Application of Q-learning based Multi-agent Reinforcement learning techniques to Level-based Foraging Environment

Third Year Individual Project – Final Report

April 2024

Mahir Pokar

10892478

Supervisor:

Dr. Joaquin Carrasco

Contents

Word count: 6761

Abstract

Multi-Agent Reinforcement Learning (MARL) has emerged as a powerful paradigm for developing intelligent systems capable of collaborative decision-making in complex environments. This project explores the paradigm of multi-agent reinforcement learning, particularly in cooperative environments. The sequential decision problem is modelled as a Markov decision process, and then an optimal policy is found using the temporal difference learning algorithm; Q-learning. Through this project, the challenges posed by MARL's non-stationarity, particularly the adaptation of learning agents to evolving policies of other agents is explored. While single-agent algorithms suffice for simpler scenarios, their scalability and optimality in multi-agent settings are limited. The Q-learning algorithm was first implemented by the traditional tabular methods. It is shown the inherent limitations of the tabular method prevent its application to complex scenarios with a large state space. Deep reinforcement learning techniques were employed to overcome this limitation, by using neural networks as function approximators to represent the action-value function. However, this comes with its own set of challenges such as the moving target problem and correlation between subsequent learning samples. These challenges are tackled by implementing a powerful technique known as a Deep Q-network (DQN) agent.

Declaration of originality

I hereby confirm that this dissertation is my own original work unless referenced clearly to the contrary, and that no portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Intellectual property statement

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420), in any relevant Dissertation restriction declarations deposited in the University Library, and The University Library's regulations (see http://www.library.manchester.ac.uk/about/regulations/_files/Library-regulations.pdf).

1 Introduction

1.1 Background and motivation

In recent decades, the acceleration of processing capabilities alongside cost reductions has propelled Artificial Intelligence (AI) research. Reinforcement learning (RL) has emerged as a powerful paradigm in the field of AI. Enabling systems and machines to learn optimal decision-making strategies through interactions with their environments. RL algorithms have been successfully applied to a wide range of problems, from robotics and game-playing to finance and healthcare [1].

Multi-agent reinforcement learning (MARL) is a subfield of machine learning that deals with agents learning to interact with each other and their environment to achieve competitive or collective goals. Unlike single-agent reinforcement learning, where an agent learns to make decisions based on its own actions and the resulting rewards, MARL involves multiple agents simultaneously learning and adapting their strategies through interaction. The motivation behind studying MARL stems from its applicability to a wide range of real-world scenarios. In various domains such as robotics, autonomous vehicles, game theory, and portfolio management, multiple agents often coexist and need to collaborate or compete to achieve optimal outcomes [2] [3] [4] [5]. Multi-agent systems have inherent features such as efficiency, redundancy and flexibility make them ideal for complex tasks. For example, even if one of the agents is incapacitated, intelligent agents can adapt and redistribute the workload [6]. MARL provides a framework to model and solve these complex interaction problems by allowing agents to learn from experience, explore different strategies, and coordinate their actions.

By studying MARL, researchers seek to unlock the potential for intelligent, adaptive systems that can collaborate, negotiate, and compete in complex environments. The insights gained from MARL research can pave the way for advancements in autonomous systems, human-robot interaction, decentralized decision-making, and beyond, contributing to the development of more efficient and capable AI systems in various domains.

1.2 Aims and Objectives

The primary aim of this project is to delve into the complexities and challenges inherent in Multi-Agent Reinforcement Learning (MARL), specifically in dynamic and evolving environments where agents must engage in cooperative decision-making. The project seeks to model sequential

decision problems using Markov decision processes and implement the temporal difference learning algorithm, Q-learning, to derive optimal policies. A key objective is to thoroughly analyse the non-stationarity challenges posed by MARL, particularly focusing on how learning agents adapt to the continually changing policies of other agents within the environment. Additionally, the project aims to evaluate the efficacy of single-agent algorithms in addressing the intricacies of multi-agent scenarios, emphasizing aspects such as scalability and optimality. Through the implementation of the Q-learning algorithm using traditional tabular methods, the project aims to uncover the inherent limitations of these approaches, especially in scenarios with extensive state spaces. Furthermore, the project intends to leverage deep reinforcement learning techniques, employing neural networks as function approximators, to overcome the shortcomings of tabular methods and enhance performance in complex environments. A critical objective is to address challenges such as the moving target problem and correlation issues in subsequent learning samples by implementing a Deep Q-network (DQN) agent. The project concludes with a comprehensive evaluation and comparative analysis of the DQN agent against traditional tabular methods and deep reinforcement learning techniques, documenting key findings and insights to contribute to the advancement of multi-agent reinforcement learning methodologies.

The specific objectives include:

- Implement Q-table-based reinforcement learning algorithms.
- Employ Deep Q-learning techniques.
- Observe the stability and features of the algorithms.

1.3 Report structure

This report is structured as follows. Section 2 provides reviews of several pieces of literature regarding multi-agent reinforcement learning. Section 3 elaborates on the proposed methodology and outlines the theoretical prerequisites. Section 4 shows the results and evaluates them with an analysis and discussion. Finally, section 5 concludes the report and states the planned future work.

2 Literature review

2.1 Introduction

Multi-agent systems (MAS) excel in handling diverse interactions and ensuring fair representation of interests, making them essential for complex scenarios. They enhance efficiency through parallel computation, offer robustness against failures, and are easily scalable and adaptable [7]. This project deals with autonomous agents learning how to perform a cooperative task, using Q-learning [8] based approaches which are in turn based on Dynamic programming [9] and temporal-difference learning [10]. Besides that, MARL is heavily shaped by Game theory, Evolutionary computation, and optimization theory [11]. Multi-agent deep reinforcement learning (MADRL) integrates the representation abilities of function approximators and the decision-making abilities of reinforcement learning.

2.2 Q-learning

The foundational concept of Markov decision process and Markov games laid the groundwork for RL and MARL advancements. MARL shares the foundational ideas from single-agent reinforcement learning, early research focused on decentralised policy learning, such as independent Q-learning [12]. In many practical multi-agent learning scenarios, e.g., multi-robot control [13] agents make decentralised decisions without a coordinator. Agents take actions independently based on local observations of their own reward and action history, usually in a myopic fashion. This decentralised method makes it much easier to implement and highly scalable making it ubiquitous in practice [14]. Unfortunately, decentralised MARL algorithms tend not to converge. The key challenge in multiagent learning is learning the best response to the behaviour of other agents, which may be non-stationary, this is also known as the moving target problem [11]. Q-learning employs the simplest method to tackle non-stationarity, “Ignore” [15]. Q-learning essentially ignores the policies of other agents; therefore, it cannot be used in all scenarios.

Traditionally Q-learning is reliant on tables to store the correlation between state-action values, but this requires a large amount of memory as states increase exponentially as the scenario becomes more complicated. [16] presents a novel agent, a deep Q-network (DQN) which is able to integrate artificial neural networks known as deep neural networks with reinforcement learning. DQN is also able to address the instability and divergence issues inherent in using neural networks for action-value function approximation in reinforcement learning. This approach incorporates

two key strategies: first, the utilization of experience replay to randomize data and eliminate observation sequence correlations, and second, an iterative update mechanism that adjusts action values towards periodically updated target values, reducing correlations and stabilizing learning dynamics.

Although a multi-agent system can be modelled as a distributed single-agent system to then apply conventional reinforcement learning techniques to learn an optimal joint policy, the state space grows exponentially with each new agent rendering this approach unfeasible for most scenarios. [17] recognises that in a multi-agent scenario, the agents must coordinate only in a handful of states with a subset of the agents and proposes that agents have two Q-value functions, one for independent actions and a joint policy for states requiring coordination. Decentralised execution also allows for experience sharing between agents, and even parameter sharing between homogeneous agents.

2.3 Policy gradient methods

Policy gradient methods learn a parameterised policy that can select actions without consulting a value function [18]. [19] [20] employs a centralised training and decentralised execution (CTDE) paradigm, unlike independent Q-learning which follows decentralised training and execution. Partial observability and communication constraints during application necessitates decentralised execution, however during training in a simulated environment additional data about the environment is available and the communication constraints may be bridged, allowing for centralised training. This centralised training of decentralised policies is a standard paradigm for multi-agent planning [19].

