Robotic Navigation in Simulated Urban Environments

An investigation on the effect of uncertainty in the observed environment

This dissertation is submitted in part requirement for the Master of Science in Spatial Data Science and Visualisation at the Centre of Advanced Spatial Analysis, Bartlett Faculty of the Built Environment, University College London.

Candidate: Kristian Emil Lunow Nielsen

Date: 30/08/2019

MSc Spatial Data Science and Visualisation, TMSSDSAVIS01

Supervisor: Ed Manley

Word count:

# Abstract

# Declaration

I, Kristian Emil Lunow Nielsen, hereby declare that this dissertation is all my own original work and that all sources have been acknowledged. This dissertation is xxx words in length, from introduction to conclusion inclusive, excluding footnotes. Word count by Word.

Date: 30/08/2019

Kristian Email Lunow Nielsen

Table of Contents

[Abstract 2](#_Toc15890572)

[Declaration 3](#_Toc15890573)

[List of tables 5](#_Toc15890574)

[List of figures 6](#_Toc15890575)

[List of acronyms and abbreviations 7](#_Toc15890576)

[Acknowledgements 8](#_Toc15890577)

[1. Introduction 9](#_Toc15890578)

[1.1 Theoretical background 10](#_Toc15890579)

[1.2 Purpose of the Study 13](#_Toc15890580)

[1.3 Significance of the Study 13](#_Toc15890581)

[1.4 Scope of the Study 14](#_Toc15890582)

[1.5 Literature Review 15](#_Toc15890583)

[1.5.1 Reinforcement learning 15](#_Toc15890584)

[1.5.1.1 Deep reinforcement learning 20](#_Toc15890585)

[1.5.1.1.1 Policy Gradient Methods 20](#_Toc15890586)

[1.5.1.1.2 Proximal Policy Optimisation 22](#_Toc15890587)

[1.5.2 Robotic Navigation in Urban Environments 23](#_Toc15890590)

[1.5.2.1.1 Robotic Navigation in Urban Environments using Reinforcement Learning 25](#_Toc15890591)

[2. Prerequisites 26](#_Toc15890592)

[2.5 Notation 26](#_Toc15890593)

[2.6 Autonomous delivery robots today 26](#_Toc15890594)

[2.7 Unity – as a simulation engine for research in DRL 27](#_Toc15890595)

[2.3.1 The environment 29](#_Toc15890596)

[2.3.1.1 Environment 31](#_Toc15890597)

[2.3.1.1.1 Walls & Obstacles 33](#_Toc15890598)

[2.3.1.1.2 Pedestrians 33](#_Toc15890599)

[2.3.1.1.3 Crowded areas 37](#_Toc15890600)

[2.3.1.2 Academy 40](#_Toc15890601)

[2.3.1.3 Agent 44](#_Toc15890602)

[2.3.1.4 Target 48](#_Toc15890603)

[2.3.2 Training 50](#_Toc15890604)

[2.3.3 TensorBoard 50](#_Toc15890605)

[2.4 Curriculum Learning 51](#_Toc15890606)

[3. Exploration/Exploitation trade-off 51](#_Toc15890607)

[Insights: 55](#_Toc15890608)

[Stuff for the discussion: 55](#_Toc15890609)

[Thoughts: 55](#_Toc15890610)

[4. Policy Evaluation 55](#_Toc15890611)

[5. Appendix 55](#_Toc15890612)

[5.3.1.1 Tuning of hyperparameters 55](#_Toc15890613)

[6. References 56](#_Toc15890614)

# List of tables

# List of figures

# List of acronyms and abbreviations

# Acknowledgements

# Introduction

The topic for this dissertation is robotic navigation in simulated urban environments, and the purpose of the study is to explore the use of deep reinforcement learning (DRL) in this context. It is an area with growing attention within urban analytics, as urban infrastructure is being transformed by the advances in artificial intelligence and robotics.

