Multi-Agent Reinforcement Learning

Jeremy du Plessis

DPLJER001

University of Cape Town

Department of Mathematics & Applied Mathematics

Abstract

The central topic of this report is the exploration of Multi-Agent Reinforcement Learning algorithms and an analysis of their performance when applied to multi-agent, sequential social dilemma tasks in which agents must learn to cooperate in order to maximise long term return, where rewards are obtained by consuming shared, finite resources. The report is split into three sections. Part A covers the basics of single-agent reinforcement learning, outlining four of the most widely used and effective algorithms developed to date. Part B provides a detailed definition of multi-agent reinforcement learning, where the framework for multi-agent tasks is shown to be grounded in game theory, going on highlight some of the challenges faced by multi-agent learning algorithms, and finally and exploration of six recently proposed multi-agent learning algorithms which have shown promising results and received a large amount of attention from the research community. In closing, part C details an implementation of actor-critic variations of the six algorithms covered in part B, along with an analysis of the results obtained from their application to two sequential social dilemma MARL tasks, *Cleanup* and *Harvest*.

Part A: Single Agent Reinforcement Learning

1 Formal Definition of Single Agent Reinforcement Learning

In the context of single-agent reinforcement learning, a single agent interacts with an environment to accomplish a goal or a set of goals, which constitutes a task that is either finite or ongoing in nature. The success, or failure, of the agent is quantified by the relative magnitude of the numerical reward received over the duration of the task. In all single agent learning algorithms developed to solve such tasks, the learning agent aims to optimise a decision making policy, used to select actions which influence the state of the environment and alter the magnitude of the reward received at each of a sequence of discrete time steps, such that the accumulated reward over time is maximised. An illustration of the so-called 'agent-environment interface' [1] is shown in Figure 1. The single-agent reinforcement learning problem may be viewed as a *Markov Decision Process* (MDP). An MDP is a four tuple $(\mathcal{S}, \mathcal{A}, p, r)$, where $\mathcal{S}$ is the set of all possible environment states, $\mathcal{A}$ is the set of all possible

actions available to an agent[1], $p : S \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the state transition probability function, which gives the probability distribution over possible next states given a current state and action, and $r : S \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function, which produces a numerical reward given a state and an action selected by an agent.

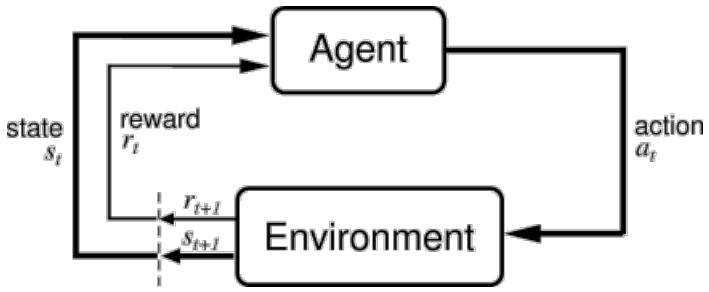


Figure 1: The agent-environment interface [1]. At each time step the single agent observes the environment state and performs an action based off a decision making policy, receiving a numerical reward in return.

The agent interacts with the environment over time, where at each discrete time step t the agent observes the state of the environment $s_t \in \mathcal{S}$, and responds according to its decision making policy $\pi : S \times A \to [0, 1]$ by sampling from a multinomial distribution of actions over the current state, selecting an action $a_t \sim \pi(\cdot|s_t)$ to execute in the environment. In turn the agent receives a numerical reward r_{t+1}, and an updated state s_{t+1}. The goal of the agent is to maximise its return, the sum of discounted rewards following time t, $R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$, over some time horizon, where $\gamma \in [0, 1]$ is a temporal discount factor which determines how myopic (concerned with rewards received in the immediate future) or not (concerned with rewards obtained over the long run) the agent is. In order to accomplish this goal, the agent must iteratively improve its decision making policy such that the optimal action is selected at each time step so as to maximise rewards over time. In order to iteratively improve π over time, the agent needs a way to quantify the quality of it's action choices and/or the quality of the state it finds itself in as a consequence of it's actions. This leads directly to the concept of the *state value*, the return observed following a specific state under a specific policy, and the *action value*, the return observed after taking an action in a specific state under a specific policy. Formally we define state values in terms of the state-value function, which computes the expected return following a state s observed at some time t:

$$V^{\pi}(s) = E[R_t|s_t = s]$$

Similarly, we define the action value in terms of the action-value function, which computes the expected return following an action a, taken in a state s, at some time t:

$$Q^{\pi}(s, a) = E[R_t|s_t = s, a_t = a]$$

Notice that both the state-value and action-value functions are dependant on the decision-making policy π, which makes sense since the action choices determine the trajectory of the agent through the space of all states and actions, $\mathcal{S} \times \mathcal{A}$. The vast majority of modern reinforcement learning algorithms are based, in one way or another, on the (implicit or explicit) learning of value functions. In fact, SARL algorithms which rely on the learning of value functions as a means of policy improvement may be described as using a form of generalised policy improvement [1], illustrated in Figure 2, wherein an agent uses it's policy π to collect experience which

[1]sometimes the set of actions available to an agent is constrained by the state of the environment at a given time, $s_t \in S \implies A = A(s_t)$ for each time t

is used to better approximate it's value function V^{π} (or Q^{π}) under that policy, whereafter the agent acts in a greedy manner with respect to the policy in order to improve it.

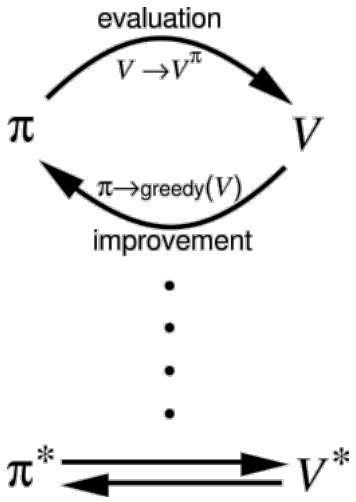


Figure 2: Generalised Policy Improvement [1]. An agent collects experience to estimate the value function, then improves the policy by acting in a greedy manner, creating a cycle of improvement converging (ideally) on an optimal policy and value fucntion.

Of course, a change to the policy means a change to the value function, and so the cycle of *policy evaluation* and *policy improvement* continues, until some sort of convergence is reached. Ideally we would like to converge on an optimal policy, denoted by π^*, and an optimal value function, denoted as V^* and Q^*. The recursive definitions for the optimal value functions are given below:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad \forall s \in \mathcal{S} \tag{1}$$

$$Q^*(s,a) = E\left[r_{t+1} + \gamma V^*(s+1) | s_t = s, a_t = a\right] \tag{2}$$

$$= E\left[r_{t+1} + \gamma \max_{a'} Q^*(s+1, a') | s_t = s, a_t = a\right] \tag{3}$$

Where we can define the optimal policy π^* as the policy under which the expected value over all states and/or actions in the state and/or action space is maximised (i.e. the optimal sequence of decisions are made following each state and/or action such that the return is as large as possible on average). One of the most important points to note about convergence in SARL is that it depends on the stationarity of the distribution over states, defined by the state-transition probability function p of the underlying MDP, with respect to which a single agent would learn a decision-making policy. When p defines a stationary distribution, the agent can learn expected values arising from expected outcomes from actions selected in observed states and, given certain conditions are satisfied (specifically that all states may be visited by the agent an infinite number of times), in the limit of experience gained, certain single-agent learning algorithms can offer mathematically provable convergence guarantees (to optimal policies and/or value functions) [1]. As is shown below, designing multi-agent learning algorithms for which such convergence is provable (or even practically achievable) is much harder, due to the non-stationarity that arises as a result of multiple agents interacting based off individual, dynamic policies [6].

2 (Classic) Single-Agent RL Algorithms (TBC)

What follows is an explanation of, arguably, the four most well-established SARL algorithms in the literature to date. While many (or perhaps nearly all) of the record results on the current benchmark SARL tasks have been achieved using variations and/or extensions of these algorithms, the essence of those algorithms can be understood by examining the four core algorithms as they are outlined below. A final note is that, while many RL tasks have discrete state spaces and thus may be implemented using tabular methods, we will focus on deep learning implementations wherein approximations to value functions and/or policies are assumed to be parameterised deep neural networks.

2.1 DQN

The first single-agent deep reinforcement learning algorithm to be outlined is the classic Deep Q-Network (DQN) algorithm, proposed in 2013 in the paper *"Playing Atari with Deep Reinforcement Learning"* [2], based on the tabular Q-learning algorithm developed by Christopher Watkins in 1989. We consider a single agent interacting with an environment defined by a Markov decision process $(\mathcal{S}, \mathcal{A}, p, r)$. We assume the state space $\mathcal{S}$ is continuous, and that the space of possible actions $\mathcal{A}$ available to the agent is discrete. Under DQN, as in regular Q-learning, the agent maintains an approximation of an action-value function - a so-called Q-function - $Q_\theta : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ which, instead of being a table that stores discrete value approximations for state-action pairs, is a function approximator parameterised by $\theta \in \mathbb{R}^m$ for some integer $m >> 1$. For simplicity, it is assumed from here on that the parameterised Q-function is a simple feed-forward, multi-layered deep neural network, having input and output layers the same dimension as the real-valued vector representing the state of the environment, and the number of possible discrete actions, respectively. The reason for the move towards using deep neural networks to approximate value functions is simply because the use of tabular methods for Q-learning is limited to tasks where the action space is discrete (or in some cases continuous, but simple enough to be discretised). The overall strategy underpinning the DQN algorithm is simple: have the agent collect experience in the form of *(state, action, next-state, reward)* transition tuples by interacting with the environment and store the tuples in a memory buffer. Then, at each time step, learn from the collected experience by sampling random mini-batches from the buffer which are used in computing and applying gradients of a TD-error loss function, which uses a bootstrapping technique based on the bellman optimality equation shown in equation (3), in order to iteratively improve the approximation $Q(s, a; \theta) \approx Q^*(s, a)$. Specifically, the process of experience collection and training happens as follows. Initially the parameters of the Q-network are initialised randomly as $\theta^{(0)}$. At each time step, the agent observes the environment state $s_t \in \mathcal{S}$, and selects an action according to an $\epsilon-$greedy policy; with probability $\epsilon << 1$, the action a_t is drawn randomly from the set of possible actions in $\mathcal{A}$, otherwise, with probability $1 - \epsilon$, a_t is selected according to $\arg\max_a Q_(s_t, a; \theta^{(k)})$, which selects the action with the highest value-estimate given s_t, based on $\theta^{(k)}$, the value of the parameters after the k^{th} parameter update. The agent then executes the action in the environment, receiving a numerical reward $r_t = r(s_t, a_t)$, generated by the environment reward function, along with the updated state s_{t+1}. The experience tuple $(s_t, a_t, r_{t+1}, s_{t+1})$ from each time step t is collected and stored in a memory buffer $\mathcal{B}$. During training, which occurs once per time step, a random mini-batch of experience $B \sim \mathcal{B}$ is drawn from the memory buffer, and each transition tuple $(s_j, a_j, r_{j+1}, s_{j+1})$ in B is used to perform an optimization step with the objective of minimising the following loss function:

$$\mathcal{L}_k(\theta^{(k)}) = \mathbb{E}_{a,s\sim\rho(\cdot)}\left[(y - Q(s,a;\theta^{(k)}))^2\right] \tag{4}$$

Where $\rho(s,a)$ is, what the authors term, the "behaviour distribution"; the joint probability distribution which determines agent trajectories through the state-action space, which is in turn determined by (i) $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$, the state-transition probability distribution function, (ii) the current parameters $\theta^{(k)}$ of the Q-network which determine action selection, and (iii) ϵ. Notice that since the the loss function $\mathcal{L}_k(\theta^{(k)})$ depends on the current parameter vector, thus both are indexed with the iteration number k. Then, y in equation (4) is the DQN bootstrap target, which computed from a single transition tuple in B as:

$$y = r_{j+1} + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^{(k-1)}) \tag{5}$$

where $Q(s,a;\theta^{(k-1)})$, the target network, is a time-delayed copy of the principle action-value network, updated every C steps during training. The reason for using a time-delayed copy of the Q-network to compute the target y is because the correlations that would arise naturally when attempting to use value estimates, determined by $\theta^{(k)}$, in the loss function used to update $\theta^{(k)}$ cause learning instability, which is alleviated by the use of the target network. Finally, in order to update the parameters of the Q-network, the gradient of the loss function (4) is computed as:

$$\nabla_\theta \mathcal{L}_k(\theta^{(k)}) = \mathbb{E}_{a,s\sim\rho(\cdot),s'\sim p(\cdot)}\left[\left((r + \gamma \max_{a'} Q(s', a'; \theta^{(k-1)})) - Q(s,a;\theta^{(k)})\right)\nabla_\theta Q(s,a;\theta^{(k)})\right] \tag{6}$$

Where the expectation is typically approximated during training by computing and averaging out the inner expression over the entire batch B of experience. The approximate gradients computed over the batch may be applied directly to update the Q-network parameters, but are typically applied using a gradient descent algorithm like *Stochastic Gradient Descent* (SGD), or the well-known *Adam* algorithm. The main point to note about the DQN algorithm is that the target (5) has the same form as the bellman optimality equation (3), and that actions are also selected during experience collection (with probability $1 - \epsilon$) according to the maximum value estimate over actions. This gives us the same theoretical convergence guarantee of $Q(s,a;\theta^{(k)}) \rightarrow Q^*$, as $k \rightarrow \infty$ shown in [1], given that the same conditions are satisfied. Ultimately, the learning of the optimal value function amounts to, as mentioned above, the learning of the optimal decision making policy, since the agent will choose, in any given state, actions with the highest expected value. A beautiful illustration of the values over a continuous state space, taken from the letter *"Human-level control through deep reinforcement learning"* published by the DQN authors in the Journal Nature in 2015, is shown in Figure 3. The image shows a 2D latent space projection (using t-SNE dimensionality reduction) of the continuous state (pixel) space of the Atari game "Space Invaders" where points representing similar states are clustered together, and each point is colored according to it's respective state-value estimate, as determined by a DQN agent. The full DQN algorithm is given below.

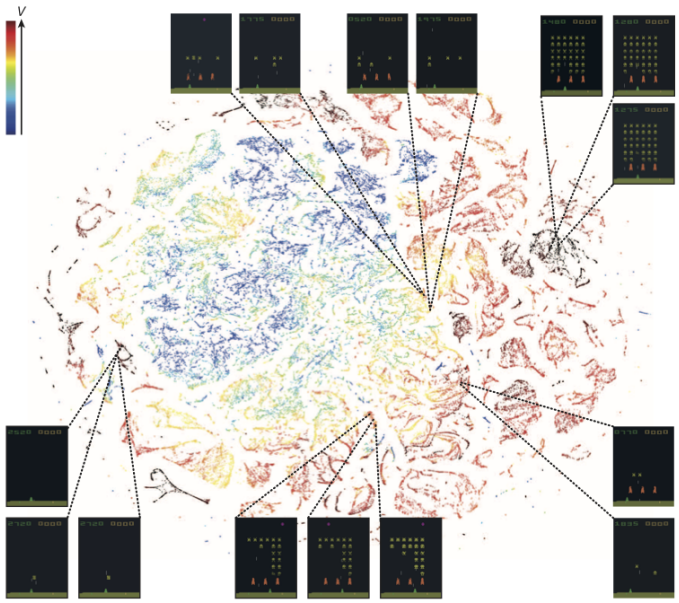


Figure 3: A 2D latent space embedding of the state (pixel) spce of the Atari Game "Space Invaders", where points representing similar states are clustered together and each point is colored according to it's state value estimate, determined by a DQN agent after 2-hours of game play. Source:"Human-level control through deep reinforcement learning", Journal Nature, 2015.

Algorithm Deep-Q Learning with Experience Replay

Parameters: ϵ, γ, C;

Initialise: $\mathcal{B} \leftarrow \emptyset$;

Initialise the principle and target Q-networks with the same random weights;

for *episode =1 to M* **do**

observe s_0;

for *t=1 to T* **do**

With probability ϵ select a random action a_t;

Else select $a_t = \arg\max_a Q_(s_t, a; \theta)$;

Execute a_t and observe s_{t+1}, r_{t+1};

Store experience in memory buffer $\mathcal{B} \leftarrow \mathcal{B} \cup (s_t, a_t, r_{t+1}, s_{t+1})$;

Sample random mini-batch $B \sim \mathcal{B}$;

For each tuple $(s_j, a_j, r_{j+1}, s_{j+1})$ in B set:

$$y_j = \begin{cases} r_{j+1} & \text{if } s_{j+1} \text{ is terminal} \\ r_{j+1} + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^{(k-1)}) & \text{otherwise;} \end{cases};$$

Update $\theta^{(k)}$ using the gradient $\nabla_\theta \mathcal{L}_k$ as per equation 6;

Every C steps set $\theta^{(k-1)} \leftarrow \theta^{(k)}$ and increment k by 1;

end

end

2.2 Classic Policy Gradient (REINFORCE)

While REINFORCE is in fact a class of reinforcement learning algorithms proposed in 1989 by Ronald J. Williams in his paper titled *"Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning"*, when one hears references to *the* REINFORCE algorithm in the RL community, it is probably in reference to the Policy-Gradient algorithm proposed by Richard Sutton et. al in the 1992 paper *"Policy Gradient Methods for Reinforcement Learning with Function Approximation"* [3] which applied REINFORCE algorithms in developing a new class of single-agent learning methods, widely popularised in the last decade, called Policy-Gradient Methods. Here, REINFORCE is taken to mean the simplest in this class of methods. In the DQN algorithm covered above, the idea was to learn the action-value function explicitly, approximating it with a deep-neural network, and select actions according to an $\epsilon-$greedy policy. In REINFORCE, the idea is to learn an explicit policy $\pi_\theta : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, taken to be a deep-neural network parameterised by $\theta \in \mathbb{R}^m$ for some integer $m >> 1$, which gives a probability distribution of actions over states in a Markov Decision process $(\mathcal{S}, \mathcal{A}, p, r)$, which is used to define the interaction between an agent and an environment as per usual. The key distinction between DQN and REINFORCE is that while the Q-network in DQN explicitly approximates the action-value function, in REINFORCE the output of the parameterised policy network $\pi_\theta(a|s)$ is conditioned directly on rewards observed by the agent when interacting with the environment, causing it to learn the relative value of actions implicitly (as it outputs probabilities, not value estimates). As per usual, the objective is for the agent to learn an optimal policy, which in this case translates to an optimal mapping of states from a continuous state space $\mathcal{S}$ to a multinomial probability distribution over discrete actions in the set $\mathcal{A}$, such that the expected return under π_θ over all states and actions $(\mathcal{S} \times \mathcal{A})$ is maximised. The strategy in REINFORCE is then to construct an objective function which encapsulates this goal explicitly:

$$J(\theta) = \mathbb{E}_{\tau \sim \rho_\theta(\cdot)} \left[R(\tau) \right] \tag{7}$$

Which, in words is the expected return over all trajectories τ through the state action space, $\mathcal{S} \times \mathcal{A}$, given some fixed policy parameters θ. Here, a trajectory τ is defined as a distinct sequence of states and actions observed and chosen by an agent, respectively, when interacting with an environment: $s_0, a_0, s_1, a_1, \ldots, s_{T-1}, a_{T-1}, s_T$, where s_T is a terminal state. The probability distribution $\rho_\theta(\tau)$ determining the probability of a given trajectory through the state-action space under θ is the same "behaviour distribution" mentioned above in the description of the DQN algorithm; it is simply a joint probability distribution determined by the agents policy $\pi_\theta(a|s)$, used to determine action selection probabilities, and the state transition probability function $p(s'|s, a)$, which determines the probability of transitioning to a resultant state s' given the current state s and the chosen action a. Now, in it's present form the objective function $J(\theta)$ is not tractable (more on that in a moment), so the strategy will be for the agent to collect enough experience, by interacting with the environment (as in DQN), in order to approximate $J(\theta)$, compute it's gradient and perform the optimization step:

$$\theta^{(k+1)} = (1 - \alpha)\theta^{(k)} + \alpha \nabla_\theta J(\theta^{(k)}) \tag{8}$$

where $\alpha \in (0, 1)$ is a step size parameter. Updating the parameters θ in this way should theoretically lead to more optimal agent behaviour, i.e. action-selection which increases the expected return over all possible trajectories. That is, as with DQN, if the right conditions (as outlined in [1]) are satisfied we should get convergence

$\lim_{k\to\infty} \pi(s,a;\theta^{(k)}) \to \pi*$. In order to compute an approximation to the gradient of the the objective function with respect to the policy parameters, notice that we can take the gradient operator inside the expectation in equation (7) in the following way:

$$\nabla_\theta J(\theta) = \nabla_\theta \int_\tau R(\tau)\rho_\theta(\tau)d\tau \tag{9}$$

$$= \int_\tau R(\tau)\nabla_\theta \rho_\theta(\tau)d\tau \tag{10}$$

$$= \int_\tau R(\tau)\frac{\nabla_\theta \rho_\theta(\tau)}{\rho_\theta(\tau)}\rho_\theta(\tau)d\tau \tag{11}$$

$$= \int_\tau R(\tau)\nabla_\theta log\rho_\theta(\tau)\rho_\theta(\tau)d\tau \tag{12}$$

$$= \mathbb{E}_{\tau\sim\rho_\theta(\cdot)}\left[R(\tau)\nabla_\theta log\rho_\theta(\tau)\right] \tag{13}$$

Next, consider the expansion of the behaviour distribution ρ_θ for a given trajectory τ, which is just the product of probabilities for each state and action in the sequence:

$$\rho_\theta(\tau) = p_0(s_0)\pi_\theta(a_0|s_0)p(s_1|s_0,a_0)\pi_\theta(a_1|s_1)\ldots\pi_\theta(a_{T-1}|s_{T-1})p(s_T|s_{T-1},a_{T-1}) \tag{14}$$

$$= p_0(s_0)\prod_{t=0}^{T}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t,a_t) \tag{15}$$

where p_0 gives the probability distribution over starting states, defined implicitly by the environment. Now, it is not possible to compute the gradient of $p(s'|s,a)$ since it is not explicitly defined, but expanding the gradient portion of equation (13) we see that:

$$\nabla_\theta log\rho_\theta(\tau) = \nabla_\theta log\left[p_0(s_0)\prod_{t=0}^{T}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t,a_t)\right] \tag{16}$$

$$= \sum_{t=0}^{T}\nabla_\theta log\pi_\theta(a_t|s_t) \tag{17}$$

Then, finally since $\mathbb{E}_{x\sim P(\cdot)}\left[f(x)\right] = \lim_{N\to\infty}\frac{1}{N}\sum_{j=1}^{N} f(x_j)_{x_j\sim P(\cdot)}$, we can use this fact and equation (17) to rewrite an approximation to equation (13) as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\rho_\theta(\cdot)}\left[R(\tau)\nabla_\theta log\rho_\theta(\tau)\right] \tag{18}$$

$$\approx \frac{1}{N}\sum_{j=1}^{N}\left[\sum_{t=1}^{T}\left(\nabla_\theta log\pi_\theta(a_t|s_t)(\sum_{k=t}^{T}\gamma^{k-t}r_{k+1})\right)\right] \tag{19}$$

$$= \frac{1}{N}\sum_{j=1}^{N}\left[\sum_{t=1}^{T}\nabla_\theta log\pi_\theta(a_t|s_t)R_t\right] \tag{20}$$

where $N >> 1$ is the number of trajectories sampled from having the agent interact with the environment, and T is the number of time steps in each trajectory. The REINFORCE algorithm is given in full below.

