Deep Multi-Agent Reinforcement Learning for Path Planning

Amrinder Singh Rai

MSc Data Science
The University of Bath
September 2020

Signed:

Deep Multi-Agent Reinforcement Learning for Path Planning

submitted by

Amrinder Singh Rai

for the degree of MSc Data Science of the

University of Bath

September 2020

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the author unless otherwise specified below, in accordance with the University of Bath's policy on intellectual property (see http://www.bath.ac.uk/ordinances/22.pdf). This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of MSc Data Science in the Department of Computer Science. No portion of the work in this thesis has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signature of Author .....

Abstract

In this thesis we will look at a multi-path planning problem, where we propose a co-operative multi-agent reinforcement learning perspective. We consider co-operation through shared utilities and communication via learnt meta-protocols. To guide us, we look at a routing problem, widely addressed in PCB design, and resulting in an interesting, but intractable state space problem.

We first develop our problem scope, by designing and developing our multi-agent environment. We then implement DRQN, leveraging Deep Neural Networks to deal with the large state space and a Recurrent Neural Network to model dependencies between sequential state observations. We show that DRQN outperforms DQN by a large margin in our problem domain, particularly in partially observable settings.

We then distribute our single-agent DRQN solution within a multi-agent setting, considering the unique challenges that arise. We first look at the concept of shared utility and information sharing as a means to co-operate. We then go onto look at explicit, but adaptive communication via Reinforced Inter-Agent Learning (RIAL), a proposed framework for learning meta-protocols. We find relations in the inner workings of our single-agent DRQN solution and the RIAL framework, as internalising intent and then communicating it.

Our research shows some promises for this approach, however many challenges arise due to the complexity and sparse nature of our problem domain. Thus, with this end-to-end project we hope to bring forth the promises and challenges within multi-agent reinforcement learning, while directing research attention into this area, which has real world potential, such as in PCB design routing. We finally finish with a discussion of key challenges, drawing profound conclusions and a proposal for future works.

Acknowledgements

Many thanks to Özgür Şimşek for taking me on for this project and for inspiring me to do so with her fantastic Intro to RL module. Also a huge thanks to Chithrupa and Erik from Zuken Inc. for sharing their expertise and experiences, further encouraging me with this project.

Contents

1 Introduction **10**
1.1 Motivation 10
1.2 Problem Domain 11
1.3 Structure 12

2 Literature Review **14**
2.1 Introduction 14
2.1.1 Planning 14
2.1.2 Reinforcement Learning 16
2.2 Reinforcement Learning and Planning 18
2.2.1 Planning through Resampling 18
2.2.2 Persistence with RNNs 19
2.3 Co-operative MARL 20
2.3.1 Multi-agent Systems 20
2.3.2 Distributed Q Learning 21
2.3.3 Limitations and potential solutions 22
2.4 Learning to Communicate in Co-operative MARL 23
2.4.1 Message Passing 23
2.4.2 Reinforced Inter-Agent Learning (RIAL) 25
2.4.3 Differentiable Inter-Agent Learning (DIAL) 26
2.5 Conclusion 27
2.5.1 Human-level perspective 27
2.5.2 Uses in Path Planning and scope for future research 28

3 Environment Design and Implementation **30**
3.1 State Representation 30
3.1.1 High-level view 30
3.1.2 Agent-level view 31
3.1.3 Partial Observability 32
3.2 Logic and Rewards Design 33
3.2.1 Taking Steps 33
3.2.2 Termination conditions 33
3.2.3 Rewards scheme 35
3.3 Board Generation 36

4 Agent Development **37**
4.1 Single Agent 37
4.1.1 DQN 38
4.1.2 DRQN 39
4.2 Muli-Agent 40
4.2.1 Distributed DRQN 40
4.2.2 Shared Experience Replay 40
4.2.3 Reinforced Inter-Agent Learning (RIAL) 41

5 Results and Analysis **43**
5.1 Methodology Overview 43
5.2 Single-Agent 44
5.3 Multi-Agent 47
5.3.1 Two Agents 47
5.3.2 Three Agents 50

6 Conclusion **53**
6.1 Contributions 53
6.2 Limitations 54
6.3 Future Works 54

A Ethics Checklist **59**

B Source Code **60**
B.1 Environment 60
B.1.1 board.py 61
B.1.2 connector.py 62
B.1.3 pcb_env.py 64
B.2 Agents 71
B.2.1 DRQN with RIAL 71
B.2.2 MARL training 78
B.2.3 Shared Buffer Replay (DRQN) 81

Chapter 1

Introduction

1.1 Motivation

Systems or agents, are increasingly being delegated tasks in order to make their own decisions and as these systems become more interconnected into larger distributed networks, they begin to interact with one-another [Wooldridge, 2009]. Therefore it is important to develop systems that can intelligently interact and co-operate in order to solve larger global objectives.

We will look at such interactions, and the emergence of co-operative planning behaviour, through communication within a multi-agent path planning problem. Particularly, communication which can be learnt through reinforcement, Reinforced Inter-Agent Learning (RIAL), a proposed framework [Foerster et al., 2016]. The proposed solution would learn meta-protocols allowing for adaptive communication, and thus co-ordination, in highly dynamic and complex environments.

Such a solution has many use cases, with such systems becoming more important in a coming age of IoT and self driving cars. The specific problem we propose in the next section (1-1) for instance, would be very useful in the automatic design of complex electronics, like PCB circuit boards.

We hope to bring forth a discussion of how such multi-agent systems can emerge, through the methodology of first developing a single-agent solution and then distributing it. By doing so we hope to draw links between the internal learnings of such an agent and those of which it communicates to other agents, adaptively.

1.2 Problem Domain

Printed Circuit Boards (PCBs) have revolutionised electronics, allowing for efficient production, in cost, time and manufacturing. However the boards themselves require careful designing, which is still cumbersome today, despite software aids. The design process can be broken down into several key stages, requiring domain expertise, one of which is routing.

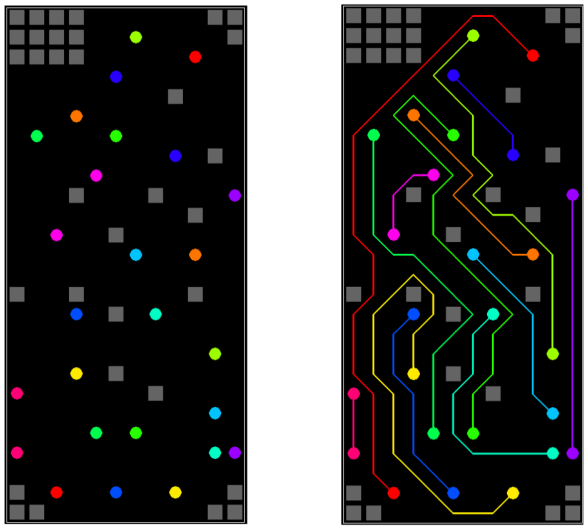

Figure 1-1: 'real-env' environment; a multi-agent path planning problem (Şimşek & Zuken inc., 2020)

The above routing problem has been proposed in figure 1-1. This problem constitutes to a discrete multi-path planning environment. In a MARL setup each connection is viewed as an agent, shown by colour, which must reach their respective destinations with the shortest path, without crossing the paths of other agents. In a multi-agent setting such a problem requires complex co-operative coordination which we hope can be learnt.

Essentially by taking a single agent perspective on each connection, where they jointly co-operate, they approximate a global ground truth for effective PCB design. This is easier said than done as the joint state-action space in a multi-agent set-up already exacerbates a difficult to navigate environment in the single agent setup.

Assuming a 15x10 grid with 9 possible actions (ways to move), while disregarding obstacles, along with the different lengths and orientations a connection may have in a given time step, we have an incredibly large state-action space of 9^{150}, making it infeasible to experience most of them, even while leveraging powerful function approximators, such as Deep Neural Networks. There is also an unavoidable sparsity issue faced in such a problem, which only further hinders the agents learning. This sparsity problem will be a key issue when discovering and applying our solutions.

1.3 Structure

Our thesis is structured as follows:

1. **Introduction**
   The motivation for our chosen topic and an introduction to our specific problem domain.

2. **Literature Review**
   A review of planning and reinforcement Learning, while relating these two concepts together with a look at experience replay buffers and recurrent neural networks. We then consider the multi-agent setup and the challenges that emerge. We look at possible solutions, before looking at two proposed meta-protocol frameworks, RIAL and DIAL.

3. **Environment Design and Implementation**
   A look at our environment design, including the state representation and logic, which sets out the problem to be solved. We also discuss other details, such as board generation.

4. **Agent Development**
   A look at the agents developed for this project, both single-agent and multi-agent.

5. **Results and Analysis**
   An outline of our experimentation methodology. We then present our experimentation results and analyse them.

6. **Conclusion**
   A conclusion of contributions, limitations and avenues for future work.

Chapter 2

Literature Review

2.1 Introduction

In this literature review we will explore the interconnected components of a Multi-Agent Reinforcement Learning (MARL) problem for co-operative path planning. We will look at Reinforcement Learning for Planning, the Co-operative MARL setup and its limitations, before discussing communication learning amongst agents to overcome them, before finally concluding and relating these ideas, in order to motivate future research.

But first we will have a background review of Planning and then of Reinforcement Learning. We will do so from the perspective of an autonomous agent capable of independent actions.

2.1.1 Planning

Planning has been a long standing sub-area of AI. It is a process that involves finding a sequence of actions from a current state description, in order to achieve a given goal state. This is achieved through the generation of representations for future behaviour [Wolfram et al., 2013].

In figure 2-1, two states are shown, a start and an end goal. In planning the state representations of these two states belong to a domain, along with representations of effects and preconditions of available actions in the environment. State representations can involve anything from visual or logical, to discrete or continuous, and which result in a state space which captures all possible situations

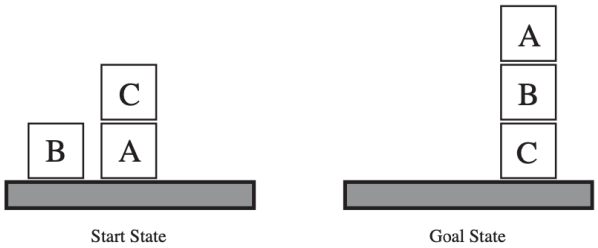

Figure 2-1: A classic planning domain problem, Block world [Russell and Norvig, 2016].

that may arise [LaVelle, 2006]. A planning algorithm can then find a sequence of actions, through these future representation over time, to reach the final goal state.

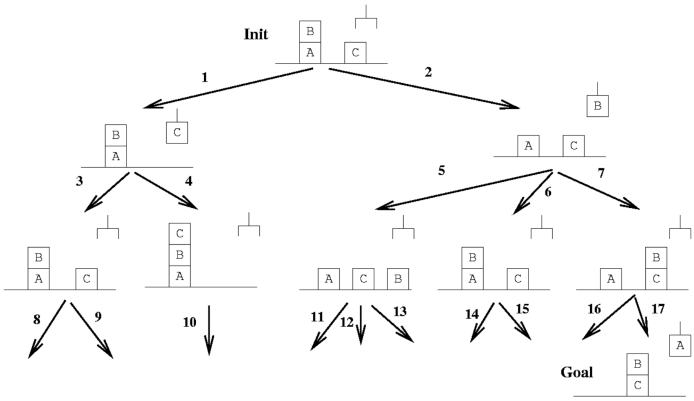

Figure 2-2: A brute force search over the domain of a planning problem, where every future representation is enumerated until we reach a goal state. [Kerber, 2004]

In classical planning algorithms this requires a state search [Russell and Norvig, 2016], the most primitive of which is a brute force search (Figure 2-2). There are many classical planning algorithms, from using heuristics to logical induction, in order to find optimal sequence of actions. However these are typically only for static environments and correspond to deterministic methods for planning.

On the other hand, control theory has additionally developed feedback policies which enable adaptive responses during planning execution [LaVelle, 2006]. This brings us onto an important adaptive control framework, Reinforcement Learning.

2.1.2 Reinforcement Learning

Reinforcement Learning is also a goal orientated approach, however instead of having a domain of effects and pre-conditions for actions, we now learn through feedback from taking available actions. In this section we will primarily focus on dynamic programming and the model-free approach of the Q-learning algorithm, in order to gain an intuition for this framework.

The Reinforcement Learning control framework can be interpreted as a Markov Decision Process (MDP), where an agent at discrete time steps t and in state S_t interacts with an environment with action A_t, to then observe a new state S_{t+1} of the environment along with a reward signal R_{t+1} [Sutton and Barto, 2018].

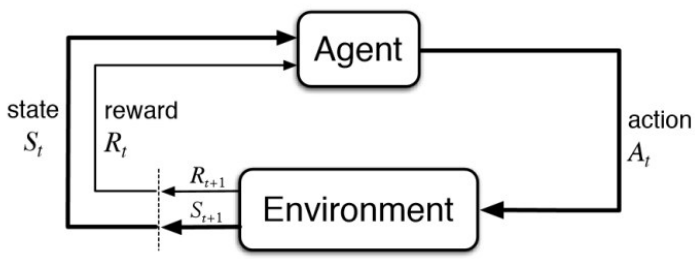

Figure 2-3: The Markov Decision Process for Reinforcement Learning [Sutton and Barto, 2018].

The agent learns from every action in order to make better decisions in the future, with the feedback signal that it receives. The action A_t is chosen by the agent from S_t using a state-action mapping, this is known as the policy:

$$\pi(s) = a$$

This state-action mapping in value-based methods constitutes to a distribution of 'values' determined by a value function:

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\}$$

Where E_{π} is the expected value from following policy π, and R_t is the cumulative rewards (returned) following time step t, and can be viewed as the weighted sum of future rewards:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{n-t} r_n = r_t + \gamma R_{t+1}$$

Where γ is the discount factor, a weightage on future rewards which typically decreases as future returns become less certain with time steps t.