2.4 Communication in Multi-agent Reinforcement Learning

[21] shows that sharing information among agents provides better cooperation and more effective execution. For multiple agents to cooperate efficiently there must be a medium of communication among them, especially since each agent can only observe a part of the environment. [22] demonstrates a method to include communication in multi-agent systems by adding a communication sub-stage before the action stage, where agents decide whether to communicate. This can result in agents voluntarily sending information or agents querying for it. However, in the presence of a large number of agents, it becomes necessary for agents to differentiate between key information that may aid cooperation from globally shared information else the

communication may hinder the learning process altogether [23]. Decentralised execution offers another advantage wherein agents can learn from the experiences of other agents. This experience-sharing technique method was used in [24] in a trainer and trainee relationship, where a more experienced agent can help teach an untrained agent. This idea can be expanded to other scenarios where multiple agents performing similar tasks can learn from one another making exploration of states more efficient.

3 Methods

The following sections have been adapted from [11] .

3.1 Reinforcement Learning

Reinforcement learning (RL) algorithms learn solutions for sequential decision processes via repeated interaction with an environment. A sequential decision process is defined by an agent who makes decisions over multiple time steps within an environment to achieve a specified goal. In each time step, the agent receives an observation from the environment and chooses an action. This is depicted in Figure 1.

3.1.1 Environment

The environment in a reinforcement learning context refers either to a virtual or physical system whose state evolves as the agents act upon it. The environment prescribes the set of actions an agent can take, and the percentage of the environment observable by the agents. The action space and the observation space may be continuous or discrete.

3.1.2 Agent

An Agent is in turn the actor which acts upon the environment based on the observation received from it. Agents may have prior knowledge of the environment, such as the state transition probabilities. Agents are goal-oriented and choose actions to achieve their goals. Both RL and MARL define reward functions to quantify these goals. The decision-making function of an agent is referred to as its policy. The policy maps the state observation to actions.

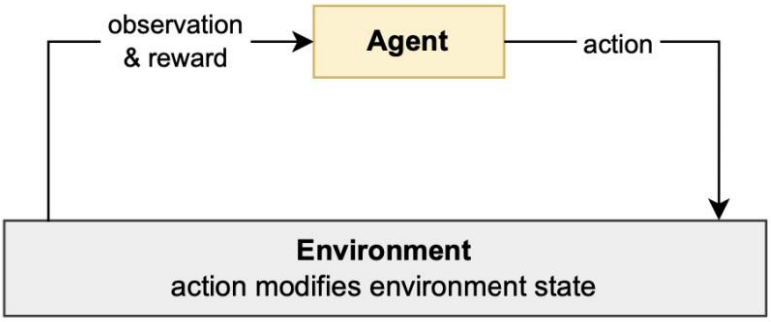


Figure 1. Basic RL loop for a single-agent system [9]

3.2 Multi-Agent Reinforcement Learning

A multi-agent system consists of an environment and multiple decision-making agents that interact in the environment to achieve certain goals which may be cooperative or competitive. A MARL loop is depicted in Figure 2. The level-based foraging problem tackled in this project is a multi-agent system with near-homogeneous agents. In this case, the task is fully cooperative, as both agents must coordinate to be able to achieve a common goal.

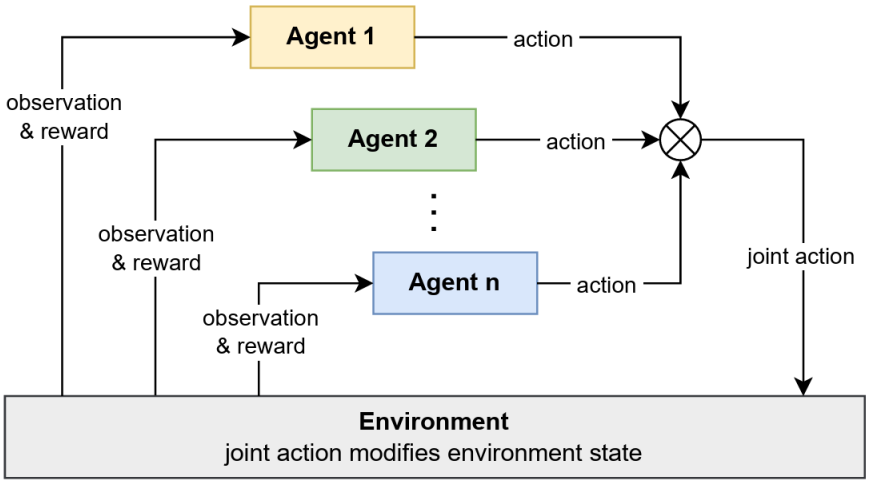


Figure 2. MARL loop [11].

MARL has its own set of challenges; these are detailed in [11]:

Non-stationarity caused by learning agents: agents now must adapt to the policies of other agents, which in turn change continuously. These cyclic and unstable learning dynamics violate the Markov property [18]. Single-agent algorithms applied to MARL achieve adequate results for simpler scenarios, but they are not optimal [25]. One way to extend single-agent algorithms to

multi-agent systems is to use joint-action space, wherein one policy chooses actions for both agents, but this method limits scalability.

Multi-agent credit assignment: In single agent RL the reward received from the environment can be directly assigned to the sole agent, but with multiple agents, there needs to be a reward arbitration mechanism.

Optimality of policies and equilibrium selection: In single-agent RL, a policy is optimal if it achieves maximum expected returns in each state. However, in MARL due to the interdependence of agent policies, a more sophisticated definition of optimality is required.

Scaling in number of agents: The possible state-action combinations grow exponentially with each agent.

3.3 Markov Decision Process

Modelling sequential decision processes (SDP) as a Markov Decision Process (MDP) is standard practice in RL. MDP provides a mathematical formalism to model sequential decision-making problems where an agent interacts with an environment over time. What distinguishes MDPs is their adherence to the Markov property, which posits that the future state of the system depends only on the current state and action, independent of the past history. Following provides a formal definition of a Markov Decision Process.

A finite Markov Decision Process (MDP) consists of the following elements.

- A finite set of states $S = \{s^1, \ldots\ldots, s^N\}$, where N is the size of the state space. A state is a unique characterisation of all the vital information about the environment.
- A finite set of actions $A = \{a^1, \ldots\ldots, a^K\}$ where K is the size of the action space.
- Reward function R. The reward function can specify rewards for being in a particular state. But depending on the scenario it can also specify the reward for performing an action or achieving a certain state transition.
- State Transition probability function T.
- Initial State probability $Pr^0: S \rightarrow [0,1]$.

A policy $\pi(a^t|s^t)$ is a computable function that outputs an action a for each state s. A MDP starts with an initial state sampled from Pr^0, at time t agent observes the current state s^t and chooses an action a^t with probability given by its policy $\pi(a^t|s^t)$. Given the state s^t and action a^t, the MDP transitions into a next state s^{t+1} with probability given by $T\ (s^{t+1}|s^t, a^t)$, and the agent

receives a reward r^tgiven by the reward function $R(s^t, a^t, s^{t+1})$.These steps are repeated until a termination criterion is met. Where $(a|s)$ notation denotes the probability of choosing action a given state s.

3.3.1 Optimality Criteria and Discounting

Optimality can be characterised in two ways, whether the goal is met and whether is it done efficiently. MDP learns by gathering rewards, although if done so by solely optimising the expected reward at each state it would lead to myopic results. The finite horizon model defines a length h and optimises the expected rewards in this interval. In the Infinite horizon model, the long-run rewards are taken into account although the future rewards are discounted with a discount factor γ, with $0 \leq \gamma \leq 1$. Rewards obtained further down the line are discounted more than rewards obtained earlier, due to the non-deterministic nature of a stochastic MDP. The discount factor also ensures that the reward remains finite in the infinite horizon model.

3.3.2 Value Function and Bellman Equation

Given that all MDPs hold the Markov property, the future returns for each state can be quantified, this is done via Value Functions. Value Functions are one of the central ideas of RL. The cumulative future rewards at each timestep is given by:

$$u^t = r^t + \gamma r^{t+1} + \gamma^2 r^{t+2} + \cdots \quad (1)$$

where γ is the discount factor, u^t is the total expected reward at timestep t. r^t and r^{t+1} are the rewards obtained at timestep t and t+1 respectively. This can be written recursively as:

$$u^t = r^t + \gamma u^{t+1} \quad (2)$$

Given a policy π, the state-value function $V_\pi(s)$ gives the "value" of state s under policy π, which is the expected return when starting in state s and following the policy to select actions, formally :

$$V_\pi(s) = \mathrm{E}_\pi[u^t | s^t = s] \quad (3)$$

$$V_\pi(s) = \mathrm{E}_\pi[r^t + \gamma u^{t+1} | s^t = s] \quad (4)$$

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} T(s'|s,a)\left[R(s,a,s') + \gamma \mathrm{E}_\pi[u^{t+1} | s^{t+1} = s]\right] \quad (5)$$

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} T(s'|s,a)[R(s,a,s') + \gamma V_\pi(s')] \quad (6)$$

where E_π is the expected u^t under policy π. Equation 5 is a recursive equation called the Bellman equation.