Algorithm REINFORCE (Classic Policy Gradient)

Parameters: γ, α;

Initialise: $\mathcal{B} \leftarrow \emptyset$;

for *k=0,1,2,...* **do**

1. Collect trajectories $\mathcal{B}_k = \{\tau_j\}_{j=1}^{N}$, consisting of tuples $(s_t, a_t, r_{t+1}, s_{t+1})$ for each t in each τ_j;
2. Compute returns R_t for each time step t in each $\tau_j \in \mathcal{B}_k$;
3. Compute $\nabla_\theta J(\theta^{(k)}) \approx \frac{1}{N} \sum_{j=1}^{N} \left[\sum_{t=1}^{T} \nabla_\theta log\pi(a_t|s_t;\theta^{(k)}) R_t \right]$;
4. Update the policy parameters $\theta^{(k+1)} \leftarrow (1-\alpha)\theta^{(k)} + \alpha \nabla_\theta J(\theta^{(k)})$;
5. Reset $\mathcal{B} \leftarrow \emptyset$;

end

2.3 Advantage Actor Critic (A2C)

Naturally, the range over which observed rewards fluctuate varies widely from task to task. For some tasks rewards are normalised to be within the range [0,1], but other tasks may not have this normalisation built in. As a result, some observed rewards may translate to distributions of returns (the sum of discounted rewards) with very high variance (i.e. in comparison to a variance of 1). In the context deep learning generally, a number of strategies for improvement in performance involve normalisation; both training data and labels are often normalised to be within the range [0,1] and activation functions are designed to prevent gradients and activation from growing (or shrinking) excessively (»1). In the context of reinforcement learning and policy gradient algorithms in particular, performing policy optimization using using unbounded, unnormalised returns can result in high-variance gradient estimates which in turn translates to high-variance parameter updates to the policy parameters, and hence instability in learning, which is undesirable. Reducing learning instability is a major theme in deep reinforcement learning, and many of the breakthroughs in learning algorithms have contributed some technique to the literature to enable higher stability in learning. The *Advantage Actor Critic* (A2C) algorithm, a simple variant of the REINFORCE algorithm also covered in [3], is one such algorithm. A2C is based on actor-critic single agent learning algorithms in general, the first of which were tabular methods (see [1]). The simple idea is that, instead of defining the objective function $J(\theta)$ as the expected (unnormalised) return, which is known to lead to high variance gradient estimates, the objective function should be redefined as:

$$J(\theta) = \mathbb{E}_{\tau \sim \rho(\cdot)} \left[R(\tau) - b(\tau) \right] \tag{21}$$

where $b(\tau)$ is a real-valued baseline computed for each return $R(\tau)$. Now, in order to approximately normalise an empirical distribution (a sample of numerical observations) the strategy is usually to subtract the sample mean and divide by the sample variance, which may be applied to distributions of returns generated during experience collection in RL tasks in order to reduce variance between sample distributions. Thus we would like the baseline $b(\tau)$ to be something akin to the mean of the returns computed during training, and since the mean is likely to change over time (hopefully the agent improves it's policy as it learns), we would like the baseline to adjust accordingly. This is accomplished in A2C by choosing the baseline to be state state-value function, which for any state s is defined as $V(s) = \mathbb{E}[R_t|s_t = s]$, which should in theory serve as the perfect normalising factor

for the empirical return R_t following s. Concretely, the strategy is to have the agent learn, in addition to it's parameterised policy π_θ, a parameterised state-value function approximator $V_\omega : \mathcal{S} \to \mathbb{R}$; again, a deep neural network, which may be trained in a supervised manner on the reward data collected in each set of trajectories. The new formula for the estimating the gradient of the objective function in A2C will be:

$$\nabla_\theta J(\theta) =\approx \frac{1}{N}\sum_{j=1}^{N}\left[\sum_{t=1}^{T}\nabla_\theta log\pi(a_t|s_t;\theta^{(k)})\hat{A}_t\right] \tag{22}$$

Where $\hat{A}_t = R_t - V(s_t;\omega)$ is the the so-called advantage estimate, and computes the difference between the empirical return R_t generated by an agent following some state s_t and the expected return computed by the state-value network $V_\omega(s_t)$. Under A2C V_ω is optimised, usually through gradient descent, by minimising the loss function:

$$\mathcal{L}(\omega) = \mathbb{E}_{s\sim\rho(\cdot)}[(y - V(s;\omega))^2] \tag{23}$$

Where $y = R_t$ is simply the return following the state s.

2.4 Proximal Policy Optimization (PPO)

Further to the challenge of instability in deep reinforcement learning, and deep learning in general, is the phenomenon known as "catastrophic forgetting"; when performing gradient optimization on the parameters of a policy under a normal policy gradient algorithm (e.g. A2C), or value function under DQN, since the parameter space is so large (sometimes on the order of 10'000 parameters or more) it can sometimes happen that a gradient step may occur too much in one direction, in a portion of the parameter space where the gradients are very steep, and as a result one may observe a sharp, nearly immediate drop in performance. This phenomenon is called catastrophic forgetting since the agent appears to forget - almost instantly - the behaviour it has learned over the course of training. The reason this occurs has to do with the "shape" of the surface as defined by the objective (or loss) function in policy parameter space; when moving through the parameter space during training, by taking steps in the direction of the gradient, it is possible to move too far in the direction of steepest ascent and end up falling off a cliff (in the parameter space). Proximal Policy Optimization (PPO), which is actually a family of RL algorithms first proposed in the 2017 paper *"Proximal Policy Optimization Algorithms"* [4] by researchers from OpenAI, was developed to solve exactly this problem [2]. PPO is, as with A2C, a simple variation of the classic policy gradient algorithm, developed with the objective of designing an algorithm which would allow for more stable policy optimization through constraining the **magnitude** of gradient updates at each step, with the ultimate goal being to prevent gradient updates which results in too great a step in parameter space, ending up with a policy which produces instantly different (and undesirable) behaviour. The goal of limiting the magnitude of updates to the parameters of a policy in achieved under PPO by iteratively performing the optimization:

$$\theta^{(k+1)} = \arg\max_\theta \mathbb{E}_{s,a\sim\rho(\cdot)}\left[J^{CLIP}(\theta, s, a; \theta^{(k)})\right] \tag{24}$$

[2] In fact, PPO takes inspiration from a much more complicated algorithm called Trust-Region Policy Optimization (TRPO)

In words, the goal is to perform iteravtive updates such that the PPO objective J^{CLIP} is maximised with respect to θ. Notice, too that in each optimization the fixed parameter vector $\theta^{(k)}$ from the previous (k^{th}) iteration is used. Note too that the optimization is typically performed, with a modern optimization method such as Adam or SGD. The PPO objective is defined as:

$$J^{CLIP}(\theta, s, a; \theta^{(k)}) = \min\left(r(s,a;\theta)\hat{A}^{\pi_{\theta^{(k)}}}(s,a), clip\left(r(s,a;\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}^{\pi_{\theta^{(k)}}}(s,a)\right) \tag{25}$$

with

$$r(s,a;\theta) = \frac{\pi(a|s;\theta)}{\pi(a|s;\theta^{(k)})} \tag{26}$$

where $\epsilon << 1$ (say 0.2) and both $\pi_{\theta^{(k)}}$ and $\hat{A}^{\pi_{\theta^{(k)}}}$, each computed before optimization occurs, remain fixed quantities throughout the optimization process. The first thing to be aware of with respect to this new objective function is *how* it is used in the optimization process. During training, the expectation in equation (24) would, as usual, be approximated over a batch of experience. The goal of the optimization is to maximise the approximate expectation of J^{CLIP} with respect to θ, but to ensure that the magnitude of the derivative with respect to θ is bounded, both above and below. This is achieved in the following way. Consider a state-action pair (s, a) from a batch of experience collected by under $\theta^{(k)}$ and the ratio $r(s,a;\theta) = \frac{\pi(a|s;\theta)}{\pi(a|s;\theta^{(k)})}$ from the PPO objective function. Now, if for s and a we have $\hat{A}^{\pi_{\theta^{(k)}}}(s,a) > 0$, then the optimization process for that single observation would result in an increase in $r(s,a;\theta)$, since we would want the agent to perform a when observing s more often on average. However, to ensure that the change is not too large, an upped bound of $(1+\epsilon)\hat{A}^{\pi_{\theta^{(k)}}}(s,a)$ is placed on the value of of J^{CLIP}. The result of this upper bound is that, at a certain point in the optimization, the gradients of J^{CLIP} with respect to θ for state-action pairs similar to (s, a) will become zero, or very close to zero, and thus prevent any further gradient updates to θ in the direction in parameter space induced by (s, a). Similarly, if $\hat{A}^{\pi_{\theta^{(k)}}}(s,a) < 0$ for some (s, a), the optimization process would attempt to decrease $r(s,a;\theta)$, but again, having a (different) upper bound of $(1-\epsilon)\hat{A}^{\pi_{\theta^{(k)}}}(s,a)$, that is, the negative value would be prevented from getting too close to zero.

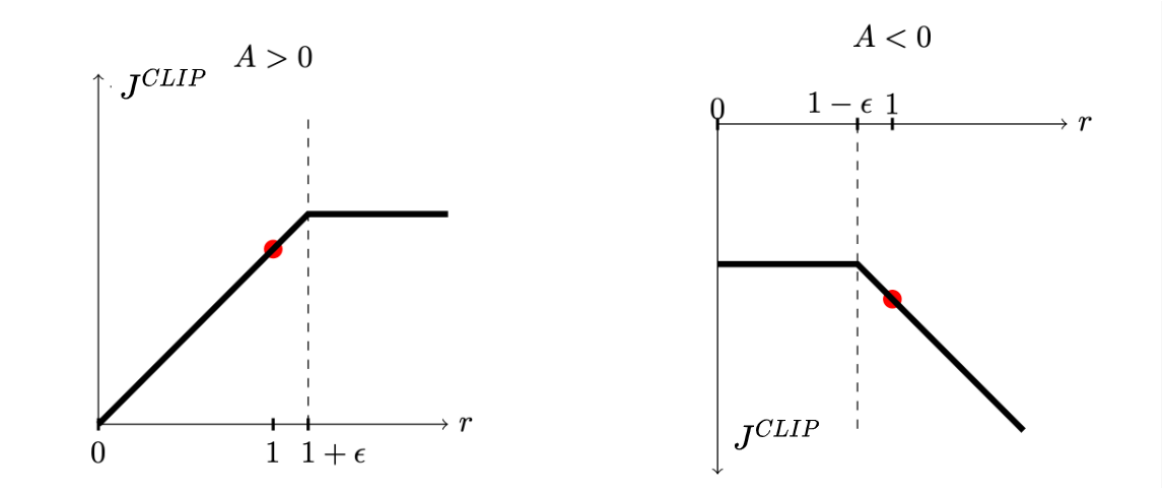


Figure 4: The PPO objective J^{CLIP} as a function of the ratio $r(\theta)$ which defines the magnitude of the change of the agent policy during the optimization process.

Figure 4 illustrates the effect of the clipping function on the objective; the red dot indicates $r(\theta) = 1$, which is the starting point for every optimization step during training. The important part to notice is that as $r(\theta)$ changes, for both positive and negative values of the advantage estimate, the value of J^{CLIP} is bounded and thus the derivative with respect to θ will be driven to zero if the magnitude of change in the parameters relative to $\theta^{(k)}$ is too extreme. This simple change to the objective function often yields great performance improvement on tasks where simpler policy gradient methods struggle to perform [4].

3 Motivation for Multi-Agent RL

Single-agent reinforcement learning has been successfully applied to a wide variety of tasks including single player Atari games, robotics control, recommender systems (applied in e-commerce), and more. However, the need for multiple agents, and multi-agent learning algorithms, is apparent when considering some of the tasks in which single-agent learning algorithms struggle to perform. The first category of tasks, broadly speaking, are those which require physically decentralised control [5]. These are the most obvious, since control of a distributed system over a network is often subject to delays in data transmission which are unacceptable for the task. A simple example of such a task is the control of phase for a networked grid of traffic lights, where multiple actions must be selected at each of a set of discrete time steps, each of which determines the phase of a single traffic light, with the joint objective being to reduce the density of traffic per unit of time. The second category of tasks are those tasks in which there exist multiple, possibly competing, objectives [5]. An example of this would be control over the distribution of a crucial, finite resource to multiple consumers, like water stored in a reservoir which needs to be shared amongst multiple towns. In such a scenario we would like a system which would be able to find a point, or points, of equilibrium where the distribution strategy may not be optimal for any one consumer, but may be 'as good as possible' for all consumers collectively. A third category of tasks which may require multi-agent control are those which, while they may be suited to centralised control, may have action and/or state spaces which are simply so large that single agent learining becomes computationally inefficient or intractable. An example of this may be an industrial plant where control parameters number in the tens of thousands, in which case splitting the plants process up into several sub-processes, turning the task into a decentralised one which may be controlled by multiple agents in a networked system may be far more effective. While there may be many other tasks more suited to multi-agent, rather than single-agent control, which are not mentioned here, the objective of highlighting these categories is simply to motivate for the development of multi-agent learning algorithms in general, which is what is explored below.

Part B: Multi-Agent Reinforcement Learning

4 Game Theory Foundations

The theory of games was explicitly designed for reasoning about multi-agent systems, and was developed by von Neumann and Morgenstern in 1947. The history of multi-agent reinforcement learning (MARL) is rooted in game theory, which in it's early development focused purely on competitive games, but over time has been developed into a framework for analysing strategic interactions in general between multiple players [5]. Formally, a *game* is "a mathematical object which describes the consequences of interactions between players *strategies* in terms of individual payoffs."[5]. What follows is a brief description of the game theoretic foundations of MARL, beginning with normal form games, which are games including only environments which are stateless or static; followed by a classification framework for multi-agent games based on their payoff (reward) structure; and finally, a discussion and formal definition of Markov games, which are sequential games including environments which are stateful or dynamic (states change in response to actions), and constitute the foundational framework used to describe the main MARL algorithms covered in this report.

$\sim$

Normal form games. A *normal form game* is a multi-player game in which a stateless (or single state, depending on how you view it) environment is defined, and in which n participants, called *players*, simultaneously select an action and receive a distinct payoff, or reward, which is determined by the set of all n actions [3]. A normal form game played once-off is sometimes referred to as a *static game*, whereas normal form games played in succession are sometimes referred to as *repeated games*. Notice that the description of a normal form game for multiple players is similar to the single-player bandit task described in [1]. Formally a normal form game is a tuple $(n, \{\mathcal{A}_i, r_i\}_{i=1}^n)$ where n is the number of players; $\mathcal{A}_i$ is the set of actions available to player i, such that $\mathcal{A} = \times_{i=1}^n \mathcal{A}$ is the set of joint actions and; $r_i : \times_{i=1}^n \mathcal{A}_i \to \mathbb{R}$ is the individual reward function for player i which computes the expected payoff they received after a joint action $\mathbf{a} \in \times_i \mathcal{A}_i$ is executed. Then, similar to a policy in SARL, a strategy $\sigma_i : \mathcal{A}_i \to [0, 1]$ belonging to agent i defines a probability distribution over actions, meaning that the process of choosing an action is stochastic, just as the SARL policy gradient algorithms defined above (REINFORCE, A2C and PPO) make use of stochastic policies. Similarly to a policy, a strategy determines the expected reward received by an agent in a normal game, and all multi-agent learning algorithms will have the objective of finding an optimal strategy for each agent which maximises the reward received from a single game, or a sequence of repeated games. A strategy is called *pure* if $\sigma_i(a) = 1$ for some $a \in \mathcal{A}_i$, otherwise it is called a *mixed strategy*. Since the strategies for each player are stochastic, we can work out the expected payoff for each player, given a fixed *joint strategy* $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_n)$. The expected payoff for player i, given a strategy profile $\boldsymbol{\sigma}$ is:

$$E\left[r_i(\boldsymbol{\sigma})\right] = \sum_{\mathbf{a} \in \mathcal{A}} r_i(\mathbf{a}) \left(\prod_{j=1}^{n} \sigma_j(a_j) \right)$$

where a_j is the action for player j in the joint action $\mathbf{a}$. Finally, note that normal form games involving n players may be represented as an n-dimensional *payoff matrix* in which each entry is an n-tuple specifying the reward received by each agent for each joint action.

[3] Normal form games may be contrast against *extensive form games* in which agents take turns to select and perform actions.

~

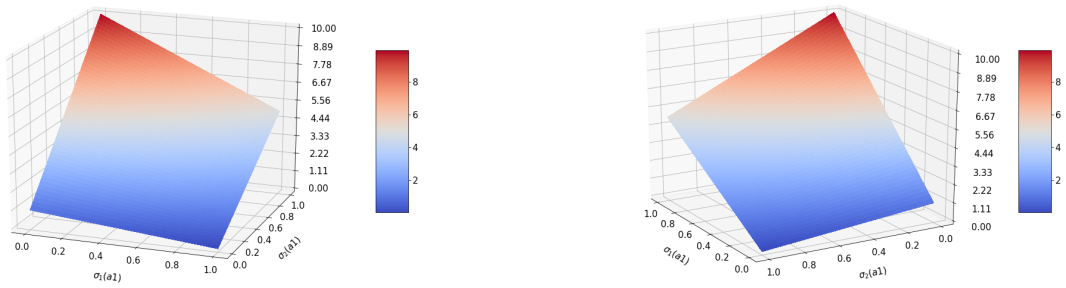
		Player 2	
		$a1$	$a2$
Player 1	$a1$	$(5, 5)$	$(0, 10)$
	$a2$	$(10, 0)$	$(1, 1)$

Table 1: A payoff matrix illustrating the well-known prisoner's dilemma. Action $a1$ is the decision not to confess, where action $a2$ is the decision to confess; the Nash equilibrium is the joint action ($a2$, $a2$).

