Advanced Topics in Reinforcement Learning
Multi-Agent Reinforcement Learning

Ruan de Kock
Department of Mathematics & Applied Mathematics
University of Cape Town
`DKCRUA001`

December 12, 2021

Abstract

In this overview type work we discuss the multi-agent reinforcement learning problem. We start by giving an overview of the single-agent reinforcement learning case by discussing fundamental concepts and terminologies. We also discuss some fundamental deep reinforcement learning algorithms. Moving on from the single-agent case, we discuss the multi-agent case. We discuss the theoretical background of how the multi-agent reinforcement learning problem may be formulated and the core challenges related to multi-agent reinforcement learning research. We then discuss some deep multi-agent reinforcement learning algorithms. Finally, we conclude this work by giving our results from implementing various multi-agent reinforcement learning algorithms in the cooperative setting.

1 Introduction

At the core of reinforcement learning (RL) lies the notion of an agent learning optimal behaviour through experience obtained form interacting with an environment through trial and error. The agent observes either a full or partial environment state, takes an action and receives a numerical reward from the environment after which the environment enters into a new state. This goal-oriented nature of RL sets it apart from traditional machine learning which requires a learner to have labelled target data to garner knowledge from.

An early notable success was when an RL agent beat human players at Backgammon [1]. More recently however, AlphaGo, an RL agent developed by DeepMind [2] was able to beat a world champion at the game of Go which has more possible game states than there are atoms in the known universe. The success in the single agent RL case paved the way for RL to be applied in the multi-agent setting as well. Sensible areas for multi-agent reinforcement learning (MARL) applications involve smart energy grids and fully autonomous self-driving cars. In the power usage case each consumer on the grid (be it a household or factory) can be seen as an agent able to observe the power usage of its neighbours and optimize energy throughput so as to not overload the grid. A single all-knowing agent would struggle with optimizing the entire grid due to shear number of users at any given time. In the self-driving car case, cars are able to observe nearby neighbours and other obstacles and can then work together for optimal and safe traffic flow. Other applications could include robots working together in warehouses or models for studying population dynamics where each individual could be modelled by a unique agent. The possibilities for MARL are vast and truly exciting!

Akin to Go in the single-agent case, a popular test-bed for MARL algorithms is the StarCraft II real-time strategy game which has orders of magnitude more possible states than Go. StarCraft II, for example has 10^{26} possible choices for every move [3]. Success in StarCraft II was once believed to be a long-term goal for MARL but AlphaStar [4], was able to achieve grandmaster-level performance (better gameplay that 99.8% of human players) in StarCaraft II in 2019.

Another video game test-bed for MARL is Dota2. In Dota2 there are two teams of 5 players competing against each other. This gets modelled as a mixed game with agents on each team cooperating with each other while jointly competing against the opposing team. This makes Dota2 a significantly more challenging game to solve with MARL than StarCraft II since agents do not only have partial observability of the game state, (as with StarCraft II) but must learn to both cooperate and compete. The OpenAI Five [5] system was designed to do exactly this and was able to beat the Dota2 world champions in 2018.

Aside from Dota2, MARL has also been shown to have human-level performance on capture-the-flag and hide-and-seek type games [6][7]. What makes these studies particularly interesting is that the MARL agents solved these environments in ways that were consistent with human behaviour. Another interesting application of MARL relating to human behaviour could be to design fair tax policies as shown in [8].

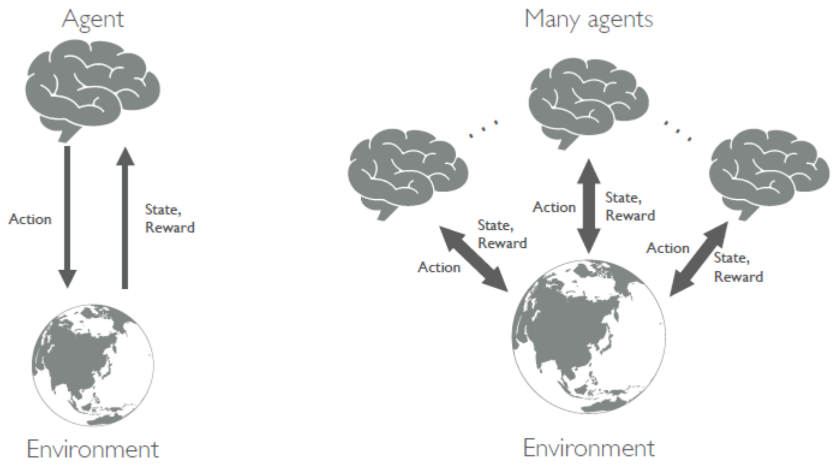


Figure 1: On the left we see a single agent learning through interacting with its environment as is the case in single-agent reinforcement learning. On the right, we see a collection of agents interacting with a single joint environment as is the case in multi-agent reinforcement learning. Figure taken from [3]

This work attempts to give an overview of multi-agent reinforcement learning and is structured as follows: In Part I we lay the foundation of fundamental RL concepts and illustrate how they are applied in the single-agent case. We then continue to discuss some fundamental single-agent deep RL algorithms. In Part II we extend the single-agent concepts to the multi-agent case, discuss various formulations of the multi-agent reinforcement learning (MARL) problem and give an overview of 3 multi-agent deep reinforcement learning algorithms. In Part III we implement selected deep MARL algorithms on cooperative tasks in the ma-gym [9] environment.

Part I

Single-Agent Reinforcement Learning

We begin by first giving a brief overview of the well known single-agent reinforcement learning case.

2 Reinforcement Learning Terminology

RL uses unique terminology in order to formulate and solve the problem of agent learning. We follow the pattern and work of Achiam [10] and Sutton and Barto [11] and, in a similar way, briefly illustrate the key concepts and terminology.

States and Observations

A state, denoted s, is a complete description of the environment an agent is interacting with while an observation o is a partial description of the state of the environment, which may omit some information. Whether or not the agent is able to observe the complete state of a given environment allows us to refer to the environment as either fully or partially observed. An example of a fully observable environment could be an agent in a fully lit room, while in the partially observable case the room might be dark and the agent can only see parts of the environment illuminated by a flash light of sorts. It should be noted at this point that in general, it is convention in RL to refer to an agent's state, s , although it would be more correct to refer to an agents observation, o. This occurs most frequently when talking about how an agent takes a particular action where we say that the action is conditioned on some state s, when in reality it might well be conditioned on some observation o due to the environment being partially observed. Formally we denote the set of all possible states as $\mathcal{S}$ with the environment state at a given time being $s_t \in \mathcal{S}$.

Action Spaces

It seems intuitive that different environments will have different possible actions and we refer the set of all possible actions in an environment can be referred to as the action space $\mathcal{A}(s)$. Action spaces fall largely into two classes, namely discrete or continuous action spaces. In a discrete action space there are only a finite number of moves available to an agent while in a continuous action space actions are vectors will real-valued entries. An example of a discrete actions space would be an Atari game where actions can take on only discrete values like up, down, left and right and an example of a continuous actions space could be some racing game where the agent has to control the degrees that the steering wheel is turned and the amount of force that is applied to the accelerator. Formally we have that the action taken at a given time can be given as $a_t \in \mathcal{A}(s)$.

Markov Decision Processes

Before mentioning Markov decision processes, we first mention the Markov property which can be given as

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_t, s_{t+1}, \ldots, s_0) \tag{1}$$

and means that the future dynamics of a system from any state s_t depend only on that current state. This implies that every observable state is self-contained to describe the future of the system. For this, the Markov property also requires the states of a system to be distinguishable from each other and unique. If this is the case, we require only one state to model the future dynamics of a system and the whole history, or a certain subset thereof. This is illustrated in equation (1) in that the probability of transitioning to state s_{t+1} given state s_t is equivalent to the probability of transitioning to state s_{t+1} given all previous states.

When an environment satisfies the Markov property, the RL problem can be formulated as a Markov Decision Process (MDP) expressible as a 6-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \rho_o \rangle$, with:

- $\mathcal{S}$ is the state space;
- $\mathcal{A}$ is the action space;
- $\mathcal{P}$: $\mathcal{S} \times \mathcal{A} \to P(\mathcal{S})$ is the transition probability function with $\mathcal{P}(s'|s, a)$ denoting the probability of transitioning to state $s' \in \mathcal{S}$ from state $s \in \mathcal{S}$ given the action $a \in \mathcal{A}$;
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward with value r which is the reward received by a agent for a transition from the state-action pair (s, a) to the state s';
- $\gamma \in [0, 1]$ is the discounting parameter that is used to compensate for the effect of instantaneous and future rewards
- ρ_0 is the starting state distribution

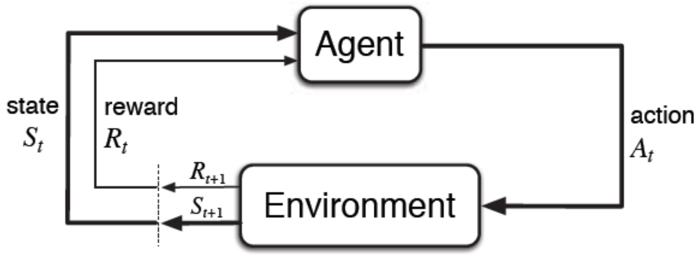


Figure 2: The Agent-Environment Interaction in a Markov Decision Process, Figure taken from [11]

It should be noted here that we have actually mentioned the classical-infinite horizon discounted MDP setting and that there are other standard formulations of MDPs such as the finite-horizon episodic setting and the time-average-reward setting [12][11].

Partially Observable Markov Decision Processes (POMDPs)

MDPs rely strongly on the Markov property which implies that agents have access to all the information in a given state and that the history of a state does not impact its future as we previously discussed. However, this is not always the case. In cases where an agent can only observe the partial state of an environment, we require a new model to formulate the RL problem.

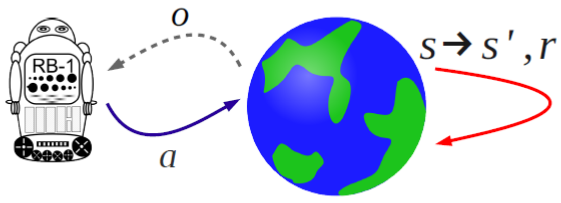


Figure 3: Figure illustrating a POMDP. Here the agent receives an observation $o \sim \mathcal{O}(\cdot|s,a)$ from the envirnoment instead of the next state s'. Figure taken from [13]

POMDPs are useful in exactly such cases where environment produce state uncertainty for the agent. POMDPs extend the normal MDP formulation by incorporating observations and their probability of occurring conditioned on the state of the environment. It is for this reason we say that, in a POMDP, an agent maintains a belief over states. This means that the agent can use the history of the observations to estimate the probability of given its current observation each state and use this knowledge to decide on an action to take. Formally, we say that the belief of an agent b_t is a probability distribution over states, such that

$$\forall_{s_t} \quad b_t(s_t) \equiv \mathrm{P}(s_t|o_t, a_{t-1}, o_{t-1}, \ldots, a_1, 0_1, a_0) \tag{2}$$

Here then, a single agent in a partially observable environment can have a policy as a series of mappings, or decsion rules from a series of beliefs to actions. While actions are being executed, an agent can update its current belief incrementally using Bayes rule. The update belief when an action a_t is taken and an observation o_{t+1} is recieved can then be given as

$$\forall_{s_{t+1}} \quad b_{t+1}(s_{t+1}) = \frac{1}{\mathrm{P}(o_{t+1}|b_t, a_t)} \sum_{s_t} \mathrm{P}(s_{t+1}, o_{t+1}|s_t, a_t) \tag{3}$$

Here $\mathrm{P}(o_{t+1}|b_t, a_t)$ is a normalization constant and $\mathrm{P}(s_{t+1}, o_{t+1}|s_t, a_t)$ is the probability that the POMDP model gives for receiving the particular new state s_{t+1} and resulting observation o_{t+1} while assuming the previous state was s_t.

One feature of POMDPs is that agents are able to reason over a belief and what the best action might be given that belief. This allows agents to reason about information gathering such that, when beneficial, agents may select actions that provide information about the state [13].

Policies

A policy is a mapping which maps states to actions, it can be deterministic in which case it is denoted as

$$a_t = \mu_\theta(s_t) \tag{4}$$

or a policy can be stochastic in which case it is denoted as

$$a_t \sim \pi_\theta(.|s_t) \tag{5}$$

The notation in the stochastic policy means that some action is sampled by the agent from the probability distribution $\pi_\theta(.|s_t)$. In both policies, θ denotes policy parameters, which are used in deep RL and denote a particular set of parameters like the weights and biases of a neural network. Two common classes of stochastic policies are categorical policies and diagonal Gaussian policies where the former can be used in discrete action spaces and the later is used in continuous action spaces. What sets deterministic polices apart is that such policies will return a specific action given an input.

Trajectories

And agent interacting with an environment gives rise to a trajectory, τ, which is a sequence of states, actions and rewards and can be given as:

$$\tau = (s_0, a_0, r_1, s_1, a_1, \dots) \tag{6}$$

It should be noted here that we denote the reward under the action at timestep t to be given at the next time step $t+1$. The first state, s_0, is sampled from the start-state distribution which can be denoted as $s_0 \sim \rho_0(.)$. Actions are taken by an agent under its current policy and state transitions dictate how an environment changes from a state at time t to time $t+1$ given a current action a_t. State transitions can be represented either by deterministic functions where $s_{t+1} = f(s_t, a_t)$ or as stochastic probability distributions where $s_{t+1} \sim P(.|s_t, a_t)$.

Reward and Return

As mentioned previously, after every action taken by the agent, it receives a real valued reward $r_{t+1} \in \mathcal{R} \subset \mathbb{R}$. This reward can be formally given as the output of a reward function R which depends on the current state and action and the future state and is denoted by

$$r_{t+1} = R(s_t, a_t, s_{t+1}) \tag{7}$$

The agent's goal is to maximize the cumulative reward over some trajectory, which we refer to as the agents return and is denoted by $R(\tau)$. One type of return is the finite-horizon undiscounted return which is the sum over all rewards in a trajectory over some fixed time

$$R(\tau) = \sum_{t=0}^{T} r_t \tag{8}$$

By introducing a discount factor, $\gamma \in (0, 1)$, we can also compute the so-called discounted return over an infinite horizon as

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{9}$$

This discounting factor serves two purposes in that it incentivises the agent to be more interested in more recent rewards and it also ensures that the returns series will be convergent when we are dealing with environments with no formal end state.

The Reinforcement Learning Problem

The overarching goal in reinforcement learning is for an agent to select a policy such that acting according to it would lead to the greatest expected return for a given trajectory. In order to discuss expected returns, we must first discuss probability distributions over trajectories. In the case where both the environment and policy are stochastic, the probability of a T-step trajectory can be given as

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{10}$$

The expected return for this trajectory follows directly from the notion of expectation values in statistics can be given as:

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)] \tag{11}$$

The goal is then to find an optimal policy, π^*, which maximizes equation (11). Formally we can express this as

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\ J(\pi) \tag{12}$$

Value Functions

In RL, it is useful to know about the value of a given state or a particular state-action pair. This value is defined as the expected return when starting in a particular state or with a particular state-action pair and always act according to a given policy.

Four key functions are:

1. The On-Policy Value Function, $V^{\pi}(s)$, which gives the expected return for starting in state s and

always action according to policy π:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] \tag{13}$$

2. The On-Policy Action-Value Function, $Q^{\pi}(s, a)$, which gives the expected return for starting in state s, followed by taking an arbitrary action a and then always acting according to particular policy, π:

$$Q^{\pi}(s, a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] \tag{14}$$

3. The Optimal Value function, $V^*(s)$, denoting the the expected return for starting in a state s and always acting according to the optimal policy and a given environment:

$$V^*(s) = \max_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] \tag{15}$$

4. The Optimal Action-Value Function, $Q^*(s)$, denoting the expected return for starting in a state s, taking an arbitrary action a and then always acting according to the optimal policy in a given environment:

$$Q^*(s) = \max_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] \tag{16}$$

Bellman Equations

The Bellman equations can be thought of as self-consistency equations that are obeyed by the four value functions previously discussed. Bellman equations give the value of a particular starting state or state-action pair as the expected value of the sum of the reward from the starting point and the value of the next state or state-action pair. The Bellman equations for the on-policy value functions are

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi \\ s'\sim P}}[r(s, a) + \gamma V^{\pi}(s')] \tag{17}$$

$$Q^{\pi}(s, a) = \mathop{\mathbb{E}}_{s'\sim P}\{r(s, a) + \gamma \mathop{\mathbb{E}}_{a'\sim\pi}[Q^{\pi}(s', a')]\} \tag{18}$$

Here $s' \sim P$ is shorthand for $s' \sim P(\cdot|s, a)$, implying that the next state s' is sampled from the transition rules of a given environment. Similarly, $a \sim \pi$ is shorthand for $a \sim \pi(\cdot|s)$ and $a' \sim \pi$ is shorthand for $a' \sim \pi(\cdot|s')$.

The Bellman equations for the optimal value functions can be given as:

$$V^*(s) = \max_{a} \mathop{\mathbb{E}}_{s'\sim P}[r(s, a) + \gamma V^*(s')] \tag{19}$$

$$Q^*(s, a) = \mathop{\mathbb{E}}_{s'\sim P}[r(s, a) + \gamma \max_{a'}[Q^*(s', a')] \tag{20}$$

An important part of the Bellman equations for the optimal value functions is the maximum over the actions. What this implies is that, when an agent has to choose an action, it has to choose the action that will lead to highest possible value in order to act optimally. Equations (19) and (20) are also referred to as the Bellman optimality equations.

