Technical University of Crete
School of Electrical and Computer Engineering

# Deep Reinforcement Learning in the Flatland Multi-Agent Environment

Diploma Thesis

Stavros Ntaountakis

Committee

Supervisor : Georgios Chalkiadakis, Associate Professor

Committee Member : Michail G. Lagoudakis, Associate Professor

Committee Member : Georgios N. Yannakakis, Professor (University of Malta)

Chania, October 2021

Πολυτεχνείο Κρήτης
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

# Βαθιά Ενισχυτική Μάθηση στο Πολυπρακτορικό Περιβάλλον Flatland

Διπλωματική Εργασία

Σταύρος Νταουντάκης

Επιτροπή

Επιβλέπων : Γεώργιος Χαλκιαδάκης, Αναπληρωτής Καθηγητής

Μέλος Επιτροπής : Μιχαήλ Γ. Λαγουδάκης, Αναπληρωτής Καθηγητής

Μέλος Επιτροπής : Γεώργιος Ν. Γιαννακάκης, Καθηγητής (Πανεπιστήμιο Μάλτας)

Χανιά, Οκτώβριος 2021

# Abstract

Over the last few years, railway traffic networks have been increasing in size and complexity due to the ever-growing transportation demands. As a result, railway companies, such as the Swiss Federal Railway company, need to constantly adapt to the increasing transportation demands. FlatLand is a simplified 2D grid simulation that mimics the dynamics of a railway network and was developed as an open sandbox to accelerate academic research on the Vehicle Rescheduling Problem (or VRSP) in the fields of Machine Learning and Operations Research.

FlatLand is characterized by many of the common problems that need to be tackled in multi-agent systems. The coexistence of multiple autonomous agents results in a non-stationary environment and a partially-observable state space. At the same time the rewards received by the agents are sparse and delayed, since coordinated sequence of actions are usually required for yielding such positive rewards.

Under these considerations, in this thesis, we implement and adapt various Deep Reinforcement Learning methods in the environment of FlatLand. We systematically compare and evaluate both value-based and policy-based methods on various metrics of performance and reliability. We ensure consistent and fair training conditions by employing each agent on a strictly defined training and evaluation setup. We implement standard DQN methods, as well the Double and Dueling Double DQN variants, and adapt them to multiple agents. Additionally, we implement a modified PPO agent as well as a superior PPO agent attached to a Replay Buffer. Lastly, we propose SIL, an agent that combines PPO with Self-Imitation and converges to a successfull policy in most environment settings. SIL is shown to excibit superior performance with respect to all other agents we implemented and tested.

# Abstract in Greek

Τα τελευταία χρόνια, τα δίκτυα σιδηροδρομικών σταθμών αυξάνονται συνεχώς σε μέγεθος και πολυπλοκότητα λόγω των συνεχώς αυξανόμενων αναγκών μετακίνησης. Ως αποτέλεσμα, οι σιδηροδρομικές εταιρίες, όπως η Swiss Federal Railway, χρειάζεται να προσαρμόζονται συνεχώς στις αυξανόμενες αυτές ανάγκες. Το FlatLand είναι ένα απλοποιημένο δισδιάστατο περιβάλλον, το οποίο προσομοιώνει τις δυναμικές ενός σιδηροδρομικού δικτύου και δημιουργήθηκε ως μια ανοιχτή πλατφόρμα με στόχο την επιτάχυνση της ακαδημαϊκής έρευνας στο πρόβλημα αναπρογραμματισμού οχημάτων, αξιοποιώντας τα πεδία της Μηχανικής Μάθησης και της Επιχειρησιακής Έρευνας.

Το FlatLand χαρακτηρίζεται από τα περισσότερα απο τα κοινά προβλήματα που πρέπει να αντιμετωπιστούν σε ένα πολυπρακτορικό σύστημα. Η συνύπαρξη πολλαπλών αυτόνομων πρακτόρων έχει ως αποτέλεσμα την μη στασιμότητα του περιβάλλοντος και την μερική παρατηρησιμότητα του χώρου καταστάσεων. Ταυτόχρονα, οι επιβραβεύσεις που λαμβάνουν οι πράκτορες στο FlatLand είναι αραιές και καθυστερημένες, διότι συνήθως πρέπει να προηγηθεί μια συγχρονισμένη ακολουθία σωστών κινήσεων, ώστε αυτές να ληφθούν.

Υπό αυτές τις θεωρήσεις, σε αυτήν την διπλωματική εργασία, εφαρμόζουμε και προσαρμόζουμε διάφορες τεχνικές Βαθιάς Ενισχυτικής Μάθησης στο περιβάλλον FlatLand. Συγκρίνουμε και αξιολογούμε αυτές τις μεθόδους συστηματικά, μέσω διαφόρων μετρικών απόδοσης και αξιοπιστίας. Εξασφαλίζουμε σταθερές και ισότιμες συνθήκες εκπαίδευσης, και εκπαιδεύουμε τον κάθε πράκτορα σε ένα αυστηρά καθορισμένο περιβάλλον εκπαίδευσης και αξιολόγησης. Υλοποιούμε μεθόδους, όπως την γνωστή και επιτυχημένη DQN, καθώς και τις παραλλαγές της, Double και Dueling Double DQN, και τις προσαρμόζουμε σε συνθήκες πολλαπλών πρακτόρων. Επιπλέον, υλοποιούμε μία τροποποιημένη εκδοχή του αλγόριθμου PPO, καθώς και μια βελτιωμένη εκδοχή ενός PPO αλγόριθμου προσδεδεμένου σε έναν Replay Buffer. Τέλος, προτείνουμε τον SIL, έναν πράκτορα που συνδυάζει την μέθοδο PPO με την τεχνική της αυτομίμησης. Μέσω μεθοδικών πειραματισμών, επιδεικνύουμε την ανωτερότητα του SIL σε απόδοση, σε σχέση με όλους τους πράκτορες που υλοποιήσαμε.

# Contents

# Chapter 1

# Introduction

Artificial Intelligence (or AI) is a deeply studied and dense field that tries to understand intelligent beings and simulate their intelligent behaviors onto machines [27]. There are many applications to benefit from AI such as speech recognition, computer vision, self-driving vehicles, robotics, natural language processing and more. Machine Learning (or ML) is a branch of AI that tries to imitate certain human behaviors by constantly processing data and gradually improving its accuracy. ML can be divided into three categories: Supervised learning, Unsupervised learning and Reinforcement learning. Reinforcement learning is a technique where a learner (or agent) learns from experience by interacting with the environment and accumulating rewards. When multiple autonomous agents that learn from experience coexist in the same environment, they constitute a *Multi-Agent System* (or MAS). Developing RL algorithms that can handle multiple agents can have a great impact in many real life applications such as autonomous vehicles, drone swarms or resource management systems. These tasks, however, can be really complicated, and experimenting on these algorithms in the real world can be a very expensive, dangerous and mostly impossible task. Hence, there is a need for simpler, well understood simulation environments (such as the OpenAI Gym[1], Power TAC[2], DeepMind Control Suit[3]) that the academic community can experiment with different methods and slowly develop algorithms that can later handle real world applications.

Now, over the last few years, Deep Reinforcement Learning has sparked academical interest, especially after the great success of DQN [18]. Subsequently, multi-agent Deep Reinforcement Learning has also grown in popularity. When dealing with multiple agents, new and exciting issues arise and need to be tackled. While a lot of work is being done in developing methods explicitly for multiple cooperative or adversary agents [7, 26], in some cases, extending single agent RL methods to multiple agents is sufficient for sub-optimal results. Especially when the coordination requirements are straightforward and clearly specified, or when the agents share common goals and behaviors, single agent RL methods are easy to adapt and operate in such settings.

[1] `https://gym.openai.com/`

[2] `https://powertac.org/`

[3] `https://deepmind.com/research/publications/2019/deepmind-control-suite`

FlatLand is a simplified 2d grid simulation environment that mimics the dynamics of a railway network. In FlatLand, multiple autonomous trains (or agents) try to reach their destination as fast as possible without colliding with each other or getting stuck in dead ends. FlatLand was developed as a tool to accelerate academical research on the Vehicle Rescheduling Problem (or VRSP) and is suitable for both Machine Learning research and Operation Research [19].

As a multi-agent platform, FlatLand is characterized by the common problems of multi-agent systems. To name a few, the existence of multiple autonomous and unpredictable agents results in a non-stationary environment. Under this non-stationarity, the agents need to constantly adapt to unforeseeable outcomes while at the same time only being able to partially observe the true underlying state space. Additionally, in FlatLand, agents only yield positive rewards when they reach their destination. When multiple trains coexist in the same environment, a sequence of correct and synchronized actions is usually required to obtain these rewards. Under these circumstances, the rewards obtained by the agents are sparse and delayed.

In this thesis, we aim to tackle the Vehicle Rescheduling problem by implementing several Deep Reinforcement Learning algorithms in the multi-agent environment of FlatLand. We select both value-based and policy-based methods with the intentions to discuss their different strengths and weaknesses. We do so by systematically comparing and evaluating them in a strictly specified training and evaluation setup. We give great emphasis in ensuring equal and fair training and evaluation conditions for all the algorithms as a way to warrant objective experimental results.

## 1.1 Contributions

Despite its few years of existance, Flatland has started to gain some academical interest in the fields of *Operations Reseach* and *Reinforcement Learning*. However, most of the published Reinforcement Learning research in Flatland revolve around the Flatland competitions (*NeurIPS 2020*, *AMLD 2021*) and usually only focus on the results of the competitions and their approach towards winning these competitions. To our knowledge, not much work has been done in systematically evaluating and comparing different RL methods under a consistent and systematic training and evaluation setup. In this thesis, we:

- Implement, adapt and scale value-based and policy-based RL algorithms to the multi-agent environment of FlatLand.
- We integrate useful extensions proposed by [6] to the well-known PPO method [30].
- We demonstrate the importance and superiority of attaching a Replay Buffer to PPO.
- We establish well-defined, consistent and dense training and evaluation conditions and compare the different algorithms in various metrics of raw performance and reliability.

- We employ reliability metrics that provide us with information about the consistency and stability of each algorithm and emphasize on the importance of also considering these metrics when examining the performance of RL methods.
- We identify SIL as our best performing agent, which is adapted to the needs of the FlatLand environment and combines PPO with Self-Imitation.

## 1.2 Thesis Outline

The thesis is structured as follows. In Chapter 2, we begin by providing the necessary Theoretical Background in Reinforcement Learning, Deep Learning, Deep Reinforcement learning, the model-free Deep Reinforcement algorithms we implemented and briefly review related work. In Chapter 3, we provide a short introduction in the environment of Flatland. We then continue in Chapter 4, by shortly discussing some of the attributes of the Flatland environment and dive into our implementation of the different algorithms. Then, in chapter 5, we define our evaluation setup and evaluate the experimental results. Finally, we conclude by taking a step back and discussing our work as well as potential future work.

# Chapter 2

# Theoretical Background and Related Work

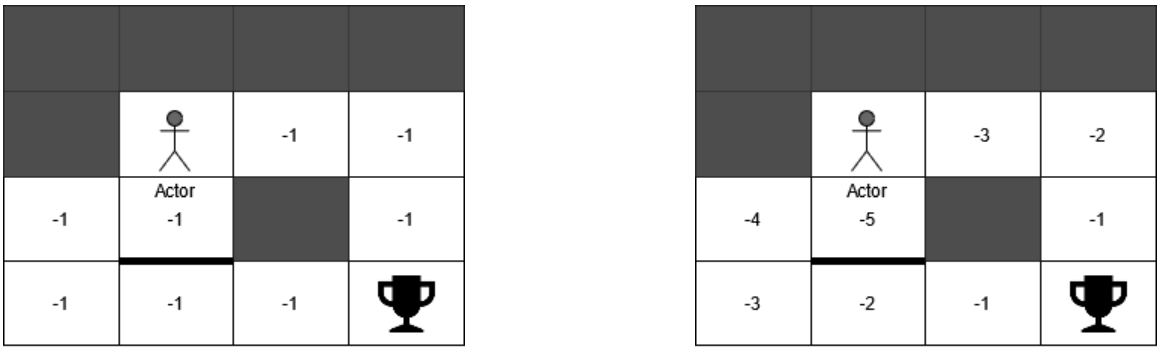
In this chapter we will bring the necessary theoretical foundation for the contents of this thesis. First, we will provide a gentle introduction to the field of *Reinforcement Learning* and general terminology. Then, we will deliver a short introduction in the field of *Deep Learning*. Lastly, we will combine the two by introducing *Deep Reinforcement Learning* and talk about related work.

## 2.1 Reinforcement Learning

*Reinforcement Learning* (or *RL*) is a technique which belongs in the field of Machine Learning. The term was first brought in literature by Waltz et. al in 1961 but the ideas behind reinforcement learning date as back as the late 50's in the fields of psychology and engineering [32]. Reinforcement learning focuses on learning from experience by interacting with the environment. Similar to how a newborn learns to walk, a learner - also referred to as an agent- has no initial belief of what actions are right or wrong and has no preconception of the environment it belongs to. It learns the given task by trial and error. By taking actions and then tracking the reward it got for those actions, the agent develops its *policy*: its belief of what actions to take at any given moment in the environment [32].

In a Reinforcement Learning setting, it is common that the environment runs in discrete time steps called *states* and a whole run of the environment is called an *episode*. At each *state* the agent decides what *action* to take based on its current *policy*. To understand the *policy*, we can think of it as a table that keeps what action to take at every environment state. After taking an action the environment returns a *reward*. The *reward* is usually only a signal that indicates whether the agent reached a *goal* or a checkpoint. In order to estimate the real value of a state however, one needs to compute a *value function* which incorporates the long-term utility associated with the corresponding state. While learning, an agent tries to approximate the real value of all states by constantly calculating its current belief of the values of those states.

To understand the difference between rewards and value functions, we can think of an agent that tries to find an exit on a labyrinth. After every action, the agent gets rewarded




(a) Rewards. (b) Values.

Figure 2.1: Difference between Rewards (a) and actual state values (b) in a simple grid world where the agent has to reach to its goal as fast as possible.

with 0 if it has reached the exit and -1 if it has not. The value function of the state can be thought of as the sum of rewards from this state onwards. A state three steps away from the exit will have a value of -3. If the agent knew the actual value of all the states, it would select actions that would lead it to the state with the least negative value. Even from this simple example, we can see that policies and value functions are mutually dependent. In the following chapters, we will expand further on their relationship.

### 2.1.1 Markov Decision Process

Following the above introduction, we will now focus on the basic concept that most RL problems rely on, that of a *Markov Decision Process* or *MDP*. An *MDP* is a term that was first introduced by Richard Bellman in 1957 and is a specific type of a *sequential decision model* [25]. An environment is described as a *sequential decision model* where at each timestep, an agent selects an *action* by observing a *state* of the environment for that timestep. By selecting that *action*, the agent then receives a *reward* and the environment transitions to a next *state* [25].

In a *Markov Decision Model*, the agent selects an action at a given timestep by only observing the state at that timestep and not considering previous states. In such settings, a state is said to have the *Markovian Property* [32]. For example, we can think of an agent that learns to drive. Any given state in the environment would be *Markovian* if the agent in that state had all the necessary information such as velocity, direction, steering, fuel etc. at hand because that information contains everything the agent needs to know for moving forward. This information is connected with the past: fuel remaining indicates that fuel was spent, a steered wheel shows that we are on a turn. If, however, the state observation only contained positional information then the agent would not be able to understand much about the situation it is in.

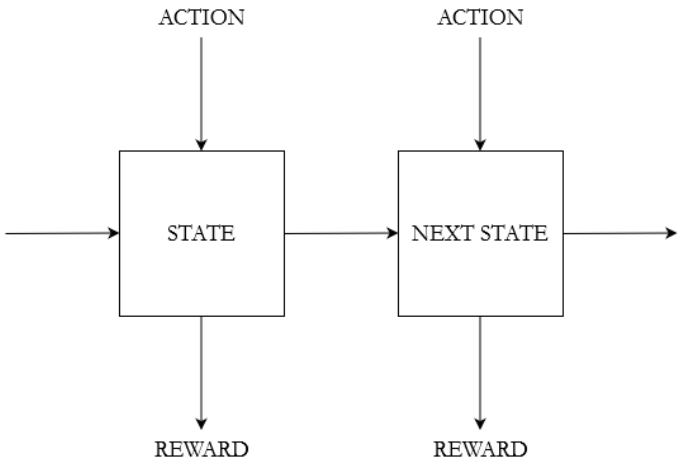



Figure 2.2: A Sequential Decision Model [25].

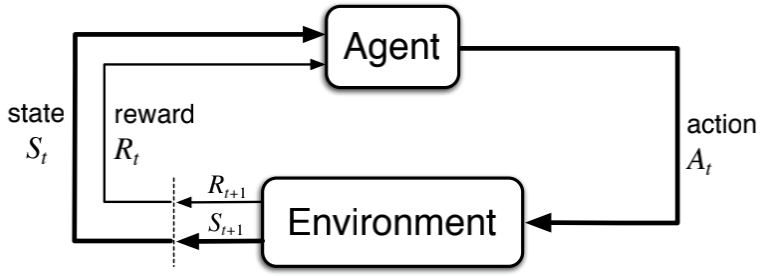



Figure 2.3: A Markov Decision Process [32].

MDPs are expressed as a tuple: $\langle s, a, p(s'), r(s, s') \rangle$. This tuple perfectly describes the one-step dynamics of the environment where:

- $s$ is the current state of the environment
- $a$ is the action that was selected from state $s$
- $p(s')$ expresses the distribution describing the probability of transitioning to all possible next states $s'$ by selecting action $a$ on state $s$
- $r(s, s')$ are the expected immediate rewards if we transition to $s'$ by selecting action $a$ on state $s$

The above description of the MDP tuple makes use of the environment's model in order to estimate the probability of transitioning to next possible states. In most RL settings, however, we are not aware of the model of the environment. Hence, we cannot assume knowledge of the transition probabilities. Instead, we sample the immediate reward and next state by interacting with the environment and the MDP tuple is described as: $\langle s, a, r, s' \rangle$.