Example: the prisoner's dilemma. Consider the following well-known example, the prisoner's dilemma, which is a 2-player normal-form game defined by the payoff matrix in Table 1. The prisoner's dilemma is an example of a so-called *social dilemma* in game theory, which may be defined informally as "a task in which the non-cooperative payoff for a player exceeds the cooperative payoff. However, if most players in the game fail to cooperate, all players suffer."[4]. Later on in the report we explore the results of applying multi-agent reinforcement learning algorithms to so-called *sequential social dilemmas*, which will be defined in more detail in a later section. The prisoner's dilemma is illustrated by the following story. Suppose two conspiring criminals, call them criminal 1 and criminal 2, are caught red-handed committing a minor crime by the police, and suppose both criminals are suspected conspirators in a much more serious crime, other than the one for which they have just been apprehended. Both criminals are taken to the police station and held in separate interrogation rooms, wherein they are interrogated simultaneously by a pair of officers who have the objective of getting either one, or both of the criminals to confess to the major crime. The criminals are prohibited from communicating with one another. Both criminals are given two options: (a1) do not confess to the major crime and, (a2) confess to the major crime. The interrogating officers then continue to communicate to each criminal the four potential outcomes, arising from the four possible combinations of action choices: (a1,a1) both criminals do not confess, in which case they are both prosecuted for the minor crime for which they were arrested, for which the penalty will likely be a 1-year jail sentence (say this is a relative payoff off +5 for each criminal); (a1,a2) and (a2,a1) criminal 1 confesses and criminal 2 does not, or visa-versa, in which case the criminal who confesses gets off with a fine - a reward for aiding the investigation - and the criminal who does not confess gets a 10-year jail sentence (say this is a relative payoff of 10 and 0, respectively); (a2,a2) both criminals confess and are sentenced to 5 years in prison each (say this is a relative payoff of +1 each). Now, the dilemma faced by each criminal is whether or not to confess, given that the other may flip; it is obvious that to not confess is poses the highest risk as it exposes the confessor to potential betrayal, and a 10 year jail sentence. On the other hand, if both confess they won't do as well as they could have done, but they will have the certainty of avoiding the 10 year jail sentence, which is the worst possible outcome. The payoff matrix for the prisoners dilemma also illustrates another important concept in multi-player games; the *Nash equilibrium*, which in the case of the two criminals is the strategy ($a2$, $a2$) - to both confess. A Nash equilibrium is a stable equilibrium, or strategy, wherein neither player would benefit from defecting. This is a very important concept in game theory since these are the equilibrium points we would want agents in a multi-agent reinforcement learning task to find if the task were competitive or mixed (more on this below). Figure 5 shows the expected payoff for each player in the prisoners dilemma as a function the probability

[4]source: https://blogs.cornell.edu/info2040/2015/09/14/social-dilemmas-and-game-theory/

that each player will select action $a1$, where the Nash equilibrium is at the point $(\sigma_1(a1), \sigma_2(a1)) = (0, 0)$ in both plots.



(a) Player 1.

(b) Player 2.

Figure 5: The expected payoff surface for each player in the prisoners dilemma. Notice that the expected payoff for each payer depends on it's own strategy, but also on the strategy of the other. The Nash equilibrium is at $(\sigma_1(a1), \sigma_2(a1)) = (0, 0)$, where neither player will benefit from changing it's strategy while the other player holds its strategy constant.

$\sim$

Classification of games. Multi-player games may be classified according to their reward functions, which define the payoff structure for each player in the game [5]. Specifically, games may be classified as either **(1)** *cooperative games* (also called *identical payoff* or *common interest* games), which are games wherein all players share the same reward function, and hence the same objective(s); **(2)** *purely competitive games* (also called *zero-sum* games), which are games wherein the total reward from all players adds up to zero, meaning wins for certain players often translate to losses for other players, and finally; **(3)** *mixed games* (also called *general-sum games*), which are games in which the payoff structure is not restricted in any special way, meaning that the game need neither be fully competitive nor cooperative, and may possibly be a mixture of both [5]. Notice that the prisoners dilemma example shown above is classified as a mixed game, as there is no clear winner/loser nor are the players incentivised to fully cooperate, instead the best-case strategy for each player leads to an equilibrium which is neither the best, nor the worst case scenario for either player. Regarding fully cooperative games, notice that if control is centralised the task reduces to an MDP [6], which may then be viewed as a single agent learning task, thus cooperative games are the most straight-forward games for which to design MARL algorithms. With that being said however, designing a multi-agent learning algorithm to determine the optimal strategies for each player remains challenging for all three categories of games. Even when all players are aligned in their objectives and attempt to maximise the same reward function, coordination is still required to reach the global optimum [5] [6]. Giving players the ability to communicate by exchanging information, or learning protocols of their own, can assist with coordination amongst cooperating agents [6], this is covered in more detail later on in this report. When agents have opposing goals, a clear optimum may not exist, in which case an equilibrium between player strategies is usually searched for wherein no agent can improve it's payoff whilst other players keep their action choices fixed [5]. One useful concept which may be employed towards finding

equilibrium in a mixed/competitive games, which was used in the design of some of the earlier MARL algorithms, is the *minimax principle*: "maximise one's benefit under the worst-case assumption that the opponent(s) will always attempt to minimize it." [6].

$\sim$

Markov games. *Markov games*, sometimes called *stochastic games*, introduced in the 1950's by Lloyd Shapley in his paper with the same name, are a game theoretic extension of the Markov Decision Process (MDP) to include multiple agents [7]. Markov games may also be viewed as an extension of normal-form games, where the environment is stateful, and wherein each player (or agent; the term which will be used from here on out) observes the state at each in a sequence of discrete time steps and must endeavour to learn an optimal decision making policy (as opposed to a strategy) which maximises the reward received by the agent which is determined not only by the joint action of all agents in the game, but by the state of the environment in which the joint action is executed [5]. Formally, a *Markov game* is a tuple $(n, \mathcal{S}, \{\mathcal{A}_i, r_i\}_{i=1}^n, p)$ [5] where: n is the number of agents; $\mathcal{S}$ is the set of states in the state-space; $\mathcal{A}_i$ is the set of actions available to agent i such that $\mathcal{A} = \times_{i=1}^n \mathcal{A}_i$ is the joint action space of all agents, with $\mathcal{A}_{-i}$ being the joint action space of all agents excluding agent i (an important definition which we will use thoughout the exploration of various MARL algorithms below); $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability function which determines the probability of transitioning from one state to another given a joint action; $r_i : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function determining the numerical reward received by agent i which depends on the joint action of all agents and the current state. Note that we may also define the joint reward function as $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^n$, which returns a real-valued vector of n rewards (one per agent) given the state of the environment and a joint action. In the case that the Markov game being played is fully cooperative, the reward received by each agent will be identical at each time step. Below are some observations regarding the difference between single-agent learning in the context of an MDP, versus multi-agent learning in the context of a Markov game

$\sim$

Given an MDP, and a single-agent attempting to learn an optimal decision-making policy, the state transition and reward functions depend only on the current state and the action chosen by the agent after observing the state. In this way the environment is stationary from the perspective of the single agent, meaning that, supposing a certain state s is observed multiple times during training, the agent may confidently expect a similar outcome - reward and resultant state - in response to the same action a, if a is executed in s repeatedly. This consistency in expected outcome allows for theoretical guarantees of convergence to be made for some single agent learning algorithms [1]. However, in a Markov game played by multiple agents, the actions of all agents are weighted equally in determining how the state of the environment changes over the course of the game, meaning that the environment's state transition and reward functions depend on the joint action of all agents playing the game. Now, since the policies of each agent in the game change over time, similar states will necessarily be met with different joint actions over the course of training. From the perspective of a single agent, which we assume for the time-being is effectively unaware of the presence of the other agents in the game, this causes the "environment" to appear non-stationary, since choosing the same action in the same state observed at two different times during training will have two different expected outcomes, depending on the policies of the other agents in the game at the time the state is observed. For the single agent attempting to learn an optimal policy, this is like

chasing a moving target. Hence, because of the necessarily non-stationary nature of a multi-agent Markov game, all mathematical convergence guarantees for single-agent learning algorithms no longer apply when the same learning algorithm is applied naively to multiple agents in a Markov game [6]. As you can imagine, this makes the task of designing an effective learning algorithm even more challenging than in the single-agent context. All the MARL algorithms we will consider in this report attempt to tackle the issue of non-stationarity and the learning instability arising from it in stochastic games through various interesting methods. The principle challenges to effective learning in MARL are covered in more detail below.

$\sim$

Partially observable Markov games. A *partially observable Markov game* (the multi-agent equivalent of the partially observable Markov decision process,or POMDP, in the single-agent case) is a variation of the Markov game where, instead of observing the true state s of the environment at each time step over the course of the game, each agent i receives a partial observation o_i which is only correlated with the true state of the environment [8]. Formally, we amend the tuple defined above in order to formally define the *partially observable Markov game* as $(n, \mathcal{S}, \{\Omega_i, \mathcal{O}_i, \mathcal{A}_i, r_i\}_{i=1}^n, p)$, where Ω_i is the observation space for agent i, and for each observation $o_i \in \Omega_i$, correlated with the true state $s \in \mathcal{S}$ where s is sampled with probability determined by the state transition probability function p (as per the usual definition), the observation is distributed according to the conditional mass/density $\mathcal{O}_i$, that is $o_i \sim \mathcal{O}_i(\cdot|s, \mathbf{a})$, where $\mathbf{a}$ is the joint action of all agents in the game [9]. Partial observability is a realistic expectation to have for many real world tasks to which we would like to apply single and multi-agent reinforcement learning, and so it makes sense to consider it explicitly when designing a learning algorithm, which is not common in the prominent work done on most prominent SARL and MARL algorithms (to the best of my knowledge). The reason this is an issue, is because in a partially observable MDP/Markov game, from the perspective of a single learning agent, the next state is best inferred by the entire action-observation history of the agent(s) as opposed to the most recent observation, in other words, the underlying process is not strictly Markov, which can make learning under the assumption of the Markov property difficult, or impossible in the worst case [8]. While no theoretical or mathematical work is outlined in the report towards improving learning in a partially observable context, there is a simple but effective practical solution to the problem which is used in some of the MARL algorithms outlined below. In the paper "*Deep Recurrent Q-Learning for Partially Observable MDPs*" [9] the authors show that, in the case where the policies or value functions are implemented as deep neural networks, performance of single learning agents under partial observability can be significantly improved with the use of a *Long Short-Term Memory* (LSTM) recurrent layer, included in the neural network used to approximate the policy or value function for the task. The LSTM layer allows the agent to maintain a hidden state vector h_t, which at each time t, represents the agents *belief* about the current state of the environment (albeit a latent representation) which may be conditioned on the full action-observation history (or joint-action-observation history in MARL). The results from the use of an LSTM layer in a deep Q-network when applying DQN to certain partially observable Atari games by [9] is shown in Figure 6. The results show that the performance of the so-called Deep Recurrent Q-Network (DRQN), with an LSTM layer, degrades more gracefully than that of a standard Deep Q-Network (DQN), without an LSTM layer, when reducing the probability of receiving a particular observation, given the true state at the time it is observed [9].

$\sim$

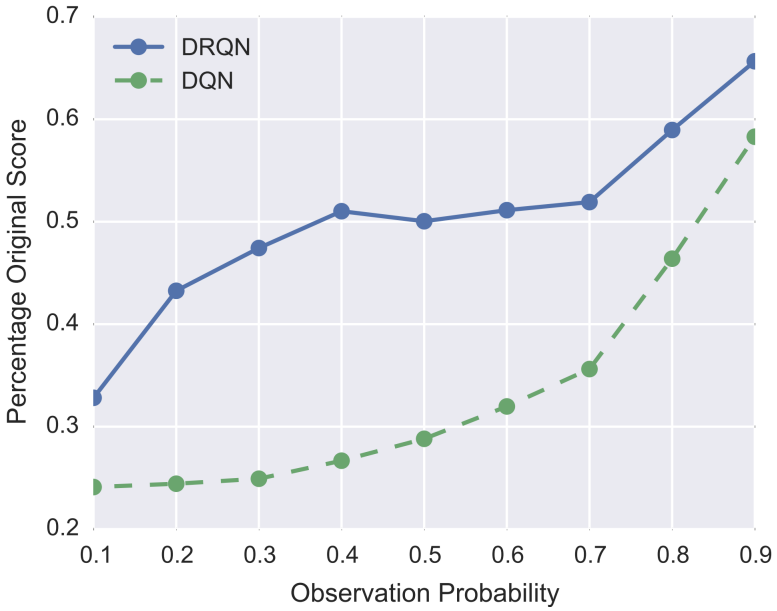


Figure 6: Taken from the paper "*Deep Recurrent Q-Learning for Partially Observable MDPs*", the plot illustrates the performance of SARL DQN, implemented with (DRQN) and without (DQN) an LSTM layer in the deep Q-network, on certain partially observable Atari games. In this plot, the probability of making an observation, given the true state at the time it is observed, is plotted against the percentage of the score achieved in the fully-observable case. Performance of the DRQN degrades more gracefully with the reduction in the probability of making the observation [9].

Markov games with networked agents. A further useful extension of Markov games is the concept of *Markov games with networked agents*, represented by the tuple $(n, G, \mathcal{S}, \{\mathcal{A}_i, r_i\}_{i=1}^{n}, p)$. Here, the only difference to the standard definition of the Markov game is the inclusion of the graph $G = (\mathcal{V}, \mathcal{E})$, in which $\mathcal{V}$ is the set of all n agents in the game and $\mathcal{E}$ is the set of connections between pairs of agents in the graph. The pairwise connections in $\mathcal{E}$ define the topology of the graph, and are usually taken to represent channels along which information and/or explicit communication flows from one agent to another, depending on the algorithm. Further to the definition of G, let $\mathcal{N}_i = \{j \in \mathcal{V} | (i, j) \in \mathcal{E}, i \neq j\}$ be the "neighbourhood" of of agent i, that is, the set of all agents adjacent to i in G. Then let $\nu_i = \mathcal{N}_i \cup \{i\}$ be the "vicinity" of agent i (a term not coined in the literature, but which is used in this report for ease of reference), that is, the set of all agents adjacent to agent i in G, including agent i. The usefulness of the definitions and concepts of G, $\mathcal{N}_i$ and ν_i, for each agent i, will become apparent when they are used in some of the MARL algorithms explored below; as it turns out, information sharing within non-disjoint, closed neighbourhoods can allow for efficient learning through the propagation of information through the graph as opposed to uniform, one-shot parameter updates based on centralised value functions or policies [10] [11].

5 Formal Definition of Multi-Agent Reinforcement Learning.

Given the preamble above, from single-agent reinforcement learning to game theory and the definition of Markov games, sufficient context has been established to properly define multi-agent reinforcement learning. See Figure 7 for an illustration.

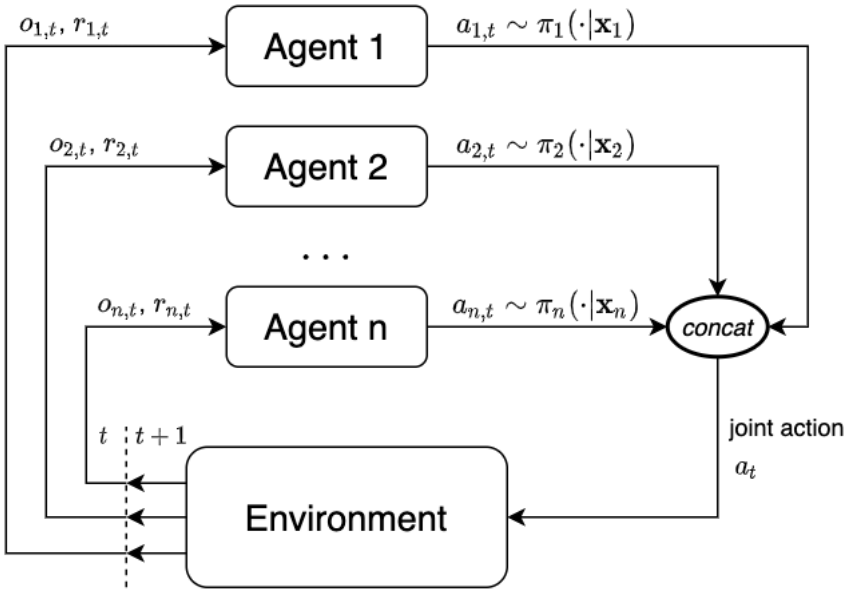


Figure 7: The agent-environment interface for multi-agent reinforcement learning. At each time step, each agent receives an observation, selects an action according to it's own policy, where $\mathbf{x}_i$ is some state information utilized by agent i, whereafter the actions are concatenated and a joint action is executed in the environment. In response, agents each received a distinct numerical reward.

The *multi-agent reinforcement learning* problem, in it's most general form (assuming partial observability), may be viewed as a Markov game $(n, \mathcal{S}, \{\Omega_i, \mathcal{O}_i, \mathcal{A}_i, r_i\}_{i=1}^n, p)$, where n agents interact with an environment over time. At each discrete time step t the environment is in some state $s_t \in \mathcal{S}$, and each agent i receives an observation $o_{i,t} \sim \mathcal{O}_i$, which is correlated with s_t. After receiving it's observation, each agent i then samples an action from a stochastic policy $a_{i,t} \sim \pi(\cdot|\mathbf{x})$, with $a_{i,t} \in \mathcal{A}_i$, according to its decision-making policy, where $\mathbf{x}$ is a state information vector which, at a minimum, includes $o_{i,t}$ but may include additional information depending on the algorithm. Note that π_i is the general expression for a decision-making policy, but it may be the case that each agent, for example, selects actions ϵ-greedily based on it's own value function Q_i. Together, the actions of all agents at time t are concatenated to make the joint action $\mathbf{a}_t \in \mathcal{A}$, where $\mathcal{A} = \times_{i=1}^n \mathcal{A}_i$ is the joint action space of all agents. $\mathbf{a}_t$ is then executed in the environment, that is, all actions are executed simultaneously as one joint action, which causes the state of the environment to transition to $s_{t+1} \sim p(\cdot|s_t, \mathbf{a}_t)$. In response, each agent then receives a numerical reward $r_{i,t+1} = r_i(s_t, \mathbf{a}_t)$ and an updated observation $o_{i,t+1}$. The most significant difference between an MDP and a Markov game is that both the new state, s_{t+1}, and the reward received by each agent i, $r_{i,t+1}$, depend not only on the individual actions and observations of each agent, but on the true state of the environment s_t and the joint action of all agents $\mathbf{a}_t$, at time t. As with single agent learning, each agent must endeavour to maximise it's individual reward signal over time, which may similarly be defined by it's return $R_{i,t} = \sum_{k=0}^T \gamma^k r_{i,t+k+1}$ following some time t, over some time horizon. If the Markov game is fully cooperative the numerical rewards received by each agent are the same, but in general that is not assumed to be the case. It is also important to note that, in the most general case, each agent seeks to optimise it's own reward signal, however, it may be the case that for some tasks it is desirable for each agent to attempt to optimise a linear combination of rewards from all agents - this idea is explored in some of the MARL algorithms below. The dependence of the state transition and reward functions for each agent on the joint action of all agents is the

principle reason that the learning of optimal policies is significantly more challenging in MARL than in SARL in general; agents must constantly adapt their individual policy such that they perform optimal action selection given the policies of all other agents, which are also dynamic. In addition to the challenge of non-stationarity in MARL, other challenges arise when attempting to design an effective multi-agent learning algorithm, each of which are outlined below.

6 Principle MARL Challenges

Using the SARL algorithms outlined in section 2 as a starting point, a naive application of any of these algorithms to the multi-agent context will likely not produce good empirical results (with the exception of DQN in some cases; more on that later) for three main reasons, although there maybe more, which are briefly considered below.

$\sim$

Non-Stationarity First is the issue of *non-stationarity* arising from the dependence of the state transition and reward functions on the joint actions of all agents, which is a problem inherent in the definition of MARL. The issue of non-stationarity in MARL often creates issues for both major classes of SARL algorithms: DQN-based and policy gradient. In the case of DQN-based algorithms, it no longer becomes feasible to use experience replay in MARL tasks when attempting to learn a value function for a single agent since the transition tuples, stored in the experience-replay buffer at a some time prior to them being sampled for training, are likely to be unrepresentative of the environment dynamics at the of sampling [12]. In the case of policy gradient algorithms, the coordination of multiple agents results in high variance of gradient estimates (of the objective function), growing exponentially with the number of agents [5]. This occurs because, in single-agent policy gradient algorithms, taking the gradient of the individual agents objective function causes terms corresponding to the log probability of state transitions to go to zero, since they do not depend on the parameters of the agents policy, however, since the probability distribution governing state transitions is stationary this is not a problem for learning. However, in the multi-agent context, the other agents action choices contribute to state-transition probabilities, and since the other agents actions are determined by dynamic policies, essentially creating a non-stationary state transition probability distribution from the perspective of the single agent, this ultimately results in a highly variance gradient estimates over the course of training. [13].

