Faculty of Natural and Mathematical Sciences
Department of Informatics

Bush House, King's College
London, Strand Campus, 30
Aldwych, London WC2B 4BG
Telephone 020 7848 2145
Fax 020 7848 2851

7CCSMPRJ

Individual Project Submission 2019/20

Name: Stefan Roesch

Student Number: 19045488

Degree Programme: Artificial Intelligence

Project Title:[MRN5] Reinforcement learning for autonomous resource allocation in D2D communication

Supervisor: Reza Nakhai

Word count:13261

RELEASE OF PROJECT

Following the submission of your project, the Department would like to make it publicly available via the library electronic resources. You will retain copyright of the project.

☒ I **agree** to the release of my project

☐ I **do not** agree to the release of my project

Signature: Stefan Roesch

Date: 21/08/2020

Experiments in Channel Selection and Power Control for D2D Communication using Deep Reinforcement Learning

Stefan Roesch
Kings College London

August 2020

Contents

Abstract

In this thesis I present my research, experimentation and evaluation of Deep Reinforcement Learning methods used to tackle the channel selection and power control problems in Device-to-Device communication (D2D). In this setting, D2D users can reuse radio resources allocated to Cellular Users (CUs) in a cellular network with the intention of exploiting these resources as efficiently as possible. This complex use of the spectrum can create harmful interference in the network and as such demands sophisticated control methods to manage the logistics of the system at all times in order to ensure an acceptable quality of operation. Deep Reinforcement Learning (DRL) is a natural choice for the development of this kind of system as it is proven to perform exceptionally on control tasks with extremely high complexity. This thesis outlines the development and exploration of a number of DRL algorithms implemented in this setting to tackle the channel selection and power control problems in Device-to-Device communication (D2D).

1 Introduction

Artificial Intelligence (AI) has enjoyed a resurgence in recent years, in part due to the increasing maturity and successful implementation of Deep Neural Networks (DNNs). DNNs posses the ability to learn complex non-linear function approximations, a component which is essential for tackling the majority of interesting and complex computation-based problems. The successful implementation of DNNs have allowed great advances in solving a vast number of problems over a broad range of domains including: stock market prediction [20], high resolution image generation [7] and beating contemporary world champions at millennia old strategy games [31]. DNNs can also be used as a supplement in traditional learning algorithms, allowing for much lower time and/or spacial complexity that these algorithms may canonically be restricted by. Reinforcement Learning (RL) is one such field of study in which DNNs have been integrated to massively increase performance, the union of which is termed Deep Reinforcement Learning (DRL). Reinforcement Learning is concerned with the development of learning algorithms that have the ability to improve their performance over time through experience, guided by a scalar reward signal. These algorithms are generally instantiated as agents exploring an environment, through which the overall learning task is defined. As time goes on, the agent interacts with it's environment and receives feedback on it's actions (the reward signal) and based on this feedback improves it's ability to carry out the task in question and, in turn, learn to take better actions in future. Naturally, DRL has been considered for use in large-scale, highly complex, communications tasks that demand ever more sophisticated automation given the increasing global reliance on cellular network technologies. This thesis focuses on a subproblem in cellular communication, namely the problem of efficient channel selection and power control for Device-to-Device communications (D2D).

D2D communication is a technology that gives mobile devices the ability to forego routing information through a central Base Station (BS) instead sending that information directly to a neighbouring device[12]. This alleviates computational pressure on the BS and has the potential to improve data rates and overall network spectral efficiency while lowering network latency for those users who are in close proximity. This thesis concentrates on the inband-underlay mode of communication. In this mode cellular users (CUs) and D2D users (D2DUs) use the same radio spectrum licensed to the cellular operator and this spectrum remains undivided, meaning that D2DUs must re-use resources (or resource blocks (RBs)) that are allocated to CUs, allowing for more efficient use of this spectrum [12]. The problem with reusing these resources is that this causes harmful interference when too many D2DUs reuse the same RB, which in turn negatively effects Quality of Service (QoS) on the network. On top of ensuring that RBs are used efficiently, QoS depends heavily on appropriate power level control and with this comes the benefits of increased energy efficiency and battery life for both D2DUs and CUs [9][10]. This creates a contradicting objective that prioritizes both the mitigation of interference, and the maintenance of efficient energy use to preserve device battery life [28]. The main goal of this thesis however is to explore methods through which we can assure a minimum level of QoS throughout the network, not to achieve optimal energy efficiency.

DRL is an enticing candidate in the search for a solution to the D2D problem as it's ability to explore and improve upon it's performance through trial and error complements a setting where the goal is to find optimal control protocols based on the dynamics of the system. This is further solidified when considering the multi-user nature of the problem. The optimal RB selection and power level selection, at any time, for one D2D device is explicitly reliant on the state of all other devices in the network at that time. As such it naturally follows that in order to fully exploit the nature of the problem setting, one must consider the effect that other users have on the system. Multi-agent reinforcement learning (MARL) is a subfield of RL that is inevitably gaining attention in a world where single agent learning is growing mature. MARL considers very similar ideas to that of traditional RL but with the added complexity of explicitly considering other agent's effects on the system in order to inform decision making.

1.1 Aims

The main aim of this thesis is to explore and compare the performance of a number of different DRL algorithms, within a simulated cellular network, for the problem of Channel Selection and Power level control in D2D communication. In this thesis are described: five different algorithms, all based on the popular Actor-Critic method. Among these are comparisons of different variations on each algorithm related to the structure of the algorithms and the level of information processing performed to encourage agents to

learn in the multi-agent setting.

2 Background

2.1 Network Simulation

As stated above, D2D communication is a technology that allows mobile devices to establish a direct line of transmission between each other without the participation of a BS[6]. This is useful as it has the potential to improve the reliability of the link between devices, enhance spectral efficiency and system capacity, all while reducing latency within the network [6]. D2D is expected to play a vital part in 5G networks which are expected to fulfill the rising demands of modern cellular infrastructure[6].

For this thesis, a D2D-enabled cellular network is simulated under a dynamic network environment with the use of Finite State Markov Channels (FSMCs)[11] to capture the fading nature of wireless channels [32]. The simulation code was supplied by King's College London with minor adjustments made by myself. The following section is a description of the simulation dynamics and optimization problem and is an adaption of the 'System Model' and 'Network Optimization Problem' sections, respectively, as described in [32].

2.1.1 System Model

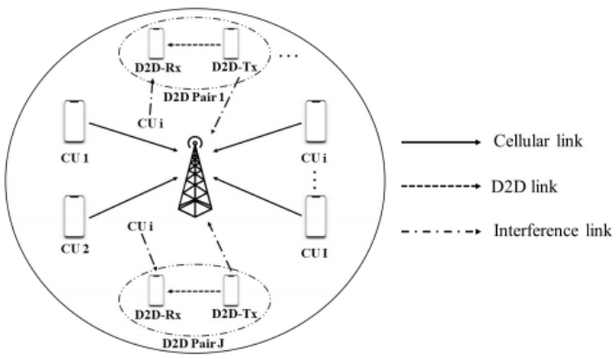


Figure 1: Structure of D2D Cellular Network [32]

Figure 1 illustrates the structure of the simulation. In this simulation there is a single BS located in the centre of the network. There are also I CUs and J D2D Pairs (D2DUs) randomly placed within the coverage area of the BS where $i \in \mathcal{I} = (1, 2, ..., I)$ and $j \in \mathcal{J} = (1, 2, ..., J)$ denote the sets of CUs and D2D pairs respectively. Time is also divided into separate, discrete time-slots t where $t \in \mathcal{T} = (1, 2, ..., T)$ denotes the set of T time-slots. Each CU is allotted a single up-link channel (or RB) at each time-slot t and each D2D pair, consisting of a transmitter (D2D-Tx) and receiver (D2D-Rx), can only reuse a single RB at any time. The channel gain g_t^{iB} between CU i and the BS is defined as

$$g_t^{i,B} = |h_t^{iB}|^2 \beta^{iB}, \ \forall t \in T \tag{1}$$

where h_t^{iB} is the short-scale channel coefficient of the channel between CU i and the BS at time t, and $\beta^{iB} \geq 0$ represents the large-scale fading components including path-loss and log normal shadowing. The channel gains of the interference links from D2D-Tx j to the BS and CU i to D2D-Tx j are similarly defined as g_t^{jB} and g_t^{ij} respectively, along with the channel gain of D2D pair j as g_t^j. Since channel gains g_t^{iB} and g_t^{jB} are known at the BS, this information can easily be transmitted to all D2D users. It is hard however for D2DUs to obtain the channel gain from the jth D2D-Tx to the ith CU (g_t^{ij}). This underlines the fact that the system is in the 'fully decentralised' setting - an extra layer of complexity that is further discussed in later sections.

As a symptom of the fully decentralised setting, D2DUs must make decisions without knowledge of the decisions of other D2DUs. As such, there is the risk that multiple D2DUs will decide to access the same RB at the same t. This act is called a 'collision' and is a major cause of interference in the network. To simplify the simulation, a perfect collision model is assumed through which none of the conflicting users posses the ability to carry out transmission. This model also assumes a lack of inter-D2D interference. Accordingly the signal to interference plus noise ratio (SINR) of the ith CU is defined as

$$\zeta_t^i = \begin{cases} \frac{p^i g_t^{iB}}{\sigma_N^2} & \text{if collision occurs or } i\text{th RB not selected} \\ \frac{p^i g_t^{iB}}{p_t^{jl} g_t^{jB} + \sigma_t^2} & \text{otherwise,} \end{cases} \tag{2}$$

where p^i is the transmit power of the ith CU (assumed to be constant), p_t^{jl} is the lth transmit power level of the jth D2D-Tx at time t, and σ_N^2 is the additive white Gaussian noise power. Finally, the SINR of the jth D2D pair at time t is defined as

$$\zeta_t^j = \begin{cases} 0 & \text{if collision occurs} \\ \frac{p^{jl} g_t^j}{p^i g_t^{ij} + \sigma_t^2} & \text{otherwise.} \end{cases} \tag{3}$$

Note again that, due to the perfect collision model, under the conditions of a collision occurring with D2D pair j as a participant, the SINR for that D2D pair is reduced to 0.

2.1.2 Optimization Problem

The overall goal for learning is to maximize the network throughput while ensuring certain QoS guarantees for D2DUs and CUs. This is done through each D2D-Tx selecting the most beneficial RB i and transmit power $p_t^{j,l}$ at each time step t. The optimization problem can therefore be formulated as

$$\max_{i \in \mathcal{I}, p_t^{j,l}} \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} W[log_2(1 + \zeta_t^i) + \mathbf{I}(i,j) log_2(1 + \zeta_t^j)] \tag{4}$$

$$s.t. \ \ \zeta_t^i \geq \zeta_i^*, \quad \forall i \in \mathcal{I}, \ \forall t \in \mathcal{T} \tag{5}$$

$$\zeta_t^j \geq \zeta_j^*, \quad \forall i \in \mathcal{I}, \ \forall t \in \mathcal{T} \tag{6}$$

$$0 \leq p_t^{i,l} \leq p_{max}, \forall j \in \mathcal{J}, , \ \forall t \in \mathcal{T} \tag{7}$$

$$\sum_j \mathbf{I}(i,j) \leq 1, \ \mathbf{I}(i,j) \in (0,1), \ \forall i \in \mathcal{I}, \ \forall t \in \mathcal{T} \tag{8}$$

$$\sum_i \mathbf{I}(i,j) \leq 1, \ \mathbf{I}(i,j) \in (0,1), \ \forall j \in \mathcal{J}, \ \forall t \in \mathcal{T} \tag{9}$$

where W is the channel bandwidth, ζ_i^* and ζ_j^* are the minimum SINR requirements for CU i and D2DU j respectively and p_{max} is the maximum power level. $\mathbf{I}(i,j)$ is an indicator function

$$\mathbf{I}(i,j) = \begin{cases} 1 & \text{if D2DU } j \text{ reuses the } i\text{th RB} \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

Constraints (5) and (6) ensure that QoS minimums are kept, constraint (7) ensures that no power level can be outside the specified bounds and (8) and (9) ensure that no more than one D2D-Tx can reuse any single RB at any time and that no D2D-Tx can reuse more than a single RB at any time respectively.

2.2 Deep Neural Networks

Artificial Neural Networks (ANNs) are a form of computational structure that loosely mimic the biological structure of the brain. Inputs are fed into the network at one end to be received by a collection of neurons, or units, which subsequently process the input, based on a set of coefficient weights. This process synthesises an output at the other end of the network. The value of this output is then compared to some target value (which can be defined by the user, inferred from the input data itself or by some other auxiliary information) in order to determine how best to adjust the value of these weights to modify the network's processing w.r.t some optimization objective.

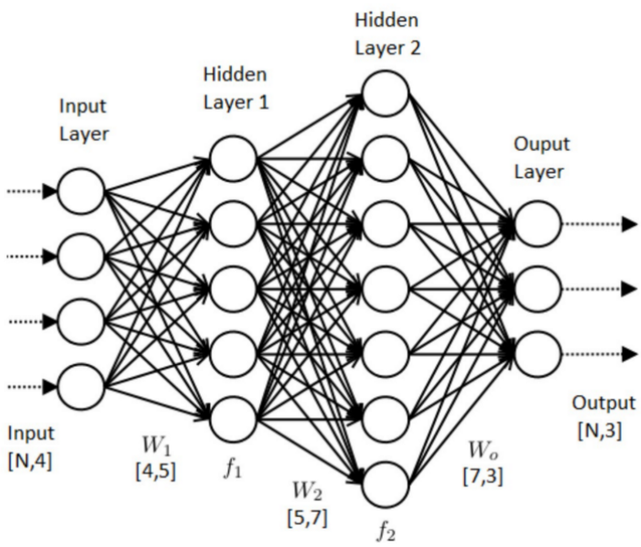


Figure 2: Topology of a simple, 3-layer, feed-forward, deep neural network with the dimensionality of the input, output and intermittent weight matrices denoted [1].

Figure 2 shows an illustration of a 'Deep' Neural Network. Deep Neural Networks (DNNs) are ANNs that contain more than three 'hidden layers' of neurons ('hidden' as they are not seen directly from the input nor the output of the network) where a layer is a collection of neurons that operate at the same depth in the network.

The power of DNNs come from their ability to competently approximate highly non-linear functions. This is due to their layered structure, with additional layers providing higher levels of non-linearity.

2.2.1 Computing the Output

As mentioned above, a network's input is processed w.r.t a set of coefficient weights. For the three layer network in **Figure 2**, this is computed as follows (ignoring bias terms)

$$\mathbf{z} = h(\mathbf{W}_o f_2(\mathbf{W}_2 f_1(\mathbf{W}_1 \mathbf{x}))) \tag{11}$$

where $\mathbf{z}$ is the output vector, $\mathbf{x}$ is the input vector, $\mathbf{W}_1, \mathbf{W}_2$ and $\mathbf{W}_o$ are the first, second and output layer's weight matrices respectively, f_1, f_2 and h are first second and output layer's activation functions respectively. These activation functions can be any differentiable function, the selection of which depends on the application.