Exploration vs. Exploitation

Through trial and error acting in an environment an agent can start to determine the value of certain state-action pairs and subsequently will determine what it deems to be the best action to take given a certain environment state. Choosing the action an agent deems to be optimal in a particular state is referred to as acting greedily or as exploiting current knowledge of the environment. Suppose, however, that a better action in a particular state exists but the agent is simply not aware of that action since it has never taken it before due to only acting greedily. To overcome falling into such a trap, agents must be encouraged to take chances by taking a new action in a particular state which it has not taken before. This process is known as exploration and is a popular research topic in and of itself. Finding the balance between exploiting the best known option while simultaneously exploring a sufficient number of options is know as the explore-exploit dilemma in RL. A very naive exploration strategy, used largely in applications, is so-called epsilon-greedy exploration. Under this exploration approach an exploration parameter $\epsilon \in (0, 1)$ is defined and in any given state an agent will take a random exploratory step with probability ϵ and act greedily with probability $(1 - \epsilon)$. As training progresses this exploration parameter is decreased gradually such that agents start exploiting states later on in the training procedure when it is assumed that they have explored enough states in the environment to have been able to learn from. This procedure is also sometimes referred to as the epsilon decreasing method.

3 Types of Reinforcement Learning Algorithms

RL algorithms can be broadly divided in to model-based and model-free algorithms. In the case of model-based algorithms the agent has access to, or strives to learn, a full model of the whole environment. This model is a function that predicts future state transitions and rewards. Whereas model-free algorithms forego this process altogether. Furthermore, model-free algorithms may be divided into two sub-classes which are value-based and policy-based methods.

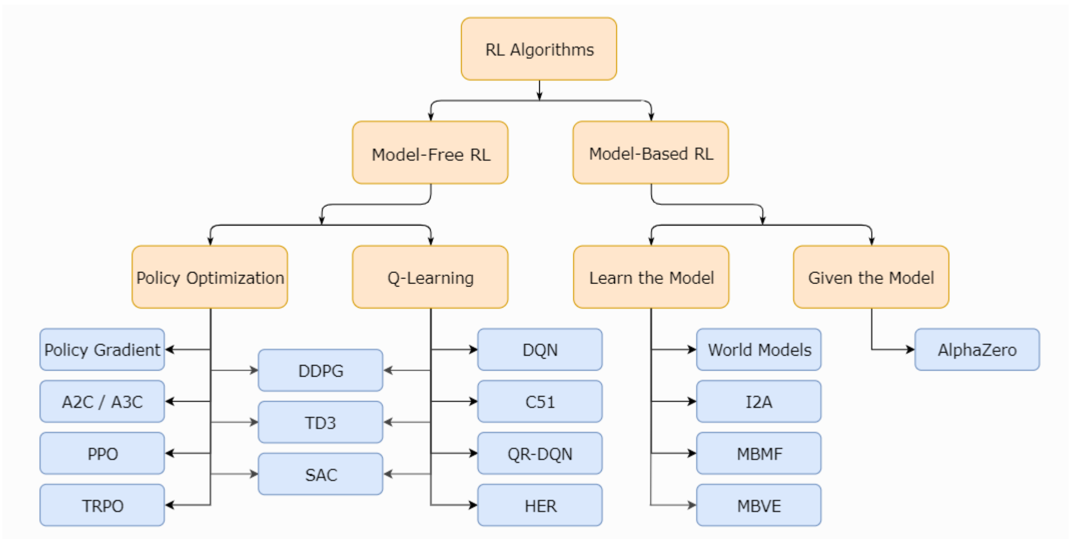


Figure 4: Partial Summary of the Deep RL Algorithm Landscape with some Example Algorithms. Figure taken from [10]

3.1 Model-Based and Model-Free Learning

In this overview, we focus primarily on model-free RL. What this means is that an implicit model of the environment is not provided to agents, nor do agents attempt to learn such a model. We wish to mention to the reader that such implementation do however exist. A well know algorithm, Dyna [11], learns a model from experience in an environment. These models, whether learned or provided, can be used by agents to 'plan' next steps in an environment before taking them. A modern example of a deep RL algorithm that makes use of model-based methods is AlphaGo introduced by DeepMind [2]. In fact, AlphaGo makes use of a combination of both model-based and model-free methods with its model-based component being Monte Carlo Tree Search. Another intuitive way of thinking about the difference between model-based and model-free learning is in terms of human human reasoning in that humans can think 'fast' and 'slow'. For example, we do not pay much thought to simple daily activities like walking or talking implying 'fast' thinking. But a trip to another town or writing a computer code requires us to reason carefully about what the best future steps to take are implying 'slow' thinking [14].

3.2 Value-Based Methods

For a given MDP with finite states and actions, there exists at least one deterministic stationary optimal policy [11]. Value-based methods try to obtain the optimal Q-function, Q^* as given in (19). Once Q^* has been found, an optimal policy can be derived from the Q-function by taking greedy actions according to it, such that $\pi^* = \underset{a}{\text{argmax}} Q^*(s, a)$.

3.3 Monte-Carlo Methods

Monte-Carlo (MC) methods estimate the value function by repeatedly generating episodes and keeping track of the average return at each state or at each state-action pair. The state-value function can therefore be given as

$$V_\pi^{MC}(s) = \lim_{i\to\infty} \mathbb{E}[r_i(s_t)|s_t = s, \pi] \tag{21}$$

where $r_i(s_t)$ denotes the return at state s_t for the i^{th} episode. Similarly, the state-action value function can be given as

$$Q_\pi^{MC}(s) = \lim_{i\to\infty} \mathbb{E}[r_i(s_t, a_t)|s_t = s, a_t = a, \pi] \tag{22}$$

In both (21) and (22) π denotes the policy under which episodes are generated. MC methods make two core assumptions to ensure convergence. Firstly they assume that the number of episodes is large and secondly that every state and every action will be visited a significant number of times. To encourage this exploration of the state and action space, an ϵ-greedy strategy in policy improvement is employed.

Generally, MC algorithms can be dived into on-policy and off-policy methods. In the case of on-policy methods, the same policy π is used for both evaluations and exploration, implying that the policy must be stochastic. Off-policy methods, on the other hand, use a different policy $\pi' \neq \pi$ to generate the episodes which, in turn, allow π to be deterministic.

3.4 Temporal-Difference Methods

Temporal-difference (TD) methods also learn from experience like MC methods, with one key difference in that TD methods do not wait until the end of an episode to make an update. TD methods make updates at every step during the episode by leveraging the 1-step Bellman equation (17). TD learning can the be applied to learning the sate value function as

$$V^i(s_t) \leftarrow \alpha V^{i-1}(s_t) + (1-\alpha)[r_{t+1} + \gamma V^{i-1}(s_{t+1})] \tag{23}$$

Where here $\alpha \in (0,1)$ is a learning rate, or step size, parameter. TD learning makes use of the previous estimated value functions to update the current value function. This technique is known as bootstrapping and, generally, methods that bootstrap learn faster than methods that don't [11]. Similarly to MC methods, TD methods can also be divided into on-policy and off-policy methods. An example of an on-policy TD control algorithm would be SARSA [11], which estimates the state-action value function as

$$Q^i(s_t,a_t) \leftarrow \alpha Q^{i-1}(s_t,a_t) + (1-\alpha)[r_{t+1} + \gamma Q^{i-1}(s_{t+1},a_{t+1})]. \tag{24}$$

Q-learning is an exmaple of an off-policy TD method which uses the Bellman optimality equation (20) to perform the update as

$$Q^i(s_t,a_t) \leftarrow \alpha Q^{i-1}(s_t,a_t) + (1-\alpha)[r_{t+1} + \overbrace{\gamma \max_{a'_{t+1}} Q^{i-1}(s_{t+1},a'_{t+1})}^{\text{Deterministic Sub-Policy}}]. \tag{25}$$

Here the max operator in the update rule substitutes for a deterministic sub-policy showing how Q-learning is an off-policy method.

3.5 Policy-Based Methods

In contrast to value-based methods which use an optimal value-function to deduce an optimal policy, policy optimization algorithms strive to learn the optimal policy directly. This is generally accomplished by a direct search over the policy space for an optimal policy π^*. This optimal policy can be parameterised as $\pi^* \approx \pi_\theta(\cdot|s)$ and then the parameters θ can be updated in the direction which maximizes the cumulative reward such that $\theta \leftarrow \theta + \alpha\nabla_\theta V^{\theta_\pi}(s)$. This class of learning methods is also refered to as policy-gradient methods. A clear problem here is that the gradient will depend on the unknown effects of policy changes on the state distribution. This problem was famously addressed through the derivation of the policy gradient theorem [11], which gives analytical solutions that do not involve the state distribution:

$$\nabla_\theta V^{\pi_\theta}(s) = \mathbb{E}_{s\sim\mu^{\pi_\theta}(\cdot),a\sim\pi_\theta(\cdot|s)}[\nabla_\theta \log \pi_\theta(a|s) \cdot Q^{\pi_\theta}(s,a)] \tag{26}$$

4 Deep Reinforcement Learning

We have mentioned that some functions need to be paramerised without further elaboration. This is usually done by using some function approximator to approximate the functions that agnets are trying to optimize. This is actually a highly non-trivial task. Recent advancements in deep neural network architectures have led to a modern approach to RL, which gained recent popularity again with the release of the paper by Mnih et al. [15]. This approach is to use deep neural networks as function approximators in RL and this combination of reinforcement learning with deep learning architectures has become known as Deep Reinforcement Learning.

A general challenge with deep reinforcement learning is that we do not have a target dataset to use during training like in the usual supervised machine learning case. A way of circumventing this problem in Deep RL is to make use of the Bellman optimality equations as targets. We will illustrate this later on when we discuss deep RL algorithms.

4.1 The Deadly Triad

Before we discuss some canonical deep reinforcement learning algorithms for the single-agent case, we must mention the deadly triad, which is a known issue in deep RL [11]. The deadly triad states that the learning of an optimal value function may become unstable or even diverge when we combine bootstrapping, off-policy learning and function approximation. When only two of these three conditions are present in a deep RL algorithm, the issue of instability can be avoided.

5 SARL Algorithms

Although this is not even nearly a comprehensive overview, we now discuss some very popular Deep RL algorithms. The recent advancements in Deep RL has lead to a plethora of algorithms solving various problems, but we aim to illustrate some of the most fundamental model-free deep RL algorithms.

5.1 Deep Q-Network

We cannot discuss deep RL without mentioning Deep Q-Networks, which were shown to have superhuman performance on Atari games in 2015. This success arguably put RL back into the mainstream research spotlight and stirred great interest in the field. A Deep Q-Network (DQN) is a model-free, off-policy, value-based algorithm that was first introduced by Mnih et al. [15] in 2013 and further expanded upon in 2015 [16]. This algorithm tries to learn an optimal action-value function, $Q^*(s, a)$, by approximating it using an artificial neural network which takes a given environment state as an input and returns the q-values for each action. We define this function approximator as $Q(s, a; \theta)$ where $\theta \in \mathbb{R}^m$ is a set of neural network parameters.

One thing to be aware of is that DQN is applicable to environments with continuous state spaces, but requires actions spaces that are discrete. This is because the use of a neural network frees the model from having to store the value for each state-action pair in some form of table in machine memory.

In the usual supervised deep learning case, we would want to optimize the weights of the neural network such that the network approximates some set of target values as accurately as possible. The model accuracy is then usually measured via some objective, or loss, function. Since these target values are not known in the RL setting, target values are computed by making use of the Bellman equations. Since

DQN is the Deep learning extension of Q-Learning, recall the update rule given as

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t)).$$

From this update rule, the following Bellman target can be inferred

$$y_i = r_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a)$$

Having a target defined, the following loss function is then defined

$$L(\theta) = \mathbb{E}\Bigg[(y_i - Q(s,a;\theta))^2\Bigg] \tag{27}$$

which we refer to as the Bellman Squared Error (BSE) Loss since it is derived from the Bellman optimality equation.

To overcome instability issues due to the deadly triad, the classic DQN algorithm employs two techniques, which are now commonly used in many Deep RL applications. The first technique is to use a replay buffer where all episode steps, or state transition tuples, $e_i = (s_t, a_t, r_{t+1}, s_t + 1)$ are stored in a replay memory buffer $\mathcal{D}_t = \{e_1, \ldots, e_t\}$. The replay buffer is also usually set to have some fixed length N with the most recent memories getting added to one end as the oldest memories are removed from the other end. While the DQN agent is trained, a random sample, or mini batch, of a given size is drawn from $\mathcal{D}_t$. This sampling from the experience replay improves data efficiency, removes correlations in the observed sequences and smooths over the changes in the data distribution ultimately leading to stabilisation in the training. The second technique is to make use of a target-network $\hat{Q}(s,a;\theta^{(k-1)})$ that lags slightly behind the online network that is being trained. This network is initialized with the same weights as the online network, but every C steps, the target network is set equal to the current online network. Using target networks makes training more stable since the parameter updates will affect only the network being trained and not the target as well. If this was not the case, an agent would be trying to learn by chasing a moving target.

Finally, DQN makes use of an ϵ-greedy policy to facilitate exploration. The algorithm can be summarized by the following pseudocode:

Algorithm 1 Deep Q-Learning with Experience Replay

```
Input Parameters: $\epsilon, \gamma, C, N, b$
  Initialize Replay buffer $\mathcal{D}$ of Length $N$ with $N$ random state-transition tuples $(s_t, a_t, r_{t+1}, s_{t+1})$
  Initialize action-value function network $Q(s, a)$ with random weights $\theta$
  Initialize target action-value function network $\hat{Q}(s, a)$ with random weights $\theta^- = \theta$
  for episode = 1, $M$ do
      Initialize a state $s_0$
      for t = 1, T do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \text{argmax}_a Q(s, a; \theta)$
          Execute the action $a_t$ and observe $r_{t+1}, s_{t+1}$
          Store current state-transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of state-transitions $B$ of size $b$, $(s_j, a_j, r_{j+1}, s_{j+1})$, from $\mathcal{D}$
          for each tuple in B do
              $y_j = \begin{cases} r_{j+1} & \text{if } s_{j+1} \text{ is a terminal state;} \\ r_{j+1} + \gamma \text{max}_{a'} \hat{Q}(s_{j+1}, a'; \theta^{(k-1)}) & \text{otherwise} \end{cases}$
              Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to the network parameters
  $\theta$
          end for
          Every $C$ steps set $\hat{Q} = Q$
      end for
  end for
```

5.2 REINFORCE

REINFORCE (Monte Carlo Policy Gradient) is a policy-gradient method. The first difference between REINFORCE and the value-based method previously defined is that we now define a reward function (as opposed to a loss function) which we want to maximize.

Following from equation (11) we define our reward function as the expected return under some policy π_θ parameterised by θ as

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta}[R(\tau)] = \int_\tau P_\theta(\tau) R(\tau)$$

where $P_\theta(\tau)$ is shorthand for $P(\tau|\pi_\theta)$.

The goal is then to optimize the policy by gradient ascent with the parameters being adjusted as

$$\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_k} \tag{28}$$

From equation (28) we can note that we need an expression for $\nabla_\theta J(\pi_\theta)$. This expression follows from the policy-gradient theorem [11] and is derived as follows:

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \int_\tau P_\theta(\tau)R(\tau) \\
&= \int_\tau \nabla_\theta P_\theta(\tau)R(\tau) \\
&= \int_\tau \frac{\nabla_\theta P_\theta(\tau)}{P_\theta(\tau)} P_\theta(\tau)R(\tau) \\
&= \int_\tau \nabla_\theta \log P_\theta(\tau) P_\theta(\tau) R(\tau) \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\Bigg[\nabla_\theta \log P_\theta(\tau)R(\tau)\Bigg]
\end{aligned} \tag{29}
$$

It is still no possible to compute the gradient of the probability of a trajectory. But we can expand the trajectory probability in the last line of (29) as

$$
\begin{aligned}
P_\theta(\tau) = P(\tau|\theta) &= \rho_0(s_0)\prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t) \\
\iff \nabla_\theta P(\tau|\theta) &= P(\tau|\theta)\nabla_\theta \log P(\tau|\theta) \\
\Rightarrow \nabla_\theta \log P(\tau|\theta) &= \nabla_\theta \rho_0(s_0) + \sum_{t=0}^{T}\Bigg[\nabla_\theta \log P(s_{t+1}|s_t, a_t) + \nabla_\theta \log \pi_\theta(a_t|s_t)\Bigg] \\
\iff \nabla_\theta \log P(\tau|\theta) &= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)
\end{aligned} \tag{30}
$$

Using the final result from (30) we can the see from (29) that

$$
\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\Bigg[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\Bigg] \tag{31}
$$

Since (31) is now just an expectation, it can be estimated using a sample mean. We can collect a set of trajectories $\mathcal{D} = \{\tau_i\}_{i=1,\dots,N}$ where, for each trajectory an agent act in the environment under a policy π_θ [17]. We can then estimate the policy gradient as

$$
\hat{g} = \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)
$$

It is from this estimation of the gradient where REINFORCE gets its name as the Monte-Carlo Policy Gradient method. We allow an agent to interact with an environment for a set amount of fixed length trajectories, compute the estimate to the gradient and perform and update to the parameterised policy. The REINFORCE algorithm can be summarized in the following pseudocode:

Algorithm 2 REINFORCE (Monte Carlo Policy Gradient)

```
Input Parameters:
  Differentiable parameterized policy $\pi_\theta$,
  Discounting factor $\gamma$,
  Learning Rate $\alpha$
  for Each Episode do
    Generate an episode $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$, following $\pi(\cdot|\cdot, \theta)$
    for Each step of the episode $t = 0, 1, \ldots, T-1$ do
      $R(\tau) \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
      $\theta \leftarrow \theta + \alpha \gamma^t R(\tau) \nabla \log \pi_\theta(a_t|s_t)$
    end for
  end for
```

5.3 Actor-Critic

So far, we have discussed a value-based and policy-based algorithm, but there exist a very important class of Deep RL algorithms that effectively combines both approaches. These methods are called actor-critic type algorithms where an optimal policy and value function are learnt simultaneously.