$\sim$

Goal-Setting A second issue that arises is that of *goal setting*; a task dependant issue which arises when attempting to learn an optimal policy for each agent, given that the policies of the other agents may not allow for it. That is, "simply maximising rewards is not a feasible strategy as it may not be possible for all [agents] to do at the same time" [5]. It is here that the concept of equilibrium between policies is a useful concept, although this too is difficult to determine, let alone engineer, especially in the context of deep learning. Depending on the reward structure, it is often challenging to explicitly define what the goal of an agent should be in a multi-agent game. For example, if the agents compete for a finite resource which may be completely depleted if all agents are greedy, but replenishes over time if all agents are moderate consumers, then a cooperative strategy of moderate

[5] See [13] for a simple scenario and proof illustrating how the probability of taking a step in the correct direction decreases exponentially with the number of agents.

consumption will yield the highest reward over time. However, the temptation then arises for a single agent to defect in order to gain a higher short-term return whilst the other are cooperating. This makes it difficult to engineer a learning algorithm which allows for sufficient coordination to arrive at group equilibrium, whilst still encouraging efforts on the part of individual agents to optimise individual returns.

$\sim$

Scalability. The third and final challenge is that of *scalability*, which is a problem in SARL, too, but for slightly different reasons. In MARL, the issue of scalability arises for two main reasons, among others. Firstly, the inclusion of multiple agents causes the cardinality of the joint-action set $\mathcal{A}$ to increase exponentially with the number of agents, which is a problem since most algorithms will, in one way or another, rely on the learning of action values. Secondly, each individual agent must be aware, at some level, of the other agents in the state space. It is often the case that more information shared between agents makes the task of coordinating, and therefore learning, easier to achieve. However, the amount of information shared between agents during training and execution translates directly into an increase in computational complexity. In short, centralised control and coordination of multiple agents requires the processing of large amounts of information per time step which increases in a non-linear way with the number of agents.

7 Six MARL Algorithms

7.1 Background

Although Multi-agent reinforcement learning has a rich history [6] [14] [15], the development of sophisticated multi-agent learning algorithms, designed to be implemented using modern deep learning techniques, has only truly happened over the latter part of the last decade, which, like many modern developments in RL, followed the publishing of the landmark paper DQN paper "*Playing Atari with Deep Reinforcement Learning*", by Mnih et. al and Google Deepmind, in late 2013 (and the subsequent letter "*Human-level control through deep reinforcement learning*" published in nature a year later). Indeed, as of 2008, as noted in "*A Comprehensive Survey of Multiagent Reinforcement Learning*" [6], most applications of MARL algorithms were only to small problems, like static games and small gridworlds, meaning most algorithms were only suited for tasks which allowed for tabular learning [10]. The authors state in conclusion that "these algorithms are unlikely to scale up to real-life management problems, where state or action spaces are large or even continuous. Few of them are able to deal with incomplete, uncertain observations." [6]. Before the surge in development of modern MARL (and SARL) algorithms, the simplest, and most commonly used MARL algorithm was independent Q-learning [12], proposed in 1993 by Ming Tan [16], wherein each agent simply learns its own Q-function, conditioned only on the true state of the environment and its own action, thus treating other agents as part of the environment and ignoring the problem of non-stationarity altogether [12]. IQL is appealing since it avoids the problem of scalability which arises when trying to learn a joint Q-function that is conditioned on the joint action $\mathbf{a} \in \mathcal{A}$, since $|\mathcal{A}|$ grows exponentially in the number of agents. As noted in [12] "[IQN] is also naturally suited to partially observable settings, since, by construction, it learns decentralised policies in which each agent's [action choices are conditioned] only on its own observations.". However, the unattended problem of non-stationarity in IQN means that all convergence guaranteed are ruled out. However, on the one hand, there are empirical results

from experiments conducted with IQN which suggest that this problem is not so bad [17], but, On the other hand, these results do not involve deep learning [12]. Thus learning stability in MARL algorithms is of primary concern in the development of modern day deep mult–agent reinfrocment learning algorithms [13] [12] [10] [8] [21] [11]. Even in the recent development of MARL algorithms, say in the last five years, a major shortcoming is that most research papers present only empirical results of the algorithms they propose, with very few offering any convergence guarantees [10], the likes of which have been established for several SARL algorithms [1]. The six MARL algorithms outlined below cover a broad range of algorithmic techniques and approaches to the problem of multi-agent learning, each attempting to address the core issues of non-stationarity, goal setting and scalability in their own way. While all six make an attempt to establish the mathematics underpinning the proposed algorithm, only one ([10]) offers a mathematical proof of convergence, being made only for fully cooperative tasks. In short, there is a lot of work still to be done on the development and successful application of MARL algorithms, but the development of the last five years, as can be seen from examining the algorithms below, is evidence of rapid progress and promises exiting future developments.

7.2 Independent Advantage Actor-Critic (IA2C)

The first algorithm to be considered was proposed in the paper "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments" [13], which is a simple extension of the classic SARL policy-gradient algorithm A2C to the MARL context, and shall be referred to henceforth as *Independent Advantage Actor-Critic* (**IA2C**). The authors of [13] state that their goal was to create a a general-purpose MARL algorithm which satisfies the following: (1) Learned policies must use only local info at execution time, (2) No assumptions about the availability of a differentiable model of environment dynamics or any structure on communication methods between agents should be made, and finally (3) It should be applicable to competitive, cooperative and mixed MARL tasks. In order to satisfy these three constraints, the IA2C algorithm follows the same steps as A2C, where individual individual agents: interact with the environment using their parameterised policies to make action choices based on local observations (i.e. it is **not** assumed that individual agents have access to the full state at any given time); observe local rewards and resultant observations in response to actions taken; compute advantage estimates based of calculated returns (discounted reward sums), using parameterised state-value function approximators, and finally; optimise individual policies using computed gradients of an objective function which takes the form ∇*log(action probability)*$\times$*action advantage*. The key difference between A2C and IA2C is that, in IA2C, the value function (i.e. the critic) belonging to each individual agent is conditioned on global information: in addition to the agents local observation, the value function takes as input the local observations and action choices of **all** other agents in the state space at each time step in order to estimate the expected action/state value, and hence the advantage estimate. The logic behind this is that, from the single agent perspective, the action choices (dependant on the stochastic policies) of other agents contribute to the state-transition probability distribution from once time step to the next, and hence determine, in part, the value of any state, or state-action pair, specific to the agent in question. In this way IA2C allows for "centralised training" and "decentralised execution" - see Figure 8 for an illustration.

$\sim$

In order to define IA2C explicitly, consider a partially observable Markov game $(n, \mathcal{S}, \{\Omega_i, \mathcal{O}_i, \mathcal{A}_i, r_i\}_{i=1}^n, p)$. At each time step t, each agent $i \in \{1, 2, \ldots, n\}$ receives a local observation $o_{i,t}$ and selects an action $a_{i,t} \sim \pi_{\theta_i}(\cdot|o_i)$,

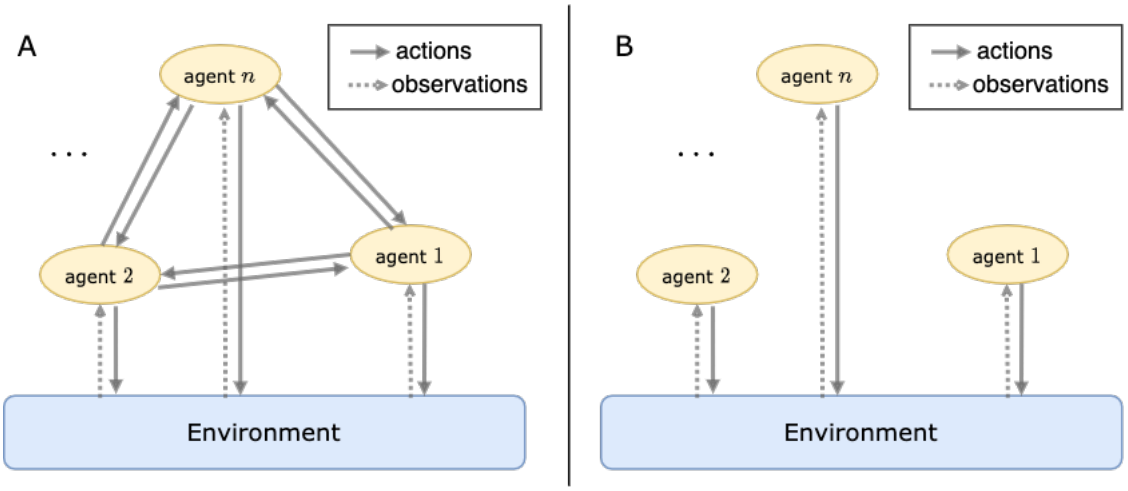


Figure 8: An illustration of IA2C. (**A**) During centralised training, in addition to receiving state information from the environment, agents share and receive action information with one another in order to condition their value functions, and therefore policies, thus attempting to deal with the issue of non-stationarity. (**B**) During decentralised execution, having conditioned their policies on global information during training, agents simply receive observations and select actions, just as they would in the single-agent A2C algorithm.

according to it's own policy parameterised by θ_i, whereafter the joint action $\mathbf{a}_t$ is executed, and each agent receives a numerical reward r_i from the environment. Just as in the single agent case, the goal of each agent is to learn an optimal policy π_i^* in order to maximise it's return over time. In addition each agent maintains its own action value function $Q_{\omega_i} : \Omega_i \times \mathcal{A} \rightarrow \mathbb{R}$ parameterised by ω_i. Let $\boldsymbol{\pi}_\theta = \{\pi_{\theta_1}, \pi_{\theta_2}, \dots \pi_{\theta_N}\}$ parameterised by $\boldsymbol{\theta} = [\theta_1^T, \theta_2^T, \dots, \theta_N^T]^T$ be the joint policy for all agents, and let $\mathbf{Q}_\omega = \{Q_{\omega_1}, Q_{\omega_2}, \dots Q_{\omega_N}\}$ parameterised by $\boldsymbol{\omega} = [\omega_1^T, \omega_2^T, \dots, \omega_N^T]^T$ bet the joint action-value function for all agents. Defining $\mathbf{a}_{-i} \in \mathcal{A}_{-i}$ to be the joint action of all agents, excluding agent i, the gradient of the objective function $J(\theta_i)$ for agent i used to compute the gradient of the policy parameters for agent i may be expressed as:

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{o_i \sim \mathcal{O}_i, \mathbf{a} \sim \boldsymbol{\pi}}[\nabla_{\theta_i} log \pi_i(a_i|o_i) Q_i^{\pi_i}(o_i, a_i, \mathbf{a}_{-i})] \tag{27}$$

where π_{θ_i} and Q_{ω_i} are abbreviated as π_i and Q_i. The important component of this expression is the centralised action-value function, $Q_i^{\pi_i}(o_i, a_i, \mathbf{a}_{-i})$, which approximates the expected return for agent i, conditioned on both the local observations of the agent as well as the action selection choices of all agents in the game. This makes intuitive sense because, as discussed previously, in a multi-agent tasks, state transitions depend on the current state at a given time-step, along with the actions of all agents. It would be nearly impossible, given only the local state and action information pertaining to a single agent, to reliably predict the next state (locally or globally) without first knowing the action choices of all other agents interacting with the environment at any given time step. Indeed, the authors note that a primary motivation behind this method of centralised training is that, if the actions taken by all agents at any given time step are known, the environment is stationary (in terms of the state transition distribution) even as policies change, since $P(s'|s, a_1, \dots, a_N, \pi_1, \dots, \pi_N) = P(s'|s, a_1, \dots, a_N) = P(s'|s, a_1, \dots, a_N, \pi_1', \dots, \pi_N')$ for any $\pi_i \neq \pi_i'$. Then, it is also important to note that the centralised action-value function of agent i, conditioned on global information, is optimised to predict the value of state-action pairs for agent i only. The authors [13] emphasise that the estimation of local expected returns

conditioned on global information allows, in theory, for agents to be successfully trained on tasks which are either cooperative, competitive or mixed, since the agents are not restricted having to optimise their policy to maximise a single, joint (global) reward signal. It is possible to balance local and global reward signals in order to study group dynamics in social dilemmas, in which case one would need to adjust the reward signal for individual agents to be a linear, weighted sum over local and non-local rewards.

$$\sim$$

When considering a practical application of IA2C to MARL tasks where the state space is discrete, it is useful to redefine the expression for the gradient of the objective function above to be:

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{o_i \sim \mathcal{O}_i, \mathbf{a} \sim \boldsymbol{\pi}} \left[\nabla_{\theta_i} log \pi_i(a_i|o_i) \hat{A}_i \right] \tag{28}$$

where at some time step t the advantage estimate for agent i will be given by $\hat{A}_{i,t} = R_{i,t} - V_{\omega_i}(o_{i,t}, \mathbf{a}_{-i,t})$, where V_{ω_i} is the parameterised state-value function approximator belonging to agent i.

$$\sim$$

Finally, it is also possible to extend IA2C to MARL tasks where the state space is continuous, not discrete. The authors refer to this extension as *Multi-Agent Deep Deterministic Policy Gradient* (MADDPG). Consider the joint deterministic policy $\boldsymbol{\mu}_\theta = \{\mu_{\theta_1}, \mu_{\theta_2}, \ldots, \mu_{\theta_N}\}$, where as before each policy takes only the local observation as input, returning a real-valued vector representing the continuous action choice of each agent. Then, by the chain rule, the expression for the gradient of the objective function becomes

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{o_i \sim \mathcal{O}_i, \mathbf{a} \sim \mathcal{B}} \left[\nabla_{\theta_i} \mu_i(o_i) \nabla_{a_i} Q_i^\mu(o_i, a_i, \mathbf{a}_{-i})|_{a_i = \mu_i(o_i)} \right] \tag{29}$$

where $\mathcal{B}$ in the expression $a \sim \mathcal{B}$ represents the memory buffer used to store sample transitions during exploration. It is written this way since, in the classic Deep Deterministic Policy Gradient algorithm, Gaussian noise is added to the deterministic, real-valued action vector output by the policy at each time step for the purposes of exploration. This means that actions stored in the buffer are distributed and non-deterministic. Notice too that the actions taken as argument by the value function are determined by the policies of each of the agents. Below a detailed outline of the *IA2C* algorithm is given.

Algorithm Independent Advantage Actor-Critic (IA2C)

Parameters: γ, α;

Initialise: $\mathcal{B} \leftarrow \emptyset$, $\boldsymbol{\pi}_\theta = \{\pi_{\theta_1}, \ldots, \pi_{\theta_N}\}$ and $\mathbf{V}_\omega = \{V_{\omega_1}, \ldots, V_{\omega_N}\}$;

for *k=0,1,2,...* **do**

1. Collect a set of trajectories $\mathcal{B}_k = \{\tau_j\}$, where each τ_j in $\mathcal{B}_k$ is a sequence of transition tuples of the form $(\mathbf{o}_j, \mathbf{a}_j, \mathbf{r}_{j+1}, s_{j+1})$;
2. Compute returns $R_{i,t}$ for each agent i, at each time step t across all $\tau_j \in \mathcal{B}_k$;
3. Compute advantage estimates $\hat{A}_{i,t} = R_{i,t} - V_{\omega_i}$ for each agent i, for each time step t, across all $\tau_j \in \mathcal{B}_k$;
4. For each agent i, perform a batch-optimization step for the state-value function approximator in order to minimise $\mathcal{L}(\omega_i) = \frac{1}{|\mathcal{B}_k|T} \Sigma_{j \in \mathcal{B}_k} \Sigma_{t=0}^{T} (V_{\omega_i} - R_{i,t})^2$;
5. For each agent i, estimate the policy gradient as $\nabla_{\theta_i} J(\theta_i) \approx \frac{1}{|\mathcal{B}_k|T} \Sigma_{j \in \mathcal{B}_k} \Sigma_{t=0}^{T} \nabla_{\theta_i} log\pi(a_{i,t}|o_{i,t}) \hat{A}_{i,t}$;
6. For each agent i, update the policy parameters $\theta_i^{(k+1)} \leftarrow \theta_i^{(k)} + \alpha \nabla_{\theta_i} J(\theta_i)$, or using another gradient optimization method.;

end

7.3 Independent Q-learning with FingerPrint (FPrint)

The second algorithm to be considered was proposed in the paper "Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning" [12]. In the paper the authors propose two methods for stabilising learning in Independent Q-Learning [16], an MARL application of SARL Q-learning, with respect to the use of the experience replay buffer. Experience replay is a technique which typically increases robustness and sample efficiency in single-agent DQN, but causes instability when applied naively to the MARL setting since, as stated previously, the state transition and reward functions for a single agent depend on the policies of other agents in the state space, which change over time. When attempting to apply Q-learning (or deep Q-learning) to multi-agent tasks, one may take two approaches depending on the nature task itself. If the full state is available to each agent at each time step then it is possible to model a cooperative MARL system as a single meta-agent, where values must be learned for pairs of states and joint-actions. However, the issue of scalability quickly becomes a problem since for n agents with a discrete set of m available actions each, there will be m^n possible joint actions at any given time step. This renders the problem of numerically approximating the Q-function intractable for most complex tasks. Proposed in [16] in 1993, Independent Q-Learning (IQL) is a popular alternative wherein each agent independently learns its own action-value function, treating other agents as part of the environment and hence dispensing with the need to approximate a joint Q-function, and solving the issue of scalability. However, IQL introduces a new problem: the issue of non-stationary from the perspective of each single agent which rules out any convergence guarantees that exist in the single agent case, as outlined in [1]. In IQL the non-stationarity arises when attempting to train a single agent using experience replay, which relies on sampling historical transitions represented by *(state, action, reward, next-state)* tuples from a memory buffer, since the reward and resultant state depend not only on the actions of the agent itself, but on the joint action of all agents in the state space. Since this is the case, batches of transitions sampled during training contain

tuples which often do not reflect the current dynamics of the 'environment', from the perspective of a single agent. In fact, the more time that elapses between recording a transition and using it in a training batch, the less relevant the tuple becomes for learning the value function for the present joint policy of all agents [12]. It is interesting to note however, that there is substantial empirical evidence to show that IQL often performs well in practice despite these challenges [17] [6]. In [12], two possible solutions to the problem of instability arising from experience-replay in IQN are proposed: *importance sampling* and the use of *multi-agent fingerprints*, which are both examined below in order.

$\sim$

Importance Sampling. In the context of an MARL task, the usual historical data one would collect during Q-learning - transition tuples - may be thought of, from the view of a single agent at some point following the time of collection, as *off-environment data* in the sense that a snapshot of the policies of all other agents in the environment at the time of collection constitute different environment dynamics when compared to some future time during training. It has been shown that it is possible to learn from *off-environment data* by weighting the samples drawn from the memory buffer using a ratio of the joint probability for the joint-action determined by policies at the time of collection versus at the time of sampling [12] [19]. Consider a partially observable Markov game $(n, \mathcal{S}, \{\Omega_i, \mathcal{O}_i, \mathcal{A}_i, r_i\}_{i=1}^n, p)$ consisting of n agents, wherein each agent i maintains its own action-value function $Q_{\theta_i} : \Omega_i \times \mathcal{A}_i \to \mathbb{R}$ parameterised by θ_i. Let At each time step t, each agent $i \in \{1, 2, \ldots, N\}$ receives an observation $o_{i,t}$, choosing an action $a_{i,t}$ either at random, with probability ϵ, or according to $\arg\max_{a_i} Q_i(o_{i,t}, a_i)$, with probability $1 - \epsilon$. Note that using the action-value function, we may implicitly define the probability that agent i selects an action a given a state s using the Boltzmann distribution as $\pi_i(a_i|s_i) = \frac{e^{Q_i(o_i,a_i)}}{\sum_{u\in\mathcal{A}_i} e^{Q_i(o_i,u_i)}}$ for agent i [17]. Using this expression we can define joint probability of $\mathbf{a}_{-i}$ given $\mathbf{o}_{-i}$ as $\boldsymbol{\pi}_{-i}(\mathbf{a}_{-i}|\mathbf{o}_{-i})$, the product of action selection probabilities for all agents excluding agent i given local observations $\mathbf{o}_{-i}$. Then, we can write down the Bellman optimality equation for a single agent in the multi-agent setting as:

$$Q_i^*(o_i, a_i) = E\left[r_{i,t+1} + \gamma \max_{a_i'} Q^*(o_{i,t+1}, a_i')|o_{i,t} = o_i, a_{i,t} = a_i\right] \tag{30}$$

$$= \sum_{\mathbf{a}_{-i}} \boldsymbol{\pi}_{-i}(\mathbf{a}_{-i}|\mathbf{o}_{-i}) \left[r_i(s, a_i, \mathbf{a}_{-i}) + \gamma \sum_{s'} p(s'|s, a_i, \mathbf{a}_{-i}) \max_{a_i'} Q_i^*(o_i', a_i')\right] \tag{31}$$

where $r_i(s, a_i, \mathbf{a}_{-i})$ is the reward resulting from being in a state s and taking joint action $a_i \cup \mathbf{a}_{-i}$, and $p(s'|s, a_i, \mathbf{a}_{-i})$ is the probability of transition to state s' whilst in state s given joint action $a_i \cup \mathbf{a}_{-i}$, determined implicitly by the environment as per usual. Then, in the expression above, $\boldsymbol{\pi}_{-i}(\mathbf{a}_{-i}|\mathbf{o}_{-i})$ is the difference between the Bellman optimality equation used in the single agent case and that used in the multi-agent case - it is the component which introduces the non-stationarity in IQL which, as discussed above, arises from multiple policies changing over the course of training rendering any recorded transition tuples *off-environment data*. In order to apply importance sampling to IQN, for each agent i at the time of collection t_c, $\boldsymbol{\pi}_{-i}^{t_c}(\mathbf{a}_{-i}|\mathbf{o}_{-i})$ is recorded in the augmented transition tuple $(o_i, a_i, r_i, \boldsymbol{\pi}_{-i}(\mathbf{a}_{-i}|\mathbf{o}_{-i}), o_i')^{t_c}$. Then, at the time of recall t_r (during training), the action-value function belonging to agent i may be optimised using *off-environment data* by minimising the loss

$$\mathcal{L}_i(\theta_i) = \sum_{j=1}^{b} \frac{\boldsymbol{\pi}_{-i}^{t_r}(\mathbf{a}_{-i}|\mathbf{o}_{-i})}{\boldsymbol{\pi}_{-i}^{t_j}(\mathbf{a}_{-i}|\mathbf{o}_{-i})} \left[y_j^{DQN} - Q(o_i, a_i; \theta_i)\right]^2 \tag{32}$$

where t_j is the time of the collection of the j-th sample, b is the batch size, and y_j^{DQN} is the DQN target for agent i derived from the j-th sample, defined in exacly the same was as in single-agent DQN. The augmented loss above essentially weights each transition tuple in the batch sampled from the buffer according to how probable a the joint action $\mathbf{a}_{-i}$ is at the time of recall (training) versus the time of collection (when it was recorded in the buffer). In this way, the more relevant the joint action is to the current dynamics, the greater the contribution it will have to the computed gradient, which will in turn be used to update the parameters of the action-value function. The benefits of augmenting the IQN algorithm with this method of importance sampling is that it provides an unbiased estimate of the true objective when optimising the Q-function. However, a practical issue is that it often yields importance ratios with large, unbounded variance [20]. Truncating the importance ratio is one way to mitigate this issue, although this only serves to re-bias the unbiased data since more recent samples are less likely to be clipped than less-recent ones. With this in mind, we turn to the method of using Finger Prints to stabilise IQL.