Similarly, the Action value function $\mathrm{Q}_\pi(\mathrm{s},\mathrm{a})$ can be defined, which gives the expected return from taking action a in state s as:

$$Q_\pi(\mathrm{s}) = \mathrm{E}\pi[u^t|s^t = s, a^t = a] \quad (7)$$

$$Q_\pi(\mathrm{s}) = \mathrm{E}\pi[r^t + \gamma u^{t+1}|s^t = s, a^t = a] \quad (8)$$

$$Q_\pi(\mathrm{s}) = \sum_{s' \in S} T(s'|s,a)[R(s,a,s') + \gamma V_\pi(s')] \quad (9)$$

$$Q_\pi(\mathrm{s}) = \sum_{s' \in S} T(s'|s,a)\left[R(s,a,s') + \gamma \sum_{a' \in A} \pi(a'|s')Q_\pi(s',a')\right] \quad (10)$$

The Bellman equation for state-values and action-value function, defines a system of linear equations with N variables, where N is the number of states. This system of equations can then be solved to obtain V_π and Q_π. Although this is computationally difficult and does not help to find the optimal policy. There are two predominant families of algorithms to compute optimal Value functions and optimal policies.

3.3.3 solving MDPs

Solving an MDP means finding an optimal policy π^*. The two widely recognised categories of algorithms are model-based and model-free algorithms. Model based algorithms generally use Dynamic Programming (DP), while model-free algorithms are RL algorithms. All algorithms use the generalized policy iteration (GPI) principle. This principle has two iterative processes. The policy evaluation step estimates the utility of the current policy, by calculating the value function V_π. The policy improvement step evaluates the actions in each state to improve the policy. Both of these steps are handled differently in the two families of algorithms.

3.3.4 Dynamic Programming

Dynamic Programming (DP) is a method wherein a large problem is broken down into smaller subproblems, and a solution can be built incrementally from previously solved subproblems. DP also is an umbrella term for a class of algorithms primarily using DP techniques to find the optimal policies for model-based learning. In dynamic programming, the two GPI steps as shown in Figure 3 are Policy Evaluation, which involves computing the value function V^π for the current policy π, and Policy Improvement, where the current policy π is enhanced with respect to V^π. Thus, policy iteration produces a sequence of policy and value function estimates, which converges to optimal value function. The Bellman equation uses the value estimate for the next state to estimate the value for the current state, this property is known as bootstrapping and forms a critical part of many RL algorithms.

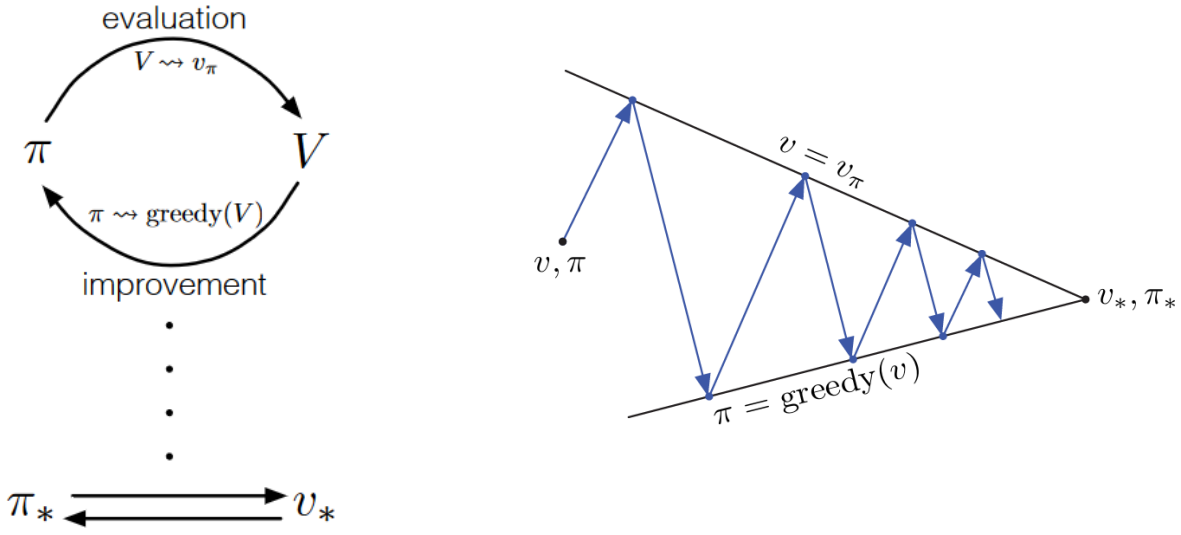
The Bellman equation (5) for V_π defines a system of N linear equations, solving them to find V_π using Gauss elimination has time complexity $O(N^3)$. Iterative policy evaluation instead produces a sequence of policy and value function estimates, starting with some initial policy π^0 and value function V^0 and then applies the Bellman equation for V_π iteratively to produce successive estimates of V_π. The k[th] iteration of the value estimate is formally given as:

$$\mathrm{V}^{\mathrm{k+1}} \leftarrow \sum_{a \in A} \pi(a|s) \sum_{s' \in S} T(s'|s,a)\left[R(s,a,s') + \gamma \mathrm{V}^{\mathrm{k}}(s')\right] \tag{11}$$

Thus, DP requires complete knowledge of the MDP model, including the reward function R and state transition probability T.

Once V_π is calculated, the policy improvement task modifies π by making it greedy with respect to V_π, that is choosing actions which provide maximum reward for all states.

$$\pi' = arg \max_{a \in A} T(s'|s,a)\left[R(s,a,s') + \gamma \mathrm{V}^{\mathrm{k}}(s')\right] \tag{12}$$



(a) (b)

Figure 3. a) Generalised Policy Iteration. b) Policy evaluation step [15].

While DP algorithms do not interact with the environment, the basic concepts of DP are an important building block in RL theory and temporal Difference learning.

3.3.5 Temporal-difference (TD) learning

Temporal-difference learning is a family of RL algorithms which learn value functions and optimal policies based on experiences from interactions with the environment. Like the DP algorithm, TD learning also learns value functions based on the Bellman equations and bootstrapping. However, TD learning does not require complete knowledge of the MDP model, making it more useful in practice.TD update rule to learn action-value functions is given by:

$$Q(s^t, a^t) \leftarrow Q(s^t, a^t) + \alpha[\chi - Q(s^t, a^t)] \tag{13}$$

Where χ is the update target, and $\alpha \in [0, 1]$ is the learning rate. The target is constructed based on the experience sample collected from interaction with the environment.

SARSA [14] is a TD algorithm which constructs an update target based on:

$$\chi = r\hat{}t + \gamma Q(s^{t+1}, a^{t+1}) \tag{14}$$

Here, the action a^{t+1} is sampled from the policy π in the successor state s^{t+1}. The complete SARSA update rule is specified as:

$$\mathrm{Q}(s^t, a^t) \leftarrow \mathrm{Q}(s^t, a^t) + \alpha[r^t + \gamma Q(s^{t+1}, a^{t+1}) - \mathrm{Q}(s^t, a^t)] \tag{15}$$

As seen by Equation 14, SARSA is an on policy algorithm, as the update target is based on the next state achieved by following the current policy.

Similarly, Q-learning is a TD algorithm which constructs an update target based on the above Bellman optimality equation. However Q-learning is off policy as the current Q value is updated based on the maximum action value in the next state. Further Q-learning uses the Bellman optimality equation instead of the Bellman equation for Q_π. The Bellman optimality equation is given by:

$$Q^*(s, a) = \sum_{s' \in S} T(s'|s, a) \left[R(s, a, s') + \gamma \max_{a' \in A} Q^*(s', a') \right] \tag{16}$$

$$\chi = r^t + \gamma \max_{a' \in A} Q(s^{t+1}, a') \tag{17}$$

The complete Q learning update rule is given as:

$$\mathrm{Q}(s^t, a^t) \leftarrow \mathrm{Q}(s^t, a^t) + \alpha \left[r^t + \gamma \max_{a' \in A} Q(s^{t+1}, a') - \mathrm{Q}(s^t, a^t) \right] \tag{18}$$

Algorithm 1 gives the complete procedure to find the optimal policy using Q-learning.

The fact that Temporal difference learning updates the value function based on immediate rewards and expected rewards, the training becomes sensitive to the reward function. If the agent only gets rewarded for achieving the final goal after many steps the rewards are sparse, this may lead to very slow learning time. The agent can be awarded intermediate rewards that help the agent learn the right steps faster, it essentially breaks the task into smaller sub-tasks. However, this requires careful deliberation as the agent may just optimise to these sub-tasks and may not learn what is intended.