2.2.2 Updating the Weights

The value of the coefficient weights constitute what the network has 'learnt' during training. The goal of network training is to modify these weights so as to mould the output of the network to suit our needs. In order to do this, we first need to establish the quality of the current output and measure that with how we would like the network to perform. This is done by determining the error at each weight w.r.t some objective and adjusting that weight's value in a way that lowers this error. In order to do this, we start by defining an error term. In this case, we use the Least Mean Squares (LMS) method

$$J(w) = \frac{1}{2}||\mathbf{t} - \mathbf{z}||^2 \tag{12}$$

where $||.||$ is the Euclidean Norm and t, in this context, is a pre-determined target for the output. The specific weight we want to update is then updated according to the gradient descent algorithm (for RL we use gradient ascent as the goal is to **maximise** a reward instead of **minimise** an error, this involves simply substituting the '-' for a '+')

$$w_{new} \longleftarrow w_{old} - \alpha \frac{\delta J(w)}{\delta w} \tag{13}$$

where α is a user defined scalar called the learning rate. This calculation is called the gradient descent method and is performed for every weight in the network over a number of iterations until the error is minimized to an acceptable value.

2.3 Reinforcement Learning

Reinforcement Learning (RL) can be considered the third sub-field of Machine Learning (ML), alongside Supervised and Unsupervised Learning [33]. The main difference between RL and the other ML paradigms is in it's trial and error based learning of an environment guided by a scalar reward signal [33]. Where Supervised and Unsupervised learning methods work to uncover generalizations or structure in large, compiled data-sets, RL techniques focus on exploratory interaction within a pre-defined, but not necessarily well understood, environment. RL lies somewhere in the middle between it's two cousin fields of ML. Much like supervised learning, an external 'supervisor' assigns 'labels' to certain states of

the environment (the reward). Conversely, the learner is neglected this instruction at most states and, as such, is left to learn how best to maximise this reward through trial and error.

2.3.1 Markov Decision Processes and the Discounted Return

Markov Decision Processes (MDPs) are a mathematically idealized form of the RL problem for which precise theoretical statements can be made [33]. An MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R)$ where $\mathcal{S}$ denotes the state space of the system, $\mathcal{A}$ the action space, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the transition probability from state $s \in \mathcal{S}$ to state $s' \in \mathcal{S}$ given action $a \in \mathcal{A}$ was taken from s (the transition model) and $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the reward function that provides the reward for taking action a in state s and transitioning to state s'.

The system is also generally broken up into $\mathcal{T} = (1, ..., T)$ discrete time-steps with the reward r_{t+1} and next state s_{t+1} being observed based on the current state s_t and action to be taken a_t.

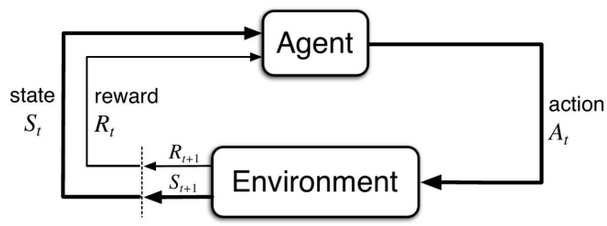


Figure 3: Illustration of the decision making process between the agent and the environment [33].

Figure 3 illustrates this decision making cycle in a generalised setting. First the agent observes the current state of the environment s_t and, based on this observation, decides on an action a_t. A reward r_{t+1} and subsequent state s_{t+1} are then generated based on the effect of the action a_t on the environment and time moves forward a single step $t = t + 1$, with the process then being repeated.

The goal of the agent, informally, is to maximise the the total amount of reward is receives [33]. This idea is clearly stated in the 'reward hypothesis' [33]: "That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (the reward)". More formally, we define this expected value as the 'expected return':

$$G_t := R_{t+1} + R_{t+2} + ... + R_T \tag{14}$$

where T is the final time step (or 'time horizon'). This definition poses a problem however for tasks, like that of D2D power level and RB selection, that do not have a well defined final time step i.e. where $T = \infty$. These kinds of tasks are call 'continuous' and require an extra parameter to compensate for a potentially infinite sequence of rewards. This element is called the 'discount rate' denoted as γ and is injected into (7) to create the 'discounted return'[33]:

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{15}$$

where $0 \leq \gamma \leq 1$ is pre-defined by the user. The closer γ is to 0 the more 'myopic' the return becomes i.e. the return places higher value on immediate rewards instead of future ones.

2.3.2 Policies and Value Functions

An agent's policy π is the way it makes decisions. The policy maps states to a probability distribution over all possible actions in that state. This means that if the agent has learnt the optimal policy in it's environment, it should always take the most rational action in each subsequent state (where the most rational action is usually defined as the one that maximises $\mathbb{E}[G_t]$). This idea of a policy is useful when the agent wants to make decisions based on which action to take in a particular state. In order to make those decisions however, we need some method with which to determine the quality of subsequent states (with the quality being based on which states lead to the highest likelihood of receiving large rewards in future). In order to determine the quality of a state, we maintain estimates of value functions. Value functions are used to give the agent insight into which states to transition into and hence, which actions are the most profitable w.r.t future rewards.

There are two different kinds of value functions which are used in the literature: The state-value function $V(s)$ describes the quality of a particular state based on the rewards that can be obtained from being in that state and following some policy π from then onwards:

$$V_\pi(s) \coloneqq \mathbb{E}_\pi[G_t | \mathcal{S}_t = s] \tag{16}$$

which denotes the value of state s given that we follow policy π from time-step t and into the future [33].

Similarly we can define the action-value function of taking an action a in state s under policy π as follows:

$$Q_\pi(s, a) \coloneqq \mathbb{E}_\pi[G_t | \mathcal{S}_t = s, \mathcal{A}_t = a] \tag{17}$$

which denotes the value of taking action a in state s given that we follow policy π from time-step t and into the future [33]. Both of these measures estimate the expected return for their respective input given the agent continues to act in accordance with the current policy in the future.

In some cases it can be sufficient to simply maintain estimates of $Q(s, a)$ for every action in every state using a lookup table and implement a simple policy to take the actions that hold the highest value with some probability. e.g. We can implement an ϵ -greedy policy where actions will be chosen as

$$a_t = \begin{cases} \text{random a} \in \mathcal{A} & \text{with probability } (1 - \epsilon) \\ max_{a'}\, Q(s, a) & \text{with probability } \epsilon \end{cases} \tag{18}$$

where $0 \geq \epsilon \geq 1$. It is important to note that the agent still retains the ability to explore the environment (the probability of taking the action with

the highest value is not 100%) as this is essential to avoiding sub optimal policies and uncovering optimal ones.

Using a lookup table to store value estimates states is called the tabular case. While the tabular case may be sufficient for simpler settings, as problems become more complex, the state and action spaces tend to grow exponentially and hence hardware and time limitations quickly make this method infeasible.

2.3.3 Deep Reinforcement Learning

In order to overcome the limitations posed by tabular methods, function approximation is used. Instead of maintaining a lookup table of values for every possible state and action, we can train a function approximator to accurately estimate the value of a state/action as the agent reaches it. The most popular method thus far has been to use ANNs as these approximators. A famous early example of which being TD-Gammon [35], a backgammon playing algorithm that used an ANN based RL algorithm to reach near master levels of play. With the maturity of DNNs came their adoption into the field of RL, giving birth to the field of Deep Reinforcement Learning (DRL). Contemporary breakthroughs [30] [19] have proved the validity and power of using DNNs as value function approximators and they have been largely adopted by RL practitioners around the world.

As stated above, DRL involves the use of DNNs as value function approximators. This generally involves passing state information as input to the network, with the output either being a scalar estimate of $V(s)$ or a distribution of outputs that give $Q(s, a)$ for each possible a in s. Deep Q-Learning [19] is one such method that uses a DNN to approximate $Q(s, a)$.

2.3.4 Monte-Carlo, Temporal Difference Methods and the baseline term

Actor-Critic is an example of what is known as a 'Policy Gradient' method and serves as the underlying theoretical background for the algorithms developed in this thesis. Policy gradient methods work by directly approximating a policy π, and using that policy to generate a probability distribution over all actions for a state from which the agent can decide which action to take. Approximating the policy directly is useful as not only does the policy approximation provide a 'built-in' system for state exploration (sampling from the outputted distribution), but also gives better convergence properties over action-value methods like standard Q-Learning [33] [32]. The general policy objective is defined as

$$J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} r_{t+1}] \tag{19}$$

where θ denotes the policy approximation parameters (weights in a DNN). Updates are made with gradient ascent according to the following policy gradient

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta \; log \; \pi_\theta(a_t|s_t) G_t \tag{20}$$

where ∇_θ is the gradient w.r.t parameters θ. The full derivation can be found in [39].

(20) shows an example of what is called a 'Monte-Carlo' method. In this setting, training is broken up into separate lengths of time called episodes. At the end of each episode, rewards are traced back from the end of the episode and updates are made retroactively (this is known as offline learning). This means that the agent undergoes no updates until the end of an episode. A downside of this is that an agent's behaviour in that episode can be highly divergent from it's behaviour in the next episode. This is characterised by a high variability of the log probabilities (log of the policy distribution i.e. $log \; \pi_\theta(a_t|s_t)$) [40] The benefit, however, to Monte-Carlo is the fact that there is no need for complete knowledge of the environment [33] (it is model-free). This means that updates are made solely on experience - sample sequences of states, actions and rewards from actual or simulated interactions with an environment [33]. This is invaluable in complex scenarios where absolute knowledge of the environment is impossible. Monte-Carlo methods are opposed to Dynamic Programming methods. Dynamic Programming methods use information about the environment dynamics to iteratively update stored values, like in the example presented in 2.3.2. They do however demand absolute knowledge of the transition probabilities in order to effectively estimate the values of states. While Monte-Carlo methods provide a foundation for model-free learning, they still impose the limitation of non-continuous, offline, learning. Temporal Difference methods bridge the gap between Monte-Carlo and Dynamic Programming methods and are a central idea to RL. Like Monte-Carlo, TD methods can learn in a model free setting - from experience - and like dynamic programming, TD methods can learn iteratively (known as online learning) from value estimates allowing for training in a continuous setting.

The difference between Monte-Carlo methods and TD methods can be illustrated in the difference between the following updates [33]

$$V(S_t) \longleftarrow V(S_t) + \alpha[G_t - V(S_t)], \tag{21}$$

$$V(S_t) \longleftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{22}$$

In (21) we observe that the target for the update is defined as G_t, the actual return following time t. This update requires the algorithm to wait until the end of an episode for the value of G_t to become known and subsequently make, offline, updates to the parameters (Monte-Carlo). In (22) the target is defined as $R_{t+1} + \gamma V(S_{t+1})$ which is dependant only on the reward and state-value estimate at $t + 1$ (TD). This means that targets, and subsequently updates, can be made as soon as the next time step and hence an episodic structure is not necessary [33] allowing for online learning.

While the high variance problem is more pronounced in Monte-Carlo methods, TD methods are still effected by this. To combat the high variance problem, we can introduce a stabilizing term called the baseline $b(s_t)$. If we can determine a value for the baseline that significantly reduces this variance then we can subtract it from the policy gradient and reduce

the variance of the log probabilities significantly, stabilising training [15]. The form of this subtraction is as follows

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} \nabla_\theta \, log \, \pi_\theta(a_t|s_t) G_t - b(s_t)] \tag{23}$$

2.3.5 Actor-Critic

Actor-critic methods are comprised of a pair of ANNs. The first of which is used to approximate the policy (the actor) and the second is used to approximate a value function (the critic). What distinguishes actor-critics is the use of this value function approximation to provide value estimates for a state from the estimated value of subsequent states, guiding the learning of the actor [33]. This use of the critic is what allows actor critic to enjoy TD like iterative updates, allowing for it's application in continuous tasks, and the variance reduction introduced by the baseline term.

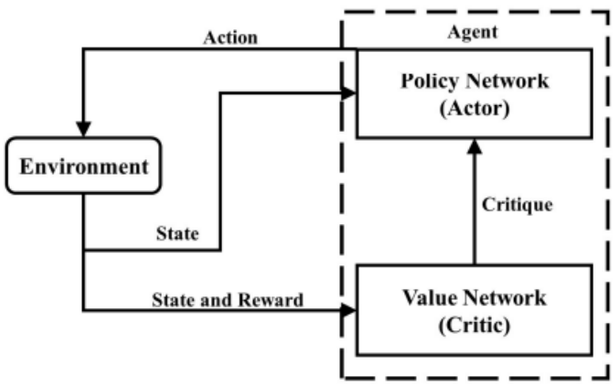


Figure 4: The general actor critic method [32].

Figure 4 shows the structure of an actor-critic based agent and the general flow of information. The policy network takes in the state and outputs a probability distribution over the actions available in that state - from which an action is sampled. The value network takes the state and reward and provides critique on the policy which is used to shape the policy's learning.

One of the most common kinds of actor-critic baselines is that of TD actor-critic, so called because of it's use of the critic to provide a TD error term in the policy gradient to guide the actor's learning

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[\nabla_\theta \, log \, \pi_\theta(s,a)\delta] \tag{24}$$

where

$$\delta = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{25}$$

is the TD error. The reader will notice the similarity between (25) and (22), highlighting the iterative, temporal difference nature of this gradient.

2.4 Multi-Agent Reinforcement Learning

Multi-Agent Reinforcement Learning (MARL) is a sub-field of RL that concerns itself with the behaviour and development of RL algorithms that are optimized for settings in which multiple agents are present and must learn to interact to achieve global and/or individual goals in which these other agents have some tangible effect on the environment. One of the core considerations to be made in a multi-agent setting is how to effectively account for the presence of other external agents in one's own learning. This is embodied in the notion that each learning agent possesses it's own long term goals, and as such, it can be assumed that their effect on the environment will be a function of the policies of all other agents [41]. This idea is what distinguishes MARL from single-agent RL - the explicit consideration of multiple agents interacting and having an effect on each other's learning.

2.4.1 Markov Games, their Settings and the Nash Equilibrium

A Markov Game [14], or Stochastic Game [29], is a continuation of the MDP framework with additions that accommodate for the presence and theory of multiple learners. A Markov Game is defined as a tuple $(\mathcal{N}, \mathcal{S}, (\mathcal{A}^j)_{j\in\mathcal{N}}, \mathcal{P}, (R^j)_{j\in\mathcal{N}})$, where $\mathcal{N}$ denotes the set of $N > 1$ agents, $\mathcal{S}$ denotes the state space, $\mathcal{A}^j$ denotes the action space of the jth agent, $\mathcal{P} : \mathcal{S} \times \mathcal{U} \times \mathcal{S} \longrightarrow R$ is the transition probability from state $s \in \mathcal{S}$ to state $s \in \mathcal{S}$ given **joint action** $u \in \mathcal{U}$ was taken from s (the transition model) and $R^j : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longrightarrow \mathbb{R}$ denotes the reward function that determines the immediate reward received by agent j for the transition from state $s \in \mathcal{S}$ to state $s \in \mathcal{S}$ given joint action $u \in \mathcal{U}$ was taken from s [41]. Note that the reward received by agent j is dependant on the **joint** action u taken at s not solely it's own action selection a^j, despite the fact that it's reward for that joint action may differ from those rewards received by other agents. In MARL, the goal of each agent is, much like single-agent RL, to find the optimal policy $\pi^j : \mathcal{S} \longrightarrow \mathcal{A}^j$ so as to maximise it's own long-term reward. The difference lies in the fact that, as a consequence of multi-agent consideration, each agent's value function $V^j : \mathcal{S} \longrightarrow \mathbb{R}$ becomes a function of the **joint** policy $\pi : \mathcal{S} \longrightarrow \mathcal{U}$ [41]. Like with single-agent RL, these functions are shaped by each agent's reward. Since other agents must be considered, rewards have to be designed appropriately to ensure an intended goal. As such, we can analytically segregate the way in which agent's receive rewards into a number of different settings as was outline by Zhang et al. in [41]. These settings are as follows

Cooperative Setting

A learning task is said to be in the cooperative setting if all agents share the same reward function $R^1 = R^2 = ... = R^N$. Intuitively it follows that, due to agents sharing a reward function, the value function for each agent will also be identical $V^1(s) = V^2(s) = ... = V^N(s)$ [41]. This fact enables the use of single agent learning algorithms given that agents are treated as a single, coordinated learner. The goal however has changed from the single-agent case of one agent trying to learn an optimal policy so as to maximize expected return, to multiple agents trying to establish an optimal Nash Equilibrium through a joint policy [41]. A Nash Equilibrium,

w.r.t a policy, is described as an equilibrium point π^*, in the joint policy, from which no agents have any incentive to deviate. In other words, $\pi^{j,*}$ is the best response to $\pi^{-j,*}$ where $-j$ denotes the policies of all other agents excluding j [41].