### 2.1.2 Value and Policy

We have already expressed the relationship of value functions and policies in RL. We will now further expand in how this relationship can lead to a successfully learning agent. Moving forward, we will now only focus on methods that assume no model of the environment.

As previously mentioned, the *value function* estimates the actual value of a state and can be simply thought of as the sum of rewards from that state and onwards. Those rewards, however, depend on the actions the agent will take by following its *policy*. Therefore, at timestep $t$, for any given state $s$, the value of that state is tied under a policy $\pi$ and is defined as:

$$u^{\pi}(s_t) = E^{\pi}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...] = E^{\pi}[\sum_{k=0}^{n} \gamma r_{t+1+k}] \tag{2.1}$$

where $\gamma \epsilon [0, 1)$ is a *discount factor* and indicates the importance of future rewards. A low discount factor (i.e. close to 0) gives focus to the rewards in the near future and vice versa.

Another value function estimator used in RL settings is called an *action-value function*, $q(s, a)$. While the *value function*, $u(s)$, estimates the expected value of a state under all possible actions, $q(s, a)$ estimates the expected value of a state $s$ after selecting an action $a$ [32].

$$q^{\pi}(s_t, a_t) = E^{\pi}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|a_t] = E^{\pi}[\sum_{k=0}^{n} \gamma r_{t+1+k}|a_t] \tag{2.2}$$

We define the *optimal value* and *action-value functions* as the true values of those states and state-action pairs. These *optimal* values are described by the *Bellman Optimality Equations*:

$$u^{\star}(s_t) = \max_{\alpha \epsilon A_t} E[r_{t+1} + \gamma u^{\star}(s_{t+1})] \tag{2.3}$$

$$q^{\star}(s_t, a_t) = \max_{\alpha \epsilon A_t} E[r_{t+1} + \gamma \max_{a_{t+1} \epsilon A_{t+1}} q^{\star}(s_{t+1}, a_{t+1})] \tag{2.4}$$

It is clear that better policies lead to better value functions and vice versa. We update policies in order to better estimate value functions and we update value functions in order to find better policies. A term that generalizes this connection of policy updates and value function evaluations is *Generalized Policy Iteration* (or *GPI*) [32].

GPI can be applied over various RL methods. In fact, there are *three* prominent RL methods:

- *Dynamic Programming methods*: These methods assume knowledge of the environment's *model* and all the states and state transitions are known. As a result, policy evaluations can occur synchronously or asynchronously at each policy iteration over all possible states.
- *Monte Carlo methods*: These methods belong in the *model-free* approaches. They evaluate and update policies by visiting states and averaging the subsequent gathered rewards of those visited states under those policies.
- *Temporal Difference methods*: Like Monte Carlo methods, they do not require a *model* of the environment. Unlike Monte Carlo methods, however, they do not require playing a whole episode for evaluating policies. Instead, they take advantage of the one-step dynamics of the MDP environment and only require the next state transitions to evaluate policies.

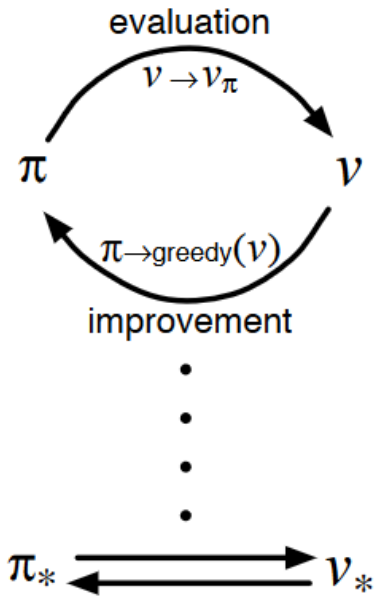



Figure 2.4: Generalized Policy Iteration [32].

In order to understand the relationship between value functions and policies and how these can be employed by a learning agent, we will focus in *Q-learning*: an *off-policy, Temporal Difference* algorithm that has heavily influenced many modern Deep RL implementations [18, 36, 37].

The term *off-policy* describes the relationship between the policy that estimates the Q-targets i.e., *target policy*, and the policy that acts on the environment i.e., *behavior policy*. Q-learning is off-policy, because the target, $R + \gamma \max_a Q(S', a)$, is greedily selecting the action $a$ that maximizes $Q(S', a)$ and does not follow the $\epsilon$-*greedy* behavioral policy.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize $S$
  Repeat (for each step of episode):
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-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 2.5: Q-learning pseudocode [32].

Alternatively, an on-policy approach would instead use a target of: $R + \gamma Q(S', a')$, by $\epsilon$-*greedily* selecting $a'$ while following its policy (Sarsa algorithm).

As mentioned above, *Q-learning's* behavioral policy is the $\epsilon$-*greedy* selection of actions derived from $Q$. This means that when selecting an action $a$ upon a state $S$, the action that gives the maximum $Q(S,a)$ is selected with probability $1-\epsilon$ and a random action is selected with probability $\epsilon$. This is done as a measure to provide sufficient exploration in the environment and not to lock on the seemingly best actions. This is Q-learning's answer to tackling the *Exploration vs Exploitation dilemma*, one of the greatest challenges in RL [32]. On one hand we want to *exploit* our knowledge of the environment to maximize our rewards but on the other hand we want to *explore* the environment to find states that could yield us greater rewards in the long term.

*Q-learning* follows an update rule on Q-values, where:

$$Q^{\pi'}_{k+1}(S,a) = Q^{\pi}_{k}(S,a) + \alpha[R + \max_{a'} Q(S',a') - Q^{\pi}_{k}(S,a)] \tag{2.5}$$

This update rule satisfies the GPI approach by simultaneously performing an *evaluation step* and an *improvement step*. When following the above update rule, we evaluate state-action values that were visited while following policy $\pi$ and we improve on those state-action values by updating them towards a greedy target:

$$R + \gamma \max_{a'} Q(S',a') \tag{2.6}$$

## 2.2 Deep Reinforcement Learning

As the name implies, *Deep Reinforcement Learning* (or *Deep RL*) merges the fields of *Reinforcement Learning* and *Deep Learning*. Combining RL methods with Deep Neural

Networks has led to great success in dealing with sequential decision-making challenges that were not possible with standard RL methods [9]. While standard RL methods perform well in simple environments and small state spaces (i.e., simple labyrinths, grid worlds), they struggle when the different states increase and get more complex. The agent would have to store and approximate state values and policies for every visited state. This not only leads to bigger computational and memory requirements but also cripples learning since the exact same state will probably never be revisited and better approximated.

Neural networks are great at tackling this problem because of their ability to generalize over high-dimensional state spaces. In Deep RL the observed states are being fed forward to neural networks and the focus is on training those networks by adjusting their parameters to better approximate the desired output which can be the state's value function, action-value function or policy. In the following sections, we provide a short theoretical background on Neural Networks and dive into the Deep RL methods that were implemented in this thesis.

### 2.2.1 Feedforward Neural Networks

Deep Neural Networks are essentially parameterized function approximators, meaning that given an input $\mathbf{x}$, they approximate a function $f(\mathbf{x})$ that produces the desired output $\mathbf{y}$ by adjusting some parameters $\boldsymbol{\theta}$. They are called *Neural* because they're inspired by the neurons of the human brain and are called *Networks* because $f(x)$ is in fact a network of chained functions [10].

*Feedforward Neural Networks* can be thought of as a function with parameters $\boldsymbol{\theta}$ defined as:

$$\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta}) \tag{2.7}$$

The name "Feedforward" implies that the input $\mathbf{x}$ gets forwarded through the network and produces an output $\mathbf{y}$. The input is processed by a chain of nonlinear functions, known as *hidden layers*, until the output is reached. Those chained functions are called hidden layers because their output is not directly accessible [10]. A simple case of a Neural Network is represented below.

The output of the first hidden layer is a non-linear transformation:

$$\mathbf{h}_1 = A(\mathbf{x}^T \mathbf{W}_1 + \mathbf{b}_1) \tag{2.8}$$

where the one-dimensional input vector $\mathbf{x}$ of size $m$, $\mathbf{x} = [x_1, x_2, ..., x_m]$ is multiplied with a parameterized weighted vector $\mathbf{W}_1$ of size $(m \times n)$ added by a *bias* term and processed by an *activation function* $A$. The activation function results in a nonlinear transformation

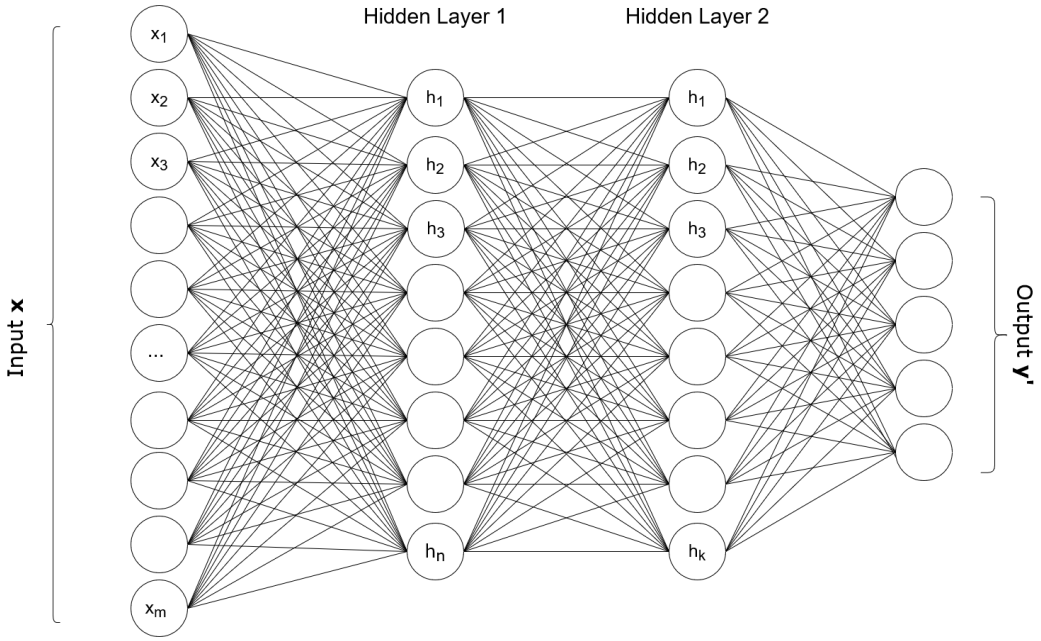



Figure 2.6: Feedforward Neural Network with two hidden layers.

of the hidden layer's linear output. Next, the $\mathbf{h}_1$ vector of size $n$ goes through another non-linear transformation:

$$\mathbf{h}_2 = A(\mathbf{h}_1^T \mathbf{W}_2 + \mathbf{b}_2) \tag{2.9}$$

where now, $\mathbf{W}_2$ is of size $(n \times k)$. Finally, the predicted output $\mathbf{y}'$, is the result of a final non-linear transformation where:

$$\mathbf{y}' = A(\mathbf{h}_2^T \mathbf{W}_3 + \mathbf{b}_3) \tag{2.10}$$

Training the *Neural Network* (or *ANN*) is the result of adjusting the values of the weighted vectors and biases. While initialization of those values can affect the performance of the neural network [38], they are usually random and the initial output predictions are random. In order to learn, the network first needs to know how far off its predictions were compared to the desired output values. This is done by a *Loss Function* which in its most common form can be a *Mean Square Error function* (or *MSE*):

$$Loss(\boldsymbol{\theta}) = \frac{1}{N} \sum_i (\mathbf{y}_i - \mathbf{y}_i')^2 \tag{2.11}$$

where $N$ is the training set size. The goal is to minimize the *loss function*. This is done by *back-propagation*: adjusting the network parameters $\boldsymbol{\theta}$ by gradient descent on the loss function w.r.t the parameters [13].

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha * \nabla_{\boldsymbol{\theta}} Loss(\boldsymbol{\theta}) \tag{2.12}$$

where $\alpha$ is the *learning rate* and indicates how big of a step to take towards updating the *weights*. While the complications of using the wrong learning rate are beyond the scope of this introductory segment, it is important to understand that using too large (or too small) learning rate can heavily affect the performance of the neural network and can lead to suboptimal learning or no learning at all.

Neural Networks can be modified to fit in many different applications. Those modifications can be the size of the hidden layers, the number of the hidden layers but also the Loss or Activation Function. Additionally, different kinds of layers can be applied such as *Convolutional Layers* (used commonly in *CNNs*). This adaptability has led Neural Networks to perform great in various settings including Reinforcement Learning.

### 2.2.2 Recurrent Neural Networks

*Recurrent Neural Networks* are a special type of *Neural Networks* that specialize in handling sequences of input data. They are able to correlate information seen in the sequence of inputs by forming an interconnected loop that passes the hidden state from one step of the sequence to the next step [10]. As a result, the hidden state at any step, carries information from the previous steps in the sequence and acts as a short memory of the past. Recurrent Neural Networks have been of great use in many fields, such as speech or text processing, music composition etc. [13].

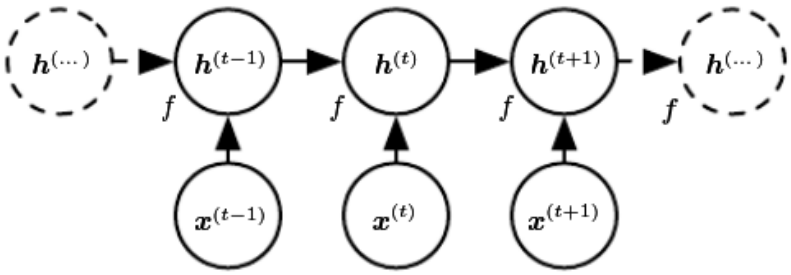



Figure 2.7: A simple Recurrent network that processes a sequence of inputs and passes forward its hidden state [10].

One of the most common Recurrent architecture is *Long Short-Term Memory* or *LSTM*. They have the ability to process longer input sequences than other recurrent Neural Network architectures while maintaining a good understanding of the long-term dependencies of the sequences. While LSTM does maintain the structure of Recurrent Neural Nets (they do pass forward a hidden state), they differ in the internal architecture of each module that handles one step of the sequence.

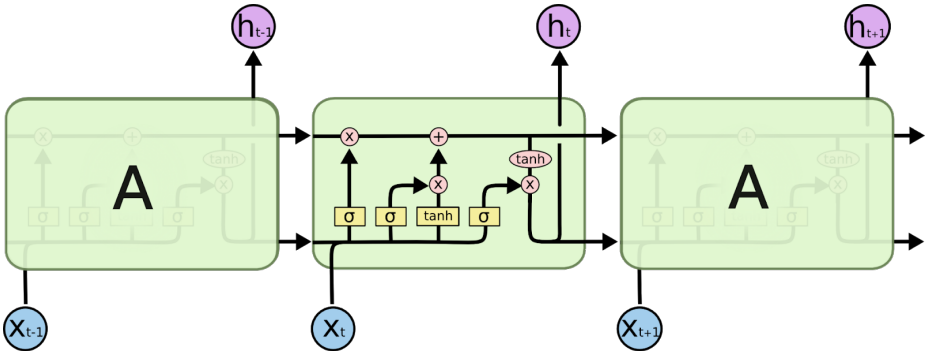



Figure 2.8: A visualisation of an LSTM network architecture [21].

### 2.2.3 Deep RL methods used in this thesis

Having provided the basic introductory information, we will now focus on the actual Deep Reinforcement Learning methods that we applied on our problem. Below we represent all the RL methods that are being presented in the later sections.

| Value - based methods | Policy - based methods |
|---|---|
| Deep Q-Networks | Policy Gradient Optimization |
| Double Deep Q-Networks | Proximal Policy Optimization |
| Dueling Double Deep Q-Networks | Self-Imitation Learning |
| Recurrent Deep Q-Networks | |
| Prioritized Experience Replay | |

Table 2.1: A table representing all of the RL methods discussed below.

#### 2.2.3.1 Deep Q-Networks

We begin with the value-based methods and more specifically with *DQN*: an algorithm that was published by DeepMind in 2015 and has been one of the most influential algorithms in recent RL research [9].

DQN combines Deep Neural Networks with a modified Q-learning algorithm. In the original implementation, it tackles the challenge of training an agent who can only visually observe an environment state in the form of a video input. The video is being split in frames and each frame is being pre-processed to a fixed size vector [18]. These image vectors, which are the agent's observation of the environment's state, are then being fed into a Neural Network architecture and the outputs are Q-values corresponding to the number of possible actions. The network's architecture is a convolutional neural network which consists of three hidden Convolutional Layers followed by a fully-connected layer.

The agent acts on the environment and stores every transition $\langle \phi_t, a_t, r_t, \phi_{t+1} \rangle$ in a Replay Buffer. The Replay Buffer's size is adapted to the implementation's requirements. In the original paper, it holds one million most recent frames. Training the Q-network involves sampling random mini-batches of N transitions from a Replay Buffer.

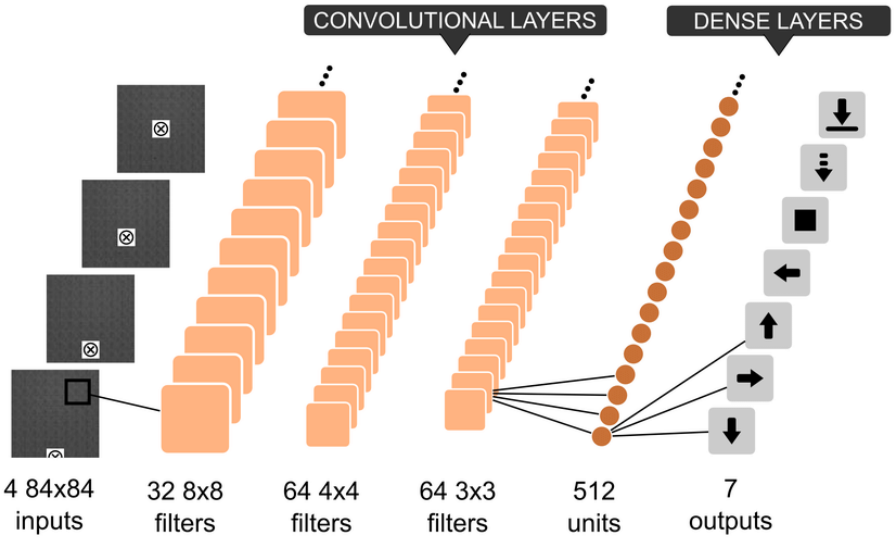



Figure 2.9: Deep Q-network architecture [23].