Before discussing the actor-critic architecture, we first mention a weakness in the REINFORCE algorithm. Recall that the policy gradient is computed as

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathbb{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right]$$

Since REINFORCE updates the policy parameters through Monte Carlo updates, it introduces very high variance in the log probabilities as well as the cumulative reward values. This is because each trajectory during training can differ greatly from the next. This high variance in log probabilities and cumulative reward values can lead to noisy gradients and cause unstable learning. Another side-effect could be that the policy distributions becomes skewed towards some sub-optimal direction.

One way of improving the stability and to reduce the variance in the gradients is to subtract some baseline $b(s_t)$ from the cumulative rewards such that

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathbb{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) (R(\tau) - b(s_t)) \right] \tag{32}$$

This baseline function can be any function, even a random variable, as long as it doesn't depend on the actions taken a. In such a case equation (32) remains valid since

$$\sum_a b(s) \nabla_\theta \pi_\theta(a|s) = b(s) \nabla_\theta \sum_a \pi_\theta(a|s) = b(s) \nabla_\theta 1 = 0$$

In fact, it has been shown that [18], in general, the policy gradient may be expressed as

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{33}$$

where Ψ_t may be any of the following:

$$\begin{array}{ll} 1.\ \sum_{t=0}^{\infty} r_t\text{: total reward of the trajectory} & 4.\ Q^\pi(s_t, a_t)\text{: state-action value function.} \\ 2.\ \sum_{t'=t}^{\infty} r_{t'}\text{: reward following an action} & 5.\ A^\pi(s_t, a_t)\text{: advantage function.} \\ 3.\ \sum_{t'=t}^{\infty} r_{t'} - b(s_t)\text{: baselined version of 2} & 6.\ r_t + V^\pi(s_{t+1}) - V^\pi(s_t)\text{: TD residual.} \end{array} \tag{34}$$

To illustrate how case 4. in (34) was derived, one may consider a decomposed version of equation (31)

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right] \\ &= \mathop{\mathbb{E}}_{s_0,a_0,\ldots,s_t,a_t}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \mathop{\mathbb{E}}_{r_{t+1},s_{t+1},\ldots,r_T,s_T}\left[R(\tau)\right] \end{aligned} \tag{35}$$

Where the second part of the decomposition in (35) is just the definition of $Q(s_t, a_t)$. The result then follows by substituting Ψ_t with $Q^\pi(s_t, a_t)$ to give

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) Q^w(s_t, a_t)\right] \tag{36}$$

Where now, the Q-value is a function parameterized by w similarly to what was done in DQN.

This gives rise to the notion of actor-critic methods. Here the critic estimates the value function by using the action-value or the state-value. Intuitively the critic gives an indication of how good a given action taken under a policy is. The actor is some policy which dictates how actions are taken by an agents in a particular environment. The actor in this case is a stochastic policy distribution over the actions in the action space of an environment and both the actor and critic networks are approximated using neural networks. The type of actor-critic algorithm depends on the chosen function for Ψ_t and it is for this reason that using the policy gradient (36) gives rise to the so-called Q Actor-Critic algorithm. This algorithm can be summarized in the following pseudocode:

Algorithm 3 Q Actor Critic

```
Input Parameters:
  Differentiable parameterized actor network $\pi_\theta$,
  Differentiable parameterized critic network $Q_w$,
  Discounting factor $\gamma$,
  Learning Rates $\alpha_\theta$, $\alpha_w$
  Initialize $s_0$ and sample $a_0 \sim \pi_\theta(a|s_0)$
  for t = 1, ..., T do
    Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s'|s,a)$
    Sample next action $a' \sim \pi_\theta(a'|s')$
    Update the policy parameters:
        $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a)\nabla_\theta \log \pi_\theta(s,a)$
    Compute the correction for action-value at time $t$:
        $\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$
    Update the parameters of the Q function:
        $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$
    Set $a \leftarrow a'$, and $s \leftarrow s'$
  end for
```

5.4 Deep Deterministic Policy Gradient

Deterministic Policy Gradients algorithms were first introduced by Silver et al. [19] and adapted to the deep RL setting as the Deep Deterministic Policy Gradient by Lillicrap et al. [20]. DDPG is an off-policy, actor-critic algorithm that is optimized to work well on environments with continuous action spaces. It can be though of as an extension of DQN to the continuous actions space domain. This was a significant challenge and important accomplishment, since maximizing over all possible actions becomes a very computationally expensive task when the action space contains an infinite amount of actions, as in the case of continuous actions spaces. Since DDPG strives to extend DQN in this way, it borrows two techniques from DQN namely the experience replay buffer and the notion of target networks. One difference between DDPG, DQN and classic actor-critic type algorithms is that DDPG uses target networks for both the actor and critic networks, which we denote by a dash.

When optimizing the critic network, the familiar mean squared bellman error loss function is used

$$L(\theta) = \frac{1}{N}\left[y_t - Q(s_t, a_t|\theta^Q) \right] \tag{37}$$

where

$$y_t = r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'}) \tag{38}$$

A key difference between between DDPG and other policy gradient methods is that the learnt policy for the actor is deterministic implying that the actions suggested by the actor is fixed given some input state. In the case of this actor network, we still want to maximize the expected returns which we can denote as

$$J = \mathbb{E}[Q(s, \mu(s|\theta^{\mu})|\theta^{Q})] \tag{39}$$

Since we are interested in seeing how the value of a particular action changes as the actions change we want to compute the gradient of (39) with respect to the parameters of the actor network θ^{μ}. By the deterministic policy gradient theorem, this can be shown to be

$$\begin{aligned} \nabla_{\theta^{\mu}} J &\approx \nabla_a Q(s, a|\theta^{Q}) \nabla_{\theta^{\mu}} \mu(s|\theta^{m}u) \\ &\approx \frac{1}{N}\sum_i \left[\nabla_a Q(s, a|\theta^{Q})|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s|\theta^{m}u)|_{s=s_i} \right] \end{aligned} \tag{40}$$

A valid question to ask now is, since the policy μ is deterministic, how will the agent explore? This issue is addressed during training time be introducing some random noise $\mathcal{N}$ to the action suggested by the actor network as

$$a_t = \mu(s_t|\theta^{\mu}) + \mathcal{N} \tag{41}$$

Here, the authors in [20] suggest using an Ornstein-Uhlenbeck noise generating process, but it has later been shown that uncorrelated Gaussian noise is also sufficient for exploration.

A final difference between DQN and DDPG is the way in which the target networks are updated. Instead of performing a hard update of the target network parameters after a fixed amount of training iterations, DDPG performs small, or soft, updates to the actor and critic target network parameters at every training iteration as

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \end{aligned} \tag{42}$$

This process is know as polyak averaging and $\tau \ll 1$ is a small hyper parameter that is chosen. These small updates more closely resemble standard supervised learning practices where model weights are slowly adjusted towards an optimum leading to more stable training. The algorithm is summarized by the following pseudocode:

Algorithm 4 Deep Deterministic Policy Gradient

```
Input Parameters: $N, b, \tau$
  Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor network $\mu(s|\theta^\mu)$ with weights $\theta^Q$,$\theta^\mu$
  Initialize target networks $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$,$\theta^{\mu'} \leftarrow \theta^\mu$
  Initialize target action-value function network $\hat{Q}(s, a)$ with random weights $\theta^- = \theta$
  Initialize Replay buffer $\mathcal{D}$ of Length $N$ with $N$ random state-transition tuples $(s_t, a_t, r_{t+1}, s_{t+1})$
  for episode = 1, $M$ do
      Initialize a random process $\mathcal{N}$ for action exploration
      Receive initial observation $s_0$
      for t = 1, T do
          Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to current policy and exploration noise
          Execute action $a_t$ and observe reward $r_t$ and new state $s_{t+1}$
          Store current state-transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of state-transitions $B$ of size $b$, $(s_j, a_j, r_{j+1}, s_{j+1})$, from $\mathcal{D}$
          Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
          Update critic by minimizing the loss: $L = \frac{1}{b}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
          Update actor policy using the sampled policy gradient:
              $$\nabla_{\theta^\mu} J \approx \frac{1}{b}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s-s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$
          Update the target networks
              $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
              $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
      end for
  end for
```

Part II

Multi-Agent Reinforcement Learning

In the multi-agent case, similarly to the single-agent case, each individual agent is still trying to solve sequential reinforcement learning problem through trial-and-error. One key difference is that the evolution of the environment state and reward function is now determined by the joint actions of all agents in the environment. This implies that agents do not only need to interact with the environment, but also need to take other learning agents into account. A way to model a decision-making process that involves multiple agents is through a stochastic game, also know as a Markov game [3].

6 Game Theory as a Foundation for MARL

Game theory has proved immensely useful in MARL by providing not only a framework within which to define the MARL problem, analogous to the MDP in SARL, but also a mathematical structure for mathematically rigorous solution analysis [21]. It is for this reason that we briefly introduce some game theoretical concepts before moving on to the formulation of the MARL problem.

The core idea in game theory is the modeling of strategic interaction between a set of players as a game. This game is a mathematical object serving as a means to describe the consequences of these interactions between player strategies in terms of individual payoffs. Games can be largely categorized as either being extensive form or normal form games. The main distinction being that extensive form games model games where players take turns to perform actions, whereas in normal form games all agent act independently but simultaneously [21].

6.1 Normal Form Games

Normal form games are commonly used as a means to test multi-agent earning approaches [21] and can be formally given by the a tuple $\langle N, \{\mathcal{A}^i\}_{i\in\{1,\ldots,N\}}, \{\mathcal{R}^i\}_{i\in\{1,\ldots,N\}}\rangle$. Here N denotes the number of participants, or players, in the game, $\mathcal{A}^i$ denotes the set of actions available to payer i and $\mathcal{R}^i$ denotes the individual reward function for player i. This reward functions stipulates what reward player i will receive for the joint action taken by all players. Normal form games can be represented by a so-called payoff matrix which, in the two player case, has rows which denote the actions of player 1 and columns that denote the action taken by player 2. Scaling up the number of players, requires a suitable upscaling of the dimensions of the matrix. The payoff matrix entries denote the payoff received by a given player given a particular set of actions.

In game theory, a strategy is denoted as $\sigma^i : \mathcal{A}^i \to [0, 1]$ and is an element of $\mu(\mathcal{A}^i)$ which is the set of probability distributions over the action set $\mathcal{A}^i$ of player i. A strategy is pure if $\sigma^i = 1$ for some action $a \in \mathcal{A}^i$ and 0 for all other actions, otherwise a strategy is referred to as mixed. The vector containing the strategies of all players is referred as the strategy profile σ and, in the case where all strategies in σ are pure, the strategy corresponds to the joint action.

Normal form games are also referred to as repeated games and have a a state space with $|\mathcal{S}| = 1$ implying that the game has only one state. Although this does not capture the full nature of the MARL problem, it does lay the foundation for important concepts and analysis [21].

6.2 Types of Games

The formulation of the reward function, as we will later discuss in terms of the MARL setting, determines the nature of a game. A game may be an identical payoff, or common interest, game which is when all players share the same reward function. In such games, players may strive to cooperate to achieve some maximum reward value. A way to frame games as purely competitive is by placing a constraint that all agent rewards must sum to 0. These types of games are referred to as zero-sum games. When there are no particular restrictions to a game, we refer to it as a general sum game. These games can have both competitive and cooperative elements to them.

	a1	a2
a1	(1,-1)	(-1,1)
a2	(-1,1)	(1,-1)

(a)

	a1	a2
a1	(5,5)	(0,10)
a2	(10,0)	**(1,1)**

(b)

	a1	a2
a1	**(5,5)**	(0,0)
a2	(0,0)	**(10,10)**

(c)

	a1	a2
a1	**(2,1)**	(0,0)
a2	(0,0)	**(1,2)**

(d)

Figure 5: Some Examples of 2-player, 2-action games: (a) Matching pennies, a zero-sum purely competitive game. (b) The prisoner's dilemma, a general sum game. (c) The coordination game, a common interest (identical payoff) game. (d) Battle of the sexes, a coordination game. Figure taken from [21]

In Figure 5, game (a) is an example of a strictly competitive, zero-sum game. Players select a side of a coin, heads or tails, to show to an opponent. If both players show the same face player 1 wins and receives 1 unit from player 2. When the players have chosen opposite faces for their coins, player 2 wins and receives 1 unit from player one. The notion of a zero-sum game is clearly demonstrated here as a win for one player equates to a loss for the other causing their joint rewards to sum to zero. Game (d) on the other hand illustrates a game where players need to cooperate. Something to note is that the players receive different rewards and player one prefers (a1, a1) whereas player 2 prefers (a2, a2). Although the joint reward is the same for both cases, players need to reason and come to a form of agreement on which preferred outcome to settle on.

6.3 Solution Concepts in Games

A naive way to reason about solving a game would be that all players should maximize their individual rewards at all costs, but clearly this is not feasible and could lead to a sub-optimal reward for the collective. Moreover in the competitive case this approach is not even possible. This is where game theory introduces the concept of a best response, which is when a player maximizes its payoff with respect to current strategies of all other players in a game. This implies that it is not possible for a player to improve its reward while the strategies of all other players remain the same. Formally, we let $\sigma = (\sigma^1, \ldots, \sigma^n)$ be a strategy profile and σ^{-i} denote the strategies of all players except for player i that has the strategy σ^k. The best response for player i, $\sigma^{i*} \in \mu(\mathcal{A}^i)$ should satisfy

$$\mathcal{R}^i(\sigma^{-i} \cup \sigma^{i*}) \geq \mathcal{R}^k(\sigma^{-i} \cup \sigma^{i'}) \forall \sigma^{i'} \in \mu(\mathcal{A}^i) \tag{43}$$

with $\sigma^{-i} \cup \sigma^{i'}$ denoting the strategy profile where all other players play the same strategy while player i plays some other non-optimal strategy $\sigma^{i'}$.

The Nash Equilibrium

A fundamental concept in game theory is the Nash Equilibrium (NE). In the NE, all players play using mutual best replies, which means that players are constantly playing according the best response relative to the strategies played by other players. It has been shown by Nash [22] that every normal form game has at least 1 NE.

Formally, we say that a strategy profile $\sigma = (\sigma^1, \ldots, \sigma^n)$ is a Nash Equilibrium if the strategy σ^i is the best response to the strategies of all other players σ^{-i} for every player i.

This implies that, when playing a NE, no player can improve their payoff by deviating from the equilibrium strategy profile. A useful way of thinking about the NE is like a fixed point of a dynamical system or like an optimum.

As previously mentioned normal games lack an important part of formulating the MARL problem, namely that the state of the environment is not dynamic. This means that, when using normal form games as a framework for the MARL problem we miss out on an important aspect of reinforcement learning, since the environment is no longer changing as a consequence of the actions taken by agents. A framework that is able to capture the dynamical nature of the environment is a Markov Game which we discuss in the next section.

7 Formulating the Multi-Agent Reinforcement Learning Problem

Unlike the single-agent case, which can get formulated in terms of an MDP only, there exist multiple frameworks within which to define the multi-agent reinforcement learning problem. We discuss some of these frame works which can be largely divided into two classes: The fully observable and partially observable setting.

7.1 Fully Observable Setting

In the fully observable setting, we assume that all agents have access to the full state space of the system. An example of this could be the Atari game of Pong where the two agents are aware of the entire game state, including each other's observations, at each time step.