$\sim$

Finger Prints. The second approach to solving the problem of non-stationarity with respect to the replay buffer proposed in [12] is inspired by an earlier variant of Q-Learning proposed by the authors of [18] in 2003 called "Hyper Q-Learning", which avoids the non-stationarity problem in IQL by conditioning each agents policy on the inferred policies of other agents in the state space. In this way, instead of viewing other agents in the space as part of the environment from the single agent perspective, thus rendering the environment non-stationary, each agent learns an explicit mapping from the policies of other agents to its own action-values. This essentially reduces the problem to a single-agent Q-learning, but has the effect of ballooning the state space from the single agent perspective, especially as the number of agents grows large. The authors in [12] point out that this can quickly become impractical, with the problem becoming exacerbated in the case where agents policies consist of deep-neural networks where the parameters of each network are high dimensional (sometimes $\mathbf{O}(10000)$ or more). In that case we would have have an augmented observation for agent i being $\tilde{o_i} = \{o_i, \boldsymbol{\theta}_{-i}\}$, where $\boldsymbol{\theta}_{-i}$ represents the policy parameters of all agents excluding agent i, and each agent i would essentially learn a mapping from the weights $\boldsymbol{\theta}_{-i}$ to expected returns. As mentioned above, in most cases $\boldsymbol{\theta}_{-i}$ would certainly be too large to use as input to the Q-function. However, an agent need not learn a mapping from any possible $\boldsymbol{\theta}_{-i}$ to action-values, only from the the values of $\boldsymbol{\theta}_{-i}$ that are actually stored in the memory buffer. The sequence of policies, which would be defined by a sequence of parameter values of the parameters $\boldsymbol{\theta}_{-i}$, for any single tuple in the memory buffer, may be thought of as tracing out a single trajectory in a higher dimensional policy space. Now, in order to stabilise experience replay an agent must learn to disambiguate tuples in the replay buffer according to where along the policy trajectory each tuple was recorded. And while this may be achieved in theory by recording policy parameters and learning mappings to action values, the authors of [12] suggest it may be possible for an agent to learn to disambiguate the position in policy trajectory with which each transition tuple is associated using a low dimensional *fingerprint* which satisfies the following properties:

1. It should be correlated with the true action-values specific to each agent.

2. It should vary smoothly over training to allow the model of the Q-function to generalise across experiences sampled from the replay buffer which are determined by the dynamic policies of other agents.

Surprisingly, the authors of [12] suggest, as a candidate for such a fingerprint, the combination of e, the training iteration number (the global step counter during training) and ϵ the exploration rate, which typically is set to decay gradually over the course of training. The proposition is simply then that each time a tuple is recorded, the local observation vector for each agent is concatenated with the fingerprint, forming the augmented observation $\tilde{o}_i = o_i \cup e \cup \epsilon$. Notice that both e and ϵ vary smoothly over the course of training, and should in theory allow the agent to learn to disambiguate the relative trajectory positions in higher dimensional policy space associated with any transition tuple stored in memory. The IQN + fingerprint algorithm is applied to an MARL StarCraft micromanagement task in [12], where each agent is associated with single military unit forming part of a group who's collective goal is to cooperate in the mission of seeking out and killing enemy units in a large map. The results demonstrate significant improvement when compared to straightforward IQN, as well as IQN with importance sampling.

7.4 Networked Actor-Critic with Consensus Update (ConsNet)

The next MARL algorithm to be considered was proposed in a paper titled "Fully Decentralized Multi-Agent Reinforcement Learning with Networked Agents" [10] where the authors consider the fully-decentralised MARL problem where agents are embedded in a graph structure constituting a communication network used to propagate local information throughout the graph instead of having a central controller. The authors note that, as of 2018, their work "appears to be the first theoretical study of fully decentralized MARL algorithms for networked agents that use function approximation...with provable convergence guarantees". The algorithm proposed in [10] is decentralised in the same sense that IA2C is decentralised; agents may select actions based solely off local observations, but maintain a local approximation of a global action-value function which takes as input the joint actions of all agents in the graph. The most significant difference is that in the proposed algorithm ([10]) agents optimise their local action-value functions by aggregating parameter information from adjacent agents in the graph structure to form consensus parameter updates. The algorithm shall therefore be referred to as **ConseNet** from here on. It is important to note three things about the so-called 'consensus updates'. First, updating the action-value function by consensus implies that the algorithm will be particularly well-suited to cooperative tasks, but not competitive or even mixed tasks. Secondly, the consensus update also allows for the propagation of local information through the graph structure should theoretically enable all agents to approximate something close to a global action-value function without having to communicate directly, which is the principle contribution of the paper. Lastly, ConseNet tackles the issue of scalability by dispensing with the need for a global action-value function, but without resorting to completely independent Q-learning, where the propagation of information should enable agents to maintain a fair approximation of a global action-value function in a way that stabilises learning and dispenses with the massive computational overhead. We now go on to explicitly define ConseNet.

$\sim$

Consider the networked Markov game $(n, G, \mathcal{S}, \{\mathcal{A}_i, r_i\}_{i=1}^n, p)$, where agents are embedded in a graph structure $G = (\mathcal{V}, \mathcal{E})$. As defined above, the set of of vertices $\mathcal{V}$ is the set of n agents connected by weighted edges $\mathcal{E}$, each of which represents a connection between two agents (Note: the phrase 'communication link' is begin avoided explicitly here, as agents are not given the ability to communicate explicitly by message passing as is the case in the three algorithms final MARL algorithms examined in this report). The set of edges are defined by an $n \times n$ matrix C where $c_{ij} \in C$ is the $(i, j)^{th}$ element in C representing the weighted connection between

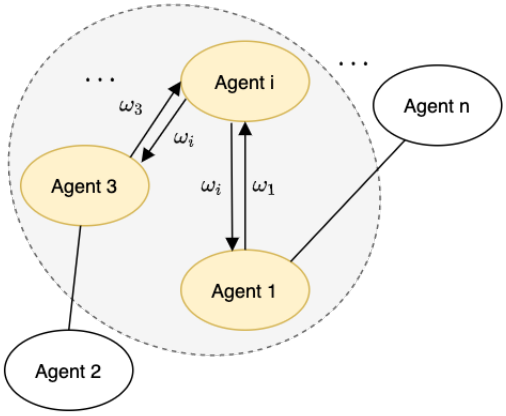


Figure 9: Agents in a graph structure sharing parameters to perform a *consensus update*.

agent $i \in \mathcal{V}$ and agent $j \in \mathcal{V}$. A connection exists between one agent and another if the weighting represented by the corresponding entry of C is non-zero, meaning $\mathcal{E} = \{ij | i, j \in \mathcal{V}, c_{ij} > 0, i \neq j\}$, thus determining the topology of the graph G, see figure 9 for an illustration. Let $\pi_{\theta_i} : \mathcal{S} \times \mathcal{A}_i \to [0, 1]$ and $Q_{\omega_i} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ be the parameterised policy and action-value function belonging to agent i, respectively. Let the global policy be defined as $\boldsymbol{\pi}_\theta : \mathcal{S} \times \mathcal{A} \to [0, 1]$ where $\boldsymbol{\theta} = [\theta_1^T, \theta_2^T, \ldots, \theta_N^T]^T$ is the concatenation of all policy parameters. Then, at each time step t during interaction with the environment each agent observes the true environment state $s_t \in \mathcal{S}$ and selects an action $a_{i,t} \sim \pi_{\theta_i}(\cdot|s_t)$ according to it's local policy. The joint action $\mathbf{a}_t \in \mathcal{A}$ is then executed and each agent receives a reward $r_{i,t+1}$, and observes the updated state s_{t+1}. What follows are some important definitions to examine how training occurs under ConseNet. Let the joint probability of selecting a joint action $\mathbf{a} \in \mathcal{A}$, given the state $s \in \mathcal{S}$, be defined as $\boldsymbol{\pi}(\mathbf{a}|s) = \Pi_{i=1}^N \pi_i(a_i, s)$, where π_{θ_i} is abbreviated as π_i for brevity. Additionally, if $r_{i,t}$ is the reward received by agent i following time $t-1$, then define $\bar{r}_t = \frac{1}{N} \sum_{i \in \mathcal{V}} r_{i,t}$ to be the average reward received by all agents. Then if we define the expected average reward per time step over an infinite time-horizon to be $\bar{R}(s, \mathbf{a}) = \lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T-1} \bar{r}_{t+1} = \mathbb{E}[\bar{r}_{t+1} | s_t = s, a_t = \mathbf{a}]$, for some state $s \in \mathcal{S}$ and some joint-action $\mathbf{a} \in \mathcal{A}$, the joint objective of all agents is to solve the optimisation problem

$$\max_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{s \sim d_\theta, \mathbf{a} \sim \boldsymbol{\pi}_\theta} \left[\bar{R}(s, \mathbf{a}) \right] \tag{33}$$

$$= \sum_{s \in \mathcal{S}} d_\theta(s) \sum_{\mathbf{a} \in \mathcal{A}} \boldsymbol{\pi}_\theta(\mathbf{a}|s) \bar{R}(s, \mathbf{a}) \tag{34}$$

i.e. find the optimal values for the joint policy parameter $\boldsymbol{\theta}$ such that the expectation of the average reward received per agent is maximised over all possible states in $\mathcal{S}$. Note that $d_\theta(s)$ is the stationary distribution, induced by $\boldsymbol{\pi}_\theta$, of the Markov chain associated with the stochastic game (a multi-agent MDP) where it is assumed that the chain is aperiodic and irreducible (it is always possible to reach any state from any other state, given a large enough time horizon). Note that this is not always the case with tasks such as video games, but requires that the task be continuing, for example, the canonical MARL task of optimising of traffic flow by having multiple agents control the phase of traffic lights in a city (for finite time-horizon tasks the objective function can be reformulated as an expectation over trajectories instead). The objective function $J(\boldsymbol{\theta})$ is the expected average reward per agent any given state action pair, over all states and actions. We can use the objective function to define a global, relative action value function (in a slightly unusual way):

$$Q_\theta(s, \mathbf{a}) = \mathbb{E}\left[\sum_t (\bar{r}_{t+1} - J(\boldsymbol{\theta}))|s_0 = s, a_0 = \mathbf{a}, \boldsymbol{\pi}_\theta\right] \tag{35}$$

In words, $Q(s, \mathbf{a})$ is the value of s given state action pair, where "value" is a relative quantity computed as the infinite sum of the difference between the average reward per agent for each state-action pair, following s and a, and the quantity $J(\boldsymbol{\theta})$ which is the expectation of the average reward per agent over all possible states and actions. Starting in a global state s, taking a joint action $\mathbf{a}$ and following a joint policy $\boldsymbol{\pi}_\theta$ will yield a high Q-value if the agents' action choices yield rewards that are on average greater than $J(\boldsymbol{\theta})$. Now, in practise $J(\boldsymbol{\theta})$ is not tractable and a global action-value function is not implemented, but rather approximated locally by each agent. So, instead of $J(\boldsymbol{\theta})$, each agent keeps an exponentially moving average of it's own local rewards over time, which is used as an approximation for $J(\boldsymbol{\theta})$ in creating the target for the temporal-difference (TD) error when updating the parameters of the local action-value function. Let $\beta_{\omega,t} \in [0, 1]$ be the step size parameter used to compute the TD error. Note that the step size may be adjusted over the course of training to improve stability, hence it is dependant on t. The exponentially moving average of the reward received by agent i can then be computed at time $t + 1$ as:

$$\mu_{i,t+1} = (1 - \beta_{\omega,t}) \cdot \mu_{i,t} + \beta_{\omega,t} \cdot r_{i,t+1} \tag{36}$$

Now, recall that for each agent i, $Q_{\omega_i}(s, \mathbf{a}) \approx Q_\theta(s, \mathbf{a})$ is a local function approximator of the global action-value function. Then, at each time step $\mu_{i,t}$ is used to compute the local TD error, $\delta_{i,t}$ as:

$$\delta_{i,t} = (r_{i,t+1} - \mu_{i,t}) + Q(s_{t+1}, \mathbf{a}_{t+1}; \omega_{i,t}) - Q(s_t, \mathbf{a}_t; \omega_{i,t}) \tag{37}$$

where $Q_t(s_t, \mathbf{a}_t; \omega_{i,t}) = Q_{\omega_{i,t}}(s_t, \mathbf{a}_t)$, and a local update estimate of for the parameters $\tilde{\omega}_{i,t}$ as

$$\tilde{\omega}_{i,t+1} = \omega_{i,t} + \beta_{\omega,t} \cdot \delta_{i,t} \cdot \nabla_{\omega_i} Q(s_t, \mathbf{a}_t; \omega_{i,t}) \tag{38}$$

Finally, the updated parameters $\omega_{i,t+1}$ for the local action-value function belonging to agent i at time t are computed as the weighted average of local parameter-update estimates $\tilde{\omega}_{j,t+1}$ of each agents j in the neighbourhood of agent i (i.e. all agents adjacent to i in G), with weights taken from the weighted adjacency matrix C:

$$\omega_{i,t+1} = \sum_{j \in \mathcal{V}} c_{ij} \cdot \tilde{\omega}_{j,t} \tag{39}$$

A few important things to notice about the algorithm above are the following. Firstly, the exponentially moving average $\mu_{i,t}$ may track the performance of agent i over a very long time horizon, which stabilises the updates to Q_{ω_i} and dispenses with the need to perform batch updates. If one were to perform one-step optimisation on Q_{ω_i} using a TD error computed using only the one-step reward (in the usual way), the variance would be very high and it would cause instability in training. However, because the action-value functions are optimised using one-step updates in this way, it is not necessary to use a memory buffer to sample batches for training,

and hence the problem of training on "off-environment" data - as tackled in the **FPrint** algorithm [12] - no longer applies here. Secondly, updating the action-value functions using neighborhood consensus, as shown above, should theoretically ensure local information is propagated throughout the graph, encouraging steady approximation of $Q_\theta(s, \mathbf{a})$ and stabilising training. The last part of the algorithm that need be discussed is the updating of the policy parameters, done using local advantage estimates. For each agent i at time t we can compute the local advantage estimate using an approximation of the state-value by taking the expectation of $Q_{\omega_i}(s, \mathbf{a})$ over action probabilities as

$$\hat{A}_{i,t} = Q(s_t, \mathbf{a}_t; \omega_{i,t}) - V_i(s_t) \tag{40}$$

$$= Q(s_t, \mathbf{a}_t; \omega_{i,t}) - \sum_{a_i \in \mathcal{A}_i} \pi_i(a_{i,t}|s_t) Q(s_t, a_{i,t}, \mathbf{a}_{-i,t}; \omega_{i,t}) \tag{41}$$

where $\mathbf{a}_{-i,t}$ is the joint action of all agents excluding agent i. Then, the advantage estimate can be used to compute the one-step update to the policy parameters θ_i for agent i, for each time step t, as:

$$\theta_i^{(k+1)} = \theta_i^{(k)} + \beta_{\theta,t} \cdot \left[\nabla_{\theta_i} log \pi_{\theta_i}(a_{i,t}|s_t) \hat{A}_{i,t} \right] \tag{42}$$

where $\beta_{\theta,t} \in [0, 1]$ is the step size parameter for policy updates, and $\pi_{\theta_i}(a_{i,t}|s_t)$ uses parameters $\theta^{(k)}$. The full ConseNet algorithm is shown below. See section 4.1 in [10] for a proof of convergence of the ConseNet arising from the sharing of local information towards estimating the global Q-function.

$\sim$

Algorithm Networked Actor-Critic with Consensus Update (ConseNet)

Initialise: $\mu_{i,0}$, $\omega_{i,0}$, $\omega\tilde{}_{i,0}$, $\theta_{i,0}$ $\forall i \in \mathcal{V}$, $\{\beta_{\omega,t}\}_{t\geq 0}$ and $\{\beta_{\theta,t}\}_{t\geq 0}$;

Receive initial local observations $\{s_{i,0}\}_{i\in\mathcal{V}}$;

Select actions $a_{i,0} \sim \pi_{\theta_{i,0}}(\cdot|s_{i,0})$ for each $i \in \mathcal{V}$ and execute joint action $\mathbf{a}_0 = \cup_{i=i}^{N} a_{i,0}$;

Initialise step counter $t \leftarrow 0$;

repeat

  forall $i \in \mathcal{V}$ **do**

    Observe $s_{i,t+1}$ and $r_{i,t+1}$;

    Update $\mu_{i,t+1} \leftarrow (1-\beta_{\omega,t}) \cdot \mu_{i,t} + \beta_{\omega,t} \cdot r_{i,t+1}$;

    Sample $a_{i,t+1} \sim \pi_{\theta_{i,t}}(\cdot|s_{i,t+1})$;

  end

  Execute joint action $\mathbf{a}_{t+1} = \cup_{i=i}^{N} a_{i,t+1}$;

  forall $i \in \mathcal{V}$ **do**

    Update the local parameter-update estimate for the critic:

    $\delta_{i,t} \leftarrow (r_{i,t+1} - \mu_{i,t}) + Q(s_{t+1}, \mathbf{a}_{t+1}; \omega_{i,t}) - Q(s_t, \mathbf{a}_t; \omega_{i,t})$;

    $\tilde{\omega}_{i,t+1} \leftarrow \omega_{i,t} + \beta_{\omega,t} \cdot \delta_{i,t} \cdot \nabla_{\omega_i} Q(s_t, \mathbf{a}_t; \omega_{i,t})$;

    Update the actor parameters using the advantage estimate:

    $\hat{A}_{i,t} \leftarrow Q(s_t, \mathbf{a}_t; \omega_{i,t}) - \sum_{a_i \in \mathcal{A}_i} \pi_i(a_{i,t}|s_t) Q(s_t, a_{i,t}, \mathbf{a}_{-i,t}; \omega_{i,t})$;

    $\theta_{i,t+1} \leftarrow \theta_{i,t} + \beta_{\theta,t} \cdot \left[\nabla_{\theta_i} log(\pi_{\theta_{i,t}}) \hat{A}_{i,t}\right]$;

  end

  forall $i \in \mathcal{V}$ **do**

    $\omega_{i,t+1} \leftarrow \sum_{j\in\mathcal{V}} c_{ij} \cdot \tilde{\omega}_{j,t}$;

  end

  Update $t \leftarrow t+1$;

until *convergence*;

7.5 Differentiable Inter-Agent Learning (DIAL)

It has been argued that the primary differentiating factor between Homo Sapiens and other social animals is our aptitude for complex communication which has allowed us to organise in large groups whilst maintaining social order, and eventually dominate the planet (Harari, 2011). It makes sense then that in attempting to design a multi-agent algorithm one would consider giving agents the ability not only to share information, but to **learn** to communicate. With this in mind the fourth algorithm considered in this report is Differentiable Inter-Agent Learning, or **DIAL**, the first of the six MARL algorithms covered in this report which allows for explicit agent communication through differentiable channels. The DIAL algorithm was one of two proposed in the paper *"Learning to Communicate with Deep Multi-Agent Reinforcement Learning"* in 2016 [8], and is engineered to solve cooperative, partially observable multi-agent tasks. As discussed earlier, a fully cooperative task is one in which all agents share the goal of maximising the same return, hence any learned communication should theoretically be an attempt to coordinate or share useful information rather than deceive or distract. Then, partial observability, also discussed previously, means that each agent receives, instead of the true state,

observations which are correlated with the true state but do not allow for disambiguation. The authors of [8] state explicitly that "[they] are interested in [partially observable, fully cooperative settings] because it is only when multiple agents and partial observability coexist that agents have the incentive to communicate.", implying that the hope for DIAL is that agents learn a communication protocol in which useful information is shared to compensate for any one agents lack of complete knowledge about the true state, with the goal of collective success. A few things to note about the explicit communication allowed under DIAL: (1) no existing communication protocol is given a priori (as has been the case with much previous, related work in this area [8]), agents must learn an effective protocol for the task at hand by exploring an extremely high-dimensional abstract protocol space, making the task significantly challenging. Then, (2) unlike previously related work, DIAL uses discrete communication during execution, however, in order to facilitate end-to-end training allowing gradients to flow between all agents, messages are real-valued during training (this is explored in more detail below). DIAL is formally explained below in two parts; the forward pass, which happens during both training and execution, and the backward pass, which happens only during training.