A Loss is calculated for back-propagating on the network and updating its parameters. The Loss Function is:

$$Loss(\boldsymbol{\theta}) = \frac{1}{N} \sum_i (Q(s_t, a_t; \theta)_i - target_i)^2 \tag{2.13}$$

where the target in DQN is the $TD(1)$ error:

$$target = r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta) \tag{2.14}$$

and is translated as the immediate reward added by an estimate of the following rewards.

**Algorithm 1** Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  **end for**
**end for**

Figure 2.10: Deep Q-network pseudocode [18].

A step-by-step representation of the algorithm is presented below:

1. Initialize a replay memory. A replay memory stores all the experiences in the form of transition tuples: $\langle \phi_t, a_t, r_t, \phi_{t+1} \rangle$
2. Start an episode and get the first state from the environment. Pre-process the state to a fixed length input vector.
3. Feed the input to the network and select the action $a$ that maximizes $Q(s, a)$ or with a probability $\epsilon$ select a random action in order to establish necessary exploration in the environment.
4. Perform the selected action and obtain the immediate reward and next state from the environment.
5. Store the transition tuple in the Replay Buffer
6. Every once in a while, randomly sample a mini-batch from the replay buffer
7. For all the transitions in the mini-batch compute the target values and perform a gradient descent on the Loss Function with respect to the weights of the network.
8. Repeat 3,4,5,7 for every time-step in the episode and 6 for every few timesteps.
9. Repeat over many episodes.

#### 2.2.3.2 Double Deep Q-networks

Double Deep Q-networks expand on DQN by implementing a secondary Q-network which acts as the target network i.e., a network that calculates the Q-values for the target values. This is done in order to address the DQN's problem of overestimating values [34, 36]. In short, DQN minimizes a loss function towards a target which is:

$$target_t = r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta) \tag{2.15}$$

The main problem with this approach is that the maximum target Q-value is always selected but that Q-value is an estimate produced by the same network that we try to optimize. If the network estimates Q-values with a hypothetical error e, then the maximum Q-value of the network will most likely be greater than the true maximum Q-value:

$$\max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta) + e > \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \tag{2.16}$$

As a result, the network is updated towards overestimated targets and learning is destabilized [34]. Double DQN aims to minimize the overestimation error by introducing a secondary network called a *target network* which is a copy of the *behavioral network*. The target network copies the behavioral network's parameters at a slower pace - usually every few training episodes. As a result, the target network produces more restrained Q-values and does not enforce every training error of the behavioral network.

**Algorithm 1 : Double Q-learning (Hasselt et al., 2015)**

Initialize primary network $Q_\theta$, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
**for** each iteration **do**
  **for** each environment step **do**
    Observe state $s_t$ and select $a_t \sim \pi(a_t, s_t)$
    Execute $a_t$ and observe next state $s_{t+1}$ and reward $r_t = R(s_t, a_t)$
    Store $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $\mathcal{D}$
  **for** each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Figure 2.11: Double DQN pseudocode [36].

#### 2.2.3.3 Recurrent Deep Q-Networks

As we have seen in 2.2.2, Recurrent Neural Networks are great for processing sequential data. In [11], recurrency was implemented as a way to integrate experiences from the past to partially observed states (as in a partially observed MPD (POMDP)). Specifically, instead of manually combining four sequential frames in order to unlock features such as direction or velocity, [11] uses a modified Q-network with an LSTM layer that, given a sequence of frames, can unlock these features by itself.

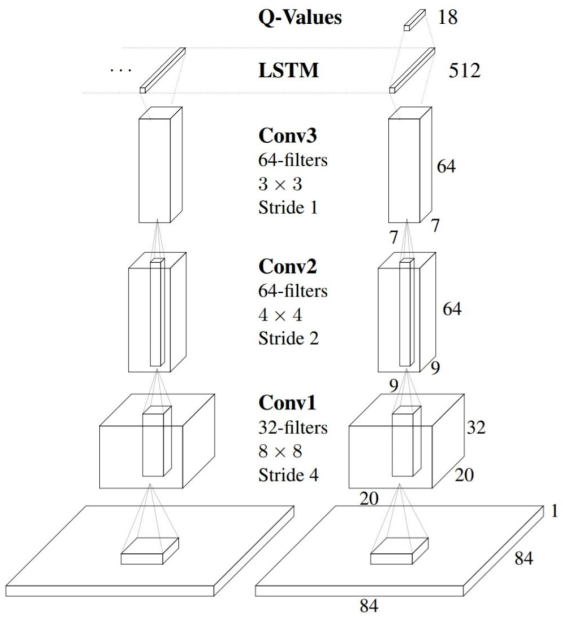



Figure 2.12: Recurrent Deep Q-Network architecture.[11]

The Q-network is modified by introducing an LSTM layer that replaces the first fully connected layer. The frame sequence gets processed by three convolutional layers and the output is being fed to the LSTM layer (see fig. 2.12). Then, the LSTM output gets through a final Linear layer that outputs Q-values. LSTM also requires an initial hidden state at each forward pass. When updating the Recurrent Network, [11] proposes two alternatives:

1. Sample whole episodes and carry the hidden state from one transition to the next
2. Randomly sample transitions from the Replay Buffer and set the initial hidden-state to zeros at each update.

Both alternatives are said to converge similarly, therefore [11] prefers the second approach as it is closely tied with the DQN's update strategy.

#### 2.2.3.4 Dueling Double Deep Q-networks

The Dueling Network architecture is an alternative way of estimating Q-values. Dueling methods primarily affect the architecture of the network and do not require modifications to the learning procedure. Hence, they can be combined with existing Deep Q-Network implementations [37]. In Dueling Network architectures, the Q-estimates are calculated by combining two separate networks: one that estimates the state value $V(s;\theta_1)$ and one that estimates an advantage function $A(s,a;\theta_2)$.

By definition, the advantage function is defined as to how preferable it is to select an action $a$ on a state $s$ compared to the state's value and is described as:

$$A(s,a) = Q(s,a) - V(s) \tag{2.17}$$

Expanding from the above equation for the dueling network architecture, one could estimate the state-action value estimates by:

$$Q(s,a;\theta_1,\theta_2) = V(s;\theta_1) + A(s,a;\theta_2) \tag{2.18}$$

However, this equation would require a lot of trust in both the state value and Advantage estimate. During training, however, these estimates are expected to be noisy and therefore an alternative, more stable method is used where:

$$Q(s,a;\theta_1,\theta_2) = V(s;\theta_1) + A(s,a;\theta_2) - Mean(A(s,a;\theta_2)) \tag{2.19}$$

This architecture formulated on the idea that Q-values can be estimated without having to directly estimate the value of each state-action pair [37]. Additionally, at each back-propagation, instead of updating the selected state-action values, the whole state value function is being updated.

#### 2.2.3.5 Prioritized Experience Replay

Prioritized Experience Replay involves sampling batches of experiences with a priority rather than sampling them randomly [28]. While the priority of a transition can be any attribute one wants to prioritize, [28] sets the priority of selecting actions as:

$$P_T = \delta(Q(s,a;\theta), target) = abs(Q(s,a;\theta) - [r + \max_{a}{}' Q(s',a';\theta_{target})]) \tag{2.20}$$

where in the case of DQN is the absolute TD Error of the experience. Each time a transition is added to the Replay Buffer it gets assigned to the maximum existing priority. This way it is ensured that new experiences will get sampled at least once when training and not get lost in the buffer. The probability of sampling a transition is defined as:

$$p_T = \frac{P_T^a}{\sum_n P_{T_n}^a} \tag{2.21}$$

where $P_T$ is the *priority* of the transition T and is divided by the cumulative sum of all the transition priorities in the buffer. The parameter $a$ is an indication of how much prioritization plays a role in the probability of selecting transitions. As we proceed in training, we want to minimize the parameter alpha since we want to more uniformly sample transitions to avoid overfitting our network to certain preferable trajectories [28].

Early in training, when sampling experiences with seemingly high priority, a bias is introduced in the loss function that can negatively affect the update of the network parameters. As a solution, [28] proposes that the loss will be multiplied by a weight parameter where:

$$W_T = (\frac{1}{N} * \frac{1}{p_T})^\beta \tag{2.22}$$

The parameter $\beta$ is slightly smaller than 1 and meant to be slowly increased to $\beta = 1$ as the loss function gets less biased when proceeding in training.

#### 2.2.3.6 Policy Gradient Optimization

Policy based methods focus on optimizing the policy directly instead of approximating *action-value functions*. Policy gradient methods with function approximators have been introduced as a mathematically proved alternative to the value based approximators that lack a theoretical proof [33]. Additionally, these methods optimize a *stochastic* policy which means that each action has a probability to be selected and can help enforce *exploration* while training.

Policy gradient optimization relies on two networks called the *actor* and the *critic*. The *actor* tries to *approximate* a policy i.e., the action probability distribution given a state observation as input. The critic, given the same input, approximates the state's value function. The actor is updated in correlation with the critic's estimates by performing gradient descent on the Loss:

$$L(\theta) = -E[\log(\pi_\theta(a_t|s_t))A_t] \tag{2.23}$$

where the loss is minimized by updating the actor's parameters in order to encourage actions that are associated with a positive advantage $A_t$ and discourage actions with negative $A_t$. The advantage function indicates how "advantageous" is to select a certain action in a given state and can ideally be described as:

$$A(s, a) = Q(s, a) - V(s) \tag{2.24}$$

In a reinforcement learning setting, however, the true $Q$ and $V$ values are not known. As a result, $Q$ can be estimated, for example by accumulating the subsequent rewards after selecting the action $a$ from the state $s$ and onwards. Additionally, the value $V$ is an estimate produced by the critic, resulting in the advantage function to be in the form of:

$$A_t = A(s_t, a_t; \theta) = R_t(\pi) - V(s_t; \phi) \tag{2.25}$$

Optimizing the critic is done by calculating the Mean Square Error between the value estimates and the targets which were gathered by collecting subsequent rewards.

#### 2.2.3.7 Proximal Policy Optimization

*Proximal Policy Optimization* (or *PPO*) is a *Policy Gradient* algorithm variant that has been demonstrating great performance in various robotic tasks as well as the Atari benchmark [30] and other more demanding environments like the game of Dota 2 [1]. PPO is considered as an *on-line* policy optimization algorithm. The agent learns directly from the observed experiences by following its own policy instead of relying on a replay buffer where experiences are gathered from different policies and sampled randomly. Additionally, it is more sample efficient than other policy gradient methods by training on the same batch of experiences multiple times (where each iteration is called an *epoch*) and finally it provides a simple method for controlling each policy update within reasonable limits [30]. Similar to other Policy Gradient methods, given a state as input, an *actor network* is being used for estimating action probabilities while a *critic network* estimates the state's value function. A combined Loss Function is used for updating the parameters of the networks and is defined as:

$$L(\theta) = L_{CLIP}(\theta) - c_1 L_{VF}(\theta) + c_2 S(\pi_\theta) \tag{2.26}$$

$\boldsymbol{L_{CLIP}}$ is referred to as surrogate loss objective and is defined as:

$$L_{CLIP}(\theta) = E[min(r_t(\theta)A_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)] \tag{2.27}$$

This loss objective is an expansion of the loss objective commonly used in policy gradient methods:

$$L_{PG}(\theta) = E[\log(\pi_\theta(a_t|s_t))A_t] \tag{2.28}$$

The main problem with policy gradient methods is that if multiple update steps are taken in the same set of experiences, we risk taking big policy updates and derailing from the desired target policy [30]. While there have been approaches in addressing this problem by introducing a Trust Region where policy updates are being performed within limits [29], they are costly and difficult to implement [30]. PPO uses the surrogate objective in 2.28 where the $\log(\pi_\theta)$ is replaced by a ratio:

$$r(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2.29}$$

The ratio is then *clipped* within the limits of a hyperparameter $\epsilon$ (commonly $\epsilon = 0.2$ or $\epsilon = 0.1$). This limits the update of the policy to smaller steps and allows for stability in learning.

$\boldsymbol{L_{VF}(\theta)}$ is the loss of the critic and is defined as the Mean Square Error of the predicted state values and the target values. The effect of the critic loss to the total loss is controlled by the hyperparameter $c_1$.

$\boldsymbol{S(\pi_\theta)}$ is the entropy bonus of the policy and expresses the level of exploration. It is controlled by a separate hyperparameter $c_2$ where $c_2$ is usually within 0.1 and 0.01.

Similar to the Policy Gradient methods, the *Advantage* is calculated by:

$$A_t = A(s_t, a_t; \theta) = R_t(\pi) - V(s_t; \theta) \tag{2.30}$$

The returns $R_t$ are calculated by gathering the subsequent rewards for $T$ timesteps. Then an advantage estimator is being used to calculate the advantage. There are two proposed advantage estimators used in PPO:

1. Finite Horizon Estimator (or FHE):

$$A_t = [r_t + \gamma r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(S_T)] - V(s_t) \tag{2.31}$$

2. Generalized Advantage Estimate (or GAE):

$$A_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{2.32}$$

where

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.33}$$

When training, a trajectory is sampled for $T$ timesteps. Then the advantages are being calculated for each transition in the trajectory batch. For $K$ epochs, a combined loss function is calculated and the actor-critic weights are updated through the *Adam* optimizer. After the learning step, the trajectory batch is discarded and the process is repeated for the next T timesteps.

**Algorithm 1** PPO, Actor-Critic Style

**for** iteration=$1, 2, \ldots$ **do**
  **for** actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  **end for**
  Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
**end for**

Figure 2.13: PPO pseudocode [30].

#### 2.2.3.8 Self-Imitation Learning

*Self-Imitation learning* (or *SIL*) focuses in accelerating learning by revisiting past positive experiences [20]. SIL does not act as an independent algorithm when training on an environment. Instead, it is attached to existing on-policy, actor-critic methods and acts as an added off-policy learning step to these methods. In the paper's implementation, Self-Imitation is attached to an *Advantage Actor-Critic* (or *A2C*) algorithm. The authors, however, also mention positive performance when combined with PPO methods.

Similar to existing actor-critic approaches, training begins by sampling the on-policy trajectories of a whole episode. Then, the *Monte Carlo Returns* are estimated for each transition of the trajectory:

$$R_t = \sum kn\gamma^{k-t} r_k \tag{2.34}$$

Learning begins at the end of each episode from the gathered experience. First, the A2C module is used for learning the on-policy samples. Then, instead of discarding the episode trajectories after the learning step, they are being stored in a separate Prioritized Replay Buffer. The Self-Imitation learning step begins by sampling mini-batches on multiple epochs from the Replay Buffer and off-policy updating the same actor-critic network from the loss:

$$L_{sil} = E_{s,a,R\epsilon D}[L_{sil}^{policy} + \beta L_{sil}^{value}] \tag{2.35}$$

where:

$$L_{sil}^{policy} = -\log \pi_\theta(a|s)(R - V_\theta(s))_+ \tag{2.36}$$

$$L_{sil}^{value} = \frac{1}{2} \|(R - V_\theta(s))_+\|^2 \tag{2.37}$$

The $(+)$ symbol indicates that only the positive difference between $R, V_\theta(s)$ is considered. If the difference is negative, then it is set to zero. This is the fundamental measure that SIL takes in order to update only on positive experiences. An experience is described as *positive*, when the agent's Returns on that state are greater than the critic's estimate. In other words, the Self-Imitation module guarantees that the agent re-adjusts its belief on states that it undervalued.

Since only positive experiences contribute to the Self-Imitation learning, a prioritized Replay Buffer is used to prioritize experiences with $R > V_\theta$. This measure assists in sampling more positive experiences for each batch.

**Algorithm 1** Actor-Critic with Self-Imitation Learning

Initialize parameter $\theta$
Initialize replay buffer $\mathcal{D} \leftarrow \emptyset$
Initialize episode buffer $\mathcal{E} \leftarrow \emptyset$
**for** each iteration **do**
  ***# Collect on-policy samples***
  **for** each step **do**
    Execute an action $s_t, a_t, r_t, s_{t+1} \sim \pi_\theta(a_t|s_t)$
    Store transition $\mathcal{E} \leftarrow \mathcal{E} \cup \{(s_t, a_t, r_t)\}$
  **end for**
  **if** $s_{t+1}$ is terminal **then**
    ***# Update replay buffer***
    Compute returns $R_t = \sum_k^\infty \gamma^{k-t} r_k$ for all $t$ in $\mathcal{E}$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, R_t)\}$ for all $t$ in $\mathcal{E}$
    Clear episode buffer $\mathcal{E} \leftarrow \emptyset$
  **end if**
  ***# Perform actor-critic using on-policy samples***
  $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}^{a2c}$
  ***# Perform self-imitation learning***
  **for** $m = 1$ to $M$ **do**
    Sample a mini-batch $\{(s, a, R)\}$ from $\mathcal{D}$
    $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}^{sil}$
  **end for**
**end for**

Figure 2.14: Self-Imitation Learning pseudocode [20].

## 2.3 Related Work

*Multi-Agent Systems* (or *MAS*) and *Reinforcement Learning* have been a long-researched subject over the last few decades [5, 3, 2]. In [35], *MAS* and *Machine Learning* is defined as multiple autonomous agents that directly learn from experience. The agents can be either *cooperative* or *adversary*, they might or might not communicate with each other, they might be oblivious to the existence of other agents in the environment and can have different levels of coordination requirements. There are many problems to be dealt with when dealing with multiple agents:

- It is difficult to define a clear objective while at the same time trying to distinguish the roles of each agent in order to reach that goal.
- Multi-Agent environments are non-stationary, as multiple agents constantly interact and alter the dynamics of these environments.
- Under the non-stationarity of such environments, the state observations are only partially observable.
- Credit assignment from a global reward function can be a difficult or sometimes impossible task. At the same time, rewards can be sparse or delayed as complex coordination is required for obtaining positive rewards.