Competitive Setting

In the competitive setting, rewards are said to be zero sum: $\sum_{j \in \mathcal{N}} R^j = 0 \ \forall t \in \mathcal{T}$ [41]. Less formally, this means that rewards gained by one agent results in an equal reward loss between the other agents. This is the common setting for two-player games like Go, Chess and Backgammon [41]. While the literature doesn't explicitly mention MARL, there is a wealth of research that can be said to focus on this setting [31], [35]. The goal for this setting is also based in the establishment of a Nash Equilibrium, similar to the Cooperative Setting.

Mixed Setting

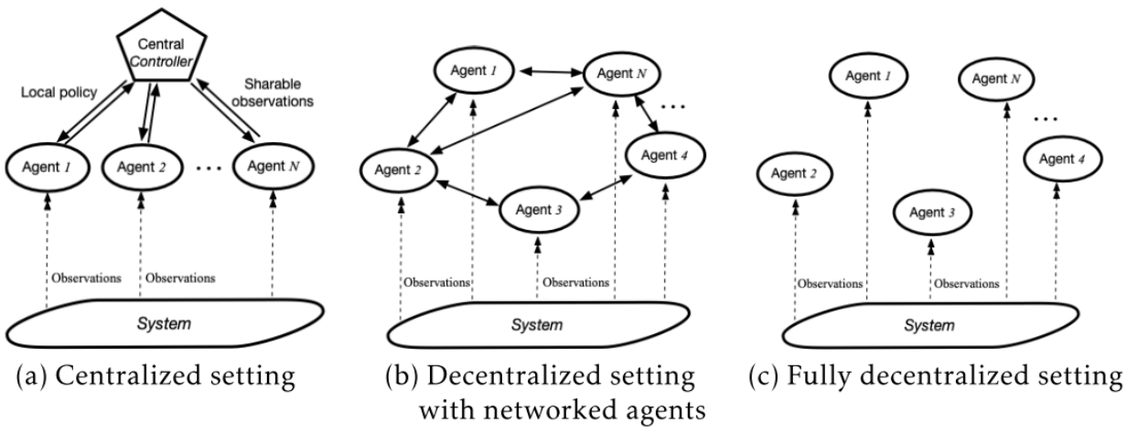
The Mixed setting is middle-ground between the cooperative and competitive settings. In this setting there is no restriction on the goals and relationships among agents and each agent is self interested and their goal may, or may not, conflict with those of other agents [41]. The most natural example of this setting would be team-games. In these games, agents are said to cooperate with their teammates but compete with the other team(s).

2.4.2 Information Structure

Along with how agent's share rewards, MARL demands another consideration - that of how information is shared among agents. Since many MARL applications are largely based in the real world, considerations have to made w.r.t the agents' ability to communicate with each other. While in the competitive setting it is beneficial to avoid direct communication, goals that require agents to act cooperatively can be easier to achieve through possessing some line of communication between agents [34]. While the degrees of communication are also worth consideration, communication protocols generally incur some cost, this section is exclusively focused on the means of communication.

Figure 5 illustrates three examples of information structures in MARL. The first, and leftmost, being the **Centralized Setting**. In this setting, a central controller maintains communication among agents. Functioning much like a BS, agents must send information about their observations, actions etc. to the central controller which distributes that information among the other agents. This central controller can also make decisions based on this information and perform critic-like duties, and can even design policies for the agents [41]. This is powerful as the central controller has access to the joint actions, joint observations etc of all agents and as such has a much more broad and informed view of the environment than any single agent. The downside however is that, while this is an information-rich setting, given that the environment is not fully observable by any one agent, the communication overhead can be prohibitively large, especially in tasks with many agents.

The **fully decentralized** setting is the polar opposite of the centralized setting. Here, agents have no communication protocols available to them whatsoever. This means that policies must be learnt solely on an agent's individual observations of the environment with no additional information on the internal state of other agents. This is difficult because MARL is inherently non-stationary w.r.t the evolution of states. As multiple agents are learning and interacting with the environment concurrently, an action taken by one agent is likely to affect both the reward of the other agents and the state at the next time step [41] making it difficult for policies to converge. Additionally, the limited information enforced by this setting will compound this problem and cause difficulties during learning.



(a) Centralized setting (b) Decentralized setting with networked agents (c) Fully decentralized setting

Figure 5: Illustration of three different information structures in MARL [41]. Solid arrows show lines of communication, dotted arrows show observations.

One can strike a middle ground between these two extremes by including some networking protocol between the agents themselves. This setting is called decentralized with networked agents [41]. Typically this is limited to local communication between proximate agents, to reduce communication overhead, and information is allowed to flow through the network over time [41].

While information sharing is proven to be beneficial to learning [34], in some tasks it is simply impossible or impractical to guarantee the necessary communication infrastructure, whether due to cost, scalability or robustness concerns. As such, there is great value in methods that allow for fully decentralized execution. Fortunately, there is a way to attain the training benefits of a centralized setting, while allowing for the low financial and computational cost of fully decentralized agents. The centralized-learning decentralized-execution learning scheme involves training decentralized policies in a centralized setting [41]. This means that, during training, a centralized controller can be used to aggregate information obtained from the local observations of all agents and use that information to assist in the training of **decentralized** policies. It is important to note that these policies are trained to function for completely decentralized execution, meaning that agent's decision making at execution time must be reliant solely on their own local sensory input and cannot rely on the presence of a centralised entity or any other means of communication. Due to this limitation, the practical use for centralized-learning decentralized-execution algorithms is limited to those tasks whose training can be performed in a simulated environment.

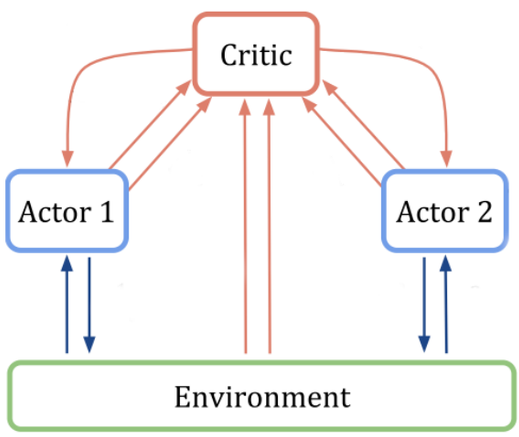


Figure 6: Example of centralized-learning decentralized-execution information sharing structure [4]. Arrows illustrate the direction of information flow. Elements in red are not necessary during execution and as such are discarded after training.

2.4.3 The Credit Assignment Problem and Counterfactual Baseline

The inclusion of a centralised component during training is valuable as it allows for centralized policies to be explicitly influenced by, potentially essential, global information present in the system where otherwise that information would have to be inferred. A reliance on inferring information from the system may drastically effect the rate of convergence or completely exclude certain optimal policies. This transparency allows for the use of methods that rely on the availability of this information to tackle problems like credit assignment which, without it, could be vastly more complex to address.

Multi-agent credit assignment, in RL, is a problem specific to MARL. In settings where a reward function is shared among agents, it can be difficult for agents to ascertain the value of their own individual actions. This is due to the value function being approximated with a reward that is received due to joint actions instead of taking individual actions into account. To illustrate why this is an issue, consider a cooperative system in which three separate agents A, B and C are undergoing learning. A and B have learnt efficient policies and are performing well, resulting in a high positive contribution to the global reward. C's policy however is sub-par and having a slightly negative effect on the global reward. Since agents share a reward function however, C has no feedback regarding it's poor performance as it's policy is shaped on the same high reward as A and B's. This causes C's policy to converge to a sub-optimal point, leading to sub-optimal performance of the system. The opposite of this scenario can also be considered where one agent is making good choices, but the poor choices of other agents leads to beneficial actions being perceived as harmful.

While designing bespoke reward functions for each agent can work in some situations, this can encourage agents to become self interested in their own goals and therefore negatively effect group performance. Counterfactual multi-agent (COMA) policy gradients [4] is a method that exploits the benefits of the centralized-learning decentralized-execution setting while also addressing the credit assignment problem using a shared reward function. COMA uses a counterfactual baseline that is based on the idea of difference rewards [38]. Difference rewards work on the principle that each agent learns by what is called a shaped reward $\mathcal{R}^j = R(s,u) - R(s,(a^{-j},d^j))$, d is a pre-selected default action. This method compares the global reward $R(s,u)$ with the reward received when the action of agent j is replaced with a default action d [4]. In this case, any action by agent j that improves their difference reward $\mathcal{R}$ will also improve the global reward [4]. While this is a powerful solution to the credit assignment problem, it requires multiple simulations to be run in order to obtain $R(s,(a^{-j},d^j))$ at each time step which can be computationally expensive, especially with high numbers of agents [4]. It also requires appropriate selection of the default action d which can be non-trivial, and potentially needing expert opinion [4].

COMA is an actor critic method that addresses the credit assignment problem by maintaining a centralized critic, $Q(s,u)$ that estimates the Q-values for joint action u in state s. Using this we can then compare the Q-value for an agent's current action to a counterfactual baseline that marginalises out that action while keeping the other agent's actions fixed, defining an advantage function [4]

$$A^j(s,u) = Q(s,u) - \sum_{a'^j} \pi^j(a'^j|o^j) Q(s,(u^{-j},a'^j)) \quad (26)$$

where, $A^j(s,u)$ is the advantage for agent j (individual advantages are computed for each agent for each update), a^j is the action taken by agent j, a'^j denotes all available actions for agent j other than a^j, o^j is the observation of the system state made by agent j (in fully observable systems this is replaced with the state s), $\pi^j(a'^j|o^j)$ is the probability of taking action a'^j conditioned on agent j's observation o^j given policy π^j, u is the joint action over all agents and u^{-j} are the joint actions of each agent excluding agent j [4]. This allows for a separate advantage to be computed for each agent whilst avoiding the extra burden of additional simulations. This critic computed advantage is then used in the policy gradient update of each agent (the actors) as follows

$$\nabla^j_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla^j_\theta \; log \; \pi^j_\theta(a^j_t|o^j_t) A^j(s,u) \tag{27}$$

where, ∇^j_θ is the gradient of the jth policy parameterized by θ. This reward function is equal for all agents, making the problem a cooperative one. This method is usable in the centralized-learning decentralized-execution setting as the actors, who maintain the policy networks, have no reliance on information outside of their own limited observation space in order to operate. Once training is completed, the critic can be removed and any information transfer that was necessary for learning will be removed along with it.

3 Algorithm Design

3.1 Problem Environment

As is mentioned previously, in this thesis the D2D channel selection and power level selection problem is formulated as a simulated cellular network. While section 2.1 describes the mechanics of this simulation, this section will describe how this simulation is formulated as a Markov Game, with which the learning algorithms can interact. Again, this section is adapted from the description given in [32] as the environment is formulated in much the same way, with a few alterations made.

Agents are represented by the D2D-Tx in a D2D pair. The system **state** s, at any t, is represented by the collective SINR values of all CUs at that time step t i.e. $s_t = [\zeta^1_t, \zeta^2_t, ..., \zeta^I_t]$. It is also important to note that since the BS transmits this information to each D2D pair, the state is fully observable - each agent has full transparency w.r.t this state.

At each time-step, each agent takes an **action** a^j_t which consists of two separate decisions: the RB selection i and the power level selection $p^{j,l}_t$, i.e. $a^j_t = [i, p^{j,l}_t]$. In order to simplify learning, power levels are made discrete by limiting the amount of possible selections to L different selections. This process is formulated as $p^{j,l}_t = \frac{p_{max}}{L} l$.

Finally, **rewards** are computed as

$$r^j_t = \begin{cases} -0.2 \quad \text{if (5), (6) or (8) are violated} \\ \sum_{i \in \mathcal{I}} log_2(1+\zeta^i_t) + \mathbf{I}(i,j) log_2(1+\zeta^j_t) \text{ otherwise.} \end{cases} \tag{28}$$

the punishment of -0.2 was selected over 0, as in [32], to entice rapid convergence, the intuition being that agents will want to get to a state of positive reward as quickly as possible as there is now a punishment associated with sub-optimal behaviour. This reward function falls in the Mixed Setting as the majority of the reward is identical for each agent ($\sum_{i \in \mathcal{I}} log_2(1+\zeta^i_t)$) with each agent receiving additional reward based on their own SINR ($\mathbf{I}(i,j) log_2(1+\zeta^j_t)$). This suggests the benefits for multi-agent consideration in the learning algorithm - since a majority of the magnitude of this reward is shared, the author hypothesises that using MARL techniques can be highly beneficial to learning.

Table 1. Simulation Parameters

Parameter	Value
Bandwidth(W)	10MHz
No of CUs (I)	30
No of D2DUs (J)	15
p_{max}	12dBm
Levels of transmit power (L)	10
CU's transmit power (p^i)	20dBm
SINR minimum for CUs (ζ^*_i)	6dB
SINR minimum for D2DUs (ζ^*_j)	6dB
AWGN (^2_N)	-174dBm

3.2 Individual Actor-Critic

The first algorithm is Individual Actor-Critic (IAC). IAC is so named as it is implemented using standard single-agent actor-critic. This means that there is no consideration, in it's design, of any multi-agent principles like inter-agent communication or credit assignment. Agents learn solely based on their own, individual, observations of the environment. Figure 4 illustrates the structure of this algorithm. Each agent is equipped with both it's own policy approximator $\pi^j_\theta(a^j|s)$ (actor) and a value function approximator $V^j_\omega(s)$ (critic), the state value function is used as we are learning with the TD error (25). This means that each agent maintains it's own separate approximation of the value function. Policy parameter updates are performed based on the policy gradients with the TD error using gradient ascent

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta J(\theta_t) \tag{29}$$

where, $\nabla_\theta J(\theta) = \mathbb{E}_\pi[\nabla_\theta \; log \; \pi_\theta(s,a^j)\delta^j_t]$, α_θ is the learning rate and δ is the TD error (25).

Similarly, the critic updates it's parameters using gradient ascent

$$\omega_{t+1} = \omega_t + \alpha_\omega \nabla_\omega V^j_\omega(s_t)\delta^j_t \tag{30}$$

Each agent's policy $\pi^j_\theta(a|s)$ is parameterized with a DNN. The input to this network is the state and the output is a probability distribution over that agent's action space, from which it's actions are sampled. In order to sample actions from the DNN, the output layer must contain the same number of units as there are action combinations ('combinations' as there are two different decisions to make at each time step, power level and RB selection). The output layer is then passed through a softmax activation function

in order to determine probabilities for each action - making it possible to sample actions from the output based on those probabilities. The critic is also parameterized by a DNN. It's function is to provide state value estimates so as to be used in the TD error calculation which assists the policy's training.