Markov Games

A Markov, or stochastic, game can be interpreted as a multi-player extension of the Markov Decision Process discussed in Section 2. It is arguably the most natural extension of the MDP and subsequently is also denoted by a tuple $\langle N, \mathcal{S}, \{\mathcal{A}^i\}_{i \in \{1,\ldots,N\}}, \mathcal{P}, \{\mathcal{R}^i\}_{i \in \{1,\ldots,N\}}, \gamma \rangle$, where:

- N is the number of agents. Here $N = 1$ degenerates to the single-agent MDP case;
- $\mathcal{S}$ is the state spaces shared by all agents;
- $\mathcal{A}^i$ is the set of all actions taken by agent i. Here the full state space is denoted as $\mathcal{A} \equiv \mathcal{A}^1 \times \cdots \times \mathcal{A}^N$;
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to P(\mathcal{S})$ is the transition probability from state $s \in \mathcal{S}$ to state $s' \in \mathcal{S}$ given the agents' joint actions $\mathbf{a} \in \mathcal{A}$;
- $\mathcal{R}^i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function that returns a scalar value to the i^{th} agent for a transition from $(s, \mathbf{a})$ to s';
- $\gamma \in [0, 1]$ is the discount factor used to compensate for the effect of instantaneous and future rewards

We adopt the convention in this work to use the superscript notation $(\cdot^i, \cdot^{-i})$ to distinguish between and agent i and all other $N - 1$ agents in an environment respectively. The stochastic game (SG) extends the

MDP and allows simultaneous moves by agents in the decision making process. The framework works as follows: At a every time step t, the environment has some state $s_t \in \mathcal{S}$ and given this state s_t each agent takes an action a_t^i simultaneously with all other agents. The joint action by all agents $\mathbf{a}_t$ then causes the environment to transition to the next state $s_{t+1} \sim P(\cdot|s_t, \mathbf{a}_t)$ and to give an immediate reward $\mathcal{R}^i(s_t, \mathbf{a}_t, s_{t+1})$ to each agent. The goal of each agent, akin to the SARL MDP case, is to solve the SG by finding a a policy $\pi^i \in \Pi^i : \mathcal{S} \to P(\mathcal{A}^i)$ that leads to actions which maximizes the discounted cumulative reward which can be given as

$$V^{\pi^i,\pi^{-i}} = \mathrm{E}_{s_{t+1}\sim P(\cdot|s_t,a_t),a^{-i}\sim\pi^{-i}(\cdot|s_t)}\left[\sum_{t\geq 0}\gamma^t \mathcal{R}_t^i(s_t, \mathbf{a}_t, s_{t+1})|a_t^i \sim \pi^i(\cdot|s_t), s_0\right] \tag{44}$$

Here we note the clear difference between the single-agent and multi-agent case in that the optimal policy for an agent does not only depends on its own policy, but also on the policies of all other agents.

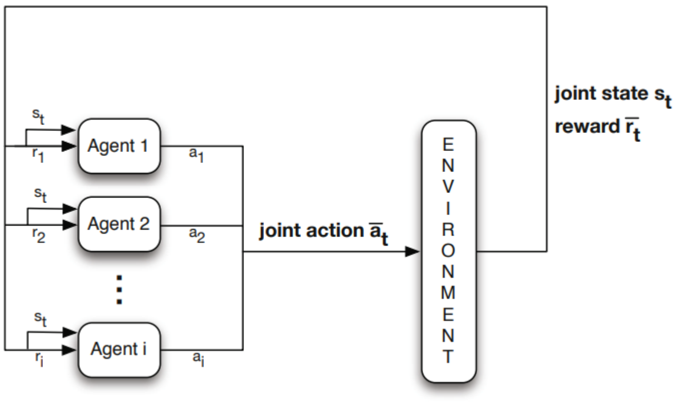


Figure 6: Agents taking a collective joint action in the same environment. Figure taken from [21]

7.2 The Partially Observable setting

In the partially observable setting agents are not able to observe the entire state space, similarly to what was mentioned about POMDPs in the single-agent case. Agents may have insights about the actions of their immediate neighbours or some subset of the state space. An example could be the game of Dota2, where multiple agents are competing against each other in teams, but the fog of war conceals certain parts of the map from an agent.

Partially Observable Stochastic Games

Partially observable stochastic games (POSGs) are an extension of SGs which are defined by the tuple $\langle N, \mathcal{S}, \{\mathcal{A}^i\}_{i\in\{1,\dots,N\}}, \mathcal{T}, \{\mathcal{R}^i\}_{i\in\{1,\dots,N\}}, \{\Omega^i\}_{i\in\{1,\dots,N\}}, \mathcal{O}, \gamma\rangle$. POSGs add two terms to the SG tuple, namely Ω and O. These denote the observation space for agent i and the observation transition probability function respectively. We will discuss these additions in detail in the Decentralized-POMDPs section.

While POSGs are one of the most general game classes, we wish to discuss a particular subclass of POSGs, the decentralized-POMDP. Here, all agents share the same reward and the scenario can be defined as follows:

Decentralized POMDPs

POMDPs give a way to treat state uncertainty, but are only able to consider single agents. We now wish to extend this notion to the multi-agent case. This can be done by formulating the problem as a decentralized POMDP (Dec-POMDP) which generalizes POMDPs to multiple agents enabling the modelling of teams of agents in stochastic, partially observable environments. Dec-POMDPs are particularly useful for modelling the cooperative case in partially observable settings [12].

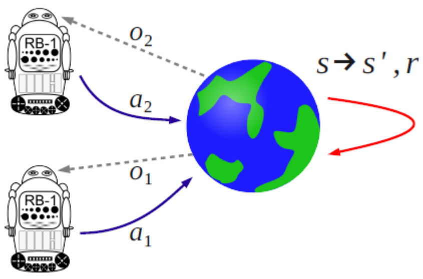


Figure 7: Figure illustrating a Dec-POMDP. Here each agent receives an observation $o_i \sim \mathcal{O}_i(\cdot|s, a)$ from the environment instead of the next state s'. Each agent then choses an action based on its own observation of the environment. Figure taken from [13]

Similarly to POSGs we can denote a Dec-POMDP as a tuple $\langle N, \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, \gamma \rangle$ where,

- N is the set of $n > 1$ agents;
- $\mathcal{S}$ is the state space;
- $\mathcal{A} = \times_{i \in N} A^{(i)}$ is the joint action space;
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow P(\mathcal{S})$ is the environment transition probability from state s to the next state s' given the joint action of the agents $\mathbf{a}$;
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the immediate reward function;
- $\Omega = \times_{i \in N} \Omega^{(i)}$ is the joint observation space;
- $\mathcal{O} : \mathcal{S} \times \mathcal{A} \rightarrow P(\Omega)$ is the joint observation probability
- $\gamma \in [0, 1]$ is the discounting factor

We can note that the Dec-POPMDP is a multi-agent extension of the POMDP model, similarly to how Markov Games extend MDPs. Considering the joint action space, $A^{(i)}$ is the action space available to agent i and can be different for each agent. At each timestep t, each agent takes an action $a_{i,t}$ leading to the joint action $\mathbf{a} = \langle a_{1,t}, \ldots, a_{N,t} \rangle$. The transition probability functions models how this joint action influences the environment. It is also important to note that, in a Dec-POMDP, agents do not know what actions are taken by each other. Similarly to the joint action space, Ω denotes the joint observation space, where agent i has observation space $\Omega^{(i)}$ available to it. Each agent receiving its own observation also gives rise to the joint observation at timestep t given as $\mathbf{o} = \langle o_{1,t}, \ldots, o_{N,t} \rangle$. Here $\mathcal{O}$ defines the probability of the joint observation as $P(\mathbf{o}|\mathbf{a}, s')$. At each timestep, the agents receive a joint reward $\mathbf{r}_t = \mathcal{R}(s_t, \mathbf{a}_t)$ and the reward function defines the objective of the agents. An important feature to note here is that the reward function only specifies the immediate reward, but the goal is to optimize the behaviour of the team of agents over longer times. This gives rise to the need for defining how the immediate rewards are combined into a single number. These various reward configurations are discussed later on. The goal of the agents, however, remains the same in that agents strive to optimize the long term expected returns by selecting an optimal joint policy. Different from the single agent case however, is that the policy, $\pi^{(i)}$ of agent i is a mapping from local observations to local actions.

We have only discussed two ways in which the multi-agent reinforcement learning problem may be phrased and have only discussed settings in which agents in the environment act simultaneously. It should be noted that other frameworks also exist. A notable example is to phrase the problem as an extensive-form game where agents act in a turn-based manner [12]. An example of an extensive-from game would be a poker game, where agents take turns to act. The multiple ways in which the MARL problem may be phrased is in fact one of the challenges in MARL in that it does not have a single general model that has broad applications like the MDP in the single-agent case.

8 MARL Reward Structures

As mentioned previously, the way in which the individual agent rewards are combined for a given environment, determines how agents will interact. There are three core reward structures in MARL:

8.1 Cooperative

In the fully cooperative setting, all agents receive the same reward $\mathcal{R} = \mathcal{R}^i = \cdots = \mathcal{R}^N$ for each state transition. This model is also referred to a multi-agent MDPs (MMDPs) or as Markov teams. Here agents are encouraged to work together such that the joint reward is maximized. This ensure that agents strive to improve the performance of weaker performing agents. This model formulation has the feature that the value function and Q-function are identical for all agents. This further implies that standard single-agent algorithms like Q-learning can be be applied in the case where all agents are coordinated as one decision maker. The optimal behaviour of agents in the system then can then be viewed as a Nash Equilibrium of a game. The only difference between the game theory and MARL case is that in the MARL case the Nash equilibrium is defined in terms of agent policies

$$V^i_{\pi^{i*},\pi^{-i*}}(s) \geq V^i_{\pi^i,\pi^{-i*}}(s), \quad \forall \pi^i \tag{45}$$

instead of strategy profiles as in the game theoretic case. Here π^{i*} denotes the best response policy to the optimal policies of all other agents π^{-i*}. This of course also implies the extistence of an optimal joint policy for all agents π^*.

Instead of only formulating the reward as the sum over all identical agent rewards, each agent may also receive a unique reward and the joint reward may be considered to be the team-average reward given as

$$\overline{\mathcal{R}}(s,a,s') \equiv \frac{1}{N}\sum_{i\in N}\mathcal{R}^i(s,a,s'),\ \text{for any } (s,a,s') \in \mathcal{S}\times\mathcal{A}\times\mathcal{S}.$$

When using the average reward model, the rewards of agents may also be kept private from each other which could introduce heterogeneity among agents. This heterogeneity between agents opens the possibility of decentralized MARL algorithms and also brings with it the need for some form of communication protocol between agents in a system [12].

8.2 Competitive

Akin to the competitive game theoretic setting, the rewards in the competitive MARL setting is formulated as a zero-sum game where

$$\sum_{i\in N}\mathcal{R}^i(s,a,s') = 0,\ \text{for any } (s,a,s').$$

This is able to capture the inherent opposing interest of agents as a win for one is a loss for the other. This can also be generalized to the n-player constant sum case where

$$\sum_{i\in N}\mathcal{R}^i(s,a,s') = c,\ \text{for any } (s,a,s').$$

One can readily note that this general case degenerates to the zero-sum case when either $c = 0$ or when the constant c is subtracted from the sum of all agent rewards. The simplest way to analyze such configurations analytically is via introducing the problem as a two player game.

8.3 Mixed

The mixed setting, which may also be referred to as the general-sum setting has no clear way to define the reward structure. There is also no true restriction on how these rewards need to be structured and the formulation largely depends on the task at hand. Agents in the mixed setting are all considered to be self-interested and the rewards of agents may conflict with other agents in the environment. An example of a mixed setting could be the the two-team competitive setting where two teams of sizes N_1 and N_2 respectively are competing with each other while cooperating withing teams. The reward functions would then be the set $\{\mathcal{R}^{1,1},\ldots,\mathcal{R}^{1,N_1},\mathcal{R}^{2,1}\ldots,\mathcal{R}^{2,N_2}\}$ and team members on the same team share similar objectives where

$$\mathcal{R}^1 = \sum_{i\in N_1}\mathcal{R}^{1,i}/N_1$$

and

$$\mathcal{R}^2 = \sum_{j\in N_2}\mathcal{R}^{2,j}/N_2$$

subject to the joint constraint that $\mathcal{R}^1 + \mathcal{R}^2 = 0$. Of course this can also be generalised to the case where are arbitrary many teams each consisting of an arbitrary number of agents.

9 Principle Challenges in Multi-Agent Reinforcement Learning

Aside from not having a broad model to frame the MARL problem, MARL also suffers from several other challenges with regards to theoretical analysis which we briefly discuss.

9.1 Non-Stationarity

A key challenge in MARL is that of non-stationarity which is caused by the fact that multiple agents are simultaneously learning in the same environment. In particular, the action taken by each agents affects not only its own reward, but the reward received by all other agents as well as the state transition of the environment. This means that, from an individual agent's perspective the environment appears to be changing, or is non-stationary. As previously mentioned, this then means that agents have to take other agents into account and come up with a way to adapt such that they find the best joint policy for a given environment. This causes a problem for using standard SARL algorithms in the multi-agent case since the fundamental assumption of states having the Markov property are no longer true which implies that convergence is no longer guaranteed. In this case the current state and individual agent rewards no longer depend only on the previous state and the action taken, but on potentially the history of actions taken not only by a particular agent, but by all other agents as well. The violation of the Markov property immediately makes it very challenging to use SARL mathematical tools to analyze the MARL case. Agents can, theoretically, completely ignore the notion of environment stationarity, but algorithms may fail to converge entirely. Agents acting in such a way are referred to as independent learners [12]. In addition to simply ignoring the non-stationarity, some other ways to deal with the problem are given in [23] as:

- **Forgetting**: In this case agents handle the changing environment by keeping a very short memory of past observations while updating information with recent observations. This usually occurs in model-free approaches and an example might be to use a modification to DQN with a very small replay buffer constantly being updated with observations from all agents.
- **Responding to target opponents**: In this scenario, agents have a clearly defined target opponent and optimize their strategies against this opponent.
- **Learn opponent models**: In these model-based approaches, agents learn a model of opponent behaviours and then develop a strategy according to this opponent model. When an opponent changes its behaviour, these agents struggle and have to learn a new opponent model in response.
- **Theory of mind**: In this case, agents model opponents, while assuming that opponents are modelling them.

Non-stationarity is arguably the most well-known challenge in MARL and, as such, is well discussed in the literature.

9.2 Various Information Structures

In MARL, the information structure at training and at execution must also be taken into account. The information in a particular multi-agent system dictates which information is available to agents. In the case of Markov games, for example, it is sufficient for agents to observe the environment state s_t so that each agent may chose its action. This is sufficient because the individual agent policies contain the equilibrium policy. In the extensive-from case on the other hand, agents might have to recall the history of past decisions assuming common perfect recall. In the case of purely self-interested agents, agents might have access to only actions taken by other agents and not the rewards or underlying policies of all other agents. Since information is partial in this case, it makes the environment appear even more non-stationary to agents. This most extreme case is the independent learners scheme, where only local actions and rewards are available to each agent.

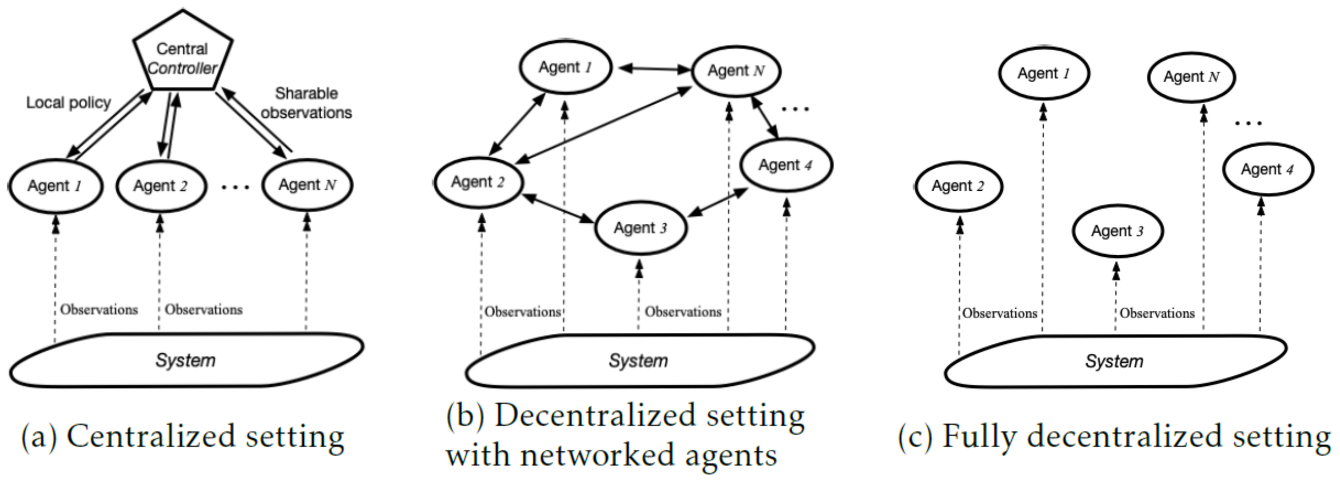


Figure 8: Three different information structures used in MARL. In (a) there exists a central controller which has access to all agent information. This central controller can design local policies for agents since it has access to all information in the system. Since (b) and (c) do not contain such a central controller, they are referred to as decentralized structures. In (b) agents are connected to each other via some communication network leading to this configuration tending to be favoured in cooperative settings. In (c) agents are taken to be fully decentralized and can only make decisions based on local observations. These agents are also taken to have no form of information sharing with each other. Figure taken from [12]

Different information structures give rise to various learning schemes which have varying levels of difficulty in theoretical analysis. The existence of a central controller can mitigate the issue of partial information since it can collect information on the joint rewards, actions and observations of agents and can even design individual agent policies. A very popular learning scheme, centralized-learning-decentralized-execution has arisen form this particular structure and is widely using in recent deep MARL applications. For the cooperative setting, this learning structure simplifies the analysis to the extent that SARL analysis tools may by used. The simplification is not as drastic in the non-cooperative case though as agents are heterogeneous and their goals are unaligned. In general however, a centralized controller is not always feasible and doesn't even exist in many cases. It is for this reason that a decentralized learning scheme is required of which the independent learning scheme is a special case. A natural by-product of independent learners is non-convergence which is most commonly addressed through introducing some sort of communication sharing network between agents. This setting is referred to by the authors in [12] as a decentralized setting with networked agents.