Algorithm 1: Q-Learning for MDPs

Initialise: Q(s, a) = 0 for all $s \in S, a \in A$
Repeat for every episode:
for t=0, 1, 2, ... **do**
Observe current state s^t
With probability ε: choose a random action at $a^t \in A$
Else: choose action $a^t \in argmax_a Q(s^t, a)$
Apply action a^t, observe reward r^t and next state s^{t+1}

$$\mathrm{Q}(s^t, a^t) \leftarrow \mathrm{Q}(s^t, a^t) + \alpha\left[r^t + \gamma \max_{a' \in A} Q(s^{t+1}, a') - \mathrm{Q}(s^t, a^t)\right]$$

Exploration vs. Exploitation

A critical aspect of reinforcement learning is the trade-off between exploration and exploitation while an agent interacts with an environment. Exploitation is taking the action with the highest estimated value based on past experience. Exploration is taking a random action to discover potentially better options Since reinforcement learning is model-free, the agent's actions determine the information that is returned from the environment. A greedy agent who chooses already known states that give immediate rewards may not explore other strategies that may be more efficient in fulfilling the final objective. The ε-greedy strategy is a widely used approach that elegantly addresses this challenge. It involves the agent choosing between exploitation and exploration based on a parameter epsilon. When epsilon is high, the agent prioritizes exploration, making random choices to gather information about the environment and potentially uncover better strategies. Conversely, when epsilon is low, the agent leans more towards exploitation, favouring actions with higher expected rewards based on its learned knowledge. This adaptive nature of epsilon-greedy allows agents to gradually transition from exploration to exploitation as they gather more experience and refine their decision-making abilities, striking a fine balance between discovering new possibilities and exploiting known effective strategies.

3.4 Deep Learning

Function approximators allow the representation of value functions which generalise and therefore can give a reasonable estimation for similar states even though they may not be directly trained.

3.4.1 Feedforward Neural Networks

Feedforward neural networks – also called deep feedforward networks, fully- connected neural networks, or multilayer perceptrons (MLPs) – are the most prominent architecture of neural networks and can be considered the building block of most neural networks [11] . Figure 4 shows the basic structure of a neural network and Figure 5 shows the structure of each node in a neural network.

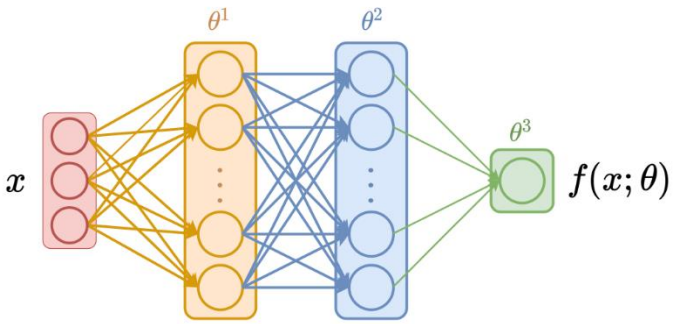


Figure 4. Illustration of a feedforward neural network with three layers [11].

As a general-purpose function approximator, a neural network learns a function f (x; θ) for some input $x \in \mathbb{R}^d$. Training involves optimizing θ to make f approximate a target function $f^*(x)$.

Feedforward neural networks are structured into multiple sequential layers with the first layer processing the given input x and any subsequent layer processing the output of the previous layer.

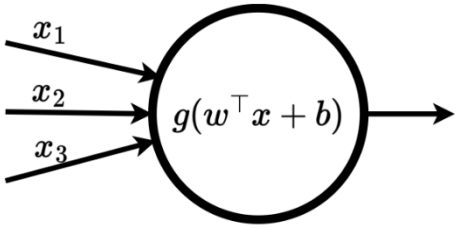


Figure 5. Illustration of a single neural unit computing a scalar output given an input $x \in \mathbb{R}^3$ [11].

An individual unit in a neural network processes inputs from the previous layer using weighted sums and non-linear activation functions. Applying non-linear activation functions to the output of

each unit is essential because the composition of two linear functions can only represent a linear function itself and hence non-linear functions could not be approximated by composing an arbitrary number of neural units without non-linear activation functions.

3.4.2 Gradient Descent Optimisation

The parameters θ of a neural network sometimes referred to as weights of the network, must be optimised to obtain a function f which accurately represents a target function f^*.

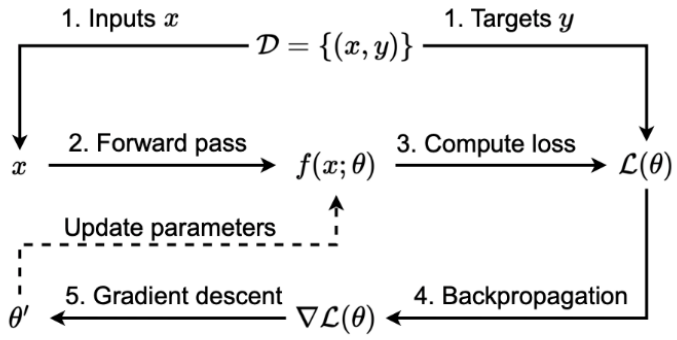


Figure 6. The training loop for gradient-based optimisation of a neural network parameterised by θ [11].

There are three key aspects to gradient descent optimisation: Loss function, Gradient-based optimiser, and Backpropagation as depicted in Figure 6. Training a neural network involves an optimization process to find the optimal set of parameters that minimize a defined loss function. Gradient-based optimization methods, such as gradient descent variants like stochastic gradient descent (SGD) and mini-batch gradient descent, iteratively update the network parameters to minimize the loss. These methods calculate gradients of the loss function with respect to the parameters and update them in the direction that reduces the loss, effectively "descending" the loss landscape towards a local minimum.

3.5 Deep Q-Learning

In tabular Q-learning, the agent maintains an action-value function Q, in the form of a table. The essential idea behind Deep Q-Learning is to replace this table with a neural network to reap the benefits of generalisation. To realise this a feedforward neural network is initialised for each agent. The network receives states as inputs and a forward pass returns the action-value estimates for all actions. The loss function may be defined as the squared error between the value function

estimate and the target value using the same experience of a single transition as for tabular Q-learning, formally:

$$\mathrm{L}(\theta) = \left(\mathrm{r}^{\mathrm{t}} + \gamma \max_{a'} Q(s^{t+1}, a'; \theta) - Q(s^t, a^t; \theta)\right)^2 \quad (19)$$

This simple extension of Q-learning suffers from two major issues. First, the moving target problem is further exacerbated by the application of an approximated value function and the strong correlation of consecutive samples used to update the network leads to an undesirable specialisation of the most recent experiences, sometimes referred to as catastrophic forgetting. To reduce the instability of training caused by the moving target problem, [16] proposes an additional network called the target network. The target network $\hat{Q}$ with parameters $\overline{\overline{\Theta}}$ and the same architecture as the primary network is initialised. The target network can then be used instead of the primary network to compute bootstrapped target values in the following loss function:

$$\mathrm{L}(\theta) = \left(\mathrm{r}^{\mathrm{t}} + \gamma \max_{a'} \hat{Q}(s^{t+1}, a'; \theta) - Q(s^t, a^t; \theta)\right)^2 \quad (20)$$

Instead of optimising the target network with gradient descent, the parameters of the target network are periodically updated by copying the current parameters of the value function network to the target network .

The second problem of naive deep Q-learning is the correlation of consecutive experiences. In many paradigms of machine learning, it is generally assumed that the data used to train the function approximation are independent of each other and identically distributed or “i.i.d. data” in short. This assumption guarantees that, firstly, there are no correlations of individual samples within the training data, and secondly, that all data points are sampled from the same. training distribution. Both of these components of the i.i.d. assumption are typically violated in RL. To address this issue [16] suggests an experience replay buffer be implemented. A minibatch of random experiences can be selected and values loss over the entire batch can be calculated. The replay buffer is implemented with a fixed capacity as a first-in-first-out queue. Putting it all together gives the Algorithm 2; the algorithm is an extension of the algorithm provided in [16] to the multi-agent scenario.