3.2.1 Multi-Headed Policy Network

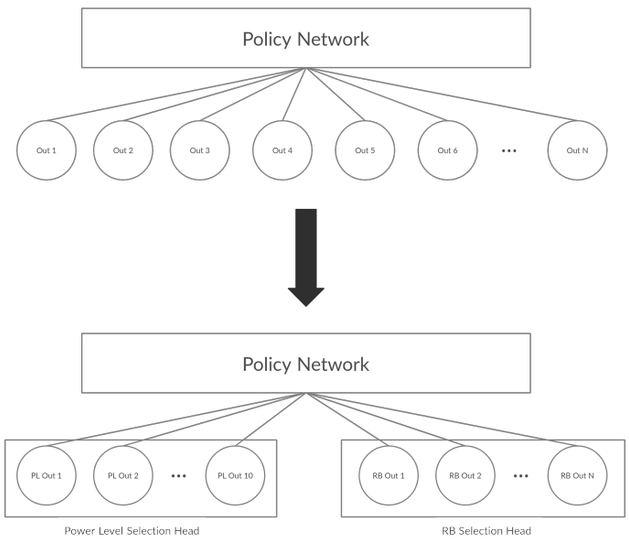


Figure 7: Representation of changing original policy network to multi-headed policy network.

The action space for each agent is made up two separate decisions $\mathcal{A}^j = ([1, p_t^{j,1}], ..., [i, p_t^{j,l}], ..., [I, p_t^{j,L}])$. This means that the number of output units at the policy must be $I \times L$ to ensure that each possible combination of these two decisions is represented. While DNNs are capable of this kind of high dimensional computation, this results in a high number of training parameters that must be updated at every time-step (assuming learning is online). e.g. if $I = 30$ and $L = 10$, this results in an output layer with 300 units. If the preceding hidden layer contained 200 units, the number of weights, at the final layer alone, would equal $200 \times 300 = 60,000$ (ignoring bias terms). This number can be drastically reduced by using a two-headed output layer (see Figure 7). With a two-headed output, we replace the original policy output layer with two separate output layers. One of these layers will provide a probability distribution for channel selection and the other for power level selection, hence the number of units in these will be I and L respectively. The number of parameters now becomes $200 * 30 + 200 * 10 = 8000$ (again ignoring bias terms), almost 10x fewer trainable parameters with identical performance.

3.3 Dual Critic

Dual critic is a novel algorithm developed by me in the course of the project. I call it 'Dual Critic' as it splits the critic from IAC into two separate critics, each estimating a value function based on different parts of the reward function. As stated above, the reward is made up of two constituent parts, the sum of the CUs SINRs $\sum_{i \in \mathcal{I}} log_2(1 + \zeta_t^i)$ and the SINR of the jth D2DU $\mathbf{I}(i,j)log_2(1 + \zeta_t^j)$. The former of these parts generally holds a much larger value than the latter, meaning that the reward is heavily biased towards the SINR of CUs. The intuition is that the D2D SINR becomes marginalised when approximating $V(s)$ as it's influence on the reward is much lower.

In order to address this, the value function approximator is split into two 'specialized' approximators, one approximating state values w.r.t CU SINR $V_{CU}(s)$ and the other approximating state values w.r.t D2D SINRs $V_{D2D}(s)$. These approximators are trained similarly to IAC critics with the only difference being the reward they are trained with. Both networks also compute their own TD errors. This creates a problem as the policy network is guided by a single TD error value, hence these separate values must be unified in some way. The method I used to merge them into a single value is through the use of a 'β' parameter.

3.3.1 The β Parameter

The first, and simplest, method is to use what I call the β parameter that weights the respective impact of both terms. The higher the value given to β the more self-interested the agent becomes. This simply involves taking each TD error value, δ_{CU} and δ_{D2D}, and normalizing them into a single term based on the value of β

$$\delta = (1 - \beta)\delta_{D2D} + \beta\delta_{CU} \tag{31}$$

where $0 \geq \beta \geq 1$ and δ is the TD error value used in the policy updates. This method is simple and robust but involves using an addition parameter that the user must specify.

3.3.2 Fully Centralized

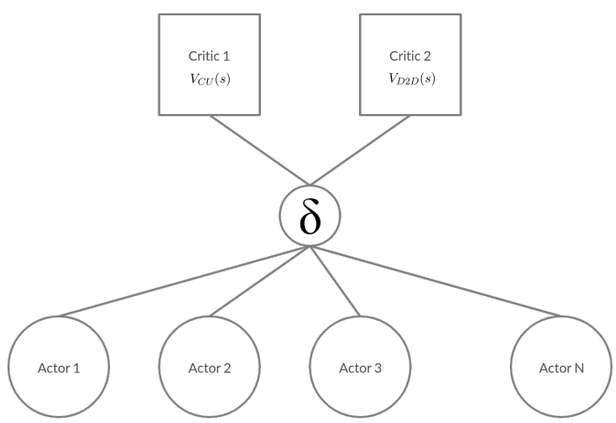


Figure 8: Illustration of the centralized Dual Critic layout.

I also experimented with different kinds of centralization w.r.t the specialized critics. The first method was to have both critics fully centralized (as in Figure 8). This was simple for $V_{CU}(s)$ as the CU SINR portion of the reward is identical for all users, meaning that the state value trained on this reward should also be identical for each user, hence it is natural that this approximator should be centralized. For $V_{D2D}(s)$

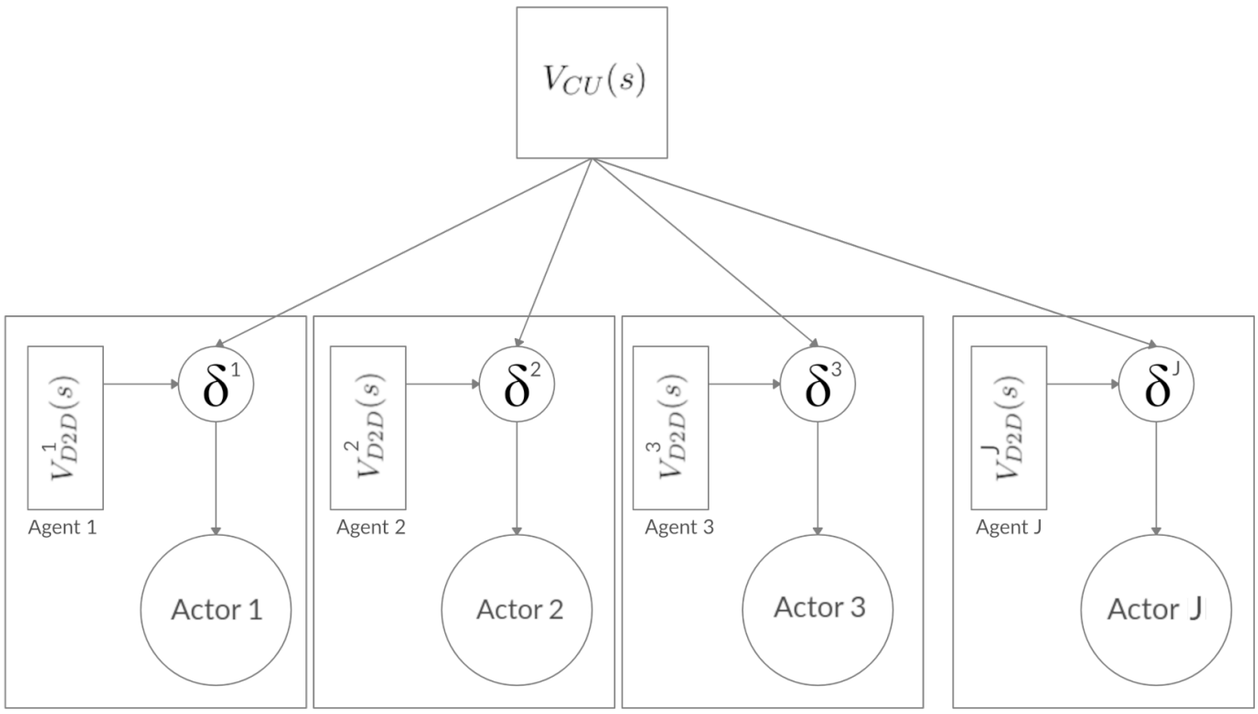


Figure 9: Illustration of the partially centralized Dual Critic layout.

centralization becomes more difficult as each user receives a different reward based on their D2D SINR. A simple way to unify these rewards would be to simply take the mean. This is robust as taking the mean will give a value that is a good general estimate of each agent's performance. The downside however is the lack of credit assignment. As discussed previously, if one agent where to perform sub-optimally while the rest perform well, that agent would have no incentive to make adjustments to it's policy as it's reward would be a misrepresentation of it's performance.

3.3.3 Partially Centralized

In order to combat the lack of credit assignment in a fully centralized D2D SINR state value approximator, an individual, decentralized, state value approximator is assigned to each agent (see **figure 9**). These decentralized critics maintain a $V_{D2D}(s)$ for each agent while a fully centralized $V_{CU}(s)$ maintains a value estimate for the CU SINR portion of the reward. This allows for full credit assignment w.r.t D2D SINR as the decentralized critics are trained with the D2D SINR reward for each agent, providing a bespoke $V_{D2D}(s)$ estimate for these agents. While each agent is equipped with a critic, this can still allow for centralized-learning decentralized-execution as, at execution time, one could simply strip each agent of their critic component (given that their policies don't rely on extra observational information).

3.4 COMA

Where Dual Critic tackles credit assignment by decentralizing part of the value approximation, this method is impractical for the CU SINR reward as this function is identical for each agent. COMA provides an elegant way to perform credit assignment for these types of reward functions by marginalising out a single agent's effect on the reward and comparing the action that agent took at some time-step t with all other actions it could have taken at t. This can be done with a centralized critic, so the value function is identical for all agents, but each individual agent receives a bespoke error term based on their own counterfactual actions. The centralized value function however must be an approximation of the action-value function $Q(s, u)$ as this gives value estimates for actions in states, which is necessary to compute the counterfactual baseline. Similarly to Dual Critic, we can split the value function from a single, centralized, $Q(s, u)$ to a centralized $Q_{CU}(s, u)$ and a decentralized $V_{D2D}(s)$ allowing for credit assignment in both constituents of the reward function.

3.4.1 Reducing the Size of the Output Space

In order for COMA to generalize over joint actions, much larger amounts of information have to be introduced at the input for the $Q(s, u)$ approximation. This is to avoid having a huge output space, as in typical deep Q networks it is required to have an amount of output neurons equal to the action space of the agent $|A^j|$. Since the $Q(s, u)$ approximator is centralized, this would require the **joint** action space of all agents to be represented, leading to an output size of $(|A^j|)^J$ (assuming we use represent every combination of actions for a D2D agent, and we use 15 agents, this is equal to $300^{15} = 1.4 \times 10^{37}$ neurons). As mentioned above, this can be reduced by introducing extra information at the input to the approximator. In particular, the joint actions of the other agents a^{-j} [4].

3.4.2 Replay Buffer

It is also observed that the $Q(s, u)$ approximator used in COMA is burdened by having to consider not only states but actions while learning, along with all the other auxiliary information passed to the input described in 3.4.1. This increases the complexity of the learning task and, as such, decreases the speed of convergence. In order to combat this a replay buffer (AKA experience replay [19]) is introduced to the algorithm that intermittently presents system states from previous time-steps for $Q(s, u)$ approximator updates. This allows the approximator to 're-live' previous experiences to promote convergence as some information may have been lost through catastrophic interference [17], a phenomena in which a neural network can forget previously learned information upon learning new information. This also encourages learning in scenarios where subsequent states are homogeneous [19]. While this homogeneity may not be present in the early stages of learning (due to somewhat random actions being taken before the network has time to mature), the addition of more diverse states can be beneficial when the network has started to converge.

4 Algorithm Implementations

All implementation work was programmed in Python 3.6.9 [5]. Neural Network computations are implemented using TensorFlow 1.14.0 [16] along with loss functions from TRFL [8]. The cellular network simulation software was provided by the authors of [32] at Kings College London and is also written in Python 3. Additional auxiliary packages include: NumPy [37] and Matplotlib [36].

4.0.1 TensorFlow

TensorFlow is a neural network framework built in python that constructs computational graphs which data is allowed to flow through. These graphs contain nodes that perform operations on the data that flow through them. These operations implemented in C++ code, resulting in efficient neural network computations. Before we can train a neural network, we must first specify the graph structure required in order to perform the operations necessary. This includes both the network topology and the update functions. Thankfully, this is simply done by describing the high-level structure of the network to be built and the form of the loss functions we need to minimize/maximize. After the graph is constructed we can feed the input data into the graph and the operations are performed automatically where specified.

4.0.2 TRFL

TRFL is an open-source collection of loss functions for popular RL algorithms. These are implemented in TensorFlow and can be easily integrated into TensorFlow models. These functions are the same used in recent works at the prestigious research group DeepMind [3].

In the following sections, explicit references to variables in code will be denoted `like_this` with a short description of the variable immediately following in brackets. A reference to the specific source code file being discussed will also be provided at the top of each section.

4.1 Individual Actor-Critic

Source code for IAC can be found at [26]

The code used for IAC is modified from [2] which implements the A2C algorithm to play the game 'Lunar Lander'. This modified code also serves a kind of boiler-plate for all subsequent implementations as many elements will remain the same between implementations (e.g. calls to simulation code, graphing/data dumping code etc.).

The actor networks consist of a two hidden layers, each with 64 units and an Exponential Linear Unit (ELU) activation function, followed by the output layer with `action_size` (number of action combinations available to an agent $I \times L$) units and no activation. The output is then passed through a softmax activation function to get the probability distribution over actions. The `policy_logits_` (raw output of the network i.e. before being passed through an activation function) are passed to the loss function operation in order to calculate the loss.

The critic network also consists of two hidden layers, each with 64 units and an ELU activation. The output is a single unit with no activation which provides the $V(s)$ estimation.

The loss function for IAC is provided in TRFL as `trfl.sequence_advantage_actor_critic_loss()`. Note that this loss is for Advantage Actor-Critic (A2C) updates. However, since we are not performing these updates in batch mode, this becomes equivalent to the one-step TD error described previously in this thesis. This function takes, as input:

- the `policy_logits_`
- critic output ($V(s)$)
- reward
- action
- discount (γ)
- a bootstrap value ($V(s_{t+1})$)
  - obtained by instantiating a second 'target' network that shares parameters with the critic network, the only difference between them being their input (s_t for the critic, s_{t+1} for the target network).
- a 'lambda' term
  - used for λ returns, not used so set to 1
- the entropy cost

  - used to set the degree of entropy enforced in the policy output to assist exploration, always set to 0.001

- the baseline cost
  - used to enforce faster convergence in the critic, always set to 10, equivalent to setting the critic learning rate to 10× the actor learning rate

- and a boolean term 'normalise entropy'
  - specifies whether or not to use the entropy cost.

and outputs the loss for both the critic and actor networks. This loss is then fed to an ADAM optimizer [13] which takes the loss and learning rate as input to perform gradient ascent.

Since each agent is equipped with it's own critic, these three components, the actor, the critic and the loss function operations, are encapsulated in a single class which can be instantiated multiple times depending on the number of agents (D2D-Tx) needed. These instantiations are simply appended to a python list, collecting all the agents and their operations in an intuitively indexed structure, which also allows the use of simple `for` loops when calling for operations across all agents.

After the network instantiations, the channel gains are obtained from the cellular network simulator and the initial state s_0 is initialized as a numpy array containing I zeros (this is a fix as initializing to a random set of CU SINRs often caused the networks to break with NaN weight values). Once this is complete we can begin the tensorflow session (`with tf.Session() as sess`) which prepares a tensorflow environment in which the network operations can be executed. With this done we can begin training.

