathbb{R}$ from the reward function $\mathcal{R}^i$. Based on s_t, each agent selects their action $a_t^i \in \mathcal{A}^i(s_t)$ based on their policy π^i simultaneously to generate a joint action $\mathbb{A}$. In response to this joint action, at the next time step $t + 1$, each agent would receive an individual reward $r_{t+1}^i \in \mathbb{R}$ and would be in a new shared state $s_{t+1} \in \mathcal{S}$ based on the state transition function $\mathcal{P}$.

Apart from interacting with the base environment, agents in multi-agent systems also engage with other agents, who are considered part of the environment. This expands the scope of interaction, introducing various forms of engagement. One such form is competition, where agents strive to outperform other agents and achieve individual goals. Another form is cooperation, where agents collaborate to accomplish a shared objective. Additionally, interactions can involve a combination of both competition and cooperation, creating a mixture of dynamics within the multi-agent system.

In the context of cooperative settings, which is the primary focus of this project, the team reward function is denoted as $\hat{\mathcal{R}}$. In this context, the reward function $\mathcal{R}^i$ for an individual agent can be expressed as a combination of the team reward function and the agent's local reward function $(1-\beta)\hat{\mathcal{R}} + \beta\tilde{\mathcal{R}}^i$, where $\tilde{\mathcal{R}}^i$ represents the agent's local reward function, and β is a parameter for local reward weighting. Consider the case where $\beta = 0$ and thus agents would seek to maximize the discounted cumulative sum of team reward $\hat{R}$ (= discounted team return $\hat{G}_t$). Now let $\Pi: \mathcal{S} \mapsto \mathbb{A} = \{\pi^i\}_1^N$ denote the set of all agents' policies, then the state value function can be defined as following in contrast to MDPs:

$$v_\Pi(s) = \mathbb{E}_\Pi\left[\sum_{i=0}^{\infty} \gamma^i \hat{R}_{t+i} \,\middle|\, S_t = s\right]$$

$$v_\Pi(s) = \mathbb{E}_\Pi[\hat{G}_t | S_t = s]$$

Because of its ability to support diverse forms of interaction, MARL naturally finds application in a broader range of scenarios such as the AI Economist(Zheng et al., 2020) which strives to improve equality and productivity with AI-Driven Tax policies; in multiagent video games such as StarCraft (J. Foerster, Farquhar, et al., 2018; Peng et al., 2017; Vinyals et al., 2019); the study of sequential social dilemmas which require multiple agents to learn policies that implement their strategic intentions (Leibo et al., 2017); in traffic light control systems(Bakker et al., 2010); playing a modified version of team-based hide and seek(Baker et al., 2020, pp. 4–10) to name a few.

3.2 Challenges in MARL

A trivial algorithm for solving MARL problems is known as the Independent Q-Learning algorithm. This algorithm trains all the agents with Deep Q-Learning individually and each agent would learn their Q function that only conditions its states and actions, essentially doing $\mathcal{N}$ separate single-agent reinforcement learnings.

However, this method of learning has been shown to perform poorly in multi-agent settings. One of the major causes is that in the multi-agent context, the policy of the individually trained agents changes at training, which causes non-stationarity of the environment in other agents' perspectives since the agents themselves are also part of the environment (Matignon et al., 2012, pp. 7–8).

This is one of the major challenges for MARL in general as the non-stationarity in the agent's observations breaks the Markov assumption that governs convergence of Q-Learning(Laurent et al., 2011, p. 5), of which agents would end up in an infinite loop of adapting to other agent's policies. This inherent property of MARL problems also prevents naïve approaches to experience replay, since the non-stationarity of stored states defeats the purpose of having a replay memory in the first place, which is used to stabilize the training of deep networks.

The credit-assignment problem, extensively addressed in (Chang et al., 2004) presents yet another formidable challenge. Within the realm of reinforcement learning (RL), this problem pertains to the difficulty faced by an agent in determining which of its previous actions contributed to the rewards it receives. However, in the context of multi-agent scenarios, the credit-assignment problem takes on a different dimension, focusing more on the allocation of responsibilities for the collective performance of the agents. Consider a fully cooperative scenario with a team reward function $\hat{\mathcal{R}}$, when the team receives a reward based on this shared reward function, individual agents encounter difficulties in discerning whether this reward is solely attributable to their own actions or if they have made any contributions towards this collective reward.

3.3 Modern MARL approaches

Up to date, the methods of MARL algorithms can be separated into 3 different paradigms (Papoudakis et al., 2019, pp. 3–5): centralised critique methods, decentralised learning methods, and communication methods.

Centralised critique approaches are the most popular and have been widely researched. Commonly for this approach, an actor-critic architecture is adopted and hence is mostly policy-based. The critics have access to all agents' states and actions and are trained in a centralised fashion, the actors on the other hand are kept to the agents themselves. Since the states and actions of all agents are known by the critics at training, the environment no longer appears to be non-stationary from the perspective of individual agents and training is stabilized through this method.

Some state-of-the-art centralised learning algorithms include the COMA policy gradients algorithm(J. Foerster, Farquhar, et al., 2018) that uses the actor-critic method as well as a counterfactual baseline to tackle multi-agent credit assignment problem; the MADDPG algorithm as the multiagent counterpart to DeepMind's DDPG(Lillicrap et al., 2019) algorithm, where opponent modelling method is used to learn approximate models of other agents, which is then augmented to the critic(Lowe et al., 2020); the MAPPO algorithm as a multiagent extension to the PPO algorithm (Schulman et al., 2017), which achieved better or comparable results when compared to the MADDPG algorithm with comparable sample efficiency (Yu et al., 2022, p. 4); and last but not least the QMIX(Rashid et al., 2018) algorithm.

The second paradigm is the Decentralised Learning technique, which is where each agent learns independent policy based on its own local experiences, without any form of centralised coordination with other agents at training. One feasible direction is by amending the Independent Q Learning algorithm in tackling the non-stationarity issue. Some notable algorithms include hysteretic Q Learning (Matignon et al., 2007); the Hyper-Q algorithm which also utilised Opponent modelling methods to estimate other agent's policies through Bayesian Inference and inputs these inferred policies as parameters to the Q function of the agent being trained to overcome the non-stationarity (Tesauro, 2003); and the deep neural network extension to Hyper-Q that conditions agent's value function on a fingerprint to disambiguates the age of the data sampled from the replay memory, essentially indexing the experiences to allow experience replay function in multiagent scenarios (J. Foerster, Nardelli, et al., 2018).

Finally, the communication method is the approach that enables agents to share information regarding their states, actions, or other pertinent details through communication channels. This project is situated within this paradigm and will delve into its details and implications in greater detail.

4 Literature Review

The objective of this chapter is to explore the various aspects of the communication problem within the framework of Multiagent Reinforcement Learning (MARL). It will provide an overview of the existing approaches in communication MARL and conduct a comprehensive survey of relevant literature pertaining to this project.

4.1 Communication problem

Prior to examining existing works on communicative MARL algorithms, it is essential to identify the key dimensions of the problem that must be addressed when developing such algorithms. By delineating these dimensions, the specific contribution of this project can be clearly defined and compared to the efforts made by other works that tackle similar challenges.

Adding onto what has been formalized in (Zhu et al., 2022, pp. 3–8), relevant dimensions have been selected and rephrased as the following questions:

- **Task Type**: Is the task cooperative, competitive, or a mixture of the two?
- **Training scheme**: Is the training method centralised or decentralised?
- **Communicatee type**: Do agents communicate directly with each other or through a proxy? Is there a limited range to this communication?
- **Communication policy**: Whom can each agent communicate with? Is it Full Communication (with all other agents), Partial communication (limited number of other agents) or individually controlled (agent actively decides whether to communicate or not)?
- **Constraints**: Does the communication channel have limited bandwidth? Is the channel noisy? Do agents share the communication channel (relevant in the case of competitive/mixed tasks)?
- **Nature of Communication Channel**: Does the channel carry discrete or analogue signals?

With these problem dimensions, the contribution of this project can be classified as a decentralized training, direct communication, and partial communication algorithm under the constraint of having a noisy discrete communication channel, for the cooperative and coordinative behaviour of multiple agents.

4.2 Related Works in Communication MARL

The topic of communication in MARL isn't new, some earliest attempts involved the study of synthetic ethology(MacLennan & Burghardt, 1993) that investigated the mechanisms and evolution of communication in finite state machines using genetic algorithms to learn to cooperate in a simplified environment. Such an approach is also adopted in allowing predator agents to learn to communicate in the grid-world predator-prey problem(Giles & Jim, 2003), hence allowing these antagonistic agents to coordinate with each other for more efficient capture of the prey. However, these genetic algorithm approaches would not be scalable for larger problems and are more so the case for multiagent scenarios.

As documented in numerous literature sources, it is widely acknowledged that there exist two distinct paradigms for communication learning in Multi-Agent Reinforcement Learning (MARL). The first paradigm assumes a continuous nature of the communication channel and employs backpropagation through gradient descent to optimize the learning process. On the other hand, the second paradigm assumes a discrete channel and utilizes RL algorithms for learning.

In the context of the backpropagation approach, it has been observed that it tends to converge more rapidly towards superior policies compared to alternative frameworks. This can be attributed to the fact that it does not require actual interactions with the channel space. The state-of-the-art algorithm DIAL (J. N. Foerster et al., 2016, p. 5) is a centralised training algorithm of which at training, it allows gradients to be pushed from one agent to another through the communication channel for richer feedback, hence making end-to-end trainable across the agents using DQN and taking the full advantage of centralised training.

Another state-of-the-art algorithm is the CommNet (Sukhbaatar et al., 2016) algorithm, where each agent would have an individual communication module (as neural networks), which are together trained and used for communicating on a common continuous channel. All the communication modules together make up the CommNet model, which takes as input the states of each agent and outputs the sampled actions for each agent.

Conversely, the RL approach is employed in scenarios where the communication channels are assumed to be discrete. This implies that the channel cannot be differentiated, making it unsuitable for optimization using backpropagation techniques. Instead, optimization is accomplished through RL algorithms. These algorithms closely align with real-world situations, such as multi-robotic systems, as contemporary networking and communication systems, and protocols are specifically designed to handle discrete signals. Within this paradigm, some early algorithms were developed using the tabular Q Learning method to learn a set of communication codes/protocols to solve the predator-prey problem(Kasai et al., 2008), but these approaches are not scalable due to the nature of tabular storage.

The current state-of-the-art for RL approach is the RIAL algorithm (J. N. Foerster et al., 2016, p. 4) which is introduced in the same paper as DIAL. RIAL adopts the independent Q-Learning architecture and is a decentralised training algorithm. Each agent would have two Q-Networks, one for agent actions and the other for agent messages, essentially treating communication as a separate action space. These networks are trained using DQN with experience-replay disabled to account for non-stationarity in observed states. An extension to RIAL has also been proposed in the same paper that allows parameter sharing to take full advantage of centralised learning. In this extension, only one network is learned centrally and shared across all agents, of which agents would then evolve their hidden states at decentralised execution through interaction with the environment and communication with other agents.

Under these paradigms, communication under constraints such as limited bandwidth is very well researched. A popular approach to tackle this issue is by learning when to communicate, an example being the IC3Net (Singh et al., 2018) which extends from CommNet and utilizes a gating mechanism giving each agent the ability to "block" communication which can be used to prevent over-crowding the communication channel.

The SchedNet (Kim et al., 2019) algorithm is also a well-known algorithm of this kind, which utilises a weighted-base scheduler to decide which agents can broadcast their messages over shared,limited-bandwidth channels based on certain weights. These weights are generated by agents themselves on how important the agent's observed information is.

Notably, though briefly discussed in the DIAL/RIAL paper(J. N. Foerster et al., 2016, p. 8), most algorithms up to date do not consider the constraint where the communication channel is noised. This is still a rather young problem with very few related works of literature and is what this project seeks to address.

One such work that has attempted to tackle this problem is the DiffDiscrete algorithm(Freed et al., 2020) which operates on a discrete communication channel with additive noise that is unknown to the agents. Instead of using the reinforcement learning approach, the algorithm uses a randomized message encoding/decoding scheme to make a mathematically equivalent analogue channel, then which estimations of gradients can be backpropagated through the channel for network optimization. The algorithm also uses a slightly modified randomised encoder to encode its messages which forces messages to be independent to channel noise and thus can be generalised to communication channels with unknown noises.

Yet there are a few concerns that should be addressed. Firstly, throughout the paper, the RCL approach is referenced as the communication MARL approach that optimizes using standard RL techniques. However, the specific algorithm or method associated with the RCL approach is not explicitly stated or defined within the paper. This lack of clarity raises questions about the exact nature of the RCL approach and how it compares to the author's own DiffDiscrete algorithm, which is used for experiments.

Another concern is related to the performance of the proposed method on different noise models. While it is claimed to be noise tolerant and capable of operating on channels with unknown noise, the only noise model that was tested in the experiments was the asymmetric bit-flip error model. This limited testing leaves room for uncertainty regarding the method's effectiveness and generalizability across various noise models. To support their claims of noise tolerance, it would be beneficial to include more extensive test results on a broader range of noise models.

Another very recent work (Tung et al., 2021) presented a complete Reinforcement Learning approach to the noised discrete channel problem, in which they proposed their algorithms by adopting the DQN and DDPG framework for communication learning and studied their effectiveness. They have also explored what they denoted as the “Joint channel coding and modulation” problem which seeks to design a channel coding and modulation scheme with an unknown channel model through the DDPG algorithm.

Notably, the Guide and Scout scenario (see section 5.1) was presented in this paper as their testbed, which has been adopted for this project as such scenarios inherently have an extra emphasis on the importance of communication between agents, thus showing more distinctive differences between noised and un-noised situations.

It is important to note that this addresses a distinct problem from the one targeted by this project. The paper under consideration focuses on the collaboration between a guide agent and a scout agent, both of whom are Deep RL agents. The guide agent aims to learn an appropriate channel input signal to transmit, while the scout agent learns how to behave based on the received noisy signal. The agents collaborate solely to devise an effective channel coding strategy, without any coordination among agents interacting with the environment to achieve objectives. Moreover, the proposed framework lacks robustness when faced with variations in noise models. Even slight changes in the noise model can lead to poor performance, necessitating the retraining and readjustment of agents to adapt to the new noise model.

5 Problem Formulation

The objective of this chapter is to present a formal formulation of the Guide and Scout scenario utilizing established Markov Games definitions. Additionally, it showcases two cooperative instances within this scenario, namely Finding-Treats and Spread. These instances will serve as the training and testing environments for the developed algorithm.

5.1 Guide and Scout

The Guide and Scout scenario was initially introduced by (Tung et al., 2021). In this scenario, two types of agents exist: Guide agents, which can observe the environment but cannot move, and Scout agents, which can move but lack observational capabilities. For this project, the scenario has been adopted and slightly adjusted to focus on fully cooperative scenarios that solely consider team rewards.

This can be formulated using an extended Markov Games definition established in Chapter 3.1 with the following tuple:

$$< \mathcal{N}, \mathcal{S}, \{\mathcal{A}^i\}_{i\in\{1,\ldots,\mathcal{M}\}}, \mathcal{P}, \hat{\mathcal{R}}, \gamma, \mathcal{M}, \mathcal{T}, \mathcal{B}, \mathcal{Z} >$$

with the first 4 components having their usual Markov Games definitions, and other components are defined as follows:

- $\hat{\mathcal{R}}: \mathcal{S} \mapsto \mathbb{R}$: The team reward function
- $\mathcal{M}$: Number of scout agents
- $\mathcal{T}$: Number of goals
- $\mathcal{B}$: The bandwidth of the communication channel
- $\mathcal{Z}$: The set of alphabets allowed on the channel

At each timestep $t \in \mathbb{N}$, only guide agents would receive a shared representation of the environment's current state $s_t \in \mathcal{S}$ and receive a team reward $\hat{r}_t \in \mathbb{R}$ from the reward function $\hat{\mathcal{R}}$. Now agents are allowed to communicate, at timestep t they can send messages $m_t^i \in \mathcal{Z}^{\mathcal{B}}$ over a shared communication channel to another agent i, and receive messages $\hat{m}_t^i \in \mathcal{Z}^{\mathcal{B}}$ sent by the agent i which may be noised. The message sent by the Guide agent is a tuple $(o_t, \hat{r}_t)$ where o_t is the encoded state s_t by the Guide agent. Let $\hat{\mathcal{M}}_t^i = \{\hat{m}_t^i\}_{i\in\{1,\ldots,\mathcal{N}\}}$ denote the joint message received by agent i, based on $\widehat{\mathcal{M}}_t^i$, scout agents select their action $a_t^i \in \mathcal{A}^i$ simultaneously to generate a joint action $\mathbb{A}$. In response to this joint action, at time step $t+1$, guide agents would receive a team reward $\hat{r}_{t+1} \in \mathbb{R}$ from $\hat{\mathcal{R}}$, and would be in the following shared state $s_{t+1} \in \mathcal{S}$ based on the state transition function $\mathcal{P}$.

Two instances of this scenario have been designed, both of which only have one Guide agent and were implemented as deterministic square grid world environments of size $K \times K$ with discrete state space:

$$\mathcal{S} = \{(x, y) \in G \times G |\ G = \{0,1,2, \dots, K\}\}$$

and discrete action space:

$$\mathcal{A} = \{(-1,0), (0, -1), (0,1), (1,0), (0,0)\}$$

where all agents and goals have randomly initialised locations at the start of every episode. In these instances, as they are all fully cooperative, the agents seek to maximize the team return $\hat{G}_t$ with differently defined team reward function $\hat{\mathcal{R}}$, which requires the cooperation and coordination of multiple scout agents.

Both instances operate using a discrete Binary symmetric channel with a parameter:

$$0 \le \wp \le \tfrac{1}{2}.$$

Let X/Y be the transmitted/received random variable correspondingly, and $\mathcal{Z} = \{0,1\}$:

$$X, Y \sim Uniform(\mathcal{Z})$$
$$\Pr[Y = 1 | X = 0] = \wp$$
$$\Pr[Y = 0 | X = 0] = 1 - \wp$$
$$\Pr[Y = 0 | X = 1] = \wp$$
$$\Pr[Y = 1 | X = 1] = 1 - \wp$$

5.2 Finding-Treats

In this instance, the goals can be treated as treats in which agents must find and eat all of them using minimal time, of which then the episode terminates. In this case, the agents operate solely on the team reward function which is defined as follows:

$$\hat{\mathcal{R}} = \begin{cases} 9 & \mathcal{E} = \mathcal{T} \\ -1 - \big(2 * (\mathcal{T} - \mathcal{E})\big) & \mathcal{E} < \mathcal{T} \end{cases}$$

Where $\mathcal{E}$ represents the total number of treats that have been eaten. This reward function encourages multiple scout agents to coordinate with each other to obtain the treats as quickly as possible.

5.3 Spread

This instance is the Guide and Scout variant of the cooperative navigation task in the Multi-Particle Environment (Lowe et al., 2020, p. 6; Mordatch & Abbeel, 2018, pp. 2–3) which is also known as the spread task. In this instance, the scout agents must cover all goals using minimal time and are required to stay on the goal. If an agent captured a goal but left, the goal would be marked as uncaptured again, which makes this task distinctively different and more difficult to solve compared to the Finding-Treat instance. Once again, the agents operate solely on the team reward function defined as follows:

$$\hat{\mathcal{R}} = \begin{cases} 99 - 2 * \mathcal{C} & \mathcal{E} = \mathcal{T} \\ -1 - 2 * \mathcal{C} - 10 * \mathcal{D} & \mathcal{E} < \mathcal{T} \end{cases}$$

Where $\mathcal{E}$ now represents the number of goals that are currently occupied by the agents, $\mathcal{C}$ represents the total number of collisions that have occurred at the current timestep, and $\mathcal{D}$ represents the sum of minimal Euclidean distances of all scout agents from the goals. As described by the team reward function, scout agents are encouraged to avoid collisions with the walls of the environment and with any other agents, while occupying all goals with minimal time.

6 Proposed Solution

This chapter commences by providing an overview of essential background knowledge, encompassing Deep Learning, DQN (Deep Q Networks), and error detection techniques. These elements constitute crucial components of the proposed solution. Subsequently, an algorithmic definition of the solution will be introduced.

6.1 Preliminaries

6.1.1 Deep Q Networks (DQN)

To briefly recap the concept of value function approximation discussed in the preceding section, the objective is to train an approximator $\hat{q}_\theta(\mathrm{s}, a)$ for $q_\pi(s, a)$ with tuneable parameters θ. Subsequently, a soft policy is employed for Policy Improvement, following the principles of Generalized Policy Iteration, ultimately leading to the derivation of an optimal policy and action-value function. To find θ, we want to minimize the error between the approximator $\hat{q}_\theta(\mathrm{s}, a)$ and the ground truth $q_\pi(s, a)$, which is done by finding θ^* that minimizes the loss function $J(\theta)$. With MSE error as the evaluation metric and using the Q-Learning algorithm, $J(\theta)$ can be defined as:

$$J(\theta) \;= \frac{1}{2}\mathbb{E}_\pi\left[\left(R_{t+1} + \gamma \max_a \hat{q}_\theta(S_{t+1}, a) - \hat{q}_\theta(\mathrm{S_t}, A_t)\right)^2\right] \qquad \textit{From (11)}$$

In the case of Deep Neural Networks, this loss value is calculated and backpropagated to allow network weights to be optimized using gradient descent.

The research community is widely aware of the "Deadly Triad," which refers to a combination of factors that can render algorithms highly unstable and hinder convergence. This triad consists of bootstrapping, function approximation, and off-policy learning. Consequently, incorporating function approximation with Q-Learning, which is an off-policy, bootstrapping algorithm, has proven to be particularly challenging due to the presence of these factors.

However, this was eventually resolved by Mnih and his team who introduced the Experience Replay and Target Networks techniques(Mnih et al., 2015), which lead to a major breakthrough in employing Deep Neural Network (DNN) value function approximators along with Q-Learning, known as Deep Q Networks, to play a series of Atari games.

The experience replay technique was first introduced in (Lin, 1993), which involves storing the agent's experience $e_t = (S_t, A_t, R_{t+1}, S_{t+1})$ at each timestep t into a replay buffer $\mathcal{D} = \{e_1, e_2, e_3 \ldots e_N\}$ which is a set of experiences where N represents the replay memory size. In DQN's case, the agent would then randomly sample batches of experience from the replay buffer $\mathcal{D}$ and feed that into its DNN instead of feeding consecutive experiences one by one to the DNN.

The intuition behind this technique is that consecutive experience samples would naturally exhibit a strong correlation with one and the other, which is undesirable due to the independent and identically distributed (i.i.d.) variable assumption that governs most convergence properties of Machine Learning algorithms. Taking random sample batches of experiences prevents such dependencies and thus leads to more efficient training and better convergence behaviours when employing DNNs.

With this technique, $J(\theta)$ becomes the following, where the expectation is now over the batch of (s, a, r, s') sampled uniformly from replay buffer $\mathcal{D}$:

$$J(\theta) \;= \frac{1}{2}\mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}\left[\left(R_{t+1} + \gamma \max_{a'} \hat{q}_\theta(s', a') - \hat{q}_\theta(s, \mathrm{a})\right)^2\right]$$

The target network technique involves having a separate DNN $\hat{Q}$ in addition to the original DNN Q, where for every C updates, $\hat{Q}$ will be updated as the clone of Q. In between the C updates, $\hat{Q}$ will be used for the generation of estimates and kept fixed in between the C updates whereas Q will be updated accordingly.

The motivation for having this additional target DNN is that in the usual DQN (11), we would be using the same network Q for both the bootstrapping operation as well as for the estimate for the current state-action pair. When utilizing parametric function approximators, the process differs from tabular Q-Learning, where only one state-action value is updated at a time. With the use of function approximators, an update to the parameter θ will impact multiple state-action values. This means that when θ is updated, it can affect the action value of the subsequent state. This frequent occurrence in practical scenarios can lead to a situation where the target is never reached, resulting in oscillations of θ updates as both the TD-target $R_{t+1} + \gamma \max_{a'} \hat{q}_\theta(s', a')$ and the network being updated have the same dependencies on the parameter θ.

This is like a dog chasing its tail, having some resemblance to the MARL non-stationarity issue, which leads to what is known as “Catastrophic interference”: Where the network abruptly forgets about what it has learned while learning new information. Having a separate target network which will be kept fixed for C steps allow the TD target to be stationary during these steps and disrelated their dependencies on the same parameter, and has proven to be effective in stabilizing training.

In addition to this result, the DDPG paper has proposed a method to soft update the target network (Lillicrap et al., 2019) instead of keeping the target network fixed for C steps. An additional hyperparameter τ is introduced and the target network parameter is updated every step using the following update rule:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \qquad \text{with } 0 \le \tau \ll 1$$

With τ being extremely small, the target network can still be considered “stationary” and is constrained to changing slowly towards the actual network. Compared to the more abrupt changes to the target network at every C step, this soft updating method has proven to be more robust and better stabilizes training.

With both techniques combined, the loss function becomes the following, with θ' being the parameter of the target network $\hat{Q}$:

$$J(\theta) = \frac{1}{2}\mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}\left[\left(R_{t+1} + \gamma \max_{a'} \hat{q}_{\theta'}(s',a') - \hat{q}_{\theta}(s,\mathrm{a})\right)^2\right]$$

(12)

Conclusively, combing all components together, the following is a very general DQN algorithm that uses experience replay, soft-update target networks and stochastic gradient descent as well as epsilon-greedy for target policy:

Algorithm 1: DQN algorithm with experience replay and soft-update target network

```
Initialise replay buffer $D$ of size $N$;
Initialise DNN $Q$ with random starting weights $\theta$;
Initialise DNN $\hat{Q}$ with starting weights $\theta' = \theta$;
for $episode = 1, M$ do
    $t = 1$;
    Initialise starting state $s_1$;
    while goal not achieved do
        $a_t = \begin{cases} \text{argmax}_a Q_\theta(s_t, a) & \text{probability } 1-\epsilon \\ \text{a} \sim Uniform(A(s_t)) & \text{probability } \epsilon \end{cases}$
        Take action $a_t$, receiving reward $r_{t+1}$ and following state $s_{t+1}$;
        Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $D$;
        Sample batch of transitions $(s_j, a_j, r_{j+1}, s_{j+1})$ from $D$;
        Set $y_j = \begin{cases} \text{r}_{j+1} & \text{If } s_{j+1} \text{ is terminal} \\ \text{r}_{j+1} + \gamma \max_{a'} \hat{Q}_{\theta'}(s_{j+1}, a') & \text{Otherwise} \end{cases}$
        Perform gradient descent step on $J(\theta)$ with respect to $\theta$;
        $\theta' = \tau\theta + (1-\tau)\theta'$;
        $t = t + 1$;
    end
end
```

6.1.2 Error Detection

Error detection plays a crucial role in information and coding theories by ensuring the reliable transmission of data through communication channels. These channels are often prone to unreliability and noise, which can result in unintended modifications to the transmitted data. To address this, error detection techniques are employed, enabling the receiver to identify such alterations and take appropriate actions in response.

A trivial example of an error detection technique is called Longitudinal Redundancy Check (LRC). The sender would generate an LRC code by first breaking the data to be sent into blocks each having n bits. Then an XOR operation is performed on all the blocks which results in a n bits LRC code that is sent along with the original message. This same algorithm would be used at the receiver's decoding step to generate the LRC code again which is checked against the received LRC code to verify whether an error has occurred.

