UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

Simulating crowds using deep multi-agent reinforcement learning

Lennard Snoeks

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Fabian DI FIORE

2019
2020

Faculteit Wetenschappen
School voor Informatietechnologie
master in de informatica

Masterthesis

Simulating crowds using deep multi-agent reinforcement learning

Lennard Snoeks
Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Fabian DI FIORE

Abstract

This thesis aims to use and compare different methods of multi-agent reinforcement learning to train crowd simulation models and generate believable steering behavior. The goal of the agents in these models is to start at point A and successfully navigate to point B, while avoiding collisions with other agents. Therefor it is logical to start exploring crowd simulation techniques and methods. What are existing approaches and how are they categorized? Next, reinforcement learning is explained. Just as supervised and unsupervised learning, reinforcement learning is a machine learning paradigm. However, the learning is done without any prior acquired labeled or unlabeled data fed into the system. The agents train its policy, which can be seen as its brains, solely by exploring the environment and obtaining rewards from executing certain actions. Initially, single-agent reinforcement learning methods are explained. Afterwards, the transition is made to deep learning, where neural networks are used as function approximators that allow for faster and better learning. As crowd simulation is inherently a multi-agent problem, it is only logical that multi-agent reinforcement learning methods are explained, namely centralized learning, concurrent learning and centralized learning with decentralized execution. Centralized learning keeps all data centralized and learns a single policy while concurrent learning learns a policy for every agent. For centralized learning with decentralized execution, two methods are implemented: parameter sharing and a centralized critic approach. Parameter sharing allows each agent to learn independently, but they all share the same brain. The centralized critic approach is similar to the concurrent one, but now agents have access to whereabouts of other agents. Each of these methods is subjected to multiple reinforcement learning algorithms and scenarios, and are subsequently compared based on performance and behavior of pedestrians. Centralized learning with decentralized execution seemed essential for successful learning, thanks to the data sharing between agents, which in turn allows for better learning.

Acknowledgements

Foremost, I would like to express my gratitude to my promotor Prof. dr. Fabian Di Fiore for giving advice when needed, and also for giving me a lot of freedom and independence during this research. Coming up with an own research question inside the field of crowd simulation allowed me to pick something I was interested in. It was not always easy, especially because deep learning is a relatively new field and I had to figure out a lot on myself. But ultimately, I think this was more helpful and also taught me way more than getting everything handed to you would.

Besides my promotor, I would also like to thank mr. Bram Vanherle. He did a master thesis in a similar subject last year and often provided useful information that helped me push this research forward. He also helped kickstart the research by providing some useful sources, many of which I ended up using.

In addition, I would like to thank students and classmates with whom I worked or spent countless library sessions with during the years. We worked and accomplished a lot, but also found time for enough extra curricular activities, which are ultimately necessary to prevent yourself from not going insane.

Last but not least, I would like to thank my family, my parents Marc and Nancy, and my two brothers Marten and Elian. Coming home after long, not always very productive, days into a welcoming family definitely lightened the burden of work.

Introduction

The behavior of crowds has consistently been a fascinating subject for people of different areas to investigate, be it in computer science, psychology or social appliances. It is particularly compelling how individuals with diverse motives and goals decide to act depending on the situations they find themselves in. What drives people to act a certain way? Not only will their own goals affect their behaviour, but also others in the crowd and the variables of their surrounding environment. In addition, depending on their location, people are often expected to follow social norms. Sometimes however, these norms are thrown out the window and people act irrationally. In the 19th century, this was one of the driving factors to start studying people's behaviour in crowds, as seen in Charles Mackay's *Extraordinary Popular Delusions and the Madness of Crowds* [1]. After spreading onto other fields, eventually computer science got involved with crowd behaviour research in the late 20th century. Craig Reynolds published a behavioural animation technique that simulated flying patterns of birds [2] and thus, computer models were used to simulate crowds.

Computer based crowd simulations have been gaining momentum over the years, and many different applications are now present. The two main directions inside the computer science field are *realism of behavioural aspects* and *high-quality visualisation* [3]. The first one shifts its focus towards simple 2D visualisations, with the goal of validating comparability to real life situations. Examples that come to mind are emergency simulations or sociological crowd models. Visualisation can be used in this case to substantiate simulation results, but are not essential. The intention of the second field is to produce high-quality animations of the crowd members. A convincing visual outcome is thus the priority, which is obtained by using rendering and animation methods. Applications include mostly movie productions and video games. In this thesis, the focus will lie on the behavioral aspects of crowd simulations.

With data being considered as the new gold, an abundance of it is available to the general public, including data of crowd or pedestrian movements. The issue however, is that human input is often necessary to make the data usable for crowd simulation purposes. Aerial images or videos need to be annotated frame by frame for each of the individuals present. This is where reinforcement learning could be of assistance. It is one of the three basic machine learning paradigms that, unlike supervised learning, does not require labelled input/output pairs to enable learning.

Optimal solutions will try to be found by exploring options and taking actions that maximize a reward. Often times, this will operate by using tables to store reward values of different states. If the state space is rather small, this will not be problematic. However, the state space in crowd simulation can grow very large in size, and thus the corresponding table will also store a lot of records. A preferable choice here would be to utilize a function approximator, with nowadays one of the most popular ones being neural networks. A neural network would then be trained using reinforcement learning, and when training is complete the model could be queried and used to steer an individual in a crowd. This is referred to as deep reinforcement learning. Standard reinforcement learning methods are single-agent approaches, but crowd exist of multiple people or pedestrians, therefor multi-agent reinforcement learning will be applied.

The goal of this thesis is to explore different deep multi-agent reinforcement learning to simulate realistic crowds, and compare these to each other or to already established methods. We will initiate this process by examining the field of crowd simulation and look at how different methods operate and classify. Afterwards, we will also explore the topic of reinforcement learning to get a better grip of the methods that are available and could be of use for our purpose. The final topic of literature to consult is neural networks. We will describe how they should be trained, used and tweaked to meet your demands. All the gathered research will then be combined and used to craft multiple deep multi-agent reinforcement learning methods to simulate crowds. Subsequently, these methods will be evaluated on multiple test scenarios and compared to each other, from which conclusions can be drawn.

Contents

1 Crowd simulation **11**
1.1 Flow-based . . . 13
1.1.1 Abstract flow models . . . 13
1.1.2 Vector fields and tiling . . . 14
1.1.3 Flowing continuum . . . 15
1.2 Entity-based . . . 17
1.2.1 Modeling Escape Panic . . . 18
1.2.2 Cellular automata model . . . 19
1.3 Agent-based . . . 20
1.3.1 Reynolds' Boids model . . . 20
1.3.2 Social Force model . . . 21
1.3.3 Hierarchical models . . . 23
1.3.4 Predictive models . . . 24
1.4 Evaluation . . . 25

2 Reinforcement learning **27**
2.1 Markov Decision Process . . . 28
2.2 Criterion of optimality . . . 30
2.2.1 Policy . . . 30
2.2.2 Value function . . . 31
2.3 Solving MDPs . . . 32
2.3.1 Generalized policy iteration . . . 33
2.3.2 DP methods . . . 33
2.3.3 RL methods . . . 37
2.3.4 Problems with RL . . . 45
2.4 Deep RL . . . 46
2.4.1 Deep Q Network . . . 47
2.4.2 Deep Deterministic Policy Gradient . . . 48
2.4.3 Proximal Policy Optimization . . . 51
2.4.4 Problems with deep RL . . . 53
2.5 Multi-agent RL . . . 54
2.5.1 Stochastic Markov Games . . . 54
2.5.2 Multi-agent taxonomy . . . 56

2.5.3 Benefits & challenges 56
2.6 Multi-agent deep RL 58
2.6.1 Centralized 58
2.6.2 Concurrent 59
2.6.3 Centralized learning, decentralized execution 59
2.6.4 Alternatives 60
2.7 Curriculum learning 62

3 Artificial neural networks 65
3.1 Fundamentals 65
3.1.1 Basic components 65
3.1.2 Activation functions 66
3.1.3 Network topology 67
3.2 Training ANNs 69
3.2.1 Paradigms 69
3.2.2 Loss function and gradient descent 70
3.3 Hyperparameter tuning 73
3.3.1 Parallel search 73
3.3.2 Sequential optimization 74
3.3.3 Population Based Training 75

4 Crowd simulation with deep multi-agent RL 77
4.1 Related work 77
4.1.1 Data driven crowd simulation 77
4.1.2 Crowd simulation with reinforcement learning 78
4.1.3 Mixed approach 80
4.2 Environment 81
4.2.1 Action 81
4.2.2 Observation 82
4.2.3 Reward 85
4.3 Expanding to multi-agent 86
4.4 Training 89
4.4.1 Setup 89
4.4.2 Input data 90
4.4.3 RL algorithm and hyperparameters 91
4.5 Visualization 96
4.6 Generalization 97

5 Evaluation 99
5.1 Comparing multi-agent methods 99
5.1.1 Performance 99
5.1.2 Movement of pedestrians 104
5.2 Importance of curriculum learning 111

5.2.1 Speed-up 111
5.2.2 Learned behavior 112
5.3 Linking back to RL problems 113
5.4 Conclusion 114
5.5 Future work 116

A1 Simulatie van menigtes d.m.v. deep multi-agent reinforcement learning: Nederlandse samenvatting 117
A1.1 Simulatie van menigtes 118
A1.2 Reinforcement learning 119
A1.3 Artificiële neurale netwerken 122
A1.4 Simulatie van menigtes met deep multi-agent reinforcement learning 123
A1.5 Evaluatie 124

A2 Hyperparameters 127
A2.1 DDPG 127
A2.2 PPO 128

A3 Bug reports 131

Bibliography 140

Chapter 1

Crowd simulation

A crowd is more than a solely a collection of individuals. How one individual acts is possibly affected by others in that same crowd, which is in turn depending on various physiological, psychological and social factors. Individuals may be pushed into behaving some type of way that is believed to be right, according to the crowd they are in. A crowd may thus display interesting dynamics that are compelling to study and subsequently model. The problem however is that such delicate and complex scenarios are hard to build using pure mathematical approaches or analytical models. [4] Crowd simulation techniques use more intricate approaches to simulate the movement and/or the behavior of a large number of entities or characters. [5] This chapter is largely inspired by the work of Zhou et al. [4], which provides an excellent overview of existing crowd modeling and simulation techniques. It will review some of the existing methods of crowd simulation to aid in further understanding and developing our method.

As already mentioned in the introduction, there are two directions of crowd simulations present, where our focus lies on *realism of behavioural aspects*. The priority is to create steering behavior for agents in 2D spaces. According to Zhou et al. [4], crowd modeling and simulation can be generally categorized by:

1. The size of the simulated crowd

2. The time-scale of the crowd phenomena of interest

These are determining factors when deciding what type of method will be used to tackle the crowd simulation problem. The crowd size can range from small to medium to huge while the time scale concerns long- or short-term crowd phenomena. A combination of the two results in existing crowd models being able to be generally classified in three categories:

1. Models of long-term crowd phenomena

2. Models of short-term phenomena of huge-sized crowd

3. Models of short-term phenomena of medium to small-sized crowd

Each category is thus determined by size and time-scale. Often it is argued that believability should be added as an extra criterion. The problem herewith is that the interpretation of believability is arguable, unlike the other criterions size and time, and is thus not considered. Most crowd models will however choose to adjust the granularity when modeling a crowd. This is especially necessary when dealing with huge crowds, where the computational cost grows very large when thousands or even millions of virtual individuals need to be updated continuously. This results in three approaches based on the amount of granularity:

1. Flow-based
2. Entity-based
3. Agent-based

These methods can also be classified according to size and time-scale, which in turn can be viewed in Figure 1.1. The above-mentioned methods will be discussed in the following sections.

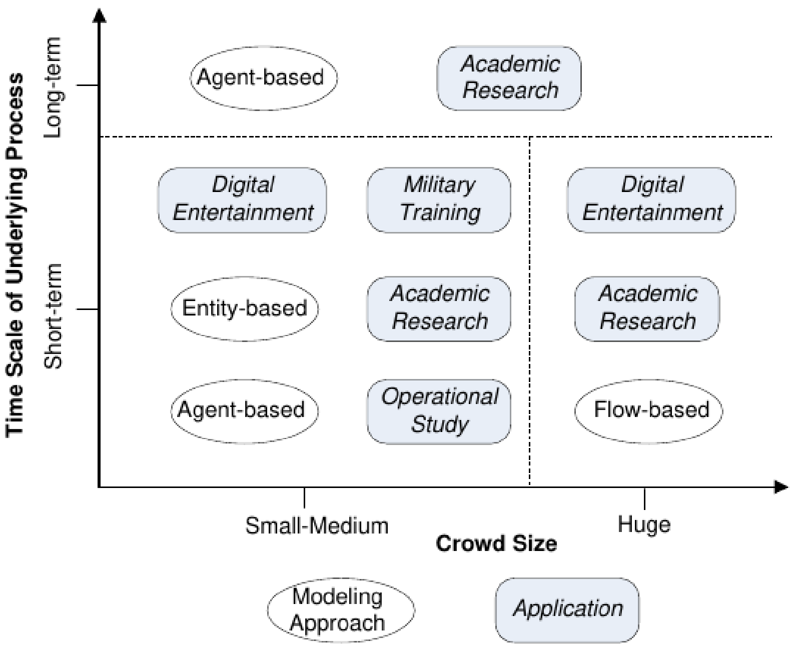


Figure 1.1: Classification of crowd models according to size and time, with granularity based models in ellipses and examples in rounded squares [4]

1.1 Flow-based

The premise for flow-based models is that features of individuals are neglected and the crowd is treated as a whole. The computational cost would simply grow too large if each individual would be dealt with separately. A layer of abstraction is thus provided in which the behaviour of a crowd is simulated by looking at it as a unit that moves, instead of modelling individualistic behaviour. As a result, the applications of flow-based models mostly include estimating the flow of movement/evacuation processes of huge crowds. Hereby, behavioral factors of individuals are left behind or reduced. Typically, a flow-based approach will utilize vector or velocity fields to display what the influence of environmental factors on the movement of the crowd is, and differential equations to describe the movement of the crowd. However, to compose these vector fields and differential equations, certain hypothesis and statistical assumptions are needed, whose validity are often questionable. [4] The rest of this section will touch on some flow-based approaches and how they operate.

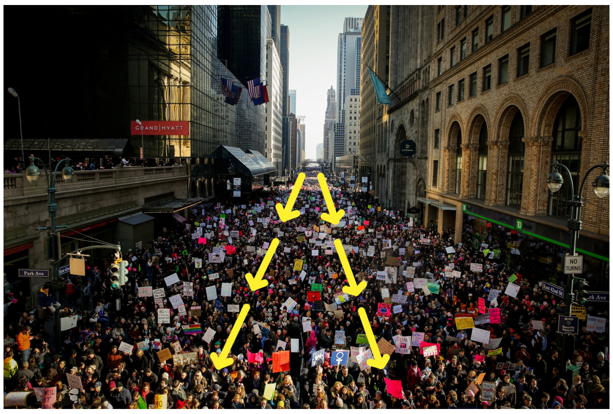


Figure 1.2: The arrows indicate the abstraction of a flow based model, applied on a march in the USA [6]

1.1.1 Abstract flow models

EVACNET is a very practical application that utilizes the flow-based concept and allows users to model building evacuations in a very user friendly approach [7]. The user is instructed to build a network model that serves as an abstraction for the desired behavior of the generated crowd that will execute the evacuation. This model consists of a set of nodes and arcs, where the nodes represent building components like rooms or stairs, while the arcs represent passages between said building components. The nodes contain a defined capacity that acts as an upper ceiling for the amount of people that can be present in the component. For the arcs, a traversal time and flow capacity set. The former will indicate how long it takes to pass through the passageway, while the latter expresses the maximum amount of people that can pass trough per time period. The algorithm will take the built network as input and subsequently generate a optimal plan for evacuation. The user may inspect results and correspondingly adjust the network to improve them. Figure 1.3 illustrates an annotated example of the representation of a two story building.

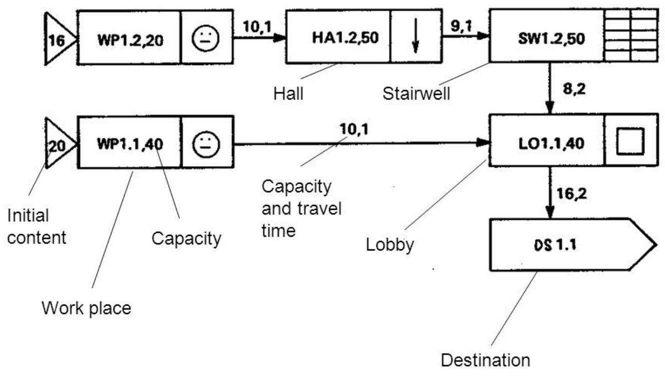


Figure 1.3: Representation of a two story building in EVACNET with annotations [8]

EVACNET is an example of using abstract flows. It is limited to modelling strictly movement of crowd, no behavioral aspects are touched upon, just as in the example shown in Figure 1.2. They both do not utilize vector fields or differential equations as mentioned at the start of this Section 1.1. [4]

1.1.2 Vector fields and tiling

A vector field is an assignment of a vector to each point in a subset of space [9], as shown in Figure 1.4. Applying these vector fields to flow-based crowd simulation logically results in velocity fields. These are simply an extension of vector fields where each vector indicates the velocity of a point in said subset of space.

Flow Tiles utilizes such velocity fields in combination with a tiling technique to display large flows [11]. Such fields are often used to simulate fluids or gasses, but other applications also include guiding of agents. Each flow tile defines a small, stationary region of the velocity field. These tiles can then be pieced together and construct large stationary fields. Subsequently, these fields are used to drive the motion of fluids or crowds. The two prominent tasks at hand when using this method are dividing the virtual environment into flow tiles and defining velocity fields on these tiles. When said division into tiles is completed, each tile is defined by velocity and flux values. All corners are assigned with a velocity while the edges get a flux, which indicates the volume flow rate, as seen in Figure 1.5. The algorithm will assess these values and take neighbouring tiles in consideration to create a velocity fields for each tile. In a crowd simulation situation, the agent just has to follow the

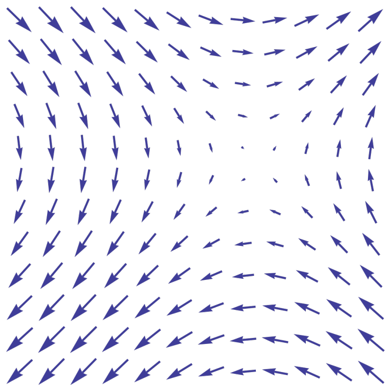


Figure 1.4: Example of a vector field [10]

velocity attached to the position it is in on the tile. Methods like these however do not implement more complex features such as collision avoidance, which can be troubling when the goal is generating believable behaviour.

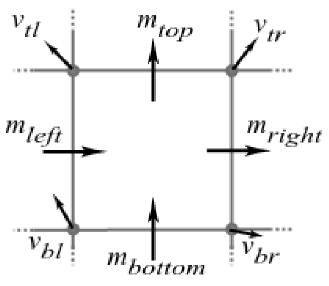


Figure 1.5: Velocity and flux values assigned to corners and edges of the tile [11]

1.1.3 Flowing continuum

Looking at crowds as a flowing continuum is a development popularized by Hughes. A crowd has the tendency to move with many of the same attributes as a fluid. However, the distinction between a crowd as a flowing continuum and a classical fluid lies in the crowd's capacity to think and reason.

Hughes' model

Hughes succeeded to develop a model that allows governing the flow of crowds by designing differential equations that are conformably mappable in space [12]. To get to these equations, Hughes' starting points were three hypotheses:

1. The speed at which pedestrians walk is determined solely by the density of surrounding pedestrians, the behavioral characteristics of the pedestrians, and the ground on which they walk.

2. Pedestrians have a common sense of the task (called potential) that they face to reach their common destination, such that any two individuals at different locations having the same potential see no advantage changing places.

3. Pedestrians seek to minimize their (accurately) estimated travel time but temper this behavior to avoid extreme densities. This tempering is assumed to be separable, such that pedestrians minimize the product of their travel time as a function of density.

Each of these three hypotheses can be supported by evidence in some sense, see [12] for details. These statements result in basic guiding equations for the flow of a single pedestrian type:

$$-\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\rho g(\rho) f^2(\rho) \frac{\partial \varphi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\rho g(\rho) f^2(\rho) \frac{\partial \varphi}{\partial y}\right) = 0 \tag{1.1}$$

and

$$g(\rho) f(\rho) = \frac{1}{\sqrt{\left(\frac{\partial \varphi}{\partial x}\right)^2 + \left(\frac{\partial \varphi}{\partial y}\right)^2}} \tag{1.2}$$

with φ the remaining travel time, which measures the remaining task or potential, ρ the density of the crowd, $f(\rho)$ the speed of pedestrians as a function of said density, $g(\rho)$ describes the discomfort of the crowd at said density, and (x, y, t) stand for the coordinates and time. The discomfort factor indicates the distance that a pedestrian is willing to walk unrestricted to avoid walking a greater unit distance in the crowd. When chosen right, the speed and discomfort function allow the above equations to be conformably mappable in space, even when they are time dependent and non linear. [12]

Continuum crowds

Continuum crowds is a real-time crowd model based on continuum dynamics [13], and takes Hughes' model as inspiration in the sense that it uses a similar potential function to guide crowds towards their goals [4]. Motion is seen as particle energy minimization by adopting a continuum perspective on the system. This, in turn, yields dynamic potential and velocity fields that will guide the pedestrians in a crowd simultaneously. Pedestrians in continuum crowds do not experience changes being in the proximity of others because global path planning is used that guarantees collision avoidance. To develop such a model, some starting hypotheses were setup yet again:

1. Each pedestrian is trying to reach a geographic goal G.

2. Pedestrians move at the maximum speed possible, which is expressed as *maximum speed field* f.

3. There exists a discomfort field g, so that pedestrians prefer to be at point x rather than x' if $g(x') > g(x)$.

4. Assuming that the goal, speed field and discomfort field are set, a pedestrian will pick a path by minimizing three factors: *path length*, *time* and *discomfort*. Depending on the type of pedestrian, these factors are adjusted.

Pedestrians are divided into large groups that have common goals. When people move in a group, they typically share the same speed, discomfort and goal. To calculate the optimal path for a pedestrian, the *potential function* ϕ can be defined so that it satisfies the *eikonal equation* $||\Delta\phi(x)|| = C$, where C is the unit cost depending on the three factors from hypotheses 4. This results in a *potential field*. The speed of a group is dependent on its density, because at high densities speed is dominated by the movement of nearby people. The first step is to convert the entire crowd into a *density field*, which in turn is used to convert the speed into a *velocity field*.

Because it not feasible to calculate these fields for every continuous value in the environment, the space is discretized into a regular 2D grid. All fields are stored as 2D array of floating points. To update positions of people in the crowd, the simulator will advance through the following steps:

```
For each timestep:
    Convert the crowd to a density field
    For each group:
        Construct the unit cost field $C$
        Construct the potential function $\phi$ and it's gradient $\Delta\phi$
        Update the location of people by interpolating the velocity field
```

[13]

1.2 Entity-based

When analysing social phenomena, the usage of physical and mathematical methods has increased considerately. The study of crowd motion applying physical theories and analytical models of individual pedestrians has even been well established. The essential feature of said models is to handle the crowd at individual level and treat individuals as homogeneous entities whose movement are governed by similar physical laws on physical particles. Essentially, the motions of said entities are influenced by global or local laws that are introduced to represent various physical, social or psychological influences on an individual's movement in a crowd. These models, often referred to as particle system models, require proper design of forces to reflect the impact of behavioral and environmental influences on navigation or movement of individuals. Entity-based models can be applied to achieve jamming or flocking behaviour. [4]

1.2.1 Modeling Escape Panic

Helbing et al. [14] concluded that, during escape situations where panic is present, following characteristics can be observed: people are moving faster than normal, they start pushing each other and interactions reach a physical stage. At exits or tight spaces, moving reaches an even more uncoordinated stage, and clogging or jamming behaviour is observed. Tunnel vision also causes people to lose focus, which in turn can lead to missing alternate exits or even trampling over fallen people. All these observations led Helbing et al. to design a model based on socio-psychological and physical forces that describes the phenomenon of escape panic in self-driven many-particle systems. Each pedestrian is represented as a particle i with mass m_i^0 and is assigned a speed v_i^0 in a certain direction e_i^0. The pedestrian adapts its velocity v_i with a time interval τ_i. To avoid collisions with other pedestrians j or walls W, a velocity-dependent distance is kept relative to those two, which can be modelled by interaction forces f_{ij} and f_{iW}. The change in velocity for a pedestrian i at time t is mathematically defined by the following acceleration equation:

$$m_i \frac{dv_i}{dt} = m_i \frac{v_i^0(t)e_i^0(t) - v_i(t)}{\tau_i} + \sum_{j(\neq i)} f_{ij} + \sum_{W} f_{iW} \tag{1.3}$$

The interaction force between two pedestrians can be divided into three separate types. The first one is the *repulsive interaction force* and describes the psychological tendency of to pedestrians to stay away from each other. The two other forces are inspired by granular interactions, with the first one being a *body force* that counteracts body compression, and the second one being a *sliding friction force*, that impedes relative tangential motion. See [14] for details on the interaction forces.

Helbing's model managed to reproduce multiple observed phenomena such as clogging at bottlenecks, jamming at widenings and even inefficient use of alternative exits. However, when modeling the psychological and social factors, looking at the individual as homogeneous particles may not be enough. For each individual, different influences can determine their behaviour, such as family ties or personalities. Helbing's model can be extended to represent other behavioral factors by designing them as generalized forces applicable to all the pedestrians or particles. [4]

1.2.2 Cellular automata model

Burstedde et al. [15] simulated pedestrian navigation by combining a particle system model with a 2D cellular automata model. Each pedestrian is represented by a particle in a grid space build of multiple cells. Each cell can either be empty or occupied by one pedestrian, with the size of such a cell being more or less the same as the average space occupied by a pedestrian in a dense crowd.

For each of its adjacent cells, the particle has a transition value that indicates the probability for a move to that cell. This results in a $3x3$ *matrix of preferences* M_{ij}, with the center value indicating the probability of the particle not moving at all. This can be seen in Figure 1.6. Particles that belong to the same species will also share the same transitional matrix. However, when species are different, transitional matrices can also differ. The update for all the particles is done in parallel at the same time. If the particle wants to move to an occupied cell, it will not move and stay in its current cell. If two particles want to move to the same cell, the particle with the highest probability gets to occupy that cell, while the other stays in its current cell.

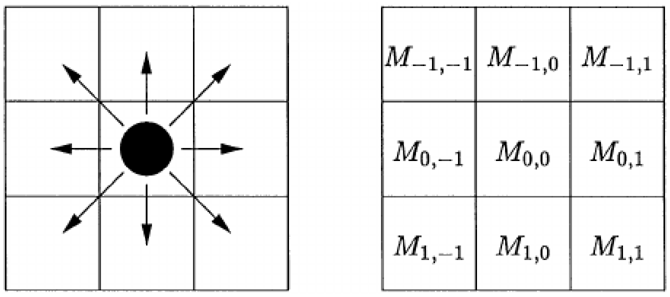

$M_{-1,-1}$ $M_{-1,0}$ $M_{-1,1}$
$M_{0,-1}$ $M_{0,0}$ $M_{0,1}$
$M_{1,-1}$ $M_{1,0}$ $M_{1,1}$

Figure 1.6: A particle with the transitions to each of its adjacent cells, and the matrix of preference M associated with these transitions [15]

During crowd simulation, it is important to take interactions between pedestrians and the geometry of the building into account in a unified and simple way. To achieve this without loosing the transitional matrices that define local transition rules, *floor fields* are introduced. Floor fields can be viewed as a secondary grid, lying underneath the grid of cells in which the pedestrians are stationed. Generally, the fields can be separated into *static* and *dynamic* floor fields. The *static floor field S* never change with time or by the presence of pedestrians. To indicate more interesting or attractive regions, such as an emergency exit, these plans are often applied. The *dynamic floor field D* is altered by the presence of the pedestrians and in turn has its own dynamics. The aim of the dynamic floor fields is to model attractive interaction between pedestrians. When moving over the fields, each pedestrian will leave a trace that increases the floor field of occupied cells (the frequency in which cells are visited). Because the transitions of each cell are proportional to the dynamic floor field, following other pedestrians is more advantageous. Taking static and dynamic floor fields into account, the *transition probability* p_{ij} in direction (i, j) is given by:

$$p_{ij} = NM_{ij}D_{ij}S_{ij}(1 - n_{ij}) \tag{1.4}$$

with n_{ij} indicating occupation (0 or 1) of target cell in direction (i, j). This ensures that transitions to occupied cells are impossible. N is used as a normalization factor so that the sum of probabilities to all nine target cells equals 1.

1.3 Agent-based

The dominating method to tackle crowd modeling and simulation is the agent-based one. This approach applies the highest granularity, which results in each pedestrian in the crowd being represented as an autonomous agent that can have its own attributes and state. A direct consequence of this is that agents have cognitive abilities and are able to reason. The agent can base its own decisions on the current surroundings, and adapt accordingly. This approach also most closely resembles real life situations, where each individual does its own thing.

Agents are typically regulated by a set of decision rules, but each agent is still capable of making a decision individually. However, some global patterns may still appear, even with a limited amount of rules. Generally, agent-based crowd modeling or simulation has to touch on three aspects: *navigation*, *decision making*, and *animation*. Navigation addresses how the agents can be moved in the virtual environment. Examples can include path planning and steering algorithms. Decision making speaks for itself as well, the agent will decide what to do under certain circumstances. Often, this is handled by introducing decision rules, which are typically intuitive and specific to the environment. Animation is an important part for digital media and entertainment such as video games or special effects. It is responsible for visually displaying the result of the modeling or simulation. In addition, navigation can also play an important rule for validation. Experts can compare the results with some existing models or with their own knowledge to locate anomalies. [4]

1.3.1 Reynolds' Boids model

Rule-based systems provide a way for a steering agent to make decisions in certain situations. The first significant rule-based model was Reynolds' Boids model [2]. The aim of this model was to replicate behavior of birds, and especially flocks of birds. Instead of specifying behavior as a whole for the entire flock, Reynolds defined behavior individually for each bird. A *boid* refers to a "bird-oid" object and thus represents a bird in this case. Reynolds used a simple set of three local rules to achieve complex and realistic flocking behavior for these boids:

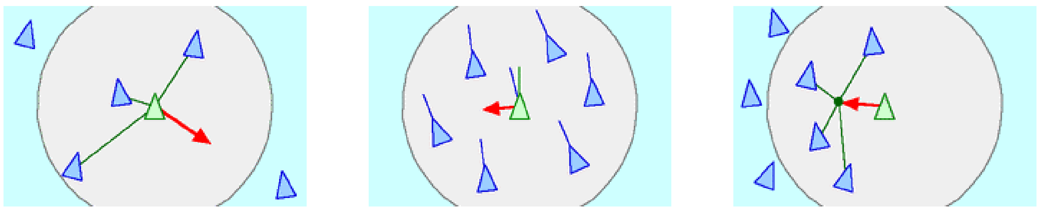


Figure 1.7: The three basic rules on which the Reynolds' Boids system is built, with the red arrows indicating behavior instilled by the rule. [16]

1. **Collision Avoidance**: Each boid inside the flock attempts to maintain a reasonable amount of distance to their flockmates. This is mainly done to prevent overpopulation or congestion.

2. **Velocity Matching**: Each boid attempts to match velocity with nearby flockmates. Velocity is expressed as a vector and thus has a direction and a speed value. The vector will be aligned according to the average alignment of birds that are nearby.

3. **Flock Centering**: Each boid attempts to stay close to nearby flockmates. This is done by taking the average position of nearby boids, and moving towards that point.

These rules are visualized in Figure 1.7. Collision avoidance and velocity matching work complementary here. The first one of the two steers away from impending impacts and is based on the relative position of surrounding boids and ignores their velocity. On the other hand, velocity matching will ignore positions of nearby boids and is dependent on their velocity. Combine these two and it is highly unlikely that two boids will collide with each other. Each boid also has its own perception of the world. For flock centering, the center of the flock will actually indicate the center of nearby flockmates. The circle on Figure 1.7 indicates what the individual boid will experience as their flock. Reynolds' model is pretty intuitive and simple, but it still succeeded in generating complicated patterns.

1.3.2 Social Force model

The Social Force model [17] shares a lot of similarities with the Escape Panic model, described in Section 1.2.1. It is also developed by Helbing, but now focuses less on modelling escape and panic situations, and more on motion of pedestrians in regular situations. The Social Force model is built on the assumption that the motion of pedestrians can be characterized as if certain *social forces* influence them.