When dealing with cooperative agents, [3] distinguishes three learning approaches. The first would be to extend the single-agent Reinforcement Learning methods to the multi-agent problem. In this setting, an agent learns to coexist instead of coordinating with other agents. This means that for the learning agent all the other agents represent a part of a non-stationary environment and the agent is not actively aware of the existence of other learning agents. This approach has proven to be sufficient to solve a range of multi-agent environment settings [31]. However, it struggles to deal with tasks that require an increased level of coordination. A second approach is to include a level of communication among the agents. This would allow the agents to share information about their observations or about their actions. Some of these methods pre-define the level of communication among the agents while others have attempted to let the agents develop their own communication protocols [8]. The third approach is to apply laws and conventions to the learners. This translates into applying constraints in the possible actions when in certain states etc. It is difficult, however, to clearly define laws for a given task [3].

Other approaches involve using *Centralized Learning and Decentralized Execution*. Using this technique, agents usually share a central state and take a joint action in the environment to obtain a global reward. When taking actions, however, they follow their own discrete policy. *Counterfactual Multi-Agent Policy Gradients* (or *COMA*) [7] optimizes a Q-function using a centralized critic and decentralized actor. The agent's policies are optimized by isolating each agent's action contribution to the global returns. Another approach, *QMIX*

[26], estimates a central Q-function by combining the separate Q-functions of each agent generated by their local observations.

*Imitation* in Reinforcement Learning has proven to be a great way for accelerating performance in various tasks. *Deep Q-learning from Demonstrations* (or *DQfD*) [12] aims to accelerate performance in the initial stages of training by using a small training data set obtained from a mentor. The agent initially trains on the mentor's experiences by combining TD and supervised losses and then continues by training on its own experiences. In a similar fashion, *Human Checkpoint Replay* [14], accelerates performance in sparse reward environments by keeping a set of checkpoints collected from humans as starting points when training. This way, instead of restarting at the beginning state of each training episode, the agent is dropped at a random checkpoint and has a chance of visiting positive states that would otherwise not be able to reach due to the sparse rewards of the environment. In the same paper, *Human Experience Replay* is used as an alternative way of enforcing exploration in the environments by training on both agent transition tuples and human extracted transition tuples.

Imitation has also received interest in Multi-Agent Reinforcement Learning environments. In the paper *Implicit Imitation in Multiagent Reinforcement Learning* [24], agents learn by observing other mentor agents in model-based environments. Specifically, an agent improves its value function by observing the state transitions of another mentor agent while at the same time learns about promising states that it might not have explored. This approach comes with its limitations, though, as only a single observer agent can exist in the environment and all the other agents can only act as mentors. In *Coordinated Multi-Agent Imitation Learning* [15], multiple agents learn to coordinate by observing demonstrations. This method works by attaching a model for approximating an encoded coordination mechanism by observing data from demonstrations. When learning, both the coordination model estimator and the separate agent policies are being optimized. The algorithm was tested in two separate environments, one was the famous predator-prey game and the other was a task of learning defense in a football game.

Flatland has also shown some academical interest. In [16], the authors won the 2020 Flatland challenge by combining various *multi-agent Path Finding* techniques such as *prioritized planning*, *large neighborhood search*, *safe interval path planning*, etc. The solution did not incorporate, however, any Reinforcement Learning methods. By only focusing in path finding techniques, the authors managed to achieve outstanding performance and could find collision-free paths for a large number of trains. The authors, also, acknowledge that Flatland was developed with the intent to enforce Reinforcement Learning techniques. They note, however, that optimization approaches can still outperform any preexisting Reinforcement Learning approaches.

In the original paper of Flatland, *Flatland-RL: Multi-agent reinforcement learning on trains* [19],various Reinforcement Learning methods have been applied in an environment setting similar to ours. Specifically, the authors have implemented standard RL algorithms

such as *PPO* and *Ape-X* as well as *Central Critic* and *Implicit Imitation* approaches. Additionally, they incorporated some techniques for improving performance such as *no-action cell skipping* and *action masking*. The cell skipping approach, which was also incorporated in our approach, allows for excluding the state transitions that the agent cannot perform any actions (i.e., straight line) from the episode history. Action masking was included as a way to assist the agents to only take valid actions in intersections. They ran the experiments on 5 agents in a 25x25 environment and trained the agents on 3 different training seeds. The evaluation was done on 50 episodes. Their experiments have resulted in various interesting conclusions. First, they disregard the previous preconceived benefits of the Central Critic approaches in Flatland as they could not prove any significant advantage over non-Central Critic approaches in PPO. Additionally, combining Imitation approaches with standard RL methods also lead to better overall performance. These methods, however, require an optimal *mentor* for the demonstrations.

# Chapter 3

# The FlatLand Environment

Over the last few years, railway infrastructures such as the *Swiss Federal Railway Company* (or SBB) have been continuously expanding due to the increasing transportation demands. As a result, the railway networks need to constantly adapt to changes in infrastructures, traffic and scheduling which on itself is a very labor-intensive task. This task belongs in the category of the Vehicle Rescheduling Problem (or VRSP) where different vehicles (or trains) need to have the capability to dynamically adjust their routes in a constantly changing environment.

FlatLand is a 2D grid simulation that mimics the dynamics of a railway network. It was developed as an open sandbox for testing and applying different Machine Learning methods in order to tackle the VRSP. The purpose of FlatLand is to train multiple agents (trains) in a randomly generated environment to reach their destination as fast as possible without blocking each other at intersections. The FlatLand library is a framework that allows us to run different Machine Learning methods to tackle the VRSP problem.

## 3.1 FlatLand environment overview

FlatLand is an adaptable 2D grid environment which is split into *cells*. The dimensions of the environment are defined as the number of cells in each dimension and each cell can hold a rail block, a destination block, an agent, an intersection or a surrounding part of the environment that is not part of the rail network.

Each cell which is part of a railway consists of a *transition map*. A transition map is a 4-bit representation of the four possible directions an agent can go to. A cell where more than one direction is available is called a *switch*.

There are eight possible transitions maps. Each of these transitions is represented as a one-hot 4-bit representation of the possible directions from the given cell. A visual representation of the possible transitions is given above. *Case 0* is simply a non-valid cell that is not part of the rail. *Case 1* is a straight line where no changes in direction are possible. *Case 2* represents a transition to the left of forward. *Case 3* is a crossing of two

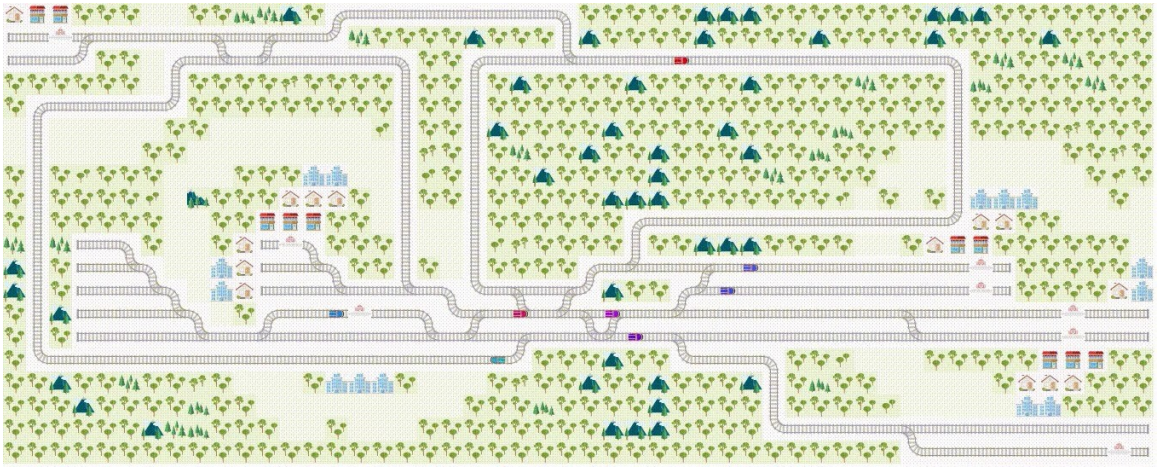


Figure 3.1: A visualisation of the FlatLand environment[19].

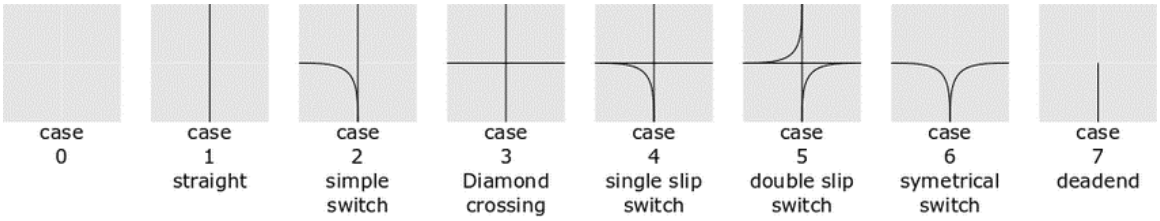



Figure 3.2: A representation of all the possible transition maps[19].

rails where no left or right transitions are possible. *Case 4 and 5 and 6* are combinations of the above. *Case 7* is a dead-end where the agent has to stop or move to the opposing direction.

This modularity of the environment allows for randomly generated worlds with predefined properties. For example, we can generate an environment with specific dimensions and number of agents but random rail structure. As for training, FlatLand is an episodic environment with finite time-steps where each episode is a randomly generated grid environment. At every time-step, all agents select an action simultaneously and the end of the episode is reached when all agents reach their destination or the max number of allowed time-steps is reached. Since multiple moving agents coexist in the environment, it is defined as a multi-agent dynamic (or non-stationary) environment. Additionally, FlatLand is partially observable, with sparse rewards and discrete actions.

## 3.2 Actions

There are five discrete actions:

- $DO - NOTHING$: Depending on the case, this action has three meanings. If the agent is already moving, it keeps moving. If the agent is stopped, it stays stopped. If it is at a dead-end (but not blocked with another agent) it turns around.
- $MOVE - LEFT$: Upon an intersection, the agent will move left if the intersection allows for a left turn. In the special case where the agent is stopped, this action will make it moving forward or left if left is allowed.
- $MOVE - FORWARD$: Upon an intersection, the agent will move forward. If the agent is stopped, this action will make it move forward.
- $MOVE - RIGHT$: Similar to the action $MOVE - LEFT$, but for the right direction.
- $STOP - MOVING$: This action will make the agent stop moving.

At any given moment an agent can select at most two actions. Additionally, we can see that an agent cannot move backwards with the exception of reaching a dead-end.

## 3.3 Observations

The observation of the agent is a partial representation of the environment state at each time-step. Three different observation styles are provided:

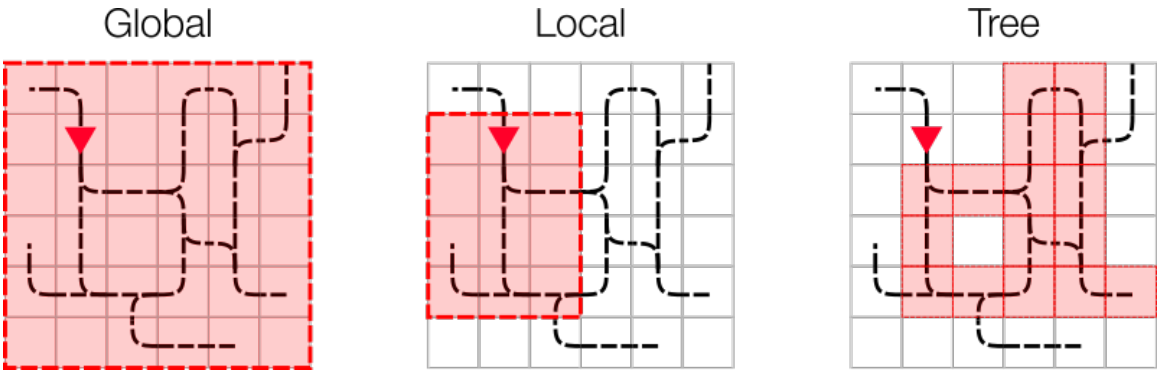



Figure 3.3: The three provided observations of FlatLand.

- Global Observation: The agent is provided with the full view of the grid-world. This observation style is very similar to the observation style in the Atari games [18]. The dimensions of the global observation are: $[h \times w \times c]$, where $h$ and $w$ are the height and width of the environment and $c$ are the number of channels. Each channel contains different information about the environment state such as agent positions and directions, agent targets etc. This observation method is not recommended as a good approach to solving the multi-Agent problem and is not preferred for this thesis.

- Local Observation: A local observation is similar to the global observation with the exception that the height and width of the observation is limited to a certain section of interest in the environment.
- Tree Observation: The tree observation works by observing the adjacent cells from the agent position. It works by expanding from the graph-like structure of the railway to all the possible transitions an agent can follow moving forward and storing the gathered information in nodes. The first node collects information from the agent's cell until a switch is reached. Then, from each switch direction begins a child node. Similar to the parent node, each child node gathers information from adjacent cells until another switch, dead-end or target is reached. The tree stops expanding when the tree depth limit is reached. Each node consists of 12 different features which include:
    1. Distance between the agent and its target, if the target is seen in the explored branch.
    2. Distance between the agent and another agent's target, if the agent's target is seen in the explored branch.
    3. Distance between the agent and another agent, if another agent is seen.
    4. Distance where a possible conflict might happen. This works by the use of a predictor that assumes each agent follows the shortest path to its target. This metric might be unreliable since it assumes the agents are ontological in the sense that they follow the best possible route at all times. Especially during training the usability of this feature is questionable.
    5. If an unusable switch is detected - the agent will have nowhere to go but agents from other directions will - we store the distance from that switch.
    6. Distance to the next node
    7. The remaining distance to the agent's target for every possible child node. This feature can help in deciding which direction is preferred based on the minimum distance to reach the target for each direction.
    8. Number of agents going in the same direction as the agent
    9. Number of agents going in the opposite direction
    10. If an agent with a malfunction is seen, count the timesteps until the malfunction ends.
    11. If an agent with a slower speed is observed in the same direction keep the speed.
    12. Number of agents that have not started playing.

The FlatLand library also allows for developing custom observations. This could allow for anyone interested to introduce new features to the observed nodes that could extract more information about the environment state. For example, different predictors could be used to predict possible conflicts or deadlocks. Developing a new observation that performs better than the provided Tree Observation however can be a difficult task on its own and is beyond the scope of this thesis.

## 3.4 Rewards

The reward signal that each agent i obtains at every timestep is defined as:

$$r_i(t) = r_l(t) + r_g(t) \tag{3.1}$$

where $r_l$ is the agent's local reward and is $-1$ for every timestep until it reaches its target location, where it gets 0. $r_g$ is a global reward and ate every timestep is equal to 0 unless all agents reach their goal where $r_g = 1$.

Under this reward function setting, the FlatLand environment is considered as a sparse reward environment meaning that the agents only get significant reward when reaching their target but not on the path until the target. This attribute of the environment needs to be considered when applying different RL methods in the later chapters.

# Chapter 4

# Our Approach

For this thesis, we systematically evaluate the performance of different value-based and policy-based Deep Reinforcement learning algorithms presented on chapter 2 on the multi-agent environment of FlatLand. We focus in scaling single agent Deep RL methods to this multi-agent environment with the end goal to monitor how well can these algorithms behave in a dynamic multi-agent environment with increasing coordination requirements.

In this chapter we begin by providing an overview of our observations about the environments dynamics and limitations that directly affected the algorithm decision process. Then, we discuss over various implementation details that are shared among the different methods. We continue, by bringing a top-level explanation of the algorithm implementations and modifications for our environment.

## 4.1 Overview

We begin by presenting some key features and our personal observations of the FlatLand environment that influenced our algorithm implementations.

- We do not acquire a model of the environment. This means that we are not aware of the *state space* nor the *state transitions*. Hence, any *Dynamic Programming* approach or algorithms that require the model of the environment (e.g., *AlphaZero*) are not suitable for training.
- The actions are discrete. Algorithms suitable for continuous control tasks are not considered (e.g., *DDPG [17]*). This does not eliminate, however, algorithms that are suitable for continuous and discrete control tasks.
- FlatLand offers *sparse* and *delayed* rewards. If we think of the reward model of FlatLand, agents only really receive a non-negative reward when reaching their target. The agents must take a sequence of good decisions before even seeing positive returns. Especially early in training, this is extremely rare.
- Agents receive *local* rewards.

- The agents (trains) are *cooperative*, *homogenous* and share a common *goal*.
- The state observation that is being used is the *Tree Observation* described in section 3.3 and is in the form of an *expanding tree* where features are detected from all the next possible paths up until a specified depth limit and then get flattened in a one-dimensional array.

Having these considered, we now provide a common ground for all of the following implementations. These are the main attributes that are shared among all the different implementations as a measure to have a fair and even evaluation and comparison.

- Among all implementations, we maintain the same number of fully connected hidden layers with identical length.
- At each implementation, all agents share the same network. We use the combined experiences of all the agents for training and the network is oblivious of whose input is processing.
- We limit the Tree Observation's depth and branches at the same lengths. All the algorithms process the same observation signal. Additionally, the observation's features are normalized and translated into a fixed size one-dimensional vector.

Additionally, when playing in the environment, there are many timesteps where an agent (train) cannot perform any action, i.e., in a straight line. The environment of FlatLand provides a built-in Boolean value at every timestep stating if an action is required. When an action is not required the agent does not act in the environment but instead performs a predefined action of $MOVE-FORWARD$. Additionally, the specific experience is not stored in the Replay Buffer and the experience is not considered when training the network. This method allows for better performance in value-based methods since we only train on experiences where the agent must learn to take good decisions.

In policy-based methods, however, this approach can lead to various complications. While in the one hand we do not need to act and improve in these states directly, the whole one-step transition cannot be ignored from the Episode Buffer. The reason is that we calculate the targets for training the network by accumulating the sequential rewards in the episode. Therefore, we need to be aware of the true episode history. We talk in more detail about our approach on dealing with these issues in the following segments.