Another option to consider instead of LRC is the sum complement checksum method. Similarly, the data to be transmitted is divided into blocks, each consisting of n bits. These blocks are then summed together, including any carrying bits, resulting in a sum of n bits. The next step involves performing a one's complement operation on the sum, flipping all the bits and obtaining the final checksum. This checksum is then sent along with the message. Upon receiving the message, the receiver also segments it into blocks of n bits, sums up all the blocks, and performs a one's complement operation on the sum. If the resulting value is 0, it indicates that no error has occurred in the message. However, if the result is 1, it suggests the presence of an error in the message.

An observation for both LRC and sum complement checksum is that these algorithms may fail to detect various trivial and common errors as they are position independent. For example, consider the following data: $00001111\ 11110000$ with $n = 8$. The LRC code for this data would be a byte of all 1s $0xFF$, and the sum complement checksum would be a byte of all 0s $0x00$. Imagine during transmission, the bit at the same index (say the 1st index) were flipped for both data blocks, the receiver would receive data: $10001111\ 01110000$, and the LRC generated would be $0xFF$ and sum complement checksum would be $0x00$, which fails to detect the error.

A more sophisticated, position-dependent method that is commonly used in practice is Cyclic Redundancy Check(CRC) (Peterson & Brown, 1961) which is widely implemented in modern digital communication systems. First, the sender and the receiver must agree on a certain polynomial through certain data handshakes. Typically, polynomials of degrees 8, 16, 32, or 64 are used in CRC to minimize the likelihood of collisions and maximize error detection capabilities. However, for explanatory purposes, a polynomial of degree 3 will be employed.

A polynomial of degree n can be encoded using $n + 1$ bits for its coefficients, for example, the degree 3 polynomial $(1)x^3 + (1)x^2 + (1)x + (0)$ can be encoded as 1110, and this will act as a divisor to generate a CRC code. Assume the sender wants to send a message of 6 bits 110100, to generate the CRC code, the message will first be padded to the right with n bits of 0s, giving $110100\ 000$. Then iteratively, the polynomial code will be aligned with the leftmost 1 of the data and perform XOR with the aligned bits:

$$\frac{110100\ 000}{1110}\ XOR\ = 001100\ 000$$

$$\frac{001100\ 000}{1110}\ XOR\ = 000010\ 000$$

$$\frac{000010\ 000}{11\ 10}\ XOR\ = 000001\ 100$$

$$\frac{000001\ 100}{1\ 110}\ XOR\ = 000000\ 010$$

In the last step, the dividend is 0 and cannot be divided further. The remainder is 010 which is the CRC code for the current degree 3 polynomial. This code is sent along with the original message, and the receiver would validate the received message by doing the same operation again. If no error has occurred during transmission, then the remainder should equal 0:

$$\frac{110100\ 010}{1110}\ XOR\ = 001100\ 010$$

$$\dots\dots$$

$$\frac{000001\ 110}{1\ 110}\ XOR\ = 000000\ 000$$

6.2 Algorithmic Solution

The solution can be separated into two phases, the training phase and the deployment phase:

6.2.1 Training Phase

During the training phase, the agents undergo training using the DQN algorithm while utilizing a communication channel that is shared among them and free from any noise. In this setup, each scout agent learns to cooperate and synchronize its actions with the other scout agents. On the other hand, the guide agent is responsible for observing the current state and distributing the encoded observation, along with the team reward, to the scout agents.

To encode the state, it is represented as a 1-dimensional array consisting of a sequence of tuples, where each tuple contains a natural number. This state representation is further encoded into a binary stream, with each natural number allocated 1 byte, enabling its transmission over the discrete communication channel:

$$o_t = [(x_{t,g}, y_{t,g}), (x_{t,s[1]}, y_{t,s[1]}), (x_{t,s[2]}, y_{t,s[2]}), \dots, (x_{t,l[1]}, y_{t,l[1]}), (x_{t,l[2]}, y_{t,l[2]})] \tag{13}$$

Where $(x_{t,g}, y_{t,g})$ represents the x-y coordinate of the Guide agent in the Gridworld at timestep t, $(x_{t,s[i]}, y_{t,s[i]})$ for each Scout agent i's coordinates, and $(x_{t,l[i]}, y_{t,l[i]})$ for goal i's coordinates.

The algorithm uses ϵ-greedy target policy with ϵ-decay (10). Here for optimization, a batch of experiences of size B will be sampled from the replay buffer, where A_j, R_j and Y_j all corresponds to vector values of size $B \times 1$ and O_j corresponds to matrix values of size $B \times N$ with N being the size of each encoded observations o_j. The $^{[i]}$ notation is to indicate the specific batch number, for example $R_{j+1}^{[1]}$ corresponds to the reward value for the very first batch. The loss metric chosen for the optimization step was MSE loss, thus the loss function $J(\theta)$ is as desribed in Equation (12), where now the TD-target may vary depending on whether the observation is terminal, and the optimization process is performed in batches.

```
Algorithm 2: Training Algorithm for Scout Agents
Initialise replay buffer $D$ of size $N$;
Initialise batch size $B$;
Initialise $\epsilon_{start}, \epsilon_{end}$;
Initialise epsilon decay factor $\lambda$;
Initialise DNN $Q$ with random starting weights $\theta$;
Initialise DNN $\hat{Q}$ with starting weights $\theta' = \theta$;
for $episode = 0, (M-1)$ do
    $\epsilon = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) * e^{-\frac{episode}{\lambda}}$;
    $t = 1$;
    while goal not achieved do
        Receive message $m_t = (o_t, r_t)$ from Guide;
        $a_t = \begin{cases} \text{argmax}_a Q_\theta(o_t, a) & \text{probability } 1-\epsilon \\ \text{a} \sim Uniform(A(o_t)) & \text{probability } \epsilon \end{cases}$
        Take action $a_t$, receive message $m_{t+1} = (o_{t+1}, r_{t+1})$ from Guide;
        Store transition $(o_t, a_t, r_{t+1}, o_{t+1})$ in $D$;
        if $length(D) >= B$ then
            Sample batch of $B$ transitions $(O_j, A_j, R_{j+1}, O_{j+1})$ from $D$;
            Set $Y_j^{[i]} = \begin{cases} \text{R}_{j+1}^{[i]} & \text{If } O_{j+1}^{[i]} \text{ is terminal} \\ \text{R}_{j+1}^{[i]} + \gamma \max_{a'} \hat{Q}_{\theta'}(O_{j+1}^{[i]}, a') & \text{Otherwise} \end{cases}$
            Perform single optimization step on $J(\theta)$ with respect to $\theta$;
        $\theta' = \tau\theta + (1-\tau)\theta'$;
        $t = t + 1$;
```

6.2.2 Deployment Phase

Once the scout agents have completed their training in a noise-free environment, additional techniques have been implemented when deploying these agents in the actual environment, which involves a binary symmetric communication channel with noise.

The Guide agent now would send messages along with an error detection code, where currently sum’s complement and CRC-8 (that uses 8-degree polynomials) techniques were implemented, such that the scout agents would be able to verify the received messages.

Alongside the error detection code, the agents are initialized with an initial majority voting number, denoted as V. During each timestep, the guide agent sends the same message V times to the scouts. The scouts then perform a majority voting on the received bit stream. The number of majority votes utilized can be adjusted based on the condition of the communication channel, ensuring that it is suitable without introducing excessive communication overhead. This adjustment is limited by a predefined bandwidth size. The status of the channel is measured by :

$$falseRatio = \frac{falseCount}{MASampleSize}$$

Where $falseCount$ indicates the number of messages that didn't pass the error detection check, and $MASampleSize$ is a preset constant that indicates how often is the adjustment being done. The majority number is adjusted if falseRatio becomes larger than a preset threshold, which is achieved by broadcasting a majority voting adjustment signal to all other agents.

For every scout agent, the received messages are decoded, and the observations are stored in a local history along with the corresponding error detection outcomes and the action taken at that step. Only the most recent H observations are kept, and these are utilized by the "Message Recoverer" to process the currently received observation. The goal is to extract as much valuable information as possible, and potentially recover the original observation that was initially transmitted. The recovered or processed observation is then used by the scout agent to select an ϵ-greedy action.

Recall the encoding of observations (13), since the coordinates of the guides and goals remain fixed, we can recover them using the coordinates with the highest occurrences. To achieve this, separate dictionaries are used to count the occurrences of these coordinate tuples. In the case of receiving error-free observations, a multiplier is applied to the count. The coordinates with the highest count are considered the "anchor position" and will be substituted for the coordinates in the currently received observation.

Now consider the scout agent i, to recover its coordinate at the current timestep t $\left(x_{t,s[i]}, y_{t,s[i]}\right)$, the agent would backtrack in its local history looking for the most recent record with no errors detected (has a valid error detection code). If such a record exists, this observation is treated as the starting recall observation and the forward recall operation is initiated. Let the starting recall observation be o_r, this operation can be thought of as having an avatar of the current agent who starts at timestep r again with coordinate $\left(x_{r,s[i]}, y_{r,s[i]}\right)$ and takes the action a_r that was taken at that timestep. This process is iterated until the end of the history, where the resulting state represents the most probable coordinate at which agent i is located. However, even in deterministic environments, this coordinate may not be exact due to possible collisions with other agents. If a valid record is not found during this process, only a one-step recall is used. In this case, the "starting recall observation" is set as the most recent observation in the local history.

To resolve the coordinates of all other scout agents $\forall j \neq i, j < \mathcal{M}$, the matter is more difficult as the action taken by other agents is unknown and cannot be resolved directly in the case of noisy communication. Let the current timestep be t and consider the perspective of agent i, once again, agent i attempts to find the starting recall observation o_r. If it exists, then all agent j's coordinates are resolved by considering their most optimal and optimistic positions that those agents would be at in $t-r$ steps with respect to a team objectives heuristic function. The team objective heuristic that has been designed is the agent proximity heuristic where agent i would heuristically assign all agents (including itself) goal coordinate to whichever treat that they are closest to using Euclidean distance, and for each agent other than itself, compute the position that they could be at in $t-r$ steps that minimizes their distance to their assigned goal the most.

To summarize, the algorithm for the scout agents can be described as follows. While an ϵ-greedy policy is utilized, the ϵ value has been significantly decayed to a very small value after the training phase. Therefore, it can be considered a fully greedy policy based on the trained Q-network.

```
Algorithm 3: Deployment Algorithm for Scout Agent $i$
  Initialize history $H$;
  Initialize Guide positions dictionary $G$;
  Initialize Goal positions dictionary $L$;
  Agree on Error Detection Scheme with Guide that generates code $EDC$;
  Initialize $MASampleSize, NoiseThresholdRatio, MessagesReceived$;
  Initalise $FalseCount = 0$;
  Initialize number of majority voting $V$;
  Initialize Correct Message Multiplier $C$;
  Initialize $t = 1$;
  while goal not achieved do
      Recieve messages $(m_{t[1]}, m_{t[2]}, ..., m_{t[V]})$ from Guide;
      Generate majority voted message $m_t = (o_t, r_t, EDC)$;
      $checkResult$ = Result of checking message using $EDC$;
      Decode observation
        $o_t = [(x_{t,g}, y_{t,g}), (x_{t,s[1]}, y_{t,s[1]}), \ldots, (x_{t,s[M]}, y_{t,s[M]}), (x_{t,l[1]}, y_{t,l[1]}), ..., (x_{t,l[T]}, y_{t,l[T]})]$;
      $MessageReceived = MessageReceived + 1$;
      $increment = \begin{cases} 1 & checkResult \text{ fail} \\ C & checkResult \text{ succeeds} \end{cases}$
      $G[(x_{t,g}, y_{t,g})] = G[(x_{t,g}, y_{t,g})] + increment$;
      for $j = 1$ to $T$ do
          $L_j[(x_{t,l[j]}, y_{t,l[j]})] = L_j[(x_{t,l[j]}, y_{t,l[j]})] + increment$;
      if checkResult fails then
          $FalseCount = FalseCount + 1$;
          if $MessageReceived >= MASampleSize$ then
              $FalseRatio = \frac{FalseCount}{MASampleSize}$;
              Adjust $V$ if $FalseRatio > NoiseThresholdRatio$;
          Set $(x_{t,g}, y_{t,g})$ as $argmax_{(x,y)} G$;
          for $j = 1$ to $T$ do
              Set $(x_{t,l[j]}, y_{t,l[j]}) = argmax_{(x,y)} L_j$;
          Find recall index $r$;
          Set $(x_{t,s[i]}, y_{t,s[i]}) = RecallMyState(r, H)$;
          for $j = 1$ to $M$ do
              Set $(x_{t,s[j]}, y_{t,s[j]}) = OptimalState(j, TeamObjectiveHeuristic(), t - r)$;
      Take action $a_t = \begin{cases} \text{argmax}_a Q_\theta(o_t, a) & \text{probability } 1 - \epsilon \\ \text{a} \sim Uniform(A(o_t)) & \text{probability } \epsilon \end{cases}$
      Append $(o_t, a_t, checkResult)$ to $H$;
      $t = t + 1$;
```

7 Numerical Results and Evaluation

The objective of this chapter is to present and analyse a collection of numerical results obtained from the Finding Treat and Spread environments, along with the associated hyperparameter adjustments.

7.1 General Experiment Settings

Below is the computer specification used for training and testing:

- OS Name: Microsoft Windows 10 Home
- Processor: Intel(R) Core(TM) i7-10700F CPU @ 2.90GHz, 2904 Mhz, 8 Core(s), 16 Logical Processor(s)
- GPU: NVIDIA GeForce RTX 2060

Below are the Python and libraries versions used for implementing the algorithm:

- Python 3.9.13
- crc==4.2.0
- gym==0.26.2
- joblib==1.2.0
- matplotlib==3.5.2
- numpy==1.21.5
- pandas==1.4.4
- scipy==1.9.1
- seaborn==0.11.2
- torch==1.13.0
- tqdm==4.64.1

For the experimental setup, CRC-8 has been utilized as the error detection code. The experiments are conducted using different values of p for the Binary symmetric channel, specifically: [0, 0.001, 0.01, 0.1, 0.2, 0.3, 0.4]. For each value of p, the experiment is repeated 100 times, and the maximum number of timesteps per run is set to 5000. In each model's experiment, the initial configuration of the environment for each test run is generated using the same random number generator seeds. These seeds are distinct from the seed used during training to prevent any potential information leakage.

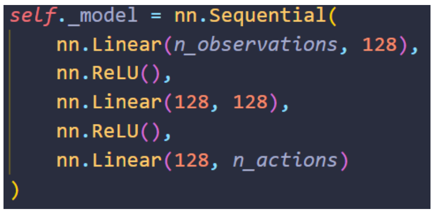
For hyperparameter tuning only a selective number of parameters were varied and the common hyperparameters used by the agents are:

- $\gamma = 0.99$ (Discount factor)
- $\epsilon_{start} = 0.9$
- $\epsilon_{end} = 0.05$

In both cases, a visual representation will be presented, showcasing the top 3 performing models. This will be followed by another figure, which compares the top-performing model against Checksum, Norm, and Norm_Noised. Checksum is a model that employs sum's complement checksum for error detection and is trained with identical hyperparameters as the top model. Norm represents the behaviour of the model when no noise is present, tested with $\wp = 0$. Norm_Noised, on the other hand, demonstrates how the model performs when deployed in an environment with a Binary symmetric channel, without utilizing any of the proposed methods.

7.2 Finding-Treats

For the Finding-Treats scenario, a 5×5 Grid-world has been chosen with 2 scouts, 2 treats and 1 guide. The DNN architecture chosen for each Scout agent is a Feed-forward network with 3 Fully-connected layers using RELU activation.



```
self._model = nn.Sequential(
    nn.Linear(n_observations, 128),
    nn.ReLU(),
    nn.Linear(128, 128),
    nn.ReLU(),
    nn.Linear(128, n_actions)
)
```

Figure 5. Finding-Treats network architecture

For hyper-parameter tuning, the agents are trained for 100000 episodes for each parameter setting and the set of parameters that were considered for hyperparameter tuning are:

- α = [0.0001, 0.001] (Step-size)
- $\mathcal{B}$ = [32,64, 128, 256] (Batch size)
- λ = [8000,10000,12000,14000, 16000, 20000] (Epsilon decay)
- τ = [0.0005, 0.001, 0.005, 0.01, 0.05]

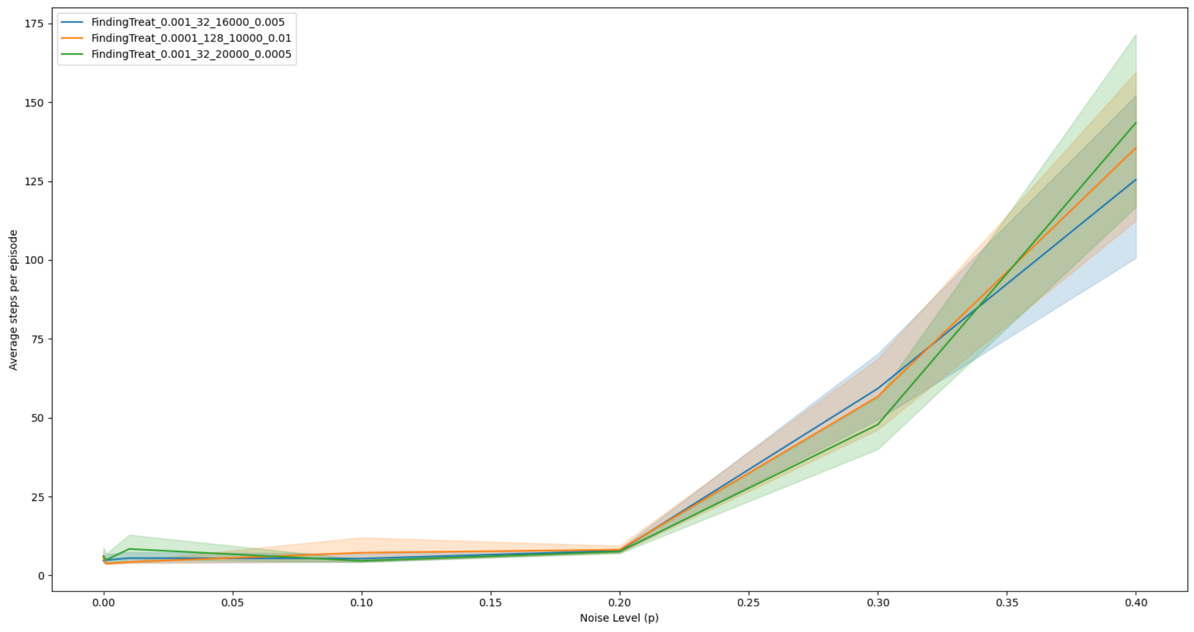


Figure 6. Showing only the Top 3 performing models for Finding-Treats based on mean steps per episode

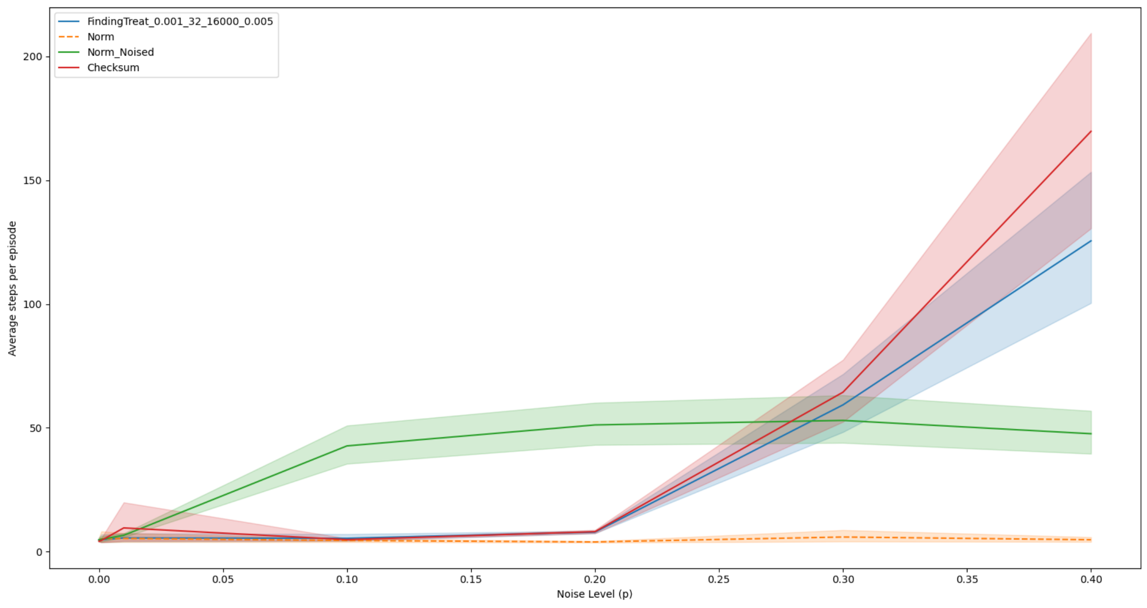


Figure 7. Showing the best-performing model against the Checksum, Norm and Norm_Noised model in Finding Treats

	Norm	Norm_Noised	FindingTreat_0.001_32_16000_0.005	FindingTreat_0.0001_128_10000_0.01	FindingTreat_0.001_32_20000_0.0005
0	4.75,	4.75	4.75	5.75	6.12
0.001	5.13	4.95	4.83	3.78	4.9
0.01	5.29	6.55	5.5	4.26	8.39
0.1	4.46	42.67	5.35	7.2	4.62
0.2	3.89	51.14	7.85	8.16	7.55
0.3	5.87	52.96	59.24	56.71	47.82
0.4	4.76	47.59	125.5	135.53	143.52

Figure 8 Numerical Figures showing the mean step counts of the Top 3 models, Norm and Norm_Noised in Finding Treats with the leftmost column representing the set of $\wp$ values

7.3 Spread

For the Spread scenario, a 3×3 Grid-world has been chosen with 2 scouts, 2 treats and 1 guide. Having trouble training with the Finding-Treats DNN architecture, the DNN architecture chosen for this scenario consists of deeper layers:

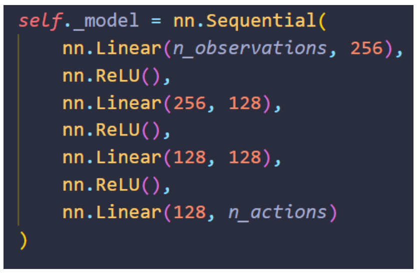

```
self._model = nn.Sequential(
    nn.Linear(n_observations, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, 128),
    nn.ReLU(),
    nn.Linear(128, n_actions)
)
```

Figure 9. Spread network architecture

Agents are trained for 150000 episodes and the set of parameters that were considered for hyperparameter tuning are:

- α = [0.0001, 0.001] (Step-size)
- $\mathcal{B}$ = [64, 128, 256] (Batch size)
- λ = [14000, 16000, 18000, 20000] (Epsilon decay)
- τ = [0.0005, 0.001, 0.005, 0.01, 0.05]

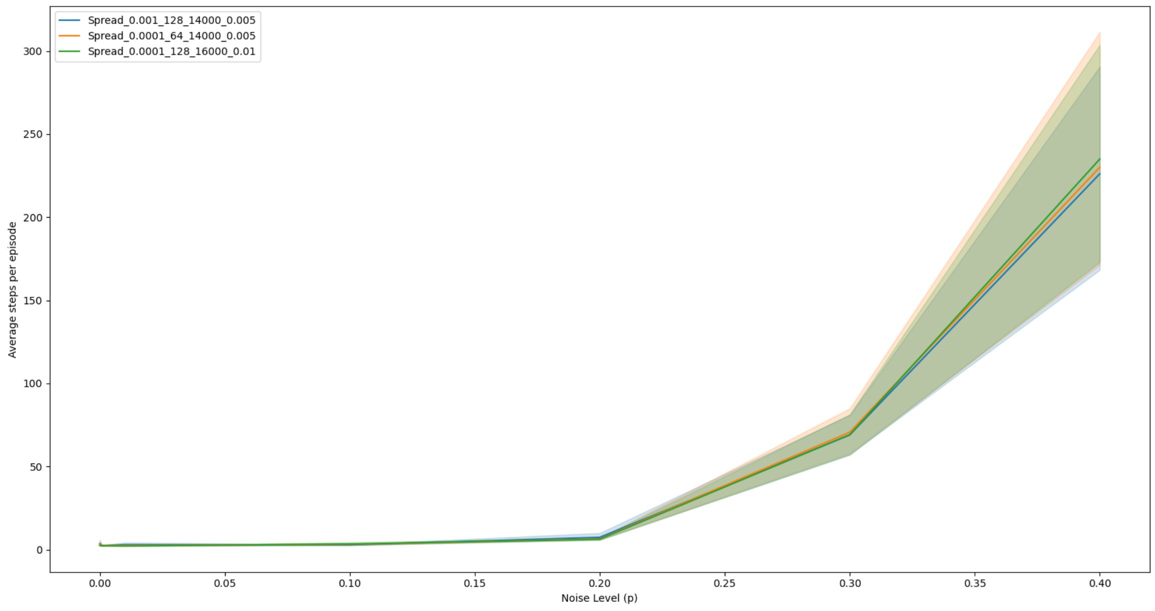


Figure 10. Showing only the Top 3 performing models for Spread based on mean steps per episode

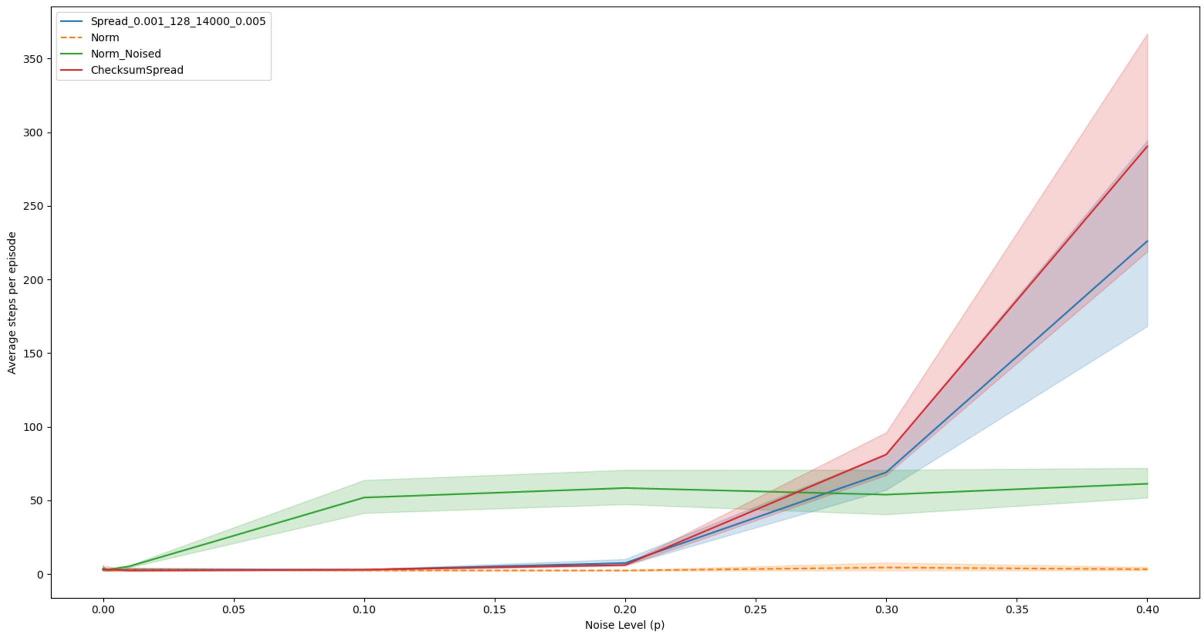


Figure 11. Showing the best performing model against the Checksum, Norm and Norm_Noised model in Spread

	Norm	Norm_Noised	Spread_0.001_128_14000_0.005	Spread_0.0001_64_14000_0.005	Spread_0.0001_128_16000_0.01
0	3.56	3.56	3.56	3.19	2.44
0.001	2.32	2.72	2.37	2.54	2.37
0.01	2.89	5.12	2.98	2.51	2.22
0.1	2.38	51.87	2.83	3.14	3.37
0.2	2.33	58.33	7.39	6.44	6.27
0.3	4.36	53.85	69.02	70.61	69.09
0.4	3.19	61.2	226.06	229.97	234.93

Figure 12. Numerical Figures showing the mean step counts of the Top 3 models, Norm and Norm_Noised in Spread with the leftmost column representing the set of $\wp$ values

7.4 Summative Analysis

The comprehensive results of hyperparameter tuning for both scenarios are available in the appendix or can be referred to in the provided code. An initial observation of the figures indicates that the checksum method generally performs poorer compared to the CRC-8 method. Further analysis of the figures from both scenarios reveals that the agents trained using the suggested approach exhibit outstanding performance when the value of $\wp$ is 0.2 or less. In these cases, they show similar results to a model operating without any noise in the communication channel (referred to as "Norm").