9.3 Scalability

A way for agents to handle the non-stationarity in the environment, assuming all information about other agents is available, is for agents to take the full joint actions space into account. Assuming there are N agents in the environment this joint action space dimensionality scales exponentially as $|\mathcal{A}|^N$. This issue is also referred to as the combinatorial nature of MARL and leads to problems becoming intractable or requiring an impossible amount of computational resources to solve in the deep MARL case. Moreover, this approach could lead to very slow convergence, if any. This problem is further implicitly highlighted in the literature since the theoretical analysis for MARL in the two-player zero-sum case is comparatively much more advanced that that of the general-sum setting with more than two agents. Aside from the fact that a centralized controller does not always exist, its use is foregone in order to help with the scalability issue. Another proposed solution to the scalability issue is to assume factorized structures of either the value of rewards functions with regard to the action dependence [12].

10 MARL Algorithms

We now move on to discuss some deep MARL algorithms. Although we cannot nearly cover an exhaustive number of algorithms, we aim to show extensions of the SARL algorithms discussed previously to the multi-agent scenario.

10.1 Multi-Agent Deep Q-Networks

One of the most popular methods for solving the MARL problem is Independent Q-Learning (IQL) first introduced by Tan [24] in 1993 which makes use of tabular methods for solving the MARL system using the independent learners framework previously discussed. In more recent times, following on from the success of Mnih et al. [16] using DQN in the SARL case, Tampuu et al. [25] sought to use similar techniques as well as the independent Q-learning to extend DQN to the multi-agent setting. They investigated the emergence of both cooperative and competitive behaviours between two agents playing the Atari game of Pong.

As was mentioned earlier, independent learning brings relief to the scalability issue in MARL but it directly leads to non-stationarity since each agents views other agents as part of the environment. It is this non-stationarity which makes the use of replay buffers in independent learning MARL challenging. Some other solutions to this problem have been to simply ignore other agents be foregoing the use of a replay buffer altogether or by using very short replay buffers leading to agents forgetting past, irrelevant, experience. This very non-stationarity issue is addressed in a unique way by Foerster et al. [26] where the authors propose two methods for stabilising experience replay which enables the use of deep Q-learning techniques in the MARL setting. These methods are to: 1) use a multi-agent variant of importance sampling such that irrelevant experience becomes largely ignored and 2) introduce a fingerprint that disambiguates the age of data sampled from the replay buffer.

Multi-Agent Importance Sampling

It has been shown that in RL, agents can not only learn from off-policy data, but that agents can similarly learn from off-environment data [26]. That is to say that agents can be trained on data that has been collected in another environment altogether. The fact that IQL treats all other agents as part of the environment, allows for the use of off-environment importance sampling to help with stabilising the experience replay. In the fully observable MARL case, since the policies of all agents are known, the change in the environment can be suitably deduced and corrected for with importance sampling. In this fully observable case the the Bellman optimality equation for a single agent given the policies of all other agents can then be given as:

$$Q^{i*}(s, a^i|\boldsymbol{\pi}^{-i}) = \sum_{\mathbf{a}^{-i}} \boldsymbol{\pi}^{-i}(\mathbf{a}^{-i}|s)\left[r(s, a^i, \mathbf{a}^{-i}) + \gamma \sum_{s'} P(s'|s, a^i, \mathbf{a}^{-i}) \max_{a'^i} Q^{i*}(s', a'^i)\right] \tag{46}$$

The non-stationarity in equation (46) arises from the term $\boldsymbol{\pi}^{-i}(\mathbf{a}^{-i}|s) = \Pi_{j\in -i}\pi^j(a^j|s)$ which will change as the policies of all other agents change. The solution to this is to record $\boldsymbol{\pi}_{t_c}^{-i}(\mathbf{a}^{-i}|s)$ in replay at the time of collection t_c, which gives rise to a transition tuple $\langle s, a^i, r, \pi(\mathbf{a}^i|s), s'\rangle^{(t_c)}$.

At replay, or training, time t_r training is done in an off-environment fashion by minimzing a loss function weighted by importance which can be given as

$$L(\theta) = \sum_{j=1}^{b} \frac{\boldsymbol{\pi}_{t_r}^{-i}(\mathbf{a}^{-i}|s)}{\boldsymbol{\pi}_{t_j}^{-i}(\mathbf{a}^{-i}|s)}\left[\left(y^j_{\text{DQN}} - Q^\theta(s, a)\right)^2\right] \tag{47}$$

Here b denotes the size of the batch sampled from the reply buffer and t_j and y^j_{DQN} are the collection time and familiar DQN target respectively for sample j. Intuitively, the ratio in equation (47) implies that more important experience has a greater chance of being sampled from the reply buffer. This prioritization is possible since the policies of all agents are known during training time. The authors go on to show that that a similar, although more complex analysis may be done in the partially observable case. It is, however, shown that this analysis leads to equations of a similar form to (46) but that has terms which the authors deem to be intractable since they have indirect dependence on policies of other agents. For this reason the importance ratio given in (47) is taken to be an approximation for the partially observable case. The use of importance sampling is not fully robust though. The authors note that although importance does give an unbiased estimate of the true loss, the importance ratios can have very high variance. This variance can also be mitigated by clipping the importance weights, but this procedure then introduces bias.

Multi-Agent Fingerprints

The idea of fingerprints draws from the ideas of hyper Q-learning which exploits the fact that, if a Q-function is conditioned on the policies of other agents in the environment, it can be made stationary. Hyper Q-learning achieves this through Bayesian inference, thereby reducing each agent's problem to a single-agent problem in a stationary environment with the cost of making the environment larger. This is the weakness of hyper Q-learning, which makes the Q-function possibly infeasible to learn due to its increase in dimensionality. In deep learning, where agents' policies are modelled by larger deep neural networks this problem is even worse. A naive approach could be to simply concatenate the weights of all other agents' networks $\boldsymbol{\theta}^{-i}$ to an agent's observation such that $o'(s) = \{o(s), \boldsymbol{\theta}^{-i}\}$. Clearly this will produce an input that is much too large to be used as input to an agent's Q-function being modelled by a deep neural network.

A key observation, however, is that, in order to stabilize the experience replay it is not necessary for an agent to condition on any possible $\boldsymbol{\theta}^{-i}$ but only on those value of $\boldsymbol{\theta}^{-i}$ which occur in its replay memory. This implies that the sequence of policies that created the data in this replay buffer can be thought of as following some single, one dimensional trajectory through an otherwise high-dimensional policy space. This implies that, in order to stabilise the replay buffer, an agent must be able to distinguish where along this one dimensional trajectory the current sample originated.

The only other problem to tackle then, is how to create such a low-dimensional fingerprint. The authors note that such a fingerprint should satisfy two criteria. Firstly, it should be correlated with the true value of the state-action pairs given other agent's policies. Secondly, it should vary smoothly over the training. The reason such smoothness is desired is so that the model can generalise across experiences where other agents are executing policies of varying levels of quality as they learn.

The authors come up with a simple and elegant solution to augment the agent observations. They suggest, firstly, to add the training iteration number e. This does pose a challenge, in that once the model has converged it must then fit to multiple fingerprints of the same value making it harder to learn and to generalise from. Another smart choice to add to the observation is the rate of exploration ϵ, since it typically anneals smoothly during training and is correlated with agents' performance. This implies that the new agent observation becomes $o(s) = \{o(s), \epsilon, e\}$. The authors perform experiments on agent cooperation in the Starcraft unit micromanagement environment with the following surprising results:

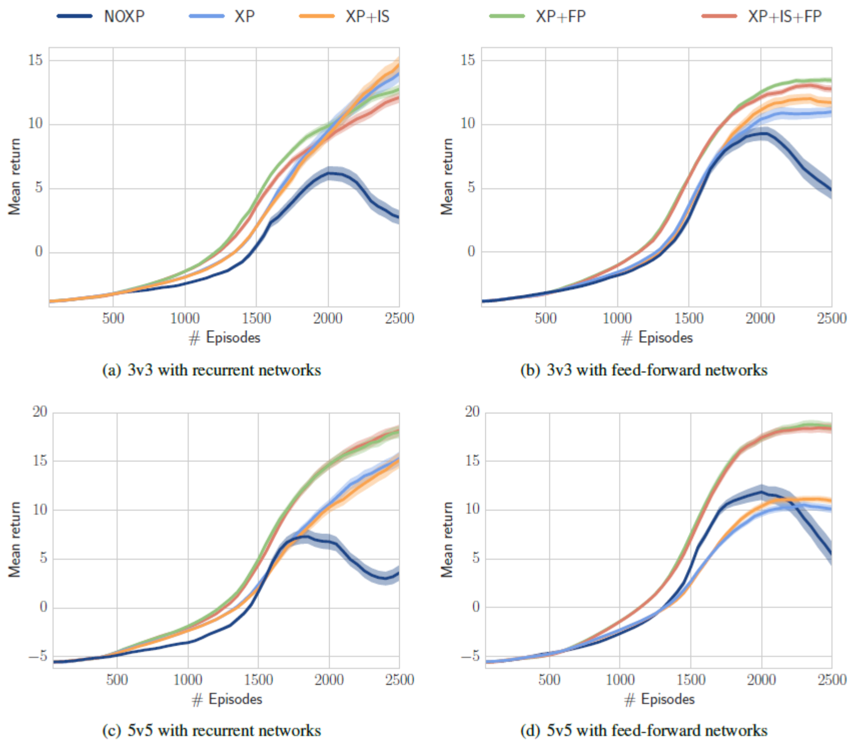


Figure 9: Mean returns of cooperative agents in the Starcraft unit micromanagement environment. Recurrent agent use a Gated Recurrent Unit recurrent Q-network architecture while the feed forward use the standard feed forward deep neural network to estmate the Q-function. In both cases these networks have 128 neurons in each hidden layer. XP and NOXP denote agents with and without experience replay. IS denotes agents with importance sampling and FP denotes agents with fingerprints. Figure taken from [26]

As seen in Figure 9 the authors use a recurrent neural network architecture in their implementations as well as a standard feed-forward neural network architecture. This is a technique introduced in [27] to help in situations where the memory of previous actions are important in agent learning. This memory is modelled by the way in which Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures are able to maintain a so-called hidden state correlated wih previous states. It is also interesting to note from Figure 9 that simply adding fingerprints leads to a performance increase that can rival, or even beat, adding importance sampling or a combination of importance sampling and fingerprints.

10.2 Multi-Agent Deep Deterministic Policy Gradient

Muli-Agent Deep Deterministic Policy Gradients (MADDPG) were introduced by Lowe et al. [28] and seek to extend the success of DDPG, and policy gradient methods in general, to the multi-agent case.

A unique feature of MADDPG when compared to Value Decomposition Networks (VDN) and Multi-Agent Deep Q-Networks (MADQN) is that it is applicable in cooperative, competitive and mixed settings. It is also able to scale to environments with continuous action spaces due to being based on an actor-critic architecture. Along with the policy gradient type architectures comes another MARL problem in that the gradients suffer from high variance which gets exacerbated with increasing numbers of agents in an environment. The authors mitigate this issue by making use of the centralized-training decentralized execution framework. What this entails is that, during training time, the critic has access to the observations and policies of all actors, while each actor only has access to its local observations. This allows for the learning of more robust actor policies and for actors which can be deployed independently. Another feature is that, since the centralized critic has inherent access to the policies of all actors, actors are able to learn approximate models of each other and use this information in their own policy learning.

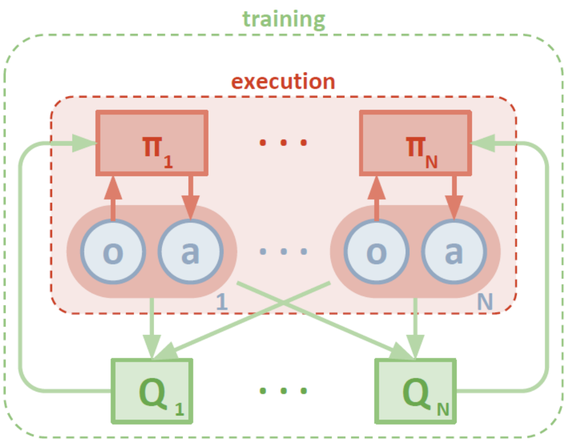


Figure 10: The multi-agent actor-critic framework. Here we can note how during training time each of the N critics have access to the observations and policies of all actors, while the actors only have access to their local observations. During execution time, the critics may be removed and actors may act fully independently of each other. Figure taken from [28]

The authors devise the MADDPG algorithm under the following constraints: 1) The learned actor policies may only have access to local information at execution time, 2) No differential model of the environment is assumed, 3) there is no particular communication structure between agents. The authors derive the algorithm by first considering a game with N agents which have policies parameterized by $\boldsymbol{\theta} = \{\boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^N\}$ and a set of all agent policies $\boldsymbol{\pi} = \{\pi^1, \dots, \pi^N\}$. The expected return for agent i can be given as $J(\theta^i) = \mathbb{E}[\mathcal{R}^i]$ which has the gradient

$$\nabla_{\theta^i} J(\theta^i) = \mathbb{E}_{s\sim p^{\mu}, a^i \sim \pi^i}\left[\nabla_{\theta^i} \log \boldsymbol{\pi}^i(a^i|o^i) Q^{\pi^i}(X, a^1, \dots, a^N)\right] \tag{48}$$

Here $Q^{\pi^i}(X, a^1, \dots, a^N)$ is the centralized action value-function which takes in the actions of all agents as some state information X. In the simplest case, X, is just the set of all agent observations but it could contain other information as well depending on the task at hand. Since each unique Q^{π^i} is learned separately, this opens the possibility for agents to have various reward structures leading to the various types of scenarios MADDPG may be applied in. Similarly to DDPG this idea may be extended such that it can be applied to deterministic policies. The authors then consider N continuous policies $\boldsymbol{\mu}_{\theta^i}$ with parameters θ^i, allowing the gradient to be written as:

$$\nabla_{\theta^i} J(\boldsymbol{\mu}_{\theta^i}) = \mathbb{E}_{X, a\sim\mathcal{D}}\left[\nabla_{\theta^i} \boldsymbol{\mu}_{\theta^i}(a^i|o^i) \nabla_{a^i} Q^{\boldsymbol{\mu}^i}(X, a^1, \dots, a^N)|_{a^i = \boldsymbol{\mu}_{\theta^i}(o^i)}\right] \tag{49}$$

Here $\mathcal{D}$ is the experience replay, as is used by DQN, but in the case of MADDPG it contains the tuples $(X, \boldsymbol{a}, \mathcal{R}, X')$. The centralized action-value function $Q^{\boldsymbol{\mu}^i}$ is then updated by optimizing according to the following loss function:

$$L(\theta^i) = \mathbb{E}_{X,\mathrm{a},\mathcal{R},X'}[(Q^{\boldsymbol{\mu}^i}(X,\mathbf{a}) - y)^2] \tag{50}$$

Where $y = \mathcal{R}^i + \gamma Q^{\boldsymbol{\mu}^i}(X', \mathbf{a}')|_{a'_j=\boldsymbol{\mu}'_j(o^j)}$ is the familiar Bellman target and $\boldsymbol{\mu}'$ is the set of target policies with parameters θ'^i. This procedure furnishes what the authors have called the MADDPG algorithm with the primary motivation behind it being that, if the actions taken by all agents is known, the environment will remain stationary despite changing agent policies. Aside from this default assumption that the policies of all agents must be known at training time such that equation (50) may be used, the authors show that this assumption can be relaxed if needed. This is achieved by learning policies of other agents from observations and is discussed in detail in [28].

The authors proceed to benchmark MADDPG against a host of SARL algorithms on various environments involving agent competition and coordination. An interesting environment is the so-called cooperative communication environment. This task consists of two agents, a speaker and a listener placed in an environment with three unique landmarks. The goal in each episode is for the listener to navigate as close as possible to a particular landmark and is rewarded proportional to how close it gets to the landmark. The listener can observe the landmark positions but it does not know which landmark it should navigate to. Conversely, the speaker is aware which landmark the listener should navigate to and it can produce some unique output that communicates the correct landmark to the listener. Therefore, the speaker and listener must learn to cooperate and understand one another for the optimal solution in the task. What makes this interesting is that no distinct differentiable communication channel exists between the agents. Only the learnt output signal from the speaker. The results on the cooperative communication environment are summarized in Figure 11.