Urban environments are complex dynamics of interactions between objects, and navigation herein requires an ability to explore, foresee, adapt and plan.   
Successful prior work on robotic navigation in crowded urban environments (CUE’s) rely on the use of particle filters, for building a probabilistic map of the environment to handle planning, and a combination of human interaction as well as goal-directed exploration for exploration and adaption of/to the environment (Lidoris et al., 2009; Kümmerle et al, 2013). The need for human interaction limits the autonomous degree of the robot, potentially limiting the usages of the robot to a certain time period doing the day.   
Recent advances in the field of deep learning, implying the rise of DRL, could present a way for a robot to improve spatial awareness. Thereby circumventing the need for human interaction and allowing the robot to operate at any time. The first work on DRL for robotic navigation and obstacles avoidance have seen the daylight (Mirowski et al, 2018; Kahn et al., 2017; Zhou et al., 2019). All three are bound to tackle static obstacles, and so do not address a major challenge of CUE’s, namely dynamic obstacles.

Therefore, the motivation and context of this study is to obtain insights on how a model-free approach with continuous actions and partial observability, tackles the challenges navigating a crowded environment with static and dynamic obstacles. The study sheds light on two additional aspects; the effect of uncertainty about the observed environment, and how different training strategies can aid the learning process.

## Theoretical background

The fundamental aspects of robotic navigation can be boiled down to learning and planning, which is also the fundamentals of reinforcement learning (RL) through the tasks of control and prediction. Control tasks are concerned with learning the best policy, while prediction tasks evaluate the policy at hand.

Traditional RL literature distinguish between *model-based* and *model-free* algorithms, where model-based algorithms rely on planning and model-free algorithms rely on learning (Sutton and Barto, 2018).

This study is mainly a control task, on obtaining an optimal policy for locating a target while avoiding obstacles, and not about planning the optimal route to the target given a policy.

The RL algorithm used in this study is called *Proximal Policy Optimization* (PPO) (Schulman et al, 2017a; Schulman et al, 2017b), which belongs to class of *policy gradient methods.* This class of methods essentially extends traditional *model-free* control algorithms, as *Q-learning* (Watkins and Dayan, 1992), into large scale real-world applications.

* + 1. The Reinforcement Learning Problem

The traditional RL set-up consists of an agent (the RL system) and environment in which it operates, see figure 1. Every RL problem is about *solving* the Markov Decision Process (MDP), in the sense of optimizing some objective function given the MDP, as the MDP fully characterise problem. This objective function can either be a *policy* function, *value* functionor the *advantage* function (difference between the policy and value function), depending of the problem and algorithm.

The decision process is a *Markov* decision process because all history of the environment is captured in the most recent value;

*Figure 1 – Markov Decision Process*A screenshot of a cell phone

Description automatically generated *Credit: (Sutton and Barto, 2018)*

Any MDP is made up by a set of actions, , a set of states,, a set of rewards, , and sometimes explicitly, yet rarely in practice, a set of state transition probabilities. The latter won’t be elaborated any further, as they are only relevant for smaller finite[[1]](#footnote-1) MDP’s, which occurs rarely in practice, at least in any interesting applications.

The order in figure 1: The agent observes the initial environment and takes an action at time *t*, transitioning the environment to state *t +1* and emitting a reward to agent. Based on the new observed state and the reward obtained, the agent chooses a new action, and this cycle repeat until the terminal state. An important thing to mention is that the reward does not say directly how good the action was, but how the good the resulting state is.  
The reward signal is often sparse, meaning the agent only receives a reward at the terminal state, i.e. at the end of an *episode*. If the terminal state is good, the agent receives a large positive reward, if bad, a large negative reward, and zero in all states[[2]](#footnote-2) leading up to the terminal state.

## Purpose of the Study

This research aims at addressing the challenges that are surrounding the application of DRL for navigation and obstacle avoidance tasks in CUE’s. The implication hereof is that the conducted research is mainly methodological. This research should be regarded as a preliminary study, to be further extended to generalise to the real world, because of the complex nature of urban environments, and especially crowded areas. This study outlines the basis for filling the gap on DRL for dynamic obstacle avoidance, as this to the knowledge of the author at the time of writing, is yet to be explored in depth. This study was at the same time an opportunity to explore Unity and the toolkit ML-Agents by (Juliani et al., 2018) for conducting DRL research, under realistic physical settings.