However, there is significant variation in the performance of the models as $\wp$ exceeds 0.2, particularly noticeable at $\wp$ =0.4. Interestingly, when $\wp$ surpasses 0.2, the models employing the proposed method exhibit poorer performance compared to the "Norm_Noised" model. This difference in performance can be attributed to the design of the recall strategy, which heavily relies on having at least one accurately transmitted observation within the agent's local history. To increase the likelihood of having at least one accurately transmitted observation, the proposed method utilizes adjustable majority voting within a predefined bandwidth. However, this bandwidth limit is reached when $\wp$ exceeds 0.2. As a result, agents are seldom able to effectively utilize the recall strategy, leading to a significant increase in step counts and greater performance variance. There appears to be a trade-off between the bandwidth size and the accuracy of message transmission across a noisy channel, which is an expected trade-off and it would be impractical to devise an approach that disregards this constraint.

Another noteworthy observation from the numerical results is that the models tend to exhibit improved performance when p=0.001 compared to the un-noised case (p=0). This trend is hinted at by the results obtained from the "Norm" model, which also demonstrates this pattern. Interestingly, whenever "Norm" outperforms the other models (excluding "Norm_Noised") for p<0.2, those models also exhibit superior performance regardless of the noise level. This could potentially be explained by the way the experiments are seeded. In each model's experiment for a specific p, the same seed is utilized for environment initialization. Therefore, for p<0.2, where the channel is not heavily affected by noise, when "Norm" performs well, it may indicate that the seed generated relatively easier problem instances at that particular p. Consequently, the other models (excluding "Norm_Noised"), which employ the proposed methods, would also be solving the same set of relatively easier problems, leading to better overall performance.

8 Conclusion

8.1 Summary of Achievements

This report encompasses various aspects of Reinforcement Learning and Multiagent Reinforcement Learning. Firstly, it provides a rigorous formulation of the foundational knowledge of Reinforcement Learning, which serves as a crucial foundation for understanding multiagent scenarios. Additionally, a wide range of RL algorithms is introduced, highlighting the diversity of approaches in this field.

The report further explores Multiagent Reinforcement Learning, discussing the major challenges associated with it. It also presents the current mainstream paradigms in MARL and their associated state-of-the-art algorithms, providing a comprehensive overview of the field. Moreover, the report addresses the problem dimensions in communication MARL and conducts a broad survey of state-of-the-art algorithms in this area. This analysis contributes to a deeper understanding of communication-based MARL approaches.

Building upon this knowledge, a novel MARL algorithm is proposed to solve the Guide and Scout Grid world problem. The algorithm aims to enable effective cooperation and coordination among Scout agents in this specific scenario. To validate the proposed algorithm, thorough testing and analysis are conducted in 2 fully cooperative instances of this scenario: Finding-Treat and Spread. The obtained results are carefully analysed as a proof of concept, taking into account the associated hyperparameter tuning.

Overall, this report combines theoretical formulations, algorithmic proposals, and empirical analysis to provide a comprehensive exploration of Reinforcement Learning and Multiagent Reinforcement Learning. The focus on the Guide and Scout Grid world problem instances demonstrates the practical application and effectiveness of the proposed MARL algorithm.

8.2 Evaluation & Future Work

Regarding the evaluation of my work, several important observations can be made. Firstly, the approach adopted in this study took almost no constraint on bandwidth, leading to a heavy reliance on the majority voting technique. However, it should be acknowledged that this heavy reliance on majority voting may not be practical in real-world scenarios. Additionally, the system heavily depends on the successful transmission of at least some messages, as evidenced by the recall operation, which initiates only upon finding a starting recall observation. This reliance on perfect transmission may also prove impracticable in certain situations. Furthermore, an analysis of the figures reveals that the method performs poorly when $\mathscr{p}$, the Binary symmetric channel noise probability exceeds 0.2 and exhibits very high variance. Nonetheless, it can be argued that such performance degradation is acceptable, as a $\mathscr{p}$ greater than 0.2 already signifies an extremely noisy channel. However, the fact that the "Norm_Noised" model, which does not employ the suggested approach, outperforms the proposed method when $\mathscr{p} \geq 0.2$ may also suggest areas where enhancements can be made.

Looking ahead, there are several avenues for future work. Firstly, it would be interesting to test the algorithm's performance in non-deterministic environments, as the current implementation operates solely in a deterministic environment. Moreover, expanding the scope of communication beyond the guide and scouts to include communication between scout agents may lead to performance improvements. Additionally, the evaluation has been limited to the binary symmetric channel, and it would be valuable to consider other noise models such as the Rayleigh Fading Channel or the Rician Fading Channel, both of which are commonly encountered in wireless digital communication. Furthermore, the handcrafted nature of the observation encodings warrants further investigation, and it may be worthwhile to explore incorporating machine learning-inspired channel coding and modulation techniques into the proposed solution.

9 Bibliography

Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., & Mordatch, I. (2020). *Emergent Tool Use From Multi-Agent Autocurricula* (arXiv:1909.07528). arXiv. https://doi.org/10.48550/arXiv.1909.07528

Bakker, B., Whiteson, S., Kester, L., & Groen, F. C. A. (2010). *Traffic Light Control by Multiagent Reinforcement Learning Systems* (R. Babuška & F. C. A. Groen, Eds.; Vol. 281, pp. 475–510). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-11688-9_18

Chang, Y.-H., Ho, T., & Kaelbling, L. P. (2004). *All learning is local: Multi-agent learning in global reward games*. https://dspace.mit.edu/handle/1721.1/3851

Foerster, J., Farquhar, G., Afouras, T., Nardelli, N., & Whiteson, S. (2018). Counterfactual Multi-Agent Policy Gradients. *Proceedings of the AAAI Conference on Artificial Intelligence*, *32*(1), Article 1. https://doi.org/10.1609/aaai.v32i1.11794

Foerster, J. N., Assael, Y. M., de Freitas, N., & Whiteson, S. (2016). *Learning to Communicate with Deep Multi-Agent Reinforcement Learning* (arXiv:1605.06676). arXiv. https://doi.org/10.48550/arXiv.1605.06676

Foerster, J., Nardelli, N., Farquhar, G., Afouras, T., Torr, P. H. S., Kohli, P., & Whiteson, S. (2018). *Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning* (arXiv:1702.08887). arXiv. https://doi.org/10.48550/arXiv.1702.08887

Freed, B., Sartoretti, G., Hu, J., & Choset, H. (2020). Communication Learning via Backpropagation in Discrete Channels with Unknown Noise. *Proceedings of the AAAI Conference on Artificial Intelligence*, *34*(05), Article 05. https://doi.org/10.1609/aaai.v34i05.6205

Giles, C. L., & Jim, K.-C. (2003). Learning Communication for Multi-agent Systems. In W. Truszkowski, M. Hinchey, & C. Rouff (Eds.), *Innovative Concepts for Agent-Based Systems* (pp. 377–390). Springer. https://doi.org/10.1007/978-3-540-45173-0_29

Howard, R. A. (1960). *Dynamic programming and Markov processes* (pp. viii, 136). John Wiley.

Kasai, T., Tenmoto, H., & Kamiya, A. (2008). Learning of communication codes in multi-agent reinforcement learning problem. *2008 IEEE Conference on Soft Computing in Industrial Applications*, 1–6. https://doi.org/10.1109/SMCIA.2008.5045926

Kim, D., Moon, S., Hostallero, D., Kang, W. J., Lee, T., Son, K., & Yi, Y. (2019). *Learning to Schedule Communication in Multi-agent Reinforcement Learning* (arXiv:1902.01554). arXiv. https://doi.org/10.48550/arXiv.1902.01554

Laurent, G. J., Matignon, L., & Fort-Piat, N. L. (2011). The world of independent learners is not markovian. *International Journal of Knowledge-Based and Intelligent Engineering Systems*, *15*(1), 55–64.

Leibo, J. Z., Zambaldi, V., Lanctot, M., Marecki, J., & Graepel, T. (2017). *Multi-agent Reinforcement Learning in Sequential Social Dilemmas* (arXiv:1702.03037). arXiv. http://arxiv.org/abs/1702.03037

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2019). *Continuous control with deep reinforcement learning* (arXiv:1509.02971). arXiv. https://doi.org/10.48550/arXiv.1509.02971

Lin, L. (1993). *Reinforcement Learning for Robots Using Neural Networks* [Carnegie Mellon University]. https://apps.dtic.mil/sti/pdfs/ADA261434.pdf

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Machine Learning Proceedings 1994* (pp. 157–163). Elsevier. https://doi.org/10.1016/B978-1-55860-335-6.50027-1

Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., & Mordatch, I. (2020). *Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments* (arXiv:1706.02275). arXiv. https://doi.org/10.48550/arXiv.1706.02275

MacLennan, B. J., & Burghardt, G. M. (1993). Synthetic Ethology and the Evolution of Cooperative Communication. *Adaptive Behavior*, *2*(2), 161–188. https://doi.org/10.1177/105971239300200203

Matignon, L., Laurent, G. J., & Le Fort-Piat, N. (2007). Hysteretic Q-learning: An algorithm for Decentralized Reinforcement Learning in Cooperative Multi-Agent Teams. *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 64–69. https://doi.org/10.1109/IROS.2007.4399095

Matignon, L., Laurent, G. J., & Le Fort-Piat, N. (2012). Independent reinforcement learners in cooperative Markov games: A survey regarding coordination problems. *The Knowledge Engineering Review*, *27*(1), 1–31. https://doi.org/10.1017/S0269888912000057

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), Article 7540. https://doi.org/10.1038/nature14236

Mordatch, I., & Abbeel, P. (2018). *Emergence of Grounded Compositional Language in Multi-Agent Populations* (arXiv:1703.04908). arXiv. https://doi.org/10.48550/arXiv.1703.04908

Papoudakis, G., Christianos, F., Rahman, A., & Albrecht, S. V. (2019). *Dealing with Non-Stationarity in Multi-Agent Deep Reinforcement Learning* (arXiv:1906.04737). arXiv. https://doi.org/10.48550/arXiv.1906.04737

Peng, P., Wen, Y., Yang, Y., Yuan, Q., Tang, Z., Long, H., & Wang, J. (2017). *Multiagent Bidirectionally-Coordinated Nets: Emergence of Human-level Coordination in Learning to Play StarCraft Combat Games* (arXiv:1703.10069). arXiv. https://doi.org/10.48550/arXiv.1703.10069

Peterson, W. W., & Brown, D. T. (1961). Cyclic Codes for Error Detection. *Proceedings of the IRE*, *49*(1), 228–235. https://doi.org/10.1109/JRPROC.1961.287814

Rashid, T., Samvelyan, M., de Witt, C. S., Farquhar, G., Foerster, J., & Whiteson, S. (2018). *QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning* (arXiv:1803.11485). arXiv. https://doi.org/10.48550/arXiv.1803.11485

Rummery, G. A., & Niranjan, M. (1994). *ON-LINE Q-LEARNING USING CONNECTIONIST SYSTEMS*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms* (arXiv:1707.06347). arXiv. https://doi.org/10.48550/arXiv.1707.06347

Silver, D. (2015). *Lecture 5: Model-Free Control*. 41.

Singh, A., Jain, T., & Sukhbaatar, S. (2018). *Learning when to Communicate at Scale in Multiagent Cooperative and Competitive Tasks* (arXiv:1812.09755). arXiv. https://doi.org/10.48550/arXiv.1812.09755

Sukhbaatar, S., Szlam, A., & Fergus, R. (2016). *Learning Multiagent Communication with Backpropagation* (arXiv:1605.07736). arXiv. https://doi.org/10.48550/arXiv.1605.07736

Sutton, R., & Barto, A. (2018). *Reinforcement learning: An introduction (2nd ed.)* (2nd ed.). The MIT Press. https://mitpress.mit.edu/9780262039246/reinforcement-learning/

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*(1), 9–44. https://doi.org/10.1007/BF00115009

Tesauro, G. (2003). Extending Q-Learning to General Adaptive Multi-Agent Systems. *Advances in Neural Information Processing Systems*, *16*.

https://proceedings.neurips.cc/paper/2003/hash/e71e5cd119bbc5797164fb0cd7fd94a4-Abstract.html

Tromp, J., & Farnebäck, G. (2007). Combinatorics of Go. In H. J. van den Herik, P. Ciancarini, & H. H. L. M. Donkers (Eds.), *Computers and Games* (Vol. 4630, pp. 84–99). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-75538-8_8

Tung, T.-Y., Kobus, S., Pujol, J. R., & Gunduz, D. (2021). *Effective Communications: A Joint Learning and Communication Framework for Multi-Agent Reinforcement Learning over Noisy Channels* (arXiv:2101.10369). arXiv. https://doi.org/10.48550/arXiv.2101.10369

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., … Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, *575*(7782), Article 7782. https://doi.org/10.1038/s41586-019-1724-z

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3), 279–292. https://doi.org/10.1007/BF00992698

Yang, Y., & Wang, J. (2021). *An Overview of Multi-Agent Reinforcement Learning from Game Theoretical Perspective* (arXiv:2011.00583). arXiv. https://doi.org/10.48550/arXiv.2011.00583

Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A., & Wu, Y. (2022). *The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games* (arXiv:2103.01955). arXiv. https://doi.org/10.48550/arXiv.2103.01955

Zheng, S., Trott, A., Srinivasa, S., Naik, N., Gruesbeck, M., Parkes, D. C., & Socher, R. (2020). *The AI Economist: Improving Equality and Productivity with AI-Driven Tax Policies* (arXiv:2004.13332). arXiv. https://doi.org/10.48550/arXiv.2004.13332

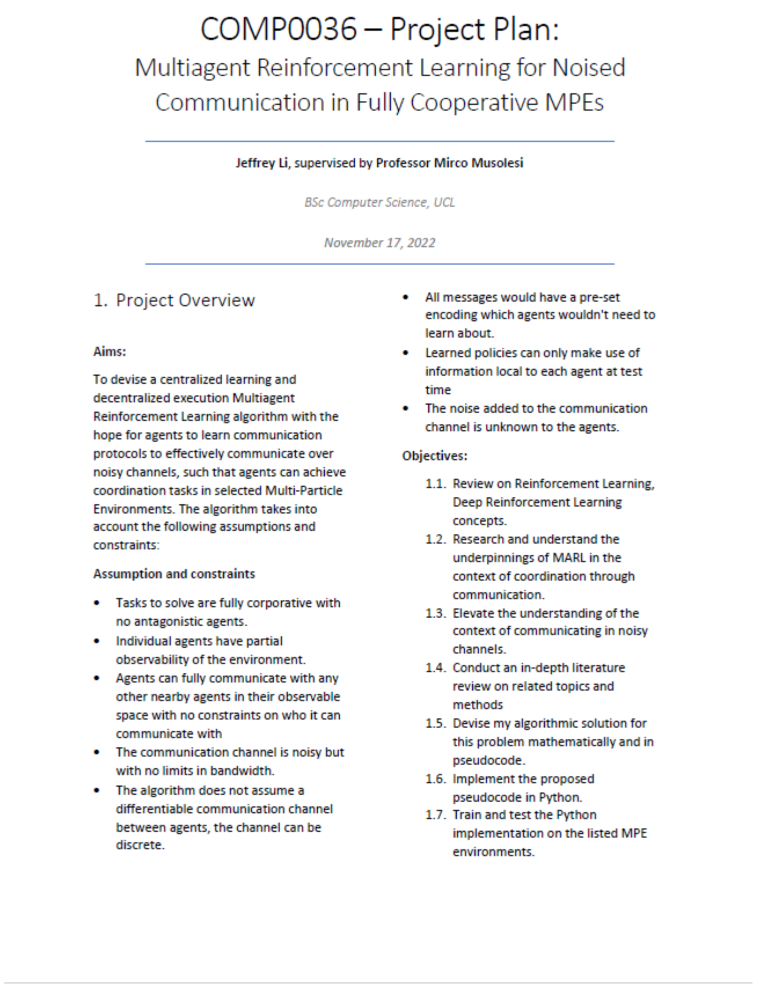
Zhu, C., Dastani, M., & Wang, S. (2022). *A Survey of Multi-Agent Reinforcement Learning with Communication* (arXiv:2203.08975). arXiv. http://arxiv.org/abs/2203.08975

10 Appendix

10.1 GitHub Repo

https://github.com/JeliPenguin/Bachelor-Project

10.2 Project Plan


COMP0036 – Project Plan:
Multiagent Reinforcement Learning for Noised Communication in Fully Cooperative MPEs

Jeffrey Li, supervised by Professor Mirco Musolesi

BSc Computer Science, UCL

November 17, 2022

1. Project Overview

Aims:

To devise a centralized learning and decentralized execution Multiagent Reinforcement Learning algorithm with the hope for agents to learn communication protocols to effectively communicate over noisy channels, such that agents can achieve coordination tasks in selected Multi-Particle Environments. The algorithm takes into account the following assumptions and constraints:

Assumption and constraints

- Tasks to solve are fully corporative with no antagonistic agents.
- Individual agents have partial observability of the environment.
- Agents can fully communicate with any other nearby agents in their observable space with no constraints on who it can communicate with
- The communication channel is noisy but with no limits in bandwidth.
- The algorithm does not assume a differentiable communication channel between agents, the channel can be discrete.
- All messages would have a pre-set encoding which agents wouldn't need to learn about.
- Learned policies can only make use of information local to each agent at test time
- The noise added to the communication channel is unknown to the agents.

Objectives:

1.1. Review on Reinforcement Learning, Deep Reinforcement Learning concepts.
1.2. Research and understand the underpinnings of MARL in the context of coordination through communication.
1.3. Elevate the understanding of the context of communicating in noisy channels.
1.4. Conduct an in-depth literature review on related topics and methods
1.5. Devise my algorithmic solution for this problem mathematically and in pseudocode.
1.6. Implement the proposed pseudocode in Python.
1.7. Train and test the Python implementation on the listed MPE environments.

1.8. Evaluate the success of implemented algorithms with other state-of-the-art MARL algorithms.

2. Expected deliverables

The expected deliverable would start with an overview of the problem to be solved, explaining its motivation and context.

This is followed by a broad survey summarizing core concepts of Reinforcement Learning, Deep Reinforcement Learning and Multiagent Reinforcement Learning for the more general readers. The literature survey would be backed by Python 3.x implementations for the following RL and Deep RL algorithms that are trained and tested in OpenAI Gym's toy environments:

- Value Iteration
- Policy Iteration
- Q Learning
- Deep Q Learning

The report would then have a greater focus on cooperative MARL in the context of coordination through communication with a literature review over an array of algorithmic approaches on similar problems.

I would then propose my algorithm for solving the proposed problem. This would include mathematical formulations as well as pseudocode and would also be backed by my Python 3.x implementation of the algorithm.

This implementation would be trained and tested on a selective of benchmark MPEs [1] as shown below with modification in adding additional noises to the agent's communication channels:

- Simple Speak Listener
- Simple Reference

And is then followed by an in-depth evaluation of its performance presented in the form of figures and graphs. This would be done by comparing results against the state-of-the-art algorithms for discrete communication listed below which would also be trained on the same MPEs:

- DiffDiscrete [2]
- RIAL [3]

The metrics used for comparison would be the mean episodic reward in training and testing. The results would be obtained on a varied number of cooperative agents with optimal (to my best ability) network hyper-parameters for each environment.

In conclusion, I will perform an analysis of the strengths and weaknesses of my proposed algorithm based on the evaluation and offer future work to be done as well as areas for improvement.

3. Work plan

- Project start to mid-Nov (4 weeks)
  - Complete reviewing, implementing and testing on Reinforcement and Deep Reinforcement Learning algorithms
- Mid-Nov to mid-Dec (4 weeks)
  - Complete the Project plan and Literature review on the proposed topic
  - Apply modification on selected MPEs and test the environments.
  - Come up with a framework for the algorithm
- Dec 19th to Early-Jan – Christmas Break (3 weeks)
  - Continue working on developing the algorithm
- Early-Jan to 18th Jan (2 weeks)
  - Work on completing the interim report
- **18th Jan - Interim Report Due**
- 18th Jan – Mid Mar (7 weeks)
  - Finalise mathematical formulation and pseudocode for my algorithm
  - Finish implementation of pseudocode in Python.

    - Obtain training and testing results of the implementation in the modified MPEs.
- Mid-Mar to Late-Mar (2 weeks)
    - Implement listed state-of-the-art algorithms.
    - Obtain training and testing results for these algorithms in the modified MPEs.
- Late-Mar to 26th April (5 weeks)
    - Work on completing the Final Project report
- **26th April - Project Submission**

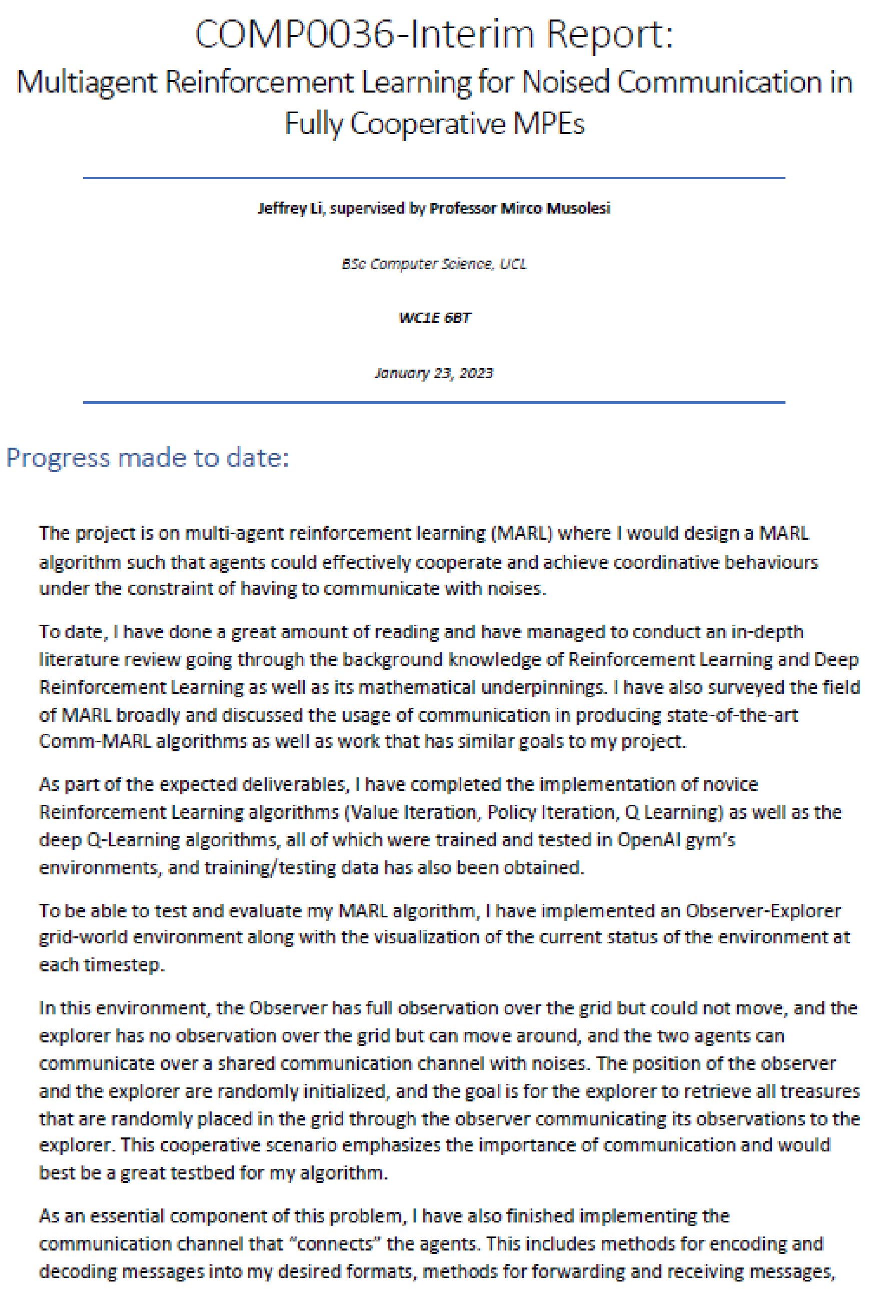
References:

[1] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments." arXiv, 2017. doi: 10.48550/ARXIV.1706.02275.

[2] Freed, B., Sartoretti, G., Hu, J., & Choset, H. (2020). Communication Learning via Backpropagation in Discrete Channels with Unknown Noise. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 34, Issue 05, pp. 7160–7168). Association for the Advancement of Artificial Intelligence (AAAI). https://doi.org/10.1609/aaai.v34i05.6205

[3] Foerster, J. N., Assael, Y. M., de Freitas, N., & Whiteson, S. (2016). Learning to Communicate with Deep Multi-Agent Reinforcement Learning (Version 2). arXiv. https://doi.org/10.48550/ARXIV.1605.0667

10.3 Interim Report



COMP0036-Interim Report:

Multiagent Reinforcement Learning for Noised Communication in Fully Cooperative MPEs

Jeffrey Li, supervised by Professor Mirco Musolesi

BSc Computer Science, UCL

WC1E 6BT

January 23, 2023

Progress made to date:

The project is on multi-agent reinforcement learning (MARL) where I would design a MARL algorithm such that agents could effectively cooperate and achieve coordinative behaviours under the constraint of having to communicate with noises.

To date, I have done a great amount of reading and have managed to conduct an in-depth literature review going through the background knowledge of Reinforcement Learning and Deep Reinforcement Learning as well as its mathematical underpinnings. I have also surveyed the field of MARL broadly and discussed the usage of communication in producing state-of-the-art Comm-MARL algorithms as well as work that has similar goals to my project.