Algorithm 2: Q-Learning for MDPs

Initialise network Q_1 for agent 1
Initialise network Q_2 for agent 2
Initialise target network $\hat{Q}_1$ for agent 1
Initialise target network $\hat{Q}_2$ for agent 2
Initialise experience replay memory D_1 for agent 1
Initialise experience replay memory D_2 for agent 2
Initialise the Environment to spawn agents and items
While not converged do
    /* sample phase
    $\epsilon \rightarrow$ decay epsilon
    Choose an action a_1 and a_2 given state s_1 and s_2 using policy ϵ-greedy
    Both Agents take corresponding actions, observe reward r, and next state s'
    Store Transitions $(s_1, a_1, r_1, s_1', done_1)$in the experience replay memory D_1
    Store Transitions $(s_2, a_2, r_2, s_2', done_2)$ in the experience replay memory D_2

    If enough experiences in D_1 and D_2 then
        /*Learn phase
        for i from 1 to 2 do
            Sample a minibatch of N transitions from D_i
            for every transition $(s_k, a_k, r_k, s_k', done_k)$ in minibatch do
                if $done_k$ then
                    $y_k = r_k$
                else
                    $y_k = r_k + \gamma \max_{a' \in A} \hat{Q}_i(s_k', a_k')$
                end
            end
            Calculate the loss $L = \frac{1}{N}\sum_{i=0}^{N-1}(Q(s_k, a_k) - y_k)^2$
            Update Q using SGD algorithm by minimising loss L
            Every C steps, copy weights from Q_i to $\hat{Q}_2$
        end
    end

end

4 Results and discussion

The DQN algorithm implemented performs as expected. The algorithm is very straightforward but is bounded by its limitations. Since the agent policies are completely independent, they struggle to learn a cooperative strategy.

4.1 Problem Description

level-based foraging (LBF) environment [26] consists of n agents/players and m items. Each field in the grid is either empty or occupied by one player or one item. All players and items have levels, where no food has a level greater than the sum of k players, where k is less than equal to n. This requires cooperation between players to be able to collect all items in each episode. Each player observes the full environment and chooses an action $a_i \in$ {up, down, left, right, collect, noop}. A group of players can load an item if they are placed on fields next to the item and if the sum of their levels is at least as high as the item's level. An episode terminates either when all items have been collected or the maximum time steps have occurred. The Python environment is provided by the authors of [27].

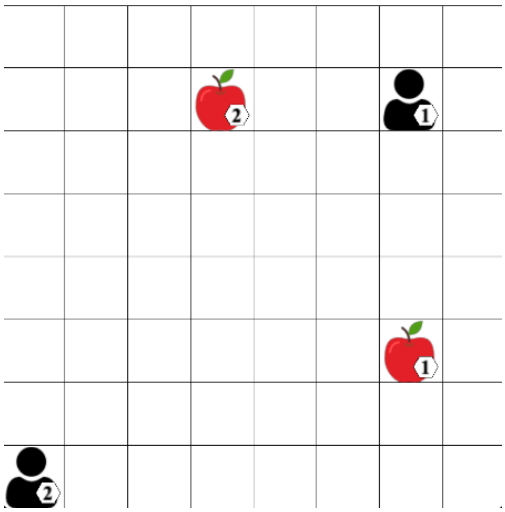


Figure 7. Level-based foraging environment render, with two agents and two items. The implementation provided by the authors of [27].

The level-based foraging environment approximates many scenarios encountered by distributed robots. For example, it can be generalised to any type of mobile robots having to navigate across an environment to complete specific tasks, these tasks can be anything from picking up an item in a warehouse to collecting soil samples on mars. That is why the LBF environment is chosen to test the DQN agents.

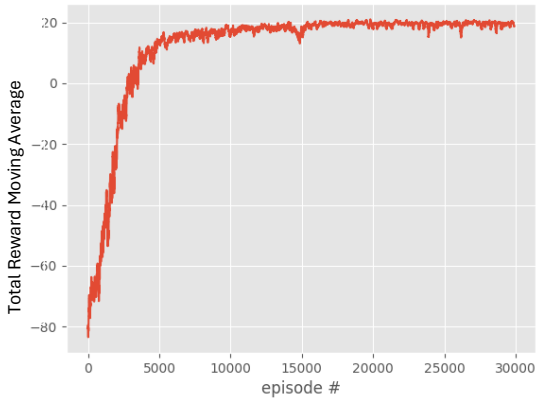
4.2 Tabular Q-learning

A separate environment is created to test tabular Q-learning methods. Wherein every episode only one agent and one item are spawned, to test the single agent case. The agent must occupy the same grid space as the item for the item to be automatically collected. The collection action is

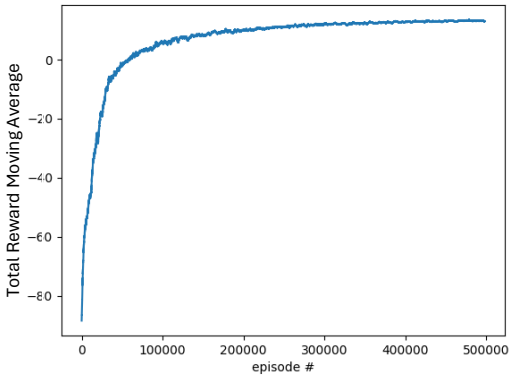
removed in testing tabular methods to keep the action space contained. The tabular method must keep track of 4 actions which are $a_i \in$ { up, down, left, right}, for all states. In the single-agent case, the state is represented by a tuple of two numbers. The first number denotes the difference in x-coordinates of the item and the agent, and the second number denotes the difference in y-coordinates of the item and the agent. For a nxn grid, the state space has a size of $(2n-1)^2$.

The agent is penalised by 1 for every step it takes, this puts emphasis on moving to the target in the most efficient way possible. The agent is rewarded by 25 when it reaches the item. Although this makes the rewards sparse, the discount factor is set to 0.99 which is relatively high, allowing the final reward to trickle down to intermediate states. The agent is initialised with an epsilon of 1, which is decremented exponentially each episode. This means that the agent initially takes completely random action and starts taking on-policy actions in later episodes. This allows the agent to explore more states, hence the agent can learn to reach the target from any state. Exploration is especially important in tabular methods because unlike neural networks a tabular representation cannot generalise to nearby states. Figure 8(a) shows the last 100 episodes moving average of rewards, the agent learns to move to the item efficiently, by the end only requiring on average 5 steps to reach the item.

A multi-agent scenario is also implemented using tabular Q learning. In this scenario, two agents are spawned instead of one and both of them have to occupy the same square as the item for it to be automatically collected. Again, the collection is set to automatic to reduce the action space. This is especially important for the multi-agent case as a single Q-function is used to represent the joint policy. For every state there are K^2 total actions, where K is the number of actions in the action space of each agent, in this case K is equal to 4. The states are represented by a tuple of the coordinates of the two agents and the item. The size of the state space is given by $(nxn)^3$, where n is the number of rows and columns in the grid. Hence the table must store $(nxn)^3 \times K \times K$ action values, which in this case is 4194304. This simplified scenario is implemented successfully as shown in Figure 8(b).



(a) single-agent tabular q-learning training results.



(b) multi-agent tabular q-learning training results.

Figure 8. Training results for Tabular q-learning.

4.3 Naïve DQN

Given the inherent problems with tabular Q-learning a naïve DQN approach is implemented for the level-based foraging environment as described in section 4.1. The algorithm is similar to the one in the tabular case, but in this approach the Q function is represented by a feed-forward neural network, in a decentralized fashion wherein each agent has its separate Q function approximator. This approach suffers from the problems described in section 3.6 and fails to learn as seen in Figure 9. As the update target is based on the previous experience, there is a high correlation between samples and the agents fail to learn. Given the update target is derived from the same network this introduces the moving target problem further disrupting the learning process.

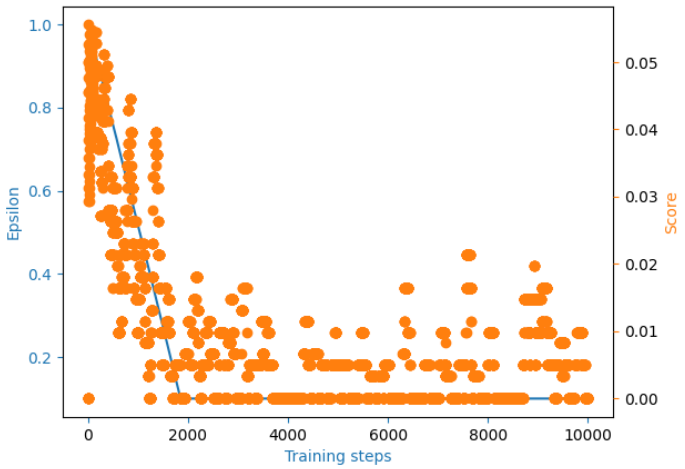


Figure 9. Training results for naïve DQN for one agent

4.3 DQN with a target network and experience replay

To deal with the issues of the naïve approach, the solutions presented in [16] as described in section 3.6 are implemented. The DQN algorithm is tested under two scenarios. In the first case, the item levels are randomised between 1 and 3. The agent levels are always 1 and 2. Hence agents only needed to cooperate for the item with level 3. The agents are penalised by 1 for every action and rewarded by 100 if all the items were collected. Proportional rewards are rewarded in case of partial collection. This reward system is programmed separately as the environment by default does not apply any penalties. Without the action penalty, the learning is unstable and fails. This is attributed to the fact that without a penalty for every action, agents are not inclined to take the most optimal path to the items. As the environment is fully observable to both agents and they don't have to find the items by exploring the grid, it is feasible to implement the penalty.