The first step of training is to feed the current state (s_0) into the policy network of each agent to obtain the policy distributions and hence, the action choices of each agent. These action choices can then be used, along with $g^{i,B}$ and $g^{j,B}$ to obtain the next set of CU SINR values i.e. the next state s_{t+1}. The next state, along with the action choices, g^j, $g^{i,j}$ and $g^{j,j}$, are passed to the reward function to produce the reward r_{t+1}. Since the reward is generated, in the simulator, as $W(\sum_{i\in\mathcal{I}} log_2(1+\zeta_t^i) + \mathbf{I}(i,j)log_2(1+\zeta_t^j))$ it is further processed as $\frac{r_{t+1}}{W}$ to greatly reduce it's size in order to avoid overflow errors. Before we can begin the network update procedure, we need to obtain $V(s_{t+1})$ which is done by passing s_{t+1} through the target network to get the value estimate for s_{t+1}. With this value obtained, the TD error and updates are computed automatically by running `trfl.sequence_advantage_actor_critic_loss()` and the optimizer and feeding the appropriate inputs through the network. Finally, the target network's parameters are updated to match those of the updated critic network's, the current state is updated $s_t = s_{t+1}$ and the process repeats T times.

4.1.1 Multi-Head Alterations

Source code for Multi-Headed IAC can be found at [27]

While the code for the generic IAC algorithm and the multi-headed variant are largely the same, there where a few alterations that had to be made to accommodate the extra output in the policy network.

To accommodate the extra head, there is a second set of output neurons. The two heads have I and L neurons for the RB selection and power level selection respectively. These, like before, have separate processing steps for the softmax computation and `policy_logits_` outputs.

The only other difference is the necessity for an additional call to `trfl.sequence_advantage_actor_critic_loss()`. A second loss function is needed due to the fact that each head requires different updates based on their output. Other than the output weights themselves, this means that the hidden weights, and the critic, go through two sets of updates at each time step, one for the RB selection loss and one for the power level selection loss.

4.2 Dual Critic (Centralized)

Source code for Dual Critic (Centralized) can be found at [25]

Dual Critic retains the multi-headed variant of the policy network, meaning that all the policy training procedures are the same as with multi-headed IAC. The difference here lies in the use of two centralized critics. Since `trfl.sequence_advantage_actor_critic_loss()` provides the losses for both the critic and policy networks, these components of the function had to be split so as to separate those computations. This separation produced two loss functions: one that computed the policy gradient loss and facilitated the entropy cost computations on the policy distribution that I named `trfl.sequence_advantage_actor_loss`, and one that computed the critic loss and TD error terms that I named `trfl.sequence_advantage_critic_loss`. Additionally, this allowed for the removal of the 'baseline cost' term as the learning rate of the critic could now be explicitly stated. While these functions are likely already implemented separately in TRFL, my aim was to keep these functions as similar to those used in IAC to make more accurate comparisons w.r.t the validity of the algorithm idea, rather than the quality of it's implementation.

The splitting of the loss function also necessitated the splitting of the class structure used previously, with two separate classes (one for the actor, one for the critic) being implemented. The content of these classes are identical to IAC, the only difference being that the respective networks and loss functions are contained in their own classes. This is useful as it allows the number of critics to be unequal to the number of actors, making centralized learning possi-

ble.

For centralized Dual Critic, two centralized critics are instantiated: `social_central_critic` and `individual_central_critic` for $V_{CU}(s)$ and $V_{D2D}(s)$ respectively. Since they are both instatiated from the same class, their network structures are identical. Like before, there are also individual policy networks instantiated for each agent (D2D-Tx) that are collected in a python list. Since the critics learn from separate rewards, the reward function in the simulator code had to be modified to account for this. However, since the reward function computed the D2D and CU components of the reward separately anyway, it was simply modified to additionally return those individual values.

The actor networks contain 3 hidden layers with an ELU activation and the same separate processing steps for the softmax computation and `policy_logits_` outputs. The critics also contain 3 hidden layers with an ELU activation and the same single neuron output with no activation.

Updates are, naturally, performed differently as they are no longer handled automatically in a single loss function. The first networks to be updated are the critics, with each critic being trained on a separate component of the reward. The `social_central_critic` ($V_{CU}(s)$) is trained on the `socialist_reward` (CU SINR reward), so called as it identical among the agents. The `individual_central_critic` ($V_{D2D}(s)$) is more difficult as the individual D2D SINR rewards are different for each agent. In order to unify these rewards, their statistical mean is used in training as it gives a rough summary to the rewards being received.

Only after the updates are performed do we compute the TD error for use in the actor updates (Note that the TD error is also computed for the critic updates, but this is automated by their respective loss functions). Since there are two actors, there are two TD error terms `socialist_advantage` (δ_{CU}) and `individualist_advantage` δ_{D2D}. As described in 3.3.1, these are mixed using a β parameter, in code: `advantage = ((1 - beta) * individualist_advantage) + (beta * socialist_advantage)`. The lower the value of `beta` the more 'individualist' the TD error term becomes. It is important to note that, while training, the individual critic used the mean of the set of D2D SINR rewards, however the TD error (δ_{D2D}) terms are bespoke for each agent based on their individual D2D SINRs. After the TD error is obtained, it is fed to the policy updates, with the rest of the code being identical to IAC.

4.3 Dual Critic (Partially Centralized)

Source code for Dual Critic (Partially Centralized) can be found at [24]

The partially centralized version of Dual Critic is implemented largely the same as centralized Dual Critic. The main difference being that the `individual_central_critic` is no longer centralized. Instead each agent is assigned it's own $V_{D2D}(s)$ critic. This is implemented much like the policy networks. Individual critic networks are appended to a python list to make for simple indexing. Individual critic updates are performed using the D2D SINR reward of the agent they are assigned to, instead of the statistical mean of these rewards like in the centralized variant and the TD error for the actor networks is computed with each agent's individual critic network instead of a centralized one.

4.4 COMA (Centralized)

Source code for COMA (Centralized) can be found at [23]

The COMA implementation is quite different to the previous ones, in large part due to the fact that the critic is no longer estimating $V(s)$ but $Q(s, a)$. In practice, this means that the first change is to ensure the critic network output is of equal size to the amount of actions available to an agent (in order to provide action values $Q(s, a)$ for each action). Since using the multi-headed variant of the policy would over-complicate this, the outputs again contain $I \times L$ neurons. Secondly, the loss function is changed to `trfl.qlambda()` which computes the loss for the critic. This function takes as input

- `q_tm1`
  - The set of action values output by the critic
- `a_tm1`
  - The action chosen by the policy network
- `r_t`
  - The reward received for taking that action
- `pcont_t`
  - The discount factor (γ)
- `q_t`
  - The action values output by the target network (equivalent to the bootstrap value)
- `lambda_`
  - used for λ returns, not used so set to 1

and outputs the loss for the $Q(s, a)$ critic. It is also important to note that this function computes the TD error for the critic as

$$\delta_t = r_{t+1} + \gamma(\max Q_\lambda(s_{t+1}, a_t^j) - Q_\lambda(s_t, a_t^j \sim \pi_\theta(a_t^j|s_t))) \tag{32}$$

where $\max Q_\lambda(s_{t+1}, a_t^j)$ is the largest action value for state s_{t+1} and $Q_\lambda(s_t, a_t^j \sim \pi_\theta(a_t^j|s_t))$ is the action value of the action selected by $pi_\theta(a_t^j|s_t)$ in state s_t.

The policy networks also use a new loss function `trfl.discrete_policy_gradient()`. This function takes as input

- `policy_logits`
  - The raw output of the policy network

- `actions`
  - The action chosen by the policy network
- `action_values`
  - the TD error

and outputs the policy network loss. This function is important due to the inclusion of the `action_values` parameter. This allows us to specify our own custom baseline value. With this we can compute the counterfactual baseline and pass it to the loss function.

As discussed in 3.4.1, in order to avoid a huge number of output neurons, the critic is fed additional input terms to approximate $Q(s, u)$. These inputs are

- `joint_action_min_a`
  - The joint actions of all agents, excluding those of the agent who's action values are currently being approximated
- `current_actor`
  - The index of the current agent who's action values are being approximated
- `joint_action_tm1`
  - the joint actions of **all** agents at $t-1$.

This allows the approximation to output the action values of a single agent, conditioned on the actions of all other agents, which we can use to compute the counterfactual baseline. A problem arises when considering that `trfl.qlambda()` only computes the loss conditioned on a single agent's action at a time i.e. the loss for $Q(s, a^j)$ instead of $Q(s, u)$. This means that we must perform the critic update once for every agent at every t in order to consider the joint action.

The counterfactual baseline is defined as

$$A^j(s_t, u_t) = Q(s_t, u_t) - \sum_{a_t'^j} \pi^j(a_t'^j|s_t) Q(s_t, (u_t^{-j}, a_t'^j))$$

where $a_t'^j$ is every action that agent j **has not** taken at time t, $\pi^j(a_t'^j|s_t)$ is the probability for every action agent j has not taken at time t in state s_t as given by agent j's policy and $Q(s_t, (u_t^{-j}, a_t'^j))$ is the action values of actions $a_t'^j$ given by the centralized critic for state s_t, conditioned on the joint actions of all agents **excluding** agent j (u_t^{-j}). In code this is computed by first obtaining the action probabilities from the policy network for each agent, stored as a nested list of size (J, A) where J is the number of agents and A is the number of possible actions for a single agent. Then the action values are obtained from the critic output for each agent, which have the same shape as the action probabilities. Since we are updating the policy networks in a `for` loop over all agents, we can simply insert another `for` loop just before the update procedure that sums the element-wise product of the jth list for both network outputs to produce a '`base`'. In pseudo-code:

```
for i = 0 to J:
    base = 0
    for j = 0 to A:
        # compute base
        base += action_probs[i][j] *
                action_values[i][j]
        # remove taken action from base
        base = base -
               (action_values[i][actions[i]] *
                action_probs[i][actions[i]])
```

after which the action taken by agent j is removed from the `base` value and the final baseline is computed by subtracting `base` from the action value of the action taken by agent j ($Q(s, u)$) and is then used in the policy update:

```
for i = 0 to J:
    base = 0
    for j = 0 to A:
        # compute base
        base += action_probs[i][j] *
                action_values[i][j]
        # remove taken action from base
        base = base -
               (action_values[i][actions[i]] *
                action_probs[i][actions[i]])
    # compute the final baseline
    baseline = action_values[i][actions[i]] -
          base
    # update network with final baseline
    update_network(j, baseline)
```

The last addition to the COMA algorithms is the use of a reply buffer. This is implemented by defining a global boolean `replay_time_step`. If `replay_time_step` is set to false, then updating continues as normal. In this case the inputs to the loss functions are stored in a replay buffer for later use (to a maximum of 10% the number of training episodes T). There are two reply buffers, one for each loss function, and they function identically. If `replay_time_step` is set to true then a random selection of inputs is made from each replay buffer and passed to the respective loss function instead of using the simulator. The probability of this happening is set before training (usually around 10-15%).

4.5 COMA (Partially Centralized)

Source code for COMA (Partially Centralized) can be found at [22]

The partially centralized version of the COMA implementation includes, like the partially centralized dual critic, an individual critic for each agent that approximates $V_{D2D}(s)$ based on the D2D SINR rewards for these agents. Since the COMA critic approximates the action value function $Q(s, a)$ we need to define an additional network class in order to instantiate a state value critic. This class is identical to the critic net class in the partially centralized dual critic implementation, using `trfl.sequence_advantage_critic_loss()` as the loss function. Also, like the dual critic implementation, the individual critics are trained with the D2D SINR reward for each individual agent, where the COMA critic is now trained solely on the CU

SINR rewards. These two critic's baseline function are mixed with the β parameter method exactly like in dual critic. One last thing to note is that the replay buffer is still used in this version of the COMA implementation, however the individual D2D SINR critics are not included in the experience replays.

5 Results and Analysis

5.1 Performance Metrics

Like in [32] three different measures where used to evaluate the performance of each algorithm.

The first is the D2D Collision Probability Ψ_T. A function is defined $\psi(a_t^j)$. This function is an indicator function of agent's actions. $\psi(a_t^j) = 1$ when a collision happens (multiple D2D users access the same RB at the same time) and $\psi(a_t^j) = 0$ otherwise. With this function the D2D collision can be defined as

$$\Psi_T = \frac{\sum_{t=0}^{T} \psi(a_t^j)}{T} \tag{33}$$

Second is the D2D access rate Λ_T. The access rate Λ_t is defines as the ratio of accessed D2D pairs that meet the minimum SINR requirements for both D2D users and CUs, to the total number of D2D pairs over T

$$\Lambda_T = \frac{\sum_{t=0}^{T} J_t^a / J}{T} \tag{34}$$

where J_t^a denoted the number of accessed D2D pairs.

Finally there is the time-averaged overall throughput F_T. The overall throughput is defined as

$$f_t = \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} log_2(1 + \zeta_t^i) + \mathbf{I}(i,j) log_2(1 + \zeta_t^j) \tag{35}$$

which is averaged over the time horizon T

$$F_T = \frac{\sum_{t=0}^{T} f_t}{T} \tag{36}$$

Note that the channel bandwidth (W) is not included in the time-averaged throughput calculation. As this is simply a coefficient value that is multiplied to each term, it's absence is minor and increases the simplicity of the calculations.

5.2 IAC

In order to compare the performance of the algorithms presented in this thesis, we need a benchmark with which to compare further algorithms. Since IAC is the simplest of the algorithms discussed it is selected as this benchmark. All algorithms are compared over a time horizon of 5000 time-steps ($T = 5000$), in a cellular network consisting of 15 D2D pairs and 30 CUs. The locations of D2DUs and CUs are kept the same for each experiment to more accurately measure performance and each plot is smoothed to improve readability.

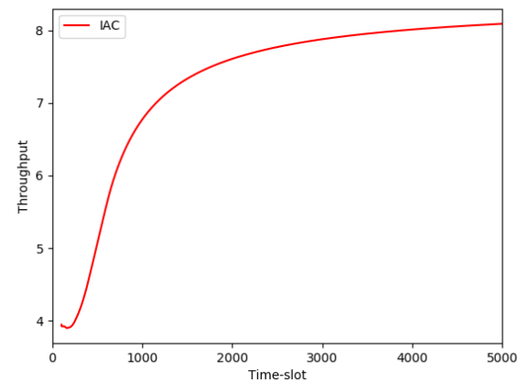


Figure 10: IAC time-averaged throughput over time

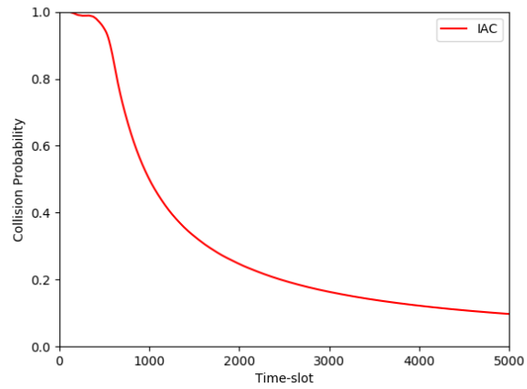


Figure 11: IAC collision probability over time

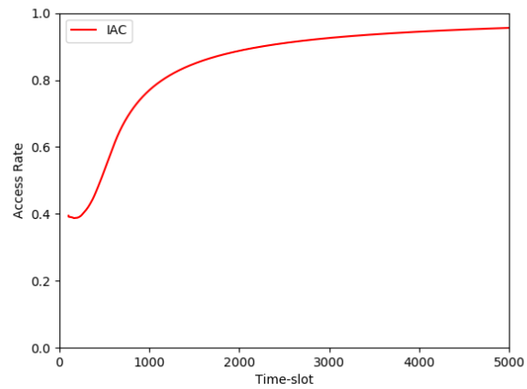


Figure 12: IAC access rate over time

As is observed in **figures 12, 11, 10**, IAC's convergence to a highly performing policy is quite fast. Access rate is approx 95% after 5000 time-steps and collision probability is approx 15% with throughput settled at around 8. This is achieved with $\gamma = 0.99$, $\alpha = 0.001$, `baseline_cost` $= 10$ (effectively making the critic learning rate 0.01) and `entropy_cost` $= 0.001$.