As part of the expected deliverables, I have completed the implementation of novice Reinforcement Learning algorithms (Value Iteration, Policy Iteration, Q Learning) as well as the deep Q-Learning algorithms, all of which were trained and tested in OpenAI gym's environments, and training/testing data has also been obtained.

To be able to test and evaluate my MARL algorithm, I have implemented an Observer-Explorer grid-world environment along with the visualization of the current status of the environment at each timestep.

In this environment, the Observer has full observation over the grid but could not move, and the explorer has no observation over the grid but can move around, and the two agents can communicate over a shared communication channel with noises. The position of the observer and the explorer are randomly initialized, and the goal is for the explorer to retrieve all treasures that are randomly placed in the grid through the observer communicating its observations to the explorer. This cooperative scenario emphasizes the importance of communication and would best be a great testbed for my algorithm.

As an essential component of this problem, I have also finished implementing the communication channel that "connects" the agents. This includes methods for encoding and decoding messages into my desired formats, methods for forwarding and receiving messages, and the method for adding noises to the message of which currently has the Binary Symmetric channel implemented.

Finally, I have completed implementing a prototype Communication-MARL algorithm in the scenario of having one Observer and one Explorer. This has successfully converged while communicating under a channel with no noise added and the agents have learned to coordinate effectively, and I aim to further test this algorithms effectiveness by adding additional Explorers to the environment. I have also completed implementing a structural framework for training and testing Comm-MARL algorithms following similar interfaces to PettingZoo such that further improvements to the existing prototype as well as state-of-the-art algorithms can be easily trained and evaluated, and the environment could be easily customised.

Remaining work and work plan:

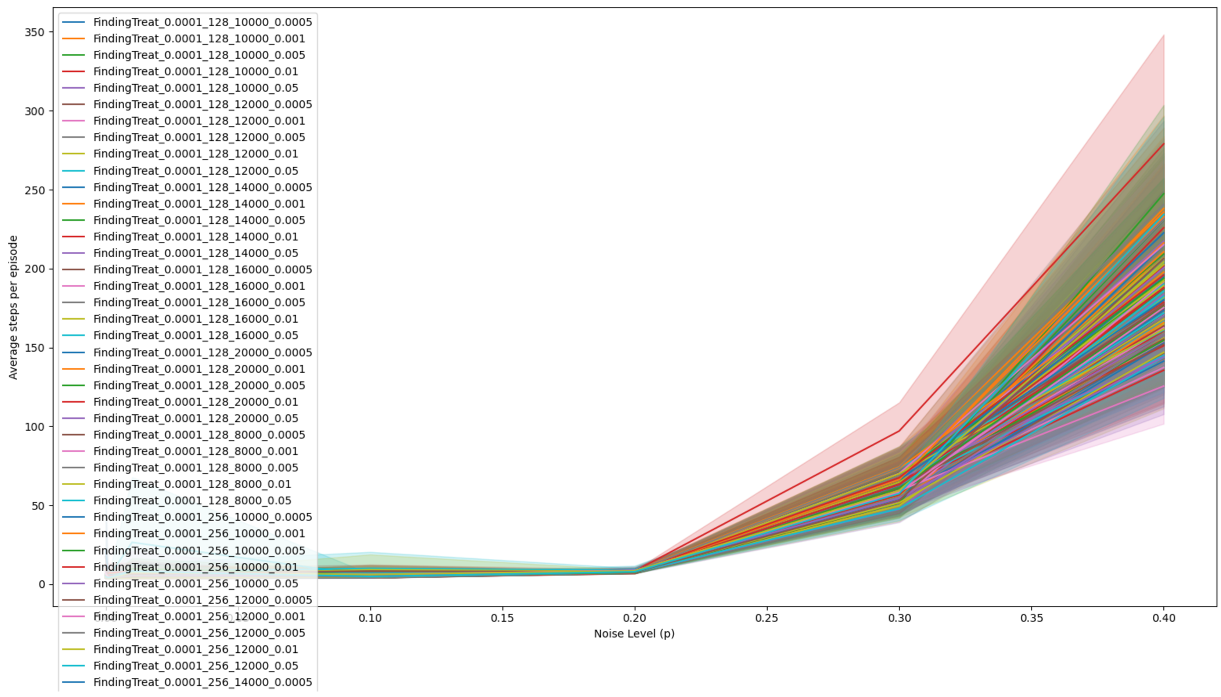
List of work to be done:

- Implement remaining noises to the communication channel
- Add in capability to the environment for a flexible number of agents to be added
- Adapt my prototype algorithm with the implemented noised communication channel
- Implement selected state-of-the-art Comm-MARL algorithms
- Train my algorithm and state-of-the-art algorithms, obtain training and testing results and evaluate my algorithm

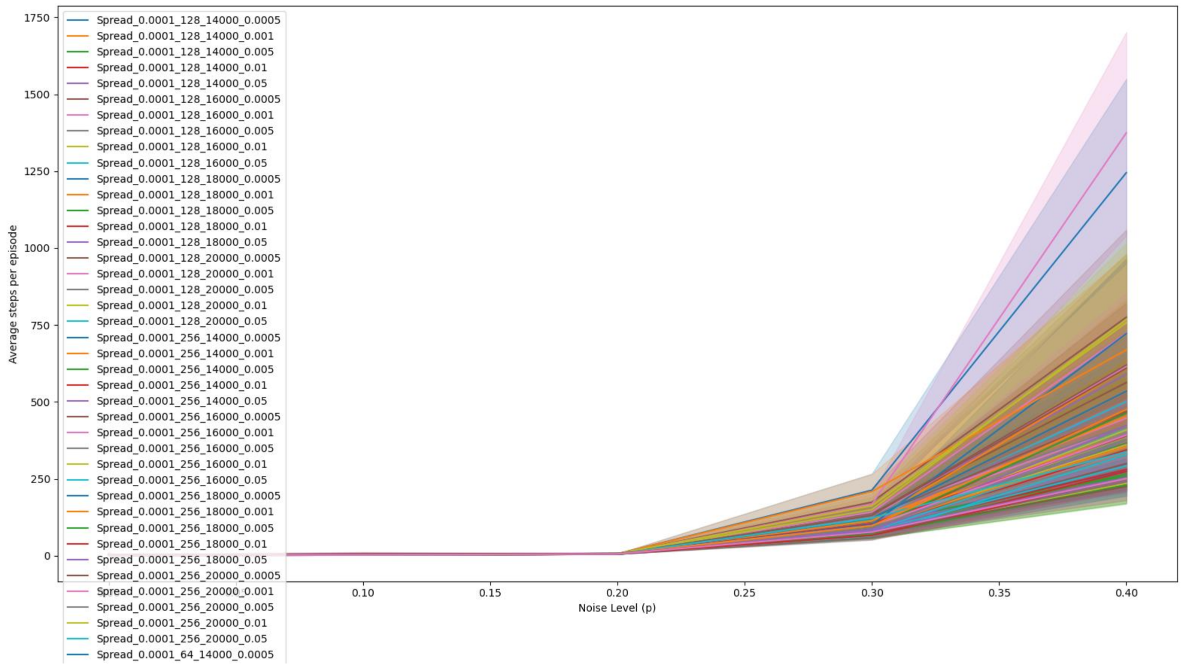
Work plan:

- Mid-Jan – Early Feb:
    - Refactor code to allow flexible number of agents to be added to the environment and trained
    - Complete implementation of noised communication channels
- Early Feb – Mid-Mar:
    - Implement my current idea of how to tackle communication under noises
    - Iteratively improve the idea with further reading and experimentation
    - Obtain training and testing results of the final version of my designed algorithm.
- Mid-Mar to Late-Mar:
    - Implement selected state-of-the-art Comm-MARL algorithms.
    - Obtain training and testing results for these algorithms.
- Late-Mar to 26th April:
    - Evaluate my designed algorithm
    - Work on completing the Final Project report
- **26th April - Project Submission**

10.4 Finding-Treats Full Hyperparameter Tuning Results

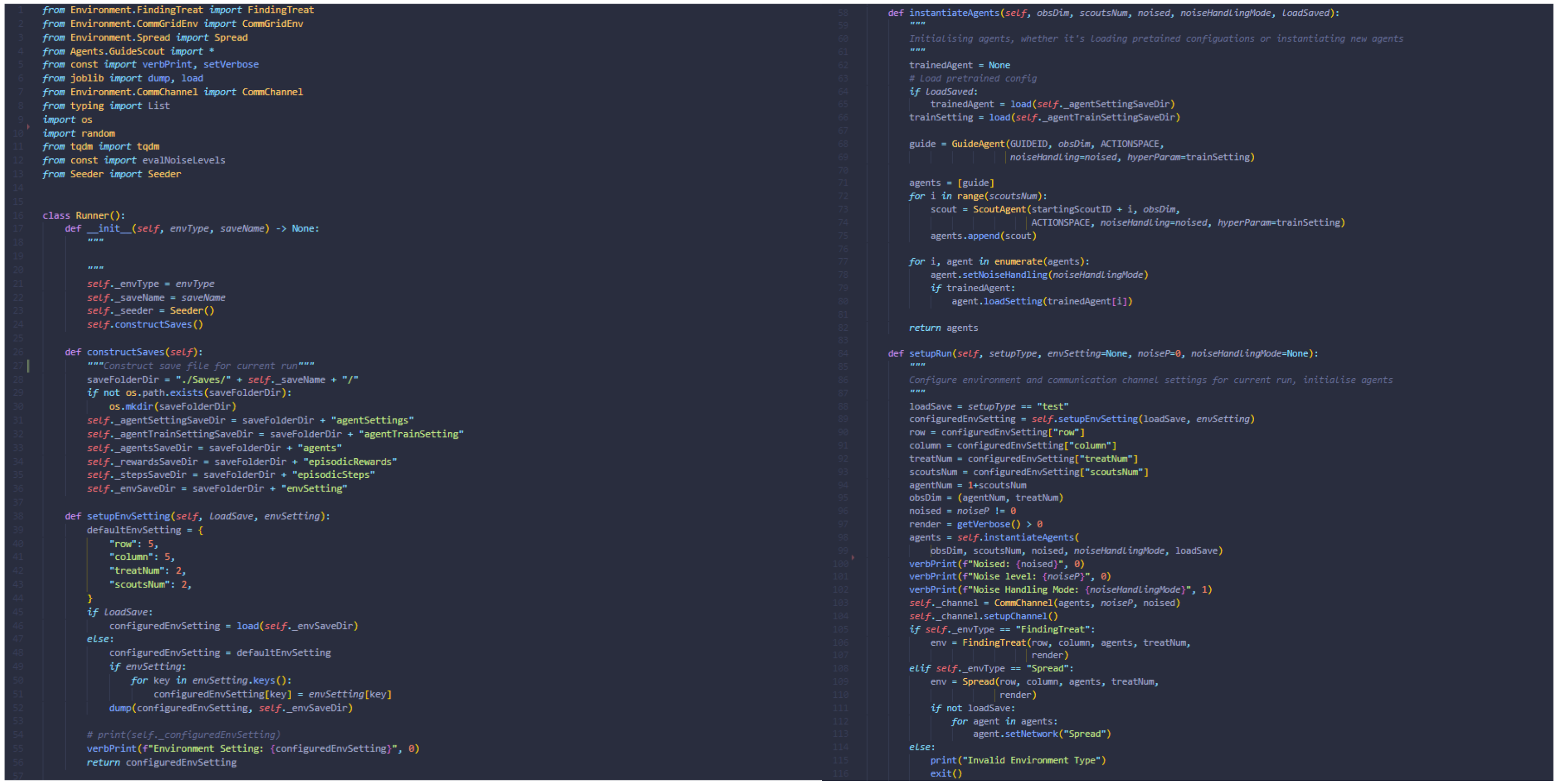


10.5 Spread Full Hyperparameter Tuning Results

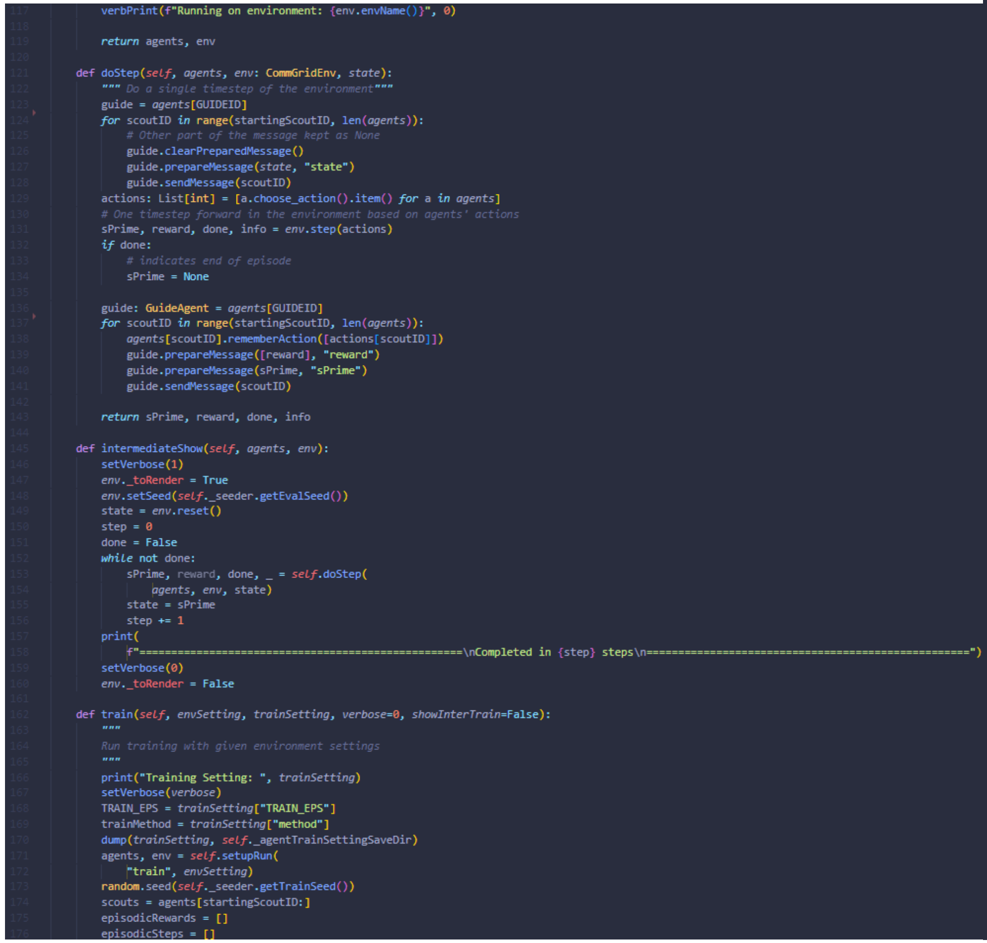


10.6 Code Listing

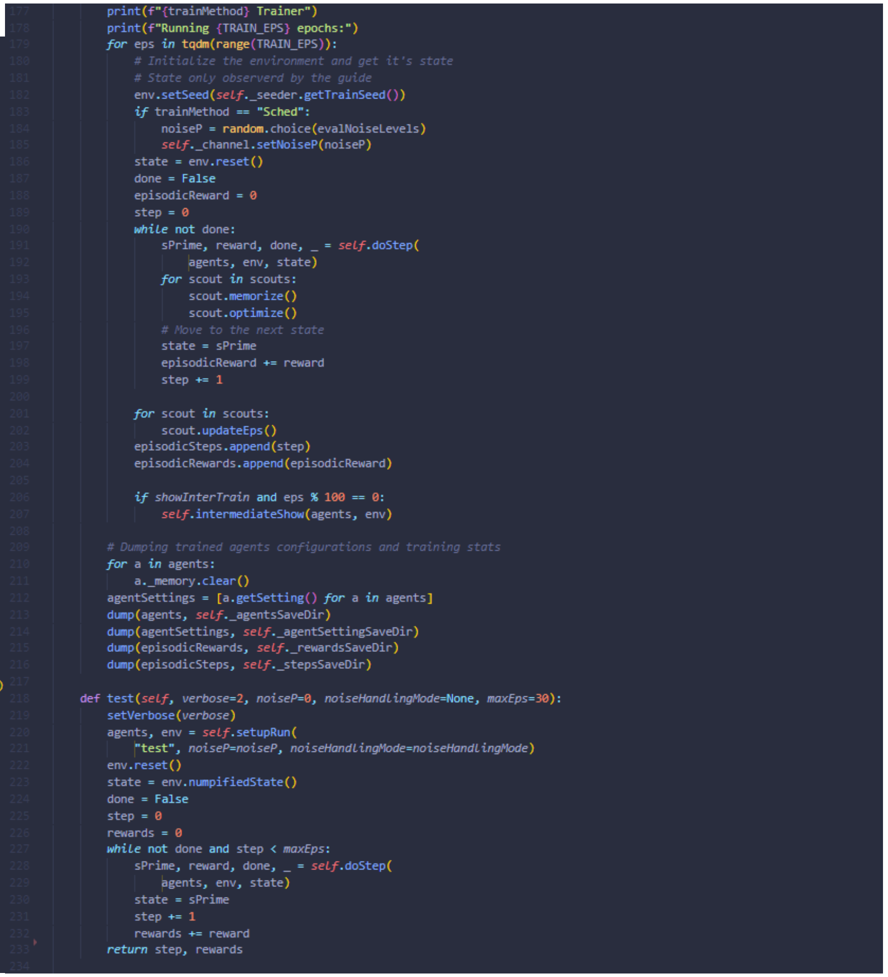
10.6.1 Runner.py



```python
from Environment.FindingTreat import FindingTreat
from Environment.CommGridEnv import CommGridEnv
from Environment.Spread import Spread
from Agents.GuideScout import *
from const import verbPrint, setVerbose
from joblib import dump, load
from Environment.CommChannel import CommChannel
from typing import List
import os
import random
from tqdm import tqdm
from const import evalNoiseLevels
from Seeder import Seeder


class Runner():
    def __init__(self, envType, saveName) -> None:
        """

        """
        self._envType = envType
        self._saveName = saveName
        self._seeder = Seeder()
        self.constructSaves()

    def constructSaves(self):
        """Construct save file for current run"""
        saveFolderDir = "./Saves/" + self._saveName + "/"
        if not os.path.exists(saveFolderDir):
            os.mkdir(saveFolderDir)
        self._agentSettingSaveDir = saveFolderDir + "agentSettings"
        self._agentTrainSettingSaveDir = saveFolderDir + "agentTrainSetting"
        self._agentsSaveDir = saveFolderDir + "agents"
        self._rewardsSaveDir = saveFolderDir + "episodicRewards"
        self._stepsSaveDir = saveFolderDir + "episodicSteps"
        self._envSaveDir = saveFolderDir + "envSetting"

    def setupEnvSetting(self, loadSave, envSetting):
        defaultEnvSetting = {
            "row": 5,
            "column": 5,
            "treatNum": 2,
            "scoutsNum": 2,
        }
        if loadSave:
            configuredEnvSetting = load(self._envSaveDir)
        else:
            configuredEnvSetting = defaultEnvSetting
            if envSetting:
                for key in envSetting.keys():
                    configuredEnvSetting[key] = envSetting[key]
            dump(configuredEnvSetting, self._envSaveDir)

        # print(self._configuredEnvSetting)
        verbPrint(f"Environment Setting: {configuredEnvSetting}", 0)
        return configuredEnvSetting

    def instantiateAgents(self, obsDim, scoutsNum, noised, noiseHandlingMode, loadSaved):
        """
        Initialising agents, whether it's loading pretained configuations or instantiating new agents
        """
        trainedAgent = None
        # Load pretrained config
        if loadSaved:
            trainedAgent = load(self._agentSettingSaveDir)
        trainSetting = load(self._agentTrainSettingSaveDir)

        guide = GuideAgent(GUIDEID, obsDim, ACTIONSPACE,
                           noiseHandling=noised, hyperParam=trainSetting)

        agents = [guide]
        for i in range(scoutsNum):
            scout = ScoutAgent(startingScoutID + i, obsDim,
                               ACTIONSPACE, noiseHandling=noised, hyperParam=trainSetting)
            agents.append(scout)

        for i, agent in enumerate(agents):
            agent.setNoiseHandling(noiseHandlingMode)
            if trainedAgent:
                agent.loadSetting(trainedAgent[i])

        return agents

    def setupRun(self, setupType, envSetting=None, noiseP=0, noiseHandlingMode=None):
        """
        Configure environment and communication channel settings for current run, initialise agents
        """
        loadSave = setupType == "test"
        configuredEnvSetting = self.setupEnvSetting(loadSave, envSetting)
        row = configuredEnvSetting["row"]
        column = configuredEnvSetting["column"]
        treatNum = configuredEnvSetting["treatNum"]
        scoutsNum = configuredEnvSetting["scoutsNum"]
        agentNum = 1+scoutsNum
        obsDim = (agentNum, treatNum)
        noised = noiseP != 0
        render = getVerbose() > 0
        agents = self.instantiateAgents(
            obsDim, scoutsNum, noised, noiseHandlingMode, loadSave)
        verbPrint(f"Noised: {noised}", 0)
        verbPrint(f"Noise level: {noiseP}", 0)
        verbPrint(f"Noise Handling Mode: {noiseHandlingMode}", 1)
        self._channel = CommChannel(agents, noiseP, noised)
        self._channel.setupChannel()
        if self._envType == "FindingTreat":
            env = FindingTreat(row, column, agents, treatNum,
                               render)
        elif self._envType == "Spread":
            env = Spread(row, column, agents, treatNum,
                         render)
            if not loadSave:
                for agent in agents:
                    agent.setNetwork("Spread")
        else:
            print("Invalid Environment Type")
            exit()
```

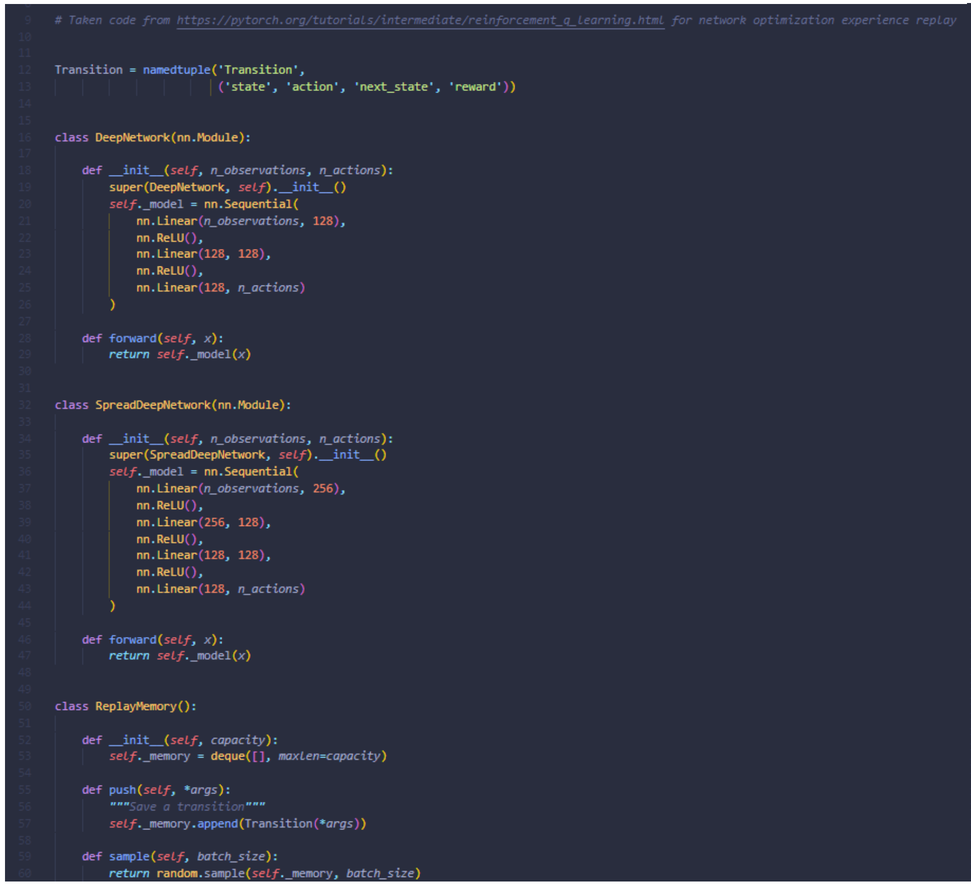


```python
        verbPrint(f"Running on environment: {env.envName()}", 0)

        return agents, env

    def doStep(self, agents, env: CommGridEnv, state):
        """ Do a single timestep of the environment"""
        guide = agents[GUIDEID]
        for scoutID in range(startingScoutID, len(agents)):
            # Other part of the message kept as None
            guide.clearPreparedMessage()
            guide.prepareMessage(state, "state")
            guide.sendMessage(scoutID)
        actions: List[int] = [a.choose_action().item() for a in agents]
        # One timestep forward in the environment based on agents' actions
        sPrime, reward, done, info = env.step(actions)
        if done:
            # indicates end of episode
            sPrime = None

        guide: GuideAgent = agents[GUIDEID]
        for scoutID in range(startingScoutID, len(agents)):
            agents[scoutID].rememberAction([actions[scoutID]])
            guide.prepareMessage([reward], "reward")
            guide.prepareMessage(sPrime, "sPrime")
            guide.sendMessage(scoutID)

        return sPrime, reward, done, info

    def intermediateShow(self, agents, env):
        setVerbose(1)
        env._toRender = True
        env.setSeed(self._seeder.getEvalSeed())
        state = env.reset()
        step = 0
        done = False
        while not done:
            sPrime, reward, done, _ = self.doStep(
                agents, env, state)
            state = sPrime
            step += 1
        print(
            f"====================================================\nCompleted in {step} steps\n====================================================")
        setVerbose(0)
        env._toRender = False

    def train(self, envSetting, trainSetting, verbose=0, showInterTrain=False):
        """
        Run training with given environment settings
        """
        print("Training Setting: ", trainSetting)
        setVerbose(verbose)
        TRAIN_EPS = trainSetting["TRAIN_EPS"]
        trainMethod = trainSetting["method"]
        dump(trainSetting, self._agentTrainSettingSaveDir)
        agents, env = self.setupRun(
            "train", envSetting)
        random.seed(self._seeder.getTrainSeed())
        scouts = agents[startingScoutID:]
        episodicRewards = []
        episodicSteps = []
```