$\sim$

The forward pass. Consider a partially observable Markov game $(n, \mathcal{S}, \{\Omega_i, \mathcal{O}_i, \mathcal{A}_i, r_i\}_{i=1}^n, p)$ involving n agents. Each agent has what the authors call a communication network, or C-Net, parameterised by θ_i which processes state information and outputs state-action values $Q_i(\cdot)$ and a message $\hat{m}_i$ at each time step. Then, during training, the message output of each agents C-Net feeds directly into the C-net of every other agent, meaning that instead of having n separate, disconnected parameterised networks/function-approximators, we have a single meta-network parameterised by $\boldsymbol{\theta} = [\theta_1^T, \theta_2^T, \ldots, \theta_n^T]^T$. This will allow for end-to-end training across all agents in the network, as it will allow gradients to flow between each agents network and backwards in time. However, for the forward pass we will only make reference to the C-Net of a single agent, since the forward pass does not require a single meta-network to be defined. This also means that, during execution, if the agents were operating on separate servers, physically separated but connected by a network, the algorithm would still be able to run efficiently. Now, onto the forward pass. During the forward pass of the DIAL algorithm, at each time step t each agent i receives an observation $o_{i,t}$ which is correlated with the true state of the environment, s_t. In addition each agent receives messages $m_{-i,t-1}$ from all other agents in the game, from the **previous** time step, $t-1$. Each agent also keeps track of outputs from it's own C-Net from the previous time step, namely it's previous action $a_{i,t-1}$ and a hidden state $h_{i,t-1}$ which is maintained by recurrent layers in the network, using each as additional input to the agents C-Net at time t, too. Both of these recurrent outputs are required to handle the assumption of partial observability; in order to allow the agent to condition it's action choices on more that just it's most recent observation. The C-Net belonging to agent i then processes the incoming information and produces, as stated above, the state-action values $Q_{i,t}(\cdot)$ and a real valued message $m_{i,t}$, which we assume for simplicity is just a real number, however it may be implemented as a higher dimensional, real-valued vector. In summary we have that the C-Net for agent i processes information at each time step t in the following way:

$$Q_{i,t}(\cdot), m_{i,t} = \text{C-Net}(o_{i,t}, m_{-i,t-1}, h_{i,t-1}, a_{i,t-1}, i; \theta_i^{(k)})$$

where i is the index of the agent. The architecture of each agents C-Net is shown in figure 10. The initial inputs are all processed in the following way. The observation $o_{i,t}$ is sent through a task-specific network, usually a

multi-layered perceptron (MLP), and the messages $\hat{m}_{-i,t-1}$ are sent (concatenated as a vector) through a single fully connected layer (FC), both producing embedding vectors of size 128.

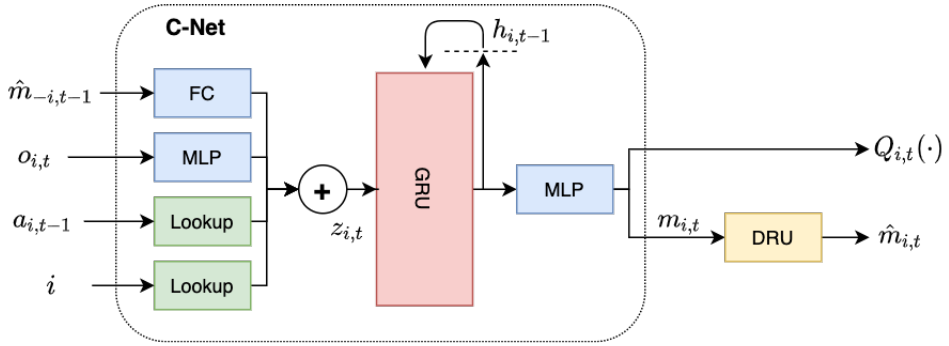


Figure 10: The C-Net for a single agent.

Additionally, the previous action $a_{i,t-1}$ and agent index i are passed through lookup tables to produce embedding vectors of the same size (it is not clear from the paper exactly why the agent index is required and used in this way but it presumably has a technical/implementation related explanation rather than a mathematical one). These embedding vectors are then added together element-wise to produce a new vector $z_{i,t}$ which is sent into the recurrent layer(s) along with the hidden state vector(s) from the previous time step. Finally, the output of the final recurrent layer is passed through a 2-layer MLP to produce the state-action values $Q_{i,t}(\cdot)$ and the message unprocessed message $m_{i,t}$, which is processed differently during execution and training, before being sent to the other agents at time $t+1$, in the following way. During training, the message is sent through the so-called *discretise/regularise unit* (DRU), in which Gaussian noise is added, where after the message is passed into the logistic function, that is, $\hat{m}_{i,t} = \text{DRU}(m_{i,t}) = \textit{Logistic}(\mathcal{N}(m_{i,t}, \sigma))$, where σ is set per-task (the authors found $\sigma = 2$ to be effective for the tasks to which DIAL was applied in the paper). Adding noise to the message before sending it regularises the learning of message protocols by each agent, that is, it prevents the agent from over-fitting on training data, resulting in more robust generalised performance during execution - see the supplementary material of [8] for a detailed explanation of why noise is essential, and a deeper analysis of its effects. During execution the output is binarised (the message signal may be 0 or 1 at each time step) in the following way: set $\hat{m}_{i,t} = \mathbb{I}\{m_{i,t} > 0\}$, where $\mathbb{I}$ is the indicator function. A binary message may be communicated using a single bit of information, making the sending and receiving of messages during execution very efficient. Action selection takes place using an ϵ-greedy policy in the usual way, using the state-action values output from each agents C-Net, and after executing the joint action a_t, the agents each receive the same common reward r_t since the task is fully cooperative. The backward pass of the DIAL algorithm is done only during training, and is covered below in detail.

$\sim$

The backward pass. During the backwards pass of the DIAL algorithm, the objective is to optimise the meta-network, comprised of the individual C-nets belonging to each agent, connected through differentiable communication channels, to (1) accurately predict the state-action values of individual agents (2) produce messages which contain useful information which will assist in predicting state-action values. This is done by first collecting a trajectory (batch) of experience, then stepping backwards in time and accumulating gradient information involving (1) partial derivatives of the well-known DQN error, computed using the bellman optimality

equation, with respect to network parameters and (2) partial derivatives of the message outputs of all agents with respect to the network parameters, multiplied by partial derivatives of the message outputs with respect to the future C-Net outputs of all agents, with the objective of optimising message protocols in order to more accurately predict state-action values. (1) is easy to understand as it does not differ much from the normal way partial derivatives are computed in single-agent DQN, but in order to understand (2) we need to examine the roll messages (outputs and inputs) play over time within a single trajectory. Notice that the messages sent from a an agent to all other agents at each time step are in fact outputs of the meta-network being fed back into itself as inputs recurrently. In this way a message sent from an agent to all other agents at some time t has an impact on the output of each agents C-Net at every time step $t' > t$; in other words the impact of all messages propagate through time, and in a way that is quite convoluted since messages at one time step are combined to make a single message in the next, and so on. See figure 11 for an illustration. This has a significant impact on the way gradients are computed during training since, as is the case with training any recurrent neural network, the network is "unrolled" over time (I know you've seen the LSTM illustration) and the gradients must be accumulated, starting at the final time step and moving backwards. Let's now take a closer look at how this is done mathematically.

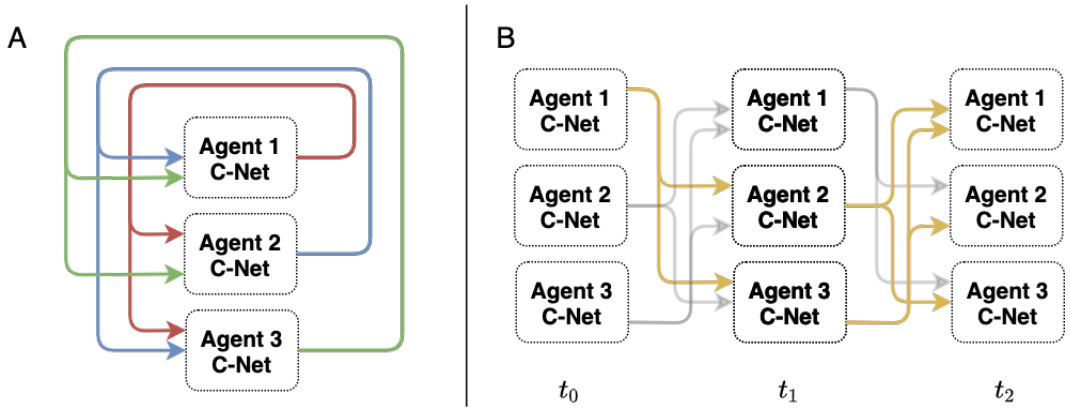


Figure 11: Message passing between agents in DIAL. **(A)** message outputs at each time step are fed, in a recurrent manner, as inputs in the following time step. **(B)** Message passing "unrolled" in time illustrating how information from one agent is propagated through time.

Before each backward pass, a single trajectory (since the time-ordering is important) is collected comprising of transition tuples from $t = 1$ to some final time step $t = T$. This is done using the forward pass as described above, and since the data is being collected for training, the messages sent by each agent are real-valued and are passed through the DRU before being sent to the other agents in the game. Additionally, before any gradients are computed, a zero vector/tensor $\nabla\theta$ is defined in order to store the gradients for the eventual gradient update step. Then, starting at $t = T$ and working backwards, for each agent i we compute and accumulate gradients for the transition tuple $(o_{i,t}, \hat{m}_{-i,t-1}, a_{i,t-1}, h_{i,t-1}, a_{i,t}, r_{i,t}, o_{i,t+1}, \hat{m}_{-i,t}, h_{i,t})$ in the following way. First, we compute the DQN target:

$$y_{i,t} = \begin{cases} r_{i,t} & \text{if } s_t \text{ is terminal} \\ r_{i,t} + \gamma \max_{a'} Q(o_{i,t+1}, m_{-i,t}, h_{i,t}, a_{i,t}, a', i; \theta_i^{(k)}) & \text{otherwise.} \end{cases} \tag{43}$$

where $\gamma \in [0, 1]$ is a temporal discount factor. Then we go on to accumulate the gradients for the action and

message chosen by agent i. We first compute the gradient of the DQN error with respect to the parameters of the meta-network θ (Note: this is done because a gradient vector/tensor which has the same shape as θ is required for the gradient update step is performed, however, the gradient will only be non-zero with respect to the parameters which belong to the C-Net of agent i, θ_i). The DQN Error is:

$$\Delta Q_{i,t} = y_{i,t} - Q(o_{i,t}, \hat{m}_{-i,t-1}, h_{i,t-1}, a_{i,t-1}, a_{i,t}, i; \tilde{\theta}_i^{(k)}) \tag{44}$$

where $\tilde{\theta}$ is a time-delayed copy of the network parameters, as per the singe-agent DQN algorithm. Then, the gradient is accumulated/added to the parameter tensor/vector variable:

$$\nabla\theta = \nabla\theta + \nabla_\theta \left(\Delta Q_{i,t}\right)^2 \tag{45}$$

Now, in addition to optimising the action-value approximations for agent i, we would like optimise the messages sent by agent i to the other agents in the game such that their state-action value estimates are more accurate, too. In order to do this we would like to compute a partial derivative of the following form (with an abuse of notation):

$$\frac{\partial}{\partial \hat{m}_{i,t}}(\text{future outputs}) = \frac{\partial}{\partial \hat{m}_{i,t}}(\Delta Q_{-i,t+1}) + \frac{\partial}{\partial \hat{m}_{i,t}}(\text{future outputs}(\hat{m}_{-i,t+1})) \tag{46}$$

$$= \frac{\partial}{\partial \hat{m}_{i,t}}(Q_{-i,t+1}) + \frac{\partial}{\partial \hat{m}_{-i,t+1}}(\text{future outputs})\frac{\partial}{\partial \hat{m}_{i,t}}(\hat{m}_{-i,t+1}) \tag{47}$$

In order to accomplish the above, we keep track of a so-called 'gradient chain', $\mu_{i,t}$, for each agent i, which essentially keeps track of the impact of the message output of each agent at time t_+1 on all future network outputs ($> t+1$). We update the gradient chain for agent i as follows:

$$\mu_{i,t} = \begin{cases} 0 & \text{if } t > T-2 \\ \sum_{j \neq i} \frac{\partial}{\partial \hat{m}_{i,t}} \left(\Delta Q_{j,t+1}\right)^2 + \mu_{j,t+1} \frac{\partial}{\partial \hat{m}_{i,t}} \left(\hat{m}_{j,t+1}\right) & \text{otherwise.} \end{cases} \tag{48}$$

where we need two future time steps worth of information to be able to compute the DQN error at $t+1$, hence we make the condition that $\mu_{i,t}$ is zero until $t < T-2$. Finally we add to $\nabla\theta$ the partial derivative of $\hat{m}_{i,t}$ with respect to the network parameters, multiplied by the value of the gradient chain computed above, along with the the partial derivative of the DRU function with respect to the un-processed message, as required by the chain rule:

$$\nabla\theta = \nabla\theta + \mu_{i,t}\frac{\partial}{m_{i,t}}\text{DRU}(m_{i,t})\nabla_\theta \hat{m}_{i,t} \tag{49}$$

which will cause the networked to be optimised such that agent i will produce messages which propagate through the network and through time to produce more accurate state-action value estimates from all agents. We repeat this process of gradient accumulation for all n agents and finally perform the gradient update step:

$$\theta^{(k+1)} = \theta^{(k)} - \alpha\nabla\theta \tag{50}$$

where α is a small, positive step size parameter. The DIAL algorithm is given below in full.

~

Algorithm 4 DIAL

Parameters: ϵ, σ, α, γ;

Initialise: $\boldsymbol{\theta}^{(1)}$, $\tilde{\boldsymbol{\theta}}^{(1)}$;

for *each episode* e **do**

initial state = s_1;

set $t \leftarrow 0$, $h_{i,1} \leftarrow \mathbf{0}$, $m_{i,0} \leftarrow 0$ and $a_{i,0}$ randomly for each agent i

while $s_t \neq$ *terminal and* $t \leq T$ **do**

$t \leftarrow t + 1$;

for *each agent* i **do**

observe $o_{i,t}$ and receive $\hat{m}_{-i,t-1}$ from all other agents;

$Q_{i,t}(\cdot), m_{i,t} \leftarrow \text{C-Net}(o_{i,t}, m_{-i,t-1}, h_{i,t-1}, a_{i,t-1}, i; \theta_i^{(k)})$;

Select $a_{i,t}$ according to an ϵ-greedy policy;

$$\hat{m}_{i,t} \leftarrow \begin{cases} \mathcal{N}(m_{i,t}, \sigma) & \text{if training} \\ \mathbb{I}\{m_{i,t} < 0\} & \textit{otherwise} \end{cases}$$

end

execute joint action a_t and observe reward $r_{i,t} = r_t$ (same for each agent);

new state = s_{t+1};

end

if *training* **then**

reset gradients $\nabla\theta \leftarrow \mathbf{0}$;

for *t=T to 1* **do**

for *each agent* i **do**

$$y_{i,t} \leftarrow \begin{cases} r_t & \text{if } s_t \text{ is terminal} \\ r_t + \gamma \max_{a'} Q(o_{i,t+1}, m_{-i,t}, h_{i,t}, a_{i,t}, a', i; \theta_i^{(k)}) & \text{otherwise} \end{cases}$$

$$\Delta Q_{i,t} \leftarrow y_{i,t} - Q(o_{i,t}, \hat{m}_{-i,t-1}, h_{i,t-1}, a_{i,t-1}, a_{i,t}, i; \tilde{\theta}_i^{(k)});$$

$$\mu_{i,t} \leftarrow \begin{cases} 0 & \text{if } t > T-2 \\ \sum_{j \neq i} \frac{\partial}{\partial \hat{m}_{i,t}} \left(\Delta Q_{j,t+1}\right)^2 + \mu_{j,t+1} \frac{\partial}{\partial \hat{m}_{i,t}} \left(\hat{m}_{j,t+1}\right) & \text{otherwise} \end{cases}$$

$$\nabla\theta \leftarrow \nabla\theta + \nabla_\theta \left(\Delta Q_{i,t}\right)^2 + \mu_{i,t} \frac{\partial}{m_{i,t}} \text{DRU}(m_{i,t}) \nabla_\theta \hat{m}_{i,t};$$

end

end

$\theta^{(k+1)} = \theta^{(k)} - \alpha\nabla\theta$;

end

every C steps set $\tilde{\theta} \leftarrow \theta$;

end

8 CommNet

The 5^{th} MARL algorithm, which will be referred to as **CommNet** from here on, was proposed in the paper *"Learning Multiagent Communication with Backpropagation"* [21] in 2016, and is, as the title suggests, also

an algorithm which employs differentiable communication links between multiple agents to enable information sharing through a learned protocol, with the objective of improved multi-agent learning in cooperative MARL tasks. The philosophy about how communication takes place, and hence the architecture of the network designed in order to implement CommNet, differs significantly from DIAL in two ways: (1) instead of communication between agents occurring from one time step to the next, the authors have designed CommNet in such a way that communication takes place within each time step during both execution and training, resulting in a much simpler network architecture than DIAL which takes in state information and outputs a distribution over actions for each agent; and (2) CommNet is designed in such a way that the number of agents may change dynamically during execution and/or training. (1) may raise questions in the readers mind about how CommNet deals with partial observability since if agents only communicating within, but not across, time steps there does not appear to be a way to condition the agents' policies on the full history of actions and partial observations. The authors of [21] do not deal explicitly with partial observability in the core definition of the algorithm, but rather suggest an extension of the proposed network architecture to include recurrent layers (e.g. an LSTM); for this reason CommNet is defined below without the inclusion of recurrent layers, however a possible concrete suggestion for their inclusion is covered at the end of this section. Point (2) is a meaningful consideration for the authors to make, since many possible real-world multi-agent learning tasks, to which MARL algorithms could be applied, may require the number of participating, communicating agents to change dynamically over the duration of the task. One obvious example of this is self-driving cars; if each car were controlled by a software RL agent and was tasked to drive safely from point A to point B along city streets, it would indeed move in and out of range of several other cars during it's journey, with which temporary communication links would need to be established and then terminated in quick succession. What follows is a formal description of the CommNet algorithm, however, the process of training will not be covered in detail since the meta-network architecture may be trained using a policy-gradient method very similar to the single-agent case (this will become clear as we proceed).