For a pedestrian α, there are a number of effects that will regulate their motion. Firstly, their goal is to reach a destination $\vec{r}_\alpha^{\,0}$ with the highest possible comfort. This implies no detours, using the shortest path, etc. The path leading to their destination is usually built up out of multiple edges, and at time step t the *desired direction* of the pedestrian is indicated by $\vec{e}_\alpha(t)$. This direction is dependent on the current position of the pedestrian and the future edges coming up on their path. If undisturbed, the pedestrian will continue its journey to their goal with a *desired speed* v^0_{alpha}. Of course, it is possible to experience deviations along the way, which are indicated by the *actual velocity* $\vec{v}_\alpha(t)$. The *relaxation time* τ is used to indicate how long deviations take. Bringing all this information together, the *acceleration* of a pedestrian is described as:

$$\vec{F}^0_\alpha(\vec{v}_\alpha, v^0_\alpha \vec{e}_\alpha) = \frac{1}{\tau_\alpha}(v^0_\alpha \vec{e}_\alpha - \vec{v}_\alpha) \tag{1.5}$$

The deviations experienced by pedestrians are mostly caused by other pedestrians. Each pedestrian will attempt to keep a distance relative to other pedestrians, that is also dependent on the density of the crowd they find themselves in and the desired speed v^0_α. Usually, the closer a pedestrian is to others, the more uncomfortable they will feel. They will thus feel *repulsive effects* towards other pedestrians β. This repulsive effect is represented as:

$$\vec{f}_{\alpha\beta}(\vec{r}_{\alpha\beta}) = -\nabla_{\vec{r}_{\alpha\beta}} V_{\alpha\beta}[b(\vec{r}_{\alpha\beta})] \tag{1.6}$$

with $V_{\alpha\beta}(b)$ being the repulsive potential function, that, among other things, will ensure that a pedestrian has enough space to take their next step. In addition, the pedestrian also feels more uncomfortable when they are close to walls or borders. Hence, a border B also evokes repulsive effects described by:

$$\vec{f}_{\alpha B}(\vec{r}_{\alpha B}) = -\nabla_{\vec{r}_{\alpha B}} U_{\alpha B}(||\vec{r}_{\alpha B}||) \tag{1.7}$$

again with $U_{\alpha B}(||\vec{r}_{\alpha B}||)$ being the repulsive potential function. So far only repulsive effects have been addressed, however, pedestrians can also be attracted to others (friends, artists, etc.). These *attractive effects* is represented as:

$$\vec{f}_{\alpha i}(||\vec{r}_{\alpha i}||, t) = -\nabla_{\vec{r}_{\alpha i}} W_{\alpha i}(||\vec{r}_{\alpha i}||, t) \tag{1.8}$$

with $W_{\alpha i}(||\vec{r}_{\alpha i}||, t)$ being the attractive potential function. The difference here is that *attractiveness* can decrease over time, and thus a time factor is introduced. Attractive effects allow for groups of pedestrians to be formed. To lessen the impact of forces when there is no direct contact between pedestrians, for example when other pedestrians or objects are in between them, the *effective angle of sight* is applied. This ensures that the effect of perception is taken into account by making the influence weaker if there is no direct angle of sight. When all these forces are now brought together, the pedestrians total motivation $\vec{F}_\alpha(t)$ can be constructed. The Social Force model is then given by:

$$\frac{d\vec{(w)}_\alpha}{dt} = \vec{F}_\alpha(t) + fluctuations \tag{1.9}$$

The fluctuations term takes random variations of behavior into account. Certain situations are ambiguous in nature where actions are deemed equivalent. Sometimes a pass will happen along the left, another time it will happen along the right.

Again, the Escape Panic model and Social Force model do not differ a whole lot. The Social Force model was designed earlier, and the Escape Panic model was probably inspired by and build upon the findings of the SFM. Because the Escape model talks about particle systems and bigger crowds, the granularity is a bit lower and thus it was mentioned under entity-based crowd simulation. However, often times the distinction between entity or agent based is not that large and methods can belong to both types.

1.3.3 Hierarchical models

Models that were previously discussed in this section view every agent in the crowd as an individual on the same level, no distinction is made between different agents or pedestrians. Hierarchical models aim to subdivide the crowd in different groups, where each group consists of a certain amount of individuals. ViCrowd [18] is such a model that establishes a hierarchy of virtual crowds, groups and individuals. Naturally, groups are made up of individuals and, a set of groups forms a crowd. Hence, groups and individuals are the entities of the simulation and the entire hierarchy is shown in Figure 1.8. The model adapted by ViCrowd allows them to instill certain

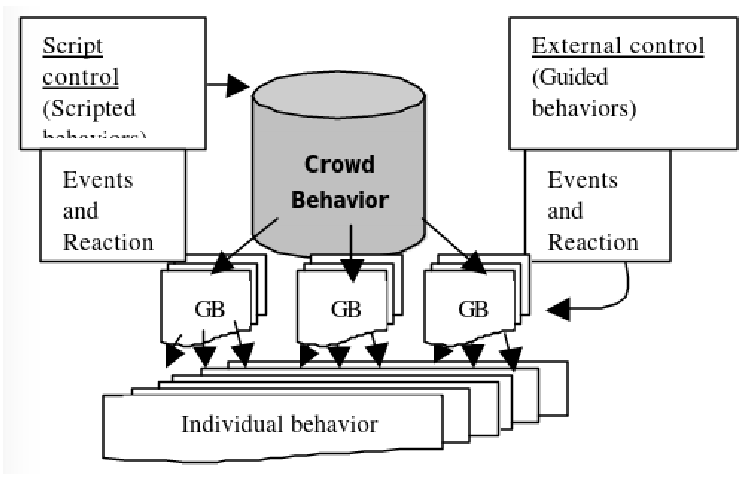


Figure 1.8: Hierarchical structure of the ViCrowd model [18]

types of behavior in the crowd at different levels. At the highest level, the entire crowd can be guided by *scripted behaviors*. Logically, this behavior is the same for all the pedestrians in every group. In addition, *guided behaviors* are used to alter how specific groups will behave. Apart from scripted and guided behavior, each agent also has *innate behaviors*. Innate behaviors are basic concept necessary to allow good crowd simulation results that are continuously present and can't be overwritten by scripted or guided behaviors. For example, when a group is ordered to navigate somewhere, basic principles of crowd simulation like collision avoidance or goal seeking can not be disabled. Introducing a model where specific groups can experience different behavior offers a lot of possibilities. For example, different social groups can be formed like friends or families, which is very interesting for crowd simulation.

Alqurashi et al. [19] also applied an hierarchical approach for agent-based traffic or pedestrian routing. At the center of the system lies the *Traffic Management Center*. At its core, their task is to prodive drivers or pedestrians with real-time information about other agents and what the best (alternative) routes are to take. The microscopic layer treats each pedestrian or driver individually, and they can be assigned to groups. Each group then contains one *intelligent* agent. The traffic center uses these intelligent agents to model best behavior or navigational options. Figure 1.9 displays that model.

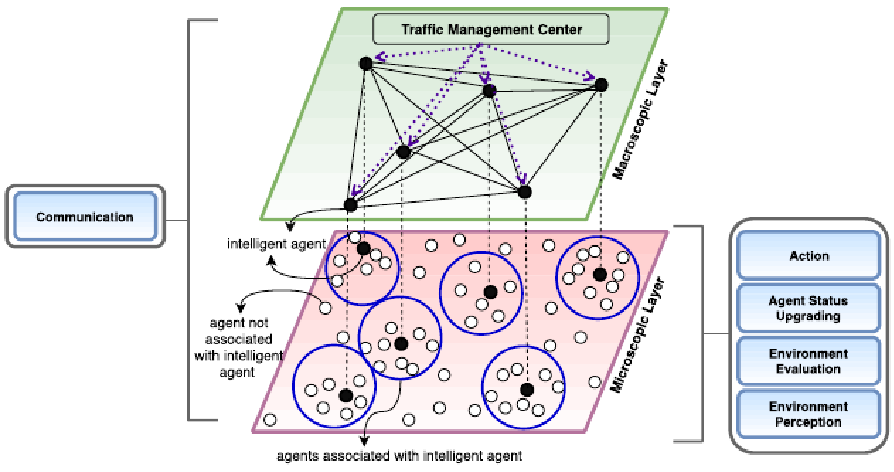


Figure 1.9: Hierarchical model applied for traffic or pedestrian simulations [19]

1.3.4 Predictive models

Most of the algorithms that were discussed up until this point, base their movement coordination solely on positions of other pedestrians. On the contrary, predictive methods operate by using other agents' instantaneous velocities to linearly extrapolate future paths. Reynolds, who previously designed the Boids model discussed in Section 1.3.1, implemented an extension of the Boids model [16] that attempts to predict where collisions between agents will take place, and subsequently prevents this by steering agents away. This simple concept is illustrated in Figure 1.10. The red arrows indicate the *repulsive forces* that agents experience. [20]

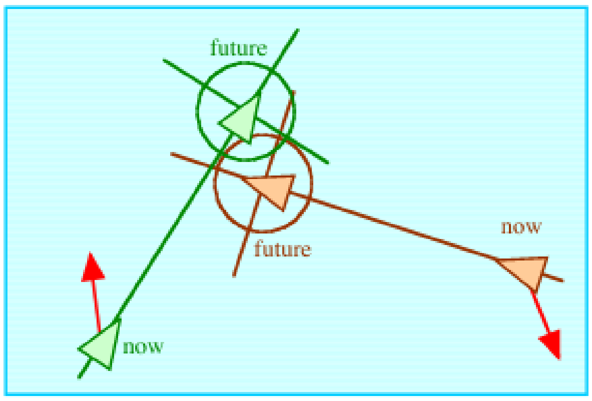


Figure 1.10: Unaligned collision avoidance using repulsive forces [16]

1.4 Evaluation

Validation and evaluation are essential parts of crowd simulation because they will actaully assess how the model performs. When designing crowd simulation methods, receiving feedback about current behavior gives the designer insight as to how their model is performing. When such information is presented in a clear manner, it makes the job of the designer also much easier. Validation is thus not only crucial at the later stages when the model is evaluated or compared with other methods, but also during the entire development process. However, model validation is still a very difficult issue in the crowd modeling world. Validity of a model mostly depends on what the objectives of the study are and how the model is used. For this reason, this issue is often orthogonal to crowd modeling techniques. Modeling techniques can be very specific to certain situations, and some validation techniques proposed by others might not work on these models. Also, progress in model validation continues to be rather slow and limited relative to the new crowd simulation methods that pop up. Often, there is also an insufficient amount of detailed data to make proper validation possible. [4]

Considering these issues, Singh et al. developed SteerBench [21], a benchmark framework that aims to objectively evaluate steering behavior for virtual agents. The framework present numerous test scenarios that vary greatly in environment, number of agents, etc. and are divided into categories that rise in difficulty. In addition, they provide various metrics of evaluation, with the three primary ones being:

- Number of collisions
- Time efficiency
- Effort efficiency

Time efficiency simply refers to the amount of time taken for an agent to reach the goal. The faster it reaches the goal, the more time efficient it is. Effort efficiency measures the effort an agent has to spend to reach its goal. Typically the kinetic energy is used for this purpose.

Chapter 2

Reinforcement learning

Situated between supervised and unsupervised learning, reinforcement learning is a machine learning paradigm that deals with learning in sequential decision making problems, in which feedback is limited [22]. An agent faced by such a problem must learn behaviour through trial-and-error interactions with a dynamic environment, while simultaneously trying to maximize a numerical reward signal [23][24]. Hence, the agent will not be told which actions to perform, but instead must determine which actions generate the highest reward.

Reinforcement learning differs from supervised learning in not requiring labeled input and output pairs. Supervised learning operates by learning labeled data that is provided by a knowledgeable overseer. Each entry in the data set describes the situation at hand and what the appropriate action is that the system should take in that situation. For such a learning paradigm, the object is to generalize its responses in such a manner that situations not present in the data set can be handled correctly. However, in interactive problems it can be a challenging task to obtain examples of desired behaviour that are both correct and representative of all the situations an agent can face. Unexplored territory will make supervised learning impossible because correct and representative examples can not be found. Here, the agent has to learn from its own experience, as reinforcement learning does. Also, reinforcement learning differs from unsupervised learning, which is mostly about finding structure concealed in unlabeled data sets. This is not the intention of reinforcement learning. Therefor, reinforcement learning is considered as a third machine learning paradigm. [24]

The two established factors in reinforcement learning that were discussed already are the *agent* and the *environment*. However, the distinction between the two might not always be clear. Everything that is not controllable by the agent is considered part of the environment. [22] Besides the agent and the environment, one can pinpoint four other main elements of a reinforcement learning model: a *policy*, a *reward signal*, a *value function*, and, optionally a *model of the environment*.

A *policy* states the agent's behaviour at a given time. It maps states of the environment to the actions to be taken in those states when an agent finds itself in those states. The policy can appear as a single function or a lookup, or it may require extensive computation such as a search process. The policy can also be considered as the core of an agent, because it is enough to determine behaviour. Often times, policies are stochastic and probabilities are used to specify the chances of each action in a certain state. A stochastic policy allows agents to explore the environment without consistently choosing the same action.

Reinforcement learning needs an incentive to define the goal of the problem. For this, a *reward signal* will be used that sends a number to the learning agent at each time step of the execution. The object of the agent is to then maximize this total reward. The value of the reward signal thus indicates the impact of the action, where a higher reward is better. Subsequently, the reward signal will affect affect the policy. Usually, reward signals are stochastic functions of the state of the environment and the actions taken.

A *value function* for a state defines the expected return when the agent finds itself in said state and follows the defined policy. Action choices will be made based on these value judgments, because actions are sought that result in entering states with the highest value, and not the highest reward. This is done because these actions result in the optimal amount of reward in the long run. Unfortunately, compared to resolving rewards, resolving values is way harder. Rewards are results of the agent's actions in the environment, which are stationary. The values however must be estimated and re-estimated based on all the actions the agent has taken over its lifetime. Efficiently estimating values is one of the most important parts of reinforcement learning algorithms.

The final component is the *model* of the environment. The model will allow for assumptions to be made about how the environment will behave. When given a state and action, the model may be capable of predicting the next state and reward. This is why models are often used for planning purposes. Planning consists of considering potential future situations before they take place. Reinforcement learning methods that use models are referred to as *model-based* methods, in contrast to *model-free* methods that are purely trial-and-error learners. [24]

As mentioned above, reinforcement learning is still inherently a sequential decision making problem. There are several classes of algorithms that solve this problem, such as *programming* and *search and planning*. The third option is *learning*, and this solution will be the central theme further discussed in this chapter. [22]

2.1 Markov Decision Process

MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations or states through those future rewards [24]. In an MDP, the learner or the *agent* will continually make decisions based on the *environment* they are situated in. The agent-environment interaction is shown in Figure 2.1. An MDP will concretely consist of states, actions, transitions between those states and a reward function definition.

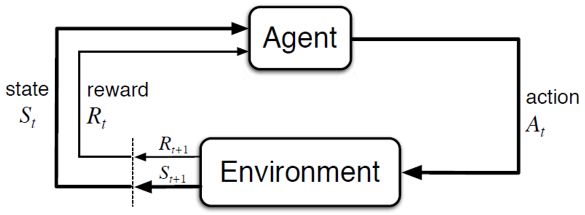


Figure 2.1: The interaction between an agent and the environment they find them self in, in an MDP [24]

States

The set of environmental states is defined as a finite set $S = \{S^1, ..., S^N\}$ of size N. During the learning process, the agent and its environment will interact at a sequence of time steps $t = 0, 1, 2, 3, ...$ The agent observes the environment and as a result obtains a representation of the environment's state at each time step, $S_t \in S$. This state is a unique depiction of everything of importance for the agent.

Actions

Likewise, the set of actions is defined as a finite set $A = \{A^1, ..., A^K\}$ of size K. Actions are used to control the state of the system. The set of actions that can be executed in a specific state $S_t \in S$ is declared as $A(S_t) \subseteq A$. In some models, it is not possible to choose every action in each state, but it is often assumed. Thus based on the state of the environment, the agent will choose and execute an action at each time step, $A_t \in A(S)$.

Reward function

The reward function specifies expected instantaneous rewards as a function of the current state, action and resulting state, $R : S \times A \times S \to \mathbb{R}$. The reward function provides feedback to the agent about the influence of the chosen action (positive or negative) and in doing so implicitly declares what the goal of the learning process is. The system is also given a direction in which way it should be controlled. It is thus of the utmost importance to choose a fitting reward function for the problem. Applying these three elements of an MDP to the agent-environment interaction results in the following trajectory of execution: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$

Transition function

The execution of an action $a \in A$ in a state $s \in S$ results in a transition from s to a new state s' and an assigned reward r. The transition function contains the probability distribution over the set of possible transitions and is defined as $T : S \times A \times S \to [0, 1]$. It can also be noted as a conditional probability.

$$T(s_t, a_t, s_{t+1}) = P(s_{t+1}|s_t, a_t) \tag{2.1}$$

The Markov property states that the conditional probability function of future states is only dependent on the present state, the preceding sequence of states is not to be considered. This is visible in equation 2.1, where no past states are taken into consideration.

To wrap it all up, an MDP is a tuple (S, A, R, T) in which S is a finite set of states, A a finite set of actions, R a reward function $R : S \times A \times S \to \mathbb{R}$, and T a transition function $T : S \times A \times S \times [0, 1]$.

[22][23][24]

2.2 Criterion of optimality

The goal of reinforcement learning is to learn an optimal policy that maps states to actions. The key questions here however are the definition of optimal in this context, and how this optimal policy will be found.

When considering MDPs, their goal of learning consists solely of gathering and maximizing rewards. [23] There are several models of optimality available for MDPs, but our attention will be restricted to the infinite-horizon discounted model shown in Equation 2.2, with $\mathbb{E}$ indicating expected return.

$$\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t] \tag{2.2}$$

The infinite-horizon model takes long-term reward into account, but the rewards that are received in the future are discounted according to how far away in time they will be received [24]. The impact of the discount factor γ, with $0 \leq \gamma < 1$, is increased as time goes on which means rewards gathered later on are discounted more. The discount factor itself will indicate how much the agent cares about rewards in immediate future or the distant future. A discount factor of $\gamma = 0$ indicates that the agent is myopic and only learns about actions that contribute immediate rewards. On the opposite, a discount factor of $\gamma = 1$ indicates that the agent will evaluate all the actions on the sum of all its future rewards.

A discounted model is attractive because it is more convenient to analyze. Discounting gives rise to a property that ensures that the sum of rewards is finite, even when dealing with an infinite horizon. [22]

2.2.1 Policy

As stated at the start of this section, the goal of reinforcement learning is to obtain an optimal policy mapping states to actions. Given an MDP tuple (S, A, R, T), a policy is a function that outputs for each state $s \in S$ an action $a \in A$ (or $a \in A(s)$). Policies can be either *deterministic* or *stochastic*. A stochastic policy maps states to probabilities of selecting each possible action and is hence defined as: $\pi : S \times A \rightarrow [0, 1]$. A deterministic policy maps a state to a single action and is defined as: $\pi : S \rightarrow A$.

Applying a policy to an MDP can be done as follows. The MDP will indicate an initial state s_0, for this state the policy π will suggest an action $a_0 = \pi(s_0)$ to perform. The execution of this action will trigger a transition to another state s_1. This state is based on the transition function T and the reward function R with probability $T(s_0, a_0, s_1)$ and reward $r_0 = R(s_0, a_0, s_1)$. Note that in the case of a deterministic policy, the probability $T(s_0, a_0, s_1)$ would equal 1. [22]

For finite MDPs, an optimal policy can be defined in the following way. Value functions define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. Namely $\pi > \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$. There will always exist one policy that is equal to or better than all the other policies. This policy is referred to as the optimal policy and is denoted as π^*. [24]

2.2.2 Value function

Value functions allow for criterion of optimality to be linked to policies [22]. Value functions provide an estimate as to how good it is for an agent to find itself in a certain state. The measure of goodness is expressed using criterion of optimality, such as Equation 2.2, that indicate the expected return for the infinite-horizon model. The value of a state $s \in S$ subject to policy π at time step t can be declared as:

$$V^\pi(s) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s] \tag{2.3}$$

This is equal to the expected reward when the agent starts in state s and follows policy π. The *state-action value function* is similar to the value function, on top of the state s it also takes action a into account:

$$Q^\pi(s, a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, a_t = a] \tag{2.4}$$

This is equal to the expected reward when the agent starts in state s, takes action a and from there on out follows policy π.

A crucial property of value functions is that they satisfy some recursive properties. *Bellman's principle of optimality* states that an optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision [25]. As suggested by this principal, the first reward or decision is considered separately from future actions or decisions. The future decisions can be replaced by a recursive definition of the value function, as seen in the transition from equation 1 to 2 of Equation 2.5:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s] \\ &= \mathbb{E}_\pi[r_t + \gamma V^\pi(s_{t+1})|s_t = s] \\ &= \sum_{s'} T(s, \pi(s), s')\Big(R(s, a, s') + \gamma V^\pi(s')\Big) \\ &= \sum_{s'} T(s, \pi(s), s')\Big(r + \gamma V^\pi(s')\Big) \end{aligned} \tag{2.5}$$

The end result is a *Bellman equation* and states that the expected value of state s is given by the immediate reward and values of possible next states (with discount factor) weighted by their transition probabilities. However, optimality is the goal and the best policy needs to be found. To achieve this goal, the value function from Equation 2.5 needs to be maximized for all states $s \in S$. As mentioned in Section 2.2.1, the policy π^* is optimal if $V^* \geq V^\pi(s)$ for all $s \in S$ and all policies π. The optimal solution $V^* = V^{\pi*}$ is called the *Bellman optimality equation* and is denoted as follows:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s, a, s')\Big(R(s, a, s') + \gamma V^*(s')\Big) \tag{2.6}$$

It states that under an optimal policy, the value of a state is equal to the expected return for the optimal action in said state. An equation to find the optimal state-action value can also be formed:

$$Q^*(s, a) = \sum_{s' \in S} T(s, a, s')\Big(R(s, a, s') + \gamma \max_{a'} Q^*(s')\Big) \tag{2.7}$$

Q-value functions can prove useful because a weighted sum over the alternatives regarding the transition function is not necessary anymore (see difference Equation 2.6 and Equation 2.7). Selection of an optimal action in a given state when the optimal value function is given, is possible using the following equation:

$$\pi^*(s) = arg \max_{a} \sum_{s' \in S} T(s, a, s')\Big(R(s, a, s') + \gamma V^*(s')\Big) \tag{2.8}$$

This can also be referred to as the *greedy policy*, because it selects the best action in each state using the value function. [22][24]

2.3 Solving MDPs

Solving an MDP equals finding the optimal policy π^*. There generally exist two classes of algorithms to tackle MDP solving, namely *model-based* or *model-free* algorithms. Model-based algorithms assume full knowledge of the MDP, which means that value functions and policies can be derived directly as in Section 2.2.2. These algorithms are often referred to as dynamic programming or DP algorithms. Model-free methods, on the other hand, do not assume full knowledge of the MDP. For these algorithms, the key component is interaction with the environment. Information is obtained by simulation of the policy, which results in samples of state transitions and rewards. These samples are then used to estimate state-value functions. The agent is thus effectively exploring the MDP to get the desired information, because none is already known. This class of algorithms is often called reinforcement learning or RL algorithms. [22][23]

2.3.1 Generalized policy iteration

Generalized policy iteration is an underlying mechanism that is present in all MDP solving methods and consist of two interacting processes, namely *policy evaluation* and *policy improvement*. Policy evaluation estimates the utility of the current policy π by computing V^π. The value function informs how good each state is and where possible improvements can be made. This information is useful for the policy improvement step, which attempts make the policy greedy and thus find actions that perform better than the action used by the current policy. Typically, this results in an improved policy π' over the current policy π using the information in V^π. [22]

As mentioned already, almost all reinforcement learning methods are well described as GPI. That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy [24]. The two steps of GPI will interact until both the evaluation and improvement process stabilize. If this happens, the value function and policy have reached an optimal state, as seen in Figure 2.2 a). Looking at Figure 2.2 b), the evaluation and improvement processes can be seen as both cooperating and competing. They go in opposite directions, because making the policy greedy makes the value function incorrect for the adjusted policy, and making the value function consistent with the policy makes it not greedy anymore. Eventually, the two steps will find a solution which encapsulates the optimal value function and policy. [22][24]

2.3.2 DP methods

To reiterate, dynamic programming methods are algorithms used to compute optimal policies when a perfect model of the environment as an MDP is given [24]. DP methods are not usually applied to reinforcement learning because full knowledge of the model is necessary and on top of that, they also tend to be computationally expensive. They are however fundamental to understand the methods explained in Section 2.3.3, because these attempt to achieve the same result, without having full knowledge of the model and with less computation. For DP methods, the environ-

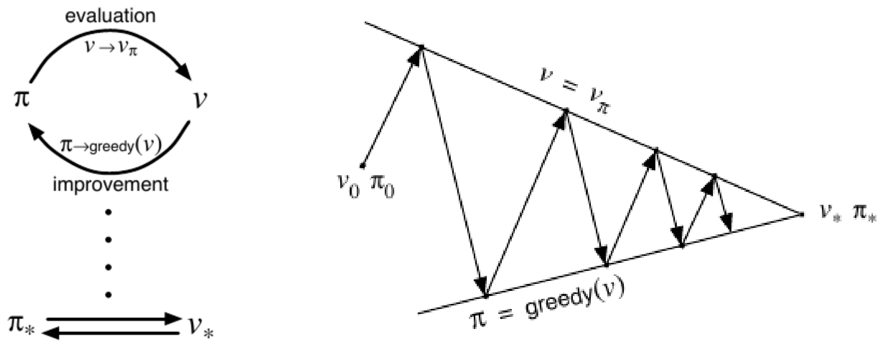


Figure 2.2: The alternation of evaluation and improvement steps until an optimal value function and policy are reached [24]

ment is assumed to be a finite MDP with discrete action and state values. The two essential DP methods are *policy iteration* and *value iteration*. [24]

Policy iteration

Policy iteration alternates the two GPI phases, *policy evaluation* and *policy improvement* until an optimal policy is achieved.

Policy evaluation

Policy evaluation makes the value function consistent with the current policy by computing V^π. This boils down to assessing the usefulness of the policy, and to see where the areas of improvement are. The first step is to find the value function V^π for an arbitrary policy π, which is called the *prediction problem*. Equation 2.5 mentioned for all $s \in S$:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[r_t + \gamma V^\pi(s_{t+1}) | s_t = s] \\ &= \sum_{s'} T(s, \pi(s), s')\Big(R(s, \pi(s), s') + \gamma V^\pi(s')\Big) \end{aligned} \tag{2.9}$$

When the dynamics of the system are known, Equation 2.9 is a system of $|S|$ simultaneous linear equations in $|S|$ unknowns (values of V^π for each $s \in S$). This is a problem that is pretty straightforward to solve, with iterative solutions being the most suitable. The Bellman equation is transformed into an update rule, which takes the current value function V^π_k and updates it by looking one step further:

$$V^\pi_{k+1}(s) = \sum_{s'} T(s, \pi(s), s')\Big(R(s, \pi(s), s') + \gamma V^\pi_k(s')\Big) \tag{2.10}$$

for all $s \in S$. As k grows to ∞, it can be shown V^π converges. [24]

Policy improvement

The goal of policy improvement is to make the policy greedy with respect to the current value function, also referred to as the *control problem.* This is done by checking for each state $s \in S$ if there is some other action $a \neq \pi(s)$ that outperforms the policy. To check if the selected action a is a better choice, select action a in state s and from there on out follow existing policy π. The state action value then is:

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}_{\pi}[r_t + \gamma V^{\pi}(s_{t+1}) | s_t = s, a_t = a] \\ &= \sum_{s'} T(s,a,s')\Big(R(s,a,s') + \gamma V^{\pi}(s')\Big) \end{aligned} \tag{2.11}$$

If $Q^{\pi}(s,a)$ is greater than $V^{\pi}(s)$, it is better to select action a and thus update current policy $\pi(s)$. This can be repeated for all the states $s \in S$, which results in selecting the best possible action in each state. The outcome is the *greedy* policy, as seen in Equation 2.12:

$$\begin{aligned} \pi'(s) &= arg \max_{a} Q^{\pi}(s,a) \\ &= arg \max_{a} \sum_{s'} T(s,a,s')\Big(R(s,a,s') + \gamma V^{\pi}(s')\Big) \end{aligned} \tag{2.12}$$

Policy iteration starts with an arbitrary policy π_0. The algorithm will first compute the value function for said policy and afterwards use this value function to improve the policy. This is repeated in an iterative manner until an optimal policy is achieved.

$$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow \pi_2 \rightarrow V^{\pi_2} \rightarrow ... \rightarrow \pi^*$$

Pseudo code for the policy iteration algorithm is shown in Algorithm 1. [22][24][26]

Value iteration

For value iteration, the evaluation and improvement steps of GPI are more intertwined and not clearly separated as with policy iteration. A drawback of the completely separated approach is that policy evaluation completely needed to converge, and this only happens when its computed in the limit. However, it is not needed to wait for full convergence, as it is possible to truncate the policy evaluation process and improve the policy on the information that is gathered up to that point. Truncating the evaluation process to just one step is called *value iteration.* Value iteration immediately merges the policy improvement step into the iterative policy evaluation step. To do this, the Bellmann optimality equation seen in Equation 2.6, is transformed into an update rule:

$$\begin{aligned} V_{t+1}(s) &= \max_{a} \sum_{s'} T(s,a,s')\Big(R(s,a,s') + \gamma V_t(s')\Big) \\ &= \max_{a} Q_{t+1}(s,a) \end{aligned} \tag{2.13}$$

Algorithm 1: Policy iteration

```
input  : $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
output: $\pi^*$

1  **POLICY EVALUATION**
2  repeat
3      $\Delta \leftarrow 0$
4      foreach $\in S$ do
5          $v \leftarrow V(s)$
6          $V(s) \leftarrow \sum_{s'} T(s, \pi(s), s')\Big(R(s, \pi(s), s') + \gamma V^{\pi}(s')\Big)$
7          $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
8  until $\Delta \leq \theta$ (small positive number)
9  **POLICY IMPROVEMENT**
10 $policy\_stable \leftarrow true$
11 foreach $s \in S$ do
12     $a \leftarrow \pi(s)$
13     $\pi(s) \leftarrow arg\max_a \sum_{s'} T(s, a, s')\Big(R(s, a, s') + \gamma V^{\pi}(s')\Big)$
14     if $a \neq \pi(s)$ then $policy\_stable \leftarrow false$
15 if $policy\_stable$ then stop
16 else go to policy evaluation
```

Value iteration, just like policy evaluation, converges when the number of iterations grows to inf. Starting with a value function V_0, the value of each state is updated in time step t using Equation 2.13. This results in a sequence of value functions as follows:

$$V_0 \to V_1 \to V_2 \to V_3 \to ... \to V^*$$

Equation 2.8 can be applied to extract the π^* from V^*. The policy iteration pseudo code is shown in Algorithm 2. [22][24]

Algorithm 2: Value iteration

```
  input  : $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
  output: $V^*$
1 repeat
2     $\Delta \leftarrow 0$
3     foreach $\in S$ do
4         $v \leftarrow V(s)$
5         $V(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\Big(R(s,a,s') + \gamma V^{\pi}(s')\Big)$
6         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
7 until $\Delta \leq \theta$ (small positive number)
```

2.3.3 RL methods

RL methods do not assume full knowledge of the MDP model. A direct consequence of this is that the MDP needs to be explored and sampled to create experience, from which can be learned. There generally exist two approaches in terms of learning for RL methods. The first option learns the model from interaction with the environment until approximately or sufficiently complete. From here on out, the DP methods from Section 2.3.2 can be applied. This method is called *indirect* RL. The second option, which is called *direct* RL, will skip the model estimation and directly estimate values. Most model-free methods apply direct RL. [22]

Exploration

An important trade-off regarding the RL methods is *exploration* vs *exploitation*. The goal of the learning agent is to obtain rewards, and to achieve that goal, the agent's knowledge needs to be exploited. It is important, however, that the agent sometimes attempts other actions that are not in its current knowledge, that could possibly improve performance and provide a greater reward. The question that appears now is how much exploiting or exploring is needed? As humans, we try to obtain as much information as possible before making a choice, which is unfortunately not feasible in reinforcement learning. [27]

The work of Thrun [28] made a distinction between *directed* and *undirected* exploration methods. Undirected exploration techniques will explore an environment based on randomness. The most uninformed undirected exploration technique is *random exploration.* Actions are randomly generated with uniform probability distribution, regardless of expected costs or rewards. Most other undirected exploration techniques will take costs into account during learning. Typically, expected costs or rewards for actions (Q-values) are assessed by a controller. Actions are still randomly selected, but the expected rewards are used to draw the probability distribution that takes care of action selection. If the expected reward for an action is high, the probability it gets chosen is also higher. There are two ways to modify these probability distributions: *semi-uniform distributed exploration* and *Boltzmann-distributed exploration.* The former of these two is also called the *max-random exploration rule* or *ϵ-greedy.* A single parameter P_{max} is used that indicates the probability of selecting the best action:

1. A random number x is generated from the uniform distribution [0,1]
2. If $x \leq P_{max}$:
   (a) Action with the highest Q value is chosen
   (b) Else, an arbitrary action is chosen

Thus, there is P_{max} chance for exploitation, and $1 - P_{max}$ chance for exploration. The Boltzmann-distributed exploration is similar to ϵ-greedy. The action selection is still random, but probabilities to select an action are weighted by their relative Q values, which increases the chance of choosing good actions. The rule computes the probability $P(a|s)$ for action a and state s and Q-values $Q(s,i)$ for all $i \in A$:

$$P(a|s) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_i e^{\frac{Q(s,i)}{T}}} \tag{2.14}$$

The T parameter is the *temperature* and is used to anneal the exploration. When T is high, the selection strategy acts more random, while lower values of T cause a greedy strategy. [22][28][29]

Directed exploration methods use some exploration-specific knowledge to guide exploration search, which makes random action selection unnecessary. The rule will directly decide which action should be taken next to explore the environment in an optimal way. An exploration reward function needs to be created so that direct exploration methods can learn the *exploration value function*, just like regular reinforcement learning methods learn a value function. [28][29]

Depending on the specific reinforcement learning algorithm used, some other, more specific exploration techniques may be applied. These will be explained when necessary later on.

Monte Carlo methods

Monte Carlo methods present a first way to estimate value functions to discover optimal policies, without having knowledge of the full model. Monte Carlo methods work by averaging sample returns. Frequency counts of transitions and rewards are kept and values are based on these estimates. For example, for each state $s \in S$, all the obtained returns from s are stored, and the value of state s becomes the average of all those values. Monte Carlo methods work well for episodic tasks, which means that the learning process can be separated in different episodes that eventually terminate. An episode terminates when the learning goal is reached. This also means that samples from complete returns can be used when an episode is terminated.