## 4.2 Value-based methods

**DQN:** We begin with our implementation of the well-established DQN algorithm. This will be the foundation upon which we expand to with Double architectures, Dueling Double architectures etc. First, we initialize the NN architecture and Replay Buffer. As mentioned above, we use two fully connected linear layers with hidden size of 125. The output layer is

of size 5 - similar to the number of possible actions. Each fully connected layer's output is being processed through a *ReLu* activation function (see Figure 4.1).

Each agent shares the same network and Replay Buffer when training and when selecting actions. This implementation is cost and sample efficient since a single network is being trained from experiences sampled from multiple agents. We assign a max size for the Replay Buffer and store experiences until the buffer is filled. After the Replay Buffer is full, we add new experiences by replacing older ones.

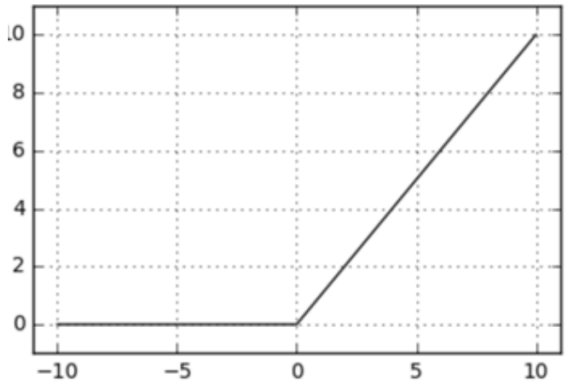



Figure 4.1: ReLu Activation.

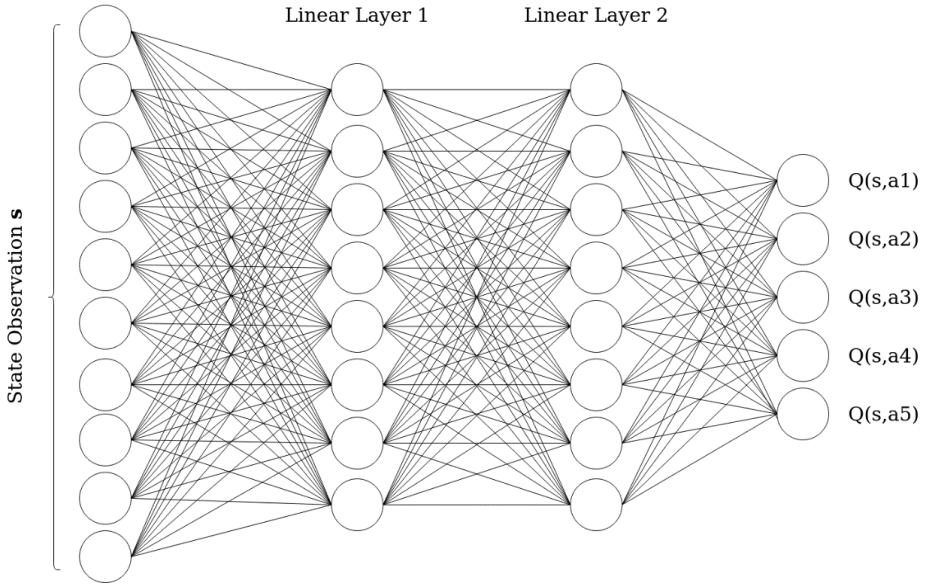



Figure 4.2: DQN Network architecture.

We begin each episode by resetting the environment and acquiring the first observation for each agent. Every time an observation is obtained, each of its features gets normalized in the range of $[-1, 0]$. The normalization step is necessary for the network to better

**Algorithm 1** DQN (IMPLEMENTED IN FLATLAND)

Initialize Q-Network $\theta$, Replay Buffer **R**
**for** every episode **do**
    **for** each environment step **do**
        **for** each agent i **do**
            Observe state $s_t^i$
            **if** action is required **then**
                With probability $\epsilon$ select random action $a$
                Else $a = \max_{a^i} Q(s_t^i, a^i; \theta_t)$
            Get $r_t^i, s_t^i$
            Store $\langle s_t^i, a_t^i, r_t^i, s_{t+1}^i \rangle$ in **R**
    **for** every few timesteps **do**
        Sample $\langle$**s, a, r, s'**$\rangle$ batch from **R**
        $target = \mathbf{r} + \gamma \max_{\mathbf{a'}} Q(\mathbf{s'}, \mathbf{a'}; \theta)$
        Minimize $M.S.E[Q(\mathbf{s}, \mathbf{a}; \theta), target]$
    Slightly Decrease $\epsilon$

approximate and process the inputs. At each timestep, agents select an $\epsilon$-greedy action by forward passing the observation to the network and selecting the action that gives the maximum Q-value. With $\epsilon$ probability, the agent chooses a random action instead. We start our training with a high $\epsilon$ probability ($\sim 0.9$) and we slowly decay the $\epsilon$ as the agent gets trained. This is done because we want to enforce high state exploration when we start training but slowly start exploiting our trained network as we progress. After selecting an action, we step on the environment and acquire the rewards, next states and completions of each agent. For each agent, we store in the Replay Buffer its transition tuple: $\langle s, r, a, s', d \rangle$ where d is a Boolean indicating if the agent has reached his goal.

Every few timesteps we train the network by randomly sampling a batch of transitions from the Replay Buffer. We optimize the Q-network by back propagating through the loss function which is the Mean Square Error of the $TD(1)$ error of Q values of the selected state action pairs and a target. The target is the immediate rewards from the states added by the max Q-value output of the next states multiplied by $\gamma$ (see Eq. 2.14). For updating the parameters, we chose the Adam optimizer.

**DDQN:** Expanding from DQN, we implement the Double DQN variant as mentioned in 2.2.3.2. Instead of updating the target network every few episodes, we perform a soft-update on the target network by slowly updating its parameters $\phi$ towards the policy network's parameters $\theta$ by a parameter $\tau$ as:

$$\phi \leftarrow (1 - \tau)\phi + \tau\theta \tag{4.1}$$

**Algorithm 2** DDQN (IMPLEMENTED IN FLATLAND)

Initialize Q-Network $\theta$, target Q-Network $\phi$
Replay Buffer **R**
**for** every episode **do**
    **for** each environment step **do**
        **for** each agent i **do**
            Observe state $s_t^i$
            **if** action is required **then**
                With probability $\epsilon$ select random action $a$
                Else $a = \max_{a^i} Q(s_t^i, a^i; \theta_t)$
            Get $r_t^i, s_t^i$
            Store $\langle s_t^i, a_t^i, r_t^i, s_{t+1}^i \rangle$ in **R**
    **for** every few timesteps **do**
        Sample $\langle$**s, a, r, s'**$\rangle$ batch from **R**
        $target = \mathbf{r} + \gamma \max_{\mathbf{a}}, Q(\mathbf{s'}, \mathbf{a'}; \phi)$
        Minimize $M.S.E[Q(\mathbf{s}, \mathbf{a}; \theta), target]$
    Slightly Decrease $\epsilon$
    Soft-update target network $\phi \leftarrow (1 - \tau)\phi + \tau\theta$

**DDDQN:** We continue by implementing a Dueling Network architecture as an alternative way of estimating Q-values. Instead of relying on a single network to estimate the state-action values, dueling architectures split the input state in two different networks: one that estimates the state value $V(s; \theta_1)$ and one that estimates an advantage function $A(s, a; \theta_2)$.

Both networks take the same input; the one which represents the state value estimate, outputs a single state value, while the other which represents the advantage function, outputs 5 different advantages (one for each action). The Q-values are estimated internally in the network's forward function with:

$$Q(s, a; \theta_1, \theta_2) = V(s; \theta_1) + A(s, a; \theta_2) - Mean(A(s, a; \theta_2)) \tag{4.2}$$

During back-propagation from the loss of the selected state-action pairs we inevitably back-propagate through Eq. 4.2 and both network parameters are being updated. Since these modifications only affect the Network architecture, the training approach is similar to the above implementations.

**DRQN:** We expand once again from the Dueling DDQN implementation by adding a recurrent layer in the network. In particular, our recurrent DDQN implementation involves modifying the above dueling architecture by replacing the first linear layer with an LSTM layer.

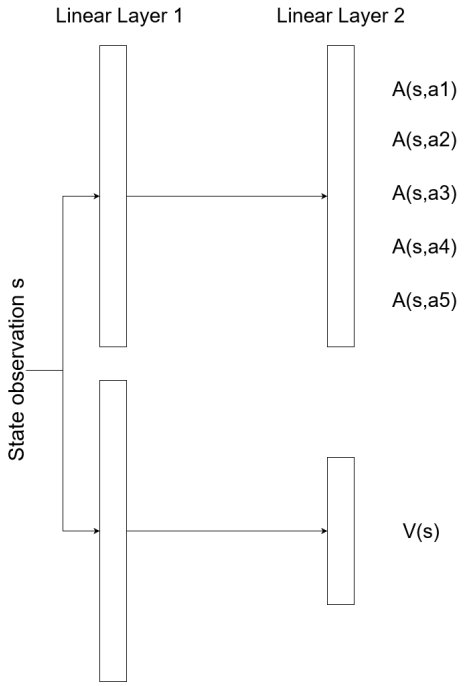



Figure 4.3: Dueling Networks architecture.

Instead of passing a single state to the network, we now concatenate a sequence of past states to our current state. If it is the beginning of the episode, we fill the sequence with zeros. After the LSTM layer, we isolate only the last output of the sequence (which is our current state) which then gets forwarded through the Linear Transformation layers. When forwarding a sequence to an LSTM layer, we also need to provide a hidden state. While there are two main approaches in the hidden state initialization [11], we initialize a zero-filled hidden state when collecting transitions (i.e., acting on the environment) but maintain the hidden state between acting steps. When training the sampled batches, we reset the hidden state to zeros and forward the batch through the network.

**Prioritized Experience Replay:** Instead of randomly selecting samples to fill our batches, we prefer samples that are prioritized. Our implementation involves attaching to the existing transition tuple $\langle s, r, a, s', d \rangle$ a secondary tuple containing three attributes: *priority*, *weight* and *probability*. These attributes were discussed in 2.2.3.5.

Each time a transition is added to the Replay Buffer it gets assigned to the maximum existing priority. That is because we want to ensure that new experiences will get sampled at least once when training and not get lost in the buffer.

When sampling mini-batches for training, instead of randomly sampling transitions, we sample them based on their probability. Each time we update our network from these mini-batches, we also update the priority of those transitions.

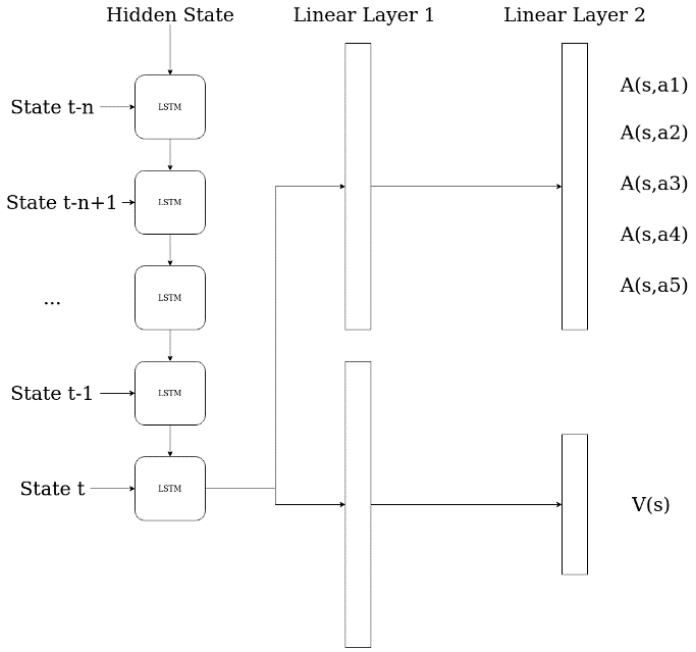



Figure 4.4: Dueling Network architecture with LSTM layer.

Every few episodes, we need to update the $\beta$ described in 2.2.3.5. This acts as a measure to control bias in the sampled experiences. The $\beta$ parameter is slowly being increased up until it reaches a value of 1.

Since we are dealing with big replay buffers, performance can be heavily affected every time we update the priorities, the parameters or we add experiences. When adding a new transition to the buffer for example, we need to assign it the maximum pre-existing priority and weight. If we recalculate the max priority and weight for every new experience, then we are dealing with running time of $O(n)$. As a counter measure, we store the maximum priorities and weights and only update them when new higher priorities are detected. Similarly, the cumulative sum of priorities is stored and updated whenever we remove, add or modify an experience and its priority.

## 4.3 Policy-based Methods

**PPO:** We have already provided the necessary theoretical background on PG methods and PPO. For our implementation, we adjusted the methods to our needs, for instance by modifying the way clipping operates, as we detail below.

We first define our Actor-Critic model. Similar to value-based methods, our actor consists of two fully connected hidden layers of size 125 each. Each layer's output is processed

by a *Tanh* activation function. Our output, which is equal to the number of possible actions, gets processed by a *SoftMax* activation function to transform the output to probabilities of selecting each action. Our critic, has identical hidden layers and Tanh activations but has a single output which estimates the state value function.

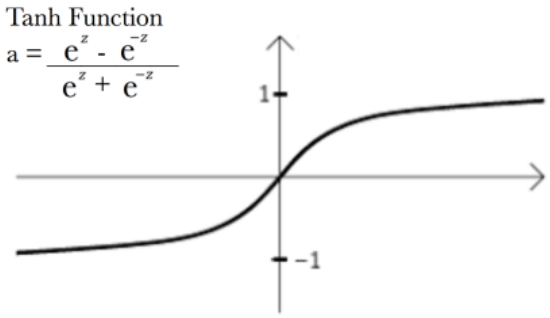



Figure 4.5: Tanh activation function.

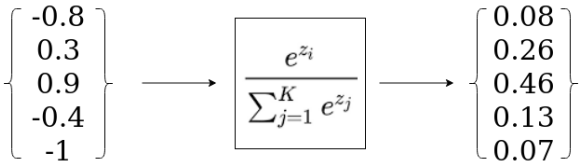



Figure 4.6: An example of a SoftMax activation on a set of numbers.

When training, we parallelly sample trajectories for every agent from the start till the end of an episode. Then, from the gathered rewards, we recursively accumulate them and calculate the *Returns* for each timestep in the episode. This is similar to the proposed *Finite Horizon Estimate* approach in [30]. The difference is that instead of sampling trajectories up until a *horizon* $T$, we set the horizon as the end of the episode. If the agent has not succeeded until the end of the episode, we estimate the remaining returns by forward passing his last observed state.

At the end of an episode, we train on the episode's experiences by performing multiple training steps (or *epochs*) on the episode's trajectories. Training multiple times on the same batch of experiences can be problematic because each episode can have great impact on the network's weights. PPO was developed as a way to avoid overfitting to each batch of experiences when training on the same batch for multiple epochs. As described in 2.2.3.7, *clipping* the actor network is primarily held responsible for this capability. In our implementation, however, we noticed that clipping the updates of the action log probabilities was not sufficient for stabilizing learning. By following the guidelines proposed by [6], we made these additions to PPO:

- Clipping the critic: Instead of updating the critic by back-propagating on the MSError of the critic's estimate and target values, the critic is clipped on each update similarly

to the actor:

$$L_{VF}(\theta_{new}) = \max[(V_{\theta_{new}} - R)^2, (clip(V_{\theta_{new}}, V_{\theta_{old}} - \epsilon, V_{\theta_{old}} + \epsilon) - R)^2] \quad (4.3)$$

Clipping both the actor and the critic allows for more controlled and restrained updates on their weights when training on the same batch of experiences.

- Normalizing the Returns: Since our Returns are the Monte Carlo sum of subsequent rewards at each episode's timestep, we expect the Returns to have a wide range of values (from 0 up to ∼90). Normalizing the Returns allows for more controlled weight updates when back-propagating on the Loss.
- Gradient Clipping: When calculating the gradients for the actor and critic network, the gradients are then clipped within a specified limit (0.5).

At each epoch, we start by first obtaining the actor and critic's estimates of the action distribution and state values respectively. We then continue, by estimating the *advantage* where $A = Returns - V_{critic}$. Then, we calculate the actor and critic loss and entropy as described in 2.2.3.7 and above. We then update the network *weights* and repeat the process for a specified number of epochs. Then the episode Trajectory is discarded and the process is repeated in the next episode.

At the beginning of this chapter, we mentioned on how some states are preferably ignored because the agent cannot act on them anyway. We also mentioned that while we do not to train our networks directly on these states, they cannot be excluded from the episode history. As a solution, we use all of the transitions for estimating the real Monte Carlo Returns for each state but mask out (set to zero) all of the 'no-action-required' transitions when training on the episode.

**PPOR (PPO with a Replay Buffer):** While training on PPO, we noticed that it was difficult for the agent to exceed mediocracy. While these experimental observations will be further described in the next Chapter, it was clear that the agent is closely tied to each episodic experience. In general, this is not a bad thing since an agent can theoretically extract most of the useful information an episode has to offer. In FlatLand's case, however, rewards are sparse and great experiences - or great episodes - are rare, especially when the agent is not trained to reach these great experiences. As a result, the agent learns mostly on mediocre episodes and optimizes his policy towards these mediocre episodes. This inevitably can lead to the agent being stuck in a local minimum of performance where he cannot escape.

As a direct solution to the problem, we added a Replay Buffer. At the end of each episode, instead of directly learning on the Episode's trajectories, we estimate the Returns and then add the Trajectory into a Replay Buffer. Each transition in the Replay Buffer holds a tuple: $\langle s, a, R, logp(a), V(s)\rangle$ where $R$ is the Return, $\log(p(a))$ is the log probability of the selected action and $V(s)$ is the critic's value estimate when the state $s$ was visited. At each *epoch*, we randomly sample a batch from the Replay Buffer and train similarly to PPO.