$\sim$

Consider a partially observable Markov game $(n, \mathcal{S}, \{\Omega_i, \mathcal{O}_i, \mathcal{A}_i, r_i\}_{i=1}^n, p)$ involving n agents. At each time step t each agent i receives an observation $o_{i,t} \in \otimes_i$. Then, in order to select a joint action $\mathbf{a}_t = \{a_{1,t}, a_{2,t}, \ldots, a_{n,t}\}$, the joint observation $\mathbf{o}_t = \{o_{1,t}, o_{2,t}, \ldots, o_{n,t}\}$ is passed through a central controller $\Phi : \Omega \to \mathcal{A}$, which is a mapping from the joint observation space to the joint action space. The controller $\Phi_\theta(\mathbf{o}_t) = \mathbf{a}_t$, as described in [21], is a meta-network parameterised by θ, which processes the observation information and facilitates inter-agent communication in the following way. First, the observation received by each agent i at each time step t is passed through an encoding function e, simply a fully-connected layer, to produce a hidden-state vector $h_{i,t}^0 = g(o_{i,t})$, which may be viewed as the 0^{th} layer of Φ_θ. Note that the same encoding function e is used for all agents (the reason for this is discussed below in more detail). Once the observations of all agents have been encoded, each encoding is passed as input into a sequence of modules f^j, $j = 0, 2, \ldots, K-1$, where j represents the position of the module in the sequence, each of which may take the form of a multi-layered neural network, where the output of module $j-1$ is fed as input to module j. Suppressing the time index for brevity, each module f^j takes two inputs belonging to each agent i; a real-valued communication vector c_i^j and the hidden state vector h_i^j, output from the previous module f^j, to produce a c_i^{j+1} and h_i^{j+1} as follows:

$$h_i^{j+1} = f^j(h_i^j, c_i^j) \tag{51}$$

$$c_i^{j+1} = \frac{1}{n-1} \sum_{i' \neq i} h_{i'}^{j+1} \tag{52}$$

In words, the communication vector sent by agent i, output from module j, is an average of the communication vectors received by all other agents in the previous time step. Note that the initial communication vector is assumed to be the zero vector, i.e. $c_i^j = \mathbf{0}$ for each agent i. Then, for simplicity, suppose f^j consists of single layer consisting of two matrices H^j and C^j such that the communication and hidden state vectors are transformed as:

$$h_i^{j+1} = \sigma(H^j h_i^j + C^j c_i^j) \tag{53}$$

where σ is a non-linear activation function; in this case the authors use the tanh function. Note that the parameters of the module f^j for each $j = 0, 2, \ldots, K-1$ in Φ_θ are shared by each agent i (i.e. the matrices C^j and H^j are used to transform the j^{th} hidden state and communication vectors of all n agents). The reason this decision was made was to allow for the dynamic raising or lowering of the number of agents n without having to add or remove additional parameters (e.g. weight matrices) to/from Φ. Finally, the hidden state vector h_i^K, corresponding to agent i, is output from the the final module, f^{K-1}, after which it is passed through a decoding layer $q(\cdot)$, which is simply a fully connected layer, to produce a multinomial distribution over actions. At each time t we sample a discrete action for agent i from this distribution: $a_{i,t} \sim q(h_i^K)$. An illustration of the full model Φ_θ is shown in figure 12.

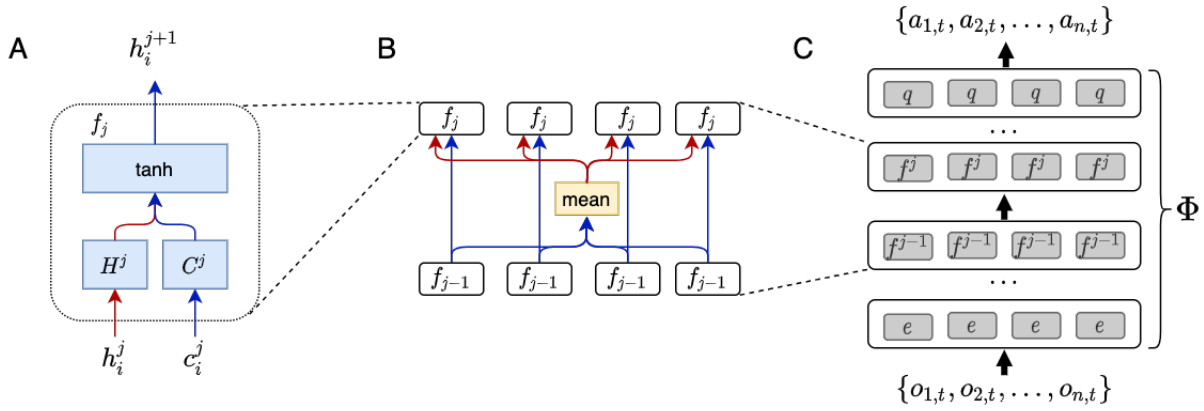


Figure 12: CommNet illustrated. **(A)** Input corresponding to agent i being processed through module f^j. **(B)** Each agent receives a message which is an average of the messages from all other agents. **(C)** The full Φ model, a large feed forward meta-network which processes observations and outputs actions for each agent.

With respect to the training of Φ_θ, which may be viewed as a joint, parameterised policy; the same methods used in the single-agent policy gradient algorithms to optimise the parameters of policies of single agents may be used. That is, we may define an objective function as the expected return received by each agent over some time horizon, which obviously depends on θ, and compute the gradient of the objective function in the same way as is done in the single agent case. It would also be possible to condition a value function $Q_\omega(\cup_{i=1}^n o_i, \cup_{i=1}^n a_i)$ to estimate the value of joint actions based on joint observations and use it as a baseline, in a similar way to the IA2C algorithm explained above. In summary we may simply view Φ_θ as a multi-agent parameterised policy

implemented as a large feed-forward network which may be trained using simple supervised learning methods. Finally, with respect to the explicit handling of partial observability, a simple strategy would be to implement each module f^j using two LSTM (or GRU) units in place of the matrices H^j and C^j, for example computing the $j + 1^{th}$ hidden state for agent i as:

$$h_i^{j+1} = \sigma\left(\text{LSTM}_H(h_i^j) + \text{LSTM}_C(c_i^j)\right)$$

where σ is the tanh activation function. In this way each module in Φ_θ would retain two hidden state vectors of their own, one pertaining to the hidden states of each agent and another pertaining to their communication vectors, which would allow the network to condition it's output on the history of all observations and action choices of all agents as desired. This is not outlined explicitly in the [21], but merely suggested as an extension. Therefore there may be other more effective strategies, but each would entail a similar implementation to the one shown above. One final point to be made about the CommNet algorithm is that, although Φ_θ would need to be trained in a centralised fashion, as the same network is used to process observation information from all agents, the algorithm may be executed in a decentralised fashion after training since one could simply copy the network parameters and have the agents pass messages over a network, with each agent aggregating the received messages within each time step. It should be reiterated that since the messages are aggregated, the number of agents participating in the task may change dynamically even (and especially) during decentralised execution.

9 NeurComm

The 6^{th} and final algorithm we cover in this report is special in that it incorporates some of the key features from the first 5 algorithms, applying them in a refined and relatively simple way to produce a model which yields state of the art performance on *networked system control* (NSC) tasks. The algorithm is called *NeurComm*, and was proposed in the paper *Multi-Agent Reinforcement Learning for Networked System Control* [11], in 2020. NeurComm was developed specifically for application to NSC tasks, which are cooperative multi-agent control tasks wherein each agent in a connected graph assumes local control of a subset of a larger networked system. In NSC tasks cooperation is achieved by the sharing of local information, and explicit communication, between neighbours. A simple example of a multi-agent NSC task considered in [11] is the canonical MARL traffic light control problem, where each agent assumes control of the phase of traffic lights in each of a number of intersections in a connected traffic grid, with the objective being for all agents to coordinate in such a way that the traffic congestion (the density of cars) in the grid is minimized. In order to handle problems of this nature, the Neurcomm algorithm, similarly to ConseNet, was designed as a *networked actor critic* (NA2C) algorithm where each agent maintains it's own decision making policy (actor) and state-value function (critic), and is additionally embedded in a graph where each agent is a node, corresponding to the points of control in the networked system. The difference between NeurComm and ConseNet is that the edges which constitute connections between pairs of agent are not weighted since the optimisation of the critic of each agent doesn't occur through a consensus update, but rather, agents in NeurComm condition both their actor and critic networks on local information shared between neighbours in the network/graph. There are two key aspects which make NeurComm especially effective when applied to NSC problems. (1) The conditioning of actors and critics on local information (e.g. other agents actions) from the closed, but overlapping neighbourhoods of each agent, as opposed to the global sharing of information, supports agent coordination by allowing information to propagate throughout the graph

and, in a similar way to ConseNet, make it easier for cooperating agents to find a stable equilibrium with respect to their policies. Then, (2) in order to stabilise learning, the authors of [11] designed Neurcomm in such a way that agents receive a not only their own reward, but a linear combination of rewards from all agents in the graph, and propose a novel spatial reward discount factor α (as a complement to the usual temporal discount factor γ), which is used when computing returns for each agent in order to down-weight the rewards corresponding to agents which a further distance away (here distance is defined as the shortest path along the graph between nodes/agents). The introduction of the spatial discount factor means that Neurcomm can be applied to tasks which are either fully cooperative, in which case α would be set such that all agents would receive the same reward signal, or to tasks which are competitive, or mixed, by tuning α such that each agent would behave with a higher degree of self-interest.

$\sim$

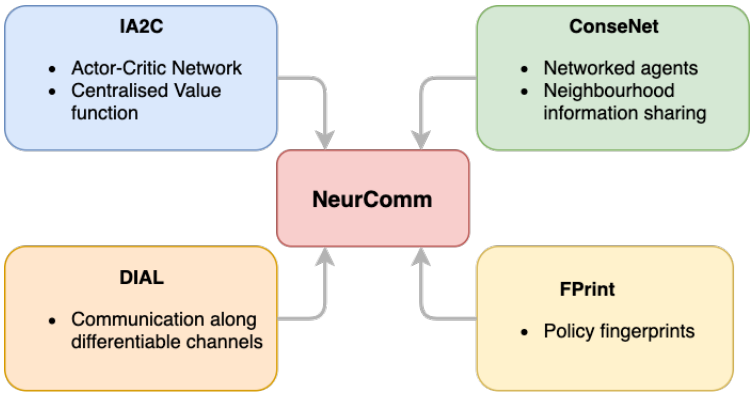


Figure 13: The NeurComm algorithm utilizes key features of the IA2C, ConseNet, DIAL and FPrint algorithms.

In addition to the aspects of NeurComm described above, the algorithm combines some of the key features of the previous MARL algorithms explored in this report; see Figure 13 for an illustration. As mentioned above, NeurComm is similar to ConseNet in that it considers agents embedded in a graph structure, where information is shared between neighbours. Then, similarly to both IA2C and ConseNet, in NeurComm each agent maintains it's own value function which is conditioned on, in addition to it's own actions, the actions of other agents in the game, however in NeurComm this only includes actions of neighbouring agents, not all agents. NeurComm also allows for explicit communication along differentiable channels, as per DIAL, but only between neighbouring agents and not all agents in the game. Additionally, communication protocols are also learned, and not prescribed. Then, finally, NeurComm allows neighbouring agents to share, and use as additional state information, so-called *policy fingerprints* which are static policies (multinomial distributions over possible actions represented as real-valued vectors), which is obviously inspired by the FPrint algorithm.

$\sim$

What follows is a detailed description of the NeurComm algorithm. Consider a partially observable, networked Markov game $(n, G, \mathcal{S}, \{\Omega_i, \mathcal{O}_i, \mathcal{A}_i, r_i\}_{i=1}^n, p)$. Assume the state and observation spaces $\mathcal{S}$ and $\Omega = \times_{i=1}^n \Omega_i$ are continuous, and that the set $\mathcal{A}_i$ belonging to each agent i consists of discrete actions. The edge set $\mathcal{E}$, defining the topology of G and thus the local neighbourhoods of each agent $i \in \mathcal{V}$, is task dependant and is therefore not

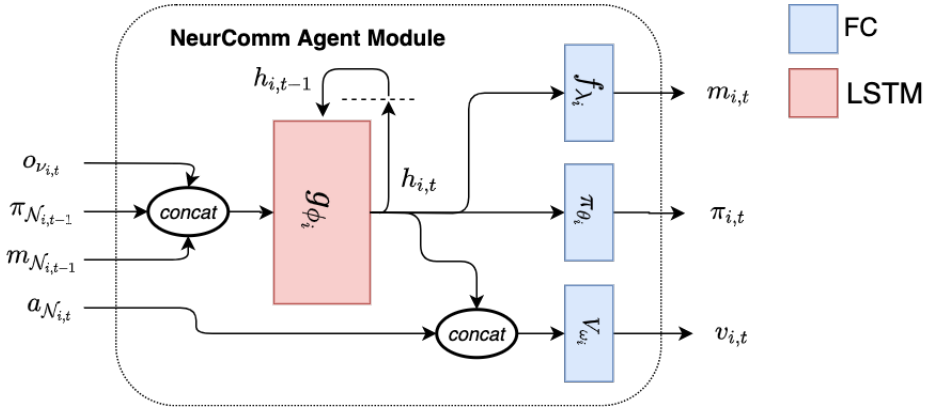


Figure 14: The agent module architecture for the NeurComm algorithm.

explicitly defined in general; rather, it is assumed that the graph is arbitrary, but connected. Furthermore, the edges are not weighted as per ConseNet.

NeurComm architecture. Under NeurComm, each agent i is associated with a single, connected deep-neural network (DNN), which processes local state information, as input, and generates, as output, the agents static policy π_i, a state value estimate v_i, and an outgoing message m_i. The DNN architecture for agent i, shown in 14, consists of an LSTM layer g_{ϕ_i}, which processes incoming state information and maintains a hidden state h_i conditioned on the full history of state information received by the agent (this is necessary since a partially observable Markov game is assumed), as well as three fully connected layers π_{θ_i}, V_{ω_i} and f_{λ_i}, which take h_i as input and produce the static policy, state value estimate and message for agent i, respectively. As with DIAL, since the message outputs from each agents DNN feed directly into the DNNs of neighbouring agents, together this constitutes a single, large meta-network which may be jointly optimized, since gradients may flow through between agent networks and backwards through time. Below the execution and training phases of the NeurComm algorithm are described.

$\sim$

Execution. At each time step t, each agent i begins by computing and broadcasting a message $m_{i,t} = f_{\lambda_i}(h_{i,t-1})$, using the hidden state from the previous time step, to all of it's neighbours. Then, each agent i receives the following neighbourhood state information as input: $o_{\nu_{i,t}}$, the local observations from the vicinity of agent i (the set of neighbours of agent i, including agent i); $\pi_{\mathcal{N}_i,t-1}$, the static policies (real-valued vectors) from each agent in the neighbourhood of agent i from the previous time step; $m_{\mathcal{N}_i,t-1}$, the real-valued message vectors from all of the neighbours of agent i from the previous time step, and; $h_{i,t}$ the hidden belief vector belonging agent i from the previous time step. The hidden belief vector is then updated as $h_{i,t} = g_{\phi_i}(o_{\nu_{i,t}} \cup \pi_{\mathcal{N}_i,t-1} \cup m_{\mathcal{N}_i,t-1}, h_{i,t-1})$, which is then used to update the static policy vector $\pi_{i,t} = \pi_{\theta_i}(\cdot|h_{i,t})$, whereafter the action $a_{i,t} \sim \pi_{i,t}$ is sampled. Once the actions are determined for each agent at time t, the state-value vector is updated for each agent i as $v_{i,t} = V(h_{i,t}, a_{\mathcal{N}_i})$, whereafter the joint action $\mathbf{a}_t = (a_{1,t}, a_{2,t}, \ldots, a_{n,t})$ is executed and each agent i receives it's numerical reward $r_{i,t}$ and an updated observation $o_{i,t+1}$. As stated above, one of the strengths of NeurComm, similarly to ConseNet, is the allowance made for local information to propagate throughout the graph since, the set of closed neighbourhoods around each agent are closed, meaning that every hidden state is conditioned on delayed global information. To see this, consider the following. Define $x \supset y$ to mean that y is used to estimate x. Then under the NeurComm algorithm we know $m_{i,t} \supset h_{i,t-1}$, and $h_{i,t} \supset h_{i,t-1} \cup s_{\nu_i,t} \cup \pi_{\mathcal{N}i,t-1} \cup m_{\mathcal{N}_i,t}$. Hence,

$$
\begin{aligned}
h_{i,t} &\supset s_{i,t} \cup \left\{s_{j,t}, \pi_{j,t-1}\right\}_{j\in\mathcal{N}_i} \cup \left\{h_{j,t-1}\right\}_{j\in\nu_i} \\
&\supset s_{i,t} \cup \left\{s_{j,t}, \pi_{j,t-1}\right\}_{j\in\mathcal{N}_i} \cup \left\{s_{j,t-1} \cup \left\{s_{k,t-1}, \pi_{k,t-1}\right\}_{k\in\mathcal{N}_j} \cup \left\{h_{k,t-2}\right\}_{k\in\nu_j}\right\}_{j\in\nu_i} \\
&= s_{i,t-1:t} \cup \left\{s_{j,t-1:t}, \pi_{j,t-2:t-1}\right\}_{j\in\mathcal{N}_i} \cup \left\{s_{j,t-1}, \pi_{j,t-2}\right\}_{j\in\{\mathcal{V}|d_{ij}=2\}} \cup \left\{h_{j,t-2}\right\}_{j\in\{\mathcal{V}|d_{ij}\leq 2\}} \\
&\supset \dots \\
&\supset s_{i,0:t} \cup \left\{s_{j,0:t}, \pi_{j,0:t-1}\right\}_{j\in\mathcal{N}_i} \cup \left\{s_{j,0:t-1}, \pi_{j,0:t-2}\right\}_{j\in\{\mathcal{V}|d_{ij}=2\}} \cup \dots \cup \left\{s_{j,0:t+1-d_{\max}}, \pi_{j,0:t-d_{\max}}\right\}_{j\in\{\mathcal{V}|d_{ij}=d_{\max}\}}
\end{aligned}
$$

which should, in theory, facilitate convergence to equilibrium during training. Then, if in training mode, a memory buffer $\mathcal{B}$ is used to hold a batch of experience collected during execution, and it is updated at the end of every time step t as $\mathcal{B} = \mathcal{B} \cup \{(o_{i,t}, \pi_{i,t-1}, a_{i,t}, r_{i,t}, v_{i,t})\}_{i\in\mathcal{V}}$; this batch of experience is then used during the training process to compute returns and gradients, and to perform optimization on the broader meta-network, which is described below.

$$\sim$$

Training. After a sufficient number of transition tuples (determined by a parameter set according to each task) have been collected and stored in the buffer $\mathcal{B}$ we obtain the minibatch $\{(o_{i,\tilde{t}}, \pi_{i,\tilde{t}-1}, a_{i,\tilde{t}}, r_{i,\tilde{t}}, v_{i,\tilde{t}})\}_{i\in\mathcal{V},\tilde{t}\in\mathcal{B}}$, which is used to compute the discounted return and advantage estimates, $\hat{R}_{\tilde{t},i}$ and $\hat{A}_{\tilde{t},i}$, for each agent i and for each time step $\tilde{t}$ (in which experience was collected) are computed as

$$\hat{R}_{i,\tilde{t}} = \sum_{k=\tilde{t}}^{T_{\mathcal{B}}-1} \gamma^{k-\tilde{t}} \left(\sum_{j\in\mathcal{V}_i} \alpha^{d_{ij}} r_{j,k} \right) + \gamma^{T_{\mathcal{B}}-\tilde{t}} v_{i,T_{\mathcal{B}}} \tag{54}$$

$$\hat{A}_{i,\tilde{t}} = \hat{R}_{i,\tilde{t}} - v_{i,\tilde{t}} \tag{55}$$

where $T_{\mathcal{B}}$ is the final time step represented recorded in the $\mathcal{B}$, γ and α are the temporal and spatial discount factors, respectively, and $d_{i,j}$ is the shortest distance between agent i and j in G. The final term in expression for the return, $\gamma^{T_{\mathcal{B}}-\tilde{t}} v_{i,T_{\mathcal{B}}}$, is the (temporally) discounted value of the final state in the buffer as estimate by the state-value function of agent i. One important note is that the above assumes the task is on-going in nature, and that if the task is finite and the the mini-batch contains a terminal state which is is not at index $T_{\mathcal{B}}$ in the buffer, one would need to split the mini-batch up into separate mini-batches where the state at $T_{\mathcal{B}}$ in each mini-batch was terminal and compute the discounted returns and advantages as above. This would not affect the remainder of the learning algorithm. When the estimates for the discounted return and advantage have been computed, gradients are computed and accumulated for each agent and each time step in the buffer, according to each agent's local loss function which takes the form:

$$\mathcal{L}_i = \mathcal{L}_i^{\text{policy}} + \mathcal{L}_i^{\text{entropy}} + \mathcal{L}_i^{\text{value}}$$

which is a sum of three distinct loss functions. This is acceptable since the distinct gradient operators which will act on this expression to compute gradients with respect to the policy, state-value, message and LSTM parameters will cause unrelated terms to go to zero. The separate loss functions take the form:

$$\mathcal{L}_i^{\text{policy}} = \frac{1}{|\mathcal{B}|} \sum_{\tilde{t} \in \mathcal{B}} \Big(-log \pi_{\theta_i}(a_{i,\tilde{t}} | h_{i,\tilde{t}}) \hat{A}_{i,\tilde{t}} \Big) \tag{56}$$

$$\mathcal{L}_i^{\text{entropy}} = \beta \sum_{a_i \in \mathcal{A}_i} \big(\pi_{\theta_i}(a_{i,\tilde{t}} | h_{i,\tilde{t}}) log \pi_{\theta_i}(a_{i,\tilde{t}} | h_{i,\tilde{t}}) \big) \tag{57}$$

$$\mathcal{L}_i^{\text{value}} = \frac{1}{|\mathcal{B}|} \sum_{\tilde{t} \in \mathcal{B}} \Big(\hat{R}_{i,\tilde{t}} - V_{\omega_i}(h_{i,\tilde{t}}, a_{\mathcal{N}_i,\tilde{t}}) \Big) \tag{58}$$

$$\tag{59}$$

where $\beta << 1$ is a weighing coefficient. Here the entropy loss is simply a regularising term which penalises the policy for becoming too certain or confident in it's distribution of probabilities over actions, which ensures a certain degree of exploration is maintained when the agent is collecting experience. Similarly to DIAL, the meta-DNN consisting of the networks corresponding to each agent, connected via differentiable communication channels, is in essence a single, large and complex recurrent neural network. As such, the network must be 'unrolled' over time during training and gradients accumulated while moving backwards in time. The full NeurComm algorithm is given below.