However, given that the agents are only aware of the location of the other agent and do not consult with a global policy nor do they communicate with each other, they are essentially competing for items to maximise individual rewards. In most cases the agent with level 2 would be able to collect both items, however agent with level 1 would also try to collect the items before they were collected by the other agent. This resulted in two positive side effects. Firstly, the items were being collected in fewer time steps as both agents were eager to collect the items. Secondly, when an item with level 3 would spawn, both agents would rush to the food to try and collect it individually, but coincidentally this would result in a coordinated collection.

Figure 10 shows the last 100 episodes moving average of the summed rewards of both agents. By the end of the training period, the agents can achieve a reward of 0.8, with the maximum being 1. However, as the maximum permitted step in each episode is limited to 50, sometimes the agents are not able to collect the level 3 food, due to the policy not being perfect.

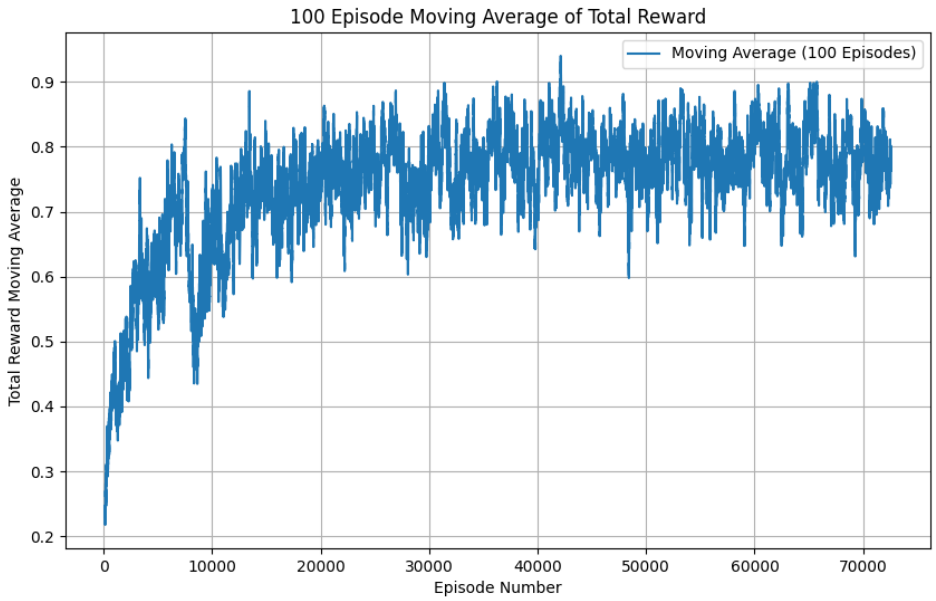


Figure 10. Training results for DQN with target network and experience replay under mixed cooperative setting.

Another key point was the algorithms sensitivity to learning rate, although the ADAM optimiser dynamically adjusts the learning rate, failing to choose an appropriate starting point led to non-convergence. When a learning rate of 0.005 was implemented, the learning was settled to a local minimum and as the agents started taking more and more actions on-policy the rewards fall as seen in Figure 11.

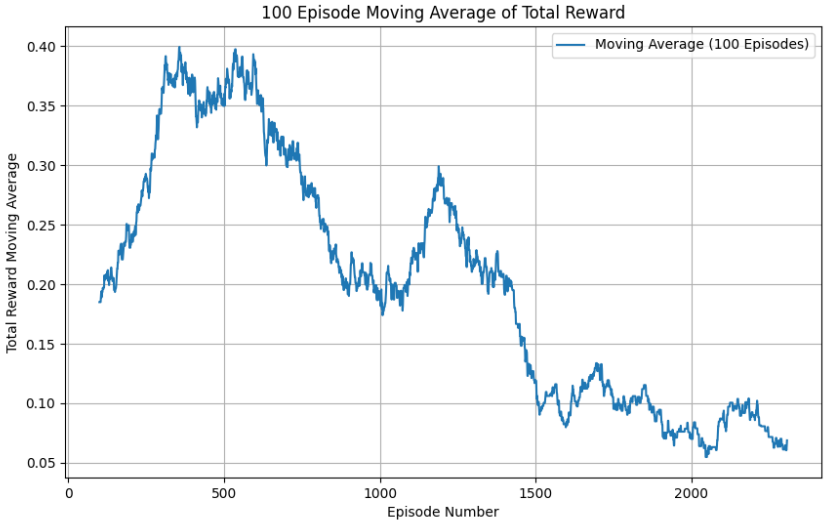


Figure 11. Training results with lr = 0.005.

When the agents are forced to cooperate, the agents fail to learn as seen in Figure 12. This is expected as there is no communication between agents. During the learning stage agents are only rewarded when an item is collected, which is a rare occurrence as the exploration is done through random actions using ε-greedy method. One way to mitigate this is to implement a comprehensive reward shaping, wherein the agents are rewarded for staying together and moving to the same target. This extensive reward shaping is essentially an added heuristics to compensate for the lack of decentralised training. The reward shaping includes a reward of 0.5 for both agents if the distance between them is less than 4 grid spaces. This distance is obtained by subtracting the respective row values and column values of both agents and finding an absolute value. This reward incentivises the agents to essentially stay in formation and move together, as required in a fully cooperative scenario. Further penalty for moving is increased to 2, but penalty for staying in the same grid is kept at 1. This is because most of the time an agent would get to one of the items first and would have to wait for the other agent to arrive. This reward shaping leads to much better results as shown in Figure 13. Although this proved an effective solution, it involves a fair amount of trial and error and is not very robust. It also requires that during exploration stage, when epsilon is still high the agents coincidentally collect the items so there are enough experiences where agents see the high final reward from collecting the item, else the agents just optimise to remain in formation, and stay at a single location.

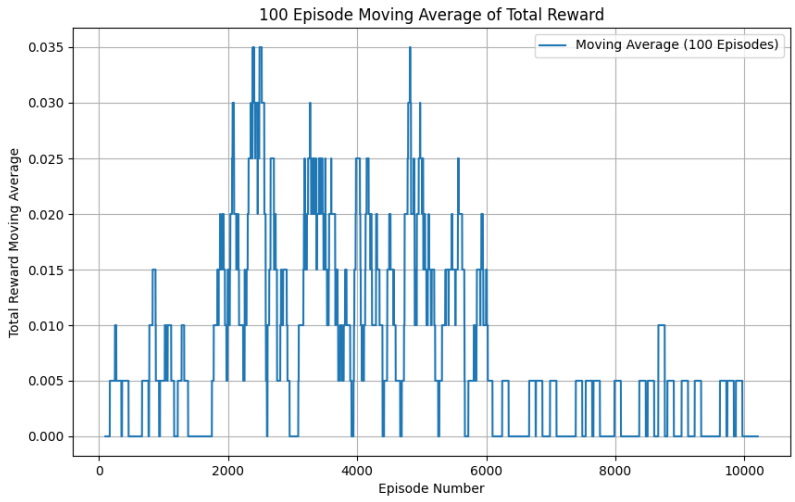


Figure 12. Training results under forced cooperation setting, without reward shaping

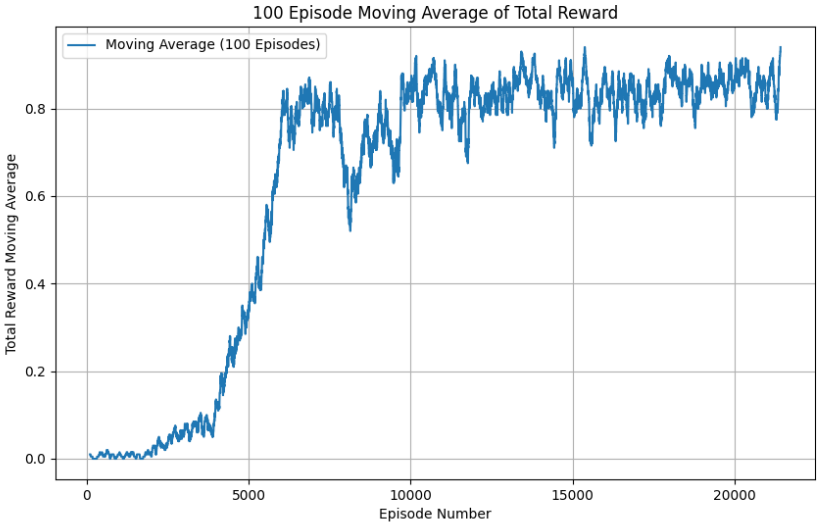


Figure 13. Training results under forced cooperation setting with extensive reward shaping.