Suppose we want to estimate the value of state s under policy π given by $V_\pi(s)$. By following π and passing through s, a set of episodes was obtained. Every time the state s occurs in an episode, it is referred to as a *visit*. This may happen more than once, but the first time is called the *first visit*. *First-visit* MC methods estimate $V_\pi(s)$ by averaging the returns obtained from the first visit to state s for every episode. On the contrary, *every-visit* MC methods estimate $V_\pi(s)$ by averaging over all visits to s. Both of these methods will converge to $V_\pi(s)$ over time when s goes to infinity. Similarly, MC methods can also be used to find action values. This is especially handy when a model is unavailable, as is the case with RL methods. Without a model, state values are not sufficient to find the optimal policy. To make state values useful to find a policy, the value of each action has to be estimated. The goal of MC methods is thus to find the Q^* or the optimal action value function. Following the principle of GPI as seen in Section 2.3.1, the first step is to look at policy evaluation. $Q_\pi(s, a)$ needs to be estimated, which represents the expected return if the current state is s and action a is taken. Afterwards, policy π will be followed. The same method is applied as for the value function V, only now visits to state-action pairs are taken into account. However, if the policy is deterministic, many possible state-action pairs are never visited. To ensure enough exploration, *exploring starts* are applied. With this method, each state-action pair has a non-zero probability to be the initial pair. When policy evaluation is complete, the policy can be made greedy by executing policy improvement. Because there is no model, the state-action values are used for this purpose. [22][24]

Temporal difference learning

Temporal difference learning is described as the one idea that is central and novel to reinforcement learning [24]. It combines ideas of Monte Carlo methods and dynamic programming. Just like with MC methods, TD methods learn directly from experiences without a model of the environment present. Similar to DP methods, TD methods update estimates based on other estimates that were previously learned, which is called *bootstrapping*. An intuitive example of this concept is hosting a dinner for some guests at your place. You have to inform the guests at what time they should ideally arrive. This point in time is dependent on all the preparation that needs to happen to make the dinner possible. Say you have to visit the supermarket, the butcher and the bakery in that specific order. You have general ideas of how long the driving will take from each place to the next one, and also how long each activity will occupy. Based on this information, you can inform the guests when they are allowed to arrive. Due to unforeseen circumstances, one of the activities takes twice as long, and your initial predictions won't work anymore. You can then adjust the estimate based on your findings or experiences along the way, which is the main principle of temporal difference learning. [22] Combining the advantages of both Monte Carlo methods and Dynamic Programming methods results in an approach that does not require a model of the MDP and does not need full sweeps of the state space to learn. [22][24]

On-policy vs off-policy

This principle is also present with Monte Carlo methods, but is explained here because it is especially relevant for the different TD algorithms. The goal is to learn an optimal policy and to reach that goal, experiences are needed. These experiences can come from following a policy μ that is different from the current policy π. In this case π is the *target* policy because the end goal is to find the value function for this policy and in turn use the value function to find the optimal policy π^*. The policy μ that is followed to sample experiences from, is called the *behavior* policy because it controls the learning agent and generates its behavior. When a target policy is used, the learning is referred to as *off-policy*. For *on-policy* methods, the target and behavior policy are the same one. [24]

TD(0)

To solve the prediction problem and find the value function V, TD(0) applies an incremental update rule:

$$V_{k+1}(s) := V_k(s) + \alpha\Big(r + \gamma V_k(s') - V_k(s)\Big) \tag{2.15}$$

This rules allows policy evaluation to be executed with $\alpha \in [0, 1]$ being the *learning rate* that determines by how much values get updated. To reach convergence, the learning rate has to be gradually decreased during learning. [24]

Now that the prediction problem is solved by TD(0), the next step is to tackle the control problem. This process drives the policy to improve locally with respect to the current value function. Two algorithms will be explained that learn the Q-functions directly from samples. This makes the presence of a transition model for action selection unnecessary. [22]

SARSA

SARSA stands for State-Action-Reward-State-Action. The goal for this on-policy method is to estimate $Q_\pi(s, a)$ for the current behavior policy π and all states s and actions a. This can be done in a way that is almost identical to equation 2.15 that was used to learn V_π. An episode alternates between states and actions, with a reward connected to each action. Instead of considering state to state transitions to learn state values, now transitions between state-action pairs are considered, and values of those are learned (Q-value). This is done with the following equation:

$$Q_{t+1}(s_t, a_t) := Q_t(s_t, a_t) + \alpha\Big(r + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)\Big) \tag{2.16}$$

with a_{t+1} being the action that is executed by the current policy in state s_{t+1}. Algorithm 3 shows the full algorithm. SARSA is most useful in non-stationary environments where an optimal policy is almost never reached. [22][24]

Algorithm 3: SARSA algorithm

```
input  : Initialize $Q(s, a)$, arbitrarily for all $s \in S, a \in A$
output: Q value function
1 foreach episode do
2     choose $a$ when in state $s$, according to the policy derived from Q
3     repeat
4         execute $a$, observe $r, s'$
5         choose $a'$ when in state $s'$, according to the policy derived from Q
6         $Q(s, a) \leftarrow Q(s, a) + \alpha\Big(r + \gamma Q(s', a') - Q(s, a)\Big)$
7         s $\leftarrow s'; a \leftarrow a'$
8     until $s$ is terminal / goal state
```

Q-learning

Q-learning was one of the most important breakthroughs in reinforcement learning. It is an off-policy TD algorithm developed by Watkins [30][31]. The incremental update rule is a variation on the theme of TD learning as seen in Equation 2.15:

$$Q_{k+1}(s_t, a_t) := Q_k(s_t, a_t) + \alpha\Big(r_t + \gamma \max_a Q_k(s_{t+1}, a) - Q_k(s_t, a_t)\Big) \tag{2.17}$$

Q-values for actions are incrementally estimated based on rewards and the Q-value function of the agent. When the agent executes action a_t and moves from state s_t to state s_{t+1}, the update takes place on the Q-value of action a_t in state s_t, according to the reward received. To reach convergence and thus the optimal policy, the only assumption is that all state-action pairs are visited an infinite amount of times. As a result, Q-learning is *exploration-insensitive* because it will always reach convergence if said assumption holds, independent of the exploration technique that is used. Q-learning is an off-policy algorithm because it computes Q-values according to a greedy policy, but this policy is not necessarily followed by the agent. The complete algorithm is shown in Algorithm 4. [22][24]

Algorithm 4: Q-learning [30][31]

```
input : discount factor $\gamma$, learning rate $\alpha$
output: Q value function, Q* if every state-action pair is visited infinite
        amount of times
1 foreach episode do
2     $s$ is initialized as starting state
3     repeat
4         choose action $a \in A(s)$ based on exploration
5         take action $a$
6         $Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma \max_{a' \in A(s')} Q(s',a') - Q(s,a)\Big)$
7         s $\leftarrow s'$
8     until $s$ is terminal / goal state
```

Policy-based RL

Up until this point, every algorithm we discussed applies a *value-based* reinforcement learning approach. The goal of all those algorithms was to find the value or state-action value function, and derive the policy from these functions. *Policy-based* reinforcement learning aims to learn the policy directly, and thereby skips the process of calculating the value functions first. Figure 2.3 displays an overview of the categories of reinforcement learning methods. Generally, policy based methods are subdivided into two categories: *gradient-based* and *gradient-free.* Gradient-free approaches will not be discussed because they serve no purpose for this research and are not used often in practice.

Gradient-based methods learn the policy directly with a parametrized function respect to θ, $\pi(a|s;\theta)$. [33] The expected return for a set of parameters θ is given by $J(\theta)$ and subsequently maximized by using gradient ascent. The expected return is defined as follows for episodic environments:

$$J(\theta) = V_{\pi\theta}(S_1) = \mathbb{E}_{\pi_\theta}[V_1] \tag{2.18}$$

with S_1 as the starting state. In gradient ascent, the update direction of the parameter is given by the gradient $\nabla_\theta J(\theta)$ because it points in the direction of the steepest ascent of the expected return. [34] The policy gradient update is given by:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t) \tag{2.19}$$

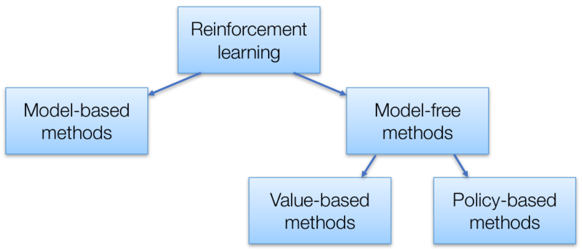


Figure 2.3: An overview of discussed reinforcement learning methods [32]

with α being the learning rate and $\nabla_\theta J(\theta)$ being the policy gradient:

$$\nabla_\theta J(\theta) = \Big(\frac{\delta J(\theta)}{\delta\theta_1} \dots \frac{\delta J(\theta)}{\delta\theta_n}\Big)^T \tag{2.20}$$

Thus, the goal of policy gradient is to change the policy parameter θ in a way that ensures improvement. The problem, however, is that performance depends on selected actions and distribution of the states in which those selections are made, and that both of these are affected by the policy parameter θ. A way needs to be found so that the gradient can be estimated with respect to the policy parameter θ when it depends on the unknown effect of policy changes on the state distribution. The *policy gradient theorem* by Sutton & Barto solves this problem. It provides a reformation so that the gradient can be expressed with respect to the policy parameter θ, in a way that does not involve the derivative of the state distribution [24]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla \ln \pi(a|s,\theta) Q_\pi(s,a)] \tag{2.21}$$

See [24] for details and proof for the theorem.

REINFORCE

REINFORCE is also known as Monte-Carlo policy gradient. This method relies on the estimated return $Q_\pi(s,a)$, provided by Monte Carlo methods using episode samples, to update policy parameter θ. The REINFORCE algorithm applies the policy gradient theorem and also commonly applies a baseline. A baseline value is substracted from the return G_t to reduce the variance of the gradient estimation without changing the bias. A baseline value that is often applied is the value function V. The gradient ascent update would then use $A(s,a) = Q(s,a) - V(s)$, which is also referred to as the advantage function. The advantage function takes the difference of the q-value and the average of the actions the agent would have taken in that state. It gives an indication what the extra reward could be if the agent takes a certain action. [33] Algorithm 5 shows the pseudo code.

Algorithm 5: REINFORCE [24][35]

```
input  : A differentiable policy paramerization $\pi(a|s;\theta)$
output: Approximation of policy parameter $\theta$
1 Initialize policy parameter $\theta$ at random
2 repeat
3     Generate an episode $s_0, a_0, r_0, \dots, s_{t-1}, a_{t-1}, r_t$ following $\pi(\cdot|\cdot;\theta)$
4     foreach step of episode $t = 0, \dots, t-1$ do
5         $G \leftarrow$ return from step $t$ (estimate return $G_t$ since time step $t$)
6         $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi(a_t|s_t;\theta)$
7 until forever
```

Actor-critic

Actor-critic algorithms combine both value-based and policy-based methods, as displayed in Figure 2.4. It can be seen as an on-policy TD approach that keeps a separate memory structure to represent a policy independent of the value function. This policy is referred to as the *actor*, while the estimated value function is called the *critic*. This is the case because the actor or policy will select the actions and the estimated value function criticizes the actions executed by the actor. This type of learning is always on-policy because the critic has to learn the policy that is being followed by the actor. Usually the critic will be a state-value function, and evaluating the new state after an action has been performed is done with the TD-error:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.22}$$

A positive error indicates that the action a_t taken in state s_t had good results and should be consulted in the future, whereas a negative error lessens the chance that the action will be taken in the future. [22] Concretely, the following roles are taken on by the actor and the critic:

- **Critic:** updates the value function parameters w (can be state-value $V(s, w)$ or action-value function $Q(a|s, w)$)

- **Actor:** updates policy parameters θ for $\pi(a|s, \theta)$ with gradient ascent, in the direction suggested by the critic

[33] Actor-critic methods can be especially useful when the action space is very large or continuous, because the extra policy ensures that for action selection, not every actions' Q-value needs to be calculated. They are also capable of learning stochastic policies naturally. [22] Algorithm 6 shows the pseudo code.

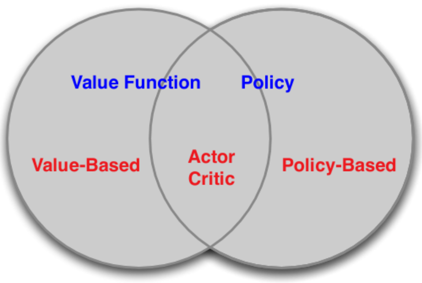


Figure 2.4: Position of the actor-critic approach in the reinforcement learning overview [36]

Advantages

Policy-based methods can naturally handle continuous state spaces and are effective in high-dimensional or continuous action spaces. They also have better convergence properties and are capable of learning appropriate levels of exploration. Disadvantages, however, include converging to local optimum rather than a global one. Also, policy evaluation is typically inefficient and results in high variance. [24][36]

2.3.4 Problems with RL

The main problem with standard reinforcement learning is the Curse of Dimensionality. This concept was introduced by Richard Bellman [25] and expresses the difficulty of optimizing a function that has a large amount of input variables. Data always has a dimensionality, but when the number of dimensions increases, the difficulty to find patterns and solve problems also tends to increase. Say you have to retrieve a coin lost somewhere along a line with a length of 100 meters. This is fairly simple to solve, but what if you need to search the coin in a 100x100 meters field. The difficulty has already increased significantly. Now image it being a cube 100 meters across. As more dimensions or input variables are added, it gets a lot more difficult to search through the space. Data also becomes a lot more sparse when the dimensionality increases. When there are more features than available data samples, overfitting can happen. Luckily, this is not really an issue for reinforcement learning, because data is generated by interacting with the environment.

How does one combat the curse of dimensionality? The most obvious option is to achieve density reduction. A set of core features is selected while the rest is discarded. The curse of dimensionality can also be made ineffective by applying deep learning.

Algorithm 6: Actor-critic [24]

```
input  : A differentiable policy paramerization $\pi(a|s;\theta)$
input  : A differentiable state-value paramerization $\hat{V}(s;w)$
input  : Learning rates $\alpha^\theta > 0$ and $\alpha^w > 0$
output: Approximation of policy parameter $\theta$
1  Initialize policy parameter $\theta$ and state-value weight $w$ at random
2  repeat
3      Initialize s (first state of episode)
4      while $s$ is not terminal (for each time step) do
5          sample action $a \sim \pi(\cdot|s;\theta)$
6          take action $a$, observe $r, s'$
7          compute TD-error:
8            $\delta \leftarrow r + \gamma\hat{V}(s';w) - \hat{V}(s;w)$
9          update value function and policy parameters:
10           $w \leftarrow w + \alpha^w \delta \nabla_w \hat{V}(s;w)$
11           $\theta \leftarrow \theta + \alpha^\theta \delta \nabla_\theta \ln \pi(a|s;\theta)$
12         s $\leftarrow s'$
13 until forever
```

2.4 Deep RL

Deep reinforcement learning has and is still revolutionizing the field of AI. Deep learning enables reinforcement learning to scale to problems that were unmanageable before. Standard RL approaches face the same complexity issues as most other algorithms: memory complexity, computational complexity, and in the case of machine learning, sample complexity. This results in bad scalability and limitation to low-dimensional problems. Deep learning can take advantage of two powerful properties of deep neural networks: *function approximation* and *representation learning.* The most important property is that neural networks are capable of learning low-dimensional representations of high-dimensional data. In turn, this makes the curse of dimensionality not applicable anymore. [37] Section 3 provides more information on the topic of deep neural networks.

Reinforcement learning methods turn into deep reinforcement learning methods when neural networks are used to represent the state or observation, and/or for function approximation of the value function, $V(s;\theta)$ or $Q(s,a;\theta)$, or the policy $\pi(a|s;\theta)$. In the cases of function approximation, the parameters θ represent the weights of the deep neural networks. [38] A number of relevant deep RL algorithms will be discussed that apply neural networks to algorithms or principles that were previously explained in this section.

2.4.1 Deep Q Network

Theoretically, the optimal action-value function Q^* could be memorized for all state action-pairs in a large table, which is the standard Q-learning approach. This approaches' likelihood of success quickly turns improbable when the state and action space grow very large. A logical solution to this problem is to apply Q-function approximation. DQN is a technique developed by Mnih et al. [39][40] that tackles $Q-$function based deep reinforcement learning. It was the first RL algorithm that successfully learned from raw visual inputs in a large number of different environments. The algorithm was trained to learn classic Atari 2600 games, with the only provided inputs being raw pixels and the game score. The learning agent displayed behavior that exceeded performance of previous algorithms, and equaled results of professional players. Before DQN, approximating Q-functions with nonlinear approximation methods resulted in very unstable behavior. These instabilities are mostly caused by correlations present in observed data, significant policy changes as a result of small updates to the Q-function which in turn changes how the data is distributed, and correlations present between Q-values and target values $r+\gamma Q(s',a')$. [40]

The goal of DQN is to find the optimal action value function $Q*$. Section 2.2.2 addressed the fact that the Q-function obeys the Bellman equation based on the following intuition: if the optimal value $Q^*(s',a')$ of a sequence s' at the next timestep is known for every possible action a', then to get the optimal scenario, the action a' has to be selected that maximizes the expected value of $r+\gamma Q^*(s',a')$ (target value):

$$Q^*(s,a) = \mathbb{E}_s[r + \gamma max_{a'} Q^*(s',a')|s,a] \tag{2.23}$$

To achieve the optimal Q-value function, the Bellman equation can be used to construct an iterative update:

$$Q_{i+1}(s,a) = \mathbb{E}_s[r + \gamma max_{a'} Q_i(s',a')|s,a] \tag{2.24}$$

When i goes to ∞, Q_i would result in Q^*. However, this is not practical, so DQN applies non-linear function approximation by neural networks to estimate the action-value function: $Q(s,a;\theta) \approx Q^*(s,a)$. The Q-network is trained to minimize a sequence of loss functions by gradient descent, with $U(D)$ indicating a uniform random draw of samples from the replay buffer:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right] \tag{2.25}$$

[39]

DQN applies two solutions to combat the instabilities of using non-linear function approximation: *experience replay* and *target networks.* Experience replay saves episode steps or observation sequences $e_t = (s_t, a_t, r_t, s_{t+1})$ in the *replay buffer* $D_t = \{e_1, \ldots, e_t\}$. These observations are then sampled randomly to remove correlations in observed data. This technique also improves data efficiency and smooths the data distribution. Target networks aim to decrease the correlation between Q-values and target values $r + \gamma Q(s', a')$. A separate network is kept with its own network parameters θ', and this separate or target network is only updated periodically. The parameters of the original Q-network θ are cloned and used as parameters for the target network. The parameters of the target network θ^- are kept frozen for a certain amount of time steps. After the time steps have expired, the target network's parameters are updated again. The pseudo code for the full DQN algorithm is shown in Algorithm 7. DQN can be applied to problems with discrete action spaces and discrete or continuous state spaces. The learning process is off-policy because target networks are used. [37][38][40]

Algorithm 7: DQN [38][40]

```
input  : Pixels and game score
output: $Q$-function (used for policy extraction and action selection)
1  Initialize replay buffer $D$
2  Initialize $Q$-function with random weights $\theta$
3  Initialize target $\hat{Q}$-function with weights $\theta^- = \theta$
4  for episode = 1 to $M$ do
5      initialize sequence $s_1 = x_1$
6      for $t$=1 to $T$ do
7          select $a_t = \begin{cases} \text{a random action with probability } \epsilon \\ \arg\max_a Q(s_t, a; \theta) \text{ otherwise} \end{cases}$
8          execute action $a_t$ and observe reward $r_t$ and image $x_{t+1}$
9          set $s_{t+1} = s_t, a_t, x_{t+1}$
10         store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$
11         ** EXPERIENCE REPLAY **
12         sample random minibatch of transitions from $D$
13         set $y_j = \begin{cases} \mathrm{r}_j \text{ if episode terminates at step } j+1 \\ \mathrm{r}_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) \text{ otherwise} \end{cases}$
14         perform gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. network
             parameter $\theta$
15         ** PERIODIC UPDATE TARGET NETWORK **
16         in every $C$ steps, reset $\hat{Q} = Q$ by setting $\theta^- = \theta$
```

2.4.2 Deep Deterministic Policy Gradient

DQN made it possible to solve problems with high-dimensional observation spaces, however, it can only handle discrete and low-dimensional action spaces. It is also impossible to apply DQN to continuous action domains because it relies on finding the action that maximizes the action-value function, which in turn requires an iterative optimization process at every step. One possible solution to apply DQN to continuous domains, is to simply make the domain discrete. This comes with many limitations, the main one being the curse of dimensionality. The number of possible actions will increase exponentially with the amount of degrees of freedom. If there are 8 degrees of freedom (or 8 different values for one action), and for each of those 8 values the coarsest discretization is chosen: $a_i \in \{-k, 0, k\}$, there are $3^8 = 6561$ options. The actions space will thus have a dimensionality of 6561. When the discretization isn't as coarse, it will be even worse. Exploring such large action spaces efficiently is very challenging or even impossible. Making action spaces discrete also risks losing important information that may be necessary to reach a solution. [41]

DDPG is based on the Deterministic Policy Gradient algorithm [42], which also uses an actor-critic approach. The actor is a parametrized function $\mu(s;\theta^{\mu})$ that expresses the policy by mapping each state to an action in a deterministic manner. In turn, the Bellman equation is used to learn critic $Q(s,a)$, just as with Q-learning. DDPG follows suit but adds deep function approximators to learn both actor and critic. To combat the instabilities that function approximators bring, the same adjustments as with DQN are employed, namely the replay buffer and target networks. The replay buffer operates in the same manner, observation sequences (s_t, a_t, r_t, s_{t+1}) are saved in a finite sized replay buffer D. When the buffer is full, the oldest samples will be thrown out. At each timestep, the actor and critic will be updated by uniformly sampling a minibatch from the buffer. The target networks approach is altered slightly. Soft updates are used instead of directly copying the parameters or weights. A copy is created of the actor and critic networks $Q'(s,a;\theta^{Q'})$ and $\mu'(s;\theta^{\mu})$. The weights of these target networks are then slowly updated by tracking the learned networks in the following way: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau << 1$. Hence, the target networks are time-delayed copies of the original ones. DDPG uses a total of four networks:

- θ^{Q}: Q-network
- θ^{μ}: Deterministic policy network
- $\theta^{Q'}$: Target Q-network
- $\theta^{\mu'}$: Target policy network

The Q-learning side of DDPG is almost the same as with DQN. However, counting the maximum over actions of the target, as with Equation 2.25, is a challenge for continuous action spaces. To solve this issue, the target policy network is used to approximately maximize $Q(s', a'; \theta^{Q'})$. The resulting loss function for the Q-learning part at iteration i is:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(Q(s,a;\theta_i^{Q}) - r + \gamma Q'(s', \mu'(s';\theta_i^{\mu'});\theta_i^{Q'})\right)^2\right] \tag{2.26}$$

To learn the policy, a deterministic policy $\mu(s;\theta^{\mu})$ needs to be found that maximizes $Q(s,a;\theta^{Q})$. Because we are dealing with continuous action spaces, it can be assumed that the Q-function is differentiable with respect to the action. [43] Subsequently, gradient ascent can be performed with respect to the actor/policy parameters:

$$\begin{aligned} \nabla_{\theta^{\mu}} J &\approx \mathbb{E}_s[\nabla_{\theta^{Q}} Q(s,a;\theta^{Q});_{s=s_t,a=\mu(s_t;\theta^{\mu})}\,] \\ &= \mathbb{E}_s[\nabla_a Q(s,a;\theta^{Q});_{s=s_t,a=\mu(s_t)}\,\nabla_{\theta_{\mu}}\mu(s;\theta^{\mu});_{s=s_t}\,] \end{aligned} \tag{2.27}$$

[41]

Exploration can be a tricky subject when dealing with continuous action spaces. Luckily, because DDPG is an off-policy algorithm, exploration can be treated separate from the learning problem. When selecting an action according to the current policy, noise sampled from noise process $\mathcal{N}$ is added to the action:

$$a = \mu(s_t;\theta_t^{\mu}) + \mathcal{N} \tag{2.28}$$

The DDPG implementation uses an Ornstein-Uhlenbeck process to generate noise, which is a time-correlated noise process. However, recent results suggest that uncorrelated noise processes like mean-zero Gaussian noise work equally well. Algorithm 8 shows full pseudo code for DDPG. [41][43]

Algorithm 8: DDPG [41]

```
1  Initialize replay buffer $D$
2  Initialize actor network $\mu(s;\theta^{\mu})$ and critic network $Q(s,a;\theta^{Q})$ with random
     weights $\theta^{\mu}$ and $\theta^{Q}$
3  Initialize target network $\mu'$ and $Q'$ with weights $\theta^{\mu'} \leftarrow \theta^{\mu}$, $\theta^{Q'} \leftarrow \theta^{Q'}$
4  for episode = 1 to M do
5      initialize random process $\mathcal{N}$ for action exploration
6      receive initial state $s_1$
7      for t=1 to T do
8          select $a_t = \mu(s_t;\theta^{\mu}) + \mathcal{N}_t$ according to policy and exploration noise
9          execute action $a_t$ and observe reward $r_t$ and image state $s_t$
10         store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$
11         sample random minibatch of $N$ transitions from $D$
12         set target $y_j = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1};\theta^{\mu'});\theta^{Q'})$
13         update critic by minimizing loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i,a_i;\theta^{Q}))^2$
14         update actor policy using sampled policy gradient:
15           $\nabla_{\theta^{\mu}} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a;\theta^{Q});_{s=s_i,a=\mu(s_i)}\,\nabla_{\theta^{\mu}}\mu(s;\theta^{\mu});_{s_i}$
16         update target networks:
17           $\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$
18           $\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$
```

2.4.3 Proximal Policy Optimization

PPO was introduced by OpenAI as a RL algorithm that performs comparably or better than state-of-the-art approaches, while being much simpler to implement and tune. Policy gradient methods have become a staple for deep reinforcement learning. Unfortunately, getting good results is not always guaranteed. These algorithms tend to be very sensitive to the choice of certain parameters like step size. Besides, their sample efficiency is almost always poor, simple tasks require too much timesteps to achieve good results. According to OpenAI, PPO succeeds in finding a good balance between ease of implementation, sample complexity, and ease of tuning. [44][45] The question that drove the development of PPO was: how can the biggest possible improvement step on a policy be taken with the available data, without degrading the performance by taking it a step too far? [46] The two main contributions that PPO brings to the table are the *Clipped Surrogate Objective* and the use of multiple epochs of stochastic gradient ascent to perform each update.

Standard policy gradient methods operate by computing an estimator for the policy gradient, and subsequently applying a stochastic gradient ascent algorithm to solve it. The most common estimator is:

$$\hat{g} = \hat{\mathbb{E}}_t\big[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t\big] \tag{2.29}$$

which displays a similar equation as Equation 2.21. π_θ is a stochastic policy and $\hat{A}_t$ estimates the advantage function at a timestep t. When automatic differentiation methods like neural networks are used, an objective function is setup such that the gradient of this function acts as the policy gradient estimator. To get the estimator $\hat{g}$, the following objective has to be differentiated:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t\big[\log \pi_\theta(a_t|s_t)\hat{A}_t\big] \tag{2.30}$$

Executing multiple optimization steps on this loss function while applying the same trajectory is not a suitable approach, because it will likely lead to large policy updates that ruin performance or results. Just as PPO, Trust Region Policy Optimization (TRPO) aims to achieve the biggest possible improvement step with the current available data without stepping too far. The vanilla policy gradient estimator given in Equation 2.29 uses the log probability to trace the impact of each action. TRPO replaces the log probability by the probability of the action under the current policy divided by the probability of the action under the previous policy. This objective is then referred to as the *surrogate* objective function:

$$L^{TRPO}(\theta) = \hat{\mathbb{E}}_t\big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\big] = \hat{\mathbb{E}}\big[r_t(\theta)\hat{A}_t\big] \tag{2.31}$$

If this probability fraction is larger than 1, the action is more likely to occur under the current policy. When it is between 0 and 1, it is more probably for the old policy than for the new. The problem with such an approach is that, when actions are way more likely to happen under the current policy, this probability fraction can grow really large and the gradient update steps can degrade your policy and thus also results. TRPO attempts to solve this issue by adding constraints or penalties. PPO tries another route and adjusts the objective function 2.31 by adding clipping:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\big] \tag{2.32}$$

with ϵ as adjustable hyperparameter. Using this clipping mechanism, the values of r_t in the clip term will always be in the interval $[1-\epsilon, 1+\epsilon]$. Figure 2.5 shows the clipping mechanism in action when $A > 0$ and the action has an estimated positive outcome, and when $A < 0$ and the action has an estimated negative outcome. For the left plot, the value of r_t is clipped if it becomes too large. This is when the action is more probable to happen under the current policy. To avoid stepping too far and prevent inaccuracy, the objective is clipped. The same holds when $A < 0$, only now the value of r_t is clipped when it becomes too small. Both clipping actions prevent stepping too far and making the policy worse.

How is it that r_t can grow indefinitely large on the right diagram? A large r_t value in this case would make the current action a lot more probable, while it makes the policy worse because $A < 0$. Such an action needs to be undone, and this is the reason for the minimization. The value of L^{CLIP} is negative in this case and will be picked by the minimization over the clipped r_t value. This negative value causes the gradient to move in the other direction and as a consequence make the action less probable. Hence, the minimization term allows undoing of bad actions.

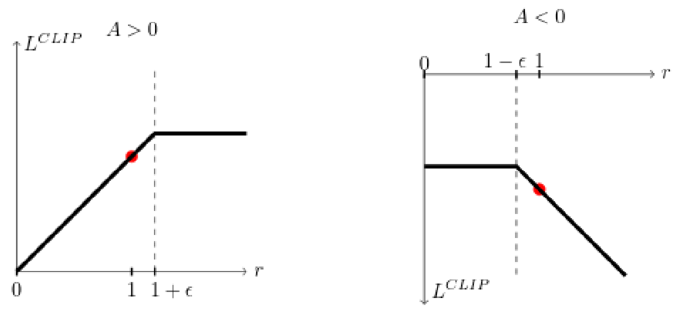


Figure 2.5: Plot showing surrogate function L^{CLIP} as a function of r for positive advantages $A > 0$ and negative advantages $A < 0$ [45]

Because of the *Clipped Surrogate Objective*, multiple gradient ascent epochs can be performed on the same samples without creating very large policy updates that destroy performance or results. In turn, this improves sample efficiency. PPO employs N amount of actors to collect T timesteps of data. The surrogate loss on NT timesteps of data is constructed, and subsequently optimized with minibatch stochastic gradient descent, for K epochs as follows:

```
For K epochs:
    Randomize train batch of size $NT$
    For each minibatch of size $M$ in train batch:
        optimize minibatch SGD
```

PPO is an on-policy algorithm. One could argue that approximating the expected return of the policy π_θ with samples from a slightly older policy $\pi_{\theta_{old}}$ is off-policy, but that's not really off-policy in the conventional sense. PPO is suitable for continuous state and action spaces. Algorithm 9 shows the full pseudo code. [45]

Algorithm 9: PPO [45]

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

2.4.4 Problems with deep RL

Reinforcement learning, and especially deep reinforcement learning is surrounded by a lot of hype. Reinforcement learning is a very useful paradigm, and coupling it with deep learning makes its strengths even greater. However, it just does not really work yet. A lot of people, including me, fall into the same trap when looking at reinforcement learning. Results of others only present learned agents that perform impressive actions. However, these results do not show everything that went into the training process of those agents or models, which usually involves a lot of struggle. People who are unfamiliar with deep reinforcement learning can be lured in without possessing the necessary knowledge upfront. During their own discovery of deep RL and everything around it, they tend to get discouraged numerous times because results are not as expected. Is this the fault of the person trying to solve a certain problem? Not really, it's more of a general problem surrounding deep RL. The results that people see of trained deep RL agents are the success stories. It is fairly easy to write in a positive manner about these results. You will likely not see many results of negative experiments that did not result in desirable outcomes. A large part of the excitement around deep RL is related to the promise of applying these methods to very large and complex, high-dimensional environments. According to Irpan [47], this especially needs addressing.

The first problem of deep RL is that it is incredible sample inefficient. Reaching human like performance on certain Atari games requires tens of millions of frames.

Finishing the training process will usually take a lot longer than you initially think or plan. This is not only the case for Atari games, but also for MuJoCo. This is a physics simulator in which certain tasks can be completed such as learning to run or walk. Mostly, millions of steps are necessary to reach wanted behavior, which is a very large amount for mostly simple environments. Ignoring sample efficiency is not always possible. In some scenarios, creating data is not that simple. Not all problems can just take frames from a video game as direct input. Good hardware can significantly speed up training, which may be necessary when that much training steps are needed, but that is also not something everybody has access to.

Secondly, there are often other methods that achieve the same results and face less difficulties. Theoretically, reinforcement learning is capable of working for anything. A disadvantage is that it can be very hard to exploit certain information that could easily be hardcoded when taking certain existing knowledge into account. Reinforcement learning is very dependent on the reward function and its design. It is difficult to create a reward function that rightly rewards sought after behavior while still being learnable. This issue is further addressed in Section 4.2.3 where the design of the reward function for this research is discussed.

Another problem that often occurs is getting stuck in local optima, even when the design of your reward function is not flawed. Mostly, this issue is related to the exploration vs exploitation trade-off. Exploration methods don't always work consistently across all the environments, there is not a single guaranteed method that works. If it does work however, and desirable results are found for your environment, they may not transfer well to others. Because reinforcement learning is just optimizing your problem for the given environment, it is free to overfit as much as it wants. Generalizing the found solution to other environments is likely to give bad results. [47]

2.5 Multi-agent RL

Just as with the single-agent case, multi-agent reinforcement learning addresses sequential decision-making problems, only now multiple learning agents are involved. A multi-agent system is a group of autonomous, interacting entities that share a common environment. The state of this environment and the reward obtained by each agent are influenced by the collective actions of all the agents. The agents will interact with the environment and other agents to make decisions. Because each agent is autonomous, they all have their long term reward that needs to be optimized, which also depends on the policies of all the other agents. [48]

2.5.1 Stochastic Markov Games

The generalization of a Markov Decision Process to the multi-agent scenario is the *Stochastic Markov Game*. Similarly to an MDP, a Stochastic Markov Game is defined by a tuple $(S, \{A^i\}_{i\in N}, P, \{R^i\}_{i\in N})$ where N is the number of agents, S is the state space observed by all agents, A^i is the action space of a single agent i, yielding the joint action set $A = \{A^1 \times \cdots \times A^N\}$, P is the transition probability function $P : S \times A \times S \to [0, 1]$ from any state $s \in S$ to any state $s' \in S$ for a joint action $a \in A$, and $R^i : S \times A \times S \to \mathbb{R}$ as the reward function for agent i as a result for executing action a and transitioning from state s to s', which also yields the joint reward set $R = \{R^1, \ldots, R^N\}$. Figure 2.6 visualizes this process.