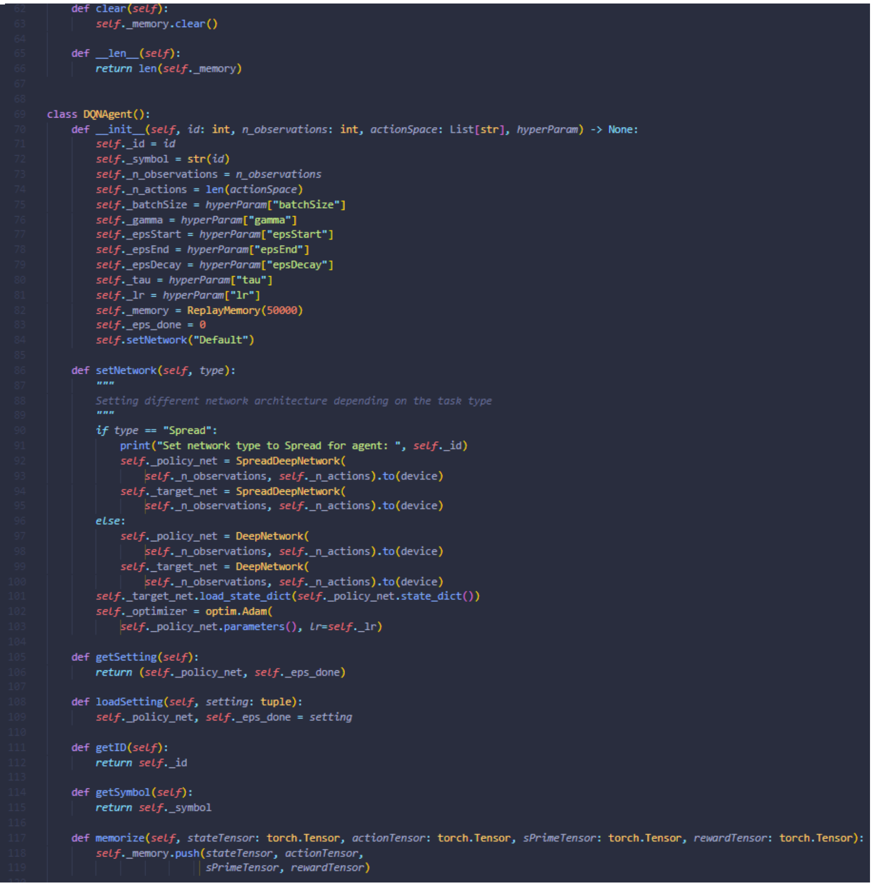


```python
        print(f"{trainMethod} Trainer")
        print(f"Running {TRAIN_EPS} epochs:")
        for eps in tqdm(range(TRAIN_EPS)):
            # Initialize the environment and get it's state
            # State only observerd by the guide
            env.setSeed(self._seeder.getTrainSeed())
            if trainMethod == "Sched":
                noiseP = random.choice(evalNoiseLevels)
                self._channel.setNoiseP(noiseP)
            state = env.reset()
            done = False
            episodicReward = 0
            step = 0
            while not done:
                sPrime, reward, done, _ = self.doStep(
                    agents, env, state)
                for scout in scouts:
                    scout.memorize()
                    scout.optimize()
                # Move to the next state
                state = sPrime
                episodicReward += reward
                step += 1

            for scout in scouts:
                scout.updateEps()
            episodicSteps.append(step)
            episodicRewards.append(episodicReward)

            if showInterTrain and eps % 100 == 0:
                self.intermediateShow(agents, env)

        # Dumping trained agents configurations and training stats
        for a in agents:
            a._memory.clear()
        agentSettings = [a.getSetting() for a in agents]
        dump(agents, self._agentsSaveDir)
        dump(agentSettings, self._agentSettingSaveDir)
        dump(episodicRewards, self._rewardsSaveDir)
        dump(episodicSteps, self._stepsSaveDir)

    def test(self, verbose=2, noiseP=0, noiseHandlingMode=None, maxEps=30):
        setVerbose(verbose)
        agents, env = self.setupRun(
            "test", noiseP=noiseP, noiseHandlingMode=noiseHandlingMode)
        env.reset()
        state = env.numpifiedState()
        done = False
        step = 0
        rewards = 0
        while not done and step < maxEps:
            sPrime, reward, done, _ = self.doStep(
                agents, env, state)
            state = sPrime
            step += 1
            rewards += reward
        return step, rewards
```

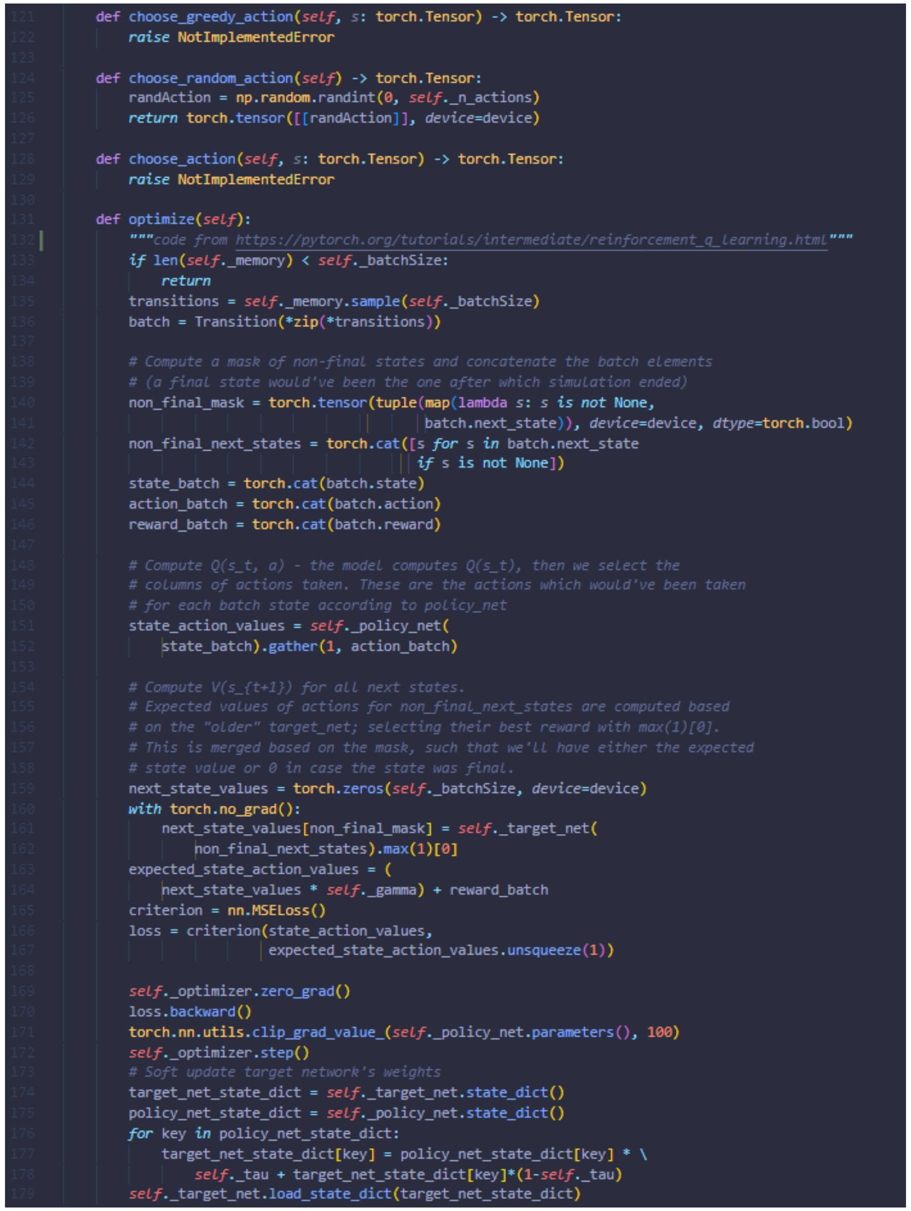
10.6.2 DQN.py



```python
# Taken code from https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html for network optimization experience replay


Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))


class DeepNetwork(nn.Module):

    def __init__(self, n_observations, n_actions):
        super(DeepNetwork, self).__init__()
        self._model = nn.Sequential(
            nn.Linear(n_observations, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        )

    def forward(self, x):
        return self._model(x)


class SpreadDeepNetwork(nn.Module):

    def __init__(self, n_observations, n_actions):
        super(SpreadDeepNetwork, self).__init__()
        self._model = nn.Sequential(
            nn.Linear(n_observations, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        )

    def forward(self, x):
        return self._model(x)


class ReplayMemory():

    def __init__(self, capacity):
        self._memory = deque([], maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self._memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self._memory, batch_size)
```

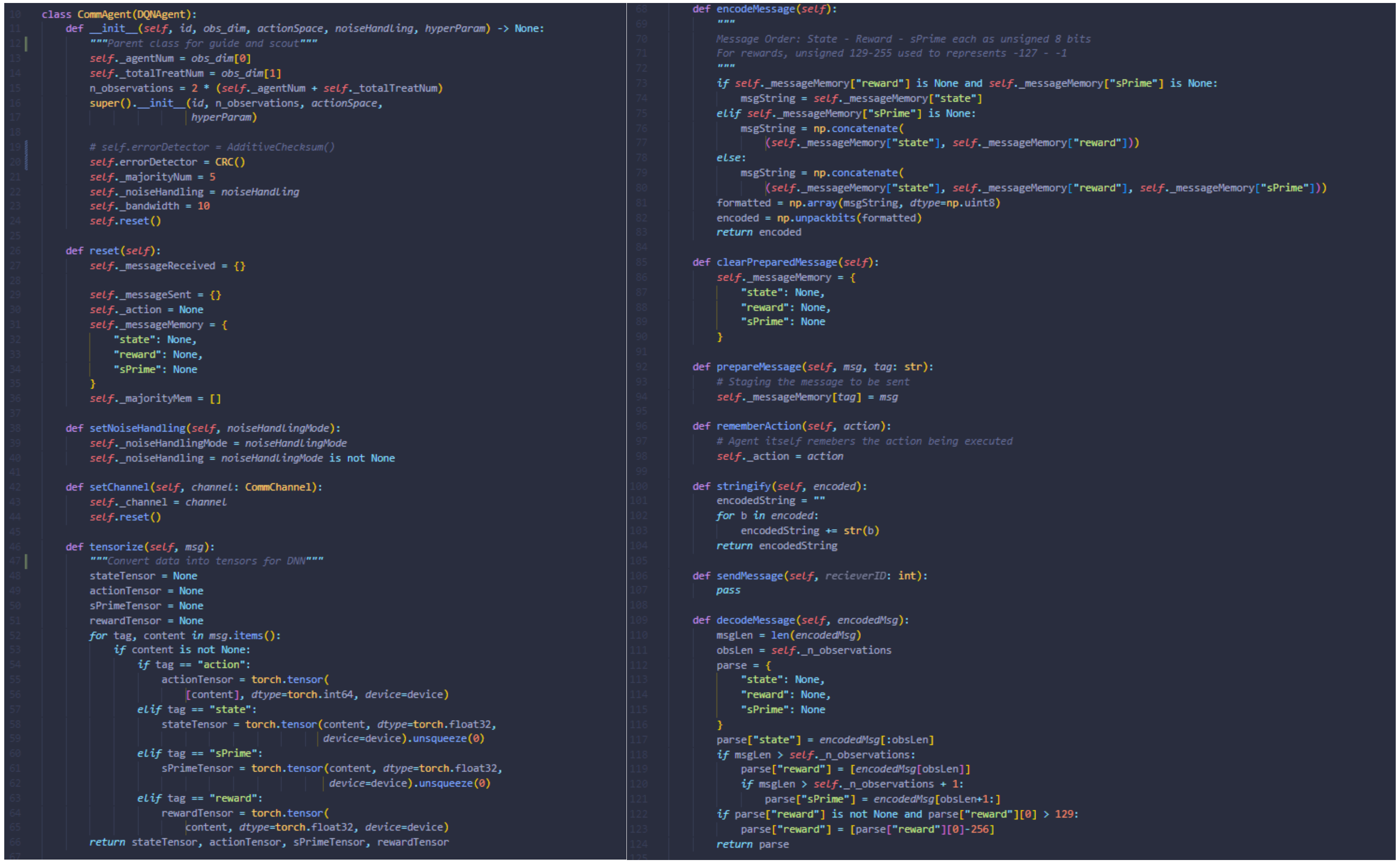


```python
    def clear(self):
        self._memory.clear()

    def __len__(self):
        return len(self._memory)


class DQNAgent():
    def __init__(self, id: int, n_observations: int, actionSpace: List[str], hyperParam) -> None:
        self._id = id
        self._symbol = str(id)
        self._n_observations = n_observations
        self._n_actions = len(actionSpace)
        self._batchSize = hyperParam["batchSize"]
        self._gamma = hyperParam["gamma"]
        self._epsStart = hyperParam["epsStart"]
        self._epsEnd = hyperParam["epsEnd"]
        self._epsDecay = hyperParam["epsDecay"]
        self._tau = hyperParam["tau"]
        self._lr = hyperParam["lr"]
        self._memory = ReplayMemory(50000)
        self._eps_done = 0
        self.setNetwork("Default")

    def setNetwork(self, type):
        """
        Setting different network architecture depending on the task type
        """
        if type == "Spread":
            print("Set network type to Spread for agent: ", self._id)
            self._policy_net = SpreadDeepNetwork(
                self._n_observations, self._n_actions).to(device)
            self._target_net = SpreadDeepNetwork(
                self._n_observations, self._n_actions).to(device)
        else:
            self._policy_net = DeepNetwork(
                self._n_observations, self._n_actions).to(device)
            self._target_net = DeepNetwork(
                self._n_observations, self._n_actions).to(device)
        self._target_net.load_state_dict(self._policy_net.state_dict())
        self._optimizer = optim.Adam(
            self._policy_net.parameters(), lr=self._lr)

    def getSetting(self):
        return (self._policy_net, self._eps_done)

    def loadSetting(self, setting: tuple):
        self._policy_net, self._eps_done = setting

    def getID(self):
        return self._id

    def getSymbol(self):
        return self._symbol

    def memorize(self, stateTensor: torch.Tensor, actionTensor: torch.Tensor, sPrimeTensor: torch.Tensor, rewardTensor: torch.Tensor):
        self._memory.push(stateTensor, actionTensor,
                          sPrimeTensor, rewardTensor)
```

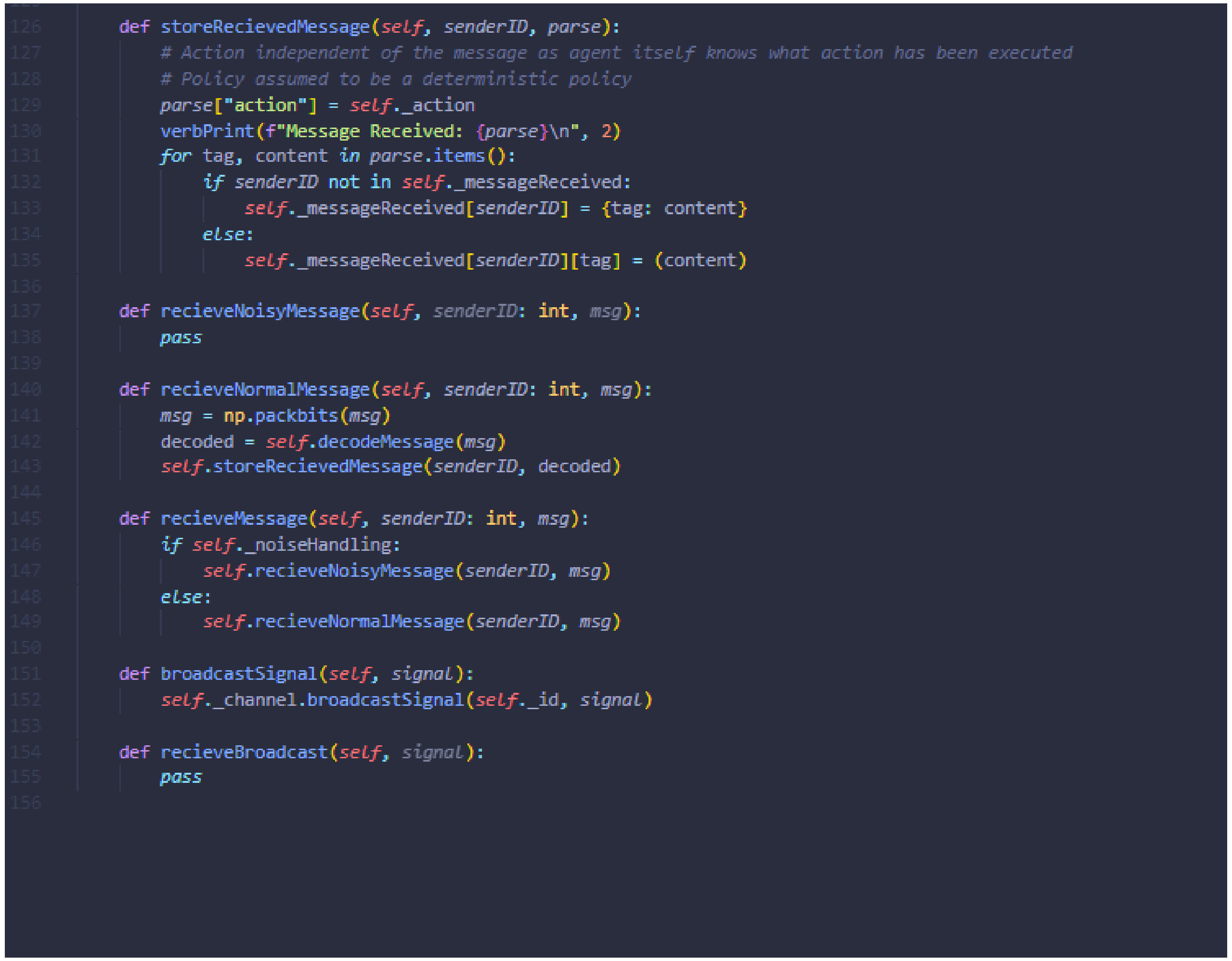


```python
    def choose_greedy_action(self, s: torch.Tensor) -> torch.Tensor:
        raise NotImplementedError

    def choose_random_action(self) -> torch.Tensor:
        randAction = np.random.randint(0, self._n_actions)
        return torch.tensor([[randAction]], device=device)

    def choose_action(self, s: torch.Tensor) -> torch.Tensor:
        raise NotImplementedError

    def optimize(self):
        """code from https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html"""
        if len(self._memory) < self._batchSize:
            return
        transitions = self._memory.sample(self._batchSize)
        batch = Transition(*zip(*transitions))

        # Compute a mask of non-final states and concatenate the batch elements
        # (a final state would've been the one after which simulation ended)
        non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                                batch.next_state)), device=device, dtype=torch.bool)
        non_final_next_states = torch.cat([s for s in batch.next_state
                                               if s is not None])
        state_batch = torch.cat(batch.state)
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)

        # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
        # columns of actions taken. These are the actions which would've been taken
        # for each batch state according to policy_net
        state_action_values = self._policy_net(
            state_batch).gather(1, action_batch)

        # Compute V(s_{t+1}) for all next states.
        # Expected values of actions for non_final_next_states are computed based
        # on the "older" target_net; selecting their best reward with max(1)[0].
        # This is merged based on the mask, such that we'll have either the expected
        # state value or 0 in case the state was final.
        next_state_values = torch.zeros(self._batchSize, device=device)
        with torch.no_grad():
            next_state_values[non_final_mask] = self._target_net(
                non_final_next_states).max(1)[0]
        expected_state_action_values = (
            next_state_values * self._gamma) + reward_batch
        criterion = nn.MSELoss()
        loss = criterion(state_action_values,
                         expected_state_action_values.unsqueeze(1))

        self._optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_value_(self._policy_net.parameters(), 100)
        self._optimizer.step()
        # Soft update target network's weights
        target_net_state_dict = self._target_net.state_dict()
        policy_net_state_dict = self._policy_net.state_dict()
        for key in policy_net_state_dict:
            target_net_state_dict[key] = policy_net_state_dict[key] * \
                self._tau + target_net_state_dict[key]*(1-self._tau)
        self._target_net.load_state_dict(target_net_state_dict)
```
]

10.6.3 CommAgent.py

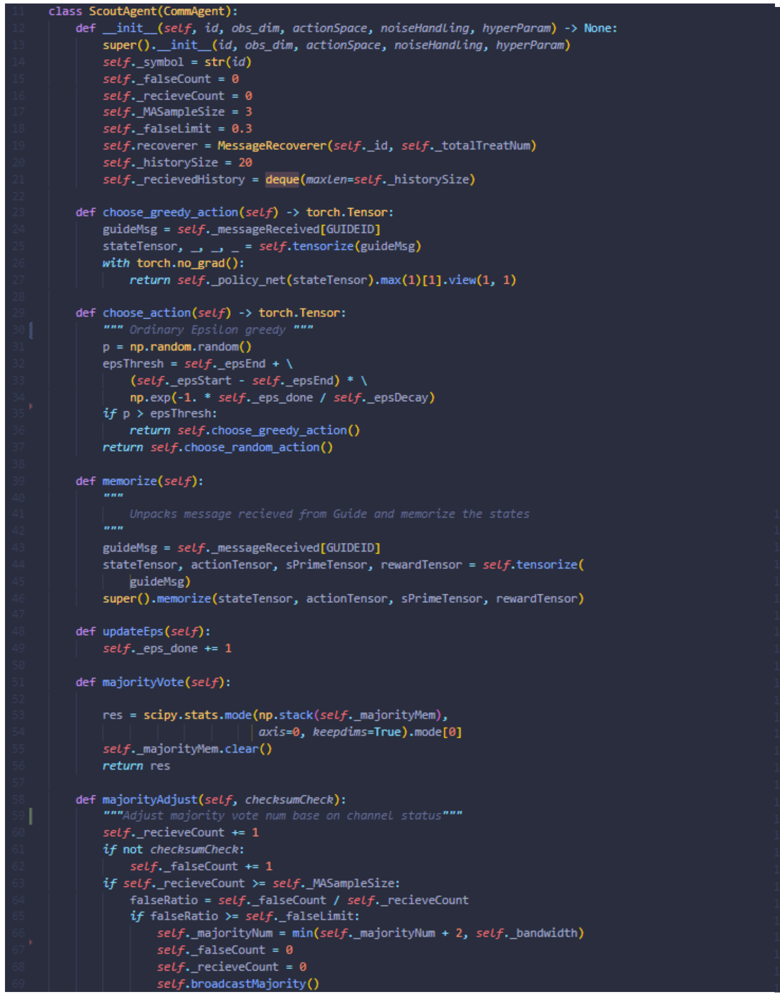


```python
class CommAgent(DQNAgent):
    def __init__(self, id, obs_dim, actionSpace, noiseHandling, hyperParam) -> None:
        """Parent class for guide and scout"""
        self._agentNum = obs_dim[0]
        self._totalTreatNum = obs_dim[1]
        n_observations = 2 * (self._agentNum + self._totalTreatNum)
        super().__init__(id, n_observations, actionSpace,
                         hyperParam)

        # self.errorDetector = AdditiveChecksum()
        self.errorDetector = CRC()
        self._majorityNum = 5
        self._noiseHandling = noiseHandling
        self._bandwidth = 10
        self.reset()

    def reset(self):
        self._messageReceived = {}

        self._messageSent = {}
        self._action = None
        self._messageMemory = {
            "state": None,
            "reward": None,
            "sPrime": None
        }
        self._majorityMem = []

    def setNoiseHandling(self, noiseHandlingMode):
        self._noiseHandlingMode = noiseHandlingMode
        self._noiseHandling = noiseHandlingMode is not None

    def setChannel(self, channel: CommChannel):
        self._channel = channel
        self.reset()

    def tensorize(self, msg):
        """Convert data into tensors for DNN"""
        stateTensor = None
        actionTensor = None
        sPrimeTensor = None
        rewardTensor = None
        for tag, content in msg.items():
            if content is not None:
                if tag == "action":
                    actionTensor = torch.tensor(
                        [content], dtype=torch.int64, device=device)
                elif tag == "state":
                    stateTensor = torch.tensor(content, dtype=torch.float32,
                                               device=device).unsqueeze(0)
                elif tag == "sPrime":
                    sPrimeTensor = torch.tensor(content, dtype=torch.float32,
                                                device=device).unsqueeze(0)
                elif tag == "reward":
                    rewardTensor = torch.tensor(
                        content, dtype=torch.float32, device=device)
        return stateTensor, actionTensor, sPrimeTensor, rewardTensor

    def encodeMessage(self):
        """
        Message Order: State - Reward - sPrime each as unsigned 8 bits
        For rewards, unsigned 129-255 used to represents -127 - -1
        """
        if self._messageMemory["reward"] is None and self._messageMemory["sPrime"] is None:
            msgString = self._messageMemory["state"]
        elif self._messageMemory["sPrime"] is None:
            msgString = np.concatenate(
                (self._messageMemory["state"], self._messageMemory["reward"]))
        else:
            msgString = np.concatenate(
                (self._messageMemory["state"], self._messageMemory["reward"], self._messageMemory["sPrime"]))
        formatted = np.array(msgString, dtype=np.uint8)
        encoded = np.unpackbits(formatted)
        return encoded

    def clearPreparedMessage(self):
        self._messageMemory = {
            "state": None,
            "reward": None,
            "sPrime": None
        }

    def prepareMessage(self, msg, tag: str):
        # Staging the message to be sent
        self._messageMemory[tag] = msg

    def rememberAction(self, action):
        # Agent itself remebers the action being executed
        self._action = action

    def stringify(self, encoded):
        encodedString = ""
        for b in encoded:
            encodedString += str(b)
        return encodedString

    def sendMessage(self, recieverID: int):
        pass

    def decodeMessage(self, encodedMsg):
        msgLen = len(encodedMsg)
        obsLen = self._n_observations
        parse = {
            "state": None,
            "reward": None,
            "sPrime": None
        }
        parse["state"] = encodedMsg[:obsLen]
        if msgLen > self._n_observations:
            parse["reward"] = [encodedMsg[obsLen]]
            if msgLen > self._n_observations + 1:
                parse["sPrime"] = encodedMsg[obsLen+1:]
        if parse["reward"] is not None and parse["reward"][0] > 129:
            parse["reward"] = [parse["reward"][0]-256]
        return parse
```