5.3 Multi-Headed IAC

The following figures show a comparison between standard IAC and the Multi-Headed variant of IAC. The parameters used in the Multi-Headed IAC implementation are identical to those of standard IAC.

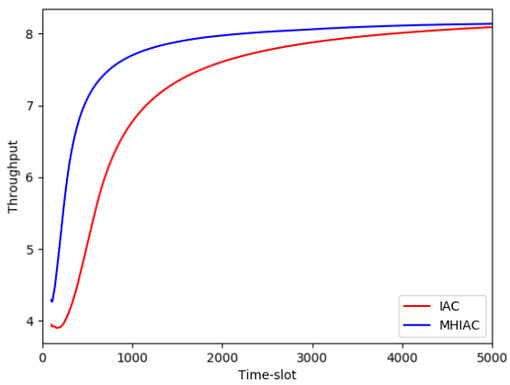


Figure 13: Multi-Headed IAC time-averaged throughput over time compared with IAC

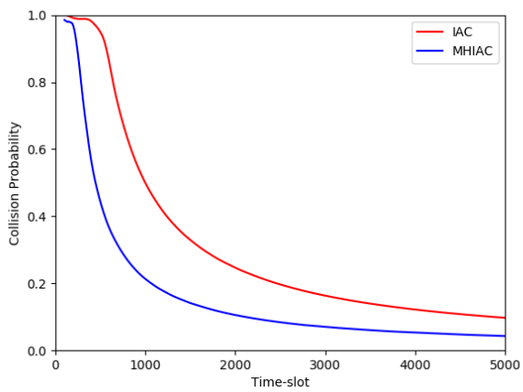


Figure 14: Multi-Headed IAC collision probability over time compared with IAC

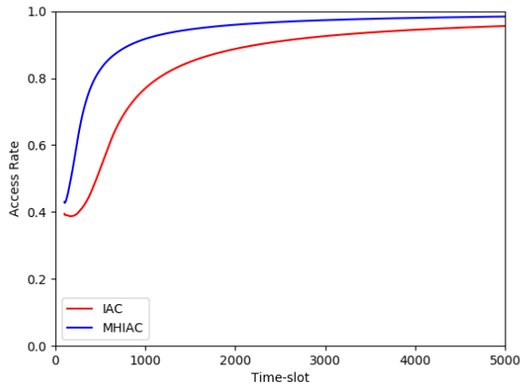


Figure 15: Multi-Headed IAC access rate over time compared with IAC

As is evident in **figures 15, 14, 13**, the multi-headed variant of IAC produces slightly improved performance of the model while reducing the amount of learnt parameters, at the output, drastically. It can be observed that multi-headed IAC, initially, has a much more rapid increase in performance as compared to standard IAC, which tends to need some time to explore the state space before finding it's footing.

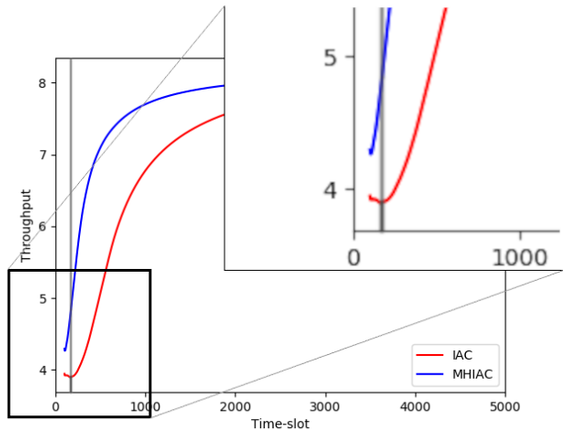


Figure 16: Illustration of the difference of the initial rate of learning between IAC and MHIAC

This suggests that the network structure of the multi-headed IAC implementation could be slightly better suited to the learning problem while, conversely, this could simply be down to better random initial weights in the multi-headed IAC network.

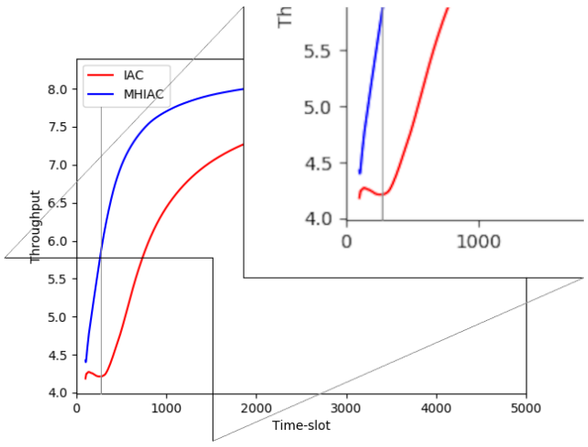


Figure 17: Second experiment showing the slow initial learning performance of IAC as compared to MHIAC

Figure 17 shows the results of a second experiment under the same conditions as in **figure 16**. These results reinforce the notion that the multi-headed IAC is better suited to the learning problem. It is still important to note however that both networks are still subject to random initial parameters, meaning that the magnitude of this phenomena varies between experiments.

The downside, however, of the multi-headed variant is the necessity for two updates to the policy network at each time step. The simplicity of the models used here meant that the impact of this was barely noticeable, but for more complex networks, that take longer to update, this could lead to a drastic increase in training time.

5.4 Dual Critic (Centralized)

Before we compare the centralized variant of dual critic to IAC, we must first establish the best performing value of the TD error mixing parameter β. The following figure shows a comparison of the time averaged throughputs of different values of β starting from 0.1 and increasing in 0.1 increments until 0.9

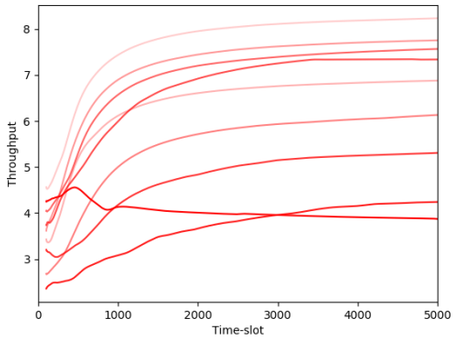


Figure 18: Plots of learning performance w.r.t throughput in Centralized Dual Critic with a range of β values. The darker the plot, the higher the β value.

All experiments here used an actor with 3 hidden layers, each of which contain 64 units with the ELU activation function and a multi-headed output. The critics contain 3 hidden layers, each of which contain 32 units with the ELU activation function. $\gamma = 0.99$, the actor learning rate $\alpha_\theta = 0.001$ and the critic learning rate $\alpha_\omega = 0.01$. As is evident in **figure 18** the performance achieved by different values of β is varied. A large proponent of this variation is the sensitivity of these networks to their random initial parameter settings. To further illustrate this variance, **figure 19** shows the final throughput values of Centralized Dual Critic over the range of β values.

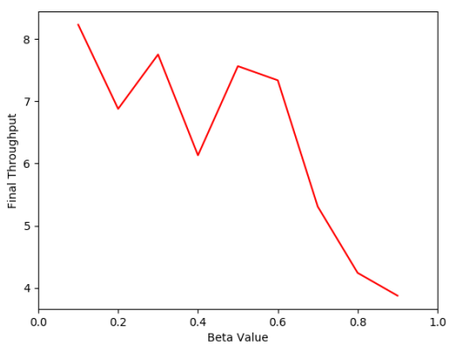


Figure 19: Plot showing the final throughput values of Centralized Dual Critic as the β value changes.

Figure 19 shows that the final throughput for βs 0.1 through 0.5 are quite unstable. Given the instability of these networks, it is still clear to see however, that lower values of β on average tend to perform better than higher values (shown by the steady fall in achieved throughput as β is increased). Recall that lower β values result in more emphasis being put on the D2D SINR reward during training. This shows that there is some merit to splitting the critics up in this way, as learning is more fruitful when both constituents of the reward function are appropriately represented during learning.

In the interest of comparison, the best performing of these networks ($\beta = 0.1$) is compared with IAC:

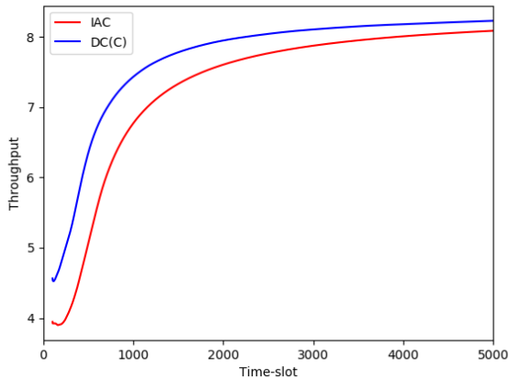


Figure 20: Dual Critic (Centralized, $\beta = 0.1$) throughput over time compared with IAC

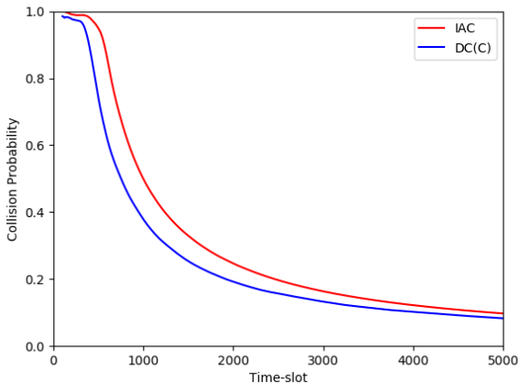


Figure 21: Dual Critic (Centralized, $\beta = 0.1$) collision probability over time compared with IAC

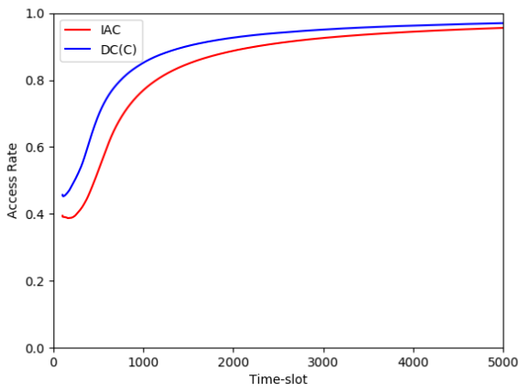


Figure 22: Dual Critic (Centralized, $\beta = 0.1$) access rate over time compared with IAC

As is evident in **figures 20, 21, 22** the performance of Centralized Dual Critic is very similar to that of IAC, although slightly increased. This suggests that, while performance is increased, the difference between the performance of these algorithms overall is minimal. This can be due to the fact that this version of Dual Critic uses a centralized critic to approximate $V_{D2D}(s)$ with the statistical mean of the D2D SINR reward, leading to sub-optimal performance in individual agents, similar to how using an integrated reward may do the same.

5.5 Dual Critic (Partially Centralized)

The partially centralized variant of Dual Critic also relies on a β parameter, so the same process is followed as with the centralized version to find the best performing value, the hyperparameters and network structures are identical to the centralized variation.

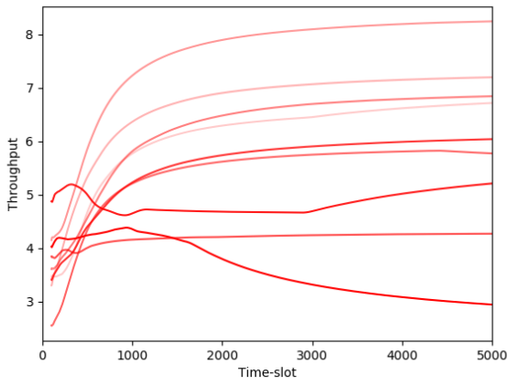


Figure 23: Plots of learning performance w.r.t throughput in Partially Centralized Dual Critic with a range of β values. The darker the plot, the higher the β value.

and then a comparison of the final throughput values:

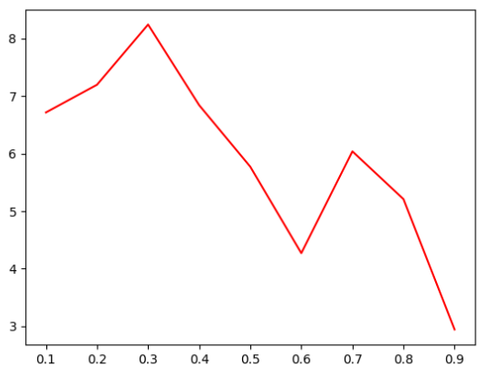


Figure 24: Plot showing the final throughput values of Partially Centralized Dual Critic as the β value changes.

Again, **figure 24** shows that a lower value of β results in higher performance for these algorithms. This reinforces the notion that there is benefit to an appropriate representation of both constituents of the reward function. The best performing β value in the partially centralized case of Dual Critic is $\beta = 0.3$ which is compared to IAC:

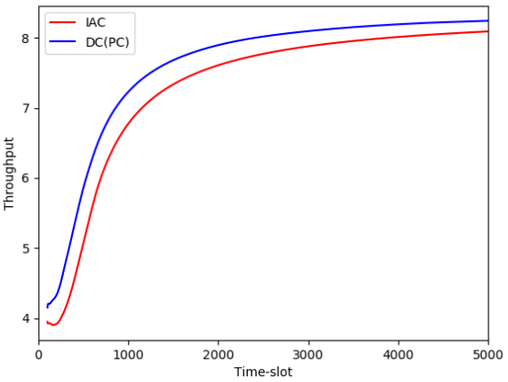


Figure 25: Dual Critic (Partially Centralized, $\beta = 0.3$) throughput over time compared with IAC

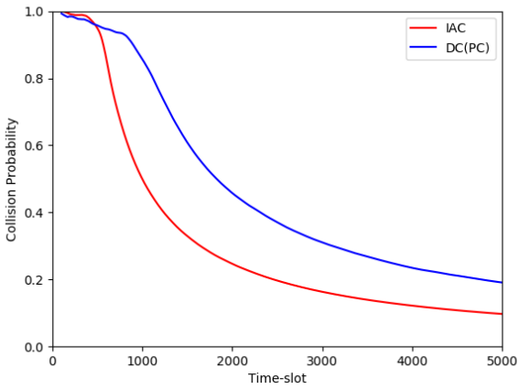


Figure 26: Dual Critic (Partially Centralized, $\beta = 0.3$) collision probability over time compared with IAC

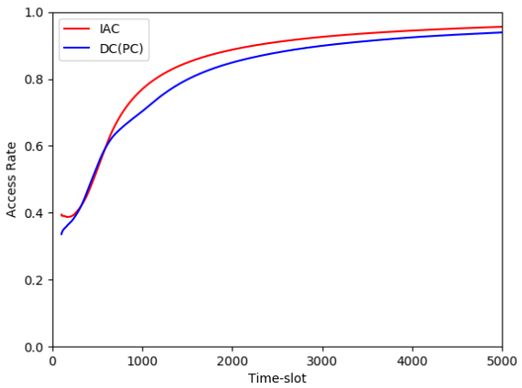


Figure 27: Dual Critic (Partially Centralized, $\beta = 0.3$) access rate over time compared with IAC

Interestingly, while **figure 25** shows that the throughput for partially centralized dual critic is strictly improved over IAC (although only slightly), both the collision probability and access rate are worse (as shown in **figures 26 and 27** respectively). This suggests that, despite having difficulties during learning, the partially centralized dual critic is still able to maintain a high performance in individual agents when other agents in the system are following sub-optimal policies (it is robust). This is likely due to the fact that each agent is equipped with their own critic and, hence, is able to maintain it's own performance more effectively when the global reward is low. It can be argued that IAC must possess this

quality as it's agents are also equipped with individual critics. However, the IAC critics use a reward function that is heavily integrated between CU SINR and D2D SINR, and as such projects the possibility that this, integrated, reward retains the ability to corrupt an individual agent's feedback w.r.t it's own performance, leading to sub-optimal behaviour.