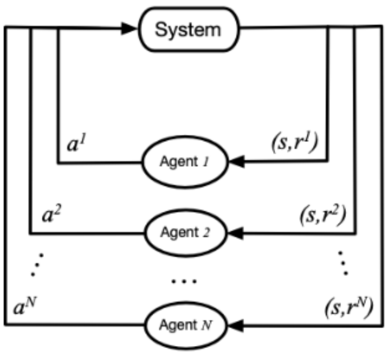


Figure 2.6: Schematic diagram for a Stochastic Markov Game with N agents where a^i indicates the action taken by agent i and (s, r^i) the observed state and reward after executing action a^i, with $i = 1, \ldots, N$ [48]

Each agent's objective is to optimize its long-term reward by finding its policy $\pi^i : S \times A^i \to [0, 1]$. Every state transition is the result of a joint action A off all the agents that is performed. The policies of each of the agents can be merged together into one single joint policy $\pi : S \times A \to [0, 1]$. The reward R that an agents receives is dependent on the joint action. As a consequence, the value function regarding agent i is dependent on the joint policy π. Applying this principle to the value function given in Equation 2.3 results in:

$$V^{i,\pi}(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r^i_{t+1} | s_0 = s, \pi] \tag{2.33}$$

The optimal policy of each agent is now also controlled by the choices of other agents. The corresponding Q-function for each agent depends on the joint action and the joint policy:

$$Q^{i,\pi}(s, a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r^i_{t+1} | s_0 = s, a_0 = a, \pi] \tag{2.34}$$

To reach the solution of a Stochastic Markov Game, the Nash equilibrium has to be reached. This is a point where the joint policy $\pi^* = (\pi^{1,*}, \ldots, \pi^{N,*})$ satisfies:

$$V^{i,\pi^{i,*}}(s) \geq V^{i,\pi^i}(s) \tag{2.35}$$

for any π^i, any $s \in S$ and $i \in N$. Each of the individual policies has thus achieved an optimal state, which also makes the merged policy optimal. [48][49]

2.5.2 Multi-agent taxonomy

A typical way to organize Multi-Agent Reinforcement Learning (MARL) algorithms is by the type of task they address and how the agents interact with each other, which can be one of three ways: cooperative, competitive or mixed. For each of these settings, specific algorithms exist to solve the corresponding SMG and reach the Nash equilibrium, or an approximation thereof. For more information on available algorithms for the different types of tasks, see [49].

Cooperative tasks

In a cooperative setting, all agents share the same reward function $R^1 = R^2 = \cdots = R^N = R$. The learning goal for these agents is to maximize the common discounted return. If all agents are being regulated by a central controller, this task turns into a Markov Decision Process, where the action space consists of the joint action space of the Stochastic Markov Game. A a result, single-agent RL methods like Q-learning can then be applied to this problem. When a central controller is absent, all agents act as individual decision makers. Subsequently, coordination is required between all the different agents or problems will arise. [48][49]

Competitive tasks

Fully competitive tasks are typically modeled as *zero-sum* Markov games. As the name implies, the sum of all the individual reward functions always equals 0, $\sum_{i \in N} R^i(s, a, s') = 0$ for any (s, a, s'). The loss of reward is the gain of reward for another agent, they essentially all fight for the same prize. *Minimax-Q* employs a minimax principle and temporal-difference learning similar to Q-learning. [48]

Mixed tasks

Mixed tasks are also referred to as *general-sum* Markov games. They have no restrictions on goals or relationships among agents. Agents in mixed tasks behave mostly self-interested, and rewards may conflict with those of other agents. [48] Crowd simulation would fall under this category, because each agent simply wants to arrive at their destination, but they won't actively try to act in a way that would make other agents perform in an inferior manner.

2.5.3 Benefits & challenges

MARL and deep MARL share most of the challenges or problems that RL and deep RL also experience, and that were previously discussed in Section 2.3.4 and Section

2.4.4. However, the curse of dimensionality is even worse in the multi-agent case, because the complexity increases exponentially with the number of agents.

Speed-up

When multiple agents are being trained, a significant speed-up can be achieved because the decentralized structure of the task can be exploited. Multiple agents may be trained in parallel, and experiences can be shared between these agents. Multi-agent reinforcement learning also allows different agents to take on different roles because multiple policies are being learned. To top it off, MARL also introduces robustness. On the off chance that an agent fails, remaining agents can take over some of their tasks. [50]

Non-unique learning goals

Defining the goal for single-agent RL is relatively straight forward. The agent simply tries to maximize the long-term discounted return or reward. For multi-agent systems, it can be rather challenging. Rewards for all the agents are correlated and depend on actions of other agents, and thus can not be maximized independently as with single-agent RL. [48][49]

Non-stationarity

Agents learn by observing the environment and basing their actions on these observations. When multiple agents are present, the agent will not only observe results of their own actions, but also those of all the other agents. Each agent is constantly reshaping the environment, which results in non-stationarity. Again, this implies that changes in the policy of one agent may cause potentially harming changes in other agents' policy. The convergence theory on which single agent RL methods are based is no longer guaranteed because the Markov property, explained in the transition function part of Section 2.1, is not valid anymore. The Markov property stated that the individual reward of an agent is only dependent on the present state, and the action taken by the agent. In a multi-agent setting, the reward is also dependent on actions of other agents. Convergence is thus not guaranteed anymore.

Different methods exist to combat non-stationarity. The most obvious approach is to just ignore the non-stationarity problem and assume that the environment is stationary. Each agent independently executes the learning process as if they are acting as a singular agent, see Section 2.6.2. Empirically however, these techniques may achieve sufficient results in practice. When the learning process is performed with an actor-critic architecture, a solution to non-stationarity is to use a centralized critic. The training of the critic is centralized and can access the observations and actions of all the other learning agents, while the training of the actor remains decentralized, as with MADDPG in Section 2.6.3. [48][51][52]

Partial observability

In most learning problems, it is not realistic that every agent possesses full knowledge over the entire environment. Usually, only a small part of the environment is observed and they need to learn based on that information. For crowd simulation this is no different. A steering agent only observes what is in their field of view, they have no idea what happens behind them. They also do not possess specific information about the state of other steering agents. For example, they will have to infer speed of others from observations.

Generally, such a problem can be modeled by a decentralized partially observable Markov decision process or Dec-POMDP. Such a model has almost all of the same elements as a Stochastic Markov Game, only now each agent can strictly access its own observations. As a result, the agent is uncertain about the state of the entire environment. To combat this, each agent will maintain a probability distribution over the set of possible states and a belief factor. The belief factor influences the importance of certain states. When the agent interacts with the environment and receives certain observations, the belief state can be updated accordingly and thus possibly improve the agent's image of the true state. CLDE schemes can prevent this issue from happening because learning is centralized and observations of all the agents are available. [48]

2.6 Multi-agent deep RL

Just as with single-agent deep RL, neural networks are applied for state representation and function approximation. When multiple RL agents are involved in the learning process, it is important to define how the agents interact and what the learning process will look like. Generally, there are three different training schemes that can be applied to cooperative or mixed multi-agent problems: centralized, concurrent and centralized learning with decentralized execution. [53]

2.6.1 Centralized

For centralized learning approaches, there exists a single joint model that includes the actions and observations for all the agents. These joint actions and observations are then mapped in a centralized policy. Because there only exists one policy for all the agents, this approach is centralized in both learning and execution. When more agents are added, the joint observation and action spaces experience exponential growth, and as a result this strategy is often not usable for more complex tasks. The curse of dimensionality is thus highly applicable to this method. Essentially, this is a single-agent approach that solves the multi-agent problem, and thus single-agent RL methods can be used to solve these problems.

2.6.2 Concurrent

Concurrent learning is the first method that actually attempts to learn multiple policies. Each agents' observations are mapped to actions to create separate, independent policies. This implies that each policy will be optimized according to the joint reward signal. This can be advantageous for heterogeneous agents because it allows agents to take on different or specific roles. There are some downsides to this approach however. Because the policies are independent, there is no way that experience or knowledge can be shared between the different agents. As a result, the sample complexity will also increase. Also, if each agent requires a policy, scaling to a lot of agents can become problematic.

2.6.3 Centralized learning, decentralized execution

To speed up the learning process for all the different agents, it would be helpful to have some insights in the experiences of the other learning agents. This way, they access information that normally would not be available to them, but can significantly speed up the learning process. This is the premise of Centralized Learning, Decentralized Execution (CLDE). CLDE also aids in combatting non-stationarity, as mentioned in Section 2.5.3. While the learning is centralized, the execution phase will still be decentralized. During execution, each agent has to independently decide what to do based on experiences gathered in the learning phase. A relevant, real life example of this would be a student that has access to books, internet, literature, etc. when studying, but only relies on their own brain during exams.

Parameter sharing

Parameter sharing allows more efficient training for agents that are homogeneous. Agents that are similar in nature often face the same goal, hence they would most definitely benefit from sharing experiences. Parameter sharing takes the CLDE concept even further, and straight up shares the entire policy between all the agents. As a result, the policy is trained with the experiences of all the different agents. This does not imply that all the agents will act the same, they still have their own observations and can act according to those. A major advantage is scalability, agents can easily be added while no other policies have to be added. Parameter sharing also allows curriculum learning to be executed very easily, as explained in Section 2.7. Because parameter sharing only learns a single policy, it also does not suffer from non-stationarity.

MADDPG

Multi-Agent DDPG is an extension of DDPG that aims to adopt a CLDE approach. During training, the critic (Q-function) is supplied with extra information about the policies of other agents, while the actor (policy) only has access to local information.

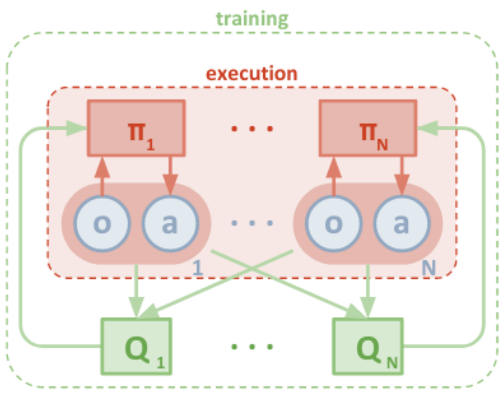


Figure 2.7: Overview of CLDE approach for MADDPG [54]

At execution time, however, only local information can be used. Suppose a game with N agents with the set of policies $\pi = \{\pi_1, \pi_2, \ldots, \pi_N\}$ parameterized by $\theta = \{\theta_1, \theta_2, \ldots, \theta_N\}$. The gradient discussed in Equation 2.27 can be adjusted as follows:

$$\nabla_{\theta_i} J(\mu_i) = \mathbb{E}_{x,a\sim D}[\nabla_{a_i} Q_i^{\mu}(x, a_1, a_2, \ldots, a_N)|_{a_i=\mu_i(s_i)} \nabla_{\theta_i}\mu_i(a_i|s_i)] \tag{2.36}$$

Q_i^{μ} is a centralized critic action-value function that has the actions of all the agents $a_1, \ldots, a_N$ as input together with state observations x, and outputs the Q-value for agent i. Normally, x just consist of the observations of all the agents $x = (s_1, s_2, \ldots, s_n)$. Although sometimes, additional state information can be added if feasible. Each Q_i^{μ} is still learned separately, so agents can still have their own reward structures, which allows all the settings from Section 2.5.2 to work. The experience replay buffer D will contain tuples of the form $(x, x', a_1, a_2, \ldots, a_N, r_1, r_2, \ldots, r_n)$. To update the centralized function Q_i^{μ}, the following update rule is applied:

$$L(\theta_i) = \mathbb{E}_{x,a,r,x'}[(Q_i^{\mu}(x, a_1, a_2, \ldots, a_N) - (r_i + \gamma Q_i^{\mu'}(x', a_1', a_2', \ldots, a_N')|_{a_j'=\mu_j'(s_j)})^2] \tag{2.37}$$

with $\mu' = \{\mu_{\theta_1'}, \mu_{\theta_2'}, \ldots, \mu_{\theta_N'}\}$ being the set of target policies with their delayed parameters θ_i'. MADDPG combats the non-stationarity issue because at each point, it is known what the actions of the other agents in the evironment are. The environment is thus stationary, even if policies are changed. Figure 2.7 display the CLDE structure of the algorithm, which is displayed in full in Algorithm 10. [54] The main difference with a parameter sharing approach is that each agent still learns its own, separate policy.

2.6.4 Alternatives

Next to the training schemes and settings discussed here, there exist a multitude of different ways to approach (deep) multi-agent reinforcement learning. Many methods focus on learning explicit communication between the agents. Communication

Algorithm 10: MADDPG for N agents [54]

```
1  for episode = 1 to M do
2      initialize random process $\mathcal{N}$ for action exploration
3      receive initial state $x$
4      for t=1 to T do
5          for each agent $i$, select action $a_i = \mu_{\theta_i}(s_i) + \mathcal{N}_t$ w.r.t. the current
             policy and exploration
6          execute actions $a = (a_1, \ldots, a_N)$ and observe reward $r$ and new state
             $x'$
7          store transition $(x, a, r, x')$ in replay buffer $D$
8          $x \leftarrow x'$
9          for agent i=1 to N do
10             sample random minibatch of $S$ samples $(x^j, a^j, r^j, x'^j)$ from $D$
11             set $y^j = r_i^j + \gamma Q_i^{\mu'}(x'^j, a'_1, \ldots, a'_N)|_{a'_k=\mu'_k(s_k^j)}$
12             update critic by minimizing loss:
13                 $L(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\mu}(x^j, a_1, \ldots, a_N)\right)^2$
14             update actor using sampled policy gradient:
15                 $\nabla_{\theta_i} J \approx \frac{1}{S}\sum_i \nabla_{\theta_i}\mu_i(s_i^j)\nabla_{a_i} Q_i^{\mu}(x^j, a_1^j, \ldots, a_i, \ldots, a_N^j)|_{a_i=\mu_i(s_i^j)})$
16         update target network parameters for each agent $i$:
17             $\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$
```

can improve coordination and thus also learning results. CLDE can be viewed as a certain method of communication, because observations of all the agents are shared amongst them, to speed up the learning process. Other communication methods focus more on setting up specific communication channels that allow for negotiation [55] or even linguistic communication [56]. Transfer learning is the process where specific knowledge received from actions performed in the past, can be used to achieve significant speed ups during learning. This concept is widely used across many machine learning applications. For the RL field, transfer learning was mostly applied for single-agent algorithms, but it starts to seep into the multi-agent field. [57] What it boils down to for most of the discussed training schemes or methods is knowledge and information sharing. This is an essential part for multi-agent RL that can significantly improve results and lower learning time.

2.7 Curriculum learning

Humans and animals tend to take extended amounts of time to learn certain tasks in life that are necessary to survive or go through life easier. Humans even more so, as almost a year goes by before we take our first steps. However, for all these tasks that we learn, it is essential that they are presented or learned in an order that becomes increasingly difficult. It starts out easy, and when we are capable to execute the current stage confidently, we can move on to the next one. Our entire education system is also build on this principle. We start by learning to read, learning easy math problems, etc. and eventually we graduate from school and are ready to start working. [58]

This concept also holds true in the context of machine learning, and is referred to as *curriculum learning*. When a machine learning (especially reinforcement learning) model is immediately tasked with the hardest problem, difficulties can arise and possible solutions may not be found. That's why it can be essential to gradually increase the level of difficulty, and make sure the model can solve the intermediate problem before moving on to the next one. Parameter sharing allows for easy curriculum learning because it scales very well to adding new agents. Other methods like concurrent learning or MADDPG require the introduction of new policies. An issue that arises when introducing new policies, is that these policies have none of the knowledge previously acquired by other policies. As a result, they start with a deficit and the learning will also suffer from that. The most simple solution would be to clone agents and their policies. This has some limitations however. How does one decide which agents to choose that are to be cloned? The policy of one agent in the current stage may not be suitable for other agents in the next stage. Long et al. [59] introduced an evolutionary selection process to aid the agents in adapting to higher scaling. Instead of just training a set amount of agents and then increasing that amount when the next stage of the curriculum should be started, they train K parallel sets of agents for each stage, and subsequently perform crossover, mutation and selection among them to move on to the next stage. They call this technique *Evolutionary Population Curriculum.*

Section 5.2 addresses some factors that emphasise the importance of curriculum learning and how it is relevant to this research.

Chapter 3

Artificial neural networks

The goal of this research is to apply deep reinforcement learning, and thus Artificial Neural Networks (ANN) will be applied for training. ANNs are mathematical inventions that were inspired by the operations of biological systems, but are not heavily based on actual biology. In a way, ANNs are abstractions of biological neural networks that are build up out of neurons and synapses. The high inter-connectivity between these neurons served as inspiration to form ANNs. Thus, the driving principle behind ANNs was not to clone or replicate operations of biological systems, but rather to be inspired by them. A similar example of such a comparison are feathers and airplanes. Biological neural networks are to ANNs what feathers are to airplanes. There is little resemblance between the topics, but the basic principles are similar and they present more insights, and as a result provide better understanding. [60]

At its core, an artificial neural network can be interpreted as a function mapper, that returns an output based on the input received. Between the input and output layers lies a highly interconnected system of artificial neurons or nodes, that is capable of highly parallel computing. This is also why ANNs are so interesting, they posses important processing characteristics such as non-linearity, high parallelism, robustness, fault tolerance, ability to generalize, etc. Being a function mapper, the ANN has to be trained to map the right output to the input its being fed. This will be done by training the ANN, which can be accomplished in different ways. [61] This chapter will explore the basics of ANNs that are necessary for this research, thus not too many details will be touched upon.

3.1 Fundamentals

3.1.1 Basic components

Essentially, an ANN will consist of a set of connected processing units called *neurons* that communicate with each other. They do so over connections, with each connection being defined by a *weight* w_{jk} that regulates the influence that unit j has on unit k. Each unit also possesses an *activation* y_k, which corresponds to the output of the unit. The *propagation rule* determines the total effective input s_k that a unit experiences from external inputs, so all the inputs coming from other units. An example of a propagation rule is the weighted summation over all the input units. Subsequently, the *activation function* $\mathcal{F}_k$ determines the new value of the activation y_k based on the total effective input $s_k(t)$ and the current activation $y_k(t)$. A *bias* θ_k can also be introduced for every unit. Figure 3.1 shows a neuron with the aforementioned components and the weighted summation as propagation rule. Generally, there are three types of neurons or units: *input*, *output* and *hidden*. Input units get their data from outside the neural network, hidden units receive input from input units and forward it to other hidden units or output units, and output units send their data out the network. [62]

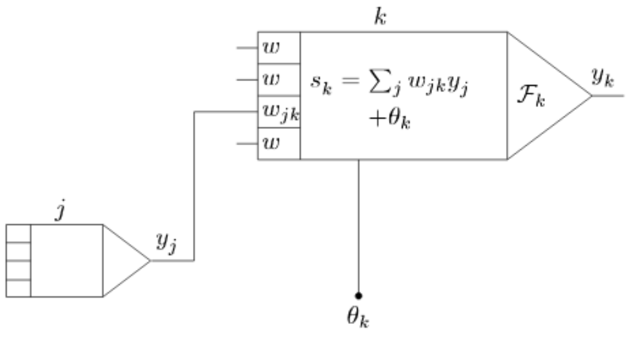


Figure 3.1: Basic components for singular unit in ANN [62]

3.1.2 Activation functions

Activation functions are used to impose bounds on the unbounded output of the units. This helps the ANN to interpret the data and make sense of it. The result of the activation function is fed into the units of the next layer (hidden or output). It also allows neurons to be activated or not. Without these activation functions, signals outputted by neurons would be linear and thus, non-linearity would not be possible, which would limit the complexity of problems that can be solved by ANNs. The goal of the activation is to take the total input $s_k(t)$ and the current activation $y_k(t)$ and calculate the new activation value for unit k in the following way:

$$y_k(t+1) = \mathcal{F}_k(s_k(t)) = \mathcal{F}_k\Big(\sum_j w_{jk}(t)y(t) + \theta_k(t)\Big) \tag{3.1}$$

Mostly, this activation function F is a non-decreasing function that applies some sort of threshold. [62] Below, some different activation functions are explained, Figure 3.2 shows correspondings graphs. [63]

Signum

The simplest example is the Signum function. This function outputs 1 if the input value is bigger than 0 and -1 if the input value is smaller than zero. The limitation for such a function is that no multi-value outputs are possible. It will always be one of two values. Classifying into more categories is thus not possible.

Sigmoid

The sigmoid function provides a smooth gradient. As a result, the values can gradually advance and won't suddenly jump to different output values, as with the Signum function. The output values range between 0 and 1, and for x values outside the [-2,2] range, y values are close to 0 and 1 respectively, which allows for clear predictions. However, when values for x become very low or very high, the y value will barely change. In turn, this could complicate the learning process.

TanH

The TanH or hyperbolic tangent function is similar to the Sigmoid function, only now y values are centered around 0, and thus range between -1 and 1. The same advantages and disadvantages count as with the Sigmoid function.

ReLU

The Rectified Linear Unit has the advantage of being computationally very efficient, which in turn allows for quicker convergence. It may look linear but it still allows for non-linear learning. However, it can suffer from the dying ReLU problem. This problem occurs when inputs get close to zero or negative values. At this point, the gradient of the ReLU function is 0 and thus learning can't continue.

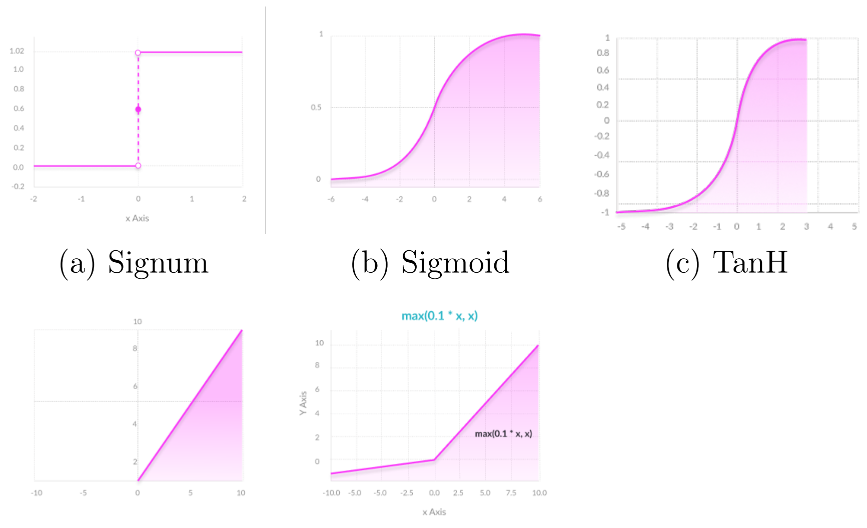
Leaky ReLU

Leaky ReLU prevents the y value from becoming 0 and thus enables backpropagation and learning for negative values. For these negative values, it remains a challenge however to have consistent predictions.

3.1.3 Network topology

Now that the units or neuron were discussed, entire networks can be formed. There always exist an input and output layer, and between these two layers are hidden layers. When the amount of hidden layers is 2 or more, the network is a deep neural network. A distinction can still be made regarding topologies of networks:

- **Feed-forward**: The data flow in this network is one directional. It goes directly from input layer to the hidden layers, and subsequently to the output



(a) Signum (b) Sigmoid (c) TanH

(d) ReLU (e) Leaky ReLU

Figure 3.2: Graphs depicting multiple activation functions [63]

layer. Such a structure is displayed on the left side of Figure 3.3. The most popular feed-forward neural networks are the *Multi-layer perceptron* (MLP) and the *convolutional neural network* (CNN). The MLP uses at least 3 layers and every node is fully connected to the nodes in the successive layer. These are the types of networks applied in this research. CNNs are mostly used in image processing due to the application of the convolution operation. Applying this operation results in extraction of local features which in turn significantly lowers the number of dimensions of the input.

- **Recurrent**: These types of networks have feedback structures between layers, and thus the data flow is not limited to one direction. This allows the network to display temporal dynamic behavior, which opens up new possibilities. Such a structure is shown on the right side of Figure 3.3.

Other variants exist within these topologies but are not discussed. [62]

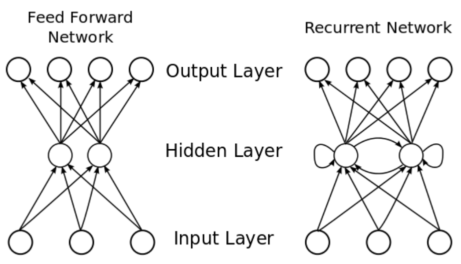


Figure 3.3: Feed-forward vs. recurrent neural network structure [64]

3.2 Training ANNs

The goal of training ANNs is to alter the weights of connections in such a way that desirable outputs are given for the set of inputs.

3.2.1 Paradigms

Different paradigms exist to achieve this goal. Each of these methods should be employable for any type of ANN. Multiple different training algorithms are available for each paradigm as well.

- **Supervised learning**: This method sets the ANN parameters based on training data it receives. Input-output pairs are fed into the ANN and it then must set the weights of connections to appropriate values that lead to the necessary output. Applying this process, the first step will be to decide on the type of training examples that will be used. Secondly, the data has to be gathered that will satisfy those conditions. The third step entails ensuring that this data is transformed in such a way that it is possible for the ANN to learn from. Finally, the learning process is executed, and the ANN's weights are set to certain values. At this point, the performance can be analyzed by using a validation test set. This needs to be data that was not fed into the ANN while learning.

- **Unsupervised learning**: For this paradigm, the ANN needs to find patterns itself in the provided inputs, no corresponding outputs are provided. The weights of the connections are set according to the input data and an accompanying cost function that needs to be minimized. The goal of unsupervised learning is finding how the given data is organized and is generally used for estimation problems regarding statistical modelling and clustering. In these cases, data will be classified based on their affinity.

- **Reinforcement learning**: For this method, the weights of the ANN are set by generating interactions with the environment, and thus not by previously generated data. The goal of reinforcement learning is to maximize a long-term reward, as mentioned in Section 2.2. ANNs will be used as parts of the reinforcement learning algorithm to approximate the value of policy functions. Samples from the state and action space with corresponding rewards are used as input to do so. At the beginning, the ANN's weights are initialized stochastically. Then, the ANN can update its weights by comparing expected reward with returned reward from the samples. For more information on deep reinforcement learning, see Section 2.4.

[65]

3.2.2 Loss function and gradient descent

The goal is to determine the weights of each node in such a way that inputs are mapped to correct outputs. To achieve such a feat, it is important to know how well the network performs. For this purpose, a loss or cost function is used. For the supervised learning paradigm, the loss function indicates how much the predictions of the ANN differ from the target values. Some common error functions include the mean squared error, sum of squared errors, L1 and L2 loss, etc. The goal would be to minimize the selected cost function w.r.t. the weights of the ANN. This is generally done by applying gradient descent. At each timestep, the gradient of the loss function is calculated by applying the backpropagation method. This process will not be discussed in detail, the gist of it is that backpropagation is applied after each forward pass in the ANN. The gradients are calculated backwards through the network, starting with the final layer of weights and working towards the first one. Computations of the gradient of one layer are used for the gradient computation in the previous layer. As a result, the gradient consists of partial derivatives of the loss function w.r.t. every variable:

$$\nabla J(\theta) = \Big(\frac{\delta J}{\delta \theta_1}, \dots, \frac{\delta J}{\delta \theta_N}\Big) \tag{3.2}$$

with θ being the weights of the ANN. The gradient will point in the direction where the loss function goes down the most. A slight step then needs to be taken into this direction. This will be repeated until the minimum is reached. The size of the step is determined by the learning rate η. The new weights are then calculated as follows:

$$\theta = \theta - \eta \nabla J(\theta) \tag{3.3}$$

Essentially, the learning rate determines how fast the weights move towards the optimal solution. If the learning rate is set too large, the optimal solution could be skipped. If it is set too low, the learning will take too long. Setting a good learning rate is thus crucial. [66]

Gradient descent alternatives

Equations 3.2 and 3.3 display the basic cases for gradient descent and how the update with the learning rate is handled. In practice, different methods exist that will work better and are optimized for certain use cases. The problem with vanilla gradient descent is that the gradients for the entire dataset need to be calculated to execute just a single update. As a result, it can thus be very slow and it also does not allow for online updates. Stochastic Gradient Descent (SGD) alters vanilla gradient descent by introducing an update for each training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{3.4}$$

This way, the redundancy of vanilla or batch gradient descent is gone. However, SGD introduces a lot of noise because frequent updates with high variance are performed. This introduced noise allows SGD to reach local minima that could never be reached with vanilla gradient descent. A disadvantage, however, is that this noise can complicate reaching convergence to the exact minimum.

Minibatch gradient descent combines vanilla and stochastic gradient descent into one. It will perform an update for every minibatch of n training examples:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{3.5}$$

The advantages are that variance of parameter updates is brought down, which can cause more stable convergence behavior. It can now also use optimized matrix optimizations that will be found all throughout state-of-the-art deep learning libraries. This makes the gradient computation very efficient and makes minibatch SGD usually the go to algorithm. [66]

Optimizers

Choosing a good learning rate is often very challenging, as we already mentioned above. Adapting the learning rate while the learning process is taking place, can often make learning easier and may significantly speed up the process. SGD usually has difficulties navigating surfaces that have a much steeper curve in one dimension than in others, which is prevalent around local optima. Momentum attempts to solve this problem by accelerating the SGD in the right direction, by adding a fraction of the update vector from the previous timestep to the current one:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{3.6}$$

The momentum value γ is mostly set to values around 0.9. Using this method, the same thing happens with parameter updates that would happen when rolling a ball down a hill. The ball picks up speed and becomes faster and faster. The momentum will increase for gradients that point in the same direction and decrease for gradients that change directions. This results in better and faster convergence.

Being able to adapt each individual parameter to allow them to perform larger of smaller updates, would also be helpful. Previous methods would update all the parameters at once, using the same learning rate. Adagrad allows a per-parameter update with a specific learning rate at each timestep. Frequent parameters will get small updates, while infrequent parameters execute larger updates. Adagrad keeps track of all past gradients and uses them to adapt the learning rate accordingly for each parameter. Using this method removes the need to tune the learning rate, as it is done automatically. Adadelta en RMSprop are extensions of Adagrad that attempt to solve the fast decrease of the adaptive learning rate, which is a problem Adagrad suffers from. Adadelta tackles this problem by adapting the amount of past gradients to a fixed size w. Instead of keeping the entire amount w, the recursive definition of the sum of gradients as a decaying average of all past squared gradients is stored. At each timestep, only the previous sum and the current gradient need to be used to perform an update. RMSprop applies a similar solution.

Adaptive Moment Estimation (Adam) also provides adaptive learning rates for all the parameters. Just as with Adadelta en RMSprop, it will compute an exponentially decaying averages of past squared gradients v_t and store those. In addition, Adam also stores an exponentially decaying average of past gradients w_t, similar to momentum:

$$\begin{aligned} v_t &= \beta_2 m_{t-1} + (1-\beta_2) g_t^2 \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \end{aligned} \tag{3.7}$$

These two values are considered estimates of the mean (first moment) and the uncentered variance (second moment) of the gradients. β_1 and β_2 are decay rates. The update for the gradients for each parameter is then executed as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{3.8}$$

with $\hat{v}_t$ and $\hat{m}_t$ being bias-corrected estimates of v_t and m_t. [66]

Linking back to deep RL

Everything explained about training ANNs up unto this point was focussed on supervised learning, the most general case. For reinforcement learning, learning and specifically backpropagation has to be executed differently. For backpropagation to work, the loss needs to be differentiable w.r.t. the weights of the ANN. The problem with reinforcement learning is that generally, the environment for the learning agent is non-differentiable w.r.t. the action the agents takes. As a result, normal backpropagation will often not work for reinforcement learning. There are two classes of algorithms that we discussed in Section 2.3.3 and further in Section 2.4 that solve this problem.

Firstly, to combat this problem for Q-learning methods, they attempt to learn the expected return for each state-action pair, instead of maximizing reward in a direct manner. This is possible with the standard backpropagation method. DQN and the Q-learning side of DDPG operate in this manner, see Section 2.4 for details. Secondly, policy gradient approaches learn the policy directly by using gradient estimation methods. The gradient of the reward w.r.t. network parameters θ is then estimated, and can subsequently be used to perform updates. The policy learning side of DDPG does this by assuming that the Q-function is differentiable w.r.t. the action, based on the gradient estimator theorem shown in Equation 2.21. PPO uses a similar estimator, displayed in Equation 2.29, which is again based on the gradient estimator theorem. DQN, DDPG and PPO all use minibatch SGD with Adam to perform the gradient descent.

3.3 Hyperparameter tuning

The success of neural networks as non-linear function approximators depends heavily on the structure of the model, the data fed into the model and how the model is optimized. Choices made before learning even starts will heavily influence outcomes. These choices can be labeled as hyperparameters. Traditional hyperparameter tuning methods can be subdivided into two categories: *parallel search* and *sequential optimization*. These two methods trade-off used resources with execution time needed to find the best hyperparameters. [67] For all hyperparameter tuning methods, the goal is to find the hyperparameters that perform the best on a validation set. This process can be defined as a simple equation:

$$x^* = \arg\min_{x \in \mathcal{X}} f(x) \tag{3.9}$$

with $f(x)$ being the objective score to minimize when evaluated on the validation set. The evaluation metric can be the root mean square error or another error rate. x^* represents the optimal set of hyperparameters, with x being any value from the domain $\mathcal{X}$. In other words, the goal is to find the set of hyperparameters that obtain the best score on the validation set, and this is done by minimizing the error. [68]

3.3.1 Parallel search

Parallel search methods will execute many parallel processes with a different set of hyperparameters. The set with the best training results can then be used as hyperparameters. This process is displayed in Figure 3.4 where three different models are tested with different initializations. The middle model is best optimized, and thus these hyperparameters will be used.

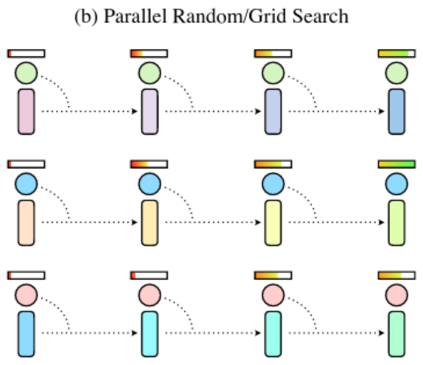


Figure 3.4: Parallel search method where three different models are optimized with random or grid search [67]

Grid search

Grid search is the most basic hyperparameter optimization method that exists. Every possible combination available for all the values of the parameters is tested, and the combination that gives the best end results is chosen. The values of the parameters have to be discrete however. Just as with reinforcement learning, grid search suffers from the curse of dimensionality. The number of evaluations increases exponentially with the dimensionality of the parameters. [69]

Random search

Random search is an alternative to grid search where configurations of parameters are chosen at random. It is especially convenient when certain parameters are more important than others. Random search is possible for both discrete and continuous settings. [69]

3.3.2 Sequential optimization

Sequential optimization methods do not run many parallel processes, but instead execute them sequentially. Hyperparameters are changed gradually and useful information from one execution can be applied to the next one. Sequential methods tend to give better results than parallel methods, but also take longer to optimize because there is little parallelization and multiple sequential executions are needed. Because everything is executed sequentially, fewer computational resources are needed. Such a process is shown in Figure 3.5.