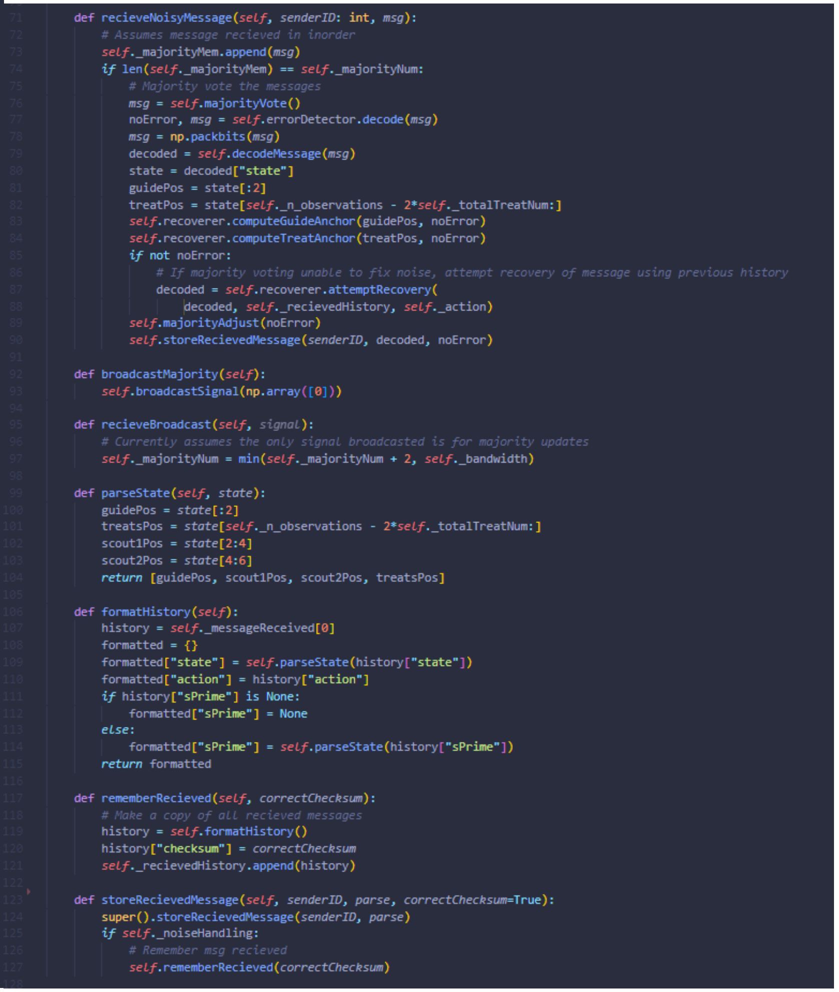


```python
    def storeRecievedMessage(self, senderID, parse):
        # Action independent of the message as agent itself knows what action has been executed
        # Policy assumed to be a deterministic policy
        parse["action"] = self._action
        verbPrint(f"Message Received: {parse}\n", 2)
        for tag, content in parse.items():
            if senderID not in self._messageReceived:
                self._messageReceived[senderID] = {tag: content}
            else:
                self._messageReceived[senderID][tag] = (content)

    def recieveNoisyMessage(self, senderID: int, msg):
        pass

    def recieveNormalMessage(self, senderID: int, msg):
        msg = np.packbits(msg)
        decoded = self.decodeMessage(msg)
        self.storeRecievedMessage(senderID, decoded)

    def recieveMessage(self, senderID: int, msg):
        if self._noiseHandling:
            self.recieveNoisyMessage(senderID, msg)
        else:
            self.recieveNormalMessage(senderID, msg)

    def broadcastSignal(self, signal):
        self._channel.broadcastSignal(self._id, signal)

    def recieveBroadcast(self, signal):
        pass
```

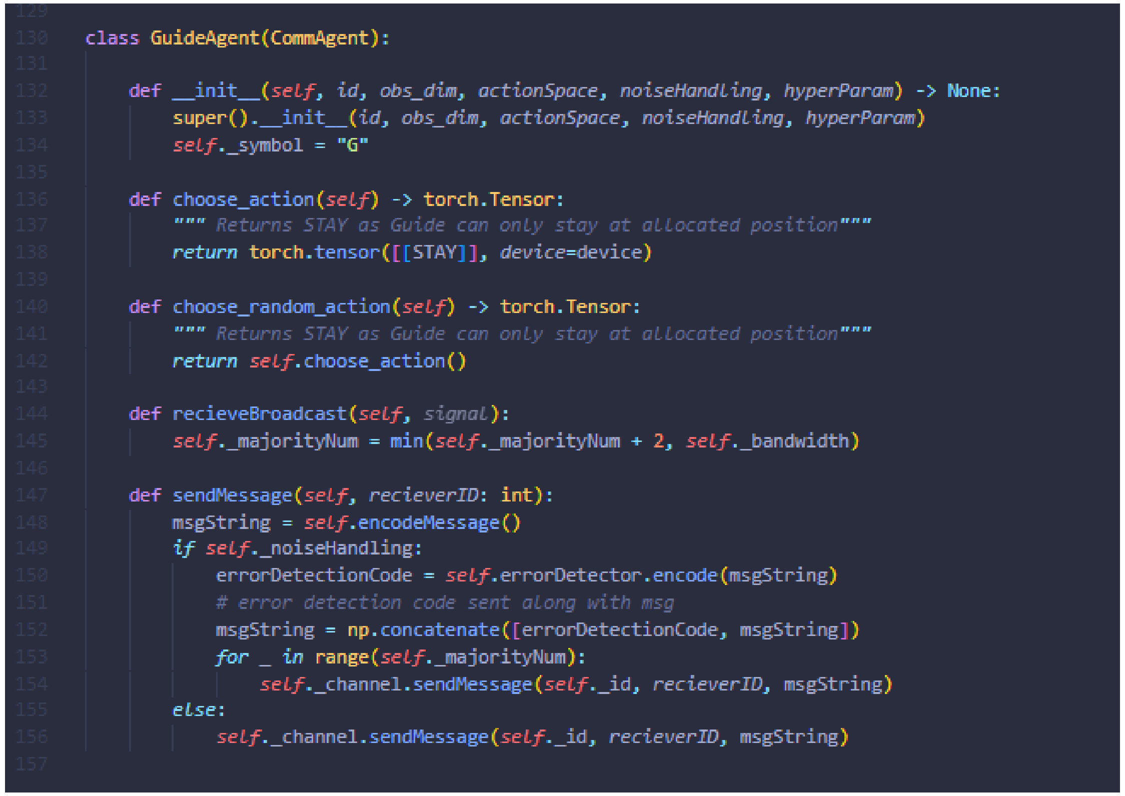
10.6.4 GuideScout.py



```python
class ScoutAgent(CommAgent):
    def __init__(self, id, obs_dim, actionSpace, noiseHandling, hyperParam) -> None:
        super().__init__(id, obs_dim, actionSpace, noiseHandling, hyperParam)
        self._symbol = str(id)
        self._falseCount = 0
        self._recieveCount = 0
        self._MASampleSize = 3
        self._falseLimit = 0.3
        self.recoverer = MessageRecoverer(self._id, self._totalTreatNum)
        self._historySize = 20
        self._recievedHistory = deque(maxlen=self._historySize)

    def choose_greedy_action(self) -> torch.Tensor:
        guideMsg = self._messageReceived[GUIDEID]
        stateTensor, _, _, _ = self.tensorize(guideMsg)
        with torch.no_grad():
            return self._policy_net(stateTensor).max(1)[1].view(1, 1)

    def choose_action(self) -> torch.Tensor:
        """ Ordinary Epsilon greedy """
        p = np.random.random()
        epsThresh = self._epsEnd + \
            (self._epsStart - self._epsEnd) * \
            np.exp(-1. * self._eps_done / self._epsDecay)
        if p > epsThresh:
            return self.choose_greedy_action()
        return self.choose_random_action()

    def memorize(self):
        """
            Unpacks message recieved from Guide and memorize the states
        """
        guideMsg = self._messageReceived[GUIDEID]
        stateTensor, actionTensor, sPrimeTensor, rewardTensor = self.tensorize(
            guideMsg)
        super().memorize(stateTensor, actionTensor, sPrimeTensor, rewardTensor)

    def updateEps(self):
        self._eps_done += 1

    def majorityVote(self):

        res = scipy.stats.mode(np.stack(self._majorityMem),
                               axis=0, keepdims=True).mode[0]
        self._majorityMem.clear()
        return res

    def majorityAdjust(self, checksumCheck):
        """Adjust majority vote num base on channel status"""
        self._recieveCount += 1
        if not checksumCheck:
            self._falseCount += 1
        if self._recieveCount >= self._MASampleSize:
            falseRatio = self._falseCount / self._recieveCount
            if falseRatio >= self._falseLimit:
                self._majorityNum = min(self._majorityNum + 2, self._bandwidth)
                self._falseCount = 0
                self._recieveCount = 0
                self.broadcastMajority()
```

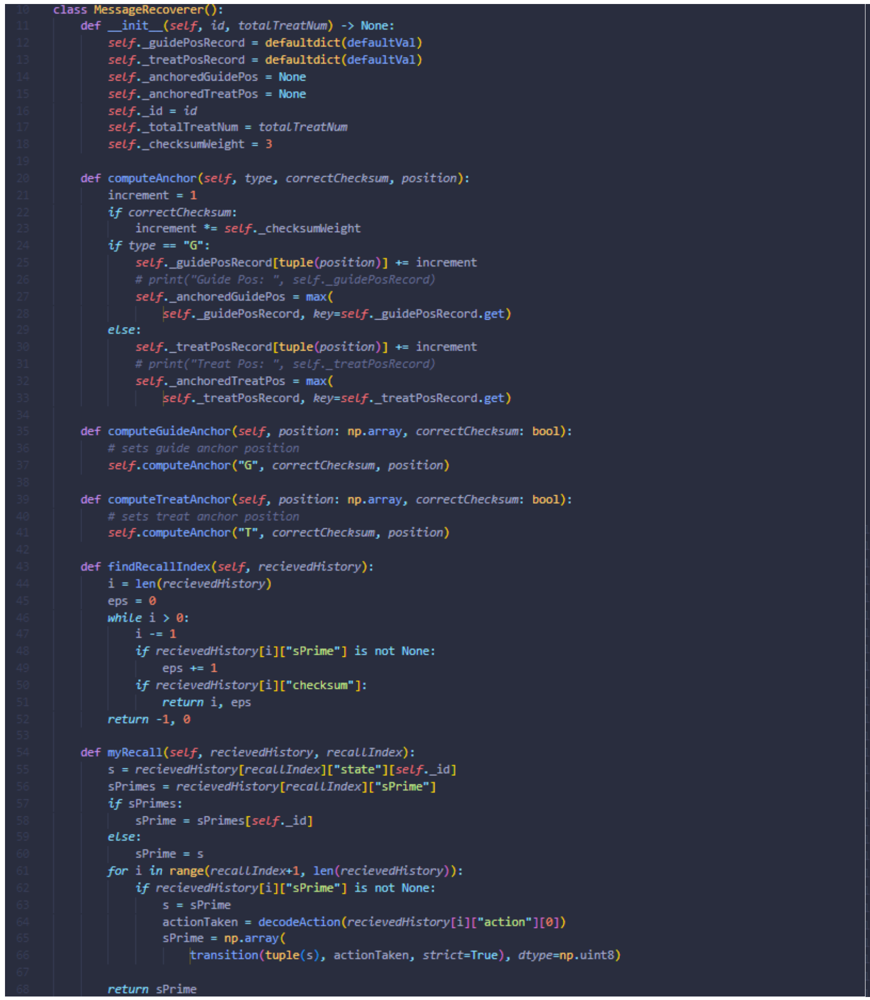


```python
    def recieveNoisyMessage(self, senderID: int, msg):
        # Assumes message recieved in inorder
        self._majorityMem.append(msg)
        if len(self._majorityMem) == self._majorityNum:
            # Majority vote the messages
            msg = self.majorityVote()
            noError, msg = self.errorDetector.decode(msg)
            msg = np.packbits(msg)
            decoded = self.decodeMessage(msg)
            state = decoded["state"]
            guidePos = state[:2]
            treatPos = state[self._n_observations - 2*self._totalTreatNum:]
            self.recoverer.computeGuideAnchor(guidePos, noError)
            self.recoverer.computeTreatAnchor(treatPos, noError)
            if not noError:
                # If majority voting unable to fix noise, attempt recovery of message using previous history
                decoded = self.recoverer.attemptRecovery(
                    decoded, self._recievedHistory, self._action)
            self.majorityAdjust(noError)
            self.storeRecievedMessage(senderID, decoded, noError)

    def broadcastMajority(self):
        self.broadcastSignal(np.array([0]))

    def recieveBroadcast(self, signal):
        # Currently assumes the only signal broadcasted is for majority updates
        self._majorityNum = min(self._majorityNum + 2, self._bandwidth)

    def parseState(self, state):
        guidePos = state[:2]
        treatsPos = state[self._n_observations - 2*self._totalTreatNum:]
        scout1Pos = state[2:4]
        scout2Pos = state[4:6]
        return [guidePos, scout1Pos, scout2Pos, treatsPos]

    def formatHistory(self):
        history = self._messageReceived[0]
        formatted = {}
        formatted["state"] = self.parseState(history["state"])
        formatted["action"] = history["action"]
        if history["sPrime"] is None:
            formatted["sPrime"] = None
        else:
            formatted["sPrime"] = self.parseState(history["sPrime"])
        return formatted

    def rememberRecieved(self, correctChecksum):
        # Make a copy of all recieved messages
        history = self.formatHistory()
        history["checksum"] = correctChecksum
        self._recievedHistory.append(history)

    def storeRecievedMessage(self, senderID, parse, correctChecksum=True):
        super().storeRecievedMessage(senderID, parse)
        if self._noiseHandling:
            # Remember msg recieved
            self.rememberRecieved(correctChecksum)
```

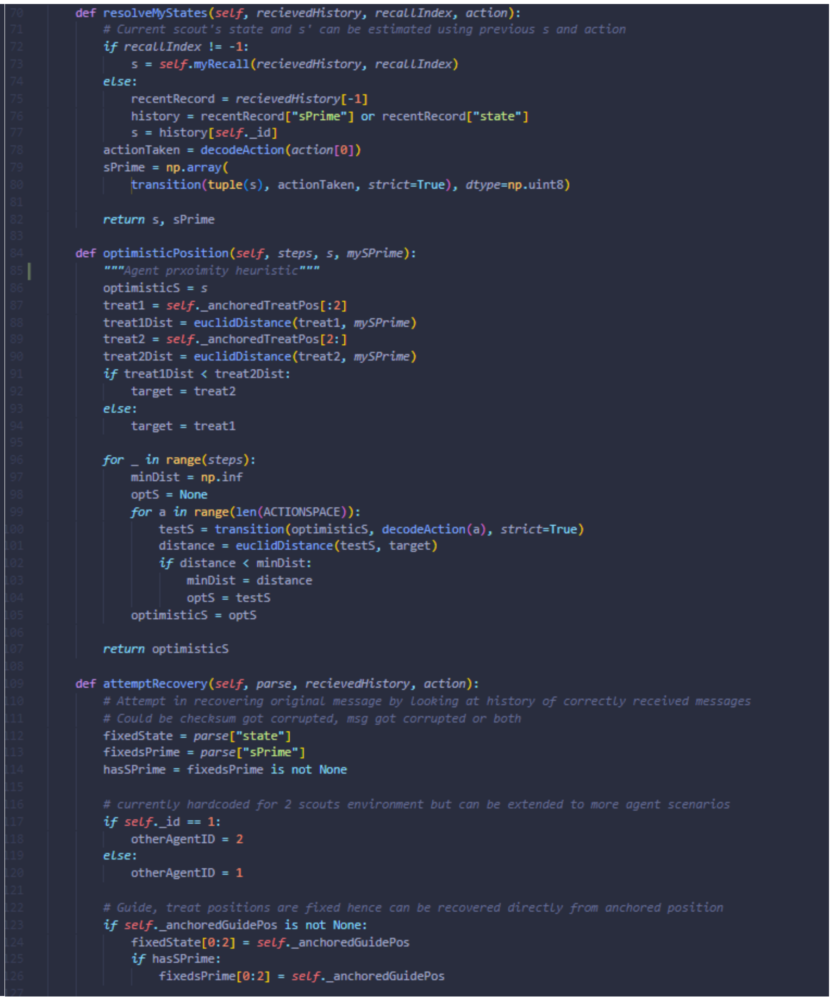


```python
class GuideAgent(CommAgent):

    def __init__(self, id, obs_dim, actionSpace, noiseHandling, hyperParam) -> None:
        super().__init__(id, obs_dim, actionSpace, noiseHandling, hyperParam)
        self._symbol = "G"

    def choose_action(self) -> torch.Tensor:
        """ Returns STAY as Guide can only stay at allocated position"""
        return torch.tensor([[STAY]], device=device)

    def choose_random_action(self) -> torch.Tensor:
        """ Returns STAY as Guide can only stay at allocated position"""
        return self.choose_action()

    def recieveBroadcast(self, signal):
        self._majorityNum = min(self._majorityNum + 2, self._bandwidth)

    def sendMessage(self, recieverID: int):
        msgString = self.encodeMessage()
        if self._noiseHandling:
            errorDetectionCode = self.errorDetector.encode(msgString)
            # error detection code sent along with msg
            msgString = np.concatenate([errorDetectionCode, msgString])
            for _ in range(self._majorityNum):
                self._channel.sendMessage(self._id, recieverID, msgString)
        else:
            self._channel.sendMessage(self._id, recieverID, msgString)

```

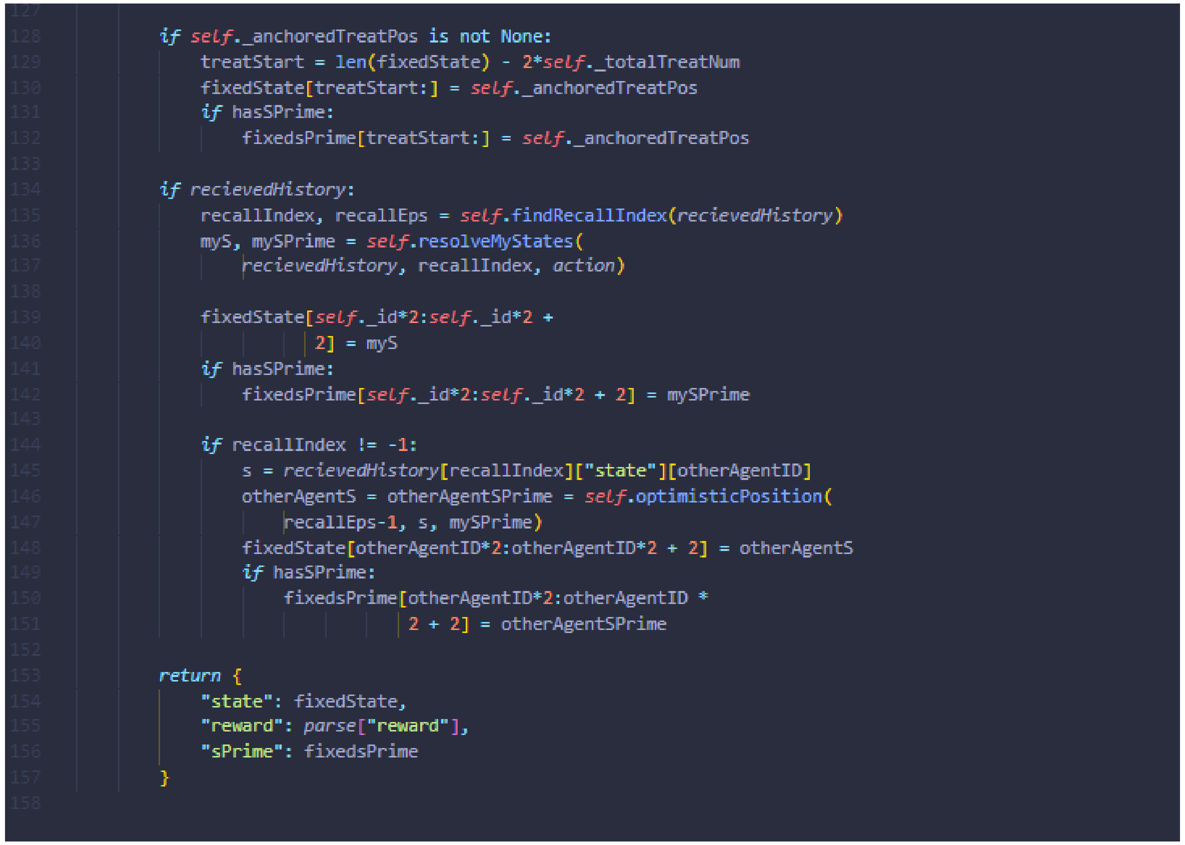
10.6.5 MessageRecoverer.py



```python
class MessageRecoverer():
    def __init__(self, id, totalTreatNum) -> None:
        self._guidePosRecord = defaultdict(defaultVal)
        self._treatPosRecord = defaultdict(defaultVal)
        self._anchoredGuidePos = None
        self._anchoredTreatPos = None
        self._id = id
        self._totalTreatNum = totalTreatNum
        self._checksumWeight = 3

    def computeAnchor(self, type, correctChecksum, position):
        increment = 1
        if correctChecksum:
            increment *= self._checksumWeight
        if type == "G":
            self._guidePosRecord[tuple(position)] += increment
            # print("Guide Pos: ", self._guidePosRecord)
            self._anchoredGuidePos = max(
                self._guidePosRecord, key=self._guidePosRecord.get)
        else:
            self._treatPosRecord[tuple(position)] += increment
            # print("Treat Pos: ", self._treatPosRecord)
            self._anchoredTreatPos = max(
                self._treatPosRecord, key=self._treatPosRecord.get)

    def computeGuideAnchor(self, position: np.array, correctChecksum: bool):
        # sets guide anchor position
        self.computeAnchor("G", correctChecksum, position)

    def computeTreatAnchor(self, position: np.array, correctChecksum: bool):
        # sets treat anchor position
        self.computeAnchor("T", correctChecksum, position)

    def findRecallIndex(self, recievedHistory):
        i = len(recievedHistory)
        eps = 0
        while i > 0:
            i -= 1
            if recievedHistory[i]["sPrime"] is not None:
                eps += 1
            if recievedHistory[i]["checksum"]:
                return i, eps
        return -1, 0

    def myRecall(self, recievedHistory, recallIndex):
        s = recievedHistory[recallIndex]["state"][self._id]
        sPrimes = recievedHistory[recallIndex]["sPrime"]
        if sPrimes:
            sPrime = sPrimes[self._id]
        else:
            sPrime = s
        for i in range(recallIndex+1, len(recievedHistory)):
            if recievedHistory[i]["sPrime"] is not None:
                s = sPrime
                actionTaken = decodeAction(recievedHistory[i]["action"][0])
                sPrime = np.array(
                    transition(tuple(s), actionTaken, strict=True), dtype=np.uint8)

        return sPrime
```

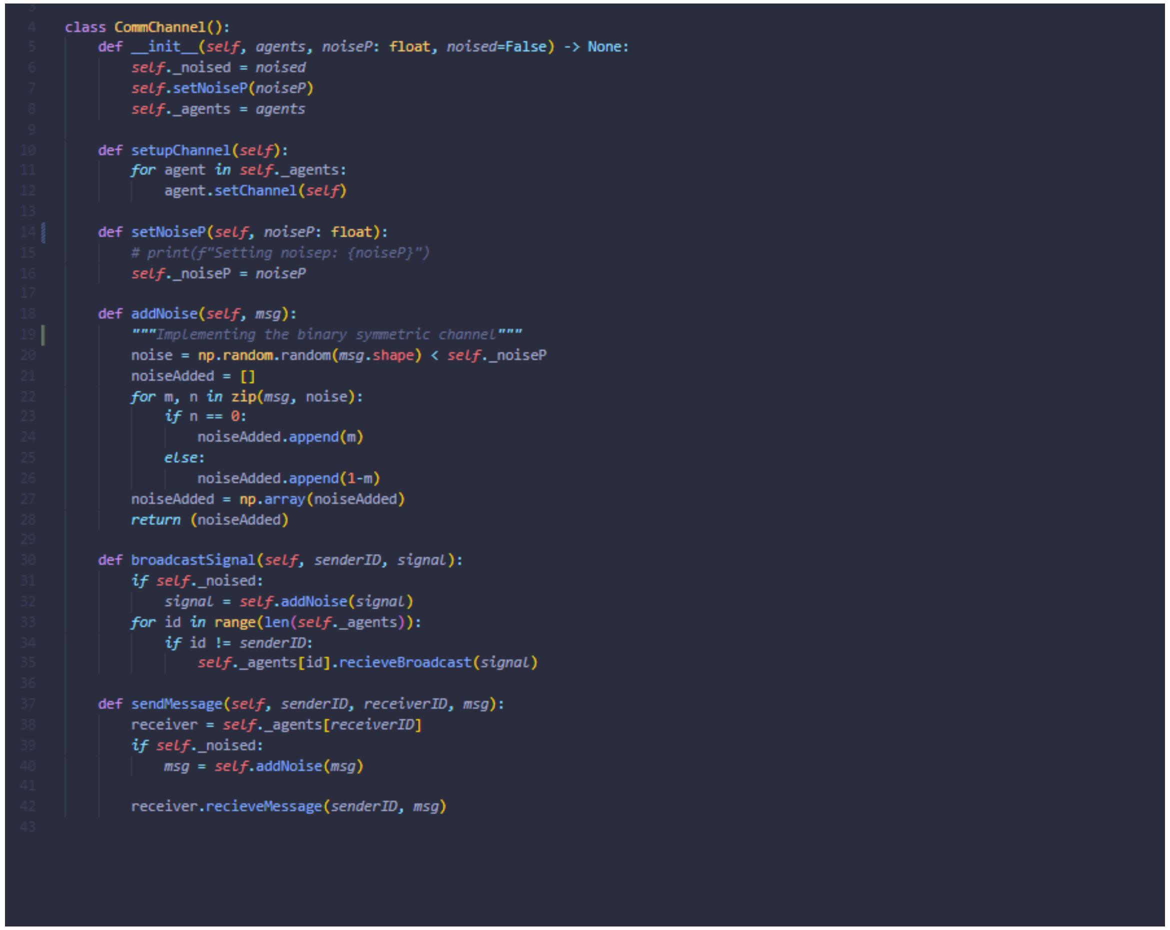


```python
    def resolveMyStates(self, recievedHistory, recallIndex, action):
        # Current scout's state and s' can be estimated using previous s and action
        if recallIndex != -1:
            s = self.myRecall(recievedHistory, recallIndex)
        else:
            recentRecord = recievedHistory[-1]
            history = recentRecord["sPrime"] or recentRecord["state"]
            s = history[self._id]
        actionTaken = decodeAction(action[0])
        sPrime = np.array(
            transition(tuple(s), actionTaken, strict=True), dtype=np.uint8)

        return s, sPrime

    def optimisticPosition(self, steps, s, mySPrime):
        """Agent prxoimity heuristic"""
        optimisticS = s
        treat1 = self._anchoredTreatPos[:2]
        treat1Dist = euclidDistance(treat1, mySPrime)
        treat2 = self._anchoredTreatPos[2:]
        treat2Dist = euclidDistance(treat2, mySPrime)
        if treat1Dist < treat2Dist:
            target = treat2
        else:
            target = treat1

        for _ in range(steps):
            minDist = np.inf
            optS = None
            for a in range(len(ACTIONSPACE)):
                testS = transition(optimisticS, decodeAction(a), strict=True)
                distance = euclidDistance(testS, target)
                if distance < minDist:
                    minDist = distance
                    optS = testS
            optimisticS = optS

        return optimisticS

    def attemptRecovery(self, parse, recievedHistory, action):
        # Attempt in recovering original message by looking at history of correctly received messages
        # Could be checksum got corrupted, msg got corrupted or both
        fixedState = parse["state"]
        fixedsPrime = parse["sPrime"]
        hasSPrime = fixedsPrime is not None

        # currently hardcoded for 2 scouts environment but can be extended to more agent scenarios
        if self._id == 1:
            otherAgentID = 2
        else:
            otherAgentID = 1

        # Guide, treat positions are fixed hence can be recovered directly from anchored position
        if self._anchoredGuidePos is not None:
            fixedState[0:2] = self._anchoredGuidePos
            if hasSPrime:
                fixedsPrime[0:2] = self._anchoredGuidePos
```