5.6 COMA (Centralized)

Since Centralized COMA only makes use of a single centralized critic, there is no need to make any comparisons w.r.t a mixing parameter. The critic network contains 4 hidden layers, each with 64 hidden units and an ELU activation function and the output contains A units with no activation. The actors contain 4 hidden layers, each with 64 units and an ELU activation function. The replay buffer is triggered with probability 0.15. The rest of the parameters are identical to the centralized Dual Critic. The following show a comparison of COMA and IAC:

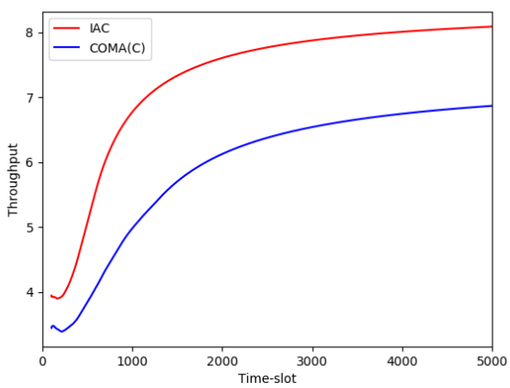


Figure 28: COMA (Centralized) throughput over time compared with IAC

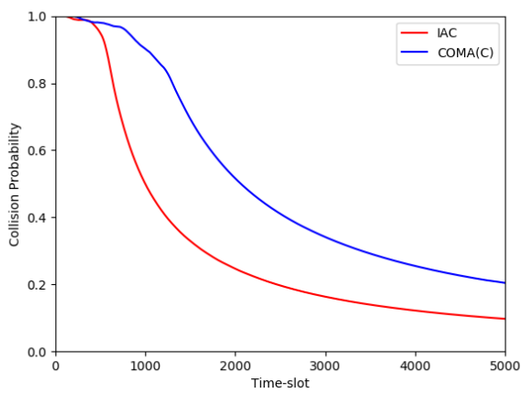


Figure 29: COMA (Centralized) collision probability over time compared with IAC

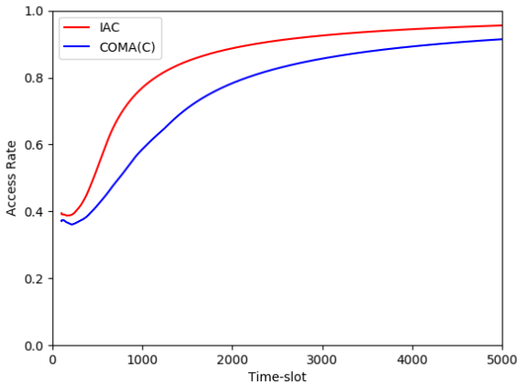


Figure 30: COMA (Centralized) access rate over time compared with IAC

As shown in **figures 28, 29, 30** centralized COMA drastically under-performs as compared to IAC. There are a number of possibilities for this, the most likely of which is that the action-value function used in the critic $Q(s, a)$ is much more unstable, and hence difficult to train, than using $V(s)$. This is due to updates being dependant on specific actions taken within a state instead of solely proving value approximations for states themselves. Another factor is the complexity of COMA in general. Where IAC and Dual Critic use the simple TD error term (or a mix thereof) in policy updates, COMA relies on a complex computation of the counterfactual baseline, which itself is highly dependant on the quality of the value function approximation in order to be effective, something which is difficult to attain with $Q(s, a)$. One accolade worth noting however is the comparatively high access rate. This suggests that the counterfactual baseline did, to some degree, provide useful feedback for individuals based on their counterfactual actions.

5.7 COMA (Partially Centralized)

We first compare the performance of partially centralized COMA with different values of β:

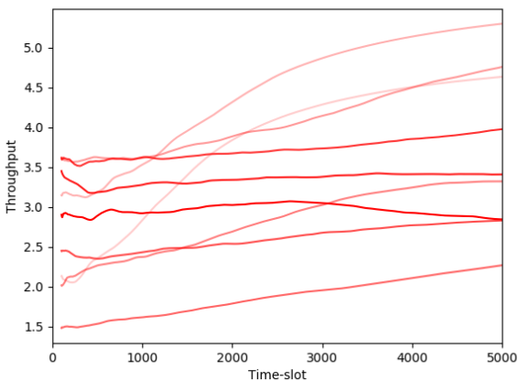


Figure 31: Plots of learning performance w.r.t throughput in Partially Centralized COMA with a range of β values. The darker the plot, the higher the β value.

And then a comparison of the final throughput values:

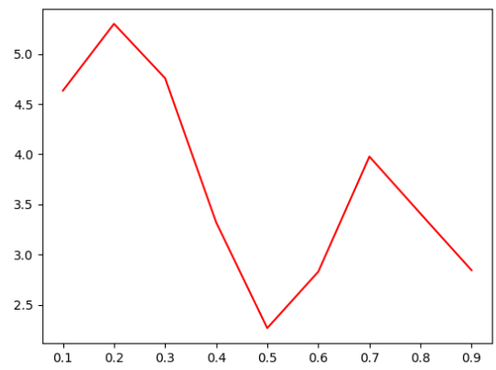


Figure 32: Plot showing the final throughput values of Partially Centralized COMA as the β value changes.

Again, as seen in **figure 32**, the highest performing β values are the lower ones. Interestingly however, the worst performing values are those from 0.4 to 0.6. The most likely explanation is simply due to the random initial parameters. This is shown when looking at **figure 31** as those experiments with a β value higher than 0.3 scarcely improve over time. This suggests that when the baseline is more emphasised on that provided by the central COMA critic (Higher β values), learning simply stops, meaning that the COMA critic is providing no benefit to learning whatsoever. The parameters are identical to the centralized version of COMA.

The best performing of these is with $\beta = 0.2$. This is compared with IAC:

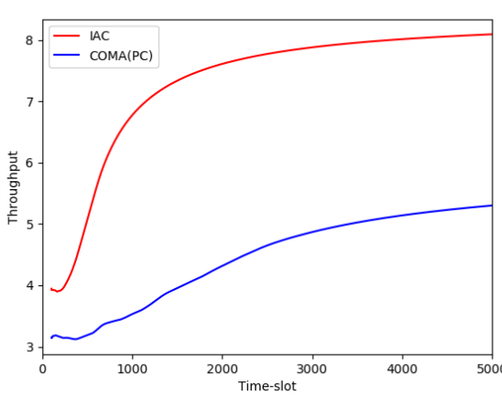


Figure 33: COMA (Partially Centralized) throughput over time compared with IAC

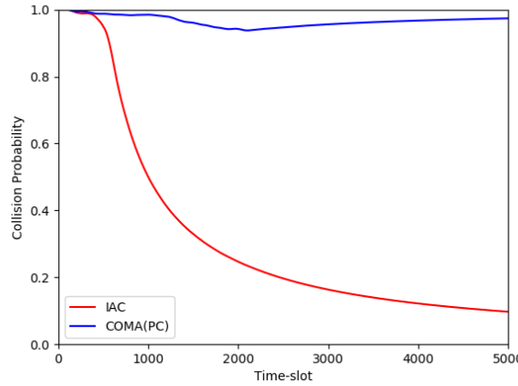


Figure 34: COMA (Partially Centralized) collision probability over time compared with IAC

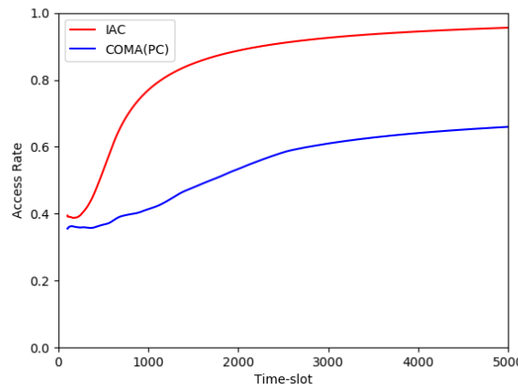


Figure 35: COMA (Partially Centralized) access rate over time compared with IAC

Figures 33, 34, 35 show that the partially centralized version of COMA drastically under performs when compared to IAC and even the fully centralized version of COMA. This is suggests that the impact of the counterfactual baseline is minimal in the D2D setting. While the centralized version of COMA did at least appear to learn, this could be solely due to the impact of the D2D SINR rewards. This notion is reinforced when considering that higher values of β (in which more emphasis will be placed on the centralized COMA critic) produced throughputs that didn't improve as time progressed, suggesting that learning didn't occur.

5.8 Overall Comparison

Finally, all algorithms are compared against each other to determine the best performing in the D2D channel and power level selection setting:

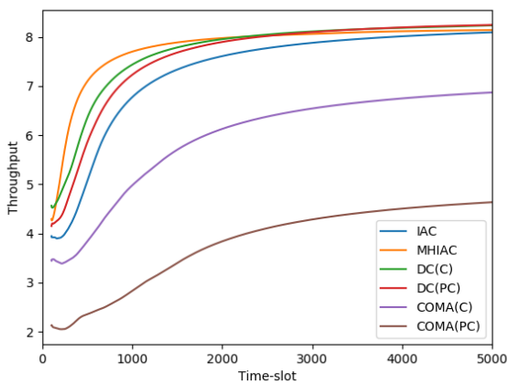


Figure 36: Throughput over time for all algorithms

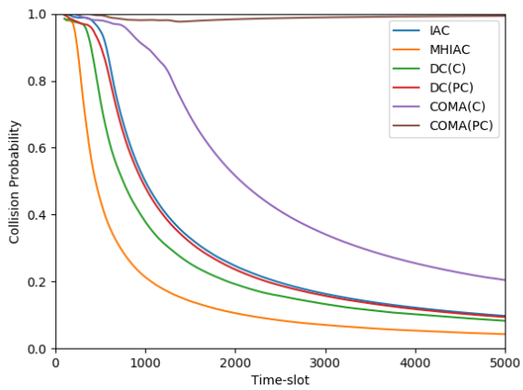


Figure 37: Collision Probability over time for all algorithms

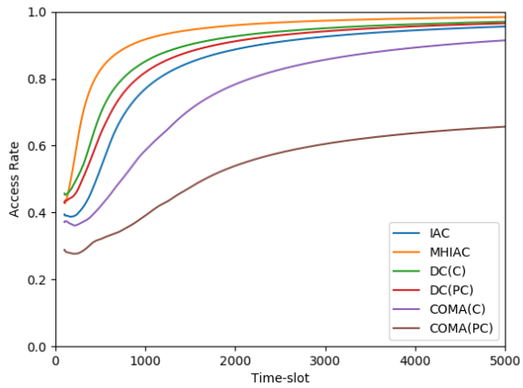


Figure 38: Access Rate over time for all algorithms

Overall the best performing algorithm w.r.t time averaged throughput is the Partially Centralized variant of Dual Critic. The best performing w.r.t access rate and collision probability however is the multi-headed variant of IAC. This suggests that while multi-agent consideration has the potential to be valuable to the D2D setting, the use of single agent methods still produce very strong results.

6 Conclusions

In conclusion, this thesis has discussed the theory, experimentation and implementation of five different DRL algorithms, based on Actor Critic, within the Device to Device communication setting tackling the problems of channel selection and power level control of these devices within a cellular network.

Channel selection and power level control in D2D communication is a highly complex problem that demands sophisticated and efficient systems to automate. Reinforcement Learning is a natural choice for a solution as it is well known to produce highly complex and impressive control policies that can learn and perform well in scenarios historically thought to be impossible to automate. Paired with Deep Neural Networks, reinforcement learning possesses the ability to learn solely through trial and error experience in an unknown system without needing to expressly define and describe every unique facet and detail of that system. While a large portion of the literature is focused on producing effective algorithms that can perform extremely well in single agent environments, the D2D enabled network is so complex, in part, due to the complicated and interdependent relationships between devices. Multi-Agent Reinforcement Learning is a slowly popularising sub-field of reinforcement learning that specialises in the development of algorithms that can perform well in scenarios in which the presence of multiple, interacting, agents is a key factor. In hopes of capitalizing off of the benefits of multi-agent consideration, this thesis describes the exploration and development of a number of multi-agent reinforcement learning algorithms and compared their performance to one of the leading single agent algorithms today.

The results of this experimentation have shown that the partially centralized Dual Critic algorithm produces the best performance w.r.t time averaged throughput and the multi-headed individual actor critic has performed best w.r.t collision probability and access rate. This shows that, while single agent focused algorithms can perform well in the D2D communication setting, there is benefit to using a multi-agent approach when trying to find a solution to the problem.

6.1 Further Study

Going forward, there is still plenty of work that can be done on the algorithms discussed in this thesis. First is to develop a more sophisticated method with which to merge the TD errors produced by separated critics. Given the power of neural networks, it is possible that this process itself can be automated, ruling out the need for an additional hyperparameter to tune. While the COMA implementation was the worst performing of all the algorithms presented, I still feel as though more work could be done to improve it's performance. One area of which to look at would be to use a $Q(s, a)$ approximator that can learn based on a single update for the joint action at every time step, instead of having to use J separate updates at each time step. The project could also be

further developed by considering higher numbers of D2D users in the network and developing the algorithms to work better in those more complex scenarios. Another, more natural, extension of the project would be to implement and compare more powerful single-agent focused algorithms like Asynchronous Advantage Actor Critic [18], and integrate them with some of the techniques developed in this thesis, likely providing a strong increase in performance over those algorithms.