From an algorithmic perspective, we can use dynamic programming in order to define and calculate the value for state-action pairs via the Bellman Equation [Sutton and Barto, 2018]:

$$Q(s,a) = r + \gamma max_{a'} Q(s',a')$$

We see this as a Q update in the Q-learning algorithm:

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

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

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

Figure 2-4: The Q learning algorithm. [Sutton and Barto, 2018]

The policy may incorporate an ϵ-greedy strategy which is an approach to tackling the exploration-exploitation problem in Reinforcement Learning [Sutton and Barto, 2018].

There are many other Reinforcement Learning algorithms and families of them. We will not touch upon these, however we will look at some extensions of Q-learning in the next Section 2.2 within the context of planning. We refer the inclined reader to Sutton & Barto (1998/2018) for further Reinforcement Learning approaches.

Function approximation is also an important aspect within Reinforcement Learning as they allow us to generalise over large state spaces. Deep Learning has particularly become a popular and powerful function approximator that has allowed researchers to work with traditionally difficult to approximate state spaces [Arulkumaran et al., 2017]. In this case Deep Neural Networks, constituting to a Q-net, approximate the bellman equation for every possible action from a particular state observation.

2.2 Reinforcement Learning and Planning

In this section we will look at where planning appears in Reinforcement Learning, combining what we have discussed in the previous section. We look at two concepts; resampling of experiences and the use of hidden variables in RNN based methods.

2.2.1 Planning through Resampling

Resampling is a commonly used method within modern statistics. This involves repeatedly sampling from a training set and refitting a model on each sample in order to obtain additional information about the fitted model [Gareth et al., 2013]. In Reinforcement Learning we often see this through the resampling of experiences.

The use of an experience replay buffers in particularly has become a norm in many RL algorithms [Zhang and Sutton, 2017]. This involves storing experiences into a buffer and then replaying them in order to generate variance and facilitate learning. Sufficient samples within a buffer can be seen as a model, or an approximation of the environment which can facilitate planning [Hamrick, 2020], treating samples as future representations as defined by the classical planning problem in 2.1. This can also be seen seen with tabular Dyna-Q [Sutton and Barto, 2018], where an explicit model used for planning, essentially performs experience replay during the planning phase.

Such replay buffers also allow for greater sample efficiency during learning, and thus faster convergence. However, since we are using rough sample estimates, it is important to factor in the idea that our model may be outdated, thus a recency weighting may be introduced when using samples to update our value function [Sutton and Barto, 2018]. There are also alternative reinforcement learning models for planning, which try to solve this sampling problem with prioritised sweeping and trajectory sampling; the inclined reader may refer to Sutton Barto. (2016), Chapter 8.

2.2.2 Persistence with RNNs

We have briefly mentioned the popular use of Deep Neural Networks, in order to estimate Q-values and policies, within complex state spaces. However, such approximations at each time step t, may be rather naive without further context. In other words, a typical feed-forward architecture assumes observations to be iid distributed, ignoring its context within the sequence of observations. This is particularly important when we are dealing with POMDP settings. Deep Recurrent Neural Network architectures have been proposed to help in such scenarios where information can persist from one observation to another [Hochreiter and Schmidhuber, 1997].

One such RL proposal, which leverages the use of RNNs, has been made, being DRQN [Hausknecht and Stone, 2015]. This is the DRQN, an extension of the DQN algorithm which incorporates an LSTM recurrent neural network between the state input (After feature any extractions) and the fully-connected layer. This LSTM network keeps a hidden variable throughout an episode, persisting from time step to time step. This hidden variable is learnt along with the optimisation process, and learns to pertain useful information about previously

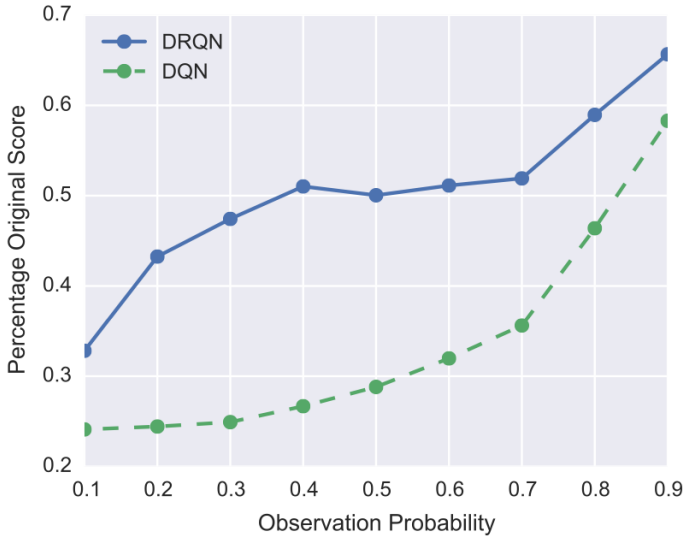

Figure 2-5: DRQN's performance scales as a function of observability. Scores are averages over 9 games. [Hausknecht and Stone, 2015]

Such context information allows us to better model the environment, and can thus facilitate planning. This is of even more importance when we are working in a partially observable state space. DRQN has been successfully applied in partially observable Atari games [Hausknecht and Stone, 2015]. It has also been shown that DRQN performance scales as a function of observability, as shown in figure 2-5 above. DRQN also better adapts during run time if the quality of observations itself were to change or have uncertainty [Hausknecht and Stone, 2015].

The ability for us to essentially weight our approximations at each time step, with this hidden variable acting as a prior, makes the DRQN fall within a Bayesian framework. Context matters, and with it the agent is able to get a sense of its trajectory, and thus able to plan better. We also encourage the view of such a hidden state, as a representation of internal intent or even an internal messaging protocol. These perspectives of internalising intent or prior knowledge, will give us a holistic view for when we look at the passing of such information to other agents in section 2.4.

2.3 Co-operative MARL

We have so far looked at solving problems with a single agent design. We will now move this discussion in order to study the interactions of multiple such agents at a macro-level. We specifically focus on the co-operative MARL setting, where agents may interact and learn through reinforcements, in order to reach a global objective.

2.3.1 Multi-agent Systems

Systems or agents, are increasingly being delegated tasks in order to make their own judgements and decisions [Wooldridge, 2009]. As these agents become more interconnected into larger distributed networks, they begin to interact with one-another. The study of such multi-agent systems becomes very important as these autonomous entities often have [Shoham and Brown, 2009]:

- Diverging information
- Diverging interests

In the co-operative setting these system of agents try to solve a global objective, while also having their own objectives or interests, which can often diverge from one another. These interests may be obvious in competitive settings, however is also very true within the co-operative setting, where agents may not share the same perspective as their peer agents due to diverging information.

Diverging information poses a fundamental cooperative challenge as different agents may have collated a different model or perspective of the environment. This differing of information has also been referred to as differences in 'spheres of influence' [Wooldridge, 2009], which can be constantly changing in a dynamic environment. This is further complicated when agents themselves are non-stationary due to actions they take according to their world view. Within the context of Deep RL and experience replay buffers, past experiences are rendered obsolete in such scenarios [Papoudakis et al., 2019].

Therefore it can be useful to model other agents in a co-operative setting, hence it may be convenient to have some form of information sharing. This direct interaction between agents can be in the intuitive form of a communication channel where messages may be passed as actions and received as part of the state representation. We will discuss such a form of communication further in Section 4.

For now we will introduce Multi-Agent Reinforcement Learning by looking at a distributed setting of Q learning.

2.3.2 Distributed Q Learning

We can introduce distributed dynamic programming to meet global constraints amongst multiple agents [Shoham and Brown, 2009]. In this case optimal action selection in a multi-agent MDP framework would be characterised as a vector of local actions $(a_1, \ldots, a_n)$ constituting to a global action of a [Shoham and Brown, 2009]. However the range of actions from a range of agents produces an exponential growth in the joint state-action space. This state space can become difficult to deal with, even with deep learning. We will further discuss potential frameworks to deal with such large state-action spaces in the next section 2.3.3.

However, a simply distributed MDP framework may work when the problem is less complex or when there are few agents. In figure 2-6 below we observe Dis-

tributed DQN learning, a distributed and deep learning variant of the Q learning algorithm that we've seen in 2.1.2.

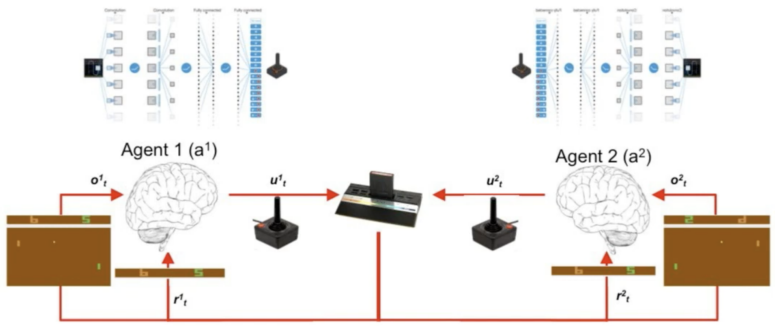

Figure 2-6: Distributed DQN for co-operative Atari pong. [Tampuu et al., 2015]

A distributed strategy can also be seen from the perspective of utility, where rational actions fits with many solution concepts, such as dominant or Nash equilibrium strategies [Wooldridge, 2009]. In the case of Figure 2-6, here we see the shared utilities amongst the DQN agents are the rewards received, which are shared amongst the agents, thus agents model each other via the environment and the value function.

Although this can be viewed as an implicit contract of co-operation in order to keep diverging interests in check, diverging information still remains a problem. This is largely due to a large joint state-action space that typically results in a multi-agent setup. In order to scale to more complex environments with more agents, information sharing or at least some kind of abstractions are required. This will be the topic of discussion for the next Section 2.3.3.

2.3.3 Limitations and potential solutions

As we have briefly discussed a multi-agent system can have an exponentially large joint state-action space, which can be extremely difficult to generalise over. Apart from centralised training, a common technique used in MARL (which we'll see in 2.4.2), another layer of information or abstraction can help us deal with these intractability issues while providing greater learning and interpretability to our models.

In recent years one such framework has developed, Hierarchical Reinforcement

Learning, which looks to decompose problems in a hierarchy of subproblems or subtasks. For instance, the options framework, which uses temporal abstractions in order to learn higher level actions over 'primitive actions' [Sutton et al., 1999].

Multi-agent systems may be linked by some organisational relationship [Wooldridge, 2009]. Learning such relationships has also been seen within the hierarchical framework, through the proposal of Feudal Reinforcement Learning [Dayan and Hinton, 1993].

As we have discussed so far, it can be useful to essentially abstract information in a system in order to, not only to help solve the intractable state-action space problem, but to also tame large amounts of diverging information within a system, which we discussed in 2.3.1. Approaches such as hierarchical learning, may be seen as being part of a large subfield of AI research, which is meta learning. Meta learning tries to understand how learning can become flexible across a domain of problem domains, to improve performance of existing algorithms or to learn the learning algorithm itself [Jurgen, 1987].

The limitations we have discussed and also solutions are still very current and ongoing research areas. In the next section we will discuss another meta learning framework, one where we define a meta-protocol in order to learn protocols.

2.4 Learning to Communicate in Co-operative MARL

In this section we will look at communication in a co-operative MARL setting. We will first go over the brief notion of communication as message passing. We will then look at two proposed approaches for end-to-end learning of protocols in complex environments, RIAL & DIAL [Foerster et al., 2016].

2.4.1 Message Passing

Some interesting problems can be solved in distributed systems with message-passing algorithms [McKay, 2004]. As an example, we will focus on the problem of finding lowest cost paths within a distributed system, using the generalised Viterbi algorithm.

We consider the case in figure 2-7, where we must find the shortest path between A and B in the network of nodes. This can be achieved with A passing its distance from itself, which is 0, which is then sent to the corresponding nodes.

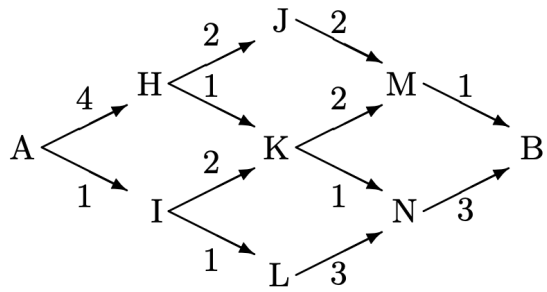

Figure 2-7: A route diagram from A to B, showing associated costs across edges. [McKay, 2004]

The corresponding nodes H and I then add the message 0 from A with the distance from A, producing 4 and 1 respectively. This then continues where nodes which receive more than one message, only establish a connection (or protocol) with the node who's message produced the overall lowest sum total. This is repeated until we reach our destination B, and we are left with the graph in figure 2-8. From B we then do a backward pass and count the total across established links in order to deduce the shortest path and cost, in this case A-I-K-M-B with cost 6. This is also known as the Min-sum message passing algorithm.

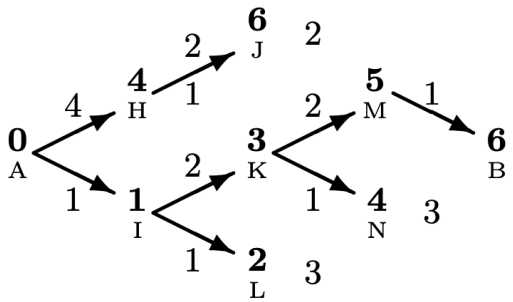

Figure 2-8: Min–sum message passing algorithm applied to our route diagram in Figure 2-7. [McKay, 2004]

We have just seen how simple communication within a network can help solve larger global objectives, such as finding the shortest path. There are also many other message passing algorithms and they have many use cases. In the multi-agent set up message passing can be interpreted as passing intent [Shoham and Brown, 2009].