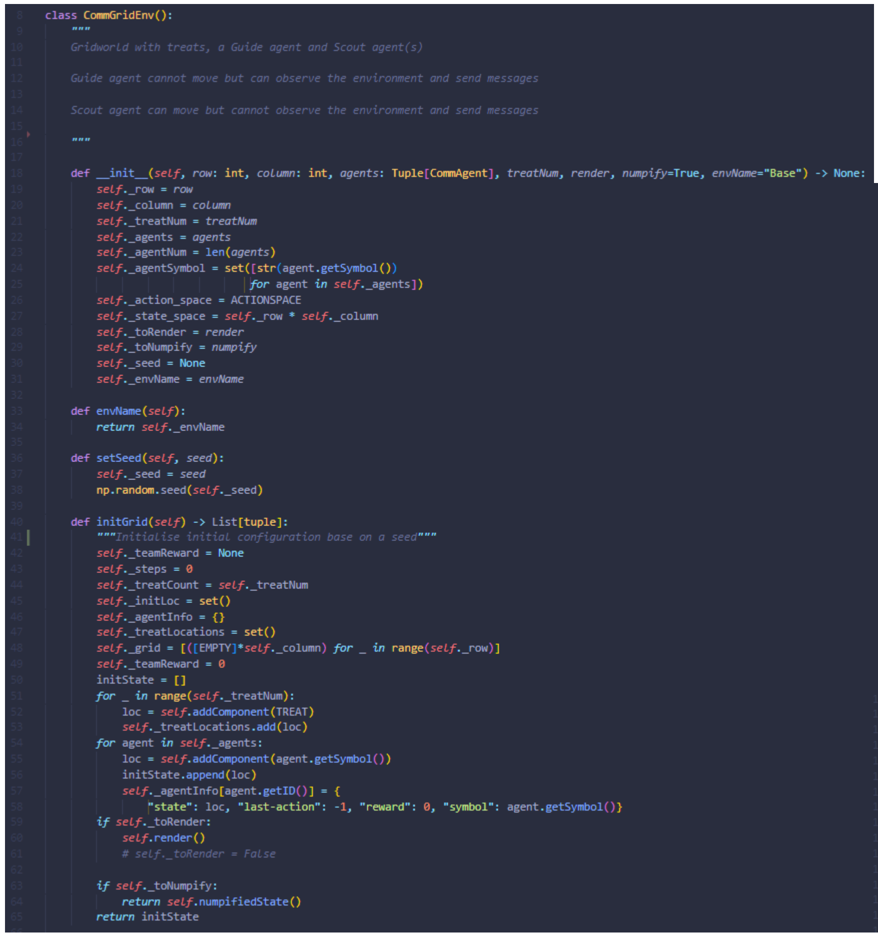
```
        if self._anchoredTreatPos is not None:
            treatStart = len(fixedState) - 2*self._totalTreatNum
            fixedState[treatStart:] = self._anchoredTreatPos
            if hasSPrime:
                fixedsPrime[treatStart:] = self._anchoredTreatPos

        if recievedHistory:
            recallIndex, recallEps = self.findRecallIndex(recievedHistory)
            myS, mySPrime = self.resolveMyStates(
                recievedHistory, recallIndex, action)

            fixedState[self._id*2:self._id*2 +
                       2] = myS
            if hasSPrime:
                fixedsPrime[self._id*2:self._id*2 + 2] = mySPrime

            if recallIndex != -1:
                s = recievedHistory[recallIndex]["state"][otherAgentID]
                otherAgentS = otherAgentSPrime = self.optimisticPosition(
                    recallEps-1, s, mySPrime)
                fixedState[otherAgentID*2:otherAgentID*2 + 2] = otherAgentS
                if hasSPrime:
                    fixedsPrime[otherAgentID*2:otherAgentID *
                                2 + 2] = otherAgentSPrime

        return {
            "state": fixedState,
            "reward": parse["reward"],
            "sPrime": fixedsPrime
        }
```

10.6.6 CommChannel.py

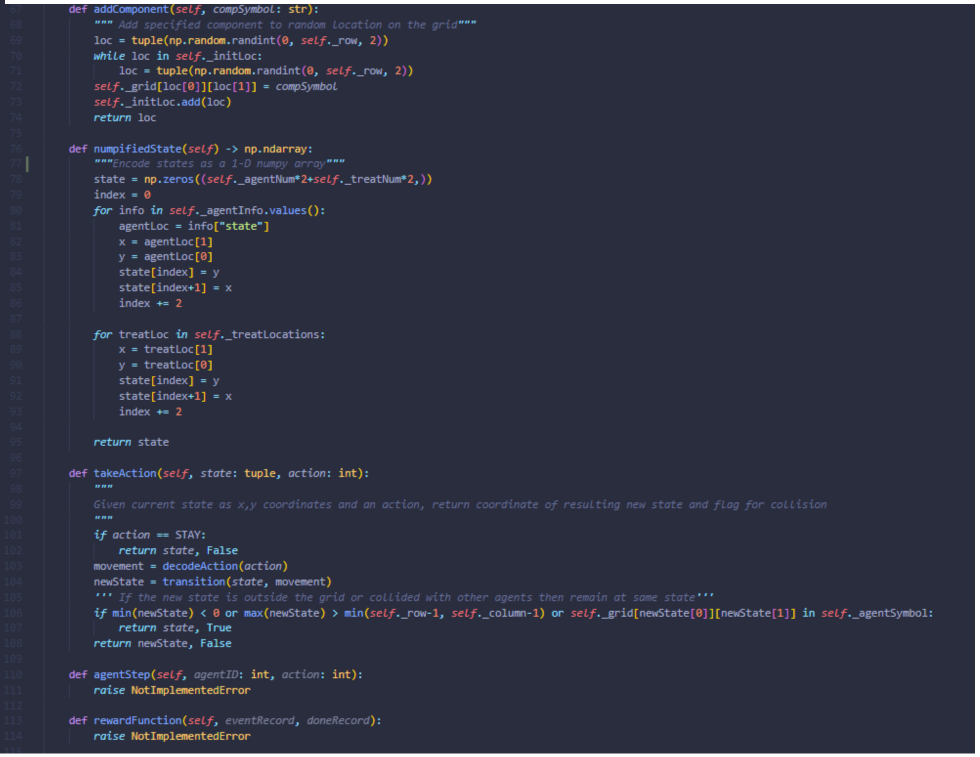


```python
class CommChannel():
    def __init__(self, agents, noiseP: float, noised=False) -> None:
        self._noised = noised
        self.setNoiseP(noiseP)
        self._agents = agents

    def setupChannel(self):
        for agent in self._agents:
            agent.setChannel(self)

    def setNoiseP(self, noiseP: float):
        # print(f"Setting noisep: {noiseP}")
        self._noiseP = noiseP

    def addNoise(self, msg):
        """Implementing the binary symmetric channel"""
        noise = np.random.random(msg.shape) < self._noiseP
        noiseAdded = []
        for m, n in zip(msg, noise):
            if n == 0:
                noiseAdded.append(m)
            else:
                noiseAdded.append(1-m)
        noiseAdded = np.array(noiseAdded)
        return (noiseAdded)

    def broadcastSignal(self, senderID, signal):
        if self._noised:
            signal = self.addNoise(signal)
        for id in range(len(self._agents)):
            if id != senderID:
                self._agents[id].recieveBroadcast(signal)

    def sendMessage(self, senderID, receiverID, msg):
        receiver = self._agents[receiverID]
        if self._noised:
            msg = self.addNoise(msg)

        receiver.recieveMessage(senderID, msg)
```

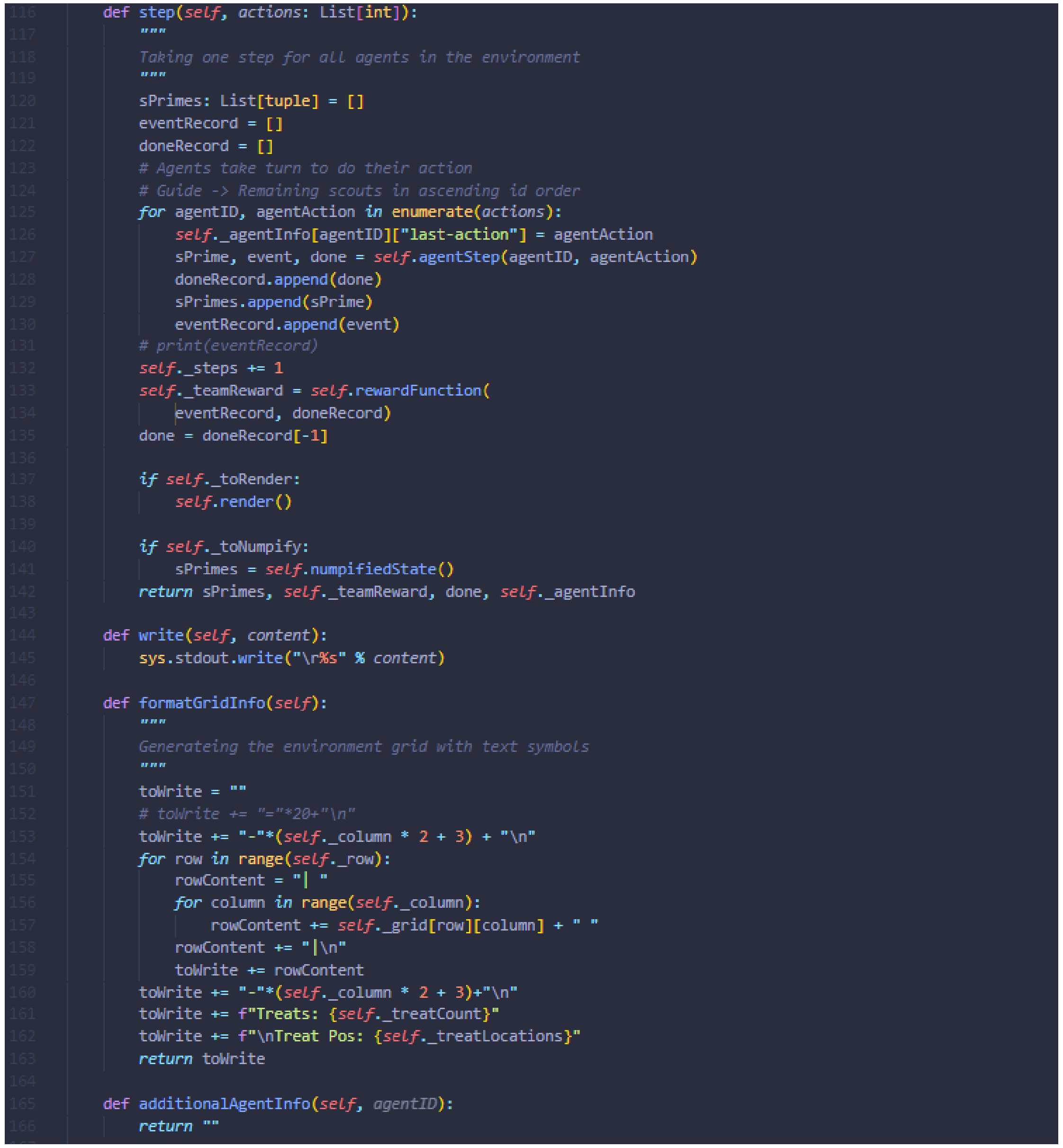
10.6.7 CommGridEnv.py



```python
class CommGridEnv():
    """
    Gridworld with treats, a Guide agent and Scout agent(s)

    Guide agent cannot move but can observe the environment and send messages

    Scout agent can move but cannot observe the environment and send messages

    """

    def __init__(self, row: int, column: int, agents: Tuple[CommAgent], treatNum, render, numpify=True, envName="Base") -> None:
        self._row = row
        self._column = column
        self._treatNum = treatNum
        self._agents = agents
        self._agentNum = len(agents)
        self._agentSymbol = set([str(agent.getSymbol())
                                 for agent in self._agents])
        self._action_space = ACTIONSPACE
        self._state_space = self._row * self._column
        self._toRender = render
        self._toNumpify = numpify
        self._seed = None
        self._envName = envName

    def envName(self):
        return self._envName

    def setSeed(self, seed):
        self._seed = seed
        np.random.seed(self._seed)

    def initGrid(self) -> List[tuple]:
        """Initialise initial configuration base on a seed"""
        self._teamReward = None
        self._steps = 0
        self._treatCount = self._treatNum
        self._initLoc = set()
        self._agentInfo = {}
        self._treatLocations = set()
        self._grid = [([EMPTY]*self._column) for _ in range(self._row)]
        self._teamReward = 0
        initState = []
        for _ in range(self._treatNum):
            loc = self.addComponent(TREAT)
            self._treatLocations.add(loc)
        for agent in self._agents:
            loc = self.addComponent(agent.getSymbol())
            initState.append(loc)
            self._agentInfo[agent.getID()] = {
                "state": loc, "last-action": -1, "reward": 0, "symbol": agent.getSymbol()}
        if self._toRender:
            self.render()
            # self._toRender = False

        if self._toNumpify:
            return self.numpifiedState()
        return initState
```

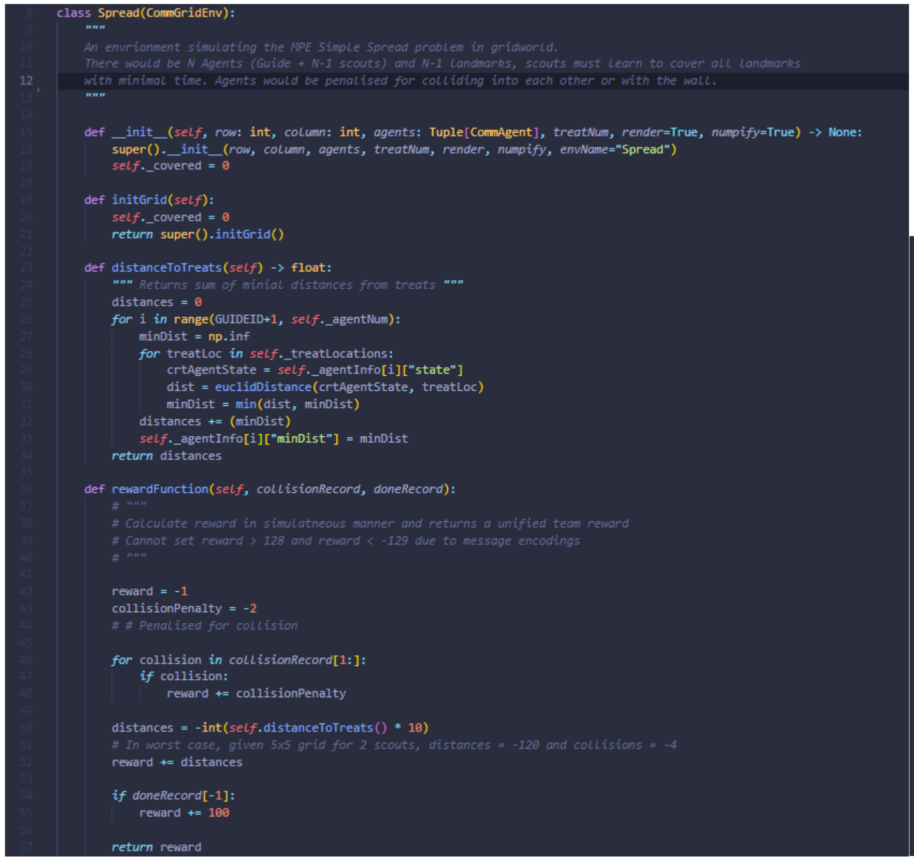


```python
    def addComponent(self, compSymbol: str):
        """ Add specified component to random location on the grid"""
        loc = tuple(np.random.randint(0, self._row, 2))
        while loc in self._initLoc:
            loc = tuple(np.random.randint(0, self._row, 2))
        self._grid[loc[0]][loc[1]] = compSymbol
        self._initLoc.add(loc)
        return loc

    def numpifiedState(self) -> np.ndarray:
        """Encode states as a 1-D numpy array"""
        state = np.zeros((self._agentNum*2+self._treatNum*2,))
        index = 0
        for info in self._agentInfo.values():
            agentLoc = info["state"]
            x = agentLoc[1]
            y = agentLoc[0]
            state[index] = y
            state[index+1] = x
            index += 2

        for treatLoc in self._treatLocations:
            x = treatLoc[1]
            y = treatLoc[0]
            state[index] = y
            state[index+1] = x
            index += 2

        return state

    def takeAction(self, state: tuple, action: int):
        """
        Given current state as x,y coordinates and an action, return coordinate of resulting new state and flag for collision
        """
        if action == STAY:
            return state, False
        movement = decodeAction(action)
        newState = transition(state, movement)
        ''' If the new state is outside the grid or collided with other agents then remain at same state'''
        if min(newState) < 0 or max(newState) > min(self._row-1, self._column-1) or self._grid[newState[0]][newState[1]] in self._agentSymbol:
            return state, True
        return newState, False

    def agentStep(self, agentID: int, action: int):
        raise NotImplementedError

    def rewardFunction(self, eventRecord, doneRecord):
        raise NotImplementedError
```

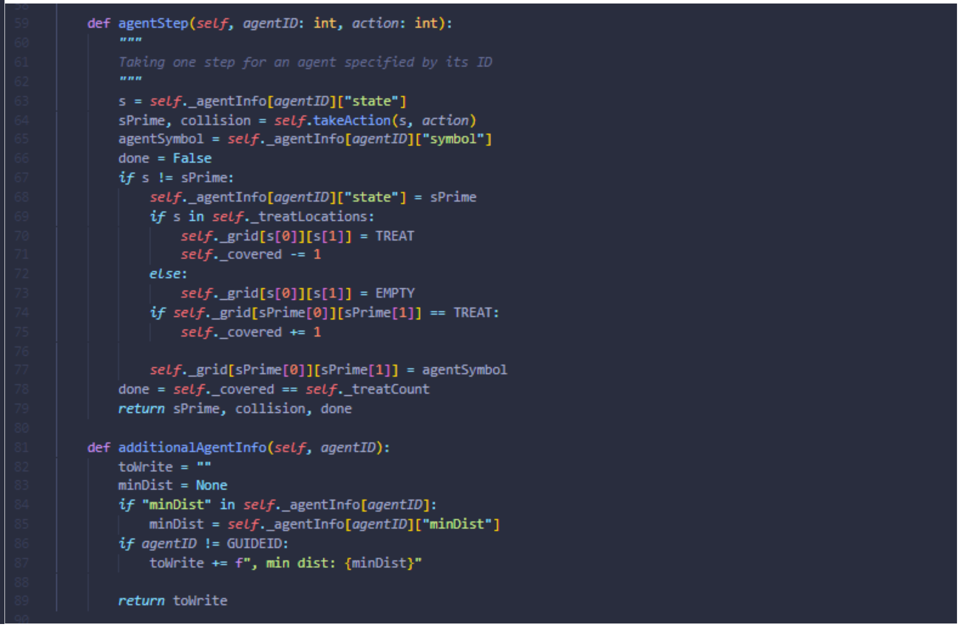


```python
    def step(self, actions: List[int]):
        """
        Taking one step for all agents in the environment
        """
        sPrimes: List[tuple] = []
        eventRecord = []
        doneRecord = []
        # Agents take turn to do their action
        # Guide -> Remaining scouts in ascending id order
        for agentID, agentAction in enumerate(actions):
            self._agentInfo[agentID]["last-action"] = agentAction
            sPrime, event, done = self.agentStep(agentID, agentAction)
            doneRecord.append(done)
            sPrimes.append(sPrime)
            eventRecord.append(event)
        # print(eventRecord)
        self._steps += 1
        self._teamReward = self.rewardFunction(
            eventRecord, doneRecord)
        done = doneRecord[-1]

        if self._toRender:
            self.render()

        if self._toNumpify:
            sPrimes = self.numpifiedState()
        return sPrimes, self._teamReward, done, self._agentInfo

    def write(self, content):
        sys.stdout.write("\r%s" % content)

    def formatGridInfo(self):
        """
        Generateing the environment grid with text symbols
        """
        toWrite = ""
        # toWrite += "="*20+"\n"
        toWrite += "-"*(self._column * 2 + 3) + "\n"
        for row in range(self._row):
            rowContent = "| "
            for column in range(self._column):
                rowContent += self._grid[row][column] + " "
            rowContent += "|\n"
            toWrite += rowContent
        toWrite += "-"*(self._column * 2 + 3)+"\n"
        toWrite += f"Treats: {self._treatCount}"
        toWrite += f"\nTreat Pos: {self._treatLocations}"
        return toWrite

    def additionalAgentInfo(self, agentID):
        return ""
```

10.6.8 Spread.py

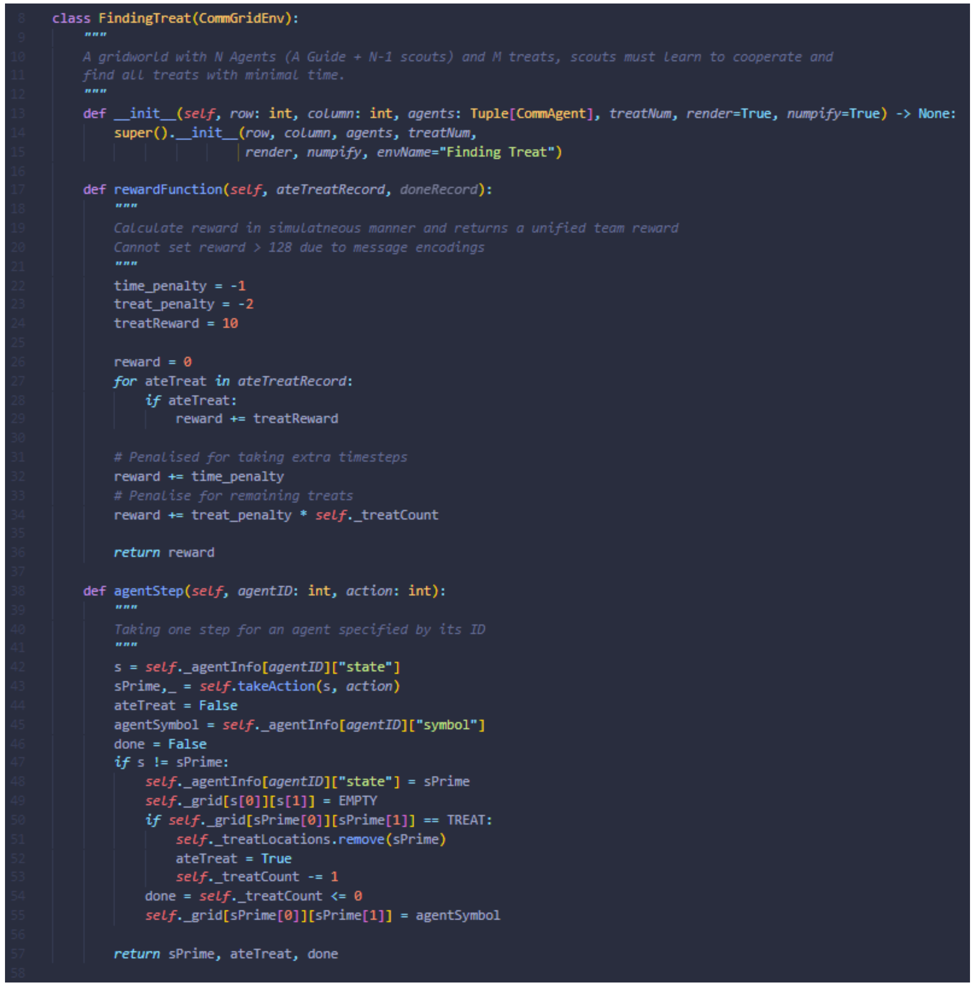


```python
class Spread(CommGridEnv):
    """
    An envrionment simulating the MPE Simple Spread problem in gridworld.
    There would be N Agents (Guide + N-1 scouts) and N-1 landmarks, scouts must learn to cover all landmarks
    with minimal time. Agents would be penalised for colliding into each other or with the wall.
    """

    def __init__(self, row: int, column: int, agents: Tuple[CommAgent], treatNum, render=True, numpify=True) -> None:
        super().__init__(row, column, agents, treatNum, render, numpify, envName="Spread")
        self._covered = 0

    def initGrid(self):
        self._covered = 0
        return super().initGrid()

    def distanceToTreats(self) -> float:
        """ Returns sum of minial distances from treats """
        distances = 0
        for i in range(GUIDEID+1, self._agentNum):
            minDist = np.inf
            for treatLoc in self._treatLocations:
                crtAgentState = self._agentInfo[i]["state"]
                dist = euclidDistance(crtAgentState, treatLoc)
                minDist = min(dist, minDist)
            distances += (minDist)
            self._agentInfo[i]["minDist"] = minDist
        return distances

    def rewardFunction(self, collisionRecord, doneRecord):
        # """
        # Calculate reward in simulatneous manner and returns a unified team reward
        # Cannot set reward > 128 and reward < -129 due to message encodings
        # """

        reward = -1
        collisionPenalty = -2
        # # Penalised for collision

        for collision in collisionRecord[1:]:
            if collision:
                reward += collisionPenalty

        distances = -int(self.distanceToTreats() * 10)
        # In worst case, given 5x5 grid for 2 scouts, distances = -120 and collisions = -4
        reward += distances

        if doneRecord[-1]:
            reward += 100

        return reward
```



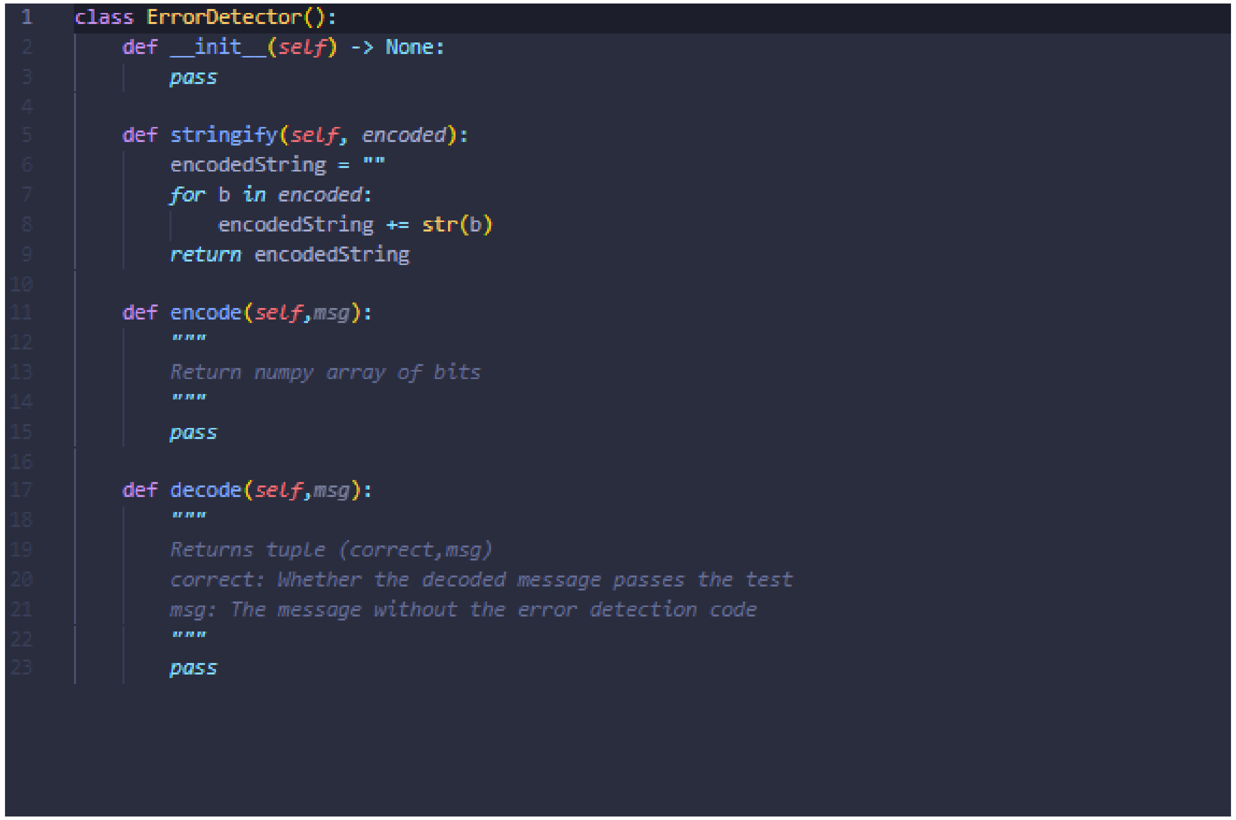
```python
    def agentStep(self, agentID: int, action: int):
        """
        Taking one step for an agent specified by its ID
        """
        s = self._agentInfo[agentID]["state"]
        sPrime, collision = self.takeAction(s, action)
        agentSymbol = self._agentInfo[agentID]["symbol"]
        done = False
        if s != sPrime:
            self._agentInfo[agentID]["state"] = sPrime
            if s in self._treatLocations:
                self._grid[s[0]][s[1]] = TREAT
                self._covered -= 1
            else:
                self._grid[s[0]][s[1]] = EMPTY
            if self._grid[sPrime[0]][sPrime[1]] == TREAT:
                self._covered += 1

            self._grid[sPrime[0]][sPrime[1]] = agentSymbol
        done = self._covered == self._treatCount
        return sPrime, collision, done

    def additionalAgentInfo(self, agentID):
        toWrite = ""
        minDist = None
        if "minDist" in self._agentInfo[agentID]:
            minDist = self._agentInfo[agentID]["minDist"]
        if agentID != GUIDEID:
            toWrite += f", min dist: {minDist}"

        return toWrite
```