$\sim$

Algorithm 6 NeurComm

Parameters: α, β, γ, T ;

Initialise: $o_{i,0}, \pi_{i,-1}, h_{i,-1}, t \leftarrow 0, k \leftarrow 0, \mathcal{B} \leftarrow \emptyset$ for all $i \in \mathcal{V}$;

repeat

  for $i \in \mathcal{V}$ **do**

    send $m_{i,t} = f_{\lambda_i}(h_{i,t-1})$;

  end

  for $i \in \mathcal{V}$ **do**

    observe $\tilde{o}_{i,t} = o_{\nu_{i,t}} \cup \pi_{\mathcal{N}_i,t-1} \cup m_{\mathcal{N}_i,t-1}$;

    update $h_{i,t} \leftarrow g_{\phi_i}(\tilde{o}_{i,t}, h_{i,t-1})$, $\pi_{i,t} \leftarrow \pi_{\theta_i}(\cdot|h_{i,t})$;

    sample $a_{i,t} \sim \pi_{i,t}$;

  end

  for $i \in \mathcal{V}$ **do**

    update $v_{i,t} \leftarrow V_{\omega_i}(a_{\mathcal{N}_i,t}, h_{i,t})$;

  end

  execute $\mathbf{a}_t = (a_{1,t}, a_{2,t}, \ldots, a_{n,t})$;

  observe $\{o_{i,t+1}, r_{i,t}\}_{i \in \mathcal{V}}$;

  update $\mathcal{B} \leftarrow \mathcal{B} \cup \{(o_{i,t}, \pi_{i,t-1}, a_{i,t}, r_{i,t}, v_{i,t})\}_{i \in \mathcal{V}}$;

  update $t \leftarrow t + 1, k \leftarrow k + 1$;

  if t=T **then**

    initialise $o_{i,0}, \pi_{i,-1}, h_{i,-1}, t \leftarrow 0$ for all $i \in \mathcal{V}$;

  end

  if $|\mathcal{B}| = k$ **then**

    compute $\hat{R}_{\tilde{t},i}$ and $\hat{A}_{\tilde{t},i}$;

    for $i \in \mathcal{V}$ **do**

      update $\{\lambda_j, \phi_j\}_{j \in \mathcal{V}} \cup \{\theta_i, \omega_i\}$, based on gradients computed from $\mathcal{L}_i$;

    end

    initialise $\mathcal{B} \leftarrow \emptyset, k \leftarrow 0$;

  end

until *stop condition is reached*;

Part C: MARL Applied to Sequential Social Dilemmas

What follows are the details of a two basic experiments where the six MARL algorithms covered in Part B were applied to two so-call sequential social dilemma (SSD) MARL tasks, Cleanup and Harvest.

10 Sequential Social Dilemma Tasks: Cleanup and Harvest

As defined above, a social dilemma is a multi-player game "in which the non-cooperative payoff for a player exceeds the cooperative payoff. However, if most players in the game fail to cooperate, all players suffer". In real life, social dilemmas often arise not as one-shot, but rather as temporally extended interactions between multiple players where the state of the environment in which the players interact changes in response to collective action. Examples of such tasks, where human beings are concerned, include the consumption of shared (common-pool) resources such as fresh water lakes or forests. As human beings we must cooperatively manage shared, finite resources such as these, even though they are regenerative, in such a way that our consumption is sustainable over time, lest we run out. Clearly in the case of common-pool resource management the temptation always exists for one player in a group of cooperators to defect and consume more than they should, however over-consumption by all players in the game will lead to deeply undesirable outcomes. *Cleanup* and *Harvest* are two multi-agent SSD tasks which simulate similar scenarios to the common-pool resource problem described above, where agents are rewarded for consuming potions of a shared resource but must learn to coordinate in some way in order to achieve sustainable, long term gains.

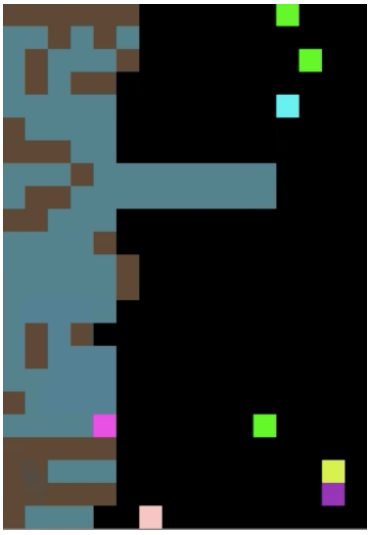


Figure 15: The SSD MARL task "Cleanup"; agents are rewarded for consuming apples but must clean up the river in order for apples to grow. The river and the mud are shown on the left, while the apples are the bright green pixels. The remaining multi-coloured pixels are agents.

Cleanup. A public goods dilemma in which agents get a reward for consuming apples, but must use a cleaning beam to clean a river in order for apples to grow. While an agent is cleaning the river, other agents can exploit it by consuming the apples that appear. Agents must learn and act based off only partial state information; at each time step each agents observes the 15×15 grid of pixels surrounding it.

~

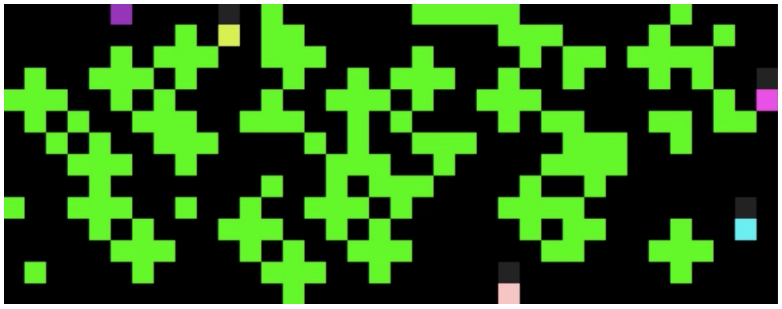


Figure 16: The SSD MARL task "Harvest"; agents are rewarded for consuming apples, but apples regenerate at a rate proportional to the number of apples remaining. The apples are the bright green pixels and the remaining multi-coloured pixels are agents.

Harvest. A tragedy-of-the-commons dilemma in which apples regrow at a rate which is proportional to the amount of apples which already exist. If individual agents employ an exploitative strategy by greedily consuming too many apples, the collective, long term reward of all agents is reduced. As with Cleanup, agents must learn and act based off only partial state information; at each time step each agents receives the 15×15 grid of pixels surrounding it.

11 Methods

The core algorithmic techniques of the six MARL algorithms covered in this report were used to adapt the same *Networked Actor-Critic* (NA2C) architecture used as the framework for NeurComm. A neat summary of the architecture, as well as the different state information utilised by each agent, denoted $I_{i,t}$, at each time step to compute its static policy and state-value estimate is illustrated in Figure 17. Note that under the CommNet adaptation, all message information received by each agent i from $\mathcal{N}_i$ is averaged to create $\overline{m}_{\mathcal{N}_i,t}$. Each algorithm computes returns and advantage estimates in the same way as NeurComm and computes gradients according to the same loss functions used in NeurComm. Gradients are also applied in a similar way to NeurComm for each algoithm with the exception of ConseNet. In the ConseNet adaptation of NA2C, updates to the LSTM parameters ϕ_i, for each agent i, are done by consensus update within the vicinity ν_i. The training set up is as follows for all six algorithms. Five agents were trained for for $2e7$ time steps on both Harvest and Cleanup. Since group success depends on cooperation the spatial discount parameter α is set to 1.0 for the duration of training, meaning that all agents received the same, global reward at each time step. Furthermore the agent graph G for each algorithm is complete, meaning that each agent is adjacent to each other agent; considering neither task is an NSC task it makes sense that all agents should share information in a single neighborhood. Further parameter settings can be seen on the GitHub repository, the link to which is provided in Appendix A.

12 Results

The training results for Cleanup and Harvest are shown in Figure 18 and Figure 19, respectively. Each plot shows mean episodic return on the Y-axis and number of training steps on the X-axis. An important note to make upfront is that not all algorithms were trained for the same number of training steps. The reasons for this are (1) training one algorithm for an average of 20 million steps takes about a week on a GPU, and training time

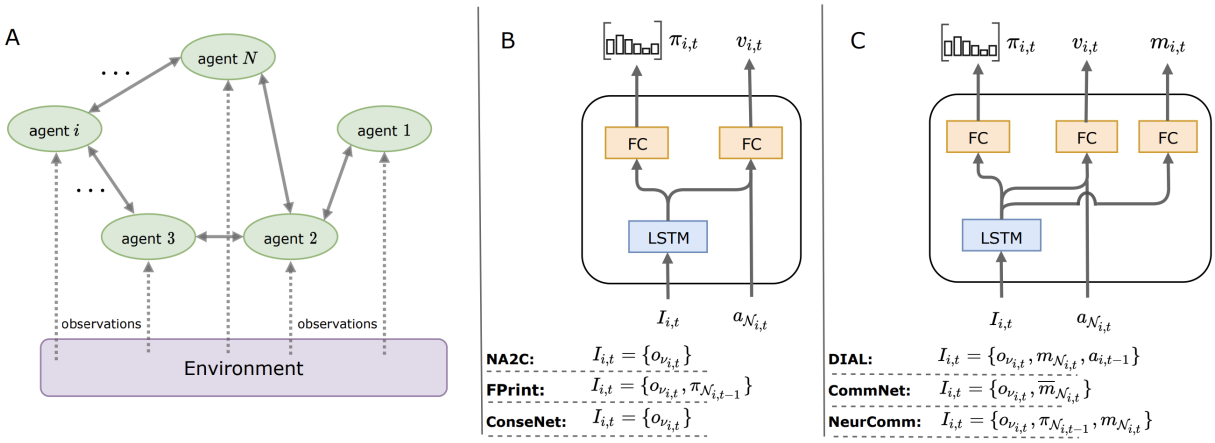


Figure 17: Networked multi-agent reinforcement learning algorithms. **(A)** Decentralised networked multi-agent system. Agents receive observations from the environment and share information with their neighbours. **(B)** Agent module for networked non-communicative algorithms (*NA2C*, *FPrint*, *ConseNet*). The agent module receives neighbourhood observation, action and/or policy information in order to compute a local policy and state-value estimates. **(C)** Agent module for networked communicative algorithms (*DIAL*, *CommNet*, *NeurComm*). The agent module has an additional fully-connected (FC) layer for message passing along differentiable communication channels.

was finite, (2) NA2C and FPrint agents were implemented as separate deep neural networks, while the remaining algorithms were implemented as a single meta-DNN, meaning that they were a lot less computationally expensive to train, despite having fewer parameters.

~

Cleanup. Regarding the Cleanup results, all algorithms appear to take about 500'000 steps before making any improvement. Both FPrint and ConseNet achieve scores exceeding the state-of-the-art (SOTA) results on this particular task, with CommNet not far behind. Interestingly, DIAL and NeurComm, the two algorithms with straightforward agent-to-agent differentiable message passing were slow to learn and did not match the top performing algorithms within the training time. An important point to consider here is that the models which have previously achieved SOTA on Cleanup have allowed for 300 million training steps, with evidence of learning only appearing at about 100 million steps. This means that is very difficult to judge the performance of NeurComm and DIAL based solely of this set of results; it may be the case that with additional training time they would produce better results, especially since both algorithms have several more parameters to optimise than ConseNet and FPrint. Surprisingly, the FPrint variation of the NA2C algorithm attained the highest score in under 20 million training steps; it is likely that allowing agents to share neighbourhood policy information evidently allows for and supports coordination between agents. Additionally, note that the ConseNet algorithm, for which there are provable convergence properties in fully cooperative tasks, shows a steady increase in performance over time and thus clear evidence of stable learning. A final comment is that, considering all agents received the same reward signal during training, the nature of the task defined in Cleanup may have allowed for a divide-and-conquer coordination strategy where a few agents focused on cleaning the river, while others picked the apples.

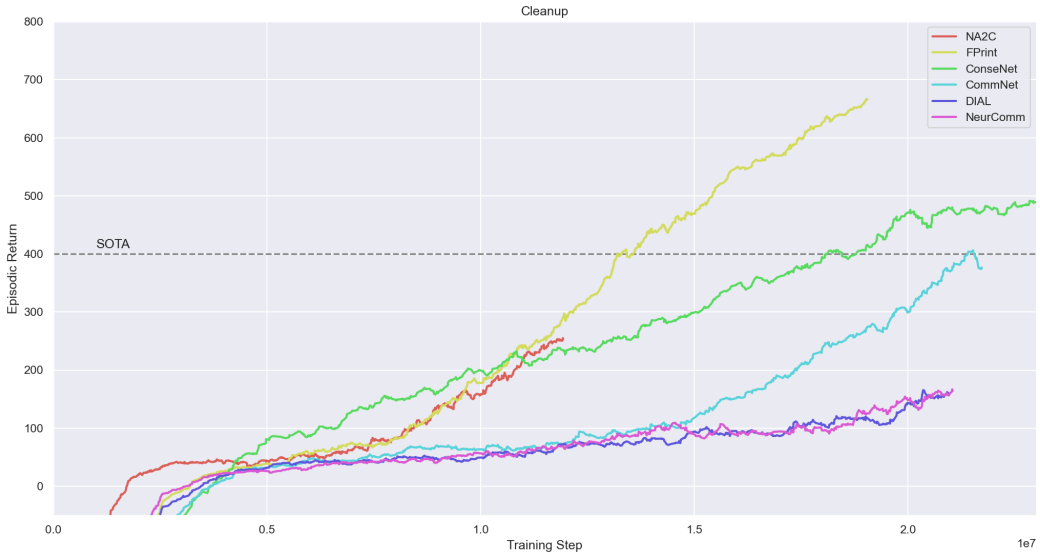


Figure 18

~

Harvest. Although some clear learning took place, the results achieved by the algorithms in Harvest were not as positive as on Cleanup. When contrasting the training performance on Harvest to that of Cleanup, two clear differences observed are (1) The initial task of figuring that harvesting apples yields positive rewards appears to be learn, which may be deduced from the sharp initial rise in return for all six algorithms, however, (2) learning cooperation is harder, and this may be the reason that the average return during training for each of the algorithms appears to plateaus at about half a million training steps. Furthermore, also at about half a million steps, the meta-networks corresponding to both CommNet and Neurcomm appear to experience 'catastrophic forgetting', as discussed above, which appears to cause a sudden, drastic change in behaviour leading to a sharp drop in return (with eventual, but not complete recovery). When contrast against the task in Cleanup, the objectives and the required strategies for success are subtly different. In Cleanup it is not possible to deplete the resource - the apples - entirely, as cleaning the river will always cause more apples to appear, so there is always a way to obtain more reward over time, even if it is not entirely straightforward. However, in Harvest the resource is finite, and a successful long term strategy therefore involves restraint on the part of some, or all of the agents, in order to give the apples a chance to regenerate. It may be that coordinated restraint is much harder to accomplish.

13 Conclusion

In conclusion, this report has covered the following content. In Part A the fundamentals of single-agent reinforcement learning were established, and four key single-agent deep-learning algorithms were explored in detail. In Part B, the game theoretic framework for the multi-agent learning problem was established, multi-agent reinforcement learning was then defined explicitly in terms of multiplayer Markov games, considering both networked and partially observable variations, after which the principle challenges in designing multi-agent

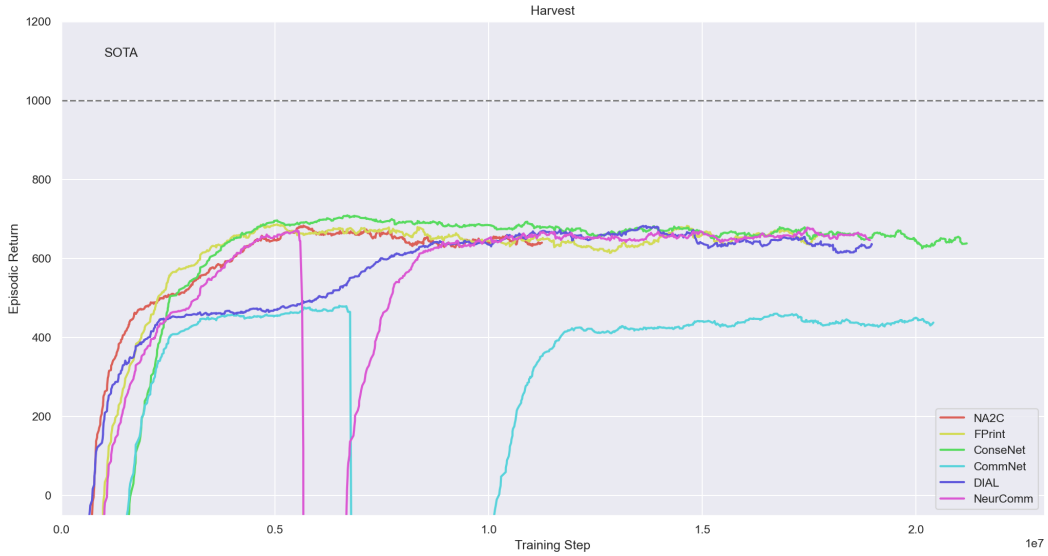


Figure 19

reinforcement learning - non-stationarity, goal setting and scalability - were outlined. Part B went on to cover six diverse, state-of-the-art deep multi-agent reinforcement learning algorithms, each with a different approach to solving multi-agent tasks. Finally, in Part C, the results of two simple experiments were examined, where the six algorithms were applied to finding equilibrium policies on the sequential social dilemma tasks, Cleanup and Harvest.

References

[1] Richard Sutton and Andrew Barto. Reinforcement Learning: An Introduction. MIT Press, 1998.

[2] Mnih et. al. Playing Atari with Deep Reinforcement Learning. arXiv preprint, arXiv:1312.5602, 2013.

[3] Sutton et. al. Policy Gradient Methods for Reinforcement Learning with Function Approximation. Proceedings of the 12th International Conference on Neural Information Processing Systems, 1999.

[4] Schulman et. al. Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347., 2017.

[5] Nowe et. al. Game Theory and Multi-agent Reinforcement Learning. Springer-Verlag Berlin, 2012.

[6] Busoniu et al. A Comprehensive Survey of Multiagent Reinforcement Learning. IEEE Transactions On Systems, Man, and Cybernetics–Part C: Applications and Reviews, Vol. 38, No. 2, March 2008.

[7] Bowling, M and Veloso, M. An Analysis of Stochastic Game Theory for Multiagent Reinforcement Learning.

[8] Foerster et. al. Learning to Communicate with Deep Multi-Agent Reinforcement Learning. 30th Conference on Neural Information Processing Systems (NIPS), 2017.

[9] Hausknecht, M. and Stone, P. Deep Recurrent Q-Learning for Partially Observable MDPs. AAAI Fall Symposium Series, 2015.

[10] Zhang et. al. Fully Decentralized Multi-Agent Reinforcement Learning with Networked Agents. Proceedings of the 35th International Conference on Machine Learning, PMLR 80:5872-5881, 2018.

[11] Chu et. al. Multi-Agent Reinforcement Learning for Networked System Control. 8th International Conference on Learning Representations (ICLR), 2020.

[12] Foerster et al. Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning. Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017.

[13] Lowe et al. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. Advances in Neural Information Processing Systems, 2017.

[14] Yang et al. Multiagent Reinforcement Learning for Multi-Robot Systems: A Survey tech. rep; 2004.

[15] Zhang et al. Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms. arXiv preprint arXiv:1911.10635 (2019).

[16] Tan, Ming. Multi-agent reinforcement learning: Independent vs. cooperative agents. In Proceedings of the tenth international conference on machine learning, pp. 330– 337, 1993.

[17] Matignon et. al. Independent reinforcement learners in cooperative Markov games: a survey regarding coordination problems. The Knowledge Engineering Review, 27(01): 1–31, 2012.

[18] Tesauro, Gerald. Extending q-learning to general adaptive multi-agent systems. NIPS, volume 4, 2003.

[19] Ciosek, K. and Whiteson, S. Offer: Off-environment reinforcement learning. 2017.

[20] Robert, CP and Casella, G. Monte carlo statistical methods springer. New York, 2004.

[21] Sukhbaatar et. al. Learning Multiagent Communication with Backpropagation. 29th Conference on Neural Information Processing Systems (NIPS), 2016.

Appendix A: Code

The single-agent code can be found at: https://github.com/JeremyDouglas91/reinforcement-learning-algorithms
The multi-agent code can be found at: https://github.com/JeremyDouglas91/deeprl_network