The intention of this study is not to promote DRL to replace existing methods in traditional robotic navigation, but hopefully to aid these, to achieve *smarter* and *safer* cities in the future.

The first objective of this study is to address the challenges emerging when using DRL for navigation and obstacles avoidance in dynamic environments.   
The second objective of the study is to address ways to tackle these challenges and promote meaningful learning in the agent.   
The final objective is to address how uncertainty around the observed environment affects the learning taking place.

## Significance of the Study

This study intends to contribute to the continuing development of DRL for robotic navigation in urban environments, by addressing some of the challenges still present and test novel design methods of the training phase. Addressing the ongoing challenges, hopefully enables focused future research (Irpan, 2018), avoiding rediscovering of known results.  
Most prior research in this area has been concerned with developing novel methods to tackle the challenges at hand, neglecting the design of the training phase.  
Two ways to design the training phase are *curriculum* and *imitation* learning, where the latter is widely adopted in prior research on robotic navigation (Zhou et al, 2019; Kahn et al., 2017). The former appears overlooked in the context of robotic navigation, especially in the specific context of robotic navigation in CUE’s using DRL.

The presented study utilises curriculum learning in the training phase, and thereby contribute by examining how the use of curriculum learning can help to address some of the challenges of robotic navigation in CUE’s using DRL.

## Scope of the Study

The study explores state-of-the-art methods and serves as a framework to evaluate the current state of the field of DRL for robotic navigation in CUE’s through simulation. Simulation is done under realistic physical settings, intending to smoot the future generalisation to real-world applications. The aim is that even though the study is limited to simulation, it could serve as a baseline for future real-world applications.

The investigated DRL algorithm is based on what is available in the ML-Agents toolkit, and this algorithm is the state-of-the-art for continuous control tasks (Schulman et al., 2017).

The literature review is not intended to be complete on RL/DRL nor robotic navigation. It serves to present the current state of the field, by the most important and latest contributions relevant for robotic navigation in CUE’s using DRL, in order to address the objectives of the study. Furthermore, fundamental concepts in RL, as well as key innovations leading to DRL, and the class of the method used in this study are outlined. That limits the literature review to viewing some concepts of model-free control, see (Sutton and Barto, 2018) for an extensive coverage of RL.

## Literature Review

The following proceeds by first reviewing RL, the emergence of DRL and *the[[3]](#footnote-3)* class of DRL algorithms, before reviewing the literature on robotic navigation in urban environments and the use of RL as well as DRL in this context. The order of the review is to provide some base knowledge of the chosen method, before reviewing the field of application surrounding this study.

### Reinforcement learning

Perhaps the most important discovery within RL is *temporal-difference learning* (TD), originating from animal learning psychology. TD was originally acknowledged in RL context by (Minsky, 1954; Samuel, 1959), and proposed in known format today by (Sutton, 1984; Anderson, 1986). TD methods leverage on both *dynamic programming* (DP) and *Monte Carlo methods[[4]](#footnote-4)* (MC), by using bootstrapping as DP, making TD online, and sampling as MC, making them model-free. Different TD methods for control tasks exists, some being *SARSA*, *Q-learning* and *Expected SARSA*, and they differ by the way they handle the estimate of the objective function in future states.   
The following focuses on Q-learning, see (Sutton and Barto, 2018) for a review of the other two.

Q-learning is an *off-policy* learning method, which mean that the actual action is drawn from a *behaviour* policy, , which is compared to an alternative successor action, drawn from the target policy, . Off-policy learning methods are useful because they allow one to re-use experience from old policies, as we will see is useful, and to learn about the optimal policy while following an exploratory policy.   
Given the two actions, , the online update of the state action-value function is done according to (1).

Both policies are improved, with the target being updated greedily[[5]](#footnote-5) and the behaviour -greedy, both w.r.t. , which simplifies the target as seen from (2).

The substituted expression is just the current estimate of the optimal future value.