5 Conclusions and future work

5.1 Conclusions

It is clear from this project that multi-agent systems offer immense benefits in completing complex tasks. But to reap the full benefits of a theoretical multi-agent system, the agents must learn to act independently and collectively. The exploration into multi-agent reinforcement learning (MARL) within the context of cooperative environments has unveiled both challenges and promising avenues for future research.

Albeit this project does not use state of the art methods, several key insights have emerged. Firstly, the complexity introduced by MARL's non-stationarity due to learning agents adapting to evolving policies of other agents poses a significant challenge. While single-agent algorithms can suffice for simpler scenarios, their scalability and optimality in multi-agent settings are limited.

The implementation of tabular Q-learning, both in single-agent and multi-agent scenarios, demonstrated the fundamental principles and limitations associated with tabular representations. Despite success in simpler setups, the scalability of tabular methods quickly becomes prohibitive as state and action spaces grow, clearly identifying the for neural networks.

Transitioning to DQN, a leap in complexity and capability is evident. The multi-agent scenario which was too complex for tabular methods is achievable using neural networks. However, the

naïve approach highlights the pitfalls of correlated experiences and the moving target problem. These issues necessitate the adoption of strategies like using a target network and experience replay to stabilize training and improve learning efficiency. Testing DQN agents in the level-based foraging environment shows compelling results for its viability. However, it is evident from the forced cooperation scenario that the DQN agent isn't ideal for complex multi agent scenarios. Since the training is decentralised and there is no form of communication between the agents during training or execution the agents may fail to learn an optimal joint policy, requiring additional reward shaping.

5.2 Future Work

Moving forward, exploring policy gradient methods can be highly beneficial. Unlike value-based methods like Q-learning, policy gradient methods directly learn the policy without estimating the value function, this overcomes the challenges faced when representing the value functions using neural networks. Exploring the paradigm of centralised training and decentralised execution can lead to more efficient multi-agent learning. During training, agents can share information and coordinate actions centrally to learn cooperative strategies. However, during execution, agents act independently based on their learned policies, possibly communicating with each other for coordination. [27] successfully implements a novel algorithm called the shared-experience actor critic method that outperforms DQN in fully cooperative level-based foraging scenario. This method can be taken further by adding a communication layer during training and execution. Further integrating spatial representation of states and leveraging Convolutional Neural Networks (CNNs) to process observations can enhance the agent's understanding of the environment. CNNs also present a unique opportunity wherein during training a random number of agents can be spawned, hence the critic network can learn how to handle the same tasks with different number of agents, this imparts modularity and redundancy to the MAS. Additionally, incorporating constraints such as partial observability between agents can simulate real-world scenarios where agents have limited information about each other's states or actions. Techniques like multi-agent reinforcement learning with partial observability (MARL-PO) or attention mechanisms can be explored to address these challenges and improve coordination in complex environments.

References

[1] Y. Li, “Deep reinforcement learning: An overview,” *arXiv preprint arXiv:1701.07274,* 2018.

[2] A. Krnjaic, R. D. Steleac, J. D. Thomas, G. Papoudakis, L. Schäfer, A. W. K. To, K.-H. Lao, M. Cubuktepe, M. Haley, P. Börsting and S. V. Albrecht, “Scalable Multi-Agent Reinforcement Learning for Warehouse Logistics with Robotic and Human Co-Workers,” arXiv:2212.11498, 2023.

[3] S. Shalev-Shwartz, S. Shammah and A. Shashua, “Safe, Multi-Agent, Reinforcement Learning for,” arXiv:1610.03295, 2016.

[4] M. Lanctot, V. Zambaldi, A. Gruslys, A. Lazaridou, K. Tuyls, J. Perolat, D. Silver and T. Graepel, “A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning,” arXiv:1711.00832, 2017.

[5] Z. Li, V. Tam and K. L. Yeung, “Developing A Multi-Agent and Self-Adaptive Framework with Deep Reinforcement Learning for Dynamic Portfolio Risk Management,” arXiv:2402.00515, 2024.

[6] A. Dorri, S. S. Kanhere and R. Jurdak, “Multi-Agent Systems: A Survey,” *IEEE Access,* vol. 6, pp. 28573-28593, 2018.

[7] P. Stone and M. Veloso, “Multiagent systems: A survey from a machine learning perspective,” *Autonomous Robots,* vol. 8, pp. 345-383, 2000.

[8] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning,* vol. 8, pp. 279--292, 1992.

[9] R. Bellman, “Dynamic programming,” *science,* vol. 153, no. 3731, pp. 34--37, 1966.

[10] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Machine learning,* vol. 3, pp. 9--44, 1988.

[11] S. V. Albrecht, F. Christianos and L. Schäfer, Multi-Agent Reinforcement Learning: Foundations and Modern Approaches, MIT Press, 2024.

[12] M. Van Otterlo and M. Wiering, “Reinforcement learning and markov decision processes,” in *Reinforcement learning: State-of-the-art*, Springer, 2012, pp. 3--42.

[13] Y. Wang and C. W. d. Silva, “A machine-learning approach to multi-robot coordination,” *Engineering Applications of Artificial Intelligence,* vol. 21, no. 3, pp. 470--484, 2008.

[14] M. O. Sayin, K. Zhang, D. S. Leslie, T. B. sar and A. Ozdaglar, “Decentralized Q-learning in zero-sum Markov games,” *Advances in Neural Information Processing Systems,* vol. 34, pp. 18320--18334, 2021.

[15] P. Hernandez-Leal, M. Kaisers and T. Baarslag, “A survey of learning in multiagent environments: Dealing with non-stationarity,” *arXiv:1707.09183,* 2017.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, . S. Petersen, . C. Beattie, A. Sadik, I. Antonoglou, H. King, . D. Kumaran, . D. Wierstra, S. Legg and . D. Hassabis, “Human-level control through deep reinforcement learning,” *nature,* vol. 518, no. 7540, pp. 529--533, 2015.

[17] J. R. Kok and N. Vlassis, “Sparse cooperative Q-learning,” in *Proceedings of the Twenty-First International Conference on Machine Learning*, 2004.

[18] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

[19] R. S. Sutton, S. Singh and D. McAllester, “Comparing policy-gradient algorithms,” *IEEE Transactions on Systems, Man, and Cybernetics,* 2000.

[20] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli and S. Whiteson, “Counterfactual multi-agent policy gradients,” *Proceedings of the AAAI conference on artificial intelligence,* vol. 32, 2018.

[21] M. Tan, “Multi-agent reinforcement learning: Independent vs. cooperative agents,” in *Proceedings of the tenth international conference on machine learning*, 1993.

[22] P. Xuan, V. Lesser and S. Zilberstein, “Communication decisions in multi-agent cooperation: model and experiments,” in *Proceedings of the Fifth International Conference on Autonomous Agents*, 2001.

[23] J. Jiang and Z. Lu, “Learning attentional communication for multi-agent cooperation,” *Advances in neural information processing systems,* vol. 31, 2018.

[24] J. A. Clouse, “On training automated agents,” *Department of Computer Science, University of Massachusetts, CmpSci Technical Report,* pp. 95--109, 1995.

[25] G. Weiss, Multiagent systems: a modern approach to distributed artificial intelligence, MIT press, 1999.

[26] S. V. Albrecht and S. Ramamoorthy, “A game-theoretic model and best-response learning method for ad hoc coordination in multiagent systems,” *arXiv:1506.01170,* 2015.

[27] G. Papoudakis, F. Christianos, L. Schäfer and S. V. Albrecht, “Benchmarking multi-agent deep reinforcement learning algorithms in cooperative tasks,” *arXiv:2006.07869,* 2020.

Appendices

A Project Outline

The project timeline recognizes the following tasks, Figure 14 provides the preliminary Gantt chart.