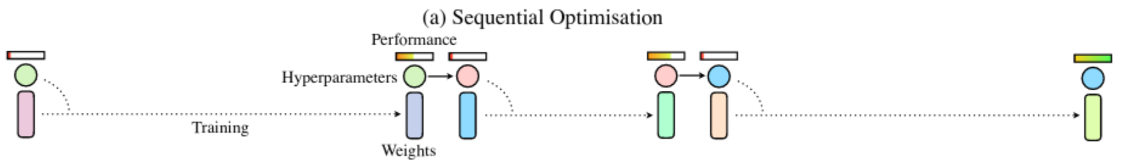


Figure 3.5: Sequential optimization process containing three sequential steps, where after each step new hyperparameters are selected based on the findings of the previous step [67]

Bayesian optimization

Unlike the parallel search methods, Bayesian optimization uses previous results to shape the next batch of hyperparameters. This is done by using a probabilistic surrogate model that maps parameters to a probability of a score on the objective function [68]:

$$P(\text{score}|\text{hyperparameters}) \tag{3.10}$$

This is referred to as the surrogate for the objective function in Equation 3.9. The advantage of this surrogate function is that it is way easier to optimize compared to the objective function. Bayesian methods operate in the following manner:

1. Construct the probabilistic surrogate model
2. Find parameters that maximize performance on the surrogate
3. Apply said parameters to objective function
4. Update surrogate model with newly found results
5. Repeat steps 2-4 until end condition

The more iterations are executed, the preciser the surrogate model becomes. Because Bayesian methods spend more time to determine the next hyperparameters, they do not have to make as much calls to the objective function, which the a very expensive operation. [68][69]

3.3.3 Population Based Training

Traditional hyperparameter tuning methods tend to be very computationally expensive, and this will increase even more with the complexity of the neural network. Deep reinforcement learning also introduces an extra level of complexity, because the training process might be highly non-stationary. Agents are dependent on specific parts of the environment they can currently explore. For these problems, hyperparameters may also be non-stationary and may need to change during the training process. Population Based Training, designed by DeepMind [67], combines both parallel and sequential methods into one technique. PBT does not require sequential runs and the run time is comparable to that of a single optimization process. It will not consume as much resources as naive parallel search methods that go through all possible options and combinations. Also, PBT allows for online adjustment of hyperparameters, which is interesting for non-stationary problems.

PBT initially takes off as a parallel search. It will randomly sample some hyperparameters and weights as initial values. Every parallel run will periodically conduct an evaluation on its performance. If the model for a specific run does not meet the requirements, and is thus under-performing, the rest of the population is exploited and the model is replaced with a better performing one. It will then explore new hyperparameters by altering the better performing model's parameters, and afterwards training is resumed. As a result, parameters can be adjusted online and resources are directed to models that perform well. All in all, this gives a hyperparameter tuning method that is relatively simple, but will result in rapid learning that uses fewer resources and may even give better solutions. Figure 3.6 display the PBT process. [67]

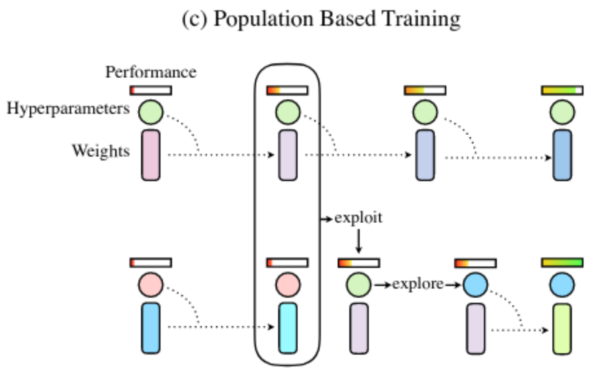


Figure 3.6: Visualization of the PBT process, where the bottom process exploits the better performing one [67]

Chapter 4

Crowd simulation with deep multi-agent RL

4.1 Related work

Before diving into the process utilized to generate crowds in this thesis, some related work is referenced. Another field that shares a lot of similarities with crowd simulation is robot navigation, which may also be sporadically consulted.

4.1.1 Data driven crowd simulation

With reinforcement learning, there is no initial data to build upon. However, for crowd simulation or pedestrian movement, aerial footage is often used as input to learn from. Hence, these techniques apply supervised learning instead of reinforcement learning and are referred to as data driven methods. Example input and output pairs are used as training data. The problem with such an approach is that input data requires strict formatting to make learning possible. A direct consequence of this is that aerial video footage needs to be analyzed frame by frame, and each pedestrian needs to be labeled with data suitable for learning. Mostly, vectors are used to indicate current speed and direction of the individual, as seen in Figure 4.1.

Once the required data is extracted, Artificial Neural Networks will be trained using this transformed real life data. Song et al. [71] used the Social Force model as reference to determine inputs needed to train the ANN. The desired moving direction of pedestrians is computed by the A* algorithm. At each iteration, the desired direction can then point the agent in the direction of the computed path. Besides desired direction, the current speed of each agent is a vector with an x and y value. Finally, to get a decent understanding of its environment, each agent has the relative positions and velocities of a certain amount of neighbouring pedestrians. Data gathered from aerial videos is then converted to this format, and the training of the ANN can be commenced. When training and validation phases are completed,

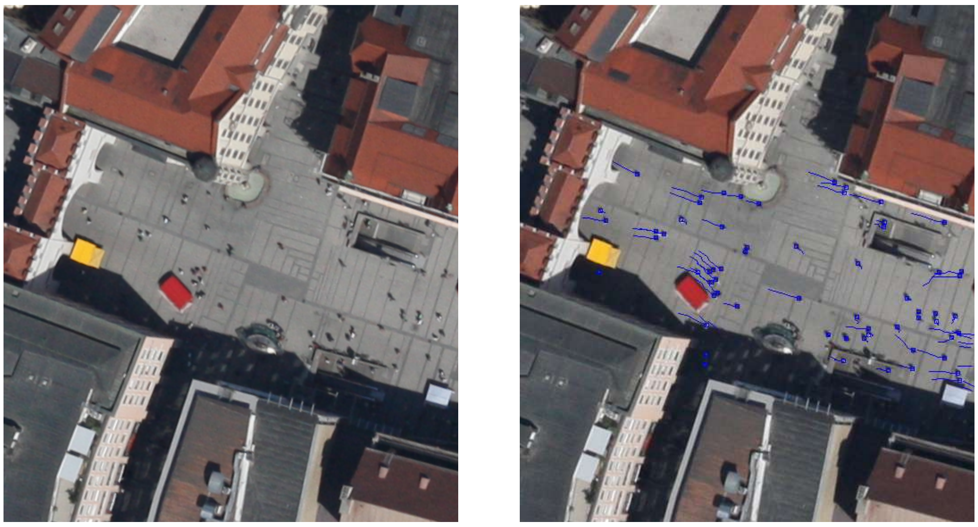


Figure 4.1: Frame of KIT dataset for tracking people in aerial image sequences, with on the left pre labeling and on the right post labeling [70]

and the input data was versatile and large enough, the ANN will now be able to perform simulations on unseen scenarios.

Zhao et al. [72] take a different approach that consists of two main parts. First, they start by clustering examples extracted from aerial data to find similar patterns of behavior. These patterns are then stored in a hierarchical database. Afterwards, an ANN classifier has to be trained using this database, so it can extract the relationship between the input state of an example and to which cluster it corresponds. Then, during the simulation, each agent will view its surroundings and use that information as input for the ANN classifier. In turn, the input gets classified into a specific cluster in the hierarchical database, and a similar set of examples is returned. Out of these returned examples, the most suitable one is selected and behavior is executed following this example.

4.1.2 Crowd simulation with reinforcement learning

The works of Casadiego et al. [73], Henry et al. [74] and de la Cruz et al. [75] all explored applying reinforcement learning in the field of crowd simulation. However, most of these approaches are on smaller scale environments and overall simpler test cases. The work of Lee et al. [76] was one of the first to explore simulating believable virtual crowds with deep reinforcement learning in continuous state and action domains. They succeeded in doing so by training a single unified policy that controls all agents and can adapt to unseen scenarios. Each agent uses visual sensory input in the form of laser scanning to obtain distances to neighbouring agents and obstacles, which are both treated as the same thing. The key to generalizing simulated behavior to unseen scenarios, was a curriculum learning scheme build up in such a way that easier examples have higher probability to occur earlier in the training process, and vice versa for the harder ones. Essentially, this was still a single-agent reinforcement learning method.

Continuing on single-agent reinforcement learning, robot navigation also falls mostly into this category. The goal of these autonomous robots is to successfully navigate unknown environments while avoiding collisions with obstacles. When running virtual examples for robot navigation of crowd simulation, it is easy to control the type of environments the robot or pedestrians will face. When these robots are employed in real-life situations, environments are truly stochastic. For these robots, it is then extremely important to know exactly what their environment looks like. In such situations, it is possible for robots to learn based on pure visual input, extracted directly from cameras. This input can then be fed to Convolutional Neural Networks to train, as is done in [77]. Other applications will use 2D LIDAR to measure distances by illuminating the environment with laser lights, and subsequently catching the reflection of these lights with a sensor, as with [78].

Generally, single-agent methods have been explored extensively, thus also in the crowd simulation field. However, crowd simulation is essentially a multi-agent RL problem. Torrey [79] explored multi-agent RL for crowd simulation in simple grid based environments that represent a school like domain. The environment consisted of multiple classrooms connected by hallways. Agents are planted at random places inside this environment. Each agent will observe the surrounding environment and choose an action accordingly. Because of the grid based system, actions and states are discrete, which will limit the credibility of generated crowd behavior. Each agent independently executes the Q-learning variant SARSA to achieve multi-agent learning, no deep learning is applied. Independently executing single-agent RL methods for the different agents violates convergence conditions, as mentioned in Section 2.5.3. For the research conducted by Torrey however, it did not prevent agents from successfully learning. Similar to this research, Yan Lim [80] explored joint state reinforcement learning algorithms for multi-agent settings, also in grid-like environments. These methods are similar to the centralized multi-agent training scheme explained in Section 2.6.1. When environments were small enough, learning could succeed. The problem with these grid-like environments is that the steering behavior of pedestrians is not believable or smooth, because no continuous state is used.

Bisagno et al. [81] used ML-agents, a build in feature of unity, to model crowded environments where multiple agents navigate certain scenarios. Here, continuous states and actions are applied so that agents can recreate believable behavior. Each agents has its own policy that they train, and for this purpose PPO is applied. There is no mention of specific coordination or communication between the agents, so this method would be comparable to the concurrent approach explained in Section 2.6.2. Curriculum learning is applied here to steadily build up, not only the number of agents, but also the speed and possible moving directions of the agents.

Looking back at the robot navigation field, the works of Everett et al. [82]

and Chen et al. [83] explored multi-robot steering behavior that includes successful collision avoidance with deep reinforcement learning methods. Similarly, the works of Fan et al. [84] and Long et al. [85] assumed that robots were homogeneous, and thus can be controlled by the same policy. As a result, the same policy is trained for all the moving robots using PPO. Opposite of these approaches was the research of Yoon et al. [86], where robots are assumed to be heterogeneous. Here, a centralized learning with decentralized execution approach is applied, with explicit communication channels between the robots. For this purpose, a modified version of MADDPG was applied.

4.1.3 Mixed approach

In addition, there also exist methods that combine elements of data driven and RL based crowd simulation. Most RL methods regarding crowd simulation or evacuation fixate on optimizing efficiency and performance of the model, but while doing so, the visual realism of the scenario suffers. Using existing data of real life crowd movement can improve the visual realism aspect of crowd simulation methods that mainly focus on performance and trajectories. Most existing methods lack the ability to adapt well to changing and dynamic environments, which in turn decreases the visual realism aspect. Yao et al. [87] proposed a RL based data-driven crowd evacuation framework. The model starts by extracting information such as velocities and positions from aerial videos, with the main goal being determining the cohesiveness of pedestrians. The extracted data is then clustered by a K-means like algorithm, resulting in groups for which the trajectories of the individuals are merged. In addition, a hierarchical path planning process is used to train a control policy by using grouped trajectories from the clustering step. This is done by applying Q-learning, which will allow individuals to learn observed features of crowd simulation but still maintain robustness regarding changing environments.

Faust et al. [88] presented Probabilistic Roadmap-Reinforcement Learning (PRM-RL), a hierarchical method to achieve long-range robot navigation in which sampling-based path planning and reinforcement learning are combined. For long range navigation especially, designing a good reward function is very hard, and often results in sparse reward signals. This can be solved by introducing hierarchical waypoints. These waypoints are predetermined by the probabilistic roadmap, and a higher-level planner will invoke the RL agent to train and search a path that leads to the next waypoint. When the waypoint is reached, the planner will change the goal of the RL agent to the next waypoint in the predetermined roadmap. This allows the system to combine the efficiency of a PRM with the robustness of the RL agent. [89] This principle is comparable to using the A* algorithm to find an optimal path in the data driven methods, and to base the ideal direction of the agent on that path.

4.2 Environment

The setup of the environment is essential to the whole reinforcement learning process, and thus should be developed with care and precision. To implement the environment with which the reinforcement learning algorithm will interact, Gym is used. Gym is a toolkit for developing reinforcement learning algorithms and provides a standardized environment interface that many existing reinforcement learning frameworks use and support [90]. The interface consists of two main methods: *step* and *reset*. Other essential factors are the *observation* and *action* space that need to be defined inside the environment. These are used to limit the possible values of chosen actions and returned observations (see Section 2.1 for a reminder on the modus operandi of an MDP). Initially, these concepts are explained from the perspective of a single steering agent, which allows for easier explanation. Later on, the expansion to multiple agents is addressed. As a side node, one reinforcement learning agent does not necessarily equal one steering agent or pedestrian in a crowd simulation. This mapping can be different, which will also be explained later on.

The *step* method takes an action as input, and proceeds to update the position of an pedestrian. This new position is evaluated according to some criteria, which results in a reward value. Eventually, the step methods return a state or observation, a reward and a Boolean that indicates if the goal is reached. This information is used by the reinforcement learning agent to assess the action that was taken.

The *reset* method restores the environment to its initial state. This usually takes place when the goal was reached and a new episode is started.

4.2.1 Action

The intention of an agent in crowd simulation is reaching a predetermined goal, while at the same time avoiding collisions. To make this possible, movement of agents is a core part that needs to be addressed first. The positional information of a moving agent in the environment consists of its position as a 2D vector, and its orientation as a single value in degrees or radians. As explained in Section 2.1, an action is chosen based on the current observation, and this action updates the positional information of the agent. In our case, this action will consist of two values: the *linear velocity* v and the *angular velocity* w. The linear velocity, expressed in $^{m}/_{s}$, determines forward or backward movement, while the angular velocity, expressed in $^{\circ}/_{s}$ or $^{\pi}/_{s}$, determines left or right turning.

Update orientation

The first step to updating the orientation, is converting the single orientation value to a 2D vector $[x, y]$, which is the same as converting the orientation from Polar to Cartesian coordinates as follows:

$$x = r * cos(\Theta), \quad y = r * sin(\Theta), \quad ori_{2D} = [x, y]$$

with Θ being the orientation in degrees or radians and $r = 1$. To perform a rotation in Eucledian space, a rotation matrix can be used. A rotation matrix can rotate the points in the Cartesian coordinate system according to an angle α, which is determined by the angular velocity and the timestep that indicates the frequency of updates. To get the angle α, simply multiply the angular velocity w with the timestep. The rotation matrix R is then calculated as follows:

$$\alpha = w * timestep, \quad R = \begin{pmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{pmatrix} \tag{4.1}$$

To determine the new orientation of the agent, a matrix multiplication of the rotation matrix and the 2D orientation vector is executed:

$$ori_{2D\,new} = ori_{2D} * R$$

The new orientation is still expressed as Cartesian coordinates, to convert back to Polar coordinates and get a single new orientation value, do as follows:

$$\Theta_{new} = tan^{-1}\left(\frac{ori_{2D}[1]}{ori_{2D}[0]}\right)$$

Update position

To update the position of the agent, the linear velocity v is multiplied with the timestep first to get the displacement for a single timestep. The next step is to multiply the displacement value with the 2D orientation vector $[x, y]$ from Section 4.2.1, and add this result to the current position:

$$displ = v * timestep, \quad pos_{new} = pos + (displ * ori_{2D})$$

The order in which position and orientation updates are executed is of no importance.

4.2.2 Observation

For the steering agent are not only its own position and orientation of importance, but also its surroundings and what is actively happening around them. Therefor, it is important to make a distinction between an *internal* and *external* state.

Internal state

The internal state refers to the position of the moving agent relative to its goal. To obtain the relative position and orientation, a transformation from global to local coordinates is required, as illustrated in Figure 4.2. Essentially, this entails making the moving agent the centre of the coordinate system. To achieve this goal, the rotation matrix R from Equation 4.1 is used again. First, the relative position with relation to the goal is calculated by subtracting the agent's position of the goal's position. Next, the inverse of the rotation matrix is multiplied with the relative position to take the orientation into account, with the local coordinates as a result.

$$pos_{rel} = pos_{goal} - pos_{agent}, \quad state_{internal} = R^{-1} * pos_{rel}$$

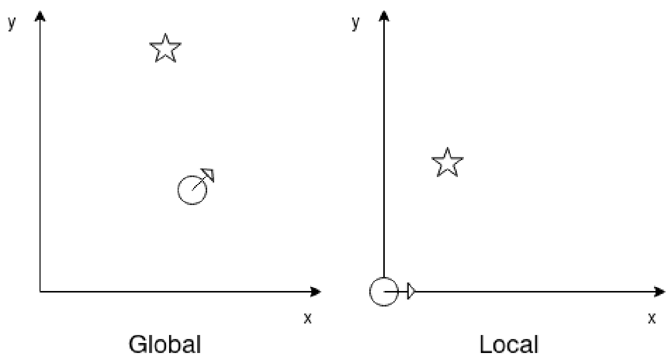


Figure 4.2: Global coordinate system vs local coordinate system of the moving agent

External state

The moving agent needs to have a notion of its surroundings, which typically include other moving agents and obstacles. The external state is formed by laser scanning, a technique regularly used within robot navigation, as shown in Figure 4.3. Lasers or rays are emitted out from the view point of the agent, covering a field of view of 180°. The laser will intersect with an entity at a certain distance.

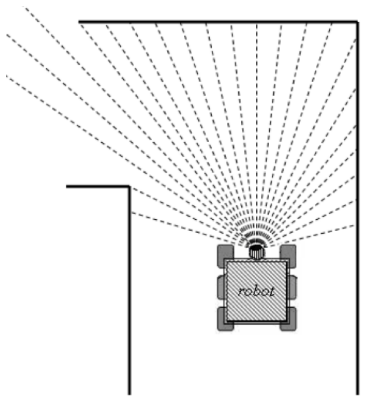


Figure 4.3: Illustration of a robot using a 2D LIDAR laser scanner to identify objects and distances [78]

In the crowd simulation field, this approach has also been used before. Lee et al. [76] used r amount of rays to save the distance for that laser if the laser intersects with a neighbouring agent. These distances are saved for t amount of timesteps, so a history of distances to neighbours emerges. Our method extends this approach by differentiating between the different types of entities the lasers can intersect with: neighbouring agents, goals and obstacle/bounds. A second data object is kept, but instead of distances, the type of entity is kept as well. The external state is thus defined by the number of lasers sent out, and the amount of timesteps that the history will consist of.

$$state_{external} = [distances, types]$$

$$\begin{aligned} distances &= [D_t, D_{t-1}, D_{t-2}, ..., D_0] \\ D_x &= [d_r, d_{r-1}, d_{r-2}, ..., d_0] \end{aligned}$$

$$\begin{aligned} types &= [T_t, T_{t-1}, T_{t-2}, ..., T_0] \\ T_x &= [t_r, t_{r-1}, t_{r-2}, ..., t_0] \end{aligned}$$

It is possible to let these lasers go until they hit an object, and thus not put a limit on the maximum distance. All our test scenarios are bounded, so eventually the lasers will hit an object, agent or goal. It is also possible to limit the maximum distance to a certain value *max_ laser_ distance*, e.g. 10 or 15. Both options, as displayed in Figure 4.4, will be explored or compared. Logically, leaving the lasers uncapped tends to be more computationally expensive. An additional problem popped up when an agent finds itself close to a goal. When the agent approaches the goal, the distance for goal intersecting lasers will shrink. In this case, the sole distinction between being close to a goal or an obstacle/agent is the type. Neural networks could struggle to properly classify these options, and by applying generalization the goal can be seen as an agent or obstacle. Agents would then hesitate to enter the goal and sometimes stall in front of it. This was solved by setting the distance value of those lasers high, e.g. the scenario range on a certain axis (distance from x_{min} to x_{max}). Doing this, the agent will view the goal as an empty space in front of them where there is no interference, and where they can move freely. Other possible solutions would include simply extending the training time so it correctly learns the types or to keep separate distance arrays for each type. Thus an array that keeps distances to obstacles, another one for agents and a last one for goals. This would however prevent agents that have been trained to dodge obstacles, to replicate that behavior when an agent is in front of them.

In summary, the observation space of the environment is determined by the internal and external state. The definition of the action space entails determining the bounds of the linear and angular velocity, which is discussed in Section 4.4.3.

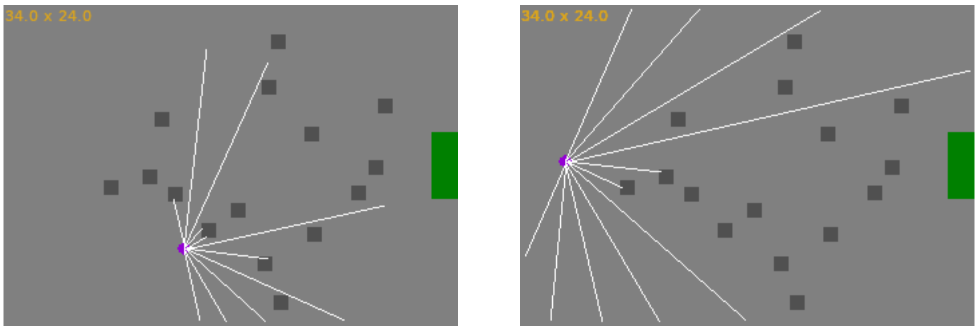


Figure 4.4: Capping laser distances vs. leaving them uncapped

4.2.3 Reward

Rewards inform the reinforcement learning agent if chosen actions are suitable or not. Without a good reward function, learning become a very difficult task. The two most prevalent factors regarding good or bad choices in crowd simulation are reaching the goal in a simple, logical manner and avoiding collisions with other entities such as moving agents and obstacles. A pitfall often observed in reinforcement learning are sparse rewards. Sparse rewards give the learning agent not enough information, which can make learning impossible or too dependent on exploration. The agent has no idea whether a performed action was advantageous or not. An extreme example of a sparse reward applied to crowd simulation is only rewarding the agent when the goal is reached. If the agent does not reach the goal, it gets no reward and is literally grabbing in the dark.

A problem that can pop up while solving sparse rewards however, is that the tuning process of the reward function is based too much on the needs of the user and the current problem they are trying to solve. This may introduce bias to the solution the agent can find. Sometimes, the reward function is deliberately kept on the sparse side. The agent is then supported during learning, which allows them to learn from a more sparse reward function, while reducing the specific prior task knowledge, and consequently also reducing the bias. [91]

The design of the reward function for this research is kept pretty general and is based on the work of Lee et al. [76].

Goal related

A good approach to avoid sparse rewards is giving the agent a reward for getting closer to or further from the goal it needs to reach, and do this at each action taken. This can be done by subtracting the previous distance to the goal to the current distance. If the agent got closer to the goal, this will be a positive number, while removing itself further from the goal will result in a negative reward. These values can be very small, so a constant is used to scale it up as follows:

$$r_{travel} = c_{travel} * (distance_{prev} - distance_{new}) \tag{4.2}$$

The agent will also receive a reward when the goal is reached:

$$r_{goal} = c_{goal} \tag{4.3}$$

Collisions

To avoid collisions with other moving agents or obstacles in the simulation environment, they have to negatively affect the reward function. No distinction between moving agents or obstacles will be made:

$$r_{collision} = c_{collision} \tag{4.4}$$

When two agents collide at a certain timestep, and they still collide at the next timestep, only the initial collision will count. This ensures that collisions are only counted once, otherwise negative rewards would grow very large, and this could hinder learning. Also, we differentiate between collisions for standard obstacles/agents and obstacles that serve as boundaries. For these boundary obstacles, a separate value is introduced that is lower than $r_{collision}$. Unlike normal collisions, boundary collisions can happen at any timestep.

$$r_{clip} = c_{clip} \tag{4.5}$$

4.3 Expanding to multi-agent

Now that the basic concepts of the environment were addressed for a single steering agent, the focus can shift to multiple steering agents and how they are mapped to RL agents. All the principles that were explained previously in this section will still apply. In general, the methods applied to transition to a multi-agent system will follow the training schemes explained in Section 2.6. RLLIB, which is the deep reinforcement learning framework used in this research, allows for easy multi-agent setups. In the single agent case, the setup in Figure 4.5 applies. A multi-agent environment in RLLIB acts as a wrapper around multiple single agents that have their own single-agent environment. This wrapper is an environment on its own, and thus also has the *step* and *reset* methods, as explained in Section 4.2. Each single-agent environment will handle the movement of one steering agent (except for the centralized case, where there is one environment for all the steering agents). The multi-agent environment also has to distribute the positions of the individual agents to every single-agent environment, at every timestep. This is necessary because each steering agents needs to be aware of the other agents' positions.

The number of policies that will be learned is determined up front, and a function is created that maps ids of learning agents to their corresponding policy. It is possible that multiple agents learn the same policy, or each agent may have their own policy they learn. This principle is shown in Figure 4.6. To explain the different multi-agent approaches, similar illustrations will be used.

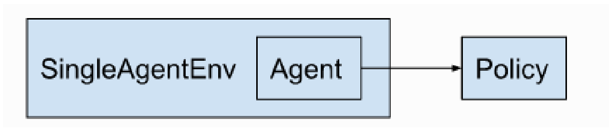


Figure 4.5: Illustration of single agent case in RLLIB, where one learning agent has a single policy [92]

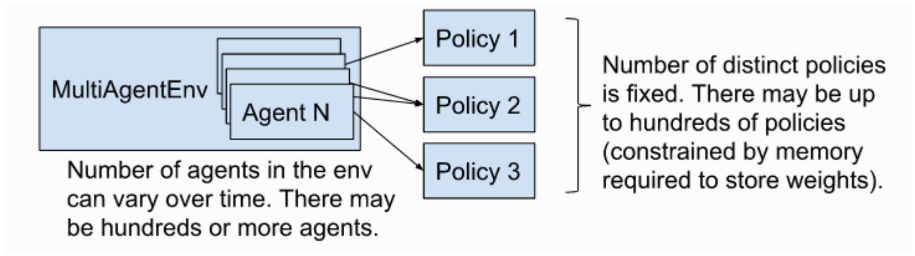

MultiAgentEnv
Agent N
Policy 1
Policy 2
Policy 3
Number of agents in the env can vary over time. There may be hundreds or more agents.
Number of distinct policies is fixed. There may be up to hundreds of policies (constrained by memory required to store weights).

Figure 4.6: Illustration of multi agent case in RLLIB, where agent to policy mapping is adjustable [92]

Centralized

A centralized approach is essentially the same as a single-agent approach. Only now, the action and observation space have to include actions and observations for all steering agents. If there are N steering agents, and each steering agent has a linear velocity v and angular velocity w, the joint action space would be $[[v_1, w_1], [v_2, w_2], \dots, [v_N, w_N]]$. As mentioned before, this method's scalability is not ideal and can fail for more complex tasks.

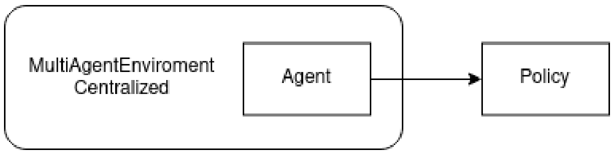


Figure 4.7: Illustration of multi-agent case that uses a centralized approach

Concurrent

The concurrent method employs a one-to-one mapping between learning agents and policies. Each pedestrian corresponds to a single learning agent. As a result, each pedestrian also has their own policy that they learn. When applying this method, the learning agents share no information between each other. There is no cooperation and they have absolutely no idea how other agents perform. The only way of observing other pedestrians is with the laser scanning used to construct the external state. In theory, it is possible to use different reinforcement learning algorithms to train the different learning agents.

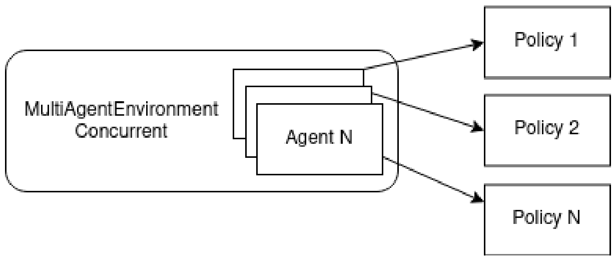


Figure 4.8: Illustration of multi-agent case that uses a concurrent approach, and each agents is mapped to its own policy

Parameter sharing

With parameter sharing, every agent learns the same policy. Using this method, the training process can be sped up because multiple agents are concurrently collecting experiences for a shared purpose. The same principles of concurrent learning also apply here. Agents learn totally independent of each other, they have their own observations and know nothing about other learning agents, except for whereabouts of agents in their view, thanks to their laser scanning observations. The only difference here is that the same policy is trained for all the agents.

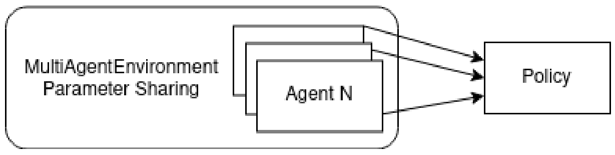


Figure 4.9: Illustration of the multi-agent case that uses a parameter sharing approach, and each agent is mapped to the same policy

MADDPG

MADDPG also applies a CLDE approach as with parameter sharing, only now each agent still learns a separate policy, as with concurrent learning. The centralized critic allows each agent to access actions and observations of other agents during training. As a result, agents can fine-tune themselves based on behavior of others. The initial plan was to train with MADDPG, which is included in RLLIB, the reinforcement learning framework used for this research. However, the implementation did not work as needed for the purpose of this research, and fixing it surpassed our capabilities. See Appendix A3 for details. Nonetheless, the principle on which MADDPG is build, can still be applied.

We implemented a system ourselves where each agent uses a centralized critic for the PPO algorithm. This was done by implementing a custom model used by the PPO algorithm and extending the PPO trainer in RLLIB to alter loss and postprocessing calculations. For the custom model, the default value function prediction is overwritten by a centralized value function that makes predictions of Q-values based on the observation of the current agent together with the observations and actions of all the other agents. Note that the centralized value function does not indicate the used of one single value function for all agents. Each agent still has its own value function, the centralized part only refers to data being centralized and available to every single one of the agents. For the centralized value function mapping, a number of hidden neurons are used, which has to be altered based on the scenario that is being executed. Section 4.4.3 expands on these values. The advantage of using a centralized critic, is that it gives the agents insights regarding behavior of others, which in turn should allow them to adapt better and reduce training times.

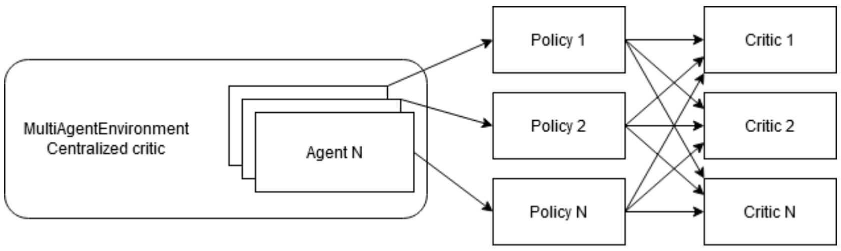


Figure 4.10: Illustration of multi-agent case that uses a CLDE approach, where each agents is mapped to its own policy and a shared critic is used

4.4 Training

Before training of the reinforcement learning model can be started, a number of preliminary steps still need to be addressed.

4.4.1 Setup

The framework used to setup deep reinforcement learning is RLLIB. RLLIB is an open-source library for reinforcement learning that offers high scalability and a unified API for a variety of applications [92]. RLLIB is part of Ray, which is a fast and simple framework for building and running distributed applications[1]. Next to RLLIB, Ray also provides Tune, which is used for experiment execution and hyperparameter tuning [93], which is also applied in Section 4.4.3. RLLIB provides a trainer class trough which the policy can be trained. In multi-agent settings, the trainer manages querying and optimization of multiple policies at once. RLLIB also introduces the concept of multiple workers per trainer. Figure 4.11 shows the trainer class on the left, and multiple rollout workers on the right all optimizing the same policy. The cycle in the rollout worker represents the reinforcement learning process. Each rollout worker has a copy of the environment and will execute the training process. The trainer class can then take the best experiences of all the rollout workers. As a result, workers introduce parallelism and can speed up the training process. However, the amount of possible workers is bound by the number of processing cores present in the system used to train (8 in our case, 4 cores with 2 threads each). 1 core is also reserved by the trainer, so 7 workers is the maximum possible.

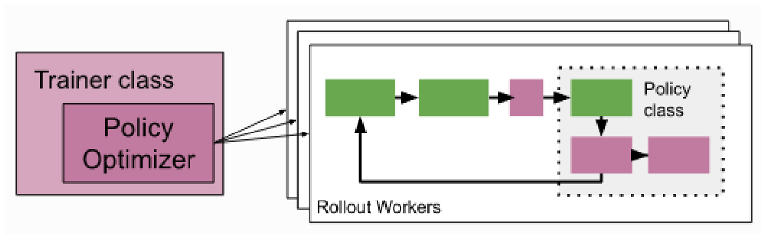


Figure 4.11: Distributed optimization used in RLLIB [92]

Tests were run on a laptop with a 4th generation Intel Haswell i7-4710HQ Quad Core CPU running at 2.5Ghz, 16GB of DDR3 RAM at 1600Mhz. No GPU computing was used in the training process.