10.6.9 FindingTreat.py



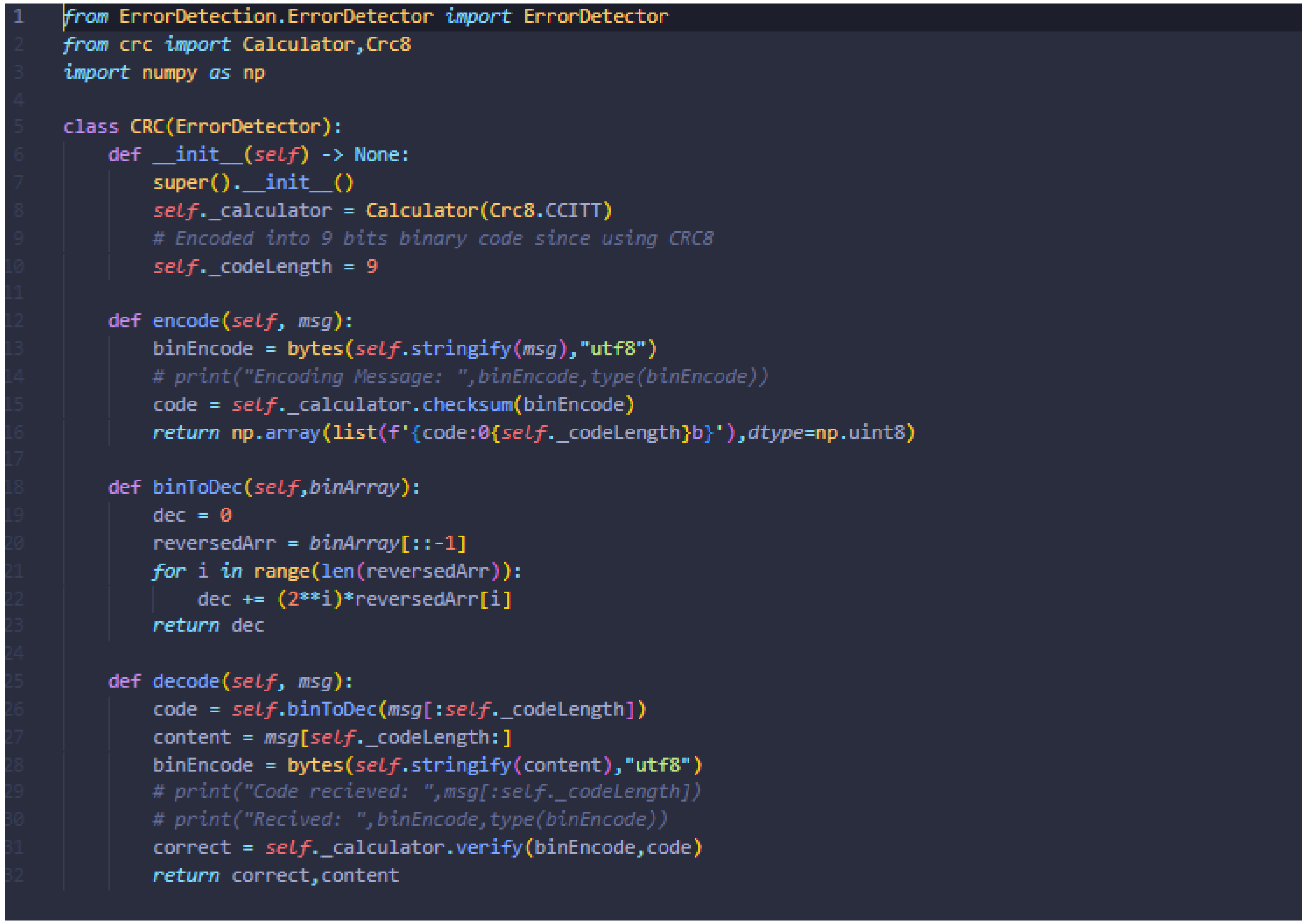
```python
class FindingTreat(CommGridEnv):
    """
    A gridworld with N Agents (A Guide + N-1 scouts) and M treats, scouts must learn to cooperate and
    find all treats with minimal time.
    """
    def __init__(self, row: int, column: int, agents: Tuple[CommAgent], treatNum, render=True, numpify=True) -> None:
        super().__init__(row, column, agents, treatNum,
                         render, numpify, envName="Finding Treat")

    def rewardFunction(self, ateTreatRecord, doneRecord):
        """
        Calculate reward in simulatneous manner and returns a unified team reward
        Cannot set reward > 128 due to message encodings
        """
        time_penalty = -1
        treat_penalty = -2
        treatReward = 10

        reward = 0
        for ateTreat in ateTreatRecord:
            if ateTreat:
                reward += treatReward

        # Penalised for taking extra timesteps
        reward += time_penalty
        # Penalise for remaining treats
        reward += treat_penalty * self._treatCount

        return reward

    def agentStep(self, agentID: int, action: int):
        """
        Taking one step for an agent specified by its ID
        """
        s = self._agentInfo[agentID]["state"]
        sPrime,_ = self.takeAction(s, action)
        ateTreat = False
        agentSymbol = self._agentInfo[agentID]["symbol"]
        done = False
        if s != sPrime:
            self._agentInfo[agentID]["state"] = sPrime
            self._grid[s[0]][s[1]] = EMPTY
            if self._grid[sPrime[0]][sPrime[1]] == TREAT:
                self._treatLocations.remove(sPrime)
                ateTreat = True
                self._treatCount -= 1
            done = self._treatCount <= 0
            self._grid[sPrime[0]][sPrime[1]] = agentSymbol

        return sPrime, ateTreat, done
```

10.6.10 ErrorDetector.py



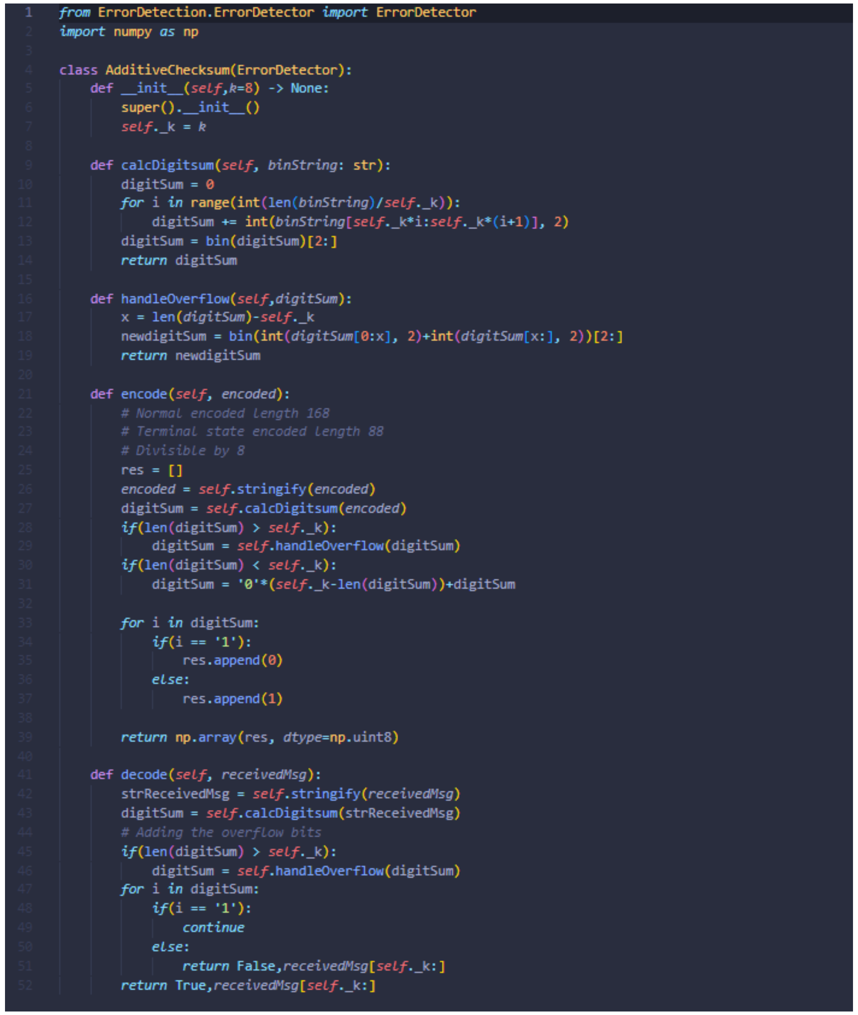
```python
class ErrorDetector():
    def __init__(self) -> None:
        pass

    def stringify(self, encoded):
        encodedString = ""
        for b in encoded:
            encodedString += str(b)
        return encodedString

    def encode(self,msg):
        """
        Return numpy array of bits
        """
        pass

    def decode(self,msg):
        """
        Returns tuple (correct,msg)
        correct: Whether the decoded message passes the test
        msg: The message without the error detection code
        """
        pass
```

10.6.11 CRC.py



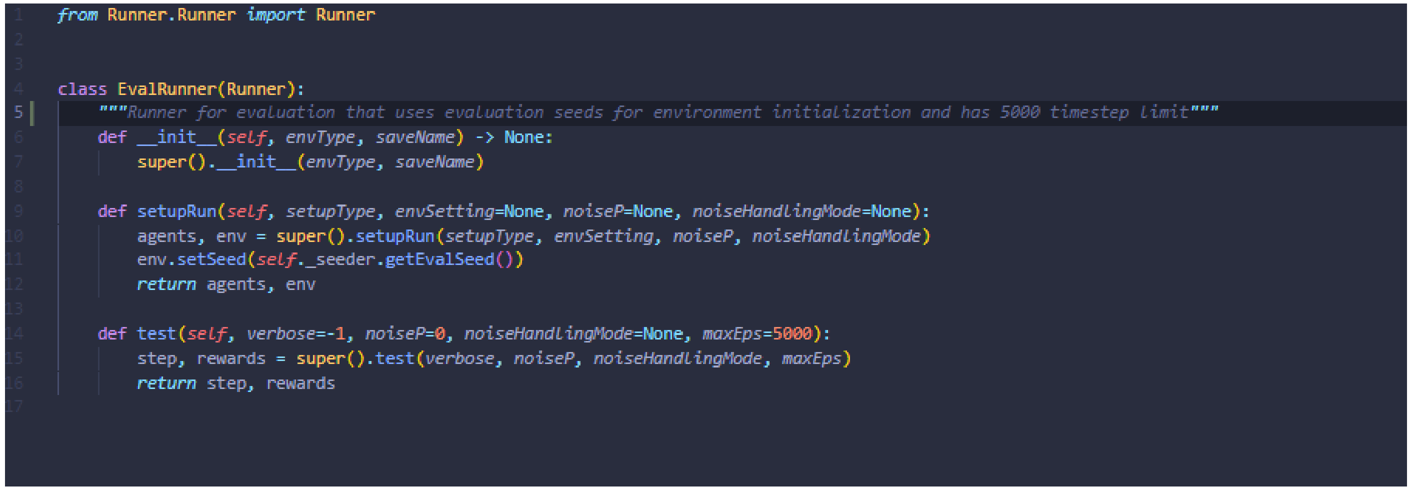
```python
from ErrorDetection.ErrorDetector import ErrorDetector
from crc import Calculator,Crc8
import numpy as np

class CRC(ErrorDetector):
    def __init__(self) -> None:
        super().__init__()
        self._calculator = Calculator(Crc8.CCITT)
        # Encoded into 9 bits binary code since using CRC8
        self._codeLength = 9

    def encode(self, msg):
        binEncode = bytes(self.stringify(msg),"utf8")
        # print("Encoding Message: ",binEncode,type(binEncode))
        code = self._calculator.checksum(binEncode)
        return np.array(list(f'{code:0{self._codeLength}b}'),dtype=np.uint8)

    def binToDec(self,binArray):
        dec = 0
        reversedArr = binArray[::-1]
        for i in range(len(reversedArr)):
            dec += (2**i)*reversedArr[i]
        return dec

    def decode(self, msg):
        code = self.binToDec(msg[:self._codeLength])
        content = msg[self._codeLength:]
        binEncode = bytes(self.stringify(content),"utf8")
        # print("Code recieved: ",msg[:self._codeLength])
        # print("Recived: ",binEncode,type(binEncode))
        correct = self._calculator.verify(binEncode,code)
        return correct,content
```

10.6.12 AdditiveChecksum.py



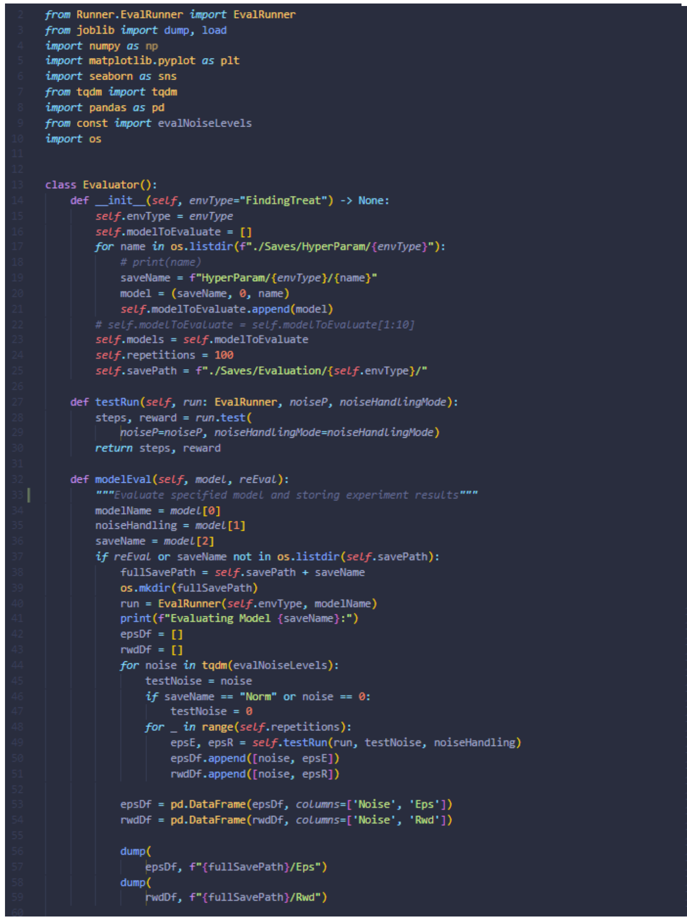
```python
from ErrorDetection.ErrorDetector import ErrorDetector
import numpy as np

class AdditiveChecksum(ErrorDetector):
    def __init__(self,k=8) -> None:
        super().__init__()
        self._k = k

    def calcDigitsum(self, binString: str):
        digitSum = 0
        for i in range(int(len(binString)/self._k)):
            digitSum += int(binString[self._k*i:self._k*(i+1)], 2)
        digitSum = bin(digitSum)[2:]
        return digitSum

    def handleOverflow(self,digitSum):
        x = len(digitSum)-self._k
        newdigitSum = bin(int(digitSum[0:x], 2)+int(digitSum[x:], 2))[2:]
        return newdigitSum

    def encode(self, encoded):
        # Normal encoded length 168
        # Terminal state encoded length 88
        # Divisible by 8
        res = []
        encoded = self.stringify(encoded)
        digitSum = self.calcDigitsum(encoded)
        if(len(digitSum) > self._k):
            digitSum = self.handleOverflow(digitSum)
        if(len(digitSum) < self._k):
            digitSum = '0'*(self._k-len(digitSum))+digitSum

        for i in digitSum:
            if(i == '1'):
                res.append(0)
            else:
                res.append(1)

        return np.array(res, dtype=np.uint8)

    def decode(self, receivedMsg):
        strReceivedMsg = self.stringify(receivedMsg)
        digitSum = self.calcDigitsum(strReceivedMsg)
        # Adding the overflow bits
        if(len(digitSum) > self._k):
            digitSum = self.handleOverflow(digitSum)
        for i in digitSum:
            if(i == '1'):
                continue
            else:
                return False,receivedMsg[self._k:]
        return True,receivedMsg[self._k:]
```

10.6.13 EvalRunner.py

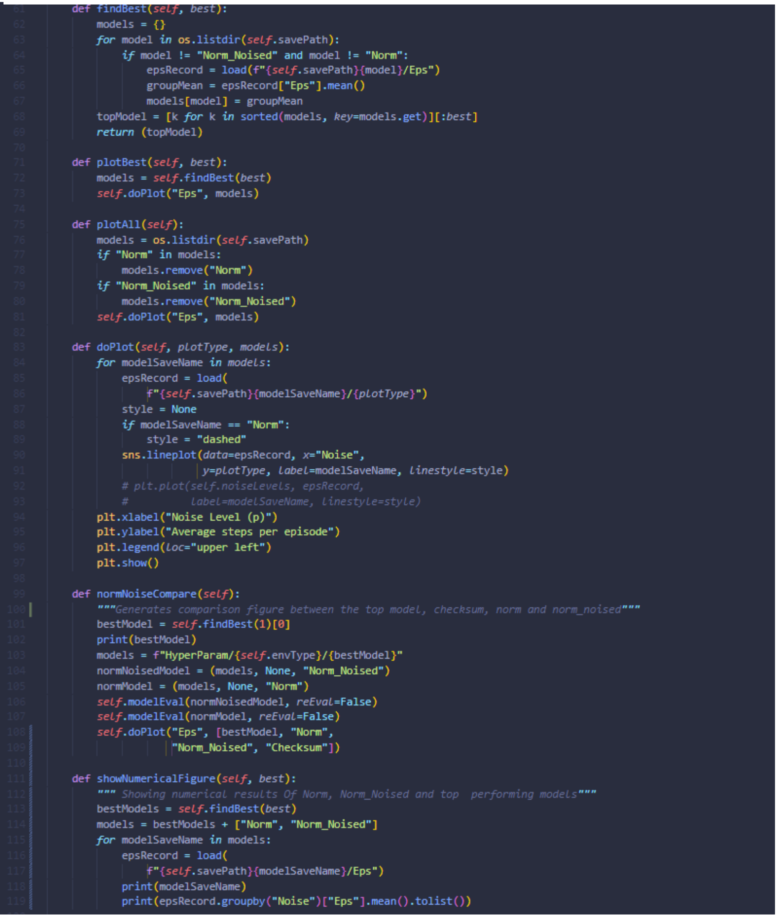


```python
from Runner.Runner import Runner


class EvalRunner(Runner):
    """Runner for evaluation that uses evaluation seeds for environment initialization and has 5000 timestep limit"""
    def __init__(self, envType, saveName) -> None:
        super().__init__(envType, saveName)

    def setupRun(self, setupType, envSetting=None, noiseP=None, noiseHandlingMode=None):
        agents, env = super().setupRun(setupType, envSetting, noiseP, noiseHandlingMode)
        env.setSeed(self._seeder.getEvalSeed())
        return agents, env

    def test(self, verbose=-1, noiseP=0, noiseHandlingMode=None, maxEps=5000):
        step, rewards = super().test(verbose, noiseP, noiseHandlingMode, maxEps)
        return step, rewards
```

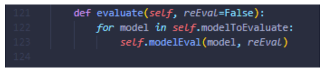
10.6.14 Evaluator.py



```python
from Runner.EvalRunner import EvalRunner
from joblib import dump, load
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
import pandas as pd
from const import evalNoiseLevels
import os


class Evaluator():
    def __init__(self, envType="FindingTreat") -> None:
        self.envType = envType
        self.modelToEvaluate = []
        for name in os.listdir(f"./Saves/HyperParam/{envType}"):
            # print(name)
            saveName = f"HyperParam/{envType}/{name}"
            model = (saveName, 0, name)
            self.modelToEvaluate.append(model)
        # self.modelToEvaluate = self.modelToEvaluate[1:10]
        self.models = self.modelToEvaluate
        self.repetitions = 100
        self.savePath = f"./Saves/Evaluation/{self.envType}/"

    def testRun(self, run: EvalRunner, noiseP, noiseHandlingMode):
        steps, reward = run.test(
            noiseP=noiseP, noiseHandlingMode=noiseHandlingMode)
        return steps, reward

    def modelEval(self, model, reEval):
        """Evaluate specified model and storing experiment results"""
        modelName = model[0]
        noiseHandling = model[1]
        saveName = model[2]
        if reEval or saveName not in os.listdir(self.savePath):
            fullSavePath = self.savePath + saveName
            os.mkdir(fullSavePath)
            run = EvalRunner(self.envType, modelName)
            print(f"Evaluating Model {saveName}:")
            epsDf = []
            rwdDf = []
            for noise in tqdm(evalNoiseLevels):
                testNoise = noise
                if saveName == "Norm" or noise == 0:
                    testNoise = 0
                for _ in range(self.repetitions):
                    epsE, epsR = self.testRun(run, testNoise, noiseHandling)
                    epsDf.append([noise, epsE])
                    rwdDf.append([noise, epsR])

            epsDf = pd.DataFrame(epsDf, columns=['Noise', 'Eps'])
            rwdDf = pd.DataFrame(rwdDf, columns=['Noise', 'Rwd'])

            dump(
                epsDf, f"{fullSavePath}/Eps")
            dump(
                rwdDf, f"{fullSavePath}/Rwd")
```

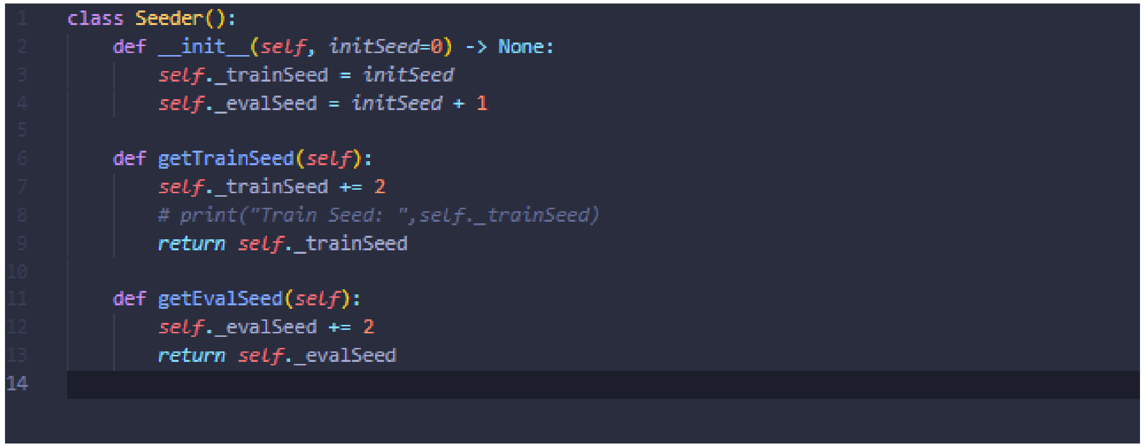


```python
    def findBest(self, best):
        models = {}
        for model in os.listdir(self.savePath):
            if model != "Norm_Noised" and model != "Norm":
                epsRecord = load(f"{self.savePath}{model}/Eps")
                groupMean = epsRecord["Eps"].mean()
                models[model] = groupMean
        topModel = [k for k in sorted(models, key=models.get)][:best]
        return (topModel)

    def plotBest(self, best):
        models = self.findBest(best)
        self.doPlot("Eps", models)

    def plotAll(self):
        models = os.listdir(self.savePath)
        if "Norm" in models:
            models.remove("Norm")
        if "Norm_Noised" in models:
            models.remove("Norm_Noised")
        self.doPlot("Eps", models)

    def doPlot(self, plotType, models):
        for modelSaveName in models:
            epsRecord = load(
                f"{self.savePath}{modelSaveName}/{plotType}")
            style = None
            if modelSaveName == "Norm":
                style = "dashed"
            sns.lineplot(data=epsRecord, x="Noise",
                         y=plotType, label=modelSaveName, linestyle=style)
            # plt.plot(self.noiseLevels, epsRecord,
            #          label=modelSaveName, linestyle=style)
        plt.xlabel("Noise Level (p)")
        plt.ylabel("Average steps per episode")
        plt.legend(loc="upper left")
        plt.show()

    def normNoiseCompare(self):
        """Generates comparison figure between the top model, checksum, norm and norm_noised"""
        bestModel = self.findBest(1)[0]
        print(bestModel)
        models = f"HyperParam/{self.envType}/{bestModel}"
        normNoisedModel = (models, None, "Norm_Noised")
        normModel = (models, None, "Norm")
        self.modelEval(normNoisedModel, reEval=False)
        self.modelEval(normModel, reEval=False)
        self.doPlot("Eps", [bestModel, "Norm",
                            "Norm_Noised", "Checksum"])

    def showNumericalFigure(self, best):
        """ Showing numerical results Of Norm, Norm_Noised and top  performing models"""
        bestModels = self.findBest(best)
        models = bestModels + ["Norm", "Norm_Noised"]
        for modelSaveName in models:
            epsRecord = load(
                f"{self.savePath}{modelSaveName}/Eps")
            print(modelSaveName)
            print(epsRecord.groupby("Noise")["Eps"].mean().tolist())
```



```python
    def evaluate(self, reEval=False):
        for model in self.modelToEvaluate:
            self.modelEval(model, reEval)
```

10.6.15 Seeder.py



```python
class Seeder():
    def __init__(self, initSeed=0) -> None:
        self._trainSeed = initSeed
        self._evalSeed = initSeed + 1

    def getTrainSeed(self):
        self._trainSeed += 2
        # print("Train Seed: ",self._trainSeed)
        return self._trainSeed

    def getEvalSeed(self):
        self._evalSeed += 2
        return self._evalSeed

```