*Figure 2 – Q-learning algorithm*  
A screenshot of a cell phone

Description automatically generated  
*Credit: (Sutton and Barto, 2018)*

Q-learning converges to the optimal policy and action-value function, with probability 1, under the assumption that all states are visited infinite number of times[[6]](#footnote-6), illustrated in (3).

The Q-learning algorithm showed in figure 2 can seem too simple to work on real-life large-scale control problems. However, it illustrates the high-level idea, about improve an estimate in online fashion – which is essential before covering the policy gradient methods.   
Q-learning is in practical purposes couple with some minor tricks, as *batch* updates, *eligibility* *traces* and *function approximation* for more efficient scalable learning. Batch updates will be explained in a section to come, but the interested reader should consult (Sutton and Barto, 2018: Chapter 12) for a description of eligibility traces.

Doing convergence of (3), the function on the left-side is a so-called *approximate* state action-value function, which is the simple case is look-up table with size . In real-life applications, such a table is often impossible to store in memory. To ensure scalability, function approximations are used, implying the look-up table is replaced with a parameterised function, which could be a linear combination of features, a neural network or something third. The full description of function approximations are covered in (Sutton and Barto, 2018; Chapter 9,10 & 11), but for now it should be noted that the approximate state action-value function changes from to , and updating the weights in the correct direction improves the approximate state action-value function in the desired direction.

Before moving on to policy gradient methods, which are mostly used in the industry today (Karpathy, 2016), it is worth to look at what drove the transition to *deep* RL.

Today’s choice of function approximators are most often neural networks (nets), and the *deep* part of DRL referrers to the structure of the net. A deep net consists of many layers, enabling learning of complex high-dimensional non-linear functions, as each layer learns different aspects of the data passed through (Lecun et al., 2015). *Deep learning*, which is the high-level designation for variations of deep nets, belongs to the class of general-purpose learning procedures (Lecun et al., 2015). General-purpose learning procedures can learn good feature representations directly from the data, avoiding the need for hand-crafted, often non-generalisable, features and at the same time managing the *selectivity-invariance dilemma* (SID). SID in feature engineering is the ability of features to produces representation that are selective to aspects of the image that are essential for discrimination, but that are invariant to irrelevant aspects such as the pose of the animal (Lecun et al., 2015).

Some of the most important advances in deep learning, leading to DRL, is properly;

* The adaption of the *rectified linear unit* (ReLU) activation function as the standard.

Activation functions are used to squeeze the output of neurons to a bounded range, typically . ReLU, , bounds the outcome to , which has shown to provide much faster training of deep nets (Lecun et al, 2015; Nair and Hinton, 2010; Krizhevsky et al., 2012).

* Increased GPU power and the possibility of parallelised GPU training.

More powerful GPU’s allows increasing storage of data in memory, and parallel GPU implementations of deep nets allow for faster training and handling of larger amounts of data than ever before (Krizhevsky et al, 2012; Lecun et al., 2015).

* The raise of convolutional nets (ConvNet).

ConvNets are build with an eye for processing data in the form of multiple arrays, and the typical architecture[[7]](#footnote-7) consists of local connections, shared weights, pooling and many layers (Lecun et al., 2015).

ConvNets has been shown to training faster and exhibit greater generalisation ability than stacked nets of fully connected layers (Lecun et al, 2015). The trend is to combine different types of nets in the final deep net, as seen in (Krizhevsky et al., 2012; Sermanet et al., 2013; Mnih et al., 2013).

These advances along with the use of *experience replay[[8]](#footnote-8)* (Lin, 1992), made the difference for the first successful deployment of a DRL model, to learn a control policy directly from high-dimensional sensory input (Mnih et al., 2013). The agent of (Mnih et al., 2013) learned to play 7 ATARI games, with no adjustment between the games, and surpassed previous implementations on six of the games while obtained above human-expert level on three of them. The work of (Mnih et al., 2013) motivated two other important papers, (Mnih et al., 2015) and (Silver et al., 2016).  
(Mnih et al., 2015) extend the work of (Mnih et al., 2013) to 49 ATARI games, beating all previous implementations, obtaining the level of a profession human game-tester across all 49 games and achieving above human performance on 23 games (Mnih et al., 2015).  
(Silver et al, 2016), deploying a combination of deep nets and a tree search algorithm, obtained master[[9]](#footnote-9) level in the boardgame GO, which was regarded as one of the grand challenges, because of its enormous state space consisting of possible moves.