4.4.2 Input data

To create test situations on which the reinforcement learning model will train, SteerBench was utilized. As explained in Section 1.4, SteerBench is a benchmark suite used to evaluate steering behaviors [21]. To make evaluation possible, SteerBench provides a number of formatted XML files that serve as test cases. The numerous amount of cases provided by SteerBench served as inspiration and were used to create the test cases applied in this research. Figure 4.12 shows these general cases used to train the reinforcement learning agents. Note that these test cases do not include difficult path finding or navigational problems. Those problems are not the main purpose of this research. We focus on interactions between pedestrians, for

[1]https://ray.readthedocs.io/en/master/index.html

more navigationally oriented crowd simulation problems, other methods are more capable, as discussed in Section 4.1.

4.4.3 RL algorithm and hyperparameters

To create believable steering behavior, it is important that actions and states/observations can be continuous. Looking at the algorithms described in Section 2.4, DDPG and PPO are the two possible options, and both will be tested and explored. For both algorithms, it is important to first figure out what their optimal test conditions are. To achieve these optimal conditions, hyperparameters have to be finetuned. Hyperparameters are parameters of which the values are determined and set before the learning process begins [69]. We differentiate between hyperparameters intended for the environment and for the learning algorithms.

Environment

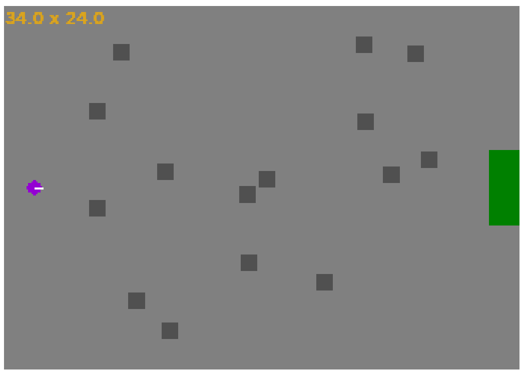
Hyperparameters intended for the environment are parameters that influence state and rewards. Most of these were discussed in Section 4.2 and are set to the values shown in Table 4.1. Some of these parameters are set as stated in [76], others were set according to test results. The linear velocity v is set in consonance with the normal walking speed of humans, the same goes for the angular velocity w and normal turning speed.

ray$_{amount}$	20
ray$_{history}$	3
c_{travel}	5
$c_{collision}$	10
c_{clip}	0.5
c_{goal}	5
timestep	0.1
$v_{min}, v_{max}(m/s)$	-0.5, 1.5
$w_{min}, w_{max}(°/s)$	-45, 45

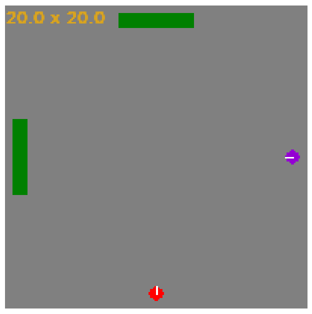
Table 4.1: Environment related hyperparameters and their assigned values

Learning

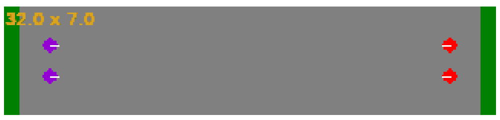
These hyperparameters influence the reinforcement learning algorithm, and thus the learning itself. Generally, when doing hyperparameter tuning, it is not feasible to include every possible parameter that can be tuned, in the tuning process. The Population Based Training method works by perturbing the current parameter specification and keeps the best performing setups. To test all possible combinations,



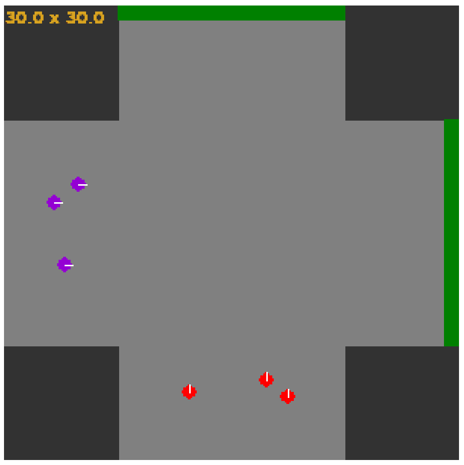
(a) Obstacles



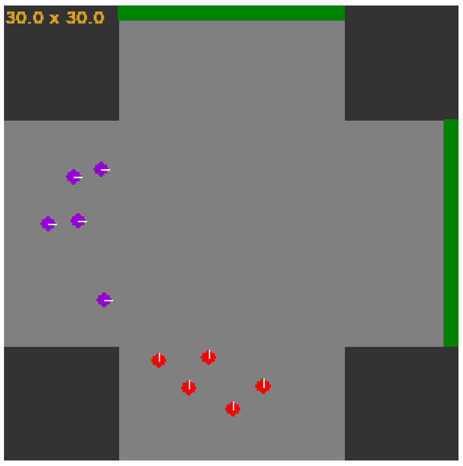
(b) 2-way confusion



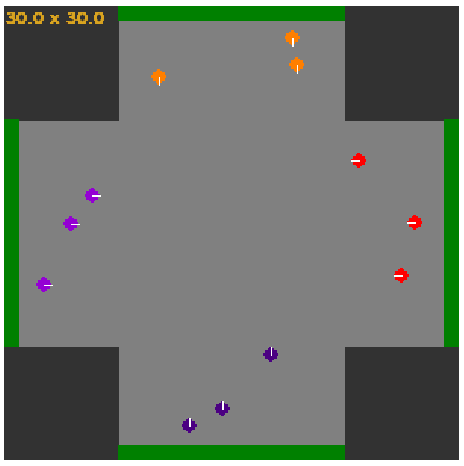
(c) Hallway



(d) Crossway 2 groups 3v3



(e) Crossway 2 groups 5v5



Crossway 4 groups

Figure 4.12: Number of test cases used for training

a lot of perturbations have to be executed. The more hyperparameters need to be tuned, the longer it will take to finish the process, because more possibilities need to be explored. In addition, the training process for a crowd simulation agent can already take a significant amount of time, so testing a lot of options, even on simple test cases, is very time consuming. Therefor, the values of certain hyperparameters are copied from other, existing works while others will be determined using Population Based Training. As input data for this process, the obstacles and hallway with two agents scenarios are used. The hyperparameter tuning is thus done for rather simple examples that still include obstacles or agents, so agents have to learn to avoid collisions. As a side note, the default networks in RLLIB have 2 hidden layers of 256 neurons and use TanH as activation. If not mentioned otherwise, these settings are used.

DDPG

First, some parameters are listed whose values are not changed with hyperparameter tuning. These are already set with optimal values according to the DDPG implementation paper [41] or were used in previous work regarding deep RL and crowd simulation [76].

learning rate$_{actor}$	0.0001
learning rate$_{critic}$	0.001
target update rate τ	0.001
replay buffer size	10^6
θ for OU exploration noise	0.15
σ for OU exploration noise	0.2

Table 4.2: Learning related hyperparameters for DDPG and their assigned values

The parameters displayed in Table 4.3 are decided by hyperparameter optimization. Values often used for the discount factor gamma range from 0.95 to 0.99. The two most common designs for the actor and critic networks in DDPG are two fully connected layers of 64 neurons, or two fully connected layer with the first one containing 400 neurons, and the second one containing 300 neurons. The observation filter determines if the states coming from the environment will be normalized or not, which corresponds to the input of the neural networks. This usually speeds up the learning process. The train batch size determines the size of the batch that is sampled from the replay buffer, values are typically 16, 32 or 64. The sample batch size indicates with how many samples at once the replay buffer is updated. When the sample batch size value was increased, pedestrians started showing suboptimal behavior, as seen in Figure 4.13. Even after training for an extended amount of time, the pedestrian would not escape this learned, unusual behavior. In this case, lowering the sample batch size causes the experience buffer to be updated more frequently. Not as many samples are then added at once, which may stabilize the training process. A slight disadvantage is that it also slows down the training duration a slight amount, but that is a small tradeoff to take. Likewise, introducing multiple parallel workers for DDPG causes similar behavior that made training unstable. As a result, testing for DDPG will not be executed in a parallel manner utilizing multiple workers. Section 5.2.2 elaborates on these issues. The results of the tuning process are shown in Table 4.3.

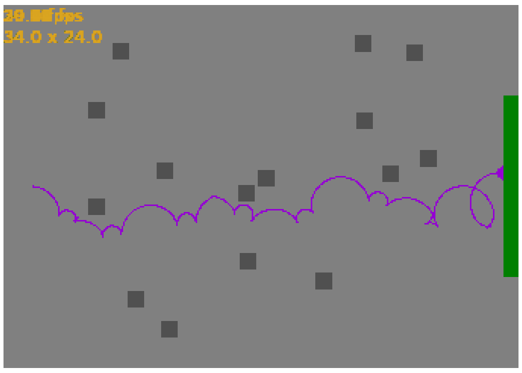


Figure 4.13: Suboptimal behavior of agent when trained with DDPG and higher sample batch size (=8)

DDPG has many more parameters that can be tuned, but most of them are kept to the default value set in RLLIB. See Appendix A2 for all the parameters with their corresponding influences and set values.

gamma	~~0.95, 0.96, 0.97, 0.98~~, **0.99**
actor network	**[64,64]**, ~~[400,300]~~
actor network	**[64,64]**, ~~[400,300]~~
observation filter	~~NoFilter~~, **MeanStdFilter**
train batch size	**16**, ~~32, 64~~
sample batch size	**1**, ~~8, 16~~

Table 4.3: Overview of chosen DDPG hyperparameters after optimization

PPO

Again, for PPO there are also some parameters whose values are assumed as right and will not be tuned, according to previous research or guidelines [81][94]. Use critic implies that an actor critic approach is adopted to implement PPO, as shown in the pseudo code for PPO in Algorithm 9. Use GAE refers the Generalized Advantage Estimator and is used to compute the advantages. Correspondingly, the GAE lambda value is applied in the advantage calculation. Finally, the learning rate indicates the step size of the stochastic gradient descent step.

use critic	True
use GAE	True
GAE lambda	0.95
learning rate	0.00005

Table 4.4: Learning related hyperparameters for PPO and their assigned values

PPO, as opposed to DDPG, should be more robust and less sensitive regarding hyperparameters. This does not imply that less effort should be put into tuning these parameters, but experiences or values may vary when comparing to other research. Table 4.5 displays the parameters that are tuned with Population Based Training. Just as with DDPG, the discount factor gamma ranges from 0.95 to 0.99 and the observation filter determines if normalization is used or not. The clip param ϵ corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating. Setting this value small will result in more stable updates, but will also slow the training process [95]. The entropy coefficient makes the policy more random and thus introduces exploration. It is applied in the calculation of the loss function, a higher entropy coefficient will ensure that more random actions are taken. Train batch size indicates how many steps need to be collected for each round of SGD, and is displayed as NT in the PPO pseudo code shown in Algorithm 9. This number should be large enough so that all important behavior of agents can be captured within a single sequence. The number of epochs indicates how many passes through the train batch are executed and corresponds to K in Algorithm 9. SGD minibatch size is the amount of experiences used for each gradient descent update and is displayed as M in Algorithm 9. Note that the train batch size and sgd minibatch size should be multiplied by the amount of parallel workers used. In the training setup used for this research, there are 7 workers possible, because 8 cores are available and 1 is reserved to the trainer. So if there are 7 workers up and running, the train batch size should be $7 \times 512 = 3584$ and the sgd minibatch size should be $7 \times 16 = 112$.

For the centralized critic example, we mentioned in Section 4.3 that the centralized value function has a certain amount of hidden neurons, depending on the scenario used. The input for the centralized value function is the observation of the current agent, the observation of all the other agents and also actions of all the other agents. The amount of inputs for a single observation is 128, while the amount of inputs for a single agent's actions is 2. To get the total amount of inputs, we simply do $128 \times N + (N-1) \times 2$, with N being the number of agents. Based on this amount, we can select an appropriate amount of hidden neurons. Table 4.6 displays these amounts.

As with DDPG, PPO also has many more parameters that can be tuned, and some of them are kept to the default value set in RLLIB. See Appendix A2 for all the parameters with their corresponding influences and set values.

gamma	~~0.95, 0.96, 0.97, 0.98~~, **0.99**
observation filter	~~NoFilter~~, **MeanStdFilter**
clip param ϵ	~~0.1~~, **0.2**, ~~0.3~~
entropy coefficient	~~0, 0.05~~, **0.01**
train batch size	~~256~~, **512**, ~~1024, 2048~~
number of epochs	~~3~~, **5**, ~~10~~
sgd minibatch size	**16**, ~~32~~, ~~64~~

Table 4.5: Overview of chosen PPO hyperparameters after optimization

2-way confusion (2 agents)	256
Hallway (4 agents)	512
Crossway 3v3 (6 agents)	768
Crossway 5v5 (10 agents)	1024
Crossway 4 groups (12 agents)	1280

Table 4.6: Number of hidden neurons for centralized value function mapping

4.5 Visualization

Reinforcement learning algorithms are not very sample efficient. As a result, training processes can take a long time to complete. For this research, it was rather important to have some insight as to what is happening during the learning. Afterwards, when the training process is over, being able to simulate learned behavior for validation and evaluation is also a key element. In addition, having more control over input data by viewing and alternating it before the training starts can also help. Altogether, there are a total of three possible visualizations provided:

1. Input data visualization
2. Live/training visualization
3. Simulation visualization

The input data visualization just shows a still frame of the environment setup like Figure 4.12: the agents, their goals (in green), all the obstacles and the dimensions. For the live training, there is an option to turn the scanning lasers on or off. Each of the lasers is color coded: red means another agent is in sight, green means the goal is spotted and white indicates obstacles. Note that in Figure 4.14, the number of lasers is lowered from the used amount in training because it improved visualization for the image. To show training results, the simulation visualization will simulate the full crowd simulation from front to back. The simulation speed and FPS are alterable and the followed path by the agents are shown, as Figure 4.15 shows. The live visualization was very useful to detect certain flaws in the system. Sometimes, agents got stuck in certain points or kept dwelling around inside obstacles. Things like this can only be noticed when visualization is present that provides extra, very useful information. Without this tool, the only feedback received are some numbers, from which it can be rather challenging to extract useful information. To solve these issues, agents were reset to their original position when they spend more than a specified amount of timesteps in said issues. Such additions can be helpful to prevent the agents from getting stuck in local optima.

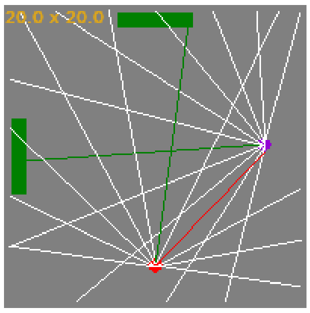


Figure 4.14: Live training visualization with color coded lasers enabled on the 2-way confusion test case

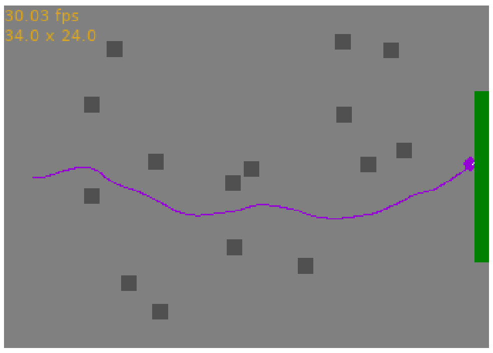


Figure 4.15: End results of visualized simulation that shows the followed path of the agent on the obstacles test case

4.6 Generalization

Training a reinforcement learning model exclusively on a specific environment causes the model to overfit heavily on this specific problem. For crowd simulation purposes, having a model that can generalize and adopt to new environments without requiring extra training is very interesting. To acquire this trait, the model has to be trained on a wide variety of test cases. The power of deep representation can then be utilized to derive behavior in similar situations. In our case, this would only be feasible for the parameter sharing approach. Here, a single policy is learned to control all pedestrians. This provides the necessary scalability to easily adapt to the number of agents. Curriculum learning also plays a very important role in the generalization process, and as previously mentioned in Section 2.7, it can be done relatively simple only for a parameter sharing approach.

Lee et al. [76] implemented such a generalized model by training a single unified policy with a curriculum learning method that utilizes an environment pool. The environment pool consist of numerous different test scenarios, similar to the test cases in Figure 4.12. Each of these test cases have different variations, starting with a low difficulty and increasing as the training advances. Early on in the learning process, simple test cases have higher probability to occur. When the learning progresses, their probability will gradually decrease and thus increase for harder tasks. For example, the training will start with a single agent learning to reach a goal in an empty room. Next, some obstacles are added into the room, then other agents are added etc.

The main purpose of this research is to explore and compare multi-agent approaches. Generalization does not fit in this objective. Another direction that could be taken is to continue with the parameter sharing method and train the model on a set of scenarios, so that it can adapt to unseen ones. This trained model could then be compared to established crowd simulation methods like the ones discussed in Section 1, e.g. Social Forces. However, the emphasis would not be on multi-agent RL anymore.

Chapter 5

Evaluation

5.1 Comparing multi-agent methods

To compare the multi-agent approaches discussed in Section 4.3, we differentiate between the performance, indicating how the learning process elapsed, and the resulting behavior displayed by the pedestrians after the learning had been completed. For both parts, the training schemes and the algorithms used will be discussed and compared for each of the test scenarios.

5.1.1 Performance

Generally, discussing performance for reinforcement learning includes evolution of gained reward by the agents in function of the elapsed time. In this case we use *mean_episode_reward*, which is a metric built into RLLIB. After each iteration, it displays the mean reward of the N last completed episodes, with one episode being ended by all pedestrians reaching their goals. Setting this parameter too high might make bad performing episodes that happened a while ago too impactful. For these tests, N was set to 20. Depending on the learning algorithms, other factors such as loss evolution may also be touched upon, but we won't do so for this thesis.

However, when comparing different training schemes and algorithms directly to each other, the time in which they complete tasks is also an interesting metric. For the different test scenarios, the reward needed for each agent to successfully reach their goals can pretty easily be determined upfront, based on the initial assumptions made when shaping the reward function. Then, each approach can be trained on these test scenarios, and the time they spent until their *mean_episode_reward* reaches the predetermined reward, can be compared. However, this assumes that the learning process will always execute perfectly without any collisions. This may be the case for the easy scenarios, but when more and more pedestrians are added, this may become infeasible. A solution to this problem would be to slightly lower the predetermined goal reward, to allow leeway. This concept is mainly introduced to provide an end goal for the test scenarios, which in turn allows for better comparison between different methods. Reaching a the perfect reward over 20 episodes is very hard, because learning tends to be a bit unstable and some episodes certainly perform a bit worse. However, this may be solved by letting the algorithm train for an extended period of time. Nevertheless, determining the right amount of leeway can be challenging and different training schemes or algorithms may require different leeways.

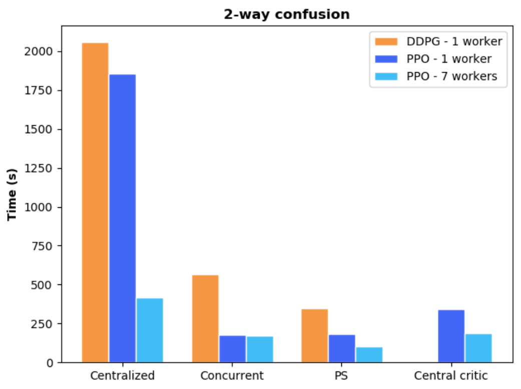
Another side node to make, is that algorithms could also be compared based on iterations executed instead of time spent. However, some algorithms may take longer to execute iterations then others, so using time seemed like a more fair option. For the easy, shorter test scenarios, at least 5 samples were ran and averages of those were taken. For longer scenarios, the aiming point was still 5 samples, however some cases need to run multiple hours so it sometimes is not feasible. In cases where algorithms or configurations fail to achieve desired behavior, evolution of reward may be shown to get more insights. For the 2-way confusion case, both options are displayed, as seen in Figure 5.1. Also, for DDPG there are no results for centralized critic, because it is only implemented for PPO.

2-way confusion

For 2-way confusion, the total available reward for two pedestrians combined was 175, this was also the goal to reach. Therefor, no collisions may occur. The leeway in this scenario is set to 0. When looking at the results displayed in Figure 5.1, it can immediately be seen that centralized learning is the worst performer of the bunch. For a single worker, the difference between DDPG and PPO is minimal. For 7 workers, performance is a lot better however. In general, DDPG is a bit slower than PPO. For the concurrent, PS and centralized critic methods, there is not that much of a difference between 1 or 7 workers. Because this is such a simple example, not much very decisive information can be extracted by comparing these three faster performing methods on this simple example. Comparing them will become easier once moving on to more complex examples.

Hallway

For the hallway scenario with two pedestrians on each side, the total available reward is 560. The goal to be reached was set to 540, which indicates that two collisions are allowed regarding the mean episode reward. At this stage, the centralized method was unable to finish any of the tasks after more than 2 hours of training, so it was left out of the comparison. When the amount of pedestrians increases, it is logical that the centralized method will struggle more. For one pedestrian, the selected joint action might be advantageous, while it is bad for others. Finding the balance where multiple agents show desirable behavior is very hard.



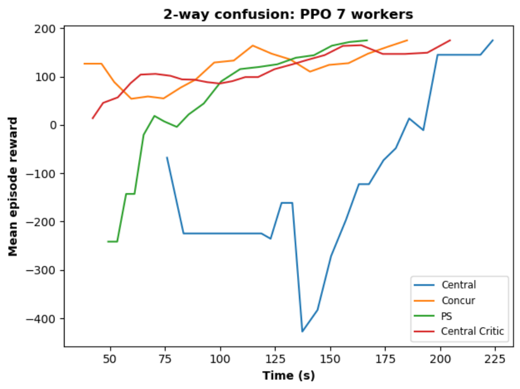


Figure 5.1: Comparison of performance of different training schemes and algorithms for the 2-way confusion example on the left, with a single sample of reward evolution on the right for the PPO 7 workers case

Looking at Figure 5.2, DDPG and PPO performed more or less the same for the single worker case. However, for 7 workers, performance drastically increases for PPO. Parameter sharing sees the biggest advancement, learning time is cut in half. The concurrent and centralized critic methods have more or less the same profit margin in terms of percentage. However, the execution time for centralized critic is way lower than for the concurrent method. For the single worker case, parameter sharing and centralized critic are more or less the same. However, for the 7 workers case, parameter sharing is a bit faster.

Here, we can already see the influence of having the centralized critic. If agents know of the whereabouts and the actions of other agents, they can adapt way better and learning will also go way faster. The two centralized learning, decentralized execution methods are on top and almost comparable for this scenario.

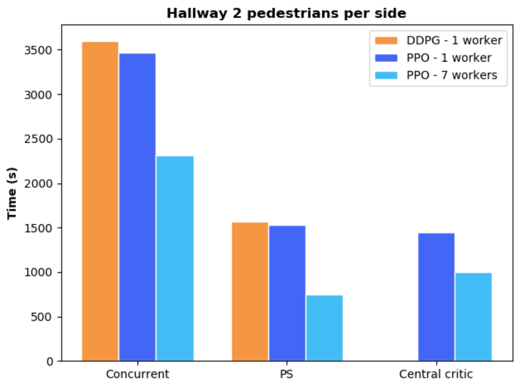


Figure 5.2: Comparison of performance of training schemes and algorithms for the hallway scenario with 4 pedestrians

Crossway 2 groups

From this point on, we decided to continue only with the PPO configuration with 7 workers. DDPG and PPO with 1 worker got the task done but were significantly slower, and as the difficulty of the test scenarios increases, parallel data collection also becomes more important, especially to speed up the learning process. For the crossway example, we tested two cases. One where the two groups consist of three pedestrians, and the other were they consist of five pedestrians. For the first test case, the total available reward was 740. Here, three collisions were allowed so the mean has to reach 710. For the second test case this total amount was 1190 and five collisions were allowed, so the mean had to reach 1140.

The results for the 3v3 scenario are displayed in Figure 5.3. We can see that the concurrent method again is the worst performer, while parameter sharing is the best one. The centralized critic method is not far behind. When moving on to the second scenario with two groups of 5 pedestrians, we ran into some problems regarding the centralized critic method. Having a centralized value function that needs to take all observation and actions into account, requires a lot more memory during execution. Also, the central value function mapping requires way more hidden neurons now, so the model also demands more memory. For previous scenarios, this was no issue as there were only up to 6 concurrent pedestrians present. However, when the amount of pedestrians increases, the required memory also goes up exponentially. For 10 pedestrians and 7 parallel workers, this seemed problematic. Here, we decided to execute the 5v5 scenario for centralized critic with a single worker. Previous tests displayed that the difference between 1 and 7 workers was the smallest for the centralize critic method. For the 5v5 scenario, similar results are displayed as with the 3v3. Concurrent again is the worst performer of the bunch, with parameter sharing almost cutting its time in half. For the centralized critic method, a single worker approach really struggled and did not succeed in reaching the needed mean episode reward. This indicates that parallel data retrieval still is very important to achieve better or faster learning. Figure 5.4 displays the results, together with a reward progress sample. This sample clearly shows that the centralized critic methods struggles and stays around the same mean value, without ever reaching the needed peak. This behavior continued with other training samples. Maybe, longer training would fix this issue, but then it would be even slower than the concurrent method. If no limitation was present for required memory, the centralized critic method would probably perform similar to the crossway scenario with 2 groups of 3 pedestrians, where it is slightly slower than the parameter sharing method.

Crossway 4 groups

The crossway scenario with 4 groups contains 3 pedestrians on each side of the 2 crossing roads. The total available reward is 1490 and a total of 7 collisions were allowed, so the mean had to reach 1420.

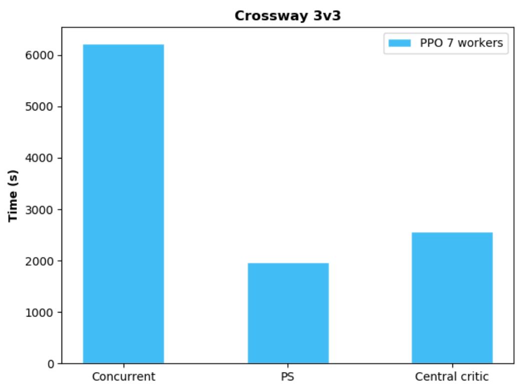
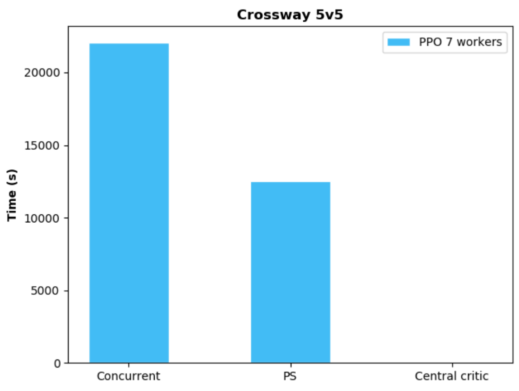


Figure 5.3: Comparison of performance of different training schemes and algorithms for the crossway 3v3 scenario



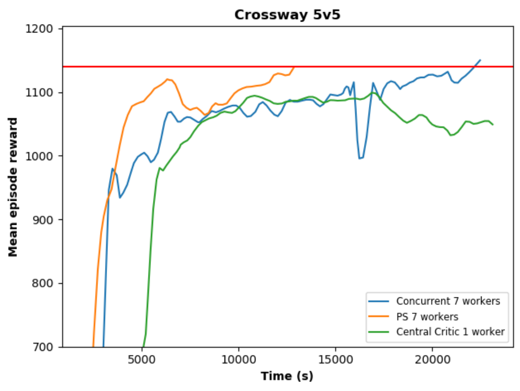
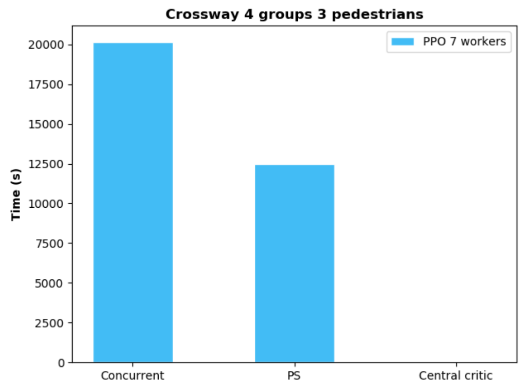


Figure 5.4: Comparison of performance of different training schemes and algorithms for the crossway 3v3 scenario the left, and reward progress for a sample on the right

When looking at the left side of Figure 5.5, the same story is told as with the other scenarios. Parameter sharing is almost double as fast as the concurrent method. For the centralized critic, the memory problem remains, so again the tests were ran with 1 worker. Just as with the 5v5 scenario, the single worker approach fails to reach the goal reward, as can bee seen on the right side of Figure 5.5.

Capping vs. uncapping lasers

All of the experiments above were executed with no cap on the outgoing lasers. To compare capping vs leaving the lasers uncapped, we tested the hallway and the crossway with 2 groups of 3 pedestrians scenarios, for both the parameter sharing and the concurrent methods (central critic is not discussed because it is similar in performance to PS) by capping the lasers on a max distance of 10. These results were compared to previously completed tests and are displayed in Figure 5.6. For



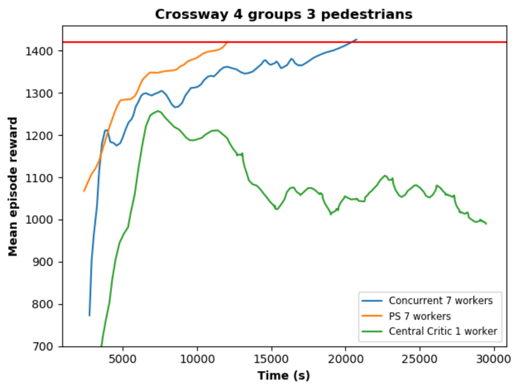


Figure 5.5: Comparison of performance of different training schemes and algorithms for the crossway scenario with 4 groups of 3 pedestrians on the left, and reward progress for a sample on the right

the hallway scenario, the difference in training time is less than with the crossway scenario. This is due to the hallway being rather narrow, so lasers going towards the sides of the hallway already have some natural capping. Lasers going straight forward do not experience capping, so here lies the possibility for some time to be won. For the crossway example, capping the lasers provides more time gain due to the scenario being more spacious. Having the lasers capped in a spacious environment could also play into the pedestrian's disadvantage. Their scanning of the environment is not endless so they might have no clue in which direction they are heading or what will eventually be in front of them. However, for this scenario, that did not seem to be the case. The difference between parameter sharing and the concurrent method is also clearly visible. Parameter sharing showed more gain from being capped than the concurrent method. This may be due to the fact that percentage wise, the laser scanning takes up more of the execution time of parameter sharing. For the concurrent method, this percentage is lower because more than 1 policy is trained which introduces more overhead and prolongs execution time. This also causes the difference in gain to be lower for the concurrent method.

5.1.2 Movement of pedestrians

When looking at movement of pedestrians, things to pay attention to are mostly how pedestrians move and how they react when other pedestrians approach or are nearby. Again, each scenario is discussed based on algorithm and configuration used. Explanations will be supported by pictures of pedestrian paths. Videos for all the test scenarios are also included with this thesis.

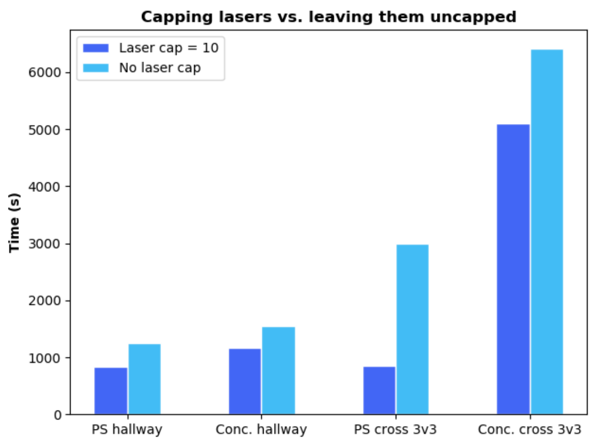
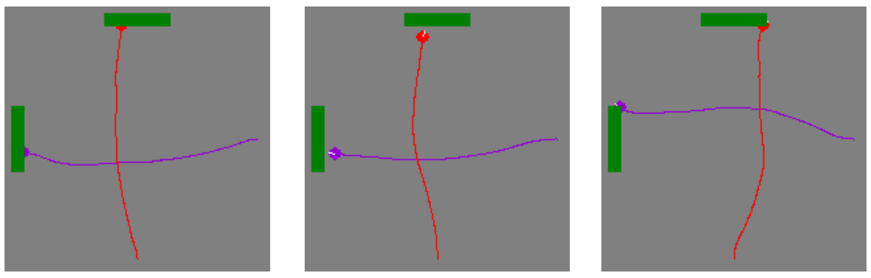


Figure 5.6: Comparison of training time capping lasers vs. leaving them uncapped for some different scenarios

2-way confusion

No significant difference is noticeable in pedestrian behavior between the three algorithm setups used (DDPG 1 worker, PPO 1 worker, PPO 7 workers). Behavior can vary between different samples of the same algorithm or between different algorithms, but nothing significant showed up and general behavior is the same. This is shown in Figure 5.7, where the paths are displayed for the parameter sharing approach using each of the three algorithm setups. In all three of the scenarios, both pedestrians display similar paths taken. This is further explained later on in this section. Similar generalization holds for the concurrent, centralized and central critic approach. This indicates that the choice of algorithm mostly influences the speed of learning, not the behavior of pedestrians.