This idea of intent passing also fits within a Bayesian Framework of passing 'prior knowledge'and such networks would relate to that of Bayesian Belief networks which use Bayesian Belief Propagation, an algorithm that is built upon Viterbi [McKay, 2004]. This passing of intent may also fit into the framework of planning seen in section 2.1.1 [Wolfram et al., 2013] where a message may be viewed as a form of future representation for the environment in order to facilitate planning behaviour.

Such messages in a multi-agent setup, may be internalised within an agent, as discussed in section 2.2.2 with DRQN. We will now look at adaptive protocols in order to communicate such information in the next two sections, in order to solve global objectives.

2.4.2 Reinforced Inter-Agent Learning (RIAL)

RIAL is the most straightforward proposal, which can be viewed as distributed DQN, that we saw previously in figure 2-6, with an additional communication layer which is independent of the environment shown in figure 2-9 below.

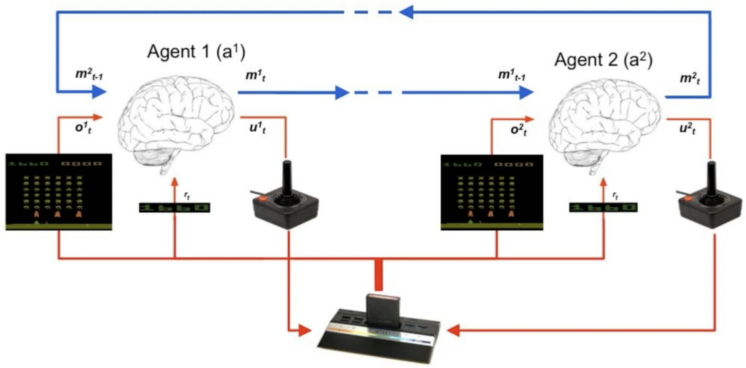

Figure 2-9: Distributed DQN with message passing. [Foerster et al., 2016]

Here, the act of sending a message itself is an action and no strict protocol needs to be defined. For instance the message could just be a random ping. This effectively implements a 'meta-protocol' to learn a protocol through reinforcements. Thus, an agent may choose to send a message and the other agent would observe this as part of the environment state representation, to then perform action selection with respect to this. Through reinforcement from shared reward

signals the agents would learn a protocol to communicate and solve the global objective.

Despite the success of RIAL, it can still be quite slow to learn these protocols in complex problems. It may be more efficient to learn model parameters through feedback, or backward passes like we have seen with Viterbi in section 2.4.1. This now brings us onto Differentiable Inter-Agent Learning, or DIAL.

2.4.3 Differentiable Inter-Agent Learning (DIAL)

DIAL extends the RIAL framework we have seen, to leverage centralised learning in order to share parameters and push gradients as feedback to senders. This essentially means that recipients can now send feedback to the sender based on the TD-error they received as a result of chosen actions, in accordance to the received message. This allows the sender to adjust its own parameters in the direction of the given gradient, to learn a better protocol.

As intuition for the reader familiar with neural network back-propagation, we recommend viewing the multi-agent system as a dynamic neural network where we back propagate on forward passed messages in order to find gradients at a given node (agent) in order to optimise the (agent's) parameters according to some global objective (loss function). It is important to note that learning only happens like this during centralised learning where communication actions are replaced with direct connections with 'continuous messaging' across edges, just like in typical neural networks.

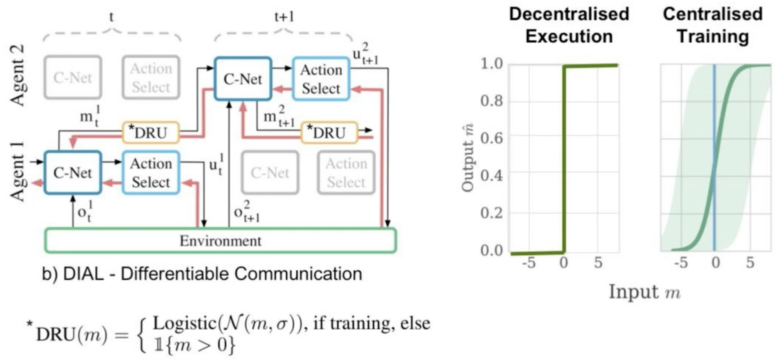

Figure 2-10: Differentiable Inter-Agent Learning (DIAL). C-Net corresponds to the policy network of the agent. [Foerster et al., 2016]

Figure 2-10 above, shows a diagram of this 'Differentiable Communication' (b), where the DRU is the 'Discretise/Regularise Unit'. This DRU regularises messages during centralised training via a logistic function with added Gaussian noise in order to make learnt continuous protocols easy to discretise later by the DRU, during decentralised execution.

This makes the DIAL setup end-to-end trainable across agents [Foerster et al., 2016]. Experiments have shown DIAL to be much faster in learning protocols, than the trial-and-error based RIAL [Foerster et al., 2016]. It also scales better to large discrete message spaces and problems [Foerster et al., 2016].

2.5 Conclusion

In this literature review we have seen the use of reinforcement learning for adaptive control and how this relates with classical planning concepts. We then saw where we tend to see such planning concepts in RL, whether through the use of experience replay or a hidden variable that relates sequential states, as in the case of DRQN.

We then moved onto a distributed setting, where diverging information and interests became a challenge. We looked at the concept of shared utility, primarily through the reward function, while then discussing frameworks for dealing with the large state space and diverging information. In particular we looked at two frameworks for learning meta-protocols, RIAL and DIAL.

We also related the ideas of message passing, as intent passing, with the 'internalisation' of intent from the single-agent DRQN perspective. We also related such notions of intent with the Bayesian framework, which deals with 'prior' knowledge to weight decisions, or intent as some future representation as defined in classical planning.

2.5.1 Human-level perspective

"There has been a movement away from machine-oriented views of programming toward concepts and metaphors that more closely reflect the way we ourselves understand the world" [Wooldridge, 2009].

Henceforth, we feel it is appropriate to provide a human-level perspective on

some of the concepts discussed in this literature review.

Particularly as Reinforcement Learning may motivate us with a neurological view of learning, which has been key for the development of primitive survival instincts in us [Eisenegger et al., 2014], but it is language that has unfolded to become considered a source of all our greater achievements [Deutscher, 2006]. Such languages have allowed us to share global future representations, allowing us to co-operate and plan in order to achieve grander global objectives.

Figure 2-11: Meta-protocols can allow us to acheive great heights.
'The Tower of Babel' (Abel Grimmer/Wikimedia)

However, language itself has been in temporal change with its environment, which you may appreciate when pondering on classical text or when you feel the need to adapt your language when attending a formal event. The meta-protocols we've seen in section 2.4 respect this natural semantic nature of language, allowing our agents to adapt and learn new protocols in dynamic and changing environments, as we have done and still do.

2.5.2 Uses in Path Planning and scope for future research

Such intuitive concepts of learnt and adaptive communication, apply to many important multi-path planning problems. For instance in autonomous vehicles or PCB routing, where an agent may appropriately adapt its protocol depending on the situation it is in, and appropriately adjust its trajectory based on the messages it receives from other agents. This may also allow for greater scalability as such protocols could adapt to the number of agents in view and the subsequent amount of messages being received.

We are essentially leveraging Deep Neural Networks for approximating adaptive messages, as we did for action selection. However the same key issues emerge, exploration of the action and protocol spaces, due to the sparse nature of such problems, along with the large state spaces.

Nonetheless we believe this is a promising framework for dealing with such co-operative MARL settings, and a good entry point for MARL research, to appreciate the frontier challenges that it presents.

Chapter 3

Environment Design and Implementation

In this chapter we walk through the design and implementation of our environment, thus defining our problem scope.

We will first look at the state representation of our developed environment, and the resulting state space. We'll then discuss our reward design, along with the logic, where we'll compare different setups. Finally we will look at board generation, allowing for experimentation and flexibility within our problem scope.

Specific implementation details of the environment can be found in the appendix Appendix B.

3.1 State Representation

3.1.1 High-level view

We develop a grid environment of size $height \times width$, resulting in a set of squares X. There are also n connectors, along with m obstacles.

Each connector has an $id \in \{0, ..., n\}$, along with a start and end goal. Grid blocks associated with each connector are rendered with a particular colour as shown in figure 3-1 above. In this paper we will label the start and end blocks with S and G respectively when convenient.

Each agent is able to move from the start as shown, where squares that have

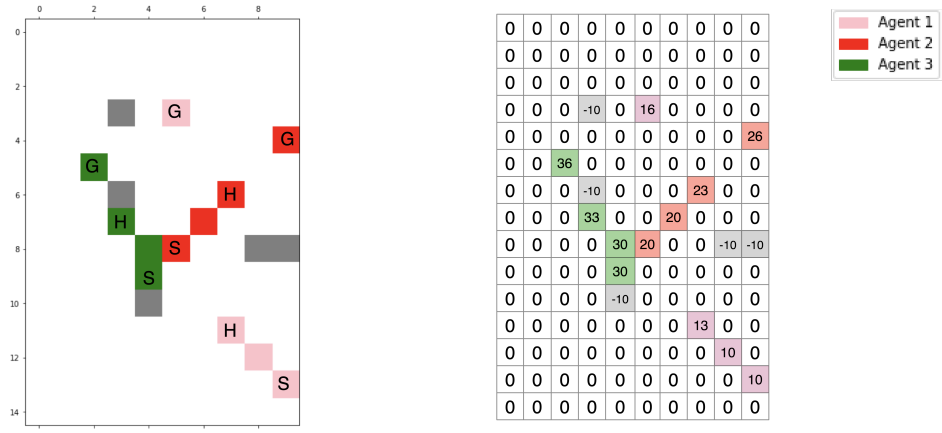

Figure 3-1: High-level rendering (left) and the corresponding Agent-level state representation (right), of a 15x10 environment grid setup.

been occupied by that agent are rendered appropriately with associated colours. The agent can move in any of eight directions at a given time step t.

3.1.2 Agent-level view

?? At each time step t, the agent itself receives a copy of the grid as a single channel grid. The head of an agent n is represented as $n3$ and its corresponding goal represented as $n6$, as shown in figure 3-1. Any trailing squares that have been traversed are represented with $n0$.

Any empty spaces are represented by a 0, while obstacles are represented with -10.

A three channel RGB representation was also tried, however provided negligible difference in performance. We also stuck with the current representation as we felt this was easily interpretable for debugging later during solution developments. It was also important to differentiate the head of the agent from the rest of the 'body' in order for the agent to get a sense of its current position and trajectory.

The agent also receives its euclidean distance from the goal to its head, along with the grid state mentioned. This information would be appropriate and easily available in most domain problems, such as PCB routing. This additional information gives the agent a significant amount of information about its state from the goal, without explicitly telling the agent how to react to it. This sequential

information at each time step, is particularly useful given a partially observable setting.

3.1.3 Partial Observability

Our developed environment supports partial observability, where an observation size may be passed into the environment. Odd observation dimension sizes are preferred, as otherwise the observation space will be slightly biased to the right of the agent (with the agent shifted left of the centre).

Padding has also been incorporated for when the observation may be off the grid, an example of which is shown in figure 3-2 below. The off grid areas were padded using the same representation for obstacles (-10), which proved effective for training.

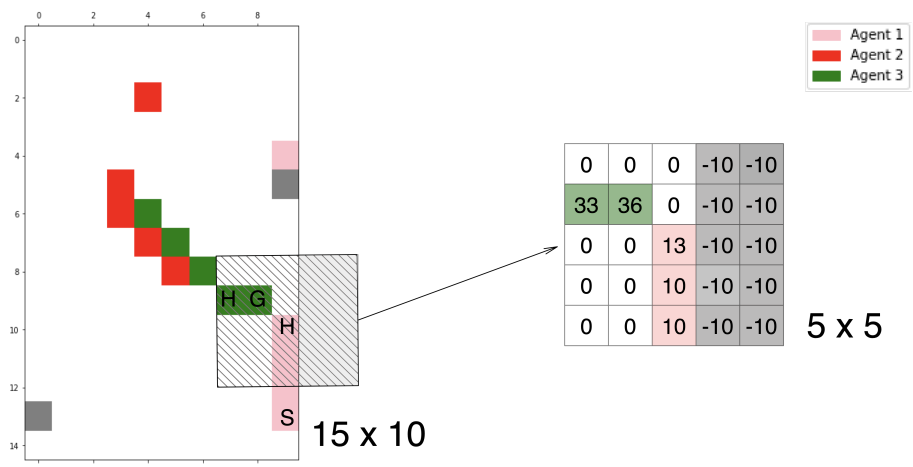

Figure 3-2: The state representation received by Agent 1 at some timestep t, in a 5x5 partially observable setting.

Incorporating partial observability was crucial in order to allow for the development of scalable solutions. This is due to the fact that in most real world application it may be inefficient or infeasible to provide all information. In the context of PCB routing, we would want our solution to generalise for a variety of situations, such as board sizes and number of agents.

3.2 Logic and Rewards Design

In this section we will discuss the core logic of our environment and our reward design, determining how our agent interacts and learns.

3.2.1 Taking Steps

Our environment takes an action for each time-step. It accommodates interaction with a multi-agent MDP, allowing a vector of local actions $(a_1, ..., a_n)$ constituting to a global action a, to be passed in at each time-step [Shoham and Brown, 2009]. This concurrent action selection introduces non-stationarity into our problem, requiring greater co-ordination amongst the agents [Papoudakis et al., 2019].

Sequential action selection was also considered, however we thought this did not represent a real-world scenario well and posed a less interesting problem which requires less co-operation. Sequential action selection would be infeasible to scale for many real world problems. Taking our PCB routing problem, a typical board problem within our multi-agent framework may have hundreds of agents at any given time, thus n agents would have to wait $n \times t$ time-steps before selecting an action. This could also introduce biases towards certain agents due to the ordering of action selections.