**Algorithm 3** PPO (IMPLEMENTED IN FLATLAND)

Orthogonal Initialize Actor, Critic
**for** every episode **do**
    Initialize Episode Buffer **E**
    **for** each environment step **do**
        **for** each agent i **do**
            Observe state $s_t^i$
            **if** action is required **then**
                Select action $a_t^i$ from $\pi_t(a_t^i|s_t^i;\theta)$
            Get $r_t^i, s_t^i$
        Parallel Store $\langle s_t, a_t, r_t, s_{t+1}\rangle$ of all agents in **E**
    Calculate returns **R**
    **for** each epoch **do**
        Get $\mathbf{S}, \mathbf{A}$ from **E**
        Get $\pi_\theta(\mathbf{A}|\mathbf{S};\theta), V_\theta(\mathbf{S})$
        Calculate advantage $A = R(\mathbf{S}) - V_\theta(\mathbf{S})$
        Maximize $L(\theta) = L_{CLIP}(\theta) - c_1 L_{VFCLIP}(\theta) + c_2 S(\pi_\theta)$

**SIL:** Similar to PPOR, at the end of an episode we store the trajectories and their Returns at a Replay Buffer. Then, for a specified number of *self-imitation epochs* we sample a batch of experiences from the Replay Buffer. We then isolate and update our actor-critic network only on the positive experiences by masking out all the negative experiences. By positive experiences, we mean the experiences where the returns $R(s)$ were greater than the critic's estimates $V_\phi(s)$. Similar to 2.2.3.8, we use a combined loss objective for updating both the actor and the critic's networks. Since we are only interested in the positive transition tuples, we use a Prioritized Replay Buffer and we set as the prioritization factor the positive experiences. This way, we become more efficient and ensure that each sampled batch contains more than average positive experiences from which we can train to.

**Algorithm 4** SIL (IMPLEMENTED IN FLATLAND)

Orthogonal Initialize Actor, Critic
Initialize Prioritized Replay Buffer **R**
**for** every episode **do**
    Initialize Episode Buffer **E**
    **for** each environment step **do**
        **for** each agent i **do**
            Observe state $s_t^i$
            **if** action is required **then**
                Select action $a_t^i$ from $\pi_t(a_t^i|s_t^i;\theta)$
            Get $r_t^i, s_t^i$
        Parallel Store $\langle s_t, a_t, r_t, s_{t+1}\rangle$ of all agents in **E**
    Calculate returns **R**
    **for** each epoch **do**
        Update actor-critic PPO style
    Store **E** in **R**
    **for** each SIL epoch **do**
        Get batch from Replay Buffer **R**
        Get $\pi_\theta(\mathbf{A}|\mathbf{S};\theta), V_\theta(\mathbf{S})$
        Isolate **S** where $R(\mathbf{S}) > V_\theta(\mathbf{S})$
        Minimize $L(\theta) = L_{sil}^{actor} + \beta L_{sil}^{critic}$

# Chapter 5

# Experimental Evaluation

In this chapter, we systematically evaluate and compare the performance of our Deep RL methods with respect to different metrics of performance. We ensure consistent conditions for training and evaluation among the different algorithms and train them on three levels of difficulty. We observe and discuss the experimental results, and analyze the best performing method for tackling the multi-agent task of FlatLand.

## 5.1 Environment Settings

We define three levels of difficulty in which the algorithms were trained and evaluated. As an

| | **Easy** | **Medium** | **Hard** |
|---|---|---|---|
| **Number of agents** | **3** | **5** | **7** |
| **Number of cities** | **2** | **2** | **3** |
| **Dimensions** | 25x25 | 25x25 | 25x25 |
| **Max Rails in City** | 3 | 3 | 3 |
| **Max Rails between Cities** | 2 | 2 | 2 |

Table 5.1: Parameters of the three levels of difficulty. Dimensions describe the dimensions of the grid-world, Max Rails in City describe the possible paths surrounding a city, Max Rails Between Cities describe the possible paths between cities.

attempt to only focus on the multi-agent problem we keep most of the level characteristics similar, but alter the number of agents. The criterion for selecting these three levels of difficulty was to distinguish the capabilities and limitations of each algorithm when the coordination complexities increase. Additionally, hardware limitations had to be considered, since adding more and more agents can heavily increase the time of training for each algorithm.

(a) Easy (b) Medium (c) Hard

Figure 5.1: Representation of the three levels of difficulty. We can clearly see how the attributes Max-rails-in-city and Max-rails-between-cities apply to each level.

## 5.2 Training and Evaluation Setup

When training, we expect algorithms to perform similarly on different training sessions. However, different Reinforcement Learning algorithms are known to be sensitive to different sequences of training episodes [4]. Hence, each algorithm is going to be trained on ten different *seeds*. Each *seed* produces a specific sequence of randomly generated episodes, therefore training these algorithms on the same ten seeds brings an even ground when training. Additionally, by training on ten seeds we can measure how sensitive or stable each algorithm is on different training runs. Each training run lasts 1500 episodes. At each episode we keep the Normalized Returns which is calculated by:

$$EpisodicReturn = \frac{sumofrewards}{numberofagents * maxepisodesteps} \tag{5.1}$$

where $maxepisodesteps$ is the max allowed episode steps and is calculated by:

$$maxepisodesteps = 4 * 2 * (height + width + \frac{Numofagents}{Numofcities}) \tag{5.2}$$

The $sumofrewards$ is the sum of the rewards of all the agents for all the timesteps of the episode until termination. In the worst case scenario, none of the agents terminates and the $sumofrewards$ is equal to the $numberofagents * maxepisodesteps$. Therefore, the Episodic Returns is a normalized metric that indicates how fast the agents have completed the episode and spans within the range of $[-1, 0)$. The worst possible result means that the agents failed to complete the episode after surpassing the max possible episode steps. When training the agents, an increase in the average Returns indicates that the agents complete the episode faster and collide less frequently.

Additionally, at each episode we keep the average Returns of the last 100 episodes. By keeping the average of the Returns, we can clearly observe the smoothed training curves

instead of the noisy normalized Returns. This can be greatly beneficial when trying to monitor the learning behaviors of different algorithms while at the same time visualizing accurately their learning curves.

For each algorithm, we select the hyperparameters that lead to the smoothest training curves i.e., smoothest learning, as a measure of guarantying stability and trust on the different training runs. The hyperparameter selection began by first isolating three of the most promising hyperparameter sets for each algorithm. Then, we compared the overall performance of the three different sets and kept the best performing hyperparameter set for each algorithm. We then run that hyperparameter set on the training procedure we mentioned above. The applied hyperparameters of each algorithm can be seen in Table 5.3

At the end of each training run we store the trained model's parameters (weights and biases) to be used later in evaluation. Since we trained on ten seeds, we end up with ten different trained models for each algorithm. Each training run, which has the length of 1500 episodes, is split into 15 evaluation points. Each point consists of the returns of 100 episodes. We compare the performance and reliability of the algorithms at each evaluation point with the methods mentioned below.

## 5.3 Performance and Reliability Metrics

The comparison of multi-agent Reinforcement Learning algorithms lacks a common and fair evaluation ground [22]. Therefore, [22] proposes a systematic comparison on different metrics of reliability and performance influenced by [4] but expanded on multi-agent RL. By following the guidelines of [22, 4], we compare the raw performance of the algorithms when training and when evaluating by calculating the below metrics:

**Average Returns during Training:** For each algorithm, we keep the average returns at each evaluation point during training. We then calculate and plot the median and 95% interval of those returns among the ten different seeds. All of the algorithms are plotted together for a direct comparison on the three levels of difficulty.

**Maximum Returns during Evaluation:** We evaluate the ten different trained models of each algorithm on a new evaluation seed. This seed has to be different than the training seeds as to ensure that the algorithms are evaluated in unknown sequence of environments. Each evaluation lasts 1000 episodes and we calculate the median and standard deviation of the average returns across the ten different trained models.

Apart from raw performance metrics, we also include reliability metrics proposed by [4] that give us information such as the stability or consistency of each algorithm during training or along multiple training runs. These metrics are usually ignored when comparing different Reinforcement Learning methods. However, aiming for reliable and steady learning during training can be a high-priority task for many real-world applications [4].

**Dispersion across Time (DT):** For this metric, we monitor the dispersion of the normalized average episodic Returns within each evaluation point when training. The dispersion is the Inter-Quartile Range (or IQR) of the returns within each evaluation point. We remind, that each evaluation point consists of 100 training episodes and that there are 15 evaluation points in total in each training run. Instead of displaying the actual dispersion at each evaluation point, [4] instructs displaying the metric in three zones of training: Beginning, Middle and End. Each zone consists, in our case, of 500 episodes (or 5 evaluation points). Hence, this metric is displaying the normalized average dispersion of each algorithm in the three training zones. The dispersion is normalized and is displayed as a ranking for each algorithm which means that higher ranking is better and translates into lower dispersion of returns. The ranking of each algorithm is averaged on all training runs (10 runs) and in all levels of difficulty. For separate rankings on each level of difficulty, see Appendix, Figure .1.

Additionally, we note that the vertical black lines (see Figure 5.3) display the 95% bootstrap confidence intervals and the horizontal black lines display the significant pairwise differences in ranking between a pair of algorithms. The line that refers to each pair spans from the one algorithm to the other.

**Dispersion across Runs (DR):** Here, we monitor the dispersion of the average Return at each evaluation point across the ten training runs. Similarly to DT, the metric is displaying the normalized average dispersion of each algorithm in three zones of training - Beginning, Middle, End - instead of displaying the dispersion at every single evaluation point. Additionally, this metric is also normalized and is displayed as a ranking where higher ranking is better and translates into lower dispersion of returns. This ranking is an average on all three levels of difficulty. For separate rankings on each level of difficulty, see Appendix, Figure .2. Additionally, the vertical black lines display the 95% bootstrap confidence intervals and the horizontal black lines across any two algorithms display the significant pairwise differences in the ranking of that pair of algorithms.

**Short-Term Risk across Time (SRT):** This metric represents the risk of the worst-case drops in episodic Returns from one evaluation point to the next. This metric is also displayed as a ranking between the algorithms and the resulting ranking is the average short-term risk of each algorithm across all seeds and all levels of difficulty. For separate rankings on each level of difficulty, see Appendix, Figure .3. Additionally, the vertical black lines display the 95% bootstrap confidence intervals and the horizontal black lines across any two algorithms display the significant pairwise differences in the ranking of that pair of algorithms.

**Long-Term Risk across Time (LRT):** This metric measures the long-term risk of sudden drops in the episodic Returns. While the SRT metric monitors sudden drops in performance in the short-term, LRT aims to capture whether an algorithm is bound to lose performance in the long-term. This metric can be really useful in cases where an algorithm might forget what it learned if it keeps playing long enough. Like SRT, this metric is also displayed as a ranking between the algorithms and the resulting ranking is the average long-term Risk

of each algorithm across all seeds and all levels of difficulty. For separate rankings on each level of difficulty, see Appendix, Figure .4. Additionally, the vertical black lines display the 95% bootstrap confidence intervals and the horizontal black lines across any two algorithms display the significant pairwise differences in the ranking of that pair of algorithms.

All of the performance and reliability metrics were captured by using the open-source tool provided by [4]. All of the reliability metrics (Dispersions, Risks) are represented by ranking the different algorithms side to side. A higher ranking of an algorithm is better and means lower risk or dispersion. The vertical black lines represent the 95% bootstrap confidence intervals (# of bootstraps = 1000). The horizontal black lines between any pair of algorithms shows that there is a statistically significant difference in their ranking.

## 5.4 Results

We will now provide the experimental results of the direct comparison in the three levels of difficulty of the following algorithms and variants:

- DQN
- Double DQN (DDQN)
- Dueling Double DQN (DDDQN)
- Proximal Policy Optimization (PPO)
- Proximal Policy Optimization with a Replay Buffer (PPOR)
- Self-Imitating agent with a Prioritized Replay Buffer (SIL)

As mentioned, the hyperparameters selected for each algorithm are presented in Table 5.3. These hyperparameters were chosen empirically. We excluded the recurrent DQN (or DRQN) variant from the final training comparison since it failed to provide any significant or noteworthy performance improvement compared to the other DQN variants. In fact it performed worce than most of the other value based variants while at the same time the LSTM layer heavily affected the time of execution.

**DQN:** As seen in Figure 5.2, DQN struggled to perform adequately in all three levels of difficulty during training. Due to a lack of a target network, DQN failed to keep consistency when training across multiple seeds and also could not converge to a sub-optimal or stable policy. Additionally, DQN performed poorly in most metrics of reliability. In the DR metric (see Figure 5.4), DQN performed only slightly better than the policy based methods in the Beginning stage of training. This slight performance advantage is explained by the fact that DQN fails to improve its episodic returns in the beginning stages of training, hence there is not a big dispersion to be seen. Similar behavior can be seen in the DT metric in Figure 5.3. The big instabilities of DQN can be clearly seen in the Middle and End stages of training, where DQN performed worse or equally as bad as PPO in both DT and DR metrics.

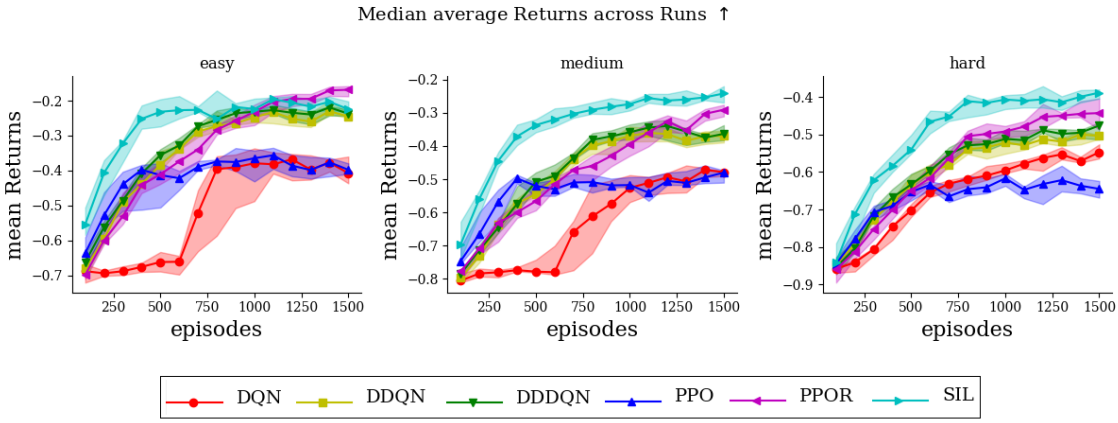



Figure 5.2: Median Average Normalized Returns across ten runs of all algorithms in all environments. Also showing the 95% interval across ten runs during training. The Returns indicate how fast the agents have completed each episode. Optimal agents would score on average −0.1.

Additionally, in figures 5.5 and 5.6 we can see that DQN is more prone to short-term and long-term sudden drops in performance than most of the other algorithms.

**DDQN:** Double DQN has shown smoother and greater levels of performance than DQN in all levels of difficulty. First, as seen in the training curves in Figure 5.2, DDQN maintained similar training curves along all training runs in all levels of difficulty. In the first level, it converged to a sub-optimal policy quite early in training (around episode 800) and maintained similar level of performance until the end of all training runs. In the medium level, it followed a similar training curve and maintained adequate performance throughout training. In the third level, DDQN slowly improved its performance until the end of the training session, but only achieved moderate results.

Additionally, DDQN maintained a high ranking in all metrics of reliability. When comparing the DR metric (see Figure 5.4), it ranks a lot higher than DQN, showing that adding a target network can not only improve learning, but also maintain consistency across training runs. DDQN also performs reasonably in the DT metric (see Figure 5.3). Also, it maintains a high rank in both short-term and long-term risk across Time metrics (Figures 5.5, 5.6), indicating that the agent is not prone to sudden drops in performance or suddenly forgetting what it learned.

**DDDQN:** Swapping the Q-network to a Dueling architecture has also led to a greater performing agent than DQN and also slightly better performing agent than DDQN, especially in the harder tasks. Since we are indirectly estimating Q-values by directly estimating state values at each batch update, we end up with a more frequent state value estimate update and as a result, we can better understand the environment dynamics. This can be especially useful in the more difficult tasks, where the state space is substantially increased and the

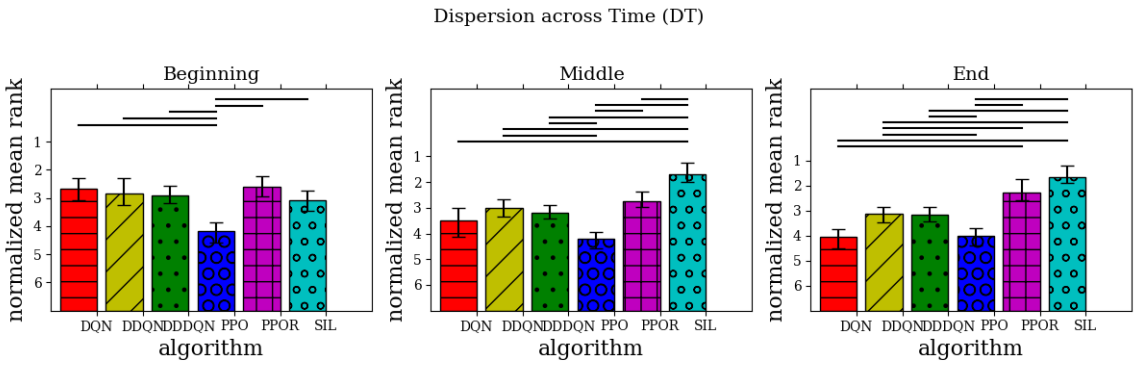



Figure 5.3: Ranking of all the algorithms in Dispersion across Time. Higher is better. The rank is averaged on all training runs and all levels of difficulty and is split into three zones (Beginning, Middle, End). These zones represent the ranking of Dispersion in the Beginning, Middle and Ending zones of training. When examining the Dispersion, we primarily focus on the results at the Middle or End, since the Beginning of training is always noisy in most algorithms. Error bars are 95% bootstrap confidence intervals. Horizontal bars display significant pairwise differences. See Appendix, Figure .1 for seperate rankings on each level of difficulty.

good state transitions can become sparse. As shown in Figure 5.2, Dueling DDQN followed a similar training curve to DDQN in all tasks, but achieved greater end results in the medium and hard level of difficulty. Additionally, DDDQN scored similar rankings of reliability to DDQN, with the exception of a greater ranking in the Dispersion of Returns across runs (Figure 5.4) and a slightly greater ranking in the Long-term risk across Time (Figure 5.6).