(a) DDPG 1 worker (b) PPO 1 worker (c) PPO 7 workers

Figure 5.7: Comparison of pedestrian paths for each of the three algorithm configurations

To further compare behavior of pedestrians depending on the chosen multi-agent approach, we will focus on the PPO 7 workers setup. Figure 5.8 shows samples of pedestrian paths for all multi-agent approaches. As mentioned above with Figure 5.7, pedestrians using the parameter sharing method display almost identical paths. This is most probably due to the shared policy that steers them both. Good learned behavior for one agents will be duplicated by others. When multiple policies are trained, as with the concurrent and centralized critic methods, pedestrians show differing behavior. Unlike parameter sharing, they act totally independent. For this test case, one pedestrian moves mostly straight forward, while the other adjusts its path based on the first pedestrian. For parameter sharing, both agents would adjust their path. In this case, no significant difference is visible between the concurrent and the centralized critic approach. The behavior of pedestrians for the centralized method does not really contain recurrent patterns. A good action for one pedestrian may negatively influence another one, so a lot of trial and error is needed to arrive at actions that positively influence both.

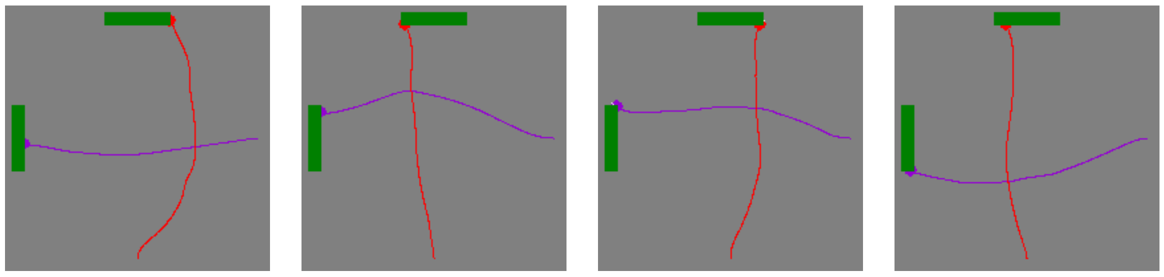

(a) Centralized (b) Concurrent (c) PS (d) Central critic

Figure 5.8: Comparison of pedestrian paths for each of multi-agent approaches for the 2-way confusion scenario with PPO 7 workers

Hallway

Just as with the performance section, starting from this scenario, the centralized method will not be considered anymore. Looking at the pedestrian paths in Figure 5.9, similar behavior is shown as discussed for the 2-way confusion scenario. For the parameter sharing method, pedestrians will again show similar behavior and flock together in pairs. For the concurrent and central critic methods, the pedestrians do not display flocking or similar behavior and act independently. In both these situations, pedestrians groups cross through each other. For the concurrent method, a parallel crossing is displayed while a more chaotic crossing is shown for the central critic method. These observations are not tied to these scenarios and can occur for both of them. At this point, no noteworthy difference in behavior is visible between the concurrent and the central critic methods.

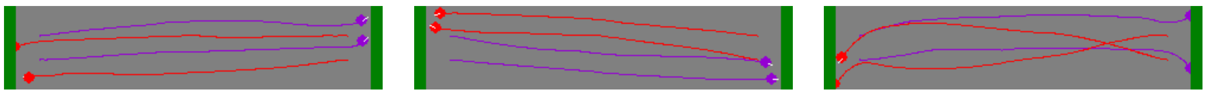

(a) Concurrent (b) PS (c) Central critic

Figure 5.9: Comparison of pedestrian paths for each of multi-agent approaches for the hallway scenario with PPO 7 workers

Crossway 2 groups

Figure 5.10 displays pedestrian paths for the tested scenarios. For the crossway with 2 groups scenario, we start with 3 pedestrians on each side. The parameter sharing method shows similar behavior again for all pedestrians. They all slightly deviate to the right and successfully dodge each other this way. For the concurrent and centralized critic method, we observe that the pedestrian groups do not show similar behavior. The red pedestrians more or less go straight forward, and the purple pedestrians adjust their paths around the movement of the red ones. This is similar to the behavior of these two methods for the 2-way confusion scenario, where one pedestrian goes straight forward and the other adapts. In this scenario, the red pedestrians will arrive at the center a tad earlier than the purple ones, which will cause the purple pedestrians to adapt. For the concurrent method, we observe that the outside pedestrians deviate to the sides. The top one does that to cut in front of the oncoming red pedestrians, while the bottom does so to go behind the red pedestrians. The middle one can just continue straight ahead as the red pedestrians passed by when he arrives at the center. For the central critic method, we observe other behavior. The purple pedestrians do not really deviate a lot from their paths, but will stop to let the red pedestrians pass, and then continue on wards (better visible in video). I believe this is due to the fact that they can access other agents' observations and actions. This way, they do not have to guess and find other paths like with the concurrent method. They know it is highly probable another pedestrian is going to cut in front of them, so they just wait and continue when the pedestrian passed.

Figure 5.11 displays pedestrian paths for the tested scenarios. For the second variation of this scenario, 5 pedestrians were placed on each side. Again, for parameter sharing we observe more flocking like behavior. Pedestrians stay more central in the crossing, and also arrive more centrally at the goal. This is again due to the fact that they share the same policy. For the concurrent method, agents show independent behavior again. Unlike the example with 3 pedestrians, both pedestrian groups now adjust their paths. The red pedestrians don't just continue straight forward, they break up. The leftmost two deviate behind the purple pedestrians and the other three cut in front of the three rightmost purple pedestrians. Similar behavior is shown by the purple pedestrians. It can clearly be seen however that there is a large difference between independent acting pedestrians and those controlled by

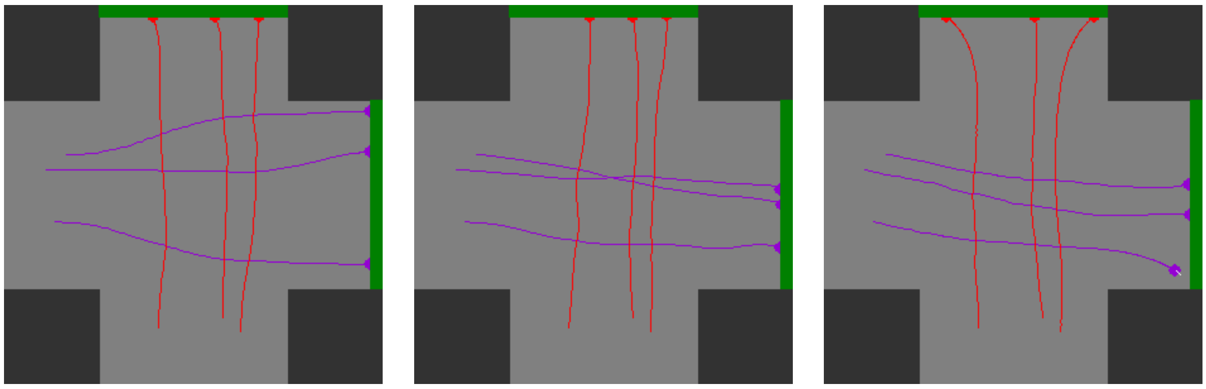

(a) Concurrent (b) PS (c) Central critic

Figure 5.10: Comparison of pedestrian paths for each of multi-agent approaches for the crossway 2 groups of 3 pedestrians scenario with PPO 7 workers

the same policy. The centralized critic method failed to successfully learn steering behavior for this scenario, as explained in the performance part. Similar behavior to the concurrent method would be expected again, only instead of avoiding collisions by taking a detour, pedestrians would maybe wait and let others pass as with the hallway 3v3 example.

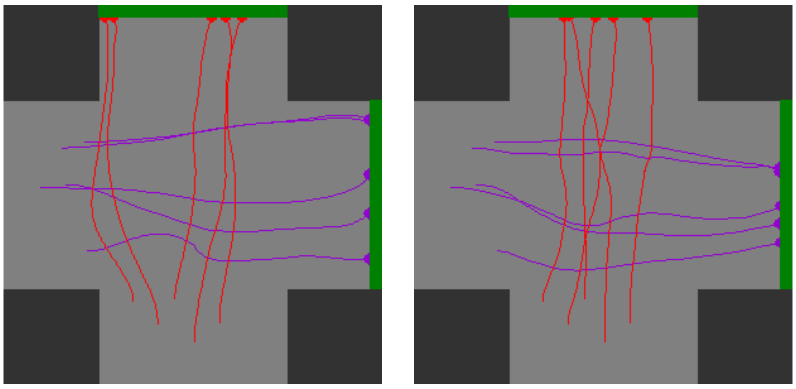

(a) Concurrent (b) PS

Figure 5.11: Comparison of pedestrian paths for each of multi-agent approaches for the crossway 2 groups of 5 pedestrians scenario with PPO 7 workers

Crossway 4 groups

Figure 5.12 displays pedestrian paths for the tested scenarios. Again, the same behavioral patterns are visible. For parameter sharing, agents generally stay closer to the center of the crossway and arrive at their goals closer to each other. Behavior is sort of similar again between the pedestrians. For the concurrent method, agents act independently and go their own ways. It would have been very interesting to see how the centralized critic approach performed for this task. Would pedestrians hesitate more because the know about positions and actions of others? Would they let other pass or smoothly pass through each other?

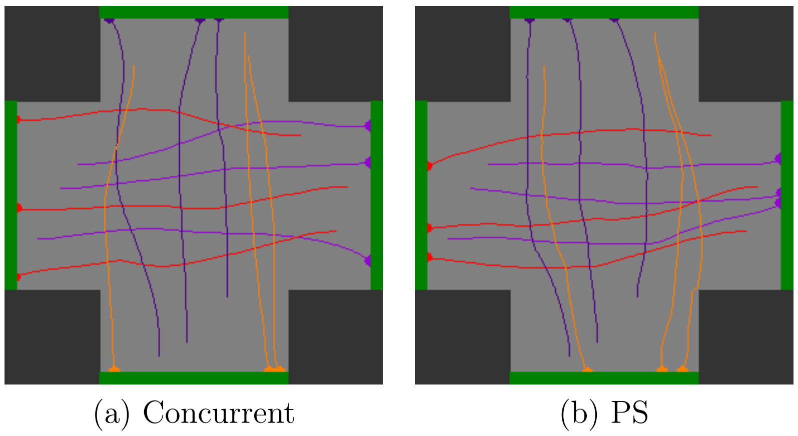

(a) Concurrent (b) PS

Figure 5.12: Comparison of pedestrian paths for each of multi-agent approaches for the crossway 4 groups of 3 pedestrians scenario with PPO 7 workers

Capping vs. uncapping lasers

Figure 5.13 displays pedestrian paths for the tested scenarios. For the hallway scenario, similar behavior was shown for the concurrent and the parameter sharing methods, so only the parameter sharing method is displayed. The pedestrians continue to go straight forward for a longer period of time, until they spot oncoming pedestrians. They will then adjust their paths a little bit if needed. For the uncapped scenario, we saw that pedestrians would pick a side earlier and also stick together in pairs. The capped PS scenario shows behavior similar to the concurrent and central critic methods that do not apply capping on the lasers, where agents cross through each other. For the crossway scenario with 2 groups of 3 pedestrians, there is not really much of a significant difference in behavior with or without capping lasers. For the PS method, both pedestrians groups deviate a bit more to the right. For the concurrent method, the red pedestrian group now adjusts their path more and avoids the purple pedestrians, where it was vice versa for the uncapped scenario. From these observation, no specific differences can be inferred.

If the laser distance would be set even lower, let's say at 5, the pedestrians would be forced to adjust even later on, which could produce interesting results and would speed up the training process even more. However, this is even less realistic, as pedestrians in real life have no limitation on what they can see, or at least not in the order of magnitude for the laser distance.

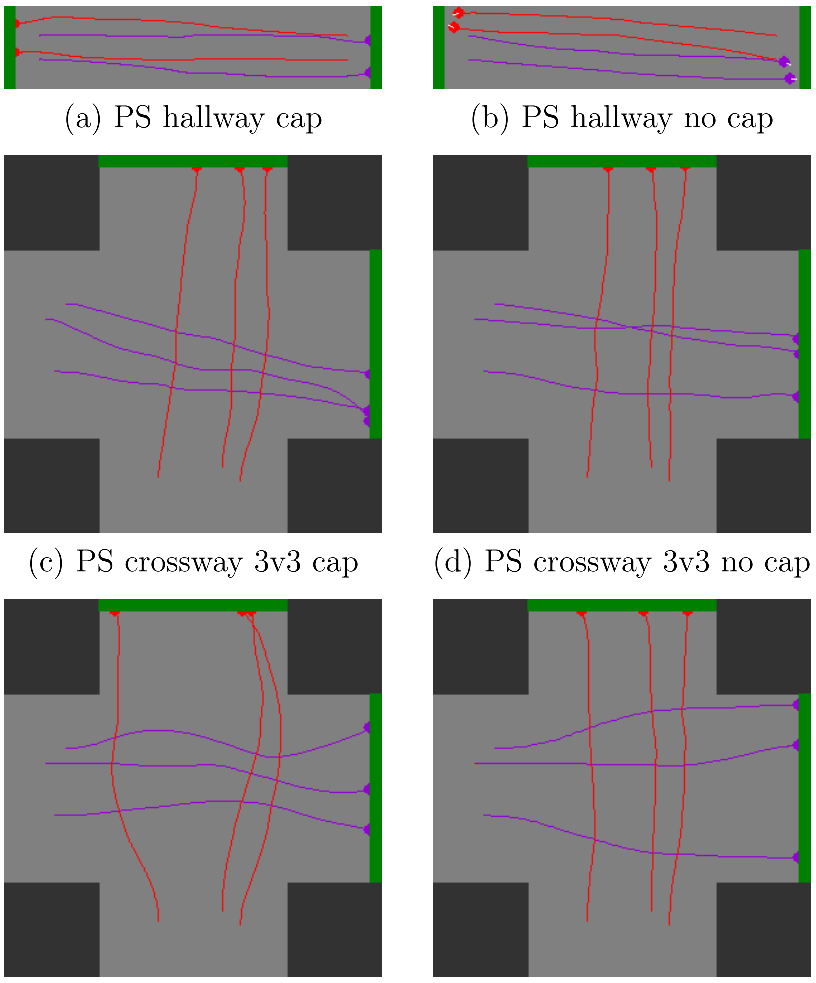

(a) PS hallway cap (b) PS hallway no cap
(c) PS crossway 3v3 cap (d) PS crossway 3v3 no cap
(e) Conc. crossway 3v3 cap (f) Conc. crossway 3v3 no cap

Figure 5.13: Comparison of pedestrian paths for each of multi-agent approaches for the crossway 2 groups of 3 pedestrians scenario with PPO 7 workers

No time constraint

The design of our reward function includes no time related section, and the relative distance to the goal is used to reward pedestrians. As a result, it makes no difference if the pedestrian takes twice as long and follows a detour to arrive at the destination without collisions. As a result, pedestrians will often not follow optimal paths, and will sometimes rather wait for others to pass by instead of going around. This could be solved by adding some sort of time efficiency mechanism into the reward function. This might also partly fix the detour problem, as detours take longer to travel.

5.2 Importance of curriculum learning

Section 2.7 mentioned how curriculum learning operates and what its purpose is for reinforcement learning. Here, we attempt to prove this purpose and support these claims with experimental evidence. To reiterate, curriculum learning is especially useful for the parameter sharing method. Learning a single policy for all agents ensures easy scalability to more complex test scenarios. We distinguish two main reasons why curriculum learning can play an important role in the RL field.

5.2.1 Speed-up

Curriculum learning can generally allow tasks to be learned more swiftly. Applied to our crowd simulation research, the process will proceed as follows: starting with a single pedestrian, the learning will continue until desirable behavior is reached. At this point, an extra pedestrian can be added, and parameter sharing ensures that this second agent has the same knowledge as the first one. Learning will then continue and both agents can adjust to the presence of extra agents. More and more agents can be added, and because the policy already has experience on interacting with other agents, it will also learn more quickly how to deal with the presence of even more agents. Curriculum learning can easily be implemented in RLLIB with the callback methods that are activated after every iteration the trainer completes. If the mean episode reward is above a certain value for a specific amount of iterations (to ensure it is no fluke), the next stage can be initialized and pedestrians can be added. Figure 5.14 shows the comparison in the evolution of the mean episode reward. Here, we again opted to use the total elapsed time as reference point instead of iterations the learning agent has executed. The reason behind this is that iterations execute way faster when the number of pedestrians is lower, so comparing the two methods based on iterations would hold no value. For this test case, the hallway scenario with four pedestrians is adopted. The first learning phase is a single pedestrian that needs to successfully cross the hallway. This succeeds relatively fast and the second pedestrian on the opposite site can be added. This way, the pedestrians already learn some simple collision avoidance behavior. Again, when learning stabilizes and behavior for two pedestrians is as desired, an extra pedestrian is added on each site to arrive at the full scenario displayed in Figure 4.12. The two dips in reward that can be observed in the learning curve coincide with both additions of pedestrians. The second dip is rather large because more pedestrians are now present and collisions definitely occur. More unseen situations are generated and pedestrians will need to adjust. However, these adjustments happen relatively quick thanks to learned behavior in previous stages. In the end, a difference is visible in total time taken between the CL and no CL method. For more complicated examples with more agents, applying CL in a logical and structured manner might even have more impact and make the learning process smoother. The dips in reward should also stabilize because at a certain point, pedestrians have been trained with a significant amount of other pedestrians around them, so the collision avoidance is already at a higher level. These findings regarding CL also coincide with methods applied in [53][76].

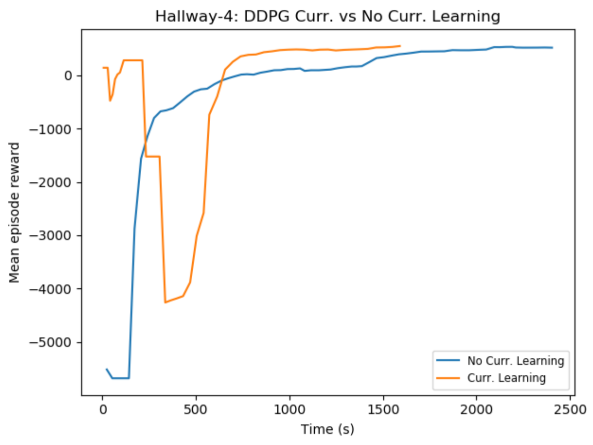
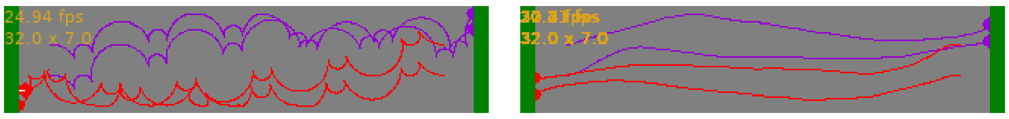


Figure 5.14: Comparing performance of curriculum learning vs. no curriculum learning for the hallway scenario with four pedestrians and DDPG as learning algorithm

5.2.2 Learned behavior

Curriculum learning can also help to improve learned behavior of pedestrians. Especially when using the DDPG algorithm, suboptimal behavior can be displayed by pedestrians when no curriculum learning is applied. Figure 5.15 a) demonstrates such behavior where pedestrians constantly turn right while moving forward, which results in circular like motions. For Figure 5.15 b), pedestrians learned in incremental steps, as explained in the speedup part above, and no such behavior is shown. Note that this behavior is similar to that explained in Section 4.4.3 concerning the hyperparameter tuning. This behavior would also show up if the sample size became too high. The sample size indicates how often the replay buffer was updated. When more pedestrians are added for the parameter sharing method, the number of samples added simultaneously to the replay buffer also increases. This may be the cause of the problem. Further, increasing the number of parallel workers, as discussed in Section 4.4.1, again inflates the number of samples simultaneously added to the replay buffer. Similar behavior is then observed. For DDPG, the number of parallel workers should be kept to one. The problems concerning learned behavior were not present for PPO, so curriculum learning would not provide significant improvements regarding behavior.



(a) No Curriculum Learning (b) Curriculum Learning

Figure 5.15: Difference in end result when Curriculum Learning is applied on the same amount of timesteps for the hallway example with 4 pedestrians

5.3 Linking back to RL problems

In Section 2.3.4, some RL related problems were explained. The first major problem was the curse of dimensionality. However, this problem is solved by using deep RL methods. For deep RL methods, the main problem that remains is the sample inefficiency, as described in Section 2.4.4. When the test scenarios used for this research increased in difficulty, the time needed to successfully train the models also increased significantly. As a result, much more samples were generated and used. For some of the harder scenarios, like the crossways with 10+ pedestrians, training to a near perfect execution took up to 10 hours for some of the training methods. There probably exist methods that are easier to implement and execute way faster, that produce nearly the same results.

Secondly, the models are only trained for one specific scenario. This introduces overfitting on these specific scenarios, and generalization to other unseen scenarios is not possible. When using the parameter sharing method, generalization should be possible when applying a curriculum learning method, as mentioned in Section 4.6. Getting stuck in local optima was not the biggest problem for our test cases. As mentioned in Section 4.5, we used the live visualization during training to see problematic areas where pedestrians would get stuck or would display unusual behavior. If this behavior persisted for a significant amount of time, the pedestrian would be reset to its original position. Most of the times, when undesired behavior was displayed, simply longer training and thus collecting more samples would solve this issue. This last point was difficult to understand at first. I thought there had to be something wrong with the hyperparameters or configurations of the models and algorithms, but often times I did not show enough patience to let the model train long enough and show decent results.

As a side note however, reinforcement learning still tends to be unpredictable and inconsistent. The smallest change in parameters or testing conditions can completely change the outcome of results. Again, this is were I struggled a lot during testing. It was often very hard to find the correct combination of hyperparameters for both the environment and the RL algorithms that resulted in desirable results.

5.4 Conclusion

The goal of this thesis was to implement several deep multi-agent reinforcement learning methods to generate crowds with, and subsequently compare these with each other. This process was initialized by dissecting the different parts needed to arrive at this goal. At first, crowd simulation and the available methods and techniques were discussed. Next up was reinforcement learning, we needed a clear understanding of how this machine learning paradigm operates, and how it extended to deep and multi-agent learning. Because deep learning is applied, an introduction to neural networks was also required. How do they operate and how does it intertwine with reinforcement learning? All this knowledge was then combined to implement four different deep multi-agent reinforcement learning methods, based on three training schemes: centralized, concurrent and centralized learning with decentralized execution. The first two methods stand on their own and have corresponding implementations, while for the last one two variations were implemented: parameter sharing and a centralized critic approach. All these methods were tested on different scenarios and for two different reinforcement learning algorithms that operate in continuous action and space domain: DDPG and PPO. Also, parallel data collection was tested for the PPO method.

For the easy scenarios, all the implemented methods succeeded in successfully learning correct navigation and collision avoidance. When scenarios became increasingly difficult, performance for centralized learning quickly deteriorated and it was unable to successfully learn. This is due to the fact that joint actions are used and a good action for one pedestrian may be bad for the others. A centralized approach is thus not suitable for larger scale multi-agent reinforcement learning. For the harder scenarios, the parameter sharing approach always performed the best. This is due to the fact that a policy is shared between all the agents. There is thus less overhead because only a single policy has to be trained, and each pedestrian is training the same brain so way more information is gathered more quickly. This works because pedestrians are seen as homogeneous entities. The second best performer was the centralized critic approach. Each pedestrian corresponds to a policy, so multiple policies are trained with this method. However, having access to observations and actions of all other pedestrians sped up learning a very significant amount. This becomes extra clear when comparing to the concurrent method, which is essentially the same as the centralized critic, but without having access to observations and actions of other pedestrians.

For the single worker cases, DDPG and PPO performed more or less the same, with DDPG lacking slightly behind most of the time. However, DDPG can not profit from multiple parallel workers as PPO. We decided to continue with the PPO 7 workers method, because it performed the fastest and for some harder scenarios, having one worker did not suffice. For parameter sharing, we observed that parallel data collection had the largest influence, it cut training time in half. For the concurrent and centralized critic approach, profit margins were less but approximately the same in terms of percentage. This may be due to the fact that parameter sharing has less overhead to maintain the different policies that need to be trained. Also each pedestrian already trains the same policy, so having even more parallel data collection logically provides more speedup. The centralized critic method has access to observations and actions of all other pedestrians, which in turn requires a lot of extra memory. We observed that, for the harder scenarios with parallel workers, the memory available on our testing system was not enough. Therefor, full testing for the centralized critic example could not be executed. If memory was unlimited, the centralized critic method would be expected to perform slightly worse than parameter sharing, but way better than concurrent in these scenarios. These findings would be conform to previous scenarios.

Comparing steering behavior of pedestrians between the different learning algorithms (DDPG and PPO) did not show any significant differences. When comparing training schemes, parameter sharing displayed almost identical behavior between pedestrians in simpler scenarios. This is caused by the shared policy that steers all pedestrians. For the concurrent and centralized critic method, pedestrians show more independent behavior. When moving on to more crowded scenarios, the parameter sharing method showed that pedestrians flock together more. They stay more in the centre of the environment and also arrive more central on the goal. This, again, is an indication of the shared policy, because they display samey behavior. The concurrent method showed that pedestrians act independently. They take detours to prevent colliding with others, and do this on their own. The centralized critic method failed to train for the more crowded scenario because of the memory issue mentioned above. However, similar behavior would be expected as with the concurrent method. Only now, pedestrians may sometimes wait and let other pass instead of looking for detours. This behavior was observed for the centralized critic with easier scenarios, and is caused by the fact they they have access to other pedestrians' observations and actions.

To wrap it all up, parameter sharing is the best performing method and also recreates believable steering behavior of pedestrians. However, the question can be asked if it even is a multi-agent reinforcement learning method, because only a single policy is being learned. If enough memory is available and pedestrians should act more independently, the centralized critic approach should be the go to method.

In addition, we have also demonstrated the importance of curriculum learning for the parameter sharing method, as it can speed up the learning process and even give better learning results regarding steering behavior. This works especially well for parameter sharing because only a single policy is being trained, which allows for easy scalability. Capping the scanning lasers used to retrieve environment observations also proved to significantly improve learning times. Regarding behavior, it did not really affect the pedestrians that much. Sometimes, they would continue straight ahead a longer amount of time instead of starting to detour from their paths earlier.

5.5 Future work

Because there is no time constraint in our reward function design, pedestrians will not be pushed to use the optimal path to reach their end goal. Using a detour and arriving at the goal without collisions results in the same reward and using the shortest path. This could be altered to make sure pedestrians display more efficient steering behavior. Also, more explicit communication methods for multi-agent reinforcement learning could be explored, similar to how pedestrians in real life could interact with each other, with vocal or visual signs.

This thesis mainly focused on simulations in open, symmetrical environments. This causes the main focus to be on steering and collision avoidance, but also puts the pedestrians on the same level. Introducing social factors into the mix could cause different hierarchies of pedestrians to arise, for example a school teacher leading all their students. In such cases, having hierarchical method may well come in handy. Both for crowd simulation and reinforcement learning, hierarchical methods already exist so they could be combined in theory. Also, having only symmetrical environments is good for our purpose, but when looking at more navigation or evacuation targeted scenarios, this would not work as well. A possible solution in such cases is to combine path planning and reinforcement learning, as explained in Section 4.1.3.

Appendix A1

Simulatie van menigtes d.m.v. deep multi-agent reinforcement learning: Nederlandse samenvatting

Hoe dat menigtes zich gedragen is al enige tijd een fascinerend onderwerp voor mensen behorend tot verschillende velden, denk maar aan informatica, maar ook psychologie of andere sociale toepassingen. Wat zijn de verschillende motieven die individuen in een menigte drijven? Hierbij zijn het niet enkel individuele doelen die een rol spelen, maar ook beslissingen en gedrag van anderen rondom zich, en ook de plaats waar men zich bevindt. Binnen de informatica bleef de interesse naar computer gebaseerde simulaties voor menigtes ook niet lang achter. De twee grote vakgebieden bestaan hier uit *realisme van gedragsaspecten* en *hoge kwaliteits visualisatie*. De eerste zal voornamelijk gaan over simpele visualisaties in 2D, waarbij het hoofddoel is om goed bewegingsgedrag te creëren. Hierbij zijn evacuaties of sociologische modellen van menigtes enkele voorbeelden. Het tweede vakgebied draait voornamelijk om de visualisaties van individuen binnen de menigtes. Beweging ze op een realistische manier en is het visueel aantrekkelijk? Hierbij denken we aan producties van films of games. Deze thesis zal zich focussen op de eerste toepassing.

Vandaag de dag kan data op sommige vlakken gezien worden als het nieuwe goud. Een overvloed ervan is beschikbaar, en hierbij hoort ook data die menigtes beschrijft. Hierbij denken we bijvoorbeeld aan luchtfoto's of filmpjes opgenomen vanuit bovenaanzicht op festivals of grote bijeenkomsten. Het nadeel hieraan is dat menselijke invoer vaak nodig is om deze data bruikbaar te maken, om vervolgens bijvoorbeeld te gebruiken voor generatie van menigtes. Hier kan reinforcement learning goed van pas komen, wat één van de drie machine learning paradigma's is, waarbij geen invoer data nodig is tijdens het leerproces. Oplossingen zullen gevonden worden door mogelijke opties af te gaan en acties te kiezen die de vooraf bepaalde beloning maximiliseren. Hierbij zullen neurale netwerken het leerproces bijstaan zodat schaling en snelheid erop vooruit gaat. Standaard reinforcement learning processen behandelen een enkele agent die het leerproces ondergaat, echter bestaat een menigte ook uit vele individuen, dus is het logisch dat de reinforcement learning ook op meerdere agenten toegepast wordt. Het doel van deze thesis is verschillende methodes hierbinnen te onderzoeken en met elkaar te vergelijken.

A1.1 Simulatie van menigtes

Een menigte is meer dan enkel een verzameling van individuen. Hoe een individu zich gedraagt is niet alleen afhankelijk van zichzelf, maar ook van andere individuen aanwezig in deze menigte. Daarnaast spelen fysiologische, fysieke en sociale factoren ook een rol in het bepalen van gedrag. Al deze factoren zorgen ervoor dat een menigte interesante dynamieken kan vertonen, die vervolgens zeer interessant zijn om te bestuderen. Wanneer er gefocust wordt op het realisme van gedragsaspecten, kunnen simulaties over het algemeen een gecategoriseerd worden door de grootte van de menigte en de tijdsschaal. Granulariteit van de menigtes is dan een rechtstreeks gevolg van de twee voorgaande parameters, en wordt vaak gebruikt tijdens het modelleren. De drie gebruikte aanpakken zijn dan stroom, entiteit en agent gebaseerd.

Flow-based

Bij flow-based aanpakken worden de eigenschappen van individuen aan de kant geschoven, en is enkel het grote plaatje belangrijk. De menigte wordt als een geheel behandeld. Dit wordt gedaan omdat de computationele kost anders veel te groot kan worden. Er is dus een abstractielaag present, en de menigte wordt gezien als een eenheid die beweegt, wat vergelijkbaar is met een vloeibare stroom. Flow-based applicaties gaan meestal om het bepalen van de stroom in evacuatie of bewegingsprocessen van erg grote menigtes. Typisch worden hiervoor vector velden gebruikt om aan te geven hoe de menigte zich op bepaalde plekken moet gedragen.

Entity-based

Wanneer de granulariteit omhoog gaat, arriveren we eerst bij entity-based methodes. De essentie hier is dat de menigte nu wel gezien wordt als een verzameling van individuen, in tegenstelling tot bij stroom gebaseerde methodes, maar elk individu is een homogene entiteit waarvan de beweging bepaald wordt door bepaalde regels. Deze globale of lokale regels zijn instaat om fysieke, sociale of fysiologische invloeden te hebben op elk van de individuen. Zulke modellen zijn vaak behoorlijk geschikt om vastloping en kuddevorming te modelleren.

Agent-based

De meest populaire methode om simulatie van menigtes aan te pakken is nog steeds agent-based. Deze methode heeft de hoogste granulariteit, wat resulteert in elk individu in de menigte als een autonome agent, die zijn eigen attributen en staat heeft. Een direct gevolg hiervan is dat deze individuen de mogelijkheid hebben om te redeneren en dus ook cognitieve capaciteiten bezitten. Agenten kunnen gestuurd worden door bepaalde beslissingsregels, maar elke agent kan onafhankelijk beslissingen maken. Deze situatie komt dus logisch gezien ook het meest overeen met real-life scenarios, waar mensen wel regels moeten volgen maar zelf invullen op welke manier ze dat doen. Het onderzoek binnen deze thesis valt in deze categorie.

A1.2 Reinforcement learning

Zoals reeds vermeld is reinforcement learning, naast supervised en unsupervised learning, het derde machine learning paradigma. Reinforcement learning houdt zich bezig met sequentiële beslissingsproblemen waarin feedback gelimiteerd is. De agent moet via een trial en error methode gaan interageren met zijn dynamische omgeving, en een bepaalde beloning maximiliseren tijdens dit proces. De agent moet dus zelf uitzoeken welke acties de hoogste beloning geven, en dus het meest voordelig zijn.

De twee belangrijkste factoren voor reinforcement learning zijn dus de *agent* en de *omgeving* waarmee hij interageert. Alles wat niet controleerbaar is door de agent wordt gezien als onderdeel van zijn omgeving. Naast deze twee elementen zijn er nog vier belangrijke elementen: een *policy*, een *reward signaal*, een *value functie* en optioneel een *model van de omgeving*. Een policy mapt de staat van een agent in de omgeving naar een bepaalde actie. Dit bepaalt dus welke actie de agent zal uitvoeren op een bepaald moment. Een reward signaal is nodig om aan te geven wat het doel van het reinforcement learning probleem is, waar willen we naartoe? De agent moet dit signaal maximaliseren, dus het signaal bepaalt de impact van acties. Hiernaast is er ook nog de value functie, die aangeeft wat de verwachte beloning is wanneer een agent zich in een bepaalde staat bevindt and de huidige policy volgt. Acties zullen gekozen worden die de hoogste verwachte beloning geven. Als laatste is er nog het model van de omgeving, dat de agent toestaat om bepaalde assumpties te maken over zijn omgeving en hoe die zich zal gedragen.

Het doel van reinforcement learning is om voor een agent de optimale policy te vinden. Een optimale policy geeft in elke staat aan wat de beste actie is die de agent kan uitvoeren. Om tot dit punt te raken, is er eerst een formalisatie nodig voor het sequentieel beslissingsprobleem. Hiervoor zal het Markov Decision Process of MDP gebruikt worden. Een MDP beschrijft de interactie van een agent met zijn omgeving, zoals aangegeven in Figuur 2.1. De value functie zorgt ervoor dat een policy optimalizeren concreet gemaakt kan worden. Als de value functie in elke staat gemaximalizeerd is, zal de policy ook optimaal zijn. Naast de value functie is er ook nog de Q-functie of de state-action value functie. Deze functie beschrijft de verwachte beloning wanneer een agent zich in een bepaalde staat bevindt en een bepaalde actie uitvoert.