When agents terminate, appropriately their actions no longer needs to be passed into the environment, which reduces the joint action space and problem complexity.

3.2.2 Termination conditions

When designing our environment logic, conditions to terminate on were particularly important, where terminating too soon or late can hinder learning. There is an obvious positive termination for reaching the goal, however this is a sparse termination condition, which is often not met during the initial episodes of training. We will mainly focus on negative termination conditions.

We tried an environment which contained a sole negative termination condition for an agent, for getting into a situation where there was no longer free squares to move into. This involved a dynamically changing action space on each time-step, in order to avoid any form of collision. In this setup, the Q agents also

required masking out excluded actions from the policy. The justification for this was to simplify the task of the agent, to one of simply routing for the shortest path, however this turned out to be counter-intuitive, or at least for our state representation.

An avoidant policy was learnt with our single-agent baseline (DQN), where we consistently observed a trend of growing episode lengths during training, as shown in figure 3-3 below.

Figure 3-3: Episode lengths during training (left) and our resulting policy (at t=34) for an unseen grid (right).

For this reason we incorporated collisions and associating terminations as part of our environment problem. We also hypothesise that obstacle detection, or a negative association with obstacles (via negative experience), may facilitate in planning, and in the multi-agent scenario such information could potentially be signalled to other agents (via messaging).

Thus, our environment terminations on the following conditions:

+ Reaching the goal state.
- Going off grid.
- Collision with obstacles.
- Collision with agents (including itself and invalid diagonal crossings).

In the MARL setting only the particular agent meeting the above conditions is terminated (or removed) from the episode.

These termination conditions are also accompanied with specific reward signals, which we will now discuss in the next section.

3.2.3 Rewards scheme

The reward scheme is also a critical component of RL, as it provides feedback to the agent on its current policy. A key challenge, particularly in our problem domain, is the issue of sparse feedback. We outline our Rewards scheme below in figure 3-4.

Reward	Situations
-1	For every move on each timestep.
20	For reaching the goal state.
-10	Any collision or going off grid (as described in section 3.2.2).

Figure 3-4: Our chosen rewards and the corresponding situations in which the agent would receive them.

Firstly we introduce a -1 reward, which is not accompanied by termination. This is to negatively incentivising traversing across the grid, and enforce the learning of the shortest path solutions.

We also initially tried offsetting this negative reward with the change in euclidian distance to the goal (thus positive for getting closer, negative for getting further away). However, this introduced a high bias towards the goal, whereas it was fairer to include the distance as part of the state representation for the agent to make its own decision based on it. This addition also allows for better design, as the agent is able to reward itself (intrinsic) independent of the environment.

Thus, we also found the importance of having a simple reward scheme in order for the agent to approximate long term returns with stability. We did not find a great deal of difference in changing the reward magnitudes in our experiments initially. However this becomes an obvious problem when needing to scale your solution as in large grid sizes (than 15x10), as agents may incentivised to negatively terminate if accumulated negative rewards by traversing to the goal outweighs those received by ending an episode through collision. In this latter

case, rewards scaling is required [van Hasselt et al., 2016].

In the MARL setup these rewards are shared amongst participating agents at a given time-step, although this approach does pose challenges as we'll see in results chapter 5.

3.3 Board Generation

Defining how to generate our boards was also important, not only to set the difficulty of the objective, but also to help facilitate learning. It was important to allow for flexibility in generating boards and setting some ground truth, in order to allow for experimentation and generalisation.

We allow for the generation of manual boards by the user, allowing for the number of blocks to be changed as a parameter. This is to allow for flexibility, as the number of blocks would contribute to the state space complexity.

We also randomly sample boards on each reset (or episode), which facilitates learning and allows for generalisation. However, complete random sampling of blocks, along with the start and end goals, proved to be difficult in facilitating learning, particularly given our sparse rewards setting. Thus, we included the ability to use other distributions such as a bi-variate binomial, rather than a bi-variate uniform by default, which introduces some ground truth with respect to obstacle detection and avoidance. We also experimented with an annealing deviation for distributions during training runs.

We also allowed for board sizes to change during training, as gradual progression in the board size during training can help in producing more general and scalable solutions, having a curriculum learning effect on our agents [Narvekar et al., 2020].

We also allow for start and end goals to be fixed, while the blocks were randomly generated, which we found useful for quick prototyping and testing of solutions. With a similar notion we also split our grid into sections where we could sample such endpoints within them, this is depicted in figure 5-1 and sets a foundation for our methodology in results chapter 5.

Chapter 4

Agent Development

In this section we will develop our agents to interact with our implemented environment from the previous chapter 3.

We then go onto distribute our single agent solution within a multi-agent setup, ultimately developing our own implementation of RIAL from section 2.4.2, for our given environment.

All solutions were developed using PyTorch and specific implementation details, including parameters, can be found for our latter solution in appendix B, while all other code is available in our GitHub repository mentioned.

4.1 Single Agent

In this section we will develop two agents, as a basis to distribute them in the MARL setting. Each agent contains a Q-value based RL policy as seen in section 2.1.2 , and we look at two particular Q-network architectures, the latter of which incorporates RNNs discussed in section 2.2.2.

We have chosen Q-value based methods, due to being an intuitive entry point in RL, thus appropriate to explain and interpret our novel experimentation in the MARL setting. They also tend to be sample efficient and have stable performance. We also do this in order to allow for TD-error gradients to be easily passed in the future, when implementing the DIAL framework from section 2.4.3. We also incorporate a greedy-epsilon policy for all agents, which is annealed by a certain degree over time-steps.

4.1.1 DQN

Our agent receives the state S_t from the environment on each time step as a tuple, containing an observation of the grid and the euclidian distance from its goal, as outlined in section **??**. Our agent transforms this state observation into S^*_t, by extracting features from the state observation and then concatenating the result (unrolled) with the distance into a 1d tensor, for input into our feed-forward Q-network. This feed-forward network then approximates Q values for each action within the action space. This process is depicted in figure 4-1 below. For extracting features from our grid observation we use a single convolution, due

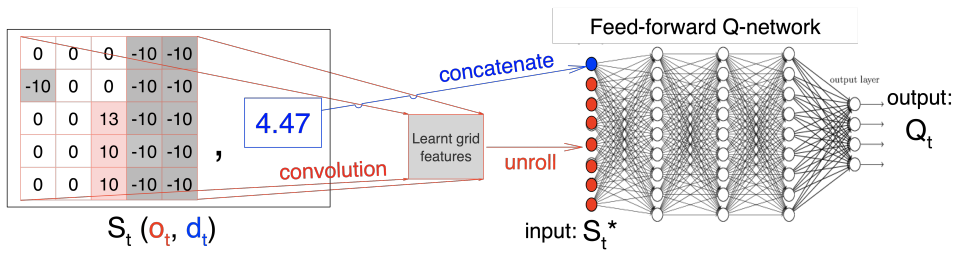

Figure 4-1: Our DQN producing Q-values for a given state tuple at some time-step t.

to the relatively small size of the observations and as our state representation is relatively feature rich and easy to interpret. The resulting Q-values drive our policy and action selection, through the process outlined in section 2.1.2.

Our agent also take in various replay buffer objects for off-policy batch sampling. Each replay buffer stores experiences and then re-sample them in order to update our Q-network as discussed in section 2.2.1.

Since we are using a Deep-Q network, we found the use of a Q-target network in our implementation, which copies parameters from the main Q-network within a specified interval, increased stability considerably and thus the learning performance.

Though replay buffers have the added benefit of providing uncorrelated samples of past experiences [Sutton and Barto, 2018], we find many benefits of correlating experiences in sequence, which brings us onto DRQN.

4.1.2 DRQN

Our DRQN follows much of the same architecture as DQN, although there is now an intermediate LSTM step between input S^*_t and the output of Q-values, as shown in figure 4-2 below.

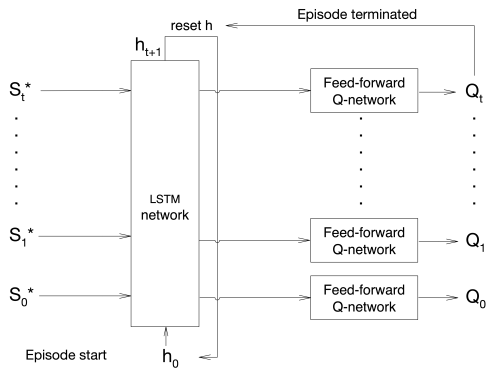

Figure 4-2: Our DRQN producing Q-values at each time-step t for a given episode.

Our replay buffers also needed to be modified as we now sample entire episodes rather than individual experiences. This then further required us to pad episodic experiences, in order to account for different episode lengths experienced, during batch updates. The padding was subsequently disregarded by the LSTM network, allowing for us to update our Q-network appropriately.

The hidden state variable for our DRQN is stored as a property of the agent, which is reset after each episodic replay during training. During execution and action selection, this hidden state is stored within the agent as a property until it is reset to an empty representation value upon termination.

The addition of the LSTM, and thus persistence via the hidden state variable, provided great improvements over DQN, where the agent is able better model the environment and its own trajectory, particularly in partially observable settings, as we will observe in our results section 5.2.

4.2 Muli-Agent

4.2.1 Distributed DRQN

In our distributed DRQN setup, we simply distribute our DRQN solution, as discussed in section 2.3.2 and depicted in figure 2-6.

Based on the co-operative concept of shared utility, we share rewards by taking the mean of accumulated rewards received at a given time-step and share this amongst non-terminated (or active) agents.

Experience buffer replay is also switched off in our implementation by simply setting the buffer length to one, as past experiences are rendered obsolete in such a setting, particularly for non-stationary settings as discussed in 2.3.1. However we did consider implementing our own replay buffer which is shared across agents, discussed in the next section 4.2.2.

4.2.2 Shared Experience Replay

With diverging perspectives and the concept of shared utility in mind, we implemented a novel shared buffer replay. This is a replay buffer where all agents store their experiences, however when sampling from this same buffer, the agent switches positions within the state grid observation (if not in the observation, then simply replaces it) with the agent that stored that observation. This process is depicted in figure 4-3 below.

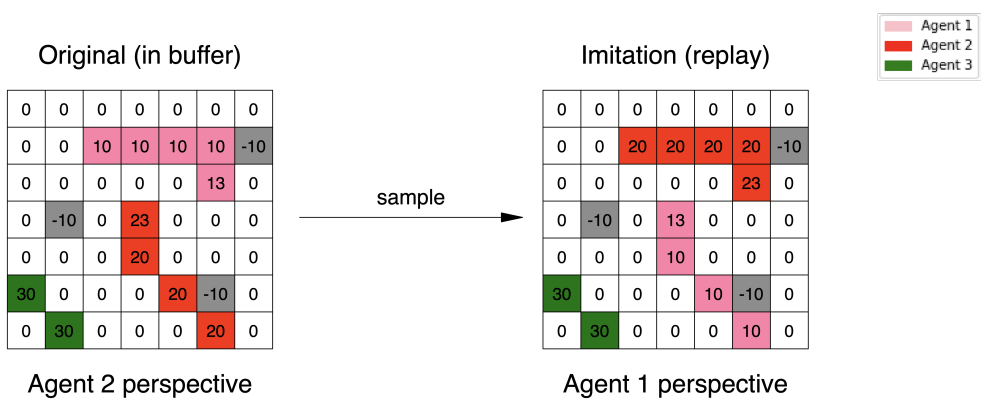

Figure 4-3: Agent 1 sampling an Agent 2 experience from our shared buffer.

This also relates to centralised training and is a form of implicit parameter sharing, common practices in MARL as we've discussed. We may also look at this replay buffer from the concept of implicit communication or imitation learning, a key sub-area in AI and RL research [Hussein et al., 2017].

By experimenting with such a buffer, we hoped to see greater sample efficiency while learning a generalised co-operative solution to our problem environment.

4.2.3 Reinforced Inter-Agent Learning (RIAL)

We also extend our DRQN agent from section 4.1.2 in order to support the RIAL framework discussed in section 2.4.2. Hence, now at each time-step t we approximate a policy action and policy message. This process is shown in figure 4-4 below.

1-bit	2-bit	3-bit
0	00	000
1	01	001
-	10	010
-	11	011
-	-	100
-	-	101
-	-	110
-	-	111

Table 4.1: Message space for three different channel sizes.

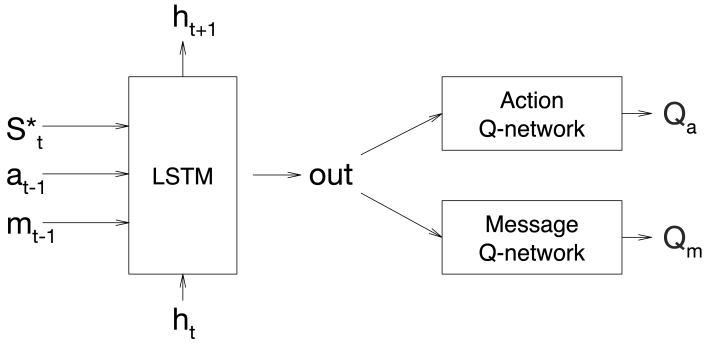

Figure 4-4: Our RIAL net at timestep t

Rather than just passing the transformed state S^*_t as in DRQN (figure 4-2), we now also pass in the previous messages received, as well as the action. Thus, our RIAL agent also learns to model other agents via the LSTM, as well as itself. The previous action is also passed in as this has shown to improve performance considerably [Foerster et al., 2016].

The resulting output from the LSTM is then passed into two separate feed-forward Q-networks, one for approximating Q-values for possible, and the other for possible messages. While the former output size is equivalent to the action

space (8), the latter output size depends on the possible number of messages, and thus is dependent on the communication channel size set.

Also shown in table 4.1 above is the message-space and all possible messages for three communication sizes. All chosen messages by each agent are passed as a 2d-tensor to every agent in the corresponding time-step. Each agent takes in these messages along with the environment state at each time-step.