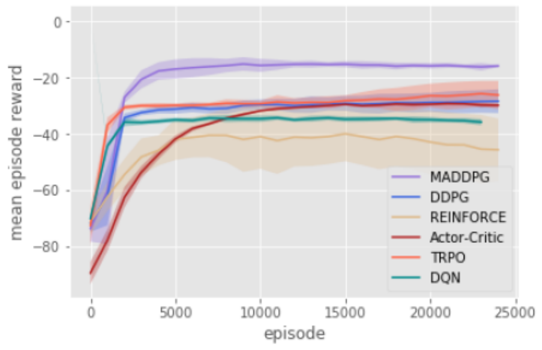


Figure 11: Reward for agents in the cooperative communication environment. Figure taken from [28]

The authors go on to show that MADDPG outperforms any SARL algorithm applied to any of the tasks considered. Please see [28] for a detailed comparison. MADDPG can be summarized by the following algorithm:

Algorithm 5 Multi-Agent Deep Deterministic Policy Gradient

```
Input Parameters: $N, n, b, \tau$
  Randomly initialize $n$ critic networks $Q^{\boldsymbol{\mu}}(X, \mathbf{a})$ and actor networks $\boldsymbol{\mu}_{\theta^i}(o)$
  Initialize $n$ target networks $Q'$ and $\mu'$
  Initialize Replay buffer $\mathcal{D}$ of Length $N$
  for episode = 1, $M$ do
      Initialize a random process $\mathcal{N}$ for action exploration
      Receive initial state $X$
      for t = 1 to maximum episode length do
          for each agent $i$ select action $a^i = \mu^{\theta^i}(o^i) + \mathcal{N}_t$ according to current policy and exploration noise
          Execute all actions $\mathbf{a}$ and observe reward $\mathcal{R}$ and new state $X'$
          Store $(X, \mathbf{a}, \mathcal{R}, X')$ in replay buffer $\mathcal{D}$
          set $X \leftarrow X'$
          for agent i = 1 to N do
              Sample random minibatch of samples $B$ of size $b$, $(X^j, \mathbf{a}^j, \mathcal{R}^j, X'^j)$ from $\mathcal{D}$
              Set $y^j = \mathcal{R}_i^j + \gamma Q^{\boldsymbol{\mu}'^i}(X'^j, \mathbf{a}'^j)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
              Update critic by minimizing the loss: $L = \frac{1}{b}\sum_j [(Q^{\boldsymbol{\mu}^i}(X^j, \mathbf{a}^j) - y^j)^2]$
              Update actor policy using the sampled policy gradient:

                  $$\nabla_{\theta^i} J \approx \left[\nabla_{\theta^i} \boldsymbol{\mu}_{\theta^i}(o_i^j) \nabla_{a^i} Q^{\boldsymbol{\mu}^i}(X^j, \mathbf{a}^j)|_{a^i = \boldsymbol{\mu}_{\theta^i}(o_i^j)}\right]$$

          end for
          Update the target network parameters for each $i$

                  $$\theta'^i \leftarrow \tau\theta^i + (1 - \tau)\theta'^i$$

      end for
  end for
```

10.3 Value Decomposition Networks

As mentioned previously, a way to mitigate the challenge of scalability in MARL is a to assume that certain structures may be factorized and this is exactly the class of solution introduced by Sunehag et al. [29] which they call Value Decomposition Networks (VDN).

Sunehag et al., also mention a general problem in cooperative MARL, which they term the problem of lazy agents. This phenomena arises when we consider the case where centralized MARL techniques can be used. By centralized we mean the case where the state observations of all agents are concatenated and the problem may be treated as one, very large, single agent problem. The authors show once again, that such a naive approach is not feasible, not only due to its intractability, but because agents become 'lazy' and stop cooperating and learning in the way they are supposed to. This leads to centralized methods not being able to find the global optimum in the reward space. A solution to the lazy agent, is to find a way in which to isolate each agent's contribution to the joint reward. The category of algorithms striving to achieve exactly this, are know as Value Function Factorization algorithms [30].

VDN was developed in the context of fully cooperative agents, where the assumption is made that the team reward is the sum of the individual agent rewards. The problem is also phrased in terms of the Dec-POMDP framework since each agent only receives the joint reward, $\mathcal{R}$, which has to be globally optimized. The novelty of VDN is that the optimal linear additive value-decomposition function is learned across multiple agents from the joint reward. Each agent's Q-values are summed to obtain a joint Q-value the gradient of which is then back-propagated through the unique deep neural networks that represent each agent's individual Q-value function. Where these individual Q-networks take only each agent's individual observation as input. What is further interesting about VDN is that although agents train in a centralized manner, they can be deployed individually. Since the problem in this case is assumed to be partially observable, each agent's actions conditions not only on its current observation, but also on its observation history. The authors incorporate this by replacing the standard feed-forward neural network architecture with a recurrent neural network. In particular, each agent's Q-network has a Long Short-Term Memory (LSTM) layer included.

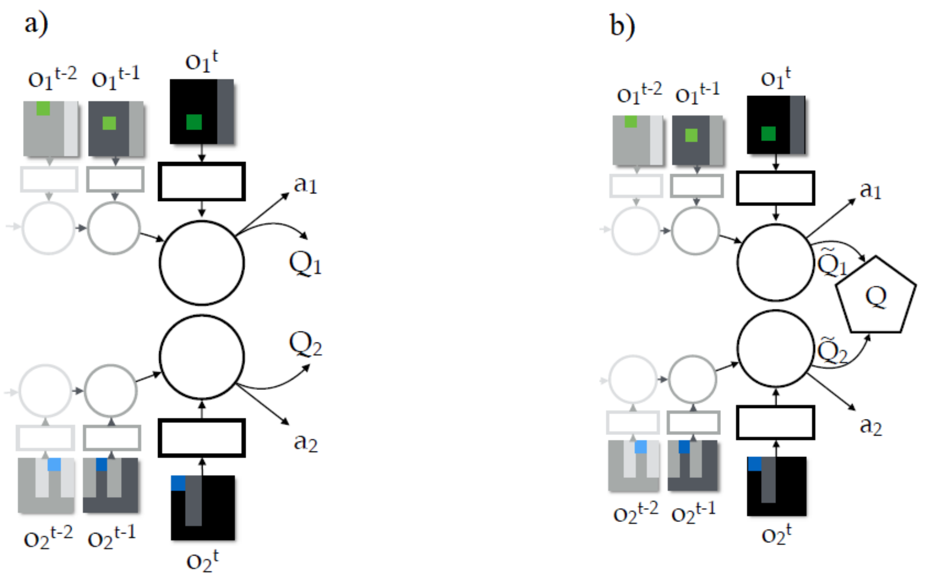

a)
o_1^{t-2} o_1^{t-1} o_1^t
a_1
Q_1
Q_2
a_2
o_2^{t-2} o_2^{t-1} o_2^t
b)
o_1^{t-2} o_1^{t-1} o_1^t
a_1
$\tilde{Q}_1$
Q
$\tilde{Q}_2$
a_2
o_2^{t-2} o_2^{t-1} o_2^t

Figure 12: A comparison between an independent agent architecture (a) and the VDN architecture (b). In (a) we can note how the individual agent Q-values are computed in the recurrent Q-network architecture. Only three consecutive agent observations are shown. In (b) we note how a joint Q-value is computed from the Q-values of each agent while agents still choose actions independently. Figure taken from [29]

As illustrated in Figure 12, the core assumption made in VDN is that this joint action-value function can be additively decomposed as

$$Q[(h^1, h^2, \dots, h^d), (a^1, a^2, \dots, a^d)] \approx \sum_{i^n} \tilde{Q}_i(h^i, a^i). \tag{51}$$

Here h^i denotes the history of agent i which is captured in the hidden state maintained by the recurrent Q-network.

It is for this very reason that agents are able to be deployed individually, since when each agents acts greedily with respect to its local $\tilde{Q}_i$ is equivalent to some central agent choosing joint actions.

To illustrate this decomposition, consider the simple case of two cooperative agents with partial observability of the state and an additive team reward such that $\mathcal{R}(s, \mathbf{a}) = \mathcal{R}^1(o^1, a^1) + \mathcal{R}^2(o^2, a^2)$. Here o^i and a^i denote the observation and action of agent i respectively. In this case we would have that

$$\begin{aligned} Q^{\pi}(s, \mathbf{a}) &= \mathbb{E}\Big[\sum_{t=1}^{\infty} \gamma^{t-1} \mathcal{R}(s_t, \mathbf{a}_t) | s_1 = s, \mathbf{a}_1 = \mathbf{a}; \pi\Big] \\ &= \mathbb{E}\Big[\sum_{t=1}^{\infty} \gamma^{t-1} \mathcal{R}^1(o_t^1, a_t^1) | s_1 = s, \mathbf{a}_1 = \mathbf{a}; \pi\Big] + \\ &\quad \mathbb{E}\Big[\sum_{t=1}^{\infty} \gamma^{t-1} \mathcal{R}^2(o_t^2, a_t^2) | s_1 = s, \mathbf{a}_1 = \mathbf{a}; \pi\Big] \\ &\equiv \overline{Q}_1^{\pi}(s, \mathbf{a}) + \overline{Q}_2^{\pi}(s, \mathbf{a}) \end{aligned} \tag{52}$$

The use of recurrent networks, allows equation (52) to be expressed as:

$$Q^{\pi}(s, \mathbf{a}) = \overline{Q}_1^{\pi}(s, \mathbf{a}) + \overline{Q}_2^{\pi}(s, \mathbf{a}) \approx \tilde{Q}_1^{\pi}(h^2, a^2) + \tilde{Q}_2^{\pi}(h^2, a^2) \tag{53}$$

which uses not only the agent's observations, but also the agent's history.

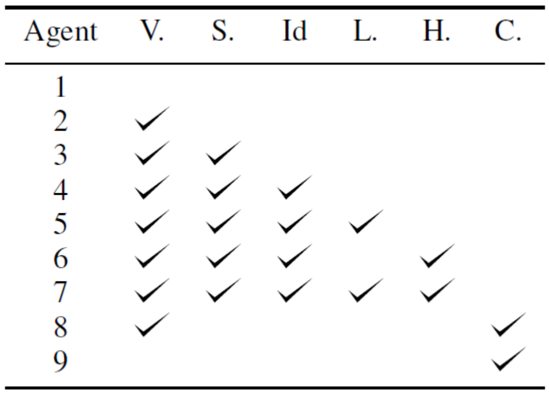
Once the total Q-value $Q^{\pi}(s, \mathbf{a})$ has been computed the loss is computed similarly to the standard DQN by making use of the Bellman target and the gradients back-propagated through the individual agent networks. The loss can be given as

$$L(\theta) = \sum_{i}^{b} \left[(y_i^{\pi} - Q^{\pi}(\mathbf{h}, \mathbf{a}, s; \theta))^2 \right] \tag{54}$$

where b is the batch size of transitions from the replay buffer, $y_i^{\pi} = \mathcal{R} + \gamma \max_{\mathbf{a}'} Q^{\pi}(\mathbf{h}', \mathbf{a}', s'; \theta^-)$ is the Bellman target and θ^- is the parameters of the target network.

Aside from the loss computations, VDN also makes use of the experience replay buffer and target networks akin to the DQN implementation as mentioned.

To investigate the effectiveness of VDN, the authors make use of VDN in conjugation with various other techniques illustrated in Figure 13



Agent	V.	S.	Id	L.	H.	C.
1						
2	✓					
3	✓	✓				
4	✓	✓	✓			
5	✓	✓	✓	✓		
6	✓	✓	✓		✓	
7	✓	✓	✓	✓	✓	
8	✓					✓
9						✓

Figure 13: Additional architectures used in conjunction with VDN. V denotes value decomposition, S denotes shared weights with invariant agent networks, Id denotes that agent role information was added to observations, L denotes lower level agent communication, H denotes higher level agent communication and C denotes centralization. Figure taken from [29]

We briefly explain some of these additional architectures. Shared weight between agents implies that the weights for a single network is shared between all agents. This implies that agents are homogeneous, but the observation of each agent is processed through the recurrent Q-network individually, differentiating this approach from the centralized approach or the approach using independent learners each with unique Q-networks. Agent role information serves as a way to disambiguate individual agents in the shared weights case or encoding some information about unique agent roles. This is accomplished by concatenating a unique one-hot encoded vector to the observations of each unique agent. C is the naive fully centralized approach discussed earlier. For high-level and low-level communication, different layer outputs between agent Q-networks are concatenated before further passing through each agent's network. For a detailed discussion of the communications structure, please see [29].

The authors test VDN on three 2D grid-world environments, namely:

Switch: In this task agents appear on opposite ends of a map and must reach a goal on the other end of where they appear. What make the environment challenging is that agents must pass through a narrow corridor in the middle of the map and only one agent can pass through the corridor at a time. If agents collide in the corridor, one agent must back up so that another agent can pass through which then allows itself to pass through afterwards. A point is scored whenever an agent reaches its goal.

Fetch: In Fetch, agents start on the same side of the map and have collection points on the other side of the map. Agents must pick-up an object on the other side of the map and return it to a drop-off point close to where the agents started. Agents receive a reward of 3 for pick up and an additional reward of 5 if the object is dropped off at the drop-off point. Once the object is dropped off, a new object appears which is available to all agents. The optimal behaviour is that agents coordinate such that one agent collects the object and, as it travels back to drop off the object, another agents travels to the pick-up point so that it can collect the object immediately when it spawns, creating a constant flow of reward for both agents.

Checkers: The map contains apples and lemons in a checkered pattern. One agents is very sensitive and obtains a team reward of 10 for eating an apple and -10 for eating a lemon. The second agent is less sensitive and scores a team reward of 1 for eating an apple and -1 for eating a lemon. Apples and lemons disappear when collected. The optimal behaviour is that the less sensitive agents eats all the lemons, allowing the sensitive agent to eat only apples such that the team's reward is maximized.

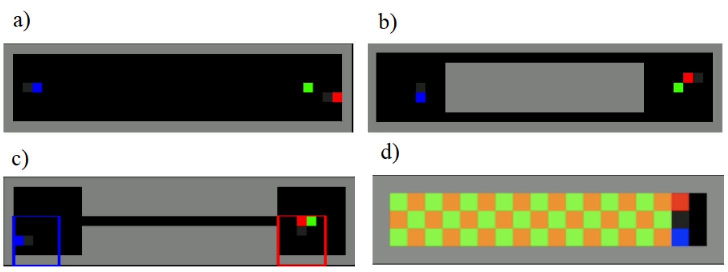
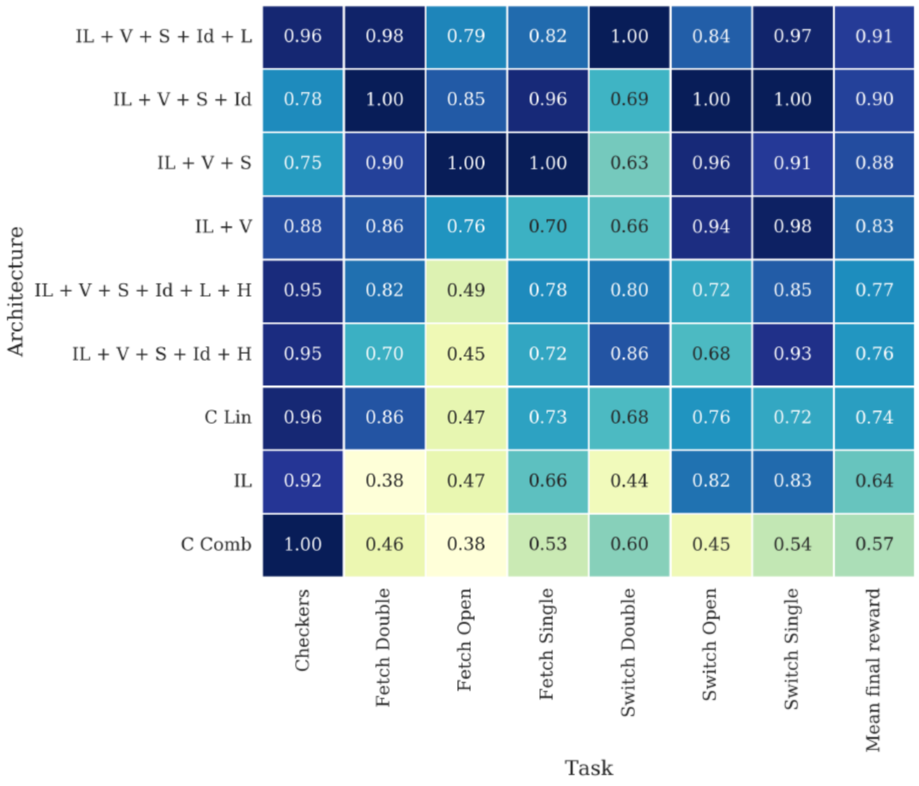


Figure 14: 2D Grid World games VDN was tested on. (a) shows the Fetch game. (b) and (c) show the Switch game with a single corridor and double corridors. (d) shows the Checkers game. Agents are shown in red and blue while objectives are in other colours. Figure taken from [29]

The authors give their findings in the following matrix. Here the scores across various architecture configurations are given. Scores are normalized by the best architecture for a particular task. The findings are not very clear to interpret. But we want to draw attention to the second-last row from the bottom (IL) and third and forth row from the top of the matrix (IL + V + S and IL + V). From this, one can note that only introducing VDN outperforms independent learners in nearly all tasks.