Having reviewed a traditional RL method, as well as the key innovations leading to DRL, including a few key applications, it is now time to review a class of DRL methods. This class is called *policy gradient (PG) methods*, and the method used in this study, *PPO*, belongs to this class.

#### Deep reinforcement learning

This section starts out with a simplified example from (Karpathy, 2016), which is most definitely worth a read, because the example gives a good intuitive idea about how PG methods work. After this example follows a review of the method used in this study.

##### Policy Gradient Methods

True PG methods seeks to learn a parameterised policy function to select actions of the agent, differentiating them a bit from action-value methods like Q-learning, where a value function is learned as well. However, some PG methods approximate a value function as well, to aid the learning process, as seen later, and these methods are referred to as *Actor-Critic* methods.  
The objective of most PG methods is to learn the policy parameters based on the gradient[[10]](#footnote-10) of some scalar performance measure, with respect to the policy parameters (Sutton and Barto, 2018).

The basic idea of PG methods is to adjust the parameters of the approximated policy function, most often the weights in a net, according to the scalar reward signal with respect to the parameters of the environment . How must to adjust, and so how to update the policy, is determined by the gradient of the scalar signal[[11]](#footnote-11), seen from (3).

With being the learning rate and the gradient of the reward function.

The task in (Karpathy, 2016) is to learn an agent to play the ATARI game of Pong, from nothing more than the pixels from the emulator[[12]](#footnote-12), using a basic PG.

The structure is as follows; we receive an image, of size 210x160x3[[13]](#footnote-13), and get to decide whether to move up or down. After every action, the agent is rewarded; +1 if the ball went past the opponent, -1 if the ball went past us and 0 otherwise. The objective is to beat the opponent, and so maximising the reward.

The architecture of the policy network is a 2-layer net, taking in the state of the game (100,800 pixels) and outputting a stochastic policy (a probability of moving the pad up or down). The actual action is sampled from the stochastic policy.  
Coming of the example net shown in (Karpathy, 2016) and in figure 3, the network consists of over 2 million parameters[[14]](#footnote-14) to be adjusted after each action.

*Figure 3 – The Policy Network  
A close up of a logo

Description automatically generated  
Credit: (Karpathy, 2016)*

It sounds impossible that it is possible to encourages the correct behaviour, with the need for adjusting +2.5 million individual parameters, and a sparse reward signal, which often comes many steps after the action has been taken. But that is the beauty of nets and training using backpropagation[[15]](#footnote-15). Say an episode consists of 200 steps, implying 200 actions to be taken. If just 101 of the actions are *good* actions, the outcome will be a reward of +1. Overtime, as we adjust the weights in favour of the good actions, the share of good actions within one episode increases, implying that the total number of episodes leading to a positive reward increase. For a more thorough description, see (Karpathy, 2016) but this hopefully give an intuitive idea about what’s going on.  
This example is rounded of with a quote from Karpathy himself, which sort of underlines the beauty of the above.

*“That’s the beauty of neural nets; Using them can feel like cheating: You’re allowed to have 1 million parameters embedded in 1 teraflop of compute and you can make it do arbitrary things with SGD. It shouldn’t work, but amusingly we live in a universe where it does.”* (Karpathy, 2016)

##### Proximal Policy Optimisation

PPO is the embedded DRL algorithm in the ML-Agents toolkit, and it has been shown to outperform far more complex PG methods while being more general and having better sampling complexity (Schulman et al., 2017b). The idea of PPO is like the example of (Karpathy, 2016) and somewhat to the idea of (Mnih et al., 2013).