This RIAL network is hence optimised in order to approximate optimal action values and message values, and uses reward feedback and resulting TD-errors to do so.

Chapter 5

Results and Analysis

5.1 Methodology Overview

In this section we will briefly discuss our methodology behind our experimentation, important for receiving sound results and to aid reproducability.

Firstly, we stick with a 15x10 sized board as a reasonable size for experimentation. Due to the intractability or long training times still experienced when sampling all object positions uniformly, we divide the board into sub-sections where we may sample either the end or start (or both), in each sub-section. We do this in a way to ensure meaningful difficulty, with the appropriate amount of variance, and yet provide a feasible state space for experimentation. In figure 5-1 below we show the number of ways we split the board.

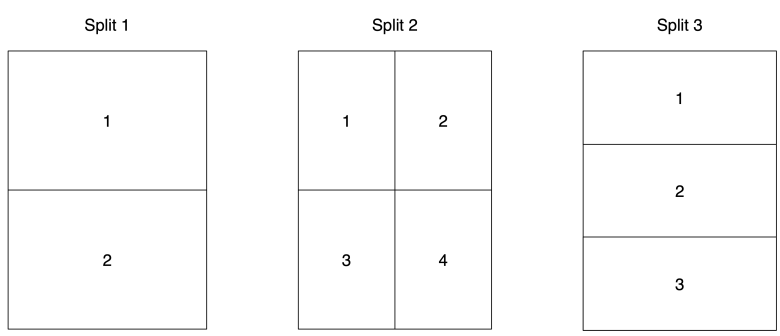

Figure 5-1: Our 3 ways of splitting the board.

We also uniformly sample a specific number of obstacles in order to add an appropriate amount of variance to our problem. During training, we also keep all parameters (batch size, learning rate, epochs, epsilon, convolutions etc.) the same for comparative runs. We produce the corresponding learning curves, with epsilon values indicating explorative phases during learning. We also smooth these learning curves by a consistent factor, while also highlighting the greater variance of the curves in the background.

We then simulate our resulting policies on 10,000 episodes, with the same sampling distribution and we produce accuracies as proportions of how many of the agents finished (full, partial or no solution).

5.2 Single-Agent

In this section we go onto compare our single-agent solutions developed in section 4.1, DQN and DRQN.

We initially trained our agents with full observations of the grid, but we found our solutions struggled to converge at a policy. However, when moving to partial observations, such as 15x15, convergence to a policy was observed. This was initially a surprise, however considering our state representation, it is likely that the agent is quickly able to localise itself, due to it always being centered in each partial observation.

For this reason we start our experimentation with a reasonable 15x15 partial observation for a 15x10 grid. We reduce this size by odd numbers and eventually witness non-convergence for 7x7 with DQN, as shown in the two charts within figure 5-2 below.

On the other hand, convergence to a policy occurs with DRQN within even smaller partial observations such as 5x5, as shown in figure 5-3. Convergence times with DRQN also tend to increase more gradually, as can be seen within the same figure 5-3. This is likely due to the DRQN better modelling the environment during training thanks to the persistent hidden state, where as the DQN agent tends to suddenly fit a policy as shown in figure 5-2.

These results were as expected and we see similar results to those presented in the original DRQN paper [Hausknecht and Stone, 2015]. Particularly we can see that figure 5-4 below resembles closely to the scaling relationship presented

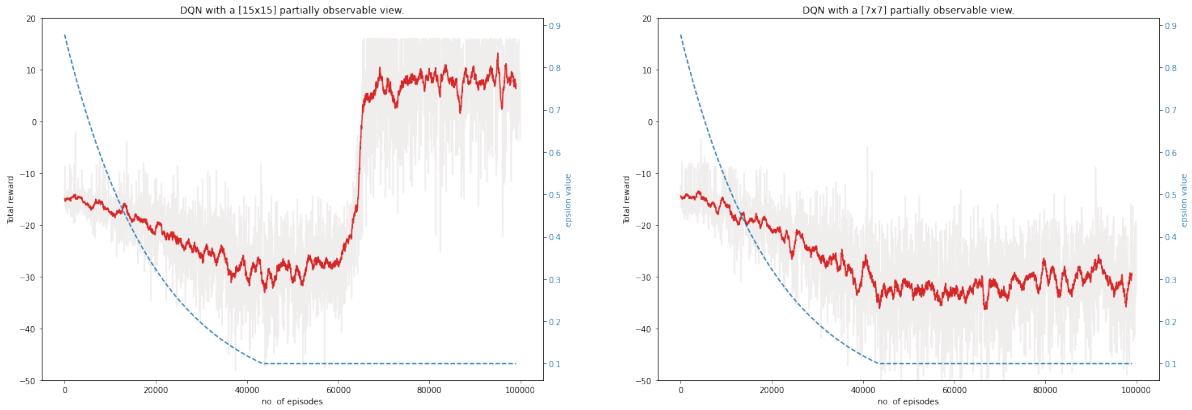

Figure 5-2: DQN with observation sizes 15x15 (left) and 7x7 (right).

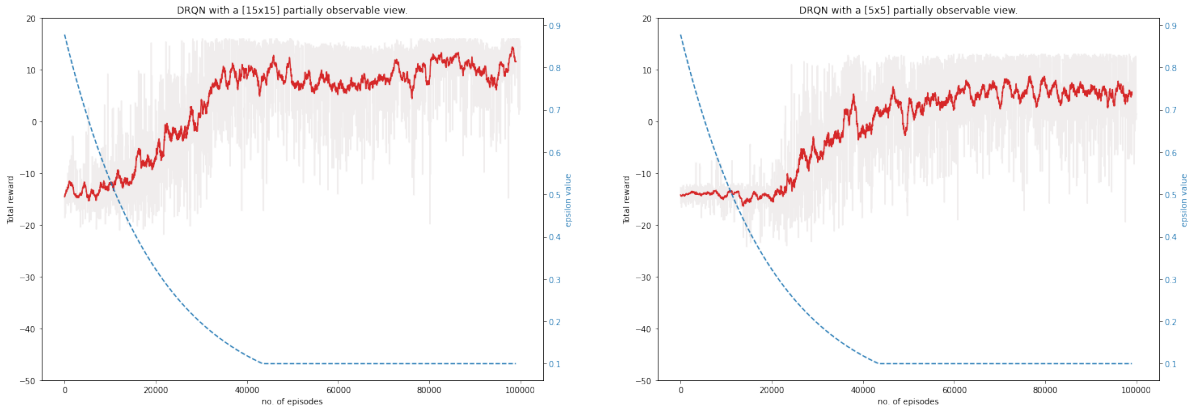

Figure 5-3: DRQN with observation sizes 15x15 (left) and 5x5 (right).

in figure 2-5 in the literature review. Although due to our particular problem, we witness a large drop due to non-convergence to an appropriate policy. In our problem it was more a matter of convergence or not, and we can see that as observation size decreases, convergence time tends to increase, shown in figure 5-6, where we present our results in full.

We can see clearly that DRQN performs considerably well, and maintains performance much better than DQN with reduced observability. Again, this is due to the DRQN modelling the environment better by better utilising the distance, over time-steps, in order to get a sense of its own trajectory. We start seeing more considerable drops in performance after 3x3 observation sizes with DRQN, as its

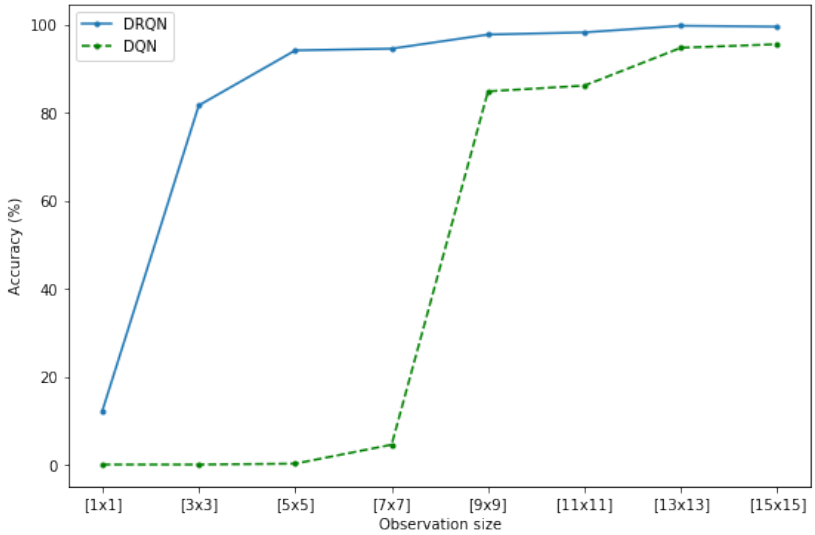

Figure 5-5: Accuracies received for each observation size, DQN and DRQN. The scaling relationship presented is similar to those from the DRQN paper, figure 2-5 [Hausknecht and Stone, 2015].

obs size	Accuracy	Convergence (approx.)
[15x15]	95.6%	62000 epochs
[13x13]	94.8%	64000 epochs
[11x11]	86.2%	78000 epochs
[9x9]	84.9%	87000 epochs
[7x7]	4.6%	-
[5x5]	0.3%	-
[3x3]	0.1%	-
[1x1]	0.1%	-

obs size	Accuracy	Convergence (approx.)
[15x15]	99.6%	41000 epochs
[13x13]	99.8%	41000 epochs
[11x11]	98.3%	42000 epochs
[9x9]	97.8%	42000 epochs
[7x7]	97.6%	44000 epochs
[5x5]	94.2%	45000 epochs
[3x3]	81.7%	51000 epochs
[1x1]	12.3%	-

Figure 5-6: DQN (left) vs DRQN (right) performances over different partial observabilities.

limited view hinders its ability to plan ahead, all the way until 1x1 where the agent essentially has no view of the grid and does not converge to an appropriate policy.

We suspect that given the DRQNs performance in very partially observable situations and its ability to quickly learn a policy to model its environment well, we suspect it would scale well given we increase the grid size gradually during

training, as mentioned in section 3.3. This is not something we tested, and we leave this beyond the scope of this project, in order to focus on scaling our DRQN to a multi-agent scenario.

5.3 Multi-Agent

In this section we look at distributing our DRQN solution into a multi-agent problem. We conduct all experiments with a fixed and reasonable partial observation size of 5x5, according to the results we received in the previous section 5.2. We present our results, while discussing the difficulties in obtaining and interpreting them.

We also fix the communication channel size to 2-bits, as this is where we saw any meaningful results. This is likely due to a 1-bit channel not sufficing to represent complex enough communication in this relatively complex environment. A 3-bit channel likely increased the joint complexity of the messages, thus hindering learning.

We also faced convergence issues and memory/resource problems, which we will discuss further in limitations section 6.2. This was a key challenge, given experience replay was switched off as in the original paper [Foerster et al., 2016] requiring longer training runs.

We also considered our implementation of a shared replay buffer from section 4.2.2, in order to help with sample efficiency but did not find this useful. This buffer can only be feasible and useful when all agents are generalising over all situations, something that we still found to be intractable. Thus, we continue to follow our methodology laid on in figure 5-1, of splitting the state space and turning experience replay off.

With all this in mind, we finally go onto our results for two-agent and three-agent scenarios.

5.3.1 Two Agents

For our two agent experimentation we show results for a simple setup, which also requires co-operative behaviour to be learnt amongst the agents. We use the splitting setup 2 shown in figure 5-1, where we sample the start and goal for

agent-1 from the bottom left and top right. Agent-2 start and goal co-ordinates are sampled from the top left and bottom right, this is further depicted below in figure 5-7 .

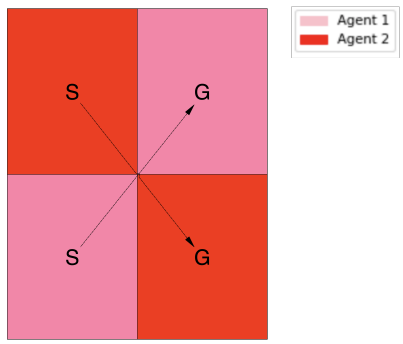

Figure 5-7: Our subsampling setup, for agent 1 and 2. We also depict the general trajectory required for each agent to traverse (arrows).

We thus run training for our 2-agent setup for 500,000+ episodes, for both DRQN with RIAL and distributed DRQN as a baseline. We observe the following learning curves shown in figure 5-8 below.

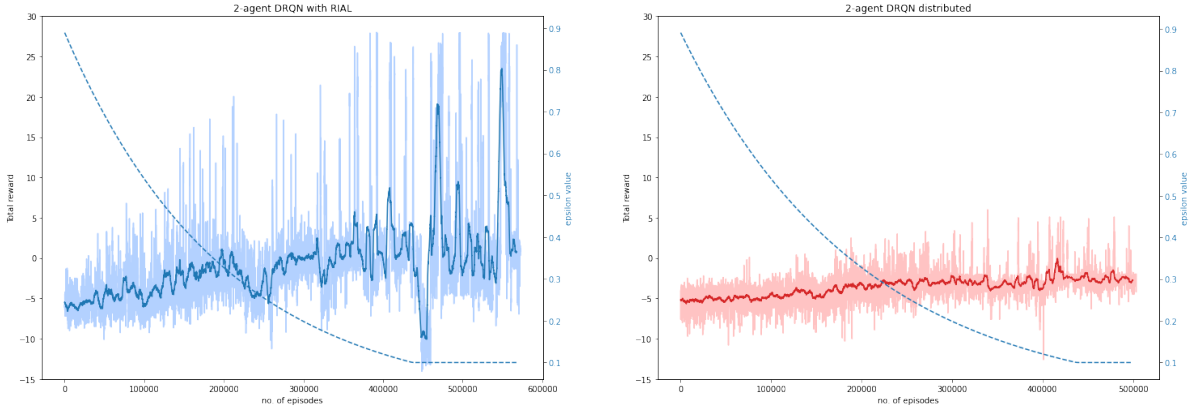

Figure 5-8: Learning curves for RIAL (left) and dist. DRQN (right) for 50,000+ episodes.