- General Research phase
- Scenario specific research
- Implementing the agents and the environment
- MARL algorithm implementation and training
- Various milestones for the final report
- And further algorithm implantation phases

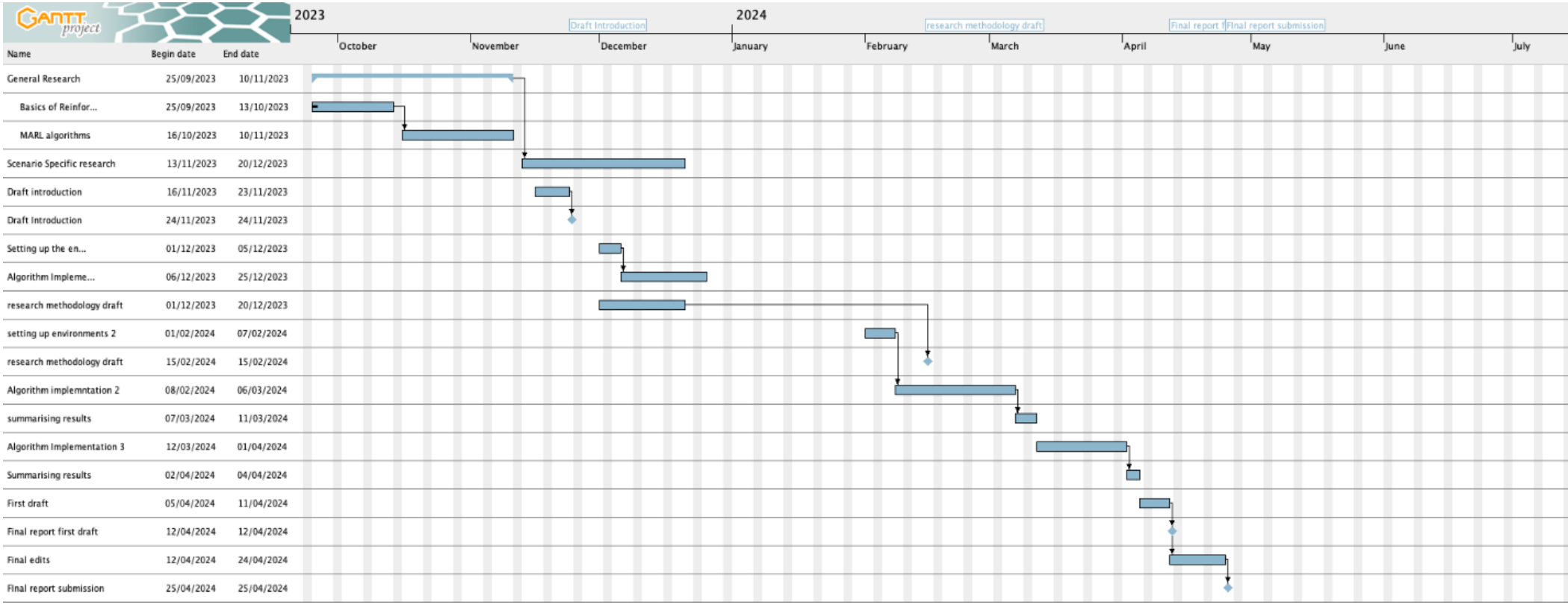


Figure 14. Preliminary Gantt Chart.

B Risk assessment

Potential risks

• Scope Creep: The project may expand and develop into new horizons as new information and insights are gained. This can cause delays. This is mitigated by trying to cement our objectives with time and then focus on them.

Unforeseen Technical errors: Reinforcement learning algorithms can be tricky to train. Algorithm convergence issues are quite common in machine learning. This can cause long training times and the need for additional training data. This may be mitigated by sticking to predefined scenarios for which abundant training data is available.

Table 1. Risk Register.

Project Risk	Severity L=1, M=2, H=3	Potential L=1, M=2, H=3	Score (Severity x Potential) L=1, M=2, H=3	Mitigation Measures
Scope Creep	2	3	6	Have fixed aims and objectives within the initial research phase and ensure no dramatic changes, unless proven necessary.
Algorithm Implementation issues	2	2	4	Choose algorithms that have been proven to work on the machine available. Make sure the training data is easily available before starting implementation.
Scheduling errors	2	2	4	Actively update Gantt chart, identify tasks that may overrun and change the objectives if necessary.

Table 2. General Risk Assessment Form

Date: 20/10/2023	**Assessed by: Mahir Pokar**	**Checked by: Dr. Joaquin Carrasco**	**Location:** Working from home	**Assessment ref no**	**Review date:**
Task / premises: Working from home					

Activity	**Hazard**	**Who might be harmed and how**	**Existing measures to control risk**	**Risk rating**	**Result**
Working from home	Lone working	Home working staff Isolated	1. Please refer to the University Lone Working policy and guidance for more information 2. Please refer to the new University Working at Home guidance 3. Please refer to the new University Wellbeing Support website Staff 4. to remain in regular direct contact with line manager and colleagues via phone, Skype, Zoom, Slack or email	Low	A
Working from home	Poor posture, repetitive movements, long periods looking at DSE (display screen equipment)	Staff, students, visitors Back strain (due to poor posture). Repetitive Strain Injury (RSI) to upper limbs. Eye strain.	1. Complete DSE self-assessment for guidance on how to set up workstation properly 2. Set up workstation to a comfortable position with good lighting and natural light where possible 3. Take regular breaks away from the screen 4. Regularly stretch your arms, back, neck, wrists and hands to avoid repetitive strain injuries. Set up a desktop working space where possible and try to avoid working on a laptop without a docking station	Low	A
Working from home	Stress / Wellbeing	Home working staff Psychosocial effects, Work / Life imbalance, Anxiety	1. Please refer to Stress Prevention and Management toolkit for policies and guidance 2. Please refer to new University guidance for Managing teams working from home 3. Please refer to Seven rules of home working published by AMBS 4. Regular contact meetings with manager and peers, Skype, Zoom, Phone 5. Define working hours, set a start & close daily routine, get dressed and prioritise your tasks. 6. Manager / Employee consultation, wellbeing focused.	Low	A

Use of electrical appliances	Misuse of electrical appliance, faulted electrical appliance.	Home working staff Electric shock, burns and fire	1. 2. 3. 4. 5. 6. 7.	All office equipment used in accordance with the manufacturer's instructions Visual checks before use to make sure equipment, cables and free from defects Avoid daisy chaining and do not overload extension leads University IT equipment brought home should already be PAT tested The domestic electrical supply and equipment owned by the employee is the responsibility of the employee to maintain Liquid spills cleaned up immediately Defective plugs, cables and equipment should be taken out of use	Med	A
Moving around the home office	Obstructions and trip hazards	Home working staff Slips, trips and falls causing physical injury	1. 2. 3. 4. 5. 6.	Floors and walkways kept clear of items, e.g. boxes, packaging, equipment etc Furniture is arranged such that movement of people and equipment are not restricted Make sure all areas have good level of lighting Reasonable standards of housekeeping maintained Trailing cables positioned neatly away from walkways Cabinet drawers and doors kept closed when not in use	Med	A
Working from home	Fire	Staff Home Working Risk of burns, smoke inhalation, asphyxiation	1. 2. 3. 4. 5. 6.	In the event of a fire evacuate out of the building and call the fire brigade All waste, including combustible waste, removed regularly. Heaters located away from combustible materials and switched off when office is left unattended. Avoid daisy chaining and do not overload extension leads Test smoke alarm routinely and replace batteries every 6-12 months Please refer to fire brigade Home Fire Safety and Smoke Alarms	Med	A

C. Software Implementation

The following is the link to the git repository:

https://github.com/MahirPokar/Third_year_MARL_project.git

Third Year Project Git Repository

Applicaiton of Q-learning to Level-based Foraging Environment

Table of Contents

Environments

This project implements three different Q-learning algorithms. The Environment for the tabular methods are Implemented by myself. However for deep RL methods, the Level-based Foraging environment created by Filippos Christianos was used. The environment is provided under MIT License. It is recommended to first install the environment from the link provided above, and then test the algorithms seperately. It is recommended to clone the enviroment git repository in a python virtual environment with python version 3.7 or older. Then this git can be cloned executed. The requirements file provide the required python modules.

Tabular Q-learning

The Single Agent case and the multi agent case both store the action values in a python dictionary object, with states acting as keys. For the single agent case each state has a list of 4 action values. In the multi agent case each state has a list of 16 action values, as a joint policy is implemented. The util file provides the code for ploting the results.

Deep Q-learning

The neural networks and deep learning techniques are implemented using pytorch. The pytorch website provides details for easy installation and usage documentation.

The naive approach implements a very similar algorithm to the tabular case, however two seperate neural networks are used to represent the Q functions for each agent, and the update target is then used to compute the gradient to optimise the nueral network. However this approach does not converge. However its still included as a stepping stone.

The DQN agent fixes the convergence issues by implementing a replay buffer and a target network for each agent. The ENV_AGENT files defines the Linear deep network class and the replay buffer class for improved readability. The raw results received from the environment are stored in a CSV file.

To change the environment to a forced_coop settings, the authors recommend declaring a new environment as described in their documentation, however this can also be done by explicitely changing the forced_coop boolean value in the environment source code. The DQN agent with reward shaping is able to successfully learn how to act in under the forced cooperation contraint.

Known Issues

1. The requirements file neeeds to be updated to remove unecessary requirements.