MDPs oplossen

Nogmaals, het doel van een MDP oplossen is om een optimale policy te vinden. Hiervoor bestaan er twee methodes: *model-based* of *model-free*. Model-based methodes gaan ervan uit dat het volledige model van het MDP gekend is, en worden ook vaak Dynamic Programming methodes genoemd. Model-free methodes hebben geen volledige kennis van het model, en vertrouwen op interacties met de omgeving om informatie te verkrijgen. Deze methodes worden ook Reinforcement Learning methodes genoemd. Voor deze thesis zijn de RL methodes voornamelijk van belang. Omdat interactie met de omgeving nodig is, moet er ook voor gezorgd worden dat er voldoende exploratie is, zodat er geen mogelijk voordelige opties niet verkend worden.

Temporal difference learning

De meest bruikbare toepassing van model-free methodes is temporal difference learning. Het concept waarop temporal difference learning gebaseerd is, is bootstrapping. Hierbij worden bepaalde schattingen geupdatet gebaseerd op schattingen die eerder geleerd werden. Een intuitief voorbeeld hiervan is het organizeren van een dinertje. Je informeert de gasten gebaseerd op hoelang je nodig hebt om alle inkopen te doen en het eten klaar te maken. Stel nu dat je onvoorzien langer onderweg bent, en een halfuurtje later thuiskomt, zal je ook de gasten moeten updaten dat ze een halfuurtje later moeten komen. Schattingen worden dus aangepast, gebaseerd op bevindingen onderweg. Het belangrijkste TD learning algoritme is *Q-learning*. Hierbij wordt een incrementele update regel gebruikt om Q-values voor acties te schatten, gebaseerd op beloningen en Q-values die de agent eerder ervaarde.

Policy-based RL

Algoritmes behorende tot de TD learning categorie kunnen allemaal gezien worden als *value-based* learning methodes. Deze methodes doen een poging tot het vinden van de value- of Q-functie, om zo de optimale policy af te kunnen leiden. Policy-based methodes kiezen een andere benadering, en proberen de policy direct te leren, en dus de stap te skippen die de value- of Q-functie zoekt. Deze methodes kunnen gewoonlijk opgedeeld worden in twee categorieën: *gradient-based* of *gradient-free*. Gradient-based methodes leren de policy direct door het gebruik van gradient ascent, en op deze methodes zal ook de verdere focus liggen. Een aantal prominente algoritmes in deze categorie zijn REINFORCE en actor-critic.

Deep RL

Het gebruik van neurale netwerken in reinforcement learning lost een heel aantal problemen op. Standaard RL methodes lijden onder hoge sample and computational complexity. Ook is de *curse of dimensionality* een grote boosdoener. Dit probleem omschrijft de uitdagingen die ontstaan wanneer een functie geoptimaliseerd moet worden met een hoog aantal input variabelen. Wanneer het aantal dimensies omhoog gaat, zal de moeilijkheid om de functie te optimalizeren ook sterk toenemen. Deep learning kan deze problemen grotendeels oplossen door gebruik te maken van *function approximation* en *representation learning*. Neurale netwerken zijn capabel om laag-dimensionale represantaties te leren van hoog-dimensionale data. Standaard reinforcement learning methodes worden deep reinforcement learning methodes wanneer neurale netwerken gebruikt worden om staat of observaties te vertegenwoordigen, of wanneer ze voor function approximation gebruikt worden van de value- of Q-functie. Prominente algoritmes binnen deep RL zijn DQN of deep Q network, DDPG en PPO. DQN is een adaptatie van Q-learning met neurale netwerken. Het nadeel van DQN is dat het enkel werkt voor discrete actie en staat domeinen. DDPG is applicatie van de actor-critic methode, waarbij DQN methodes gecombineerd worden met policy gradient methodes. PPO is ook een applicatie van de policy gradient methode waarbij implementatie- en gebruiksgemak fel omhoog gaan. Zowel DDPG als PPO kunnen in continue actie en staat domeinen gebruikt worden.

Multi-agent RL

Zoals reeds vermeld is simulatie van menigtes een multi-agent reinforcement learning probleem, net omdat een menigte uit meerdere personen bestaat die individueel beslissingen kunnen nemen. Een multi-agent systeem is dus een group van autonome, op elkaar inwerkene entiteiten die eenzelfde omgeving delen met elkaar. De staat en beloning van elke agent is dus ook afhankelijk van het gedrag van de anderen. Hierdoor zal ook de formalisatie van het sequentieel beslissingsprobleem aangepast moeten worden. Hiervoor worden Stochastic Markov Games ingevoerd, wat dus een generalisatie is van een multi-agent scenario van het MDP.

Taxonomie

Een typische manier om multi-agent RL methodes te organizeren, is door ze te onderscheiden op het type taak dat ze zullen uitvoeren, wat een van drie mogelijkheden kan zijn: *cooperatief*, *competitief* of *gemixt*. Voor cooperatieve settings hebben alle agenten dezelfde reward functie. Het doel is dus om de collectieve beloning zo hoog mogelijk te krijgen. Voor competitieve settings, is er één reward pool aanwezig voor al de agenten. Het verlies van beloning van een bepaalde agent zal een andere agent bevoordelen. Gemixte settings hebben geen restricties op doelen of relaties onder agenten. Hierbinnen valt crowd simulation ook, want agenten willen aan hun eindbestemming aankomen, maar gaan niet andere agenten daarvoor in de weg staan of helpen.

Training schemas

Wanneer de focus verlegt wordt op multi-agent deep RL, kunnen we drie training schema's onderscheiden: *centralized*, *concurrent* or *centralized learning with decentralized execution*. Een centralized training schema bestaat uit één gezamenlijk model dat alle acties en observaties van de agents bevat. Er wordt dus ook maar één policy getraind. Deze methode schaalt slecht naar meerdere agenten toe en is vaak niet bruikbaar voor moeilijke taken. Een concurrent training schema zal wel een policy leren voor elke agent. Dit kan voordelig zijn voor heterogene agenten, want het laat agenten toe een specifiek rol op te pakken. Een CLDE training schema maakt informatie centraal beschikbaar in het leerproces, zodat agenten sneller kunnen leren. Hier onderscheiden we twee methodes: *parameter sharing* en *MADDPG*. Parameter sharing laat alle agenten dezelfde policy leren. Hierdoor zijn ervaringen van andere agenten rechtstreeks beschikbaar voor de rest, maar zijn ze wel nog in staat om hun eigen ding te doen. MADDPG is een multi-agent extensie van DDPG waarbij een gecentraliseerde critic gebruikt wordt. De critic is hier de value-functie, en het feit dat deze centraliseerd is betekend dat de observaties en acties van andere agenten beschikbaar zijn voor elke agent. De agent kan zich dan beter afstellen op wat anderen doen tijdens het leren.

A1.3 Artificiële neurale netwerken

Fundamenteel kan een ANN gezien worden als een functie mapper, die een bepaalde input omzet naar een output. ANN's zijn in staat tot het uitvoeren van zeer parallelle berekeningen en hebben interessante eigenschappen zoals non-lineariteit, mogelijkheid tot generalisatie, foutentolerantie, ... Een ANN moet getraind worden om de juiste output te geven voor een bepaalde input, wat op verschillende manieren gedaan kan worden.

Bouwblokken en training

Een ANN bestaat uit een reeks aangesloten verwerkingseenheden die neuronen worden genoemd en die met elkaar communiceren. Deze communicate verloopt over connecties waaraan bepaalde gewichten vastgebonden zijn. Afhankelijk van de topologie van het ANN, zullen deze connecties op specifieke manieren lopen. Een ANN zal bestaan uit verschillende lagen van neuronen, met een input en output laag en daartussen verstopte lagen. Het doel van het trainen van een ANN is de gewichten van deze connecties op zo een manier aanpassen, dat correcte outputs gegeven worden voor de inputs. Om tot dit punt te raken wordt een loss functie geminimaliseerd door het gebruik van gradient descent. De gradients die hiervoor gebruikt worden, worden in het standaard geval berekend door de backpropagation methode. Om backpropagation toe te kunnen passen moet de loss functie differentieerbaar zijn t.o.v. de gewichten van het ANN. Echter, voor reinforcement learning is de omgeving niet differentieerbaar t.o.v. de actie die de agent neemt, en is dus backpropagation op de normale manier niet mogelijk. Policy gradient methodes, die gebruikt worden in deze thesis, lossen dit op door gradient estimators te gebruiken. De gradient wordt dus geschat gebaseerd op eerdere bevindingen.

Hyperparameter optimalisatie

Het succes van een ANN als non-linear function approximator is sterk afhankelijk van keuzes die op voorhand gemaakt worden, lang voor het trainen begint. Dit proces noemt met hyperparameter optimalisatie. Traditioneel zijn hier twee opties mogelijk: *parallel search* of *sequential optimization*. Parallel search zal vele parallelle processen opzetten met verschillende hyperparameter configuraties, waarbij uiteindelijk de configuratie van het best presterende proces gekozen wordt. Bij sequential optimization wordt er niets in parallel uitgevoerd, maar gebeurt alles sequentieel. Na een uitvoering kunnen er, afhankelijk van de uitkomst aanpassingen gedaan worden, en vervolgens kan het opnieuw uitgevoerd worden met deze aanpassingen. Sequentiële methodes geven vaker betere resultaten dan parallelle, maar hebben ook veel langer nodig om uit te voeren. *Population Based Training* is een methode voor reinforcement learning die trainen combineert met hyperparameter optimalisatie. Daarnast combineert het ook beide voordelen van parallel search en sequential optimization. Dit wordt gedaan door verschillende processen te starten, en ondertussen bepaalde processen hun configuraties aan te passen als bij andere processen gezien wordt dat bepaalde parameters het leren verbeteren. Hierdoor is er maar één enkele uitvoering nodig en wordt en zoals bij sequential optimization aanpassingen gedaan gebaseerd op bepaalde bevindingen.

A1.4 Simulatie van menigtes met deep multi-agent reinforcement learning

Omgeving

Voor er kan begonnen worden met trainen, moet er eerst een concrete omgeving opgesteld worden waarin agenten zich zullen bevinden. De basis ligt bij het bewegen van de agenten. Dit wordt mogelijk gemaakt door twee actie waardes: een lineare snelheid die bepaalt hoe snel de agent voor of achteruit beweegt en een hoeksnelheid, die bepaalt hoe snel de agent kan draaien. De omgeving zal vervolgens updaten met een bepaalde frequentie of tijdstap. De snelheden in combinatie met deze tijdstap bepalen vervolgens hoeveel afstand de agent heeft afgelegd, en dus kunnen de positie en oriëntatie geupdatet worden. Naast positie en oriëntatie is het ook belangrijk dat de agent een idee heeft wat er allemaal rondom hem gebeurt. De agent zal een laser scan systeem gebruiken om alles rondom hem te detecteren, de laser stopt op eerst voorwerp dat zijn pad kruist, en dan heeft de agent meteen het type voorwerp en de afstand. Voor elke actie die de agent uitvoert, moet er ook een beloning aan gebonden zijn. Hier kan een onderscheid gemaakt worden tussen goal-related beloningen en collision-related beloningen. De goal-related beloningen bestaan enerzijds uit een positieve beloning als de agent zich dichter bij het doel bevindt als bij de vorige iteratie, en negatief als dit niet zo is. Anderzijds krijgt de agent ook een positieve beloning als het uiteindelijke doel bereikt wordt. Voor collision-related beloningen is er een onderscheid tussen botsingen met normale objecten of agenten, of begrenzende objecten zoals muren. Botsingen met normale objecten of mensen tellen zwaarder door dan botsingen met begrenzende objecten.

Uitbreiden naar multi-agent

Om de uitbreiding naar multi-agent methodes uit te voeren, werden de training schema's die eerder besproken werken voor deep multi-agent RL geïmplementeerd. Er werden in het totaal vier methodes toegepast: centralized, concurrent en voor CLDE parameter sharing en centralized critic. De centralized critic methode werd zelf geïmplementeerd volgens hetzelfde principe als MADDPG, maar dan voor PPO.

Training

Vooraleer training kon beginnen waren er nog een aantal voorbereidende stappen die uitgevoerd moesten worden. DDPG en PPO zullen gebruikt worden als trainingsalgoritmes, maar deze moeten eerst en vooral afgestemd worden tot de juiste instellingen. Dit zal gedaan worden met Population Based Training. Hiernaast moeten ook een aantal omgevingsvariabelen, zoals de beloningen, ingesteld worden op bruikbare waardes. Vervolgens worden een aantal test scenario's opgesteld waarop de verschillende methodes zullen trainen. Hierna kon de training aanvatten.

A1.5 Evaluatie

Resultaten

Om de verschillende methodes met elkaar te vergelijken, wordt er een onderscheid gemaakt tussen nodige trainingstijd en het vertoonde gedrag van de individuen als de training compleet is. Wanneer er gekeken wordt naar de nodige trainingstijd, merken we voor de simpelere testscenario's op dat DDPG een klein beetje trager is dan PPO wanneer er geen parallelle data collectie gebruikt wordt. Wanneer deze parallelle data collectie ingeschakeld wordt, wat enkel voor PPO mogelijk is, zien we een vooruitgang in uitvoeringstijd. Deze tijdswinst is het grootst voor de centralized methode. In het algemeen zien we dat de concurrent, parameter sharing en centralized critic methodes ongeveer gelijk presteren, en de centralized methode blijft ver achter. Naarmate de moeilijkheid van de scenario's toenam, werd de centralized methode dan ook weggelaten uit de vergelijking. Voor de iets moeilijke scenario's zien we telkens opnieuw dat de PPO methode met parallelle data collectie voor alle training schema's veel beter presteert dan DDPG en PPO zonder parallelle data collectie. Er werd dan ook verder gegaan met enkel de PPO methode met parallelle data collectie. Voor de volgende scenario's was parameter sharing telkens de best presterende optie, met centralized critic methode net daarachter. De concurrent methode was steeds 2 tot 3x trager. Voor de moeilijkste scenario's slaagde de centralized critic methode er echter niet in om tot een eindresultaat te komen, omdat het beschikbaar geheugen op het trainingssysteem niet voldoende was.

Wanneer we het gedrag van de individuen gaan bestuderen, merken we geen verschil op tussen DDPG en PPO met of zonder parallelle data collectie. Om deze reden werd er dus verder gegaan met de PPO methode met parallelle data collectie. Bij de simpele scenario's merken we op dat voor de parameter sharing methode, individuen hetzelfde traject afleggen, wat te danken is aan de shared policy die ze beiden leren. Voor de concurrent en centralized critic methodes handelen individuen totaal onafhankelijk en zijn er geen gelijkenissen. In de bewegingspatronen van de centralized methode zijn geen recurrente patronen terug te vinden, en wordt weer weggelaten naarmate de moeilijkheid van de scenario's toeneemt. Wanneer scenario's met groepen van agenten opduiken, zien we dat voor de parameter sharing methode individuen vaak samen groepen en meer in groep blijven. Bij de concurrente en centralized critic methodes, zien we weer dat individuen hun eigen ding doen en niet zozeer in groepsverband blijven. Soms merken we ook op dat individuen bij de centralized critic methode stoppen en wachten om anderen te laten passeren, i.p.v. een omweg te zoeken en er rondom te gaan.

Conclusie

Parameter sharing in combinatie met PPO en parallelle data collectie is op het vlak van trainingstijd de beste configuratie. De parameter sharing methode vertoont meer gelijkaardig gedrag tussen individuen omdat er één enkele policy is die gebruikt wordt om alle individuen te sturen. Als het doel is om echt zelfstandig gedrag te vertonen voor elk individu, zouden de concurrent of centralized critic methodes gebruikt moeten worden. Als hoeveelheid geheugen geen probleem is, zal centralized critic de go to methode zijn, aangezien het niet veel trager is dan parameter sharing.

Appendix A2

Hyperparameters

The base of the learning parameters for all learning algorithms in RLLIB is the common config [1]. The specific configuration for PPO and DDPG is merged with this common config. The most relevant hyperparameters or the ones that had to be tuned were mentioned in Section 4.4.3, however those are not all available parameters. The rest of them is given here, along with an explanation of their roles.

A2.1 DDPG

gamma	0.99	Discount factor for the MDP
observation_filter	MeanStdFilter	Applies element-wise filter for normalization
clip_actions	True	Clips the actions to the action space low/high range
twin_q	False	This enables Twin-Delayed DDPG which is a DDPG alternative
use_state_preprocessor	True	Used to flatten weird observation spaces
actor_hiddens	[64,64]	Hidden layers of actor network
critic_hiddens	[64,64]	Hidden layers of critic network
actor_hidden_activation	relu	Activation of hidden layers of actor network
critic_hidden_activation	relu	Activation of hidden layers of critic network
exploration_should_anneal	False	Turns on annealing schedule for exploration noise. Exploration will be annealed from 1.0 to final value over a predetermined amount of timesteps

[1]https://github.com/ray-project/ray/blob/master/rllib/agents/trainer.py

exploration_noise_type	ou	Noise used is Ornstein-Uhlenbeck, another option is Gaussian
exploration_ou_noise_scale	0.1	Used to scale down magnitude of OU noise
exploration_ou_theta	0.15	Theta used for OU noise
exploration_ou_sigma	0.2	Sigma used for OU noise
pure_exploration_steps	1500	The policy of the agent will be ignored untill this amount of timesteps have been passed. The agent will take random actions, which decreases the dependence of exploration and optimization on initial policy parameters.
buffer_size	100000	Size of the replay buffer
prioritized_replay	True	Enables prioritized replay for the replay buffer
prioritized_replay_alpha	0.6	Alpha parameter for prioritized replay buffer
prioritized_replay_beta	0.4	Beta parameter for prioritized replay buffer
prioritized_replay_eps	1e-6	Epsilon to add to the TD errors when updating priorities
critic_lr	1e-3	Learning rate for critic optimizer
actor_lr	1e-4	Learning rate for actor optimizer
tau	1e-3	Used for target network updates
l2_reg	1e-6	Weights for L2 regularization
learning_starts	1500	How many steps the model needs to sample before learning starts
sample_batch_size	1	Update the replay buffer with this many samples at once
train_batch_size	16	Size of batched sample from the replay buffer used for training

A2.2 PPO

gamma	0.99	Discount factor for the MDP
observation_filter	MeanStdFilter	Applies element-wise filter for normalization
clip_actions	True	Clips the actions to the action space low/high range
use_critic	True	Enabels actor critic method for PPO

use_gae	True	Use Generalized Advantage Estimator with a value function
lambda	0.95	Lambda parameter for GAE
kl_coeff	1	Initial coefficient for KL divergence
sample_batch_size	128	Size of batches collected from each worker
train_batch_size	512	Number of timesteps collected for each round of SGD (for all workers toghether)
sgd_minibatch_size	16	Total SGD batch size (for all workers together)
shuffle_sequences	True	Shuffles sequences in the batch during training
num_sgd_iter	5	Number of SGD iterations performed on the train batch
lr	5e-5	Stepsize for SGD
lr_schedule	None	Schedule for learning rate
vf_share_layers	True	Enables layer sharing for the value function
vf_loss_coeff	0.5	Coefficient of the value function loss
entropy_coeff	0.01	Coefficient for entropy regularizer
entropy_coeff_schedule	None	Decay schedule for the entropy regularizer
clip_param	0.2	Clip param
vf_clip_param	10	Clip param for the value function
grad_clip	None	Will clip the global norm of gradients if specified
kl_target	0.01	Target value for KL divergence
batch_mode	truncate_episodes	Whether complete episode or truncated episodes should be rolled out. If complete episodes is specified, optimization steps are executed when the episode is done. For truncate episode, this is done for every train batch size collected.

Appendix A3

Bug reports

Using RLLIB as deep reinforcement learning framework was a good choice, as it is implemented well and the developers quicky respond to questions or issues. However, during this research we found a few issues with RLLIB that will be mentioned here.

- **Parameter sharing bug**: When using a parameter sharing approach, where each agent learns the same policy, together with the DDPG learning algorithm, only one of the pedestrians would move on the training visualization. I then tried with PPO, and here no problems were present. Printing the actions in the multi agent environment, displayed only a single action for the first pedestrian, instead of one action for each of them. Also, this issue only occurred in v0.8.3, I rolled back to v0.8.2 and continued to use this one.
- **MADDPG incomplete implementation**: The initial plan was to use MADDPG, as the RLLIB docs stated that the implementation was present. However, when attempting to use this implementation, some problems arose. No exploration was applied on the chosen actions, which is an essential part for reinforcement learning, and the observation and action spaces did only allow for certain discrete values. This was caused by the MADDPG implementation being direct port from MADDPG implemented for the OpenAI multiagent particle environment.

Both these bugs were reported on the issues page of Ray, the umbrella framework containing RLLIB, Tune and other libraries, see [1] [2].

[1]https://github.com/ray-project/ray/issues/7761
[2]https://github.com/ray-project/ray/issues/7069

Bibliography

[1] Charles Mackay. *Extraordinary Popular Delusions and the Madness of Crowds.* Richard Bentley, London, 1841.

[2] Craig Reynolds. Flocks, herds, and schools: A distributed behavior model. *ACM*, 21(4):25–34, 1987.

[3] Daniel Thalmann and Soraia Raupp Musse. *Crowd Simulation.* Springer-Verlag London.

[4] Suiping Zhou, Dan Chen, Wentong Cai, Linbo Luo, Malcolm Yoke Hean Low, Feng Tian, Victor Su-Han Tay, Darren Wee Sze Ong, and Benjamin D. Hamilton. Crowd modeling and simulation technologies. *ACM Trans. Model. Comput. Simul.*, 20(4), November 2010.

[5] Daniel Thalmann. *Encyclopedia of Computer Graphics and Games: Crowd Simulation.* Springer International Publishing Switzerland.

[6] Anemona Hartocollis and Yamiche Alcindor. Women's march highlights as huge crowds protest trump: 'we're not going away'. `https://www.nytimes.com/2017/01/21/us/womens-march.html`. [Online; accessed 2019-02-06].

[7] T.M. Kisko, R.L. Francis, and C.R. Nobel. Evacnet4 user's guide. `http://tomkisko.com/ise/files/evacnet/EVAC4UG.HTM`. [Online; accessed 2019-02-04].

[8] Alice Fleming. Introduction to evacuation modeling. `https://slideplayer.com/slide/5265665/`. [Online; accessed 2020-03-13].

[9] Antonio Galbis and Manuel Maestre. *Vector Analysis Versus Vector Calculus.* Springer-Verlag New York.

[10] Jim Belk. A portion of the vector field (sin y, sin x). `https://en.wikipedia.org/wiki/Vector_field#/media/File:VectorField.svg`. [Online; accessed 2020-03-13].

[11] Stephen Chenney. Flow tiles. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '04, page 233–242, Goslar, DEU, 2004. Eurographics Association.

[12] Roger L. Hughes. The flow of human crowds. *Annual Review of Fluid Mechanics*, 35(1):169–182, 2003.

[13] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. *ACM Trans. Graph.*, 25(3):1160–1168, July 2006.

[14] Dirk Helbing, Illés Farkas, and Tamás Vicsek. Simulating dynamical features of escape panic. *Nature*, 407(6803):487–490, Sep 2000.

[15] C. Burstedde, K. Klauck, A. Schadschneider, and J. Zittartz. Simulation of pedestrian dynamics using a two-dimensional cellular automaton. *Physica A: Statistical Mechanics and its Applications*, 295:507–525, 2001.

[16] C. Reinolds. Steering behaviors for autonomous characters. 1999.

[17] Dirk Helbing and Peter Molnar. Social force model for pedestrian dynamics. *Physical Review E*, 51, 05 1998.

[18] S. R. Musse and D. Thalmann. Hierarchical model for real time simulation of virtual human crowds. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):152–164, 2001.

[19] Raghda Alqurashi and Tom Altman. Hierarchical agent-based modeling for improved traffic routing. *Applied Sciences*, 9:4376, 10 2019.

[20] David Wolinski. *Microscopic crowd simulation : evaluation anddevelopment of algorithms.* PhD thesis, Université Rennes, 2016.

[21] Shawn Singh, Mubbasir Kapadia, Petros Faloutsos, and Glenn Reinman. SteerBench: a benchmark suite for evaluating steering behaviors. *Computer Animation and Virtual Worlds*, 9999(9999):n/a+, 2009.

[22] Martijn van Otterlo and Marco Wiering. *Reinforcement Learning and Markov Decision Processes*, volume 12. Springer International Publishing Switzerland.

[23] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey, 1996.

[24] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge Massachusetts.

[25] Richard Bellman. *Dynamic Programming.* Dover Publications.

[26] *Dynamic Programming and Markov Processes.* The MIT Press.

[27] Dariusz Sankowski. Exploration in reinforcement learning. `https://towardsdatascience.com/exploration-in-reinforcement-learning-e59ec7eeaa75`. [Online; accessed 2020-03-17].

[28] Sebastian Thrun. The role of exploration in learning control. In *Handbook for Intelligent Control: Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Reinhold, Florence, Kentucky, January 1992.

[29] Marco Wiering. Explorations in efficient reinforcement learning. 01 1999.

[30] Christopher Watkins. Learning from delayed rewards. 01 1989.

[31] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine learning*, 8, pages =, 1992.

[32] J. Zico Kolter. Introduction to reinforcement learning. `http://icaps18.icaps-conference.org/fileadmin/alg/conferences/icaps18/summerschool/lectures/Lecture5-rl-intro.pdf`. [Online; accessed 2020-03-23].

[33] Lillian Weng. A (long) peek into reinforcement learning. `https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html#key-concepts`. [Online; accessed 2020-03-23].

[34] Marc Deisenroth, Gerhard Neumann, and Jan Peters. *A Survey on Policy Search for Robotics*, volume 2. 08 2013.

[35] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992.

[36] David Silver. Lecture 7: Policy gradient. `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf`. [Online; accessed 2020-03-23].

[37] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, Nov 2017.

[38] Yuxi Li. Deep reinforcement learning, 2018.

[39] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[40] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[41] 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, 2015.

[42] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page I–387–I–395. JMLR.org, 2014.

[43] OpenAI: Spinning Up. Deep deterministic policy gradient. `https://spinningup.openai.com/en/latest/algorithms/ddpg.html#the-q-learning-side-of-ddpg`. [Online; accessed 2020-03-28].

[44] OpenAI. Proximal policy optimization. `https://openai.com/blog/openai-baselines-ppo/`. [Online; accessed 2020-03-28].

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[46] OpenAI: Spinning Up. Proximal policy optimization. `https://spinningup.openai.com/en/latest/algorithms/ppo.html`. [Online; accessed 2020-03-29].

[47] Alex Irpan. Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`, 2018. [Online; accessed 2020-04-01].

[48] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms, 2019.

[49] Lucian Busoniu, Robert Babuska, and Bart De Schutter. *Multi-agent Reinforcement Learning: An Overview*, volume 310, pages 183–221. 07 2010.

[50] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 38:156 – 172, 04 2008.

[51] Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications. *IEEE Transactions on Cybernetics*, page 1–14, 2020.

[52] Georgios Papoudakis, Filippos Christianos, Arrasy Rahman, and Stefano V. Albrecht. Dealing with non-stationarity in multi-agent deep reinforcement learning, 2019.

[53] Jayesh Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. pages 66–83, 11 2017.

[54] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments, 2017.

[55] Kris Cao, Angeliki Lazaridou, Marc Lanctot, Joel Z Leibo, Karl Tuyls, and Stephen Clark. Emergent communication through negotiation, 2018.

[56] Angeliki Lazaridou, Karl Moritz Hermann, Karl Tuyls, and Stephen Clark. Emergence of linguistic communication from referential games with symbolic and pixel input, 2018.

[57] Georgios Boutsioukis, Ioannis Partalas, and Ioannis Vlahavas. Transfer learning in multi-agent reinforcement learning domains. *EWRL: Recent Advances in Reinforcement Learning*, 2011.

[58] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41–48, New York, NY, USA, 2009. Association for Computing Machinery.

[59] Qian Long, Zihan Zhou, Abhibav Gupta, Fei Fang, Yi Wu, and Xiaolong Wang. Evolutionary population curriculum for scaling multi-agent reinforcement learning, 2020.

[60] Kevin L. Priddy and Paul E. Keller. *Artificial Neural Networks: An Introduction.* SPIE Press.

[61] Imad Basheer and M.N. Hajmeer. Artificial neural networks: Fundamentals, computing, design, and application. *Journal of microbiological methods*, 43:3–31, 01 2001.

[62] Ben Kröse, B. Krose, Patrick van der Smagt, and Patrick Smagt. An introduction to neural networks. *J Comput Sci*, 48, 01 1993.

[63] missinglink.ai. 7 types of neural network activation functions: How to choose? `https://missinglink.ai/guides/neural-network-concepts/7-`

`types-neural-network-activation-functions-right/`. [Online; accessed 2020-05-09].

[64] Dana Hughes and Nikolaus Correll. Distributed machine learning in materials that couple sensing, actuation, computation and communication, 2016.

[65] Andrej Krenker, Janez Bešter, and Andrej Kos. *Artificial Neural Networks Methodological Advances and Biomedical Applications*, chapter Introduction to the Artificial Neural Networks. InTechOpen, 2011.

[66] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[67] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. 2017.

[68] Will Koehrsen. A conceptual explanation of bayesian hyperparameter optimization for machine learning. `https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f`. [Online; accessed 2020-05-15].

[69] Matthias Feurer and Frank Hutter. *Automated Machine Learning*. springer, 2019.

[70] F. Schmidt. Datasets for tracking people in aerial image sequences. `http://www.ipf.kit.edu/downloads_People_Tracking.php`. [Online; accessed 2020-03-08].

[71] Xiao Song, Daolin Han, Jinghan Sun, and Zenghui Zhang. A data-driven neural network approach to simulate pedestrian movement. *Physica A: Statistical Mechanics and its Applications*, 509, 06 2018.

[72] M. Zhao, Stephen Turner, and Wentong Cai. A data-driven crowd simulation model based on clustering and classification. pages 125–134, 10 2013.

[73] Luiselena Casadiego and Nuria Pelechano. From one to many: Simulating groups of agents with reinforcement learning controllers. pages 119–123, 08 2015.

[74] P. Henry, C. Vollmer, B. Ferris, and D. Fox. Learning to navigate through crowded environments. In *2010 IEEE International Conference on Robotics and Automation*, pages 981–986, 2010.

[75] Gabriel de la Cruz, Bei Peng, Walter Lasecki, and Matthew Taylor. Towards integrating real-time crowd advice with reinforcement learning. 03 2015.

[76] Jaedong Lee, Jungdam Won, and Jehee Lee. Crowd simulation by deep reinforcement learning. In *Proceedings of the 11th Annual International Conference on Motion, Interaction, and Games*, MIG '18, New York, NY, USA, 2018. Association for Computing Machinery.

[77] Jonas Kulhanek, Erik Derner, Tim de Bruin, and Robert Babuska. Vision-based navigation using deep reinforcement learning. *2019 European Conference on Mobile Robots (ECMR)*, Sep 2019.

[78] A. Bieszczad. Exploring machine learning techniques for identification of cues for robot navigation with a lidar scanner. In *2015 12th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, volume 01, pages 645–652, 2015.

[79] Lisa Torrey. Crowd simulation via multi-agent reinforcement learning. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'10, page 89–94. AAAI Press, 2010.

[80] Sheng Yan Lim. Crowd behavioural simulation via multi-agent reinforcement learning. Master's thesis, University Of The Witwatersrand Johannesburg, 8 2015.

[81] Niccolò Bisagno and Andrea Montagner. Ppol - a machine learning approach to crowd modeling. `https://connect.unity.com/p/ppol-a-machine-learning-approach-to-crowd-modeling`. [Online; accessed 2020-04-23].

[82] Michael Everett, Yu Fan Chen, and Jonathan P. How. Motion planning among dynamic, decision-making agents with deep reinforcement learning, 2018.

[83] Yu Fan Chen, Miao Liu, Michael Everett, and Jonathan P. How. Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning, 2016.

[84] Tingxiang Fan, Pinxin Long, Wenxi Liu, and Jia Pan. Fully distributed multi-robot collision avoidance via deep reinforcement learning for safe and efficient navigation in complex scenarios, 2018.

[85] Pinxin Long, Tingxiang Fan, Xinyi Liao, Wenxi Liu, Hao Zhang, and Jia Pan. Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning, 2017.

[86] Hyung-Jin Yoon, Huaiyu Chen, Kehan Long, Heling Zhang, Aditya Gahlawat, Donghwan Lee, and Naira Hovakimyan. Learning to communicate: A machine learning framework for heterogeneous multi-agent robotic systems, 2018.

[87] S. Zheng and H. Liu. Improved multi-agent deep deterministic policy gradient for path planning-based crowd simulation. *IEEE Access*, 7:147755–147770, 2019.

[88] Anthony Francis, Aleksandra Faust, Hao-Tien Lewis Chiang, Jasmine Hsu, J. Chase Kew, Marek Fiser, and Tsang-Wei Edward Lee. Long-range indoor navigation with prm-rl, 2019.

[89] Dongmin Lee. Presentation prm-rl: Long-range robotics navigation tasks by combining reinforcement learning and sampling-based planning. `https://www.slideshare.net/DongMinLee32/prmrl-longrange-robotics-navigation-tasks-by-combining-reinforcement-learning-and-samplingbased-planning-200684730`. [Online; accessed 2020-05-04].

[90] OpenAI. Gym. `http://gym.openai.com/`. [Online; accessed 2020-03-09].

[91] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing - solving sparse reward tasks from scratch, 2018.

[92] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. Ray RLlib: A composable and scalable reinforcement learning library. In *Deep Reinforcement Learning symposium (DeepRL @ NIPS)*, 2017.

[93] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training, 2018.

[94] AurelianTactics. Ppo hyperparameters and ranges. `https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe`. [Online; accessed 2020-04-23].

[95] Siraj Raval. Best practices when training with ppo. `https://github.com/llSourcell/Unity_ML_Agents/blob/master/docs/best-practices-ppo.md`. [Online; accessed 2020-04-24].