We observe higher variance, or unstable learning with RIAL, which is likely due to the more complex state space presented with the addition of messages. However, the learning curves themselves are hard to interpret with multiple agents, as

we collate all shared rewards. Our learning curves are thus highly biased or offset by subsequent remaining agents whom receive a greater share of rewards due to less participating agents.

There are two further caveats, an agent would ideally be incentivised to allow another agent to finish before it, in order to collate higher rewards for itself in the long-run due to not needing to share the final reward (of 20). However, this is a complex policy which is likely never discovered. In the short term, or at least in the most immediate of experiences, it is likely an agent will learn greedy behaviours in order to reap short term gains, risking the possibility of a (shared) negative outcome along the way. This is also due to our current reward design, where the terminated agents don't in the most part receive any consequences for their previously taken actions. This explains why we see a large proportion of our simulations ending with just one agent, as shown in our results in figure 5-9 below. This puts into question simply sharing rewards in such a sparse grid planning problem.

0/2	1/2	2/2
773	7328	1899
7.7%	73.3%	19%

0/2	1/2	2/2
2737	6721	542
27.4%	67.2%	5.4%

Figure 5-9: RIAL (left) vs Dist. DRQN (right); proportion of different test solutions obtained (10,000 epochs).

Despite this, in a fair proportion of cases we see both agents finish, with a significant improvement over our baseline. This can be seen in an exceptional example below in figure . We can only assume that the agent in this particular scenario has been able to model the other agents intent due to messages observed during run-time, and took a different trajectory from the start, in order to contribute to what was an optimal joint-policy.

However this is just one of few ideal examples that we observed during our test simulation. It may be possible that had we been able to run our experiments for longer, our agents may have been able to generalise better and display more of such ideal joint-policies.

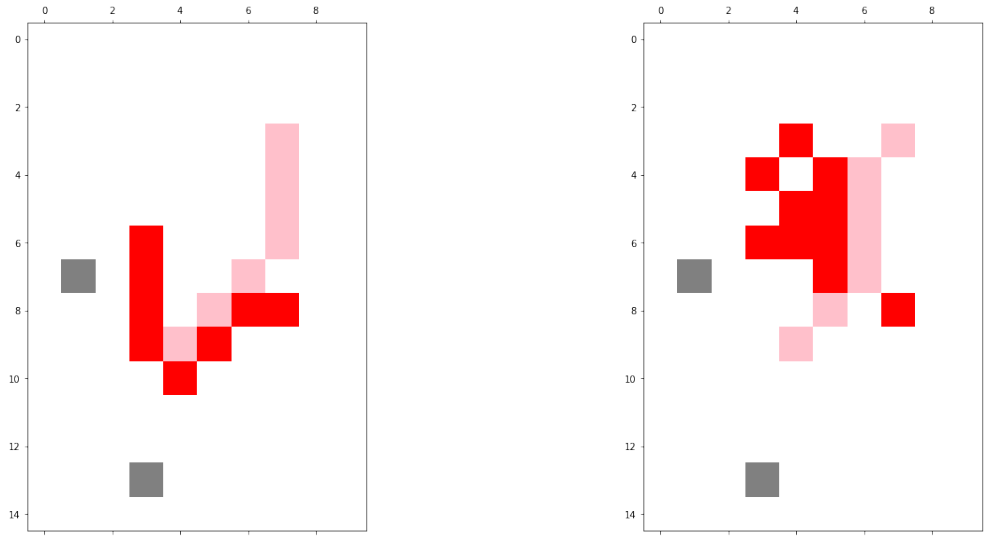

Figure 5-10: RIAL (left) vs Dist. DRQN (right), for a particular grid problem.

5.3.2 Three Agents

We also attempted to scale to a three agent setup, causing many difficulties as the joint state space is only further complicated. We nonetheless found a feasible setup to gain insightful results. For our experimentation here we sample using splitting setup 3 shown in figure 5-1, where the start of agent-1 is sampled from the bottom third and its goal from the top third, while agent-2 and agent-3 both have their endpoints sampled from the top two thirds.

We observed the following learning curves during training, shown in figure 5-11 below, again only further distorted due to the extra number of agents and complexity of the problem.

Further on, looking at the types of solutions we got for this in figure 5-12 below , we can see that the majority are partial 1-agent solutions, which furthers our point that in most cases agents are incentivised to be greedy and thus uncooperative, despite the shared rewards. However, we do observe a significant proportion of 2-agent partial solutions, and this is of course not independent of our chosen sampling methodology. From our observations practically all these 2-agent grid solutions were between agent-2 and agent-3 whom sample both their end points within the same grid region.

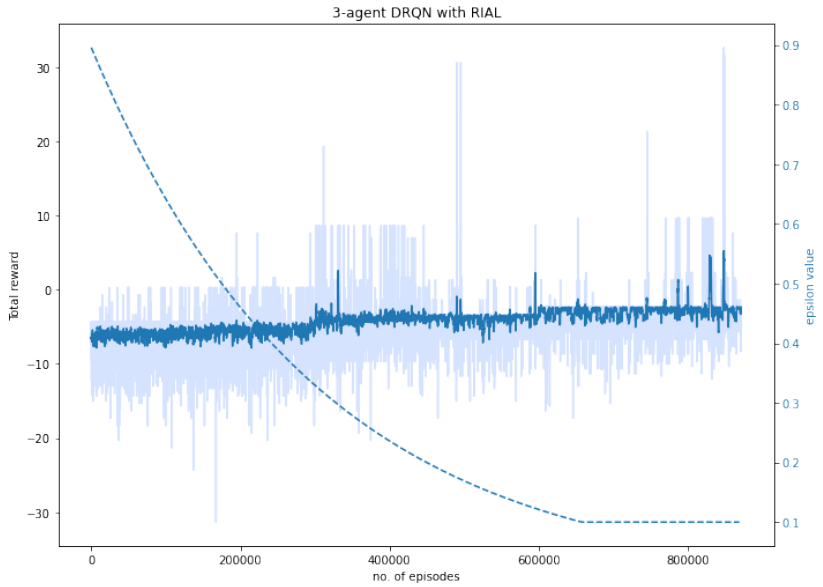

Figure 5-11: Our 3-agent RIAL learning curve.

0/3	1/3	2/3	3/3
1472	5782	2430	316
14.7%	57.8%	24.3%	3.16%

Figure 5-12: Proportion of different test solutions obtained with 3-agent RIAL (10,000 epochs).

This is what we will refer to as a 'coupling' effect, due to their close proximity during training, these two agents tend to interact with each-other more, hence are more likely to learn to co-operate amongst each-other, if at all. Due to agent-1's further distance from its end goals, it requires greater exploration than the other agents, and perhaps greater co-operation, which is not facilitated. For this reason in most episodes we observe that agent-1 tends to end the episode with a collision, which can be observed in the first two examples of figure 5-13 below.

This 'coupling' effect may be further observed by the history of Mean Squared Errors (MSE) for each agent, which is proportional to an agent's regret [Stoyan, 1987], where we can observe the latter two agents following learning and policy improvement much closer together, despite all rewards being shared equally, at least initially.

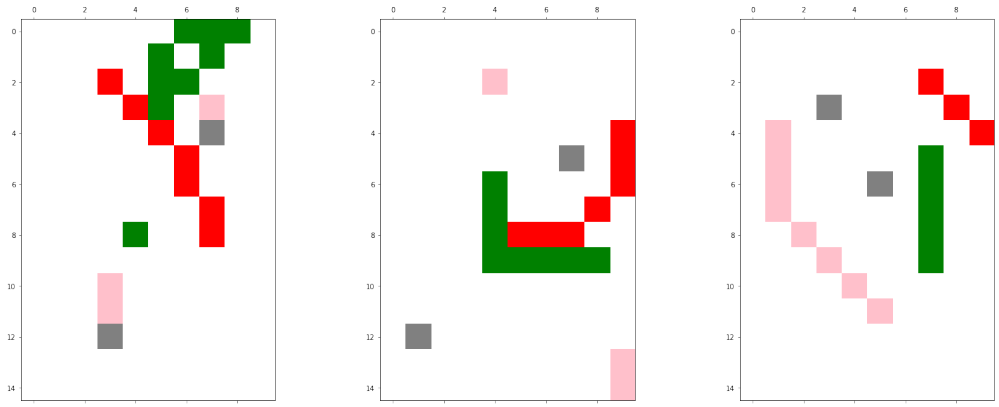

Figure 5-13: Examples of 1/3 and 2/3 partial solutions from our testing, along with 3/3 full solution (though rare).

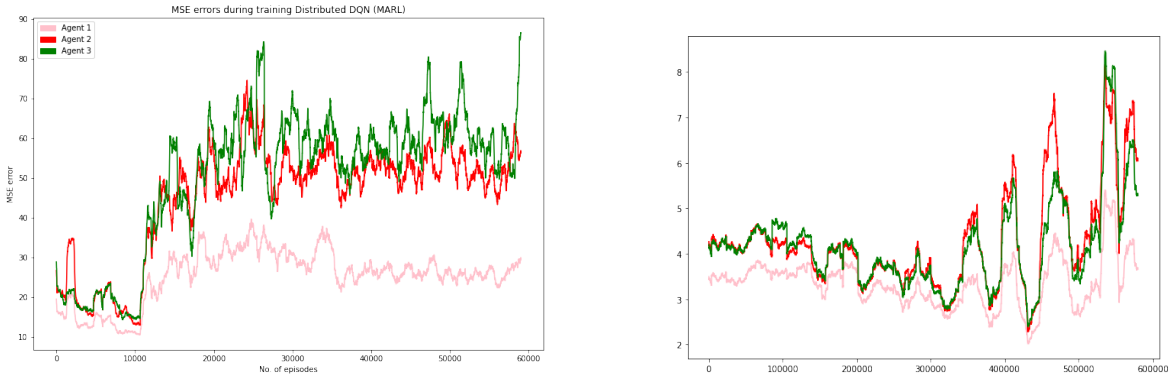

Figure 5-14: The changes in MSE error over episodes that we observed during training. We note that the MSE of agent 2 and 3 follow closely together.

This effect has profound implications in the context of our PCB routing problem, since we may need some connections to traverse long ranges towards respective end points and at some time-steps along the way require closer co-operation with particular agents. Thus, there are different levels of cooperation and a need for both exploring over long distances and closer co-operation between particular agents may be required. Therefore temporal decomposition of this problem may be useful, as discussed in section 6.2 with hierarchical abstraction and learning.

Chapter 6

Conclusion

In this final chapter we conclude our project, considering our achievements and contributions, along with the limitations. Finally we discuss future works and areas for further research.

6.1 Contributions

This was a comprehensive end-to-end project, from the design and implementation of the environment and agents, to running experiments and producing/interpreting results. All works mentioned in this thesis, apart from the use of standard libraries, were also implemented from scratch. Thus, we hope our works would provide a basis for new researchers to get an intuitive understanding for MARL, through our methodology of developing and then distributing a single-agent solution. We also hope researchers would find our environment useful for MARL experimentation, inspired by a real-world and interesting problem domain, PCB routing.

Through our single agent solution we have shown that DRQN can successfully be applied to routing in our partially observed problem environment. We conceptualised the use of the hidden variable to model sequential states, as a realisation of an agents own intent or trajectory. We then go on to relate the proposed RIAL framework as similarly using sequential messages as a means to model the intent of other agents.

Thus, we implemented our own RIAL agent and performed MARL experimen-

tations, which showed some improvements over our distributed DRQN 'baseline'. In doing so we have learnt much about the dynamics of such problems and the many challenges faced.

6.2 Limitations

However our MARL experimentation results leave much to be desired, both in scope and in-depth analysis of what has already been implemented.

A lot of our problems stem from practical issues such as hardware and time limitations. CUDA out of memory issues were a common occurrence, and meaningful runs can take days to run. Even more so, our partial observability of our RL models in the cloud during run-time, gave us an appreciation of all the hype surrounding DAGs and dedicated ML platforms.

Other than practical issues, we were perhaps severely limited by our problem at hand. The large state space and sparse rewards of our environment have resulted in multiple challenges that need to be addressed. This includes model interpretability which became a clear issue in section 5.3.

Nonetheless, these limitations leave further room for improvement in future works, as a promising directions have been shown.

6.3 Future Works

There are many obvious improvements that can be made to our RIAL solution, that were also presented in the original paper [Foerster et al., 2016]. This includes parameter sharing amongst agents for faster training and learning. It may have also been appropriate to incorporate multi-step rewards, where we would provide rewards to an agent after some given time interval, thus rewarding them for cooperative behaviour rather than having been in the 'right place at the right time' in our sparse problem.

And of course, future works could also focus on an implementation of DIAL, which has shown considerable performance advantages over RIAL [Foerster et al., 2016].

It may also be useful to consider other meta-learning frameworks, as discussed in section 5.3.2, where temporal decomposition of our problem could help, with the learning of hierarchies.

However, we believe hierarchical abstractions and meta-protocols both have a place in solving the complex problem developed and presented in this thesis. As sub-problems may be learnt, communication may help to solve those sub-problems. In this case, a hierarchical communication framework may be proposed, which would indeed be an interesting and new avenue for future research.

Bibliography

[Arulkumaran et al., 2017] Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. (2017). A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866.

[Dayan and Hinton, 1993] Dayan, P. and Hinton, G. E. (1993). Feudal reinforcement learning. In Hanson, S. J., Cowan, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*, pages 271–278. Morgan-Kaufmann.

[Deutscher, 2006] Deutscher, G. (2006). *The Unfolding Of Language*. Arrow.