7 Appendix

7.1 Algorithms

7.1.1 IAC

Algorithm 1: IAC Algorithm

initialize hyperparameters: γ, α_θ, α_ω;
initialize s_0;
initialize θ^0, θ^1, ..., θ^J, ω^0, ω^1, ..., ω^J;
for $t = 0, 1, ..., T$ **do**
  for $j = 0, 1, ..., J$ **do**
    Sample agent j's action from the jth policy: $a_t^j \sim \pi_\theta^j(a_t^j|s_t)$
  end
  Take joint action a_t in state s_t and each agent receives reward r_{t+1}^j and next state s_{t+1} from simulator;
  for $j = 0, 1, ..., J$ **do**
    Update critic parameters $\omega_{t+1}^j = \omega_t^j + \alpha_\omega \nabla_\omega V_\omega^j(s_t)\delta_t^j$;
    Calculate TD Error $\delta_t^j = r_t^j + \gamma V_\omega^j(s_{t+1}) - V_\omega^j(s_t)$;
    Update actor parameters $\theta_{t+1}^j = \theta_t^j + \alpha_\theta \nabla_\theta \pi_\theta^j(a_t^j|s_t)\delta_t^j$;
  end
  Each agent updates state $s_t = s_{t+1}$;
end

7.1.2 Dual Critic (Centralized)

Algorithm 2: Dual Critic (Centralized) Algorithm

initialize hyperparameters: γ, α_θ, α_ω α β; initialize s_0;
initialize ω, ϕ, θ^0, θ^1, ..., θ^J;
for $t = 0, 1, ..., T$ **do**
  for $j = 0, 1, ..., J$ **do**
    Sample agent j's action from the jth policy: $a_t^j \sim \pi_\theta^j(a_t^j|s_t)$
  end
  Take joint action a_t in state s_t and receive CU SINR reward r_{t+1}^{CU}, D2D SINR reward r_{t+1}^{D2D} and next state s_{t+1} from simulator;
  Calculate CU SINR TD Error $\delta_t^{CU} = r_{t+1}^{CU} + \gamma V_\omega^{CU}(s_{t+1}) - V_\omega^{CU}(s_t)$;
  Update CU critic parameters $\omega_{t+1} = \omega_t + \alpha_\omega \nabla_\omega V_\omega^{CU}(s_t)\delta_t^{CU}$;
  Calculate D2D SINR TD Error $\delta_t^{D2D} = r_{t+1}^{D2D} + \gamma V_\phi^{D2D}(s_{t+1}) - V_\phi^{D2D}(s_t)$;
  Calculate the mixed TD error $\delta_t = (1-\beta)\delta_t^{D2D} + \beta\delta_t^{CU}$;
  Update D2D critic parameters $\phi_{t+1} = \phi_t + \alpha_\phi \nabla_\phi V_\phi^{CU}(s_t)\delta_t^{D2D}$;
  for $j = 0, 1, ..., J$ **do**
    Update actor parameters $\theta_{t+1}^j = \theta_t^j + \alpha_\theta \nabla_\theta^j \pi_\theta^j(a_t^j|s_t)\delta_t$;
  end
  Each agent updates state $s_t = s_{t+1}$;
end

7.1.3 Dual Critic (Partially Centralized)

Algorithm 3: Dual Critic (Partially Centralized) Algorithm

initialize hyperparameters: γ, α_θ, α_ω α_ϕ β; initialize s_0;
initialize ω, ϕ^0, ϕ^1, ..., ϕ^J, θ^0, θ^1, ..., θ^J;
for $t = 0, 1, ..., T$ **do**
  for $j = 0, 1, ..., J$ **do**
    Sample agent j's action from the jth policy: $a_t^j \sim \pi_\theta^j(a_t^j|s_t)$
  end
  Take joint action a_t in state s_t and receive CU SINR reward r_{t+1}^{CU}, D2D SINR reward r_{t+1}^{D2D} and next state s_{t+1} from simulator;
  Calculate CU SINR TD Error $\delta_t^{CU} = r_{t+1}^{CU} + \gamma V_\omega^{CU}(s_{t+1}) - V_\omega^{CU}(s_t)$;
  Update CU critic parameters $\omega_{t+1} = \omega_t + \alpha_\omega \nabla_\omega V_\omega^{CU}(s_t)\delta_t^{CU}$;
  for $j = 0, 1, ..., J$ **do**
    Calculate D2D SINR TD Error for agent j, $\delta_t^{j,D2D} = r_{t+1}^{j,D2D} + \gamma V_\phi^{j,D2D}(s_{t+1}) - V_\phi^{j,D2D}(s_t)$;
    Update D2D critic parameters for agent j, $\phi_{t+1}^j = \phi_t^j + \alpha_\phi \nabla_\phi^j V_\phi^{j,D2D}(s_t)\delta_t^{j,D2D}$;
    Calculate the mixed TD error for agent j, $\delta_t^j = (1-\beta)\delta_t^{j,D2D} + \beta\delta_t^{CU}$;
    Update actor parameters for agent j, $\theta_{t+1}^j = \theta_t^j + \alpha_\theta \nabla_\theta^j \pi_\theta^j(a_t^j|s_t)\delta_t^j$;
  end
  Each agent updates state $s_t = s_{t+1}$;
end

7.1.4 COMA (Centralized)

Algorithm 4: COMA (Centralized) Algorithm

initialize hyperparameters: γ, α_θ, α_λ;
initialize s_0;
initialize λ, θ^0, θ^1, ..., θ^J;
for $t = 0, 1, ..., T$ **do**
  for $j = 0, 1, ..., J$ **do**
    Sample agent j's action from the jth policy: $a_t^j \sim \pi_\theta^j(a_t^j|s_t)$
  end
  Take joint action a_t in state s_t and each agent receives reward r_{t+1}^j and next state s_{t+1} from simulator;
  Calculate TD error for critic $\delta_t = r_{t+1} + \gamma(\max Q_\lambda(s_{t+1}, a_t^j) - Q_\lambda(s_t, a_t^j \sim \pi_\theta(a_t^j|s_t)))$;
  Update critic parameters $\lambda_{t+1} = \lambda_t + \alpha_\lambda \nabla_\lambda Q_\lambda(s_t, u_t)\delta_t$;
  for $j = 0, 1, ..., J$ **do**
    Calculate counterfactual baseline for agent j,
    $A^j(s,u) = Q_\lambda(s_t, u_t) - \sum_{a_t'^j} \pi_\theta^j(a_t'^j|s_t) Q_\lambda(s_t, (u^{-j}, a'^j))$;
    Update actor parameters for agent j, $\theta_{t+1}^j = \theta_t^j + \alpha_\theta \nabla_\theta^j \pi_\theta^j(a_t^j|s_t) A^j(s,u)$;
  end
  Each agent updates state $s_t = s_{t+1}$;
end

7.1.5 COMA (Partially Centralized)

Algorithm 5: COMA (Partially Centralized) Algorithm

initialize hyperparameters: γ, α_θ, α_λ α_ϕ, β;
initialize s_0;
initialize λ, ϕ^0, ϕ^1, ..., ϕ^J, θ^0, θ^1, ..., θ^J;
for $t = 0, 1, ..., T$ **do**
  for $j = 0, 1, ..., J$ **do**
    Sample agent j's action from the jth policy: $a_t^j \sim \pi_\theta^j(a_t^j|s_t)$
  end
  Take joint action a_t in state s_t and receive CU SINR reward r_{t+1}^{CU}, D2D SINR reward r_{t+1}^{D2D} and next state s_{t+1} from simulator;
  Calculate CU SINR TD error for critic $\delta_t = r_{t+1} + \gamma(\max Q_\lambda(s_{t+1}, a_t^j) - Q_\lambda(s_t, a_t^j \sim \pi_\theta(a_t^j|s_t)))$;
  Update CU critic parameters $\lambda_{t+1} = \lambda_t + \alpha_\lambda \nabla_\lambda Q_\lambda(s_t, u_t)\delta_t^{CU}$;
  for $j = 0, 1, ..., J$ **do**
    Calculate D2D SINR TD Error for agent j, $\delta_t^{j,D2D} = r_{t+1}^{j,D2D} + \gamma V_\phi^{j,D2D}(s_{t+1}) - V_\phi^{j,D2D}(s_t)$;
    Update D2D critic parameters for agent j, $\phi_{t+1}^j = \phi_t^j + \alpha_\phi \nabla_\phi^j V_\phi^{j,D2D}(s_t)\delta_t^{j,D2D}$;
    Calculate counterfactual baseline for agent j, $A^j(s, u) = Q_\lambda(s_t, u_t) - \sum_{a_t'^j} \pi_\theta^j(a_t'^j|s_t) Q_\lambda(s_t, (u^{-j}, a'^j))$;
    Calculate the mixed TD error for agent j, $\delta_t^j = (1 - \beta)\delta_t^{j,D2D} + \beta A^j(s, u)$;
    Update actor parameters for agent j, $\theta_{t+1}^j = \theta_t^j + \alpha_\theta \nabla_\theta^j \pi_\theta^j(a_t^j|s_t)\delta_t^j$;
  end
  Each agent updates state $s_t = s_{t+1}$;
end

7.2 Source Code

Due to the length of the source code and it spanning many files, the full source code for this thesis can be found at [21] (`https://github.com/Dronie/D2D_A2C/tree/master`)

References

[1] Jayesh Bapu Ahire. *The Artificial Neural Networks handbook: Part 1*. 2018. URL: https://medium.com/coinmonks/the-artificial-neural-networks-handbook-part-1-f9ceb0e376b4 (visited on).

[2] AurelianTactics. *Hands on Reinforcement Learning with Tensorflow TRFL*. 2019. URL: https://github.com/PacktPublishing/Hands-On-Reinforcement-Learning-with-TensorFlow-TRFL/blob/master/Section%203/Actor-Critic.ipynb (visited on).

[3] DeepMind. *DeepMind*. 2020. URL: https://deepmind.com/ (visited on).

[4] Jakob N. Foerster et al. "Counterfactual multi-agent policy gradients". In: *32nd AAAI Conference on Artificial Intelligence, AAAI 2018* (2018), pp. 2974–2982. arXiv: 1705.08926.

[5] Python Software Foundation. *Python 3.6.9 Release Page*. 2019. URL: https://www.python.org/downloads/release/python-369/ (visited on).

[6] Pimmy Gandotra and Rakesh Kumar Jha. "Device-to-Device Communication in Cellular Networks: A Survey". In: *Journal of Network and Computer Applications* 71 (2016), pp. 99–117. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2016.06.004. URL: http://www.sciencedirect.com/science/article/pii/S1084804516301229.

[7] Ian J. Goodfellow et al. "Generative Adversarial Networks". In: (2014), pp. 1–9. arXiv: 1406.2661. URL: http://arxiv.org/abs/1406.2661.

[8] Google Deepmind. *DeepMind TRFL*. 2020. URL: https://github.com/deepmind/trfl (visited on).

[9] Yousef Ali Al-Gumaei et al. "A novel utility function for energy-efficient power control game in cognitive radio networks". In: *PLoS ONE* 10.8 (2015), pp. 1–21. ISSN: 19326203. DOI: 10.1371/journal.pone.0135137.

[10] Yousef Ali Al-Gumaei et al. "Non-cooperative power control game in D2D underlying networks with variant system conditions". In: *Electronics (Switzerland)* 8.10 (2019), pp. 1–15. ISSN: 20799292. DOI: 10.3390/electronics8101113.

[11] Hong Shen Wang and N. Moayeri. "Finite-state Markov channel-a useful model for radio communication channels". In: *IEEE Transactions on Vehicular Technology* 44.1 (1995), pp. 163–171.

[12] Udit Narayana Kar and Debarshi Kumar Sanyal. "An overview of device-to-device communication in cellular networks". In: *ICT Express* 4.4 (Dec. 2018), pp. 203–208. ISSN: 2405-9595. DOI: 10.1016/J.ICTE.2017.08.002. URL: https://www.sciencedirect.com/science/article/pii/S2405959517301467?fbclid=IwAR1t2jPlsygsEfiwqcuWiz4NiNcr-HvZFsYNAtIgSnabGh5nAJhYilAnxk0.

[13] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[14] Michael L. Littman. "Markov games as a framework for multi-agent reinforcement learning". In: (1994).

[15] Jerry Liu. *Why does the policy gradient method have a high variance?* 2018. URL: https://www.quora.com/Why-does-the-policy-gradient-method-have-a-high-variance (visited on 06/05/2020).

[16] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[17] Michael McCloskey and Neal J Cohen. "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem". In: ed. by Gordon H Bower. Vol. 24. Psychology of Learning and Motivation. Academic Press, 1989, pp. 109–165. DOI: https://doi.org/10.1016/S0079-7421(08)60536-8. URL: http://www.sciencedirect.com/science/article/pii/S0079742108605368.

[18] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *33rd International Conference on Machine Learning, ICML 2016* 4 (2016), pp. 2850–2869. arXiv: 1602.01783.

[19] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013). ISSN: 0028-0836. DOI: 10.1038/nature14236. arXiv: 1312.5602.

[20] Jiayu Qiu, Bin Wang, and Changjun Zhou. "Forecasting stock prices with long-short term memory neural network based on attention mechanism". In: *PLoS ONE* 15.1 (2020), pp. 1–15. ISSN: 19326203. DOI: 10.1371/journal.pone.0227222.

[21] Stefan Roesch. *Benchmark Implementation for D2D Project*. 2020. URL: https://github.com/Dronie/D2D_A2C.

[22] Stefan Roesch. *COMA (Partially Centralized) Source Code*. 2020. URL: https://github.com/Dronie/D2D%7B%5C_%7DA2C/blob/master/coma%7B%5C_%7Dpartially%7B%5C_%7Dcentralized.py (visited on).

[23] Stefan Roesch. *COMA Source Code*. 2020. URL: https://github.com/Dronie/D2D%7B%5C_%7DA2C/blob/master/coma%7B%5C_%7Dcentralized.py (visited on).

[24] Stefan Roesch. *Dual Critic (Partially Centralized) Source Code*. 2020. URL: https://github.com/Dronie/D2D%7B%5C_%7DA2C/blob/master/dual%7B%5C_%7Dcritic%7B%5C_%7Dpartially%7B%5C_%7Dcentralized.py (visited on).

[25] Stefan Roesch. *Dual Critic Source Code*. 2020. URL: https://github.com/Dronie/D2D%7B%5C_%7DA2C/blob/master/dual%7B%5C_%7Dcritic.py (visited on).

[26] Stefan Roesch. *IAC Source Code*. 2020. URL: https://github.com/Dronie/D2D%7B%5C_%7DA2C/blob/master/IAC.py (visited on 08/20/2020).

[27] Stefan Roesch. *MHIAC Source Code*. 2020. URL: https://github.com/Dronie/D2D%7B%5C_%7DA2C/blob/master/MHIAC.py (visited on 08/20/2020).

[28] Cem U. Saraydar, Narayan B. Mandayam, and David J. Goodman. "Efficient power control via pricing in wireless data networks". In: *IEEE Transactions on Communications* 50.2 (2002), pp. 291–303. ISSN: 00906778. DOI: 10.1109/26.983324.

[29] L S Shapley. "Stochastic Games". In: 39 (1953), pp. 1095–1100.

[30] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489. ISSN: 14764687. DOI: 10.1038/nature16961. URL: http://dx.doi.org/10.1038/nature16961.

[31] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (2017), pp. 354–359. ISSN: 14764687. DOI: 10.1038/nature24270. URL: http://dx.doi.org/10.1038/nature24270.

[32] Zhenfeng Sun and Mohammad Reza Nakhai. "Channel Selection and Power Control for D2D Communication via Online Reinforcement Learning". In: ().

[33] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. Cambridge, MA: The MIT Press, 2018, p. 526. ISBN: 9780262039246.

[34] Ming Tan. "Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents". In: *Machine Learning Proceedings 1993* (1993), pp. 330–337. DOI: 10.1016/b978-1-55860-307-3.50049-6.

[35] Gerald Tesauro. "TD-Gammon: A Self-Teaching Backgammon Program". In: *Applications of Neural Networks*. Ed. by Alan F Murray. Boston, MA: Springer US, 1995, pp. 267–285. ISBN: 978-1-4757-2379-3. DOI: 10.1007/978-1-4757-2379-3_11. URL: https://doi.org/10.1007/978-1-4757-2379-3%7B%5C_%7D11.

[36] The Matplotlib development team. *Matplotlib*. 2012. URL: https://matplotlib.org/ (visited on).

[37] The NumPy project. *NumPy*. URL: https://www.numpy.org (visited on).

[38] DAVID H. WOLPERT and KAGAN TUMER. "Optimal Payoff Functions for Members of Collectives". In: *Advances in Complex Systems* 04.02n03 (2001), pp. 265–279. ISSN: 0219-5259. DOI: 10.1142/s0219525901000188.

[39] Chris Yoon. *Deriving Policy Gradients and Implementing REINFORCE*. 2018. URL: https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63 (visited on).

[40] Chris Yoon. *Understanding Actor Critic Methods and A2C*. 2019. URL: https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f (visited on).

[41] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. "Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms". In: (2019), pp. 1–72. arXiv: 1911.10635. URL: http://arxiv.org/abs/1911.10635.