**PPO:** Proximal Policy Optimization has performed poorly in all levels of difficulty. While it did learn faster than the value-based methods in the first episodes during training (first 300 episodes), it failed to maintain a positive learning trend throughout training (see Figure 5.2). In all of the tasks, PPO appears to be stuck in a local minimum, and is even outperformed by DQN in the third level. Also, the noisy 95 percentile hints that there is a lot of variability in its performance across multiple runs and this can be confirmed in the DR metric (Figure 5.4), where PPO seemed to perform significantly lower than most of the other methods in the Ending stages of training. Surprisingly enough, PPO seemed to rank the highest in the Middle stage of training in the medium environment setting (see Appendix, Figure .2). We believe that this might be linked to the fact that PPO is the only algorithm that does not improve between the episodes 500 and 1000 in the medium setting (see Figure 5.2). As for the DT metric, PPO's lower performance was statistically significant between most of the other algorithms (Figure 5.3). Lastly, PPO seems to be more prone to short-term and long-term risks compared to the other policy variants as well as DDQN and DDDQN (Figures 5.5, 5.6).

During experimentation, the poor performance of PPO was really troubling at first. Due to its nature, many modifications can be applied in how the episode batches are collected, or how the Advantages are estimated etc. Initially, the poor performance was thought to be

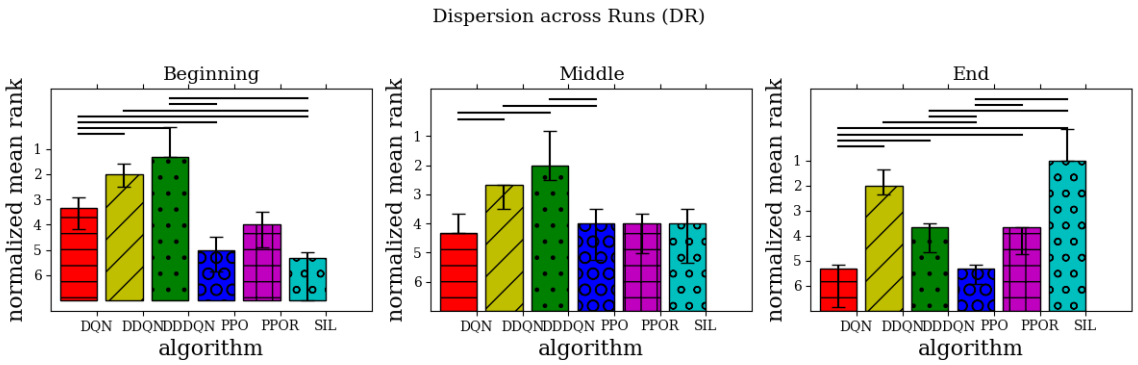



Figure 5.4: Ranking of all the algorithms in the Dispersion of Returns across all Runs and averaged on all levels of difficulty. Higher is better. Again, we primarily focus on the Middle and End zones of training. Error bars are 95% bootstrap confidence intervals. Horizontal bars display significant pairwise differences. See Appendix, Figure .2 for seperate rankings on each level of difficulty.

linked to not having found the optimal modifications or hyperparameter sets. However, after much experimentation, PPO always seemed to follow similar training curves regardless of the sampling or training strategy. The true reason behind its poor performance is thought to be that after learning on the episode Trajectories, the Trajectories are being discarded. In an environment like Flatland, where the rewards are sparse and good state transitions are rare, good episodes are not common. As a result, the agent updates its policy towards mostly mediocre play and the rare positive experiences are significantly outnumbered. At the same time a mediocre policy does not allow the agent to further improve and this leads to the agent being stuck in a local minimum of performance. Additionally, the agent is very sensitive to the sequence of the randomly generated episodes (seed). In Figure 5.4 we can see that PPO mostly scores the lowest when considering the Dispersion of Returns across multiple runs.

**PPOR:** Proximal Policy Optimization with a Replay Buffer has proven to be a key upgrade for unlocking the potential of PPO. In all three settings, PPOR, initially improves slower than most of its competitors. However, it maintains a steady and smooth learning curve up until the end of all training sessions. On the first level, PPOR eventually surpasses all of the other algorithms achieving an impressive sub-optimal policy. In the medium and hard levels, PPOR eventually surpasses all of the value-based methods and is only surpassed by SIL.

Additionally, PPOR scores greater than PPO in all metrics of reliability. Specifically, it achieves lower dispersion of rewards along multiple runs (see Figure 5.4) and also ranks higher in the short-term (Figure 5.5) and long-term risks (Figure 5.6). Also, it scored significantly higher than PPO as well as all the other value based methods in the Middle and End stages of the DT metric (see Figure 5.3). Considering these metrics, as well as its performance during training, we can confirm our assumptions of the poor performance of PPO. Storing the trajectories in a Replay Buffer allows for the rare positive experiences to be revisited

later in training and can help the agent to escape from mediocre policies. Additionally, the agent is not directly affected by each episode in the episode sequence, since it learns on randomly selected batches from recent experiences.

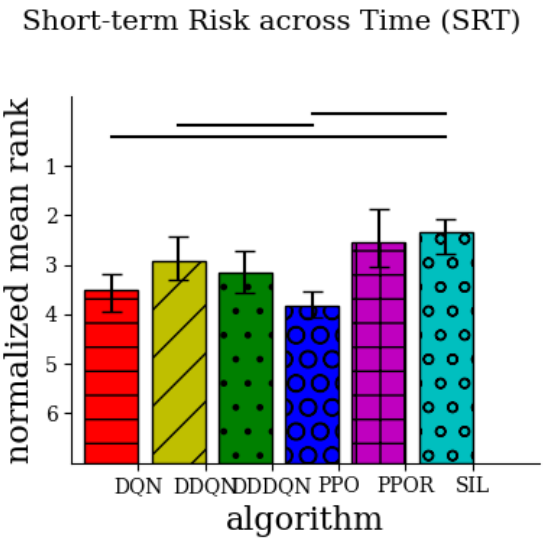



Figure 5.5: Ranking of all the algorithms in the short-term Risk acros time and averaged over all evaluation points and all levels of difficulty. Higher is better. Error bars are 95% bootstrap confidence intervals. Horizontal bars display significant pairwise differences. See Appendix, Figure .3 for seperate rankings on each level of difficulty.

**SIL:** Self-Imitation learning achieved outstanding performance in all levels of difficulty. First, SIL converged faster than all of its competitors and maintained similar convergence rates across all runs (see Figure 5.2). In the easy level of difficulty, it was able to converge faster to a sub-optimal policy and was only outperformed by a small margin by PPOR in the later stages of training. In the medium level of difficulty, SIL surpassed all of its competitors right from the beginning and successfully maintained a high level of performance until the end of training. In the last level of difficulty, SIL keeps the same outstanding level of performance and maintains an upwards training curve throughout training. Moreover, SIL ranked the highest in both the DT and DR (Figures 5.3, 5.4) metrics in the Middle and End stages of training. Its lower ranking in the Early stages is believed to be linked to the fact that SIL has the sharpest upwards training curve in the first 500 episodes (see Figure 5.2). As for the SRT and LRT metrics (Figures 5.5, 5.6), SIL seems to be on the same rankings as PPOR, DDQN and DDDQN and only ranks significantly higher than DQN and PPO.

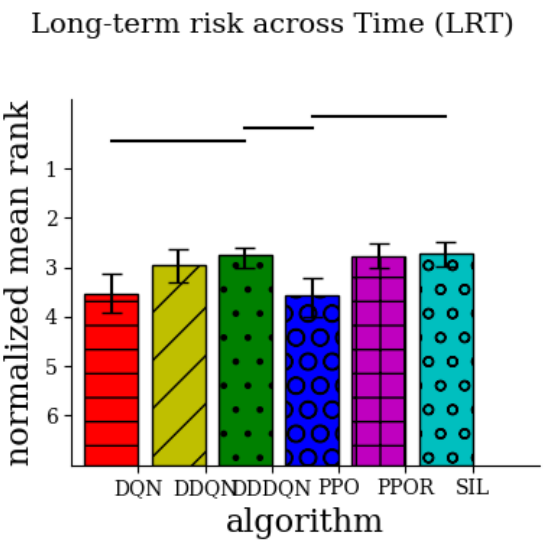



Figure 5.6: Ranking of all the algorithms in the long-term Risk acros time and averaged over all evaluation points and all levels of difficulty. Higher is better. Error bars are 95% bootstrap confidence intervals. Horizontal bars display significant pairwise differences. See Appendix, Figure .4 for seperate rankings on each level of difficulty.

| | DQN | DDQN | DDDQN | PPO | PPOR | SIL |
|---|---|---|---|---|---|---|
| Easy | -0.44±0.083 | -0.26±0.016 | -0.27±0.022 | -0.42±0.88 | **-0.18±0.14** | -0.20±0.024 |
| Medium | -0.49±0.026 | -0.37±0.027 | -0.35±0.015 | -0.50±0.027 | -0.30±0.018 | **-0.26±0.006** |
| Hard | -0.55±0.076 | -0.49±0.029 | -0.48±0.034 | -0.61±0.035 | -0.43±0.015 | **-0.38±0.017** |

Table 5.2: Evaluation of all algorithms on three levels of difficulty. Results are the average and standard deviation of Average Returns over 1000 episodes on ten agents trained on ten different seeds during training and evaluated under the same evaluation seed.

## 5.5 Discussion

In this section, we will provide intuitions and discuss over the lessons that we learned from the performance of our algorithms.

**Value-based methods:** With the exception of DQN, the value-based methods have generally resulted in steady and low risk training in all the environments. However, as the difficulty of the environment increases, none of these approaches achieved greater than adequate

performance. Taken from our experimental observations, we think this is traceable to the sparse reward nature of Flatland. As the coordination requirements increase, all of the agents need to take a sequence of good actions in order to explore better (or terminal) states. Value-based methods, however, rely only on the $\epsilon$-greedy approach as a measure of exploring such states. It is clear that it can be extremely difficult and rare for the agents to randomly take actions that can lead to such states and yield positive rewards. As a result, such agents usually train and lock towards mediocre policies.

When considering the reliability metrics, value-based methods have indicated in general higher levels of consistency when running on different environment seeds. We think this is due to the fact that they learn from randomly sampled batches from random episodes. Each batch can contain Transitions from older episodes or from any agent and therefore each network update is not directly affected from the latest sequence of episodes.

The implementation of such methods is also considered to be more straightforward. Since the network is indifferent to when or how the transitions are collected, the learning and acting steps are decoupled. This allows for a great level of freedom when considering the frequency of learning and also on how the Replay Buffer is filled. In Flatland specifically, if we did not consider a Transition important, we could just exclude it from the Replay Buffer. We also did not have to specify whom agent transitions we are storing.

**Policy-based methods:** Policy-based methods have provided mixed levels of performance and reliability across all tasks. Standard PPO, as mentioned and above, has really struggled to learn above-average policies. Our observations have shown that the poor performance is linked to the rarity of good episodes due to the sparse rewards of the environment. To better understand the problem, we can think of what happens when we train a PPO agent. The episodes start playing out by each agent following a joint untrained policy. The policy is mostly a random selection of actions; therefore, the episode outcome is mostly not great. The actor – critic then is trained on a mediocre episode. Over time, some episodes lead to slightly better results and the policy starts to slowly build higher confidence towards the actions that led to these results. At that point, some agents have probably mostly understood some basic behaviors, like reaching to their destination for example. If they want, however, to avoid blocking or to allow other agents to move to their goal they need to take highly coordinated sequence of actions. These behaviors can be really rare when using a mostly untrained policy but also can be rarer when the policy is trained towards avoiding these actions in favor of simpler and more common actions. Since these episodes are significantly outnumbered by average-performing episodes, the policy is mostly optimized towards not excellent behaviors. When good episodes do happen, the actor-critic usually does not have enough time to adjust itself towards enforcing the good decisions taken in that episode.

Additionally, PPO introduced more complexity in the implementation when compared to value-based methods. This is mostly due to the fact that PPO allows for a lot of variability in how the experiences are collected or sampled, or how the Returns and Advantages are

estimated, etc. When scaled to multiple agents more things need to be considered such as whether to parallelly update the experiences of all agents, or one by one etc.

**Adding a Replay Buffer to PPO:** As observed from the experimental results, a Replay Buffer seems to be able to alleviate PPO from the problems mentioned above. Our reasoning behind this substantial performance increase is that the Replay Buffer allows for the rare positive experiences to be revisited in the future. It needs to be noted that PPOR updates the actor-critic network in an almost identical fashion to PPO with the exception that instead of updating the episode trajectories at the end of an episode for multiple epochs, we instead randomly select for each epoch a batch of experiences from a Replay Buffer. The size of the Replay Buffer can also play a substantial role in the performance of PPOR. Value-based methods work great with Replay Buffers because we only need to store the one-step transitions to the Replay Buffer. The target for the network updates can be directly estimated from the one-step transitions (TD(1)) therefore the Q-network estimates at the moment of experiencing these transitions does not affect the network updates when these experiences are sampled. Here, however, we require the Monte Carlo Returns for updating the actor-critic network. These Returns are gathered by following the policy at that episode. Therefore, when sampling random batches of transitions, we do not want to sample really old experiences because these experiences are accompanied by Returns gathered from an older policy.

**Self-Imitation learning:** Self-Imitation has provided the most promising results when compared to all the other methods. Based on these results and our own observations, it seems that giving extra focus on the positive experiences can have a great impact on sparse reward environments. Especially when those experiences become rarer and rarer (medium, hard level of difficulty) self-imitation really prevails over all the other methods. Additionally, by imitating their own experiences and not these of a mentor, SIL agents are a lot easier to implement, since there is no need to construct a heuristic or optimal mentor-agent. Lastly, SIL also maintains the sample efficiency of PPO, since it uses the same experiences that PPO collected but just isolates the positive ones.

| | Learning rate | First Hidden | Second Hidden | Batch size | Learning freq. | Buffer size | epochs |
|---|---|---|---|---|---|---|---|
| | | | Easy | | | | |
| DQN | 0.00009 | 128 | 128 | 128 | 30 | 50000 | - |
| DDQN | 0.0003 | 128 | 128 | 150 | 30 | 50000 | - |
| DDDQN | 0.0003 | 128 | 128 | 150 | 30 | 50000 | - |
| PPO | 0.0005 | 128 | 128 | - | end | - | 10 |
| PPOR | 0.0005 | 128 | 128 | 300 | end | 10000 | 10 |
| SIL | 0.0005 | 128 | 128 | 300 | end | 20000 | 10/10 |
| | | | Medium | | | | |
| DQN | 0.00009 | 128 | 128 | 128 | 30 | 50000 | - |
| DDQN | 0.0003 | 128 | 128 | 150 | 40 | 50000 | - |
| DDDQN | 0.0003 | 128 | 128 | 150 | 40 | 50000 | - |
| PPO | 0.0005 | 128 | 128 | - | end | - | 10 |
| PPOR | 0.0004 | 128 | 128 | 300 | end | 10000 | 10 |
| SIL | 0.0005 | 128 | 128 | 300 | end | 30000 | 10/10 |
| | | | Hard | | | | |
| DQN | 0.00009 | 128 | 128 | 128 | 30 | 50000 | - |
| DDQN | 0.0003 | 128 | 128 | 150 | 40 | 50000 | - |
| DDDQN | 0.0003 | 128 | 128 | 150 | 30 | 50000 | - |
| PPO | 0.0005 | 128 | 128 | - | end | - | 15 |
| PPOR | 0.0004 | 128 | 128 | 300 | end | 10000 | 15 |
| SIL | 0.0005 | 128 | 128 | 300 | end | 40000 | 10/13 |

Table 5.3: Hyperparameters of the trained algorithms.

# Chapter 6

# Conclusions and Future Work

In this thesis we tackled the multi-agent Flatland environment by implementing and modifying various Deep Reinforcement Learning methods. We then systematically evaluated and compared the applied methods in consisted training conditions. We monitored the algorithm performances on different metrics of performance and reliability and we discussed the strengths and weaknesses of each algorithm. Then, we concluded by openly discussing our observations on the experimental results.

Our work began with the algorithm selection process which was driven by the dynamics and characteristics of the Flatland environment. After selecting suitable and promising candidates, we split them in value-based and policy-based methods. We then adapted these methods for the multi-agent Flatland environment. The implementation of the algorithms came with some difficulties. While value-based methods were for the most part flexible and easy to implement, policy-based methods required plenty of modifications and we also took some liberties in the episode sampling and training process. Specifically when working with PPO, we added some key extensions in order to better control the update steps of the actor-critic network and we trained on whole episode sequences instead of batches limited to a horizon T. Additionally, we integrated a Replay Buffer to a PPO module and demonstrated its superiority over PPO in a sparse reward environment like FlatLand. Finally, we implemented a Self-Imitating agent who remembers and revisits its past positive experiences and we showed that such method can result in impressive performance even without the need of a *mentor* agent.

We compared and evaluated the different algorithms in three levels of difficulty. For each level of difficulty, we guaranteed consistent training and evaluation conditions. We monitored the experimental results on various metrics of performance and reliability. We gave a lot of emphasis on the importance of consistency and fairness when comparing different algorithms on any environment as well as the importance of also considering the reliability metrics when examining the performance of such algorithms.

Our experiments have shown that value-based methods perform great in settings with small to medium coordination requirements, under the condition that a target network is being used. Standard DQN failed to perform great and consistently, while the Double and Dueling

Double DQN variants provided very stable and reliable results. As the complexity of the environments increased, however, none of the value-based methods exceeded mediocracy.

Policy-based methods led to mixed but overall, very positive results. Standard PPO struggled in all three levels. Our observations have linked its poor performance to the fact that PPO trains on-policy on each episodic trajectory, leading to mediocre policies. We alleviated the problem by instead sampling batches from a relatively small replay buffer and called the variant PPOR. PPOR surpassed all of the value-based methods as well as PPO in raw performance, while also maintained great levels of reliability. Our best performing policy-based method and also best overall method was the Self-Imitation method. By revisiting the positive experiences of previous episodes, SIL led to a successfull policy in all levels of difficulty while also scoring high in all levels of reliability.