[Eisenegger et al., 2014] Eisenegger, C., Naef, M., Linssen, A., Clark, L., Gandamaneni, P. K., Müller, U., and Robbins, T. W. (2014). Role of dopamine d2 re- ceptors in human reinforcement learning. *ncbi.nlm.nih.gov*.

[Foerster et al., 2016] Foerster, J. N., Assael, Y. M., de Freitas, N., and Whiteson, S. (2016). Learning to communicate with deep multi-agent reinforcement learning. *CoRR*, abs/1605.06676.

[Gareth et al., 2013] Gareth, M. J., Daniela, W., Trevor, H., and Robert, T. (2013). *An Introduction to Statistical Learning: With Applications in R*. Springer.

[Hamrick, 2020] Hamrick, J. (2020). Model-based methods in reinforcement learning.

[Hausknecht and Stone, 2015] Hausknecht, M. J. and Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 8.

[Hussein et al., 2017] Hussein, A., Gaber, M. M., Elyan, E., and Jayne, C. (2017). Imitation learning: A survey of learning methods. *ACM Computing Surveys*.

[Jurgen, 1987] Jurgen, S. (1987). *Genetic Programming*.

[Kerber, 2004] Kerber, M. (2004). Planning.

[LaVelle, 2006] LaVelle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.

[McKay, 2004] McKay, D. J. C. (2004). *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press.

[Narvekar et al., 2020] Narvekar, S., Peng, B., Leonetti, M., Sinapov, J., Taylor, M. E., and Stone, P. (2020). Curriculum learning with a progression function. *Journal of Machine Learning Research 21(181):1-50, 2020*.

[Papoudakis et al., 2019] Papoudakis, G., Christianos, F., Rahman, A., and Albrecht, S. V. (2019). Dealing with non-stationarity in multi-agent deep reinforcement learning. *CoRR*, abs/1906.04737.

[Russell and Norvig, 2016] Russell, S. and Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Prentice Hall Series In Artificial Intelligence, 3 edition.

[Shoham and Brown, 2009] Shoham, Y. and Brown, K. L. (2009). *Multiagent Systems: Algorithmic, Game-theoretic, and Logical Foundations*. Cambridge University Press.

[Stoyan, 1987] Stoyan, D. (1987). Berry, d. a. and b. fristedt: Bandit problems. sequential allocation of experiments. monographs on statistics and applied probability. chapman and hall, london/new york 1985, 275 s. *Biometrical Journal*, 29(1):20–20.

[Sutton and Barto, 2018] Sutton, R. and Barto, A. (2018). *Reinforcement Learning: An Introduction*. MIT Press, 2 edition.

[Sutton et al., 1999] Sutton, R. S., Precup, D., and Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, Volume 112.

[Tampuu et al., 2015] Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, J., Aru, J., and Vicente, R. (2015). Multiagent cooperation and competition with deep reinforcement learning. *CoRR*, abs/1511.08779.

[van Hasselt et al., 2016] van Hasselt, H., Guez, A., Hessel, M., and Silver, D. (2016). Learning functions across many orders of magnitudes. *CoRR*, abs/1602.07714.

[Wolfram et al., 2013] Wolfram, B., Maren, B., and Marco, R. (2013). Foundations of artificial intelligence (lecture 10. action planning).

[Wooldridge, 2009] Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. Wiley, 2 edition.

[Zhang and Sutton, 2017] Zhang, S. and Sutton, R. S. (2017). A deeper look at experience replay. *CoRR*, abs/1712.01275.

Appendix A

Ethics Checklist

UNIVERSITY OF BATH **Department of Computer Science**

12-Point Ethics Checklist for UG and MSc Projects

Student Amrinder Singh Rai (asr68)

Academic Year or Project Title 2019/2020

Supervisor Özgür Şimşek

This form must be attached to the dissertation as an appendix.

Does your project involve people for the collection of data other than you and your supervisor(s)? YES / NO

If the answer to the previous question is YES, you need to answer the following questions, otherwise you can ignore them.

This document describes the 12 issues that need to be considered carefully before students or staff involve other people ('participants' or 'volunteers') for the collection of information as part of their project or research. Replace the text beneath each question with a statement of how you address the issue in your project.

Appendix B

Source Code

All the other code for this project can be found here: https://github.com/AmrinderRai/MARL-Thesis

Jupyter notebooks have also been provided, as a manual and to help reproduce experiments.

B.1 Environment

Our implemented environment follows the following structure shown in figure B-1 below. Where pcb_env.py includes the logic and rewards scheme outlined in section 3.2, bringing the board and connector objects together.

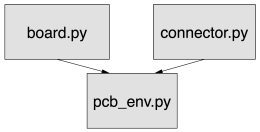

Figure B-1: Our environment files and their dependencies.

B.1.1 board.py

```
import numpy as np

class Board():
    EMPTY_CELL  = 0
    OBSTACLE_CELL = -1

    def __init__(self, board_size=[30,20]):
        self.height = board_size[0]
        self.width = board_size[1]
        self.grid = np.zeros((self.height, self.width),dtype=float)
        self.grid[:,:] = self.EMPTY_CELL

    def place_obstacle(self, coord):
        self.cover_cell(coord, self.OBSTACLE_CELL)

    def off_grid(self, coord): # Where coord is off the grid or not..
        return coord[0]<0 or coord[0]>=self.height or coord[1]<0 or coord[1]>=self.width

    def diag_collision(self, prev_pos, new_pos, direction):
        if direction==1 or direction==3:
            adj1 = (new_pos[0],new_pos[1]-1)
            adj2 = (prev_pos[0],prev_pos[1]+1)
        elif direction==5 or direction==7:
            adj1 = (new_pos[0],new_pos[1]+1)
            adj2 = (prev_pos[0],prev_pos[1]-1)
        else:
            return False
```

```
        if self.cell_val(adj1)==self.cell_val(adj2) and self.cell_val(adj1)!=self.EMPTY_CELL:
            return True
        else :
            return False

    def cover_cell(self, coord, obj):
        self.grid[coord] = obj

    def cell_val(self, coord):
        return self.grid[coord]
```

B.1.2 connector.py

```
import numpy as np

class Connector():
    NORTH = 0
    NORTHEAST = 1
    EAST = 2
    SOUTHEAST = 3
    SOUTH = 4
    SOUTHWEST = 5
    WEST = 6
    NORTHWEST = 7

    def __init__(self, start_coord, idx):
        self.id = idx
        self.trail = self.id
```

```
        self.head = self.id + 0.3 # Was 0.1
        self.goal = self.id + 0.6 # Was 0.2
        self.connection = [start_coord]

    def get_head(self):
        return self.connection[-1]

    def move(self, direction):
        self.connection.append(self._step(direction))

    def pop_head(self):
        self.connection.pop()

    def _step(self, direction):
        current_pos = self.get_head()

        if direction == self.NORTH:
            return (current_pos[0]-1, current_pos[1])

        elif direction == self.NORTHEAST:
            return (current_pos[0]-1, current_pos[1]+1)

        elif direction == self.EAST:
            return (current_pos[0], current_pos[1]+1)

        elif direction == self.SOUTHEAST:
            return (current_pos[0]+1, current_pos[1]+1)

        elif direction == self.SOUTH:
```

```
            return (current_pos[0]+1, current_pos[1])

        elif direction == self.SOUTHWEST:
            return (current_pos[0]+1, current_pos[1]-1)

        elif direction == self.WEST:
            return (current_pos[0], current_pos[1]-1)

        elif direction == self.NORTHWEST:
            return (current_pos[0]-1, current_pos[1]-1)
```

B.1.3 pcb_env.py

```
from components.board import Board
from components.connector import Connector
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import matplotlib.patheffects as pe
from IPython.display import clear_output
import numpy as np
import random
import pdb

class MultiPathGridEnv():

    def __init__(self, obstacles, starts, goals, grid_size=[20,15], obs_size=[1000, 1000], agents_n=1, train=False, in_place=Fal
        self.train = train
        self.in_place = in_place
        self.partial_view = True if obs_size<grid_size else False
```

```
        self.grid_size = grid_size
        self.obs_size = obs_size
        self.obs_height = self.obs_size[0]
        self.obs_width = self.obs_size[1]
        self.height = self.grid_size[0]
        self.width = self.grid_size[1]
        self.obstacles = obstacles
        self.connectors = []
        self.n_agents = agents_n
        self.action_space = list(range(0,8))
        # Assert that both these list sizes == n_agents after:
        self.starts = starts
        self.goals = goals
        self.episode_ended = [False]*self.n_agents
        self._init_board()

    def _init_board(self):
        self.board = Board(board_size=self.grid_size)

        if self.train:
            #obstacles_n = int((self.height/4) * (self.width/random.randint(5,6)))
            obstacles_n = 2
            coord_set = set([(i,j) for i in range(self.height) for j in range(self.width)])
            # or create multiple coord_sets to sample from ...
            # set([(i,j) for i in range(self.height/2) for j in range(self.width/2)]) etc.
            self.obstacles = random.sample(coord_set, obstacles_n)
            coord_set = coord_set.difference(self.obstacles)
            self.goals =  self.goals if self.in_place else random.sample(coord_set, self.n_agents)
            coord_set = coord_set.difference(self.goals)
```

```
        self.starts = self.starts if self.in_place else random.sample(coord_set, self.n_agents)

    for obstacle in self.obstacles:
        self.board.place_obstacle(obstacle)
    for i in range(0,self.n_agents):
        i_d = i+1
        connector = Connector(self.starts[i], i_d)
        self.connectors.append(connector)
        self.board.cover_cell(self.starts[i], connector.head)
        self.board.cover_cell(self.goals[i],  connector.goal)

def step(self, moves, idx):
    rewards = []
    all_obs = []
    for i, move in zip(idx, moves):
        reward, obs = self._move_connector(i, move)
        rewards.append(reward)
        all_obs.append(obs)

    return all_obs, rewards, self.episode_ended

def start_obs(self):
    # Assert agent == moves
    all_obs = []
    for idx in range(self.n_agents):
        connector = self.connectors[idx]
        head_coord = connector.get_head()
        dist = self._get_euclid_distance(head_coord, self.goals[idx])
        obs = (self.return_obs(head_coord), dist)
```

```
        all_obs.append(obs)

    return all_obs

def reset(self):
    self.connectors = []
    self._init_board()
    self.episode_ended = [False]*self.n_agents
    return self.start_obs()

def render(self):
    plt.ion()
    plt.figure(figsize=(20,10))
    ax = plt.gca()
    ax.clear()
    clear_output(wait = True)

    env_plot = self.board.grid.copy()
    env_plot = env_plot.astype(int)
    colors = ['grey', 'white', 'pink', 'red', 'green', 'blue', 'yellow', 'purple', 'purple']
    cmap = ListedColormap(colors[:self.n_agents+2])
    ax.matshow(env_plot, cmap=cmap)

    plt.show()


def _move_connector(self, idx, direction):
    connector = self.connectors[idx]
    prev_pos = connector.get_head()
```

```
self.board.grid[prev_pos] = connector.trail
reward = -1

connector.move(direction)
new_pos = connector.get_head()
new_dist = self._get_euclid_distance(new_pos, self.goals[idx])

if self.board.off_grid(new_pos):
    reward -=10
    self.episode_ended = [True]*self.n_agents
    #connector.pop_head()
elif self.board.cell_val(new_pos)==connector.goal:
    self.board.cover_cell(new_pos,connector.head)
    self.episode_ended[idx] = True
    reward += 20
elif self.board.cell_val(new_pos)!=self.board.EMPTY_CELL:
    reward -=10
    self.episode_ended = [True]*self.n_agents
else:
    self.board.cover_cell(new_pos,connector.head)
    if direction%2==1 and self.board.diag_collision(prev_pos, new_pos, direction):
        reward -=10
        self.episode_ended[idx] = True


return_obs = self.return_obs(connector.get_head())

return reward, (return_obs, new_dist)
```

```
def return_obs(self, head):
    obs = self.partial_obs(head) if self.partial_view else self.board.grid
    obs_shaped = np.expand_dims(obs*10, axis=0)
    return obs_shaped

def partial_obs(self, head):
    # Get bounding indices:
    top_loc = head[0]-self.obs_height//2
    bottom_loc = head[0] + 1+ self.obs_height //2
    left_loc = head[1] - self.obs_width //2
    right_loc = head[1] + 1 + self.obs_width //2

    if top_loc < 0:
        top = 0
        top_obs = np.abs(top_loc)
    else:
        top = top_loc
        top_obs = 0

    if bottom_loc > self.height:
        bottom = self.height
        bottom_obs = self.obs_height - np.abs(bottom_loc-self.height)
    else:
        bottom = bottom_loc
        bottom_obs = self.obs_height

    if left_loc < 0:
        left = 0
        left_obs = np.abs(left_loc)
```

```
        else:
            left = left_loc
            left_obs = 0

        if right_loc > self.width:
            right = self.width
            right_obs = self.obs_width - np.abs(right_loc-self.width)
        else:
            right = right_loc
            right_obs = self.obs_width

        obs = self.board.grid[top:bottom, left:right]

        padded_obs = np.ones((self.obs_height, self.obs_width))*[-1]

        padded_obs[top_obs:bottom_obs, left_obs:right_obs] = obs

        return padded_obs

    def _get_euclid_distance(self, head, goal):
        head = np.array(head)
        goal = np.array(goal)
        vector_dist = head-goal
        euclid_dist = np.sqrt(vector_dist[0]**2 + vector_dist[1]**2)

        return euclid_dist
```

B.2 Agents

In this section we have provided code for our RIAL implementation from section 4.2.3, which builds upon both our DQN and DRQN implementation from section 4.1. We also include our shared replay buffer from section 4.2.2 which equally builds upon, or relates closely to, all other replay buffers implemented.