PPO optimises a surrogate objective function, based on sampled experience (batch updates), from which the policy is updated, and an action is chosen from the updated policy. Simple and effective.  
The PPO proposed by (Schulman et al., 2017b) and used in the ML-Agents toolkit, is an Actor-Critic styled PG method, and both the approximative policy and value function are approximated by deep nets. For the full argumentation for using an actor-critic styled PG method, see (Schulman et al., 2016), but the simplified argument is to manage the bias-variance trade-off, occurring from the use of stochastic policies in PG methods.  
It is choice of surrogate objective function that differs PPO from other PG methods, with the motivation for a new surrogate objective function being that, existing surrogate objective functions tends to suffer from catastrophic large updates when multiple steps of optimisation is been performed using the same trajectory. The proposed surrogate objective function for PPO bounds the size of the update, eliminating the risk of large updates. The surrogate objective function is seen in (4) and the algorithm is shown in figure 4. Review (Shulman et al., 2017b) for a thorough description of PPO and the difference to other PG methods.

*Figure 4 – PPO Algorithm  
A screenshot of a cell phone

Description automatically generated*

With the theory outlined, it is now time to review the field of application for this study.



### Robotic Navigation in Urban Environments

Besides limiting this part of the literature review to only consider robotic navigation in urban environments (UE), it is also restricted to only consider navigation in unknown environments as it resembles to the model-free DRL approach studied in this study.  
This part of the literature review is divided into two parts; the traditional literature, which does not include the use of RL, and the DRL-based literature.

Robotic navigation in UE’s obviously comes with a lot of challenges. The four main challenges surrounding robotic navigation in UE’s are mapping, localisation, traversability analysis and planning. Common for all the papers examined on the traditional literature, is that each of the four challenges are handled by separate systems, within the robot.

The consensus to handle mapping appears to be using particle filters, often using a SLAM[[16]](#footnote-16) module (Lidoris et al., 2009; Kümmerle et al., 2013; Maria Bauer et al., 2008; Trulls et al., 2011; Georgiev and Allen, 2004). This approach implies computing an occupancy grid; however, it can be troublesome in terms of memory to store large scale occupancy grid, which is something to consider design-wise.

Localisation is done in different ways; (Maria Bauer et al., 2008; Lidoris et al., 2009; Kümmerle et al., 2013) uses sampled based methods, as Monte Carlo Localisation. (Georgiev and Allen, 2004) uses a combination of GPS coordinates, odometry and visual image processing for localisation and (Trulls et al., 2011) uses an online particle filter implementation based on a 3D geometric model of the environment.

Traversability analysis is usually done using a combination of horizontal and vertical lasers, to measure distance to objects, and potentially changing positions (needed to locate dynamic obstacles as pedestrians) (Trulls et al., 2011; Lidoris et al., 2009; Kümmerle et al., 2013; Maria Bauer et al., 2008).

Planning is done differently; (Lidoris et al., 2009; Maria Bauer et al., 2009;Trulls et al., 2011) use the A\* algorithm to calculate the shortest path through the crowded area. (Kümmerle et al., 2013) uses a hierarchical set-up, which consists of three planners, a high-level agent planning the overall go-to-route, an intermediate-level agent which calculates waypoints and a low-level agent calculating the velocity of the robot based on the waypoints.

The take-aways from the presented literature review is that robotic navigation is a field which has been subject for much research. Some challenges are handled consensually while others are subject for experimentation of novel methods.

#### Robotic Navigation in Urban Environments using Reinforcement Learning

The literature on the use of DRL in the specific field of robotic navigation in UE’s is currently sparse, but it is an area with growing interest from the research community (Zhou et al., 2019). The implication is that no papers exists, at least to the knowledge of the author, which handles all four challenges, mentioned in the previous section, directly.

One paper from DeepMind by (Mirowski et al., 2018) presents an agent, which can navigate through a simulated city based in visual input – and transfer its knowledge to other cities. However, their agent operates with discrete actions and the environment possesses only static obstacles. Their agent uses two interesting additions to the traditional DRL architecture; A method named *Long Short-Term Memory* (Hochreiter and Schmidhuber, 1997) to incorporate memory in the agent, and the use of *Curriculum Learning* (CL), enabling the agent to be introduced to increasing complexity of the environment doing training.