FlatLand is an exciting platform that has the potential to accelerate research in multi-agent Reinforcement Learning. It combines many of the challenges multi-agent systems face and at the same time its scalable difficulty makes it accessible to all levels of academical research. As such, a lot can be done to extend our work. By utilizing the same comparison and evaluation ground, more algorithms and variants can be applied and evaluated. Specifically, one can examine various centralized critic methods or other imitation methods. One could also experiment with different network architectures, such as the use of recurrent layers or Transformers, or with different methods to enforce deeper state exploration. Additionally, a lot of work can be done by experimenting with different reward signals as well as different state observations.

# Appendix

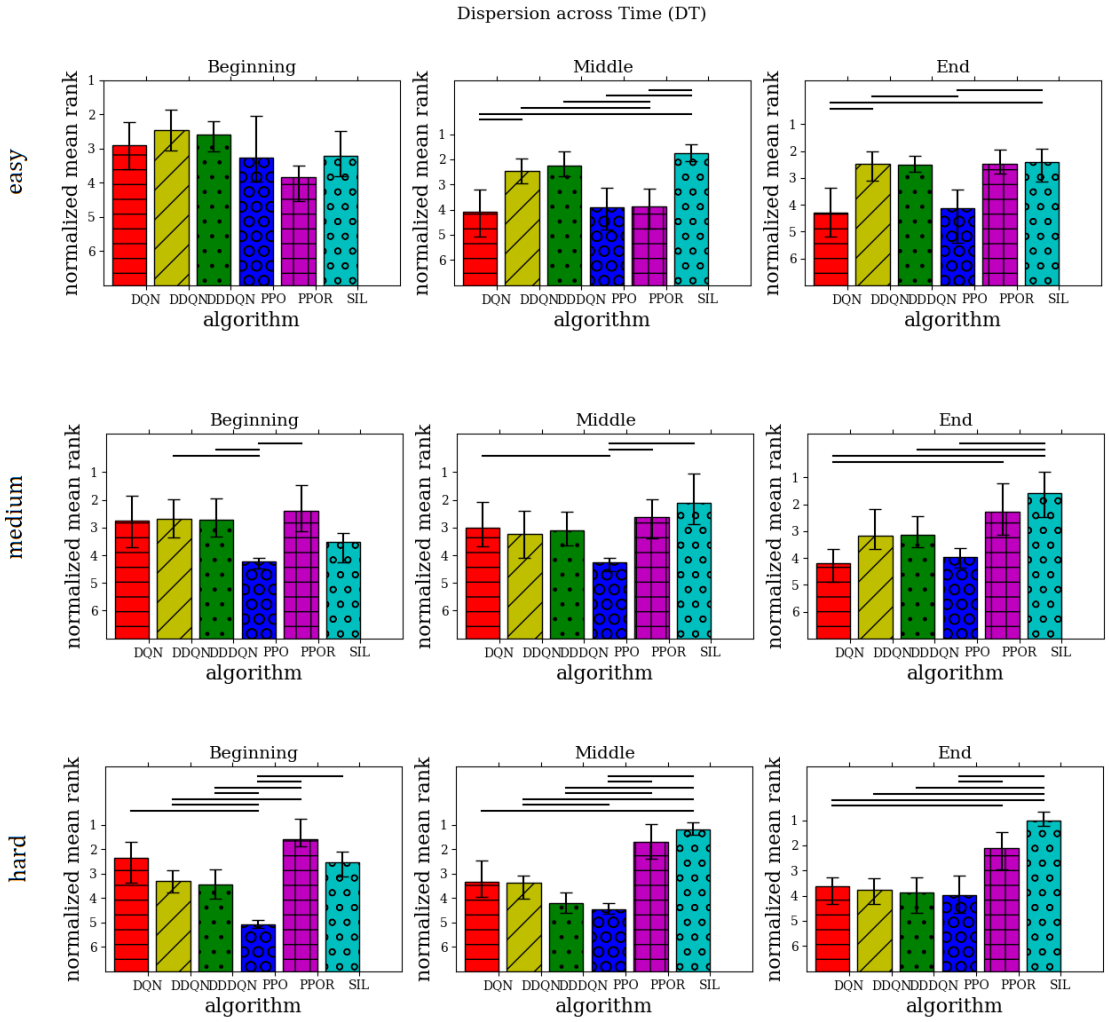



Figure .1: Dispersion across time displayed seperately in the three levels of difficulty. Each training zone displays a normalized ranking of each algorithm in that range, averaged over ten training sessions. Black error bars display 95% bootstrap confidence interval error (# bootstraps = 1000). Horizontal bars display significant pairwise differences.

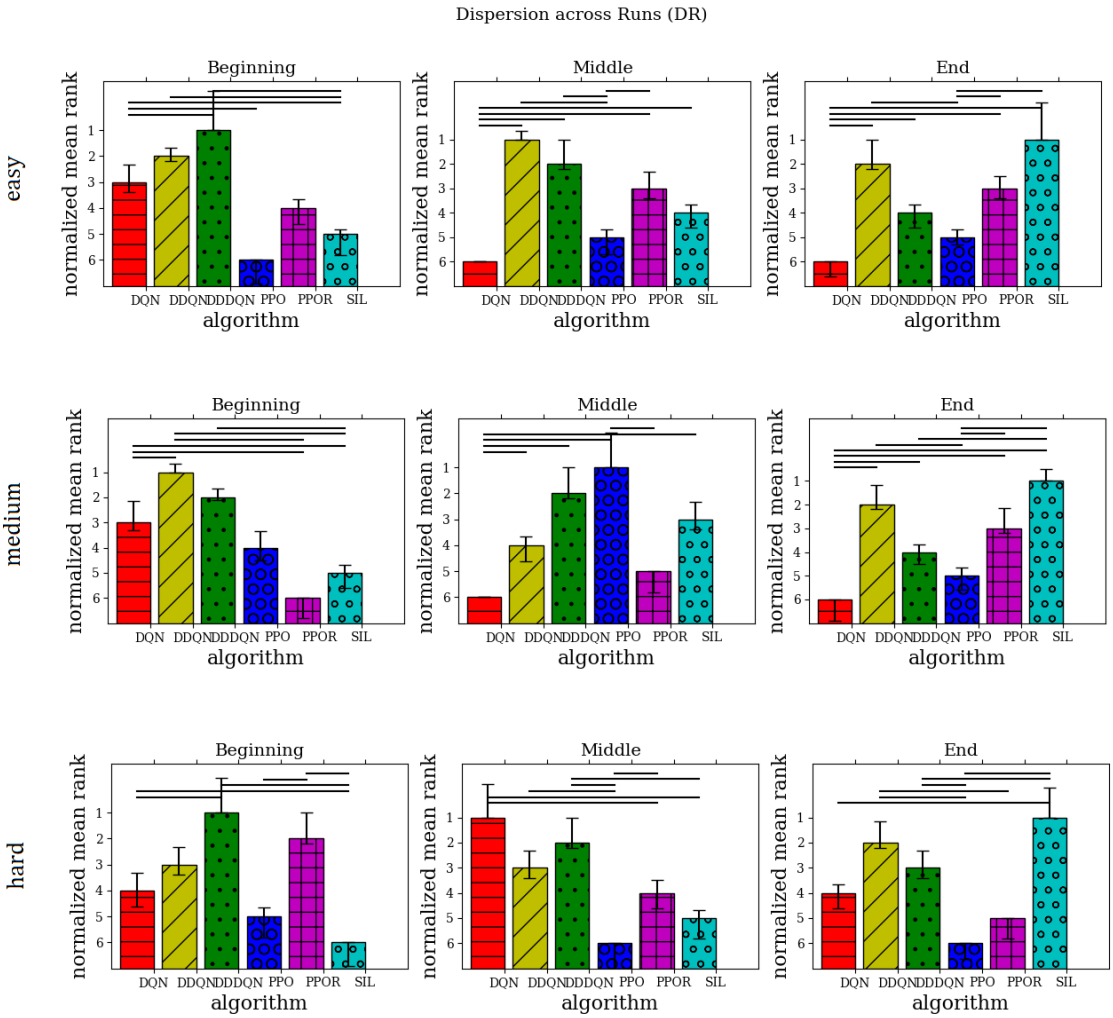



Figure .2: Dispersion across runs displayed seperately in the three levels of difficulty. Each training zone displays a normalized ranking of each algorithm in that range. Black error bars display 95% bootstrap confidence interval error (# bootstraps = 1000). Horizontal bars display significant pairwise differences.

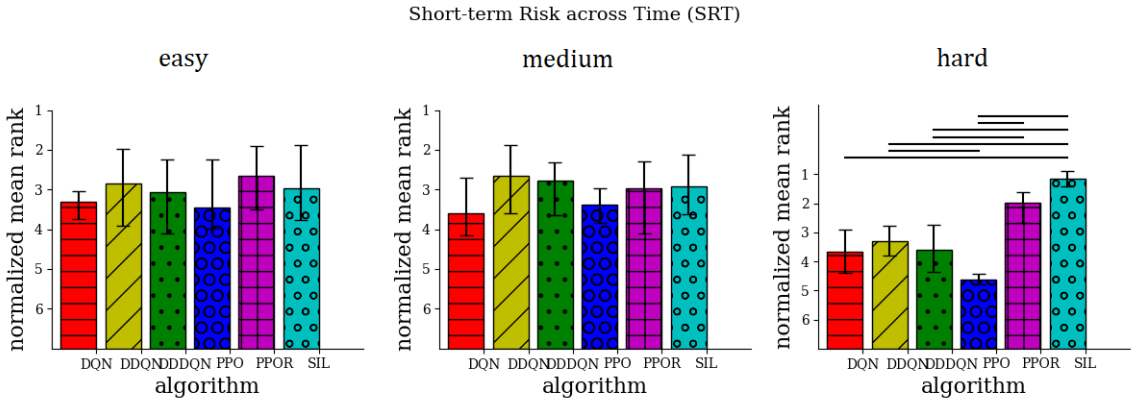



Figure .3: Short-term risk displayed seperately in the three levels of difficulty. Black error bars display 95% bootstrap confidence interval error (# bootstraps = 1000). Horizontal bars display significant pairwise differences.

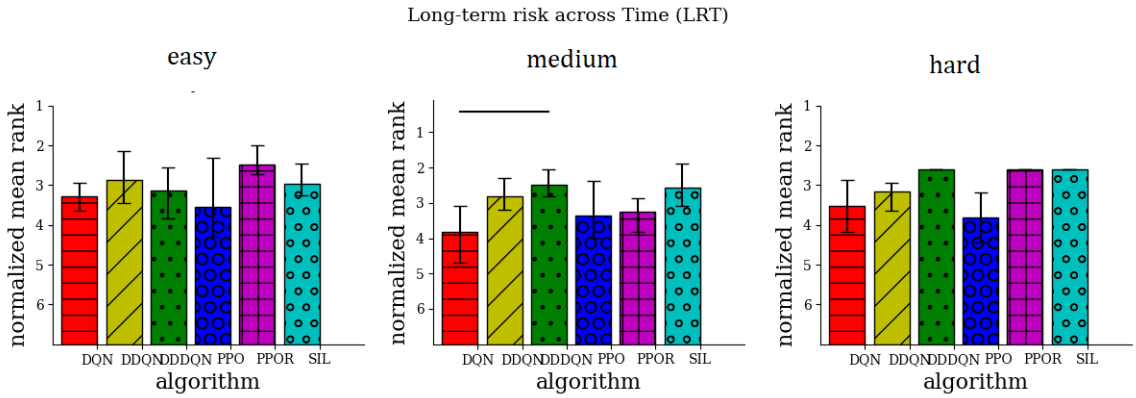



Figure .4: Long-term risk displayed seperately in the three levels of difficulty. Black error bars display 95% bootstrap confidence interval error (# bootstraps = 1000). Horizontal bars display significant pairwise differences.

# Bibliography


[1] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. "Dota 2 with large scale deep reinforcement learning". In: *arXiv preprint arXiv:1912.06680* (2019) (cit. on p. 20).

[2] Lucian Busoniu, Robert Babuska, and Bart De Schutter. "A comprehensive survey of multiagent reinforcement learning". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 38.2 (2008), pp. 156–172 (cit. on p. 24).

[3] Georgios Chalkiadakis. "Multiagent reinforcement learning: Stochastic games with multiple learning players". In: *Dept. of Computer Science, University of Toronto, Canada, Tech. Rep* 25 (2003) (cit. on p. 24).

[4] Stephanie CY Chan, Samuel Fishman, John Canny, Anoop Korattikara, and Sergio Guadarrama. "Measuring the reliability of reinforcement learning algorithms". In: *arXiv preprint arXiv:1912.05663* (2019) (cit. on pp. 44–47).

[5] Caroline Claus and Craig Boutilier. "The dynamics of reinforcement learning in cooperative multiagent systems". In: *AAAI/IAAI* 1998.746-752 (1998), p. 2 (cit. on p. 24).

[6] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. "Implementation matters in deep policy gradients: A case study on ppo and trpo". In: *arXiv preprint arXiv:2005.12729* (2020) (cit. on pp. 2, 39).

[7] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. "Counterfactual multi-agent policy gradients". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018 (cit. on pp. 1, 24).

[8] Jakob N Foerster, Yannis M Assael, Nando De Freitas, and Shimon Whiteson. "Learning to communicate with deep multi-agent reinforcement learning". In: *arXiv preprint arXiv:1605.06676* (2016) (cit. on p. 24).

[9] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, and Joelle Pineau. "An introduction to deep reinforcement learning". In: *arXiv preprint arXiv:1811.12560* (2018) (cit. on pp. 10, 14).

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016 (cit. on pp. 10, 12).

[11] Matthew Hausknecht and Peter Stone. “Deep recurrent q-learning for partially observable mdps”. In: *2015 aaai fall symposium series*. 2015 (cit. on pp. 17, 37).

[12] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. “Deep q-learning from demonstrations”. In: *Thirty-second AAAI conference on artificial intelligence*. 2018 (cit. on p. 25).

[13] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 12).

[14] Ionel-Alexandru Hosu and Traian Rebedea. “Playing atari games with deep reinforcement learning and human checkpoint replay”. In: *arXiv preprint arXiv:1607.05077* (2016) (cit. on p. 25).

[15] Hoang M Le, Yisong Yue, Peter Carr, and Patrick Lucey. “Coordinated multi-agent imitation learning”. In: *International Conference on Machine Learning*. PMLR. 2017, pp. 1995–2003 (cit. on p. 25).

[16] Jiaoyang Li, Zhe Chen, Yi Zheng, Shao-Hung Chan, Daniel Harabor, Peter J Stuckey, Hang Ma, and Sven Koenig. “Scalable Rail Planning and Replanning: Winning the 2020 Flatland Challenge”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 31. 2021, pp. 477–485 (cit. on p. 25).

[17] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. “Continuous control with deep reinforcement learning”. In: *arXiv preprint arXiv:1509.02971* (2015) (cit. on p. 32).

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013) (cit. on pp. 1, 8, 14, 15, 29).

[19] Sharada Mohanty, Erik Nygren, Florian Laurent, Manuel Schneider, Christian Scheller, Nilabha Bhattacharya, Jeremy Watson, Adrian Egli, Christian Eichenberger, Christian Baumberger, et al. “Flatland-RL: Multi-agent reinforcement learning on trains”. In: *arXiv preprint arXiv:2012.05893* (2020) (cit. on pp. 2, 25, 28).

[20] Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. “Self-imitation learning”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 3878–3887 (cit. on pp. 22, 23).

[21] Christopher Olah. “Understanding lstm networks”. In: (2015) (cit. on p. 13).

[22] Georgios Papoudakis, Filippos Christianos, Lukas Schäfer, and Stefano V Albrecht. “Comparative evaluation of cooperative multi-agent deep reinforcement learning algorithms”. In: *arXiv: 2006.07869* (2020) (cit. on p. 45).

[23] Riccardo Polvara, Massimiliano Patacchiola, Sanjay Sharma, Jian Wan, Andrew Manning, Robert Sutton, and Angelo Cangelosi. “Autonomous quadrotor landing using deep reinforcement learning”. In: *arXiv preprint arXiv:1709.03339* (2017) (cit. on p. 14).

[24] Bob Price and Craig Boutilier. "Implicit imitation in multiagent reinforcement learning". In: *ICML*. Citeseer. 1999, pp. 325–334 (cit. on p. 25).

[25] Martin L Puterman. "Markov decision processes". In: *Handbooks in operations research and management science* 2 (1990), pp. 331–434 (cit. on pp. 5, 6).

[26] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. "Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 4295–4304 (cit. on pp. 1, 25).

[27] Stuart Russell and Peter Norvig. "Artificial intelligence: a modern approach". In: (2002) (cit. on p. 1).

[28] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015) (cit. on pp. 18, 19).

[29] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. "Trust region policy optimization". In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897 (cit. on p. 21).

[30] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017) (cit. on pp. 2, 20–22, 39).

[31] Sandip Sen, Mahendra Sekaran, John Hale, et al. "Learning to coordinate without sharing information". In: *AAAI*. Vol. 94. 1994, pp. 426–431 (cit. on p. 24).

[32] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018 (cit. on pp. 4–9).

[33] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems*. 2000, pp. 1057–1063 (cit. on p. 19).

[34] Sebastian Thrun and Anton Schwartz. "Issues in using function approximation for reinforcement learning". In: *Proceedings of the Fourth Connectionist Models Summer School*. Hillsdale, NJ. 1993, pp. 255–263 (cit. on p. 16).

[35] Karl Tuyls and Peter Stone. "Multiagent learning paradigms". In: *Multi-Agent Systems and Agreement Technologies*. Springer, 2017, pp. 3–21 (cit. on p. 24).

[36] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016 (cit. on pp. 8, 16).

[37] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. "Dueling network architectures for deep reinforcement learning". In: *International conference on machine learning*. PMLR. 2016, pp. 1995–2003 (cit. on pp. 8, 18).

[38] Jim YF Yam and Tommy WS Chow. “A weight initialization method for improving training speed in feedforward neural network”. In: *Neurocomputing* 30.1-4 (2000), pp. 219–232 (cit. on p. 11).