Again, all the other code can be found at: https://github.com/AmrinderRai/MARL-Thesis

B.2.1 DRQN with RIAL

```
import numpy as np
import torch
import torch.nn as nn
import random
from collections import deque
from torch.autograd import Variable
from itertools import zip_longest
import pdb

class ReplayBuffer():
    def __init__(self, maxlen):
        self.buffer = deque(maxlen=maxlen)

    def add_to_buffer(self, experience):
        self.buffer.append(experience)

    def sample_minibatch(self, batch_size):
```

```
        sample_size = min(len(self.buffer), batch_size)
        sample_indices = random.choices(range(len(self.buffer)), k=sample_size)
        buffer_list = list(self.buffer)
        samples = [buffer_list[i] for i in sample_indices]
        return samples

class RQNet(nn.Module):
    def __init__(self, action_dim, comm_size, obs_size):
        super(RQNet, self).__init__()
        self.number_of_actions = action_dim
        self.message_comb = 2**comm_size
        self.obs_size = obs_size

        self.conv_feats = nn.Sequential(
            nn.Conv2d(1, 6, 5, stride=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(6, 12, 3, stride=1, padding=1),
            nn.ReLU()
        )

        self.lstm = nn.LSTM(input_size=109 , hidden_size=256, num_layers=3)

        self.action_q = nn.Sequential(
            nn.Linear(256, 856),
            nn.ReLU(),
            nn.Linear(856, self.number_of_actions)
        )

        self.message_q = nn.Sequential(
```

```
        nn.Linear(256, 856),
        nn.ReLU(),
        nn.Linear(856, self.message_comb)
    )

def forward(self, state, messages, hidden, episode_lengths, action_out=True, message_out=True):

    grid_state, dist_state = state

    grid_state_reshaped = grid_state.reshape(-1, 1, *self.obs_size)


    out = self.conv_feats(grid_state_reshaped)

    out = out.view(out.size(0), -1)
    dist_state = dist_state.view(out.size(0), -1)


    out = torch.cat((out, dist_state), dim=1)
    out = torch.cat((out, messages.view(messages.size(0),-1).squeeze(0))

    out = out.reshape(grid_state.size()[0], grid_state.size(1), -1)

    out = torch.nn.utils.rnn.pack_padded_sequence(out, episode_lengths.long(), batch_first=True, enforce_sorted=False)

    out, hidden = self.lstm(out, hidden)

    out, _ = torch.nn.utils.rnn.pad_packed_sequence(out, batch_first=True)
```

```
        action_out = self.action_q(out) if action_out else None
        message_out = self.message_q(out) if message_out else None

        return action_out, hidden, message_out

class DRQNAgent(object):
    def __init__(self, identity, n_actions, comm_size, device, obs_size):
        self.id = identity
        self.obs_size = obs_size
        self.device = device
        self.n_actions = n_actions
        self.action_space = [i for i in range(self.n_actions)]
        self.comm_size = comm_size
        self.qnet = RQNet(self.n_actions, self.comm_size, self.obs_size).to(device)
        self.qnet_optim = torch.optim.Adam(self.qnet.parameters(), lr=0.001)
        self.discount_factor = 0.99
        self.MSELoss_function = nn.MSELoss().to(device)
        self.memory = ReplayBuffer(1000)
        self.reset_hidden()
        self.network_loss_history_actions = []
        self.network_loss_history_messages = []
        self.message_indices = list(range(0,2**self.comm_size))

    def epsilon_greedy_action(self, state, messages, epsilon):
        q_actions, hidden, q_messages = self.qnet(state, messages, self.hidden, torch.Tensor([1]))
        self.hidden = hidden
        if np.random.uniform(0, 1) < epsilon:
            return random.sample(self.action_space,1)[0], random.sample(self.message_indices, 1)[0]
        else:
```

```
        policy_action = self.max_q_index(q_actions)
        policy_msg = self.max_q_index(q_messages)
        return policy_action, policy_msg

def policy_action(self, state, messages):
    q_actions, hidden, q_messages = self.qnet(state, self.hidden, messages, torch.Tensor([1]))
    self.hidden = hidden
    policy_action = self.max_q_index(q_actions)
    policy_msg = self.max_q_index(q_messages)
    return policy_action, policy_msg

def max_q_index(self, net_output):
    network_output_to_numpy = net_output.cpu().data.numpy()[0]
    max_value = np.max(network_output_to_numpy)
    max_indices = np.nonzero(network_output_to_numpy == max_value)[1]
    index = np.random.choice(max_indices)

    return index

def update_Q_Network(self, temporal_states, next_states, temporal_actions, temporal_messages, temporal_msg_idx, temporal_rew
    qs, _, _ = self.qnet(temporal_states, temporal_messages, hidden, episode_lengths, message_out=False)
    temporal_actions = temporal_actions.unsqueeze(2)
    temporal_msg_idx = temporal_msg_idx.unsqueeze(2)
    qsa = torch.gather(qs, dim=2, index=temporal_actions.long())

    _, _, qs = self.qnet(temporal_states, temporal_messages, hidden, episode_lengths, action_out=False)

    qsm = torch.gather(qs, dim=2, index=temporal_msg_idx.long())
```

```
qsa_mask = torch.zeros(qsa.size()[0:2]).to(self.device)
qsa_next_mask = torch.zeros(qsa.size()[0:2]).to(self.device)

for batch_no, seq_length in enumerate(episode_lengths):
    non_terminals = seq_length-1
    qsa_mask[batch_no,0:seq_length] = torch.ones(seq_length)
    qsa_next_mask[batch_no,0:non_terminals] = torch.ones(non_terminals)

qsa *= qsa_mask.unsqueeze(2)

qsa_next_actions, _, _ = self.qnet(next_states, temporal_messages, hidden, episode_lengths)
qsa_next_action, _ = torch.max(qsa_next_actions, dim=2, keepdim=True)

temporal_rewards = temporal_rewards.unsqueeze(2)

qsa_next_target = temporal_rewards + qsa_next_mask.unsqueeze(2) * self.discount_factor * qsa_next_action
qsa_next_target.to(self.device)


q_network_loss = self.MSELoss_function(qsa, qsa_next_target.detach())
self.network_loss_history_actions.append(q_network_loss.cpu().data.numpy())
self.qnet_optim.zero_grad()
q_network_loss.backward()
self.qnet_optim.step()

q_network_loss = self.MSELoss_function(qsm, qsa_next_target.detach())
self.network_loss_history_messages.append(q_network_loss.cpu().data.numpy())
self.qnet_optim.zero_grad()
q_network_loss.backward()
```

```
    self.qnet_optim.step()

def update(self, update_rate, batch_size):
    for i in range(update_rate):
        sample_batch = self.memory.sample_minibatch(batch_size)
        batch_size = len(sample_batch)
        episode_lengths = [len(episode) for episode in sample_batch]
        max_length = max(episode_lengths)
        temporal_states = np.zeros((batch_size, max_length), dtype=object)
        temporal_next = np.zeros((batch_size, max_length), dtype=object)
        temporal_messages = np.ones((batch_size, max_length), dtype=object)
        temporal_msg_idx = np.ones((batch_size, max_length))*[0]
        temporal_actions = np.ones((batch_size, max_length))*[0]
        temporal_rewards = np.ones((batch_size, max_length))* [0]
        for i, episode_length in enumerate(episode_lengths):
            sequence = np.array(sample_batch[i])
            for j, experience in enumerate(sequence[:episode_length]):
                temporal_states[i, j] = experience[0]
                temporal_next[i, j] = experience [1]
                temporal_messages[i, j] = experience[2]
                temporal_msg_idx[i, j] = experience[3][0]
                temporal_actions[i, j] = experience[4][0]
                temporal_rewards[i, j] = experience[5][0]

        temporal_states = self.get_padded_states(temporal_states)
        temporal_next = self.get_padded_states(temporal_next)
        temporal_msg_idx = torch.Tensor(temporal_msg_idx.tolist()).to(self.device)
        temporal_actions = torch.Tensor(temporal_actions.tolist()).to(self.device)
        temporal_rewards = torch.Tensor(temporal_rewards.tolist()).to(self.device)
```

```
        episode_lengths = torch.LongTensor(episode_lengths).to(self.device)

        hidden = (Variable(torch.zeros(3, batch_size, 256).float()), Variable(torch.zeros(3, batch_size, 256).float()))

        self.update_Q_Network(temporal_states, temporal_next, temporal_actions, temporal_messages, temporal_msg_idx, tempora

    def get_padded_states(self, state_np):
        batch_shape = state_np.shape
        grid_states = np.zeros((batch_shape), dtype=object)
        dist_states = np.zeros((batch_shape), dtype=object)
        for i, seq in enumerate(state_np):
            for j, ep in enumerate(seq):
                if type(state_np[i,j]) is not int:
                    grid_states[i,j] = ep[0]
                    dist_states[i,j] = [ep[1]]
                else:
                    grid_states[i,j] = np.zeros((1, *self.obs_size))
                    dist_states[i,j] = [0]
        t_states = (torch.Tensor(grid_states.tolist()).to(self.device), torch.Tensor(dist_states.tolist()).to(self.device))

        return t_states

    def reset_hidden(self):
        self.hidden = (Variable(torch.zeros(3, 1, 256).float().to(self.device)), Variable(torch.zeros(3, 1, 256).float().to(self
```

B.2.2 MARL training

```
state_dim = grid_size[0]*grid_size[1]
action_dim = 8
```

```
comm_size=2
dqn_agents = []
for i in range(n_agents):
    drqn_agents.append(DRQNAgent(i, action_dim, comm_size, device))
reward_sums_list = []
epsilon_history = []
remainin_on_termination = []

message_lookup = []
for i in range(0, 2**comm_size):
    bit_array = [int(d) for d in str(bin(i))[2:]]
    leading_zeros = [0]*(comm_size-len(bit_array))
    bit_array = leading_zeros + bit_array
    message_lookup.append(bit_array)

def train_agent(agents):
    TARGET_UPDATE = 4
    N = 30000
    max_time_steps = 3000
    epsilon = 0.9
    decay = 0.9999
    min_epsilon = 0.1
    agents_set = set(range(0,n_agents))
    for episode in range(N):
        reward_sum = 0
        state = env1.reset()
        current_msgs = torch.zeros((n_agents,comm_size))
        epsilon = max(min_epsilon, epsilon*decay)
        epsilon_history.append(epsilon)
```

```
terminated_indices = []
episode_experience = [[] for agent in agents]
for i in range(max_time_steps):
    remaining_agents = agents_set - set(terminated_indices)
    chosen_actions = []
    message_indices = []
    for i, a_id in enumerate(remaining_agents):
        t_state = (torch.from_numpy(state[i][0]).float().unsqueeze(0).to(device),torch.Tensor([state[i][1]]).to(device))
        action, message_idx = agents[a_id].epsilon_greedy_action(t_state, current_msgs, epsilon)
        message_indices.append(message_idx)
        current_msgs[a_id] = torch.Tensor(message_lookup[message_idx])
        chosen_actions.append(action)
        if type(action) is not list:
            action = [action]
    next_state, reward, terminal = env1.step(chosen_actions, remaining_agents)
    for i, a_id in enumerate(remaining_agents):
        episode_experience[a_id].append((state[i], next_state[i], current_msgs, [message_indices[i]], action, [reward[i]
    reward_sum += np.mean(reward)
    state = next_state
    #env1.render()
    if all(terminal):
        clear_output(wait=False)
        display('Episode {0} ended with {1} agents remaining'.format(str(episode), remaining_agents))
        remainin_on_termination.append(len(remaining_agents))
        reward_sums_list.append(reward_sum)
        reward_sum = 0
        break
    else:
        terminal = np.array(terminal)
```

```
                remaining_terminal = np.delete(terminal, terminated_indices)
                terminated_indices = terminal.nonzero()[0]
                idx_remove = remaining_terminal.nonzero()[0]
                if idx_remove.any():
                    for idx in sorted(idx_remove, reverse=True):
                        del state[idx]

        for i, agent in enumerate(agents):
            agent.memory.add_to_buffer(episode_experience[i])
            if episode !=0:
                agent.update(2, 20)
    train_agent(dqn_agents)
```

B.2.3 Shared Buffer Replay (DRQN)

```
class DRQN_SharedReplayBuffer():
    def __init__(self, maxlen):
        self.buffer = deque(maxlen=maxlen)
        self.priorities = deque(maxlen=maxlen)

    def add_to_buffer(self, experiences, idx):
        self.buffer.append((experiences, idx))

    def sample_minibatch(self, batch_size, idx):
        sample_size = min(len(self.buffer), batch_size)
        sample_indices = random.choices(range(len(self.buffer)), k=sample_size)
        samples = np.array(self.buffer)[sample_indices]
        samples_pov = self.switch_perspective(samples, idx)
        return samples_pov
```

```
def switch_perspective(self, episodes, pov_id):
    pov_id +=1
    pov_id *=10 =
    episode_indices = episodes[:,1]
    episodes = episodes[:,0]


    switched_episodes = []

    for i, episode in enumerate(episodes):
        switched_samples = []
        idx = episode_indices[i]
        idx +=1
        idx *=10 # hardcoded
        for sample in episode:
            pov_state = sample[0][0]
            pov_next_state = sample[1][0]

            mapping = {
                idx: pov_id,
                idx+3: pov_id+3,
                idx+6: pov_id+6,
            }

            pov_state_switched = copy(pov_state)
            pov_next_switched = copy(pov_next_state)
            for k,v in mapping.items():
                pov_state_switched[pov_state==k] = v
```

```
            pov_state_switched[pov_state==v] = k
            pov_next_switched[pov_next_state==k] = v
            pov_next_switched[pov_next_state==v] = k
        pov_state_switched = (pov_state_switched, sample[0][1])
        pov_next_switched = (pov_next_switched, sample[1][1])

        switched_samples.append((pov_state_switched, pov_next_switched, *sample[2:]))
    switched_episodes.append(switched_samples)
return switched_episodes
```