Architecture \ Task	Checkers	Fetch Double	Fetch Open	Fetch Single	Switch Double	Switch Open	Switch Single	Mean final reward
IL + V + S + Id + L	0.96	0.98	0.79	0.82	1.00	0.84	0.97	0.91
IL + V + S + Id	0.78	1.00	0.85	0.96	0.69	1.00	1.00	0.90
IL + V + S	0.75	0.90	1.00	1.00	0.63	0.96	0.91	0.88
IL + V	0.88	0.86	0.76	0.70	0.66	0.94	0.98	0.83
IL + V + S + Id + L + H	0.95	0.82	0.49	0.78	0.80	0.72	0.85	0.77
IL + V + S + Id + H	0.95	0.70	0.45	0.72	0.86	0.68	0.93	0.76
C Lin	0.96	0.86	0.47	0.73	0.68	0.76	0.72	0.74
IL	0.92	0.38	0.47	0.66	0.44	0.82	0.83	0.64
C Comb	1.00	0.46	0.38	0.53	0.60	0.45	0.54	0.57

Figure 15: Performance of VDN in conjunction with various architectures. Figure taken from [29]

Part III

Implementations

All implementations are done on the following three tasks which were obtained from the ma-gym GitHub repository maintained by Koul [9]:

Checkers: This is a modified version of the Checkers implementation used by Sunehag et al. [29]. It is a 24×3 2D grid world with 9 lemons, 9 apples and two agents. Once again there is a sensitive agent which scores -10 for eating a lemon and +10 for eating a apple and a non-sensitive agent scoring -1 for eating a lemon and +1 for eating a apple. An episode runs for either 100 time increments or until all the fruits have been consumed by the agents. Again, the goal of the agents are to maximize their joint joint reward implying that the sensitive agent should eat apples while the non-sensitive agent eats all lemons in the way of apples. The action space is discrete with each agent being able to move up, down, left, right or stand still. Each agent receives a local observation consisting of its current coordinate and a 3×3 area around it. The environment may also be adjusted such that each agent also receives the observation of the other agent.

Switch: This is modified version of the Switch environment used by Sunehag et al. [29]. In this case it is a 21×3 2D grid world with a narrow corridor. Agents start on opposite sides of the map and must reach a goal state on the other side of the corridor requiring cooperation since one agent has to make way for another to pass through the corridor. Agents receive a reward of +5 for making it to their unique goal state and the joint goal is to maximized the reward function. Episodes terminate after either 100 time steps or when both agents have reached their individual goal states. The action space is again discrete with agents being able to move up, down, left, right or stand still. For their local observations, agents only receive their own coordinates. The environment may also be adjusted such that agents receive the observations of all other agents.

Predator-Prey: This is an adaptation of the familiar predator-prey task discussed widely in the literature [28]. The environment is a 5×5 grid work with two fast moving predators and one slowly moving prey which is randomly placed. Here a prey is considered caught if it is directly adjacent to a predator. Moreover, to encourage cooperation and communication, the predators receive a reward of one when both catch the prey simultaneously but a reward of -0.5 if only one predator catches the prey. The game is not competitive since the prey is not controlled by a trainable agent, but by a pre-trained policy. Agents may take discrete actions which including moving up, down left, right or standing still. The local observation of each agent consists of its coordinates and the coordinate of the prey relative to itself in a 5×5 view if observed. Once again, the environment may be adjusted such that predators also receive each other's observations.

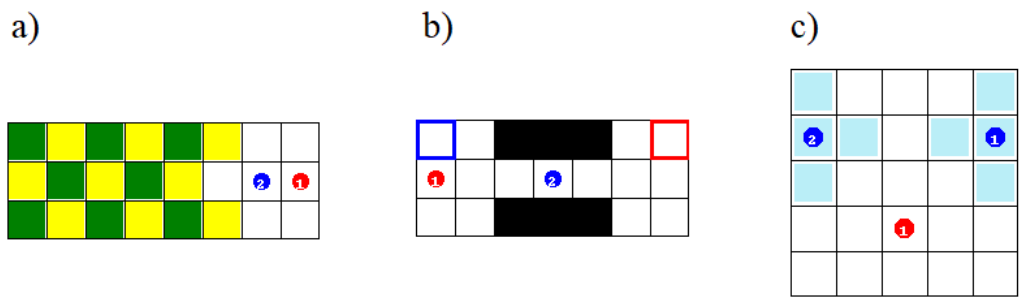


Figure 16: Tasks from the ma-gym GitHub repository. (a) is the Checkers game with lemons in yellow and apples in green. (b) is the Switch game. (c) is the Predator-Prey game. Figure taken from [9]

We firstly implement the standard DQN and Double DQN (DDQN) [31] algorithms on all environments as independent Q-learners with only partial observability. We then implement these same algorithms with full state observability where agents have access to each other's observations as well. These implementations are an adaptation of the codes by Formanek [32] where we have added the DDQN implementation. In all cases, agents have shared network parameters and can be taken to be homogeneous. A smaller replay buffer size of 5000 is used to forget experience that is no longer relevant to agents in order to mitigate non-stationarity in the environment. All agents are trained for 5000 episodes, which leads to roughly 500000 training steps. This smaller buffer size was chosen due it's suspected performance improvements.

To incorporate the partial observability of all the games considered, we implement recurrent version of DQN and DDQN which we call DRQN and DDRQN. In order to make this adaptation, the Q-networks of agents replaced with either Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) deep neural networks. This allows agents to maintain some memory about previous observations. In all our implementations, agents have shared network weights, but we disambiguate agents from each other by appending a one-hot agent ID to each agent observation. Furthermore, we make use of fingerprints by appending the current training iteration and exploration parameter values to each observation. All codes for these implementations were written from scratch and are accessible in the following GitHub repository [33]. Whenever fingerprinting is used, the size of the replay buffer is increased to 200000 since fingerprints should help to stabilize experience replay otherwise the replay buffer size is 5000 for similar reasons as in the feed-forward Q-learning case. We keep all other parameters the same across all implementations since we are not trying to find optimal implementations, but merely to compare the performance of various algorithm configurations on the same games. A more extensive study can of course be undertaken to determine the optimal hyper parameters for each algorithm.

The shared hyper parameters between all algorithms can be given as:

Parameter	Value
Batch Size	64
Exploration Decay Rate	0.99999
Minimum Exploration Value	0.05
Learning Rate	5×10^{-4}
Target Network Update Period	100
Hidden Layer Sizes	64
Discount Factor	0.99

Table 1: **Hyperparameters Used for Implementations**

In all cases we allow the exploration rate to decay exponentially at a rate of 0.99999 until it reaches a minimum value of 0.05 after which it remains constant. We also train all algorithms for 5000 completed episodes in all cases. Ideally, we would want to be able to derive some statistics by training the same agent configuration multiple times on the same environment, but due to computational and time constraints we are only able to train each agent configuration once. Doing only a single training run of all agent configurations on these three tasks took nearly 28 hours on one Nvidia Tesla P100 GPU.

11 Findings

For all task implementations, we present figures for the performance of the standard independent Q-learners implementations, the DRQN implementation configurations and the DDRQN implementation configurations. Finally we produce a Figure comparing the best performing algorithms from each of the previous results. In all plot legends, FO denotes the fully observable IQL case where agents have access to each other's observations and PO denotes the case where agents only have access to their own local observations. In all recurrent implementations, agents always only have local observations.

11.1 Checkers

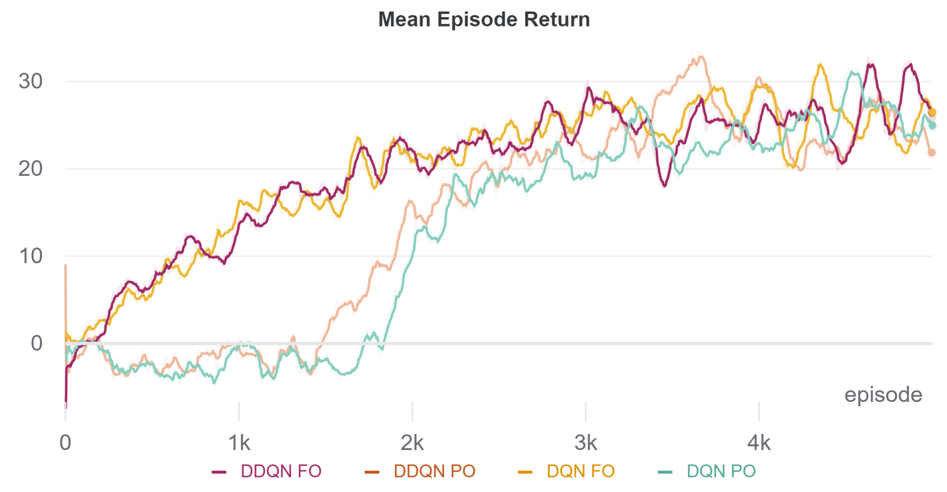


Figure 17: Mean episode return for the Checkers game with independent Q-learning agents.

From Figure 17 we can note that independent Q-learners with observability of each other's actions tend to learn faster than those that do not. After 5000 episodes, all agents have very similar mean returns for the DDQN algorithm with full agent observability performing slightly better.

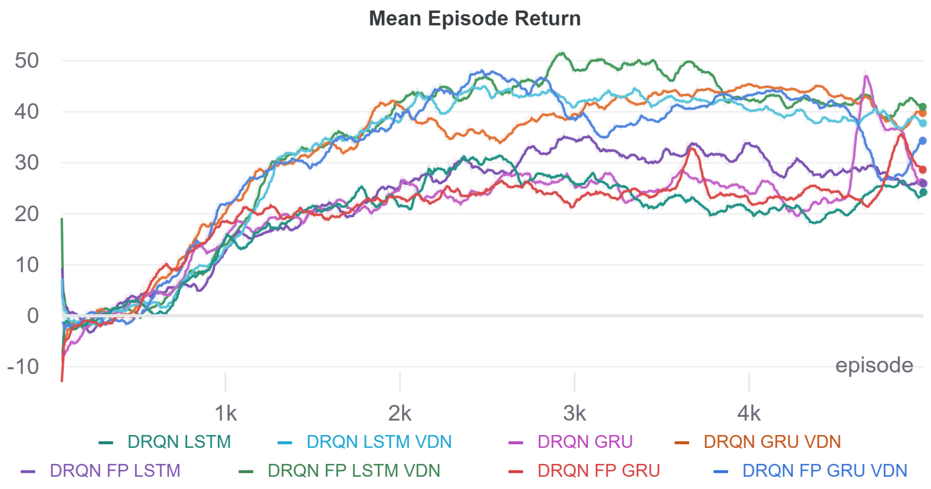


Figure 18: Mean episode return for the Checkers game with agents using Deep Recurrent Q-Networks.

From Figure 18 we can note that agents with VDN algorithms tend to perform better than agents without VDN. It seems that all agents reach peak performance around 3000 episodes and then start to forget. This could potentially be mitigated by increasing the replay buffer size or training agents for more episodes. All VDN based algorithms appear to have roughly similar final scores but the agent using an LSTM network with VDN and fingerprinting appears to have a slightly higher overall and final score.

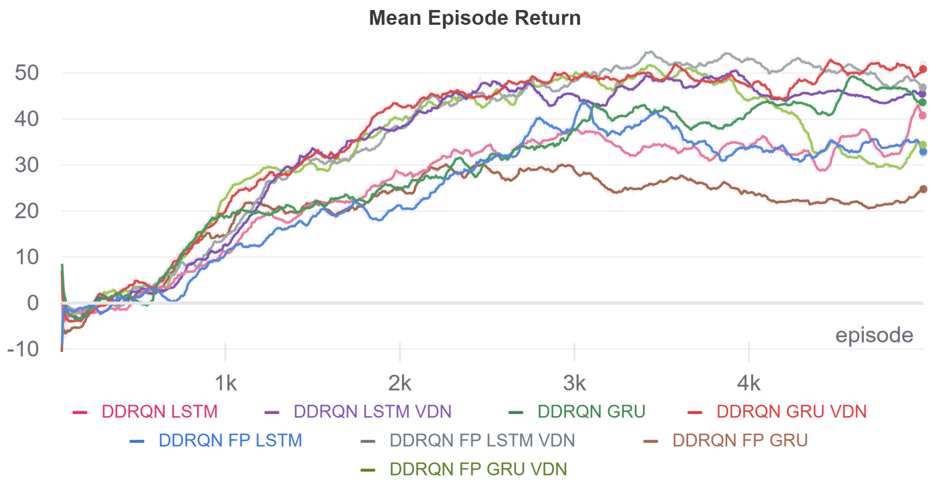


Figure 19: Mean episode return for the Checkers game with agents using Double Deep Recurrent Q-Networks.

In Figure 19 a very similar pattern emerges with VDN based algorithms performing better and learning faster. Moreover, in this case, agents to not appear to be forgetting as much as in the DRQN case with the average rewards appearing plateau. The highest scoring algorithm in this case uses VDN and a GRU network.

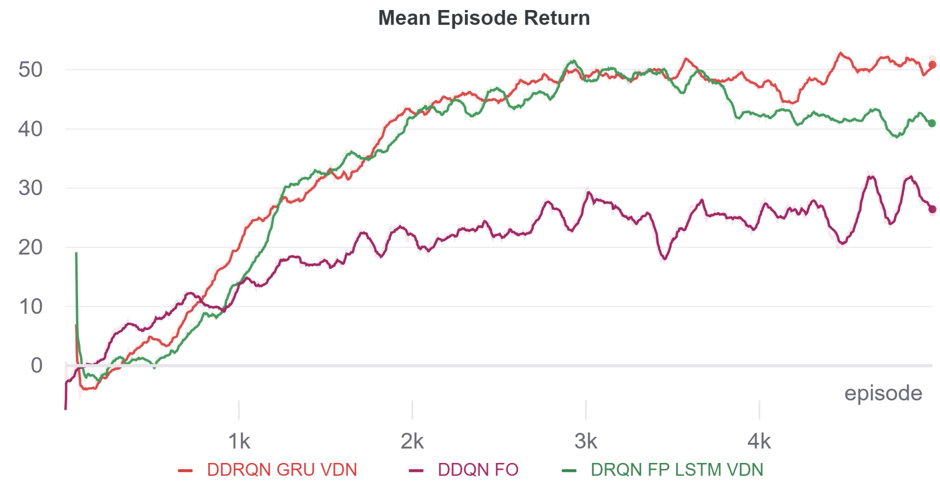


Figure 20: Mean episode return for the Checkers game with best performing agents.

From Figure 20 we can note that the overall best performing algorithm in the Checkers environment is the DDRQN using a GRU network and VDN.

11.2 Switch

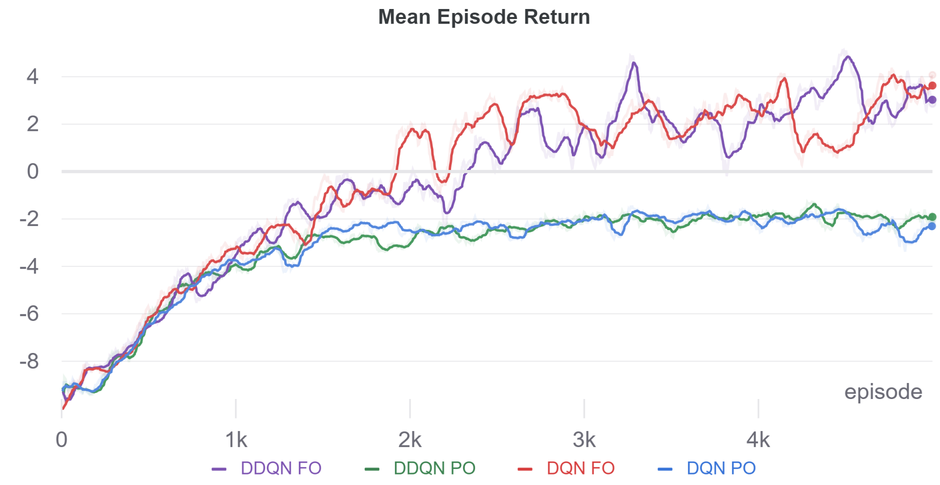


Figure 21: Mean episode return for the Switch game with independent Q-learning agents.

We can again note from Figure 21 that agents with observability of each other's actions tend to perform better overall. In contrast to the Checkers game however, the performance of agents with only local observations never achieve the same performance as their better informed counterparts. We can also note that DDQN and DQN have very similar performance in this case with DDQN achieving a slightly higher overall mean episode return.

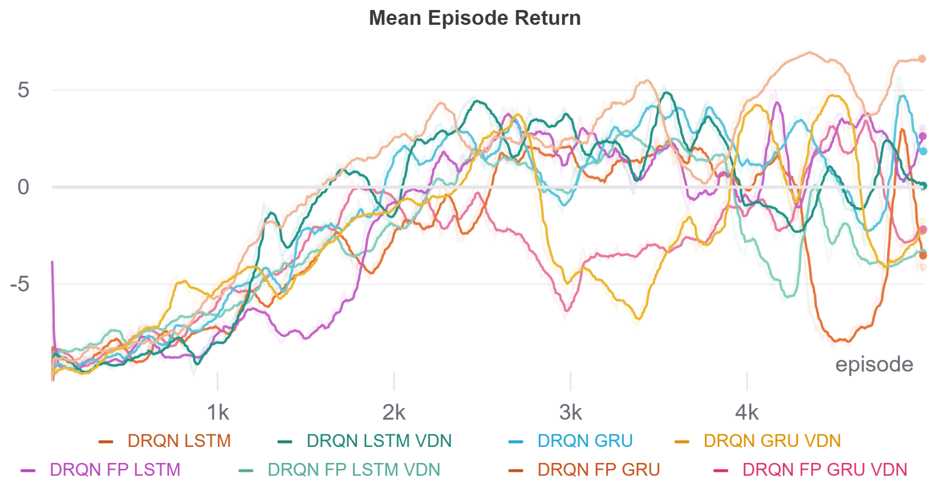


Figure 22: Mean episode return for the Switch game with agents using Deep Recurrent Q-Networks.

There is a larger variance in the mean episode returns for all DRQN agents as illustrated in Figure 22. Training does seem to slight plateau around 2500 episodes for most agents however. The best performing algorithm in this case is when agents use a GRU network and fingerprints.

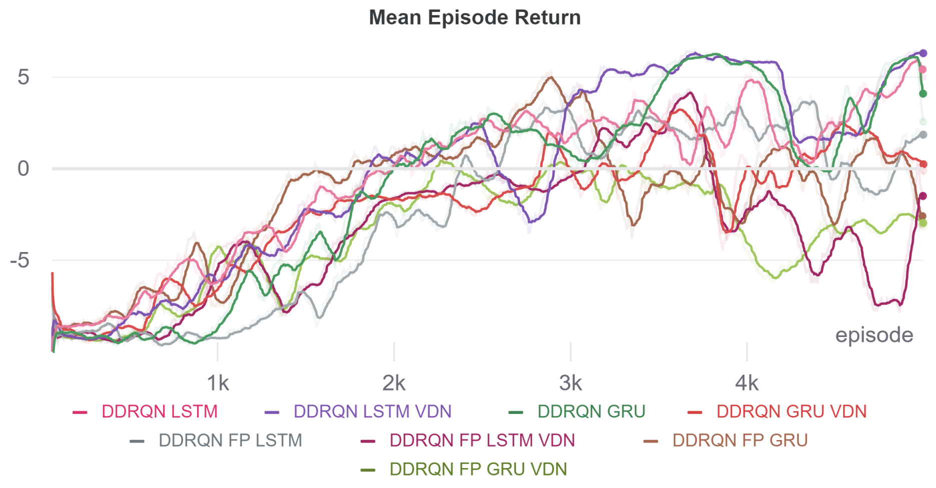


Figure 23: Mean episode return for the Switch game with agents using Double Deep Recurrent Q-Networks.

Similarly to the DRQN agents, we can see in Figure 23 that the variance in mean episode returns grows as increasing training episodes. The best performing algorithm in this case is when agents have an LSTM network with VDN.

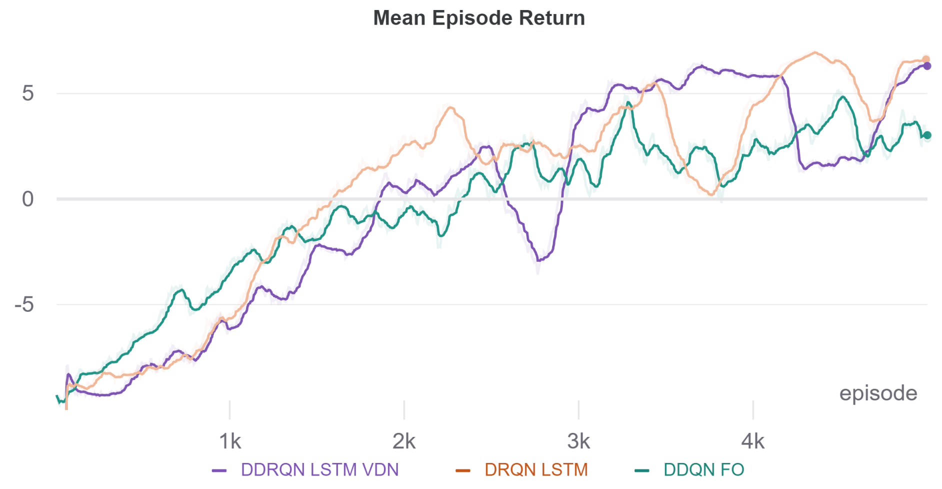


Figure 24: Mean episode return for the Switch game with best performing agents.

We can note in Figure 24 that the recurrent model agents are able to outperform the IQL agents with share observations between each other. This is an interesting result since agents have to learn interesting behaviour in this scenario. Agents have to learn to take seemingly undesirable steps in order to benefit the team as a whole. What make this even more interesting is that agents have no communication with each other, they are only able to see whether they are close to each other or not.

11.3 Predator-Prey

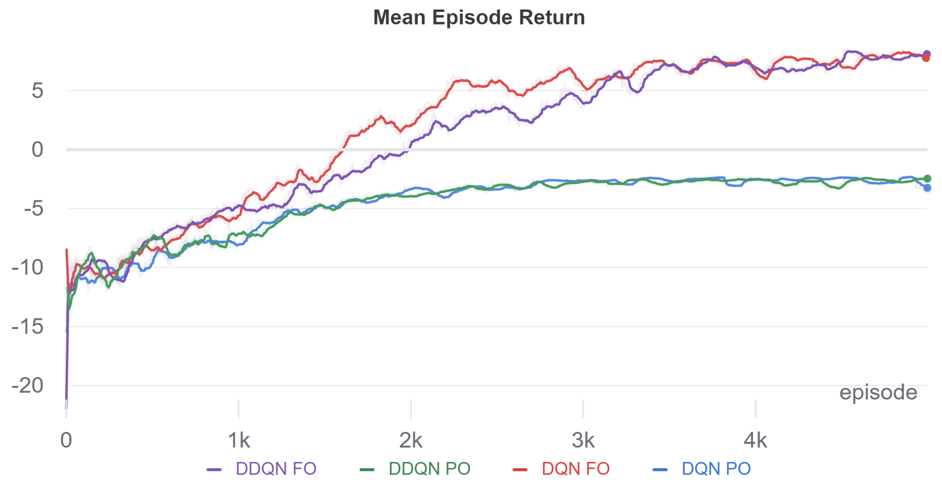


Figure 25: Mean episode return for the Predator-Prey game with independent Q-learning agents.

Predator-Prey is an environment that requires clear communication between agents and it can be noted from Figure 25 that having access to each other's observations serves as a means of communication since since agents are able to coordinate such that they catch the prey together. The agents with partial observability have no means of coordinating such that they catch the prey together. Both algorithms with agents with shared observations perform relatively similarly with DDQN obtaining a slightly greater mean episode return.

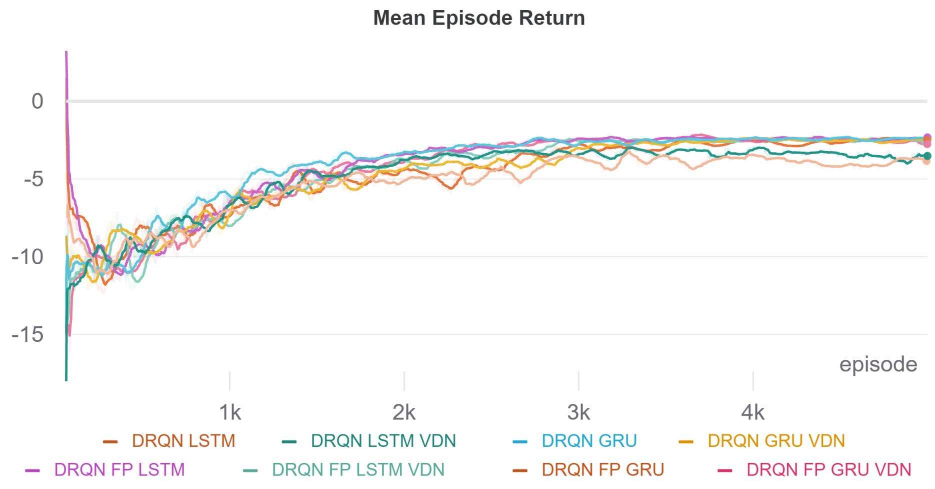


Figure 26: Mean episode return for the Predator-Prey game with agents using Deep Recurrent Q-Networks.

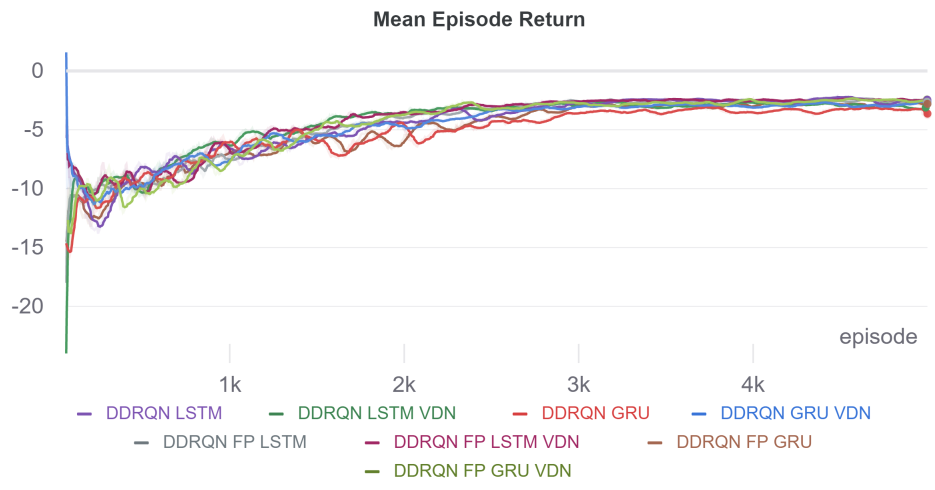


Figure 27: Mean episode return for the Predator-Prey game with agents using Double Deep Recurrent Q-Networks.

It is apparent from both Figures 26 and 27 that some form of communication is needed in the Predator-Prey task in order for agents to succeed. No algorithm configuration is able to every attain a positive score, with all converging to the same mean episode return of roughly -3. This environment is particularly challenging since agents not only must catch prey together, but are also punished for individually catching the prey. This means that all external observations like agents' position relatively to each other is not too useful to agents. Moreover, the moving prey further complicates the agents' task. Subsequently, we can not really choose a best algorithm configuration in this case.

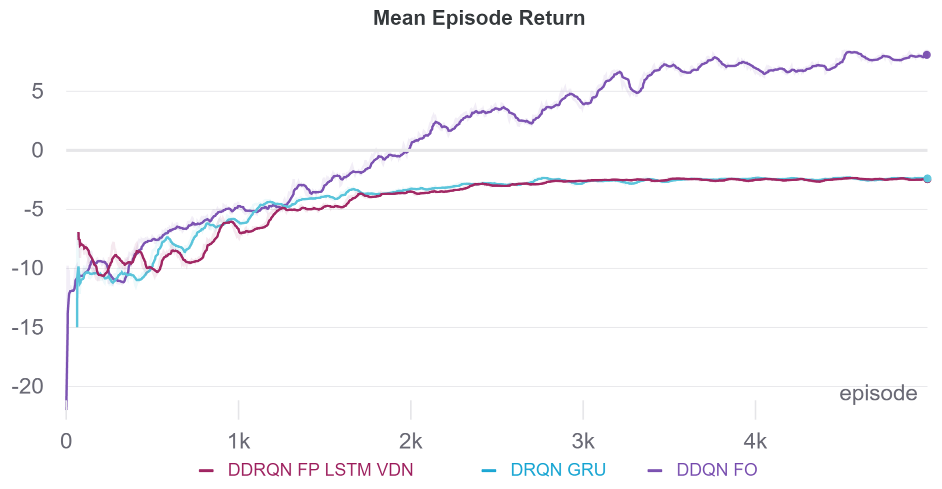


Figure 28: Mean episode return for the Predator-Prey game with best performing agents.

Figure 28 serves as a means to illustrate that communication between agents is required in this task.

12 Conclusions

We have discussed and given an overview of the SARL case as well as some fundamental algorithms. We have also given an overview of the problem formulation and main challenges in the MARL case and discussed 3 deep MARL algorithms. We then concluded by implementing various network configurations on three multi-agent tasks.

Acknowledgements

We wish to acknowledge and thank Claude Formanek for his help with coding implementations and for useful discussions and insights. We also want to thank Dr. Jonathan Shock for the opportunity and the space to investigate such a truly inspiring topic.

References

[1] Tesauro, G., "*Temporal difference learning and td-gammon*", Communications of the ACM, **38**, Issue 3, pp. 58-68 (1995)

[2] Silver, D., et al., "*Mastering the game of Go with deep neural networks and tree search*", Nature, **529**, 484-489, (2016)

[3] Yang, Y., Wang, J., "*An Overview of Multi-agent Reinforcement Learning from Game Theoretical Perspectives*", 31st International Conference on Machine Learning, CoRR, arXiv:2011.00583, (2020)

[4] Vinyals, O., et al., "*Grandmaster level in starcraft ii using multi-agent reinforcement learning*", Nature, **575**, Issue 7782, pp. 350-354 (2019)

[5] Pachoki J., et al., "*Openai five*", `https://blog.openai.com/openai-five`, (2018)

[6] Jadenberg, M., et al., "*Human-level performance in 3d multiplayer games with population-based reinforcement learning* ", Science, **364**, Issue 6443, pp. 859-865, (2019)

[7] Baker,. B., et al., "*Emergent tool use from multi-agent autocurricula* ", International Conference on Learning Representations, (2019)

[8] Zheng, S., et al., "*The AI Economist: Improving Equality and Productivity with AI-Driven Tax Policies* ", arXiv:2004.13332v1, (2020)

[9] Koul, A., "*ma-gym*", GitHub, `https://github.com/koulanurag/ma-gym`, (2021), Last Accessed November 2021.

[10] Achiam, J., "*OpenAI Spinning Up*", Accessible at `https://spinningup.openai.com/en/latest/index.html`, (2018), Last Accessed Septem

[11] Sutton, R.S., Barto, A.G., "*Reinforcement Learning: An Introduction Second Edition*", The MIT Press, (2018)

[12] Zhang, K., et al., "*Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms*", In: Vamvoudakis K.G., Wan Y., Lewis F.L., Cansever D. (eds) , "*Handbook of Reinforcement Learning and Control. Studies in Systems, Decision and Control*", **325** Springer, `https://doi.org/10.1007/978-3-030-60990-0_12`, (2021)

[13] Oliehoek, F.A., Amato, C., "*A Concise Introduction to Decentralized POMDPs*", Springer, (2015)

[14] Liping, Y., "*Model-free vs. Model-based Methods*", Deep learning Garden, `https://deeplearning.lipingyang.org/2016/12/29/model-free-vs-model-based-methods/`, Last Accessed November 2021.

[15] Mnih, V., et al., "*Playing Atari with Deep Reinforcement Learning*", arXiv:1312.5602, (2013)

[16] Mnih, V., et al., "*Human-level control through deep reinforcement learning*", Nature, **518** Issue 7540, pp. 529-533, (2015)

[17] Weng. L., "*Policy Gradient Algorithms*", lilianweng.github.io/lil-log, `https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#policy-gradient-theorem`, (2018), Last Accessed October 2021

[18] Schulman, J., et al., "*High-Dimensional Continuous Control Using Generalized Advantage Estimation*", arXiv:1506.02438, (2015)

[19] Silver, D., et al., "*Deterministic Policy Gradient Algorithms*", 31st International Conference on Machine Learning, ICML, **1**, (2014)

[20] Lillicrap, T. P, et al., "*Continuous Control With Deep Reinforcement Learning*", CoRR, arXiv:1509.02971v6, (2016)

[21] Nowe, A., et al., "*Game Theory and Multi-agent Reinforcement Learning*", Reinforcement Learning: State of the Art, Springer, (2012)

[22] Nash, J., "*Equilibrium Points in n-Person Games*", Proceedings of the National Academy of Sciences of the United States of America, 48–49 (1950)

[23] Hernandez-Leal P., et al., "*A Survey of Learning in Multiagent Environments: Dealing with Non-Stationarity*", A Survey of Learning in Multiagent Environments, arXiv:1707.09183v2, (2019)

[24] Ming, T., "*Multi-agent reinforcement learning: Independent vs. cooperative agents*", Proceedings of the tenth international conference on machine learning, pp. 330– 337, (1993)

[25] Tampuu, A., et al., "*Multiagent Cooperation and Competition with Deep Reinforcement Learning*", arXiv:1511.08779v1, (2015)

[26] Foerster, J., "*A Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning*", arXiv:1702.08887v3, (2018)

[27] Hausknecht, M., et al., "*Deep Recurrent Q-Learning for Partially Observable MDPs*", AAAI Fall Symposium Series, (2015)

[28] Lowe, R., "*Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments*", arXiv:1706.02275v4, (2020)

[29] Sunehag, P., et al., "*Value-Decomposition Networks For Cooperative Multi-Agent Learning*", arXiv:1706.05296v1, (2017)

[30] Oroojlooy, A., Hajinezhad D., "*A Review of Cooperative Multi-Agent Deep Reinforcement Learning*", arXiv:1706.05296v1, (2021)

[31] van Hasselt, H., et al., "*Deep Reinforcement Learning with Double Q-learning*", arXiv:1509.06461 , (2015)

[32] Formanek, J., "*rl-starter-kit*", GitHub, `https://github.com/jcformanek/rl-starter-kit/tree/main/06-Multi-Agent-Gym`, (2021), Last Accessed November 2021

[33] de Kock, R.J., "*marl-project-code*", GitHub, `https://github.com/RuanJohn/marl-project-code`, (2021)

[34] Canese, L., et al., "*Multi-Agent Reinforcement Learning: A Review of Challenges and Applications*", Appl. Sci., 11, 4948, (2021)