Another paper from OpenAI by (Kahn et al, 2017) focuses on obstacle avoidance using DRL, and they incorporate uncertainty-awareness in the agent, for safer navigation. The short comings of this article, compared to the traditional literature, is that they only consider static obstacles. Furthermore, they use a model-based DRL model.

The final paper considered here, is a paper by (Zhou et al., 2019) which uses a hierarchical agent to perform goal-directed navigation while being adaptive to changes in the environment. The high-level agent handles the goal orientation and the low-level agent takes care of changes in the environment. They train in a simulated environment and show that the agent generalises to the real world.

All the papers deliver promising insights, but two of them are especially interesting, in terms of this study; (Mirowski et al., 2018) uses CL successfully for navigation, and (Zhou et al., 2019) trains an agent, the low-level one, to be able to deal with changes in the environment, which also is able to transfer to the real world.

# Prerequisites

## Notation

Words used interchangeably: Sensors/Pedestrians, Robot/Agent

Episodes, steps, lessons

## Autonomous delivery robots today

Delivery robots are not a thing of the future, they are already deployed in a few pilot cities, even in London (Nichols, 2019). An increasing number of companies are putting they attention on autonomous delivery robots, to meet increasing customer expectations of companies to ride the technology weaves, to enable low-cost-low-emission products. 2019 has so far been an exciting year in this matter, with three of the biggest players (Merrit, 2019), within the field of ADR’s, launching different initiatives taking effect doing 2019.  
Starship, founded by two of the Skype co-founders, newest launch is autonomous delivery of food and beverages at George Mason University, Maryland. The partnership is to accommodate the rising need for smart solutions in a high-paced-high-expectation environment, where nutrition sometimes is overlooked (Nichols, 2019). Starship fleet of ADR’s has over 100.000 logged kilometres and more than 25.000 deliveries under the wheels, and been deployed in cities like London, New York and Washington, DC (Merrit, 2019; Nichols, 2019). Starship’s fleet has gained enough experience to surpass the need for any handholding (Nichols, 2019), which entitles them as the leader in the ADR race.

Another two contenders are two well-known giants, Amazon and FedEx, which both revealed their ADR in the first quarter of 2019 aiming at lunching in pilot cities around this time now (FedEx, 2019; Scott 2019).

All three contributions are equally interesting for this project because they appear, as little information is revealed at this stage in the race, to have same specifications (FedEx, 2019; Starship, 2019; Scott, 2019). Them having similar specifications provides guidance on the specifications of the agent, central for the simulations in Unity. For now, it is noted that the ADR’s appears to have a height around 0.5-1.5 metre, a width and depth of 0.5 metre, a total weight (including cargo) of 45-50 kgs and a speed around 5 km/h (FedEx, 2019; Starship, 2019; Scott, 2019).

## Unity – as a simulation engine for research in DRL

Unity is best described as being a multi-functional platform, enabling development of everything from complex high resolution multiplayer games to less complex mobile games over to VR/AR applications, and increasingly as a challenging set-up to conduct research on artificial intelligence (AI) (Juliani et al, 2018; OpenAI, 2018; Sadeghi & Levine; 2016).

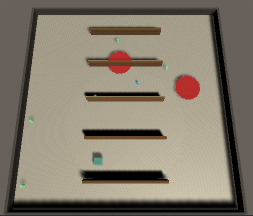
With increasing complexity, sometimes in an exponential manner, comes a need for low latency and distributed computing. Irrespectively of the increasing complexity being a derivative of a high-resolution 3D game consisting of, potentially, thousands of players with widespread interaction patterns, or research on algorithms that roughly needs 18 million video frames to surpass human performance (Hessel et al, 2017), satisfies Unity the need (Juliani, 2019). The underlying engine runs in de-synchronized fashion, supporting simulations at run times at least 100 times faster than real time. This is possible while still maintain physics and frame rendering. A final appealing feature of the computational side is the ability to run concurrent training session, internally in one application as well as externally, enabling the possibility to utilise cloud computing for further increased computational power.

The diversity in the application areas implies that Unity is a serious candidate for modelling the complex dynamics of urban environments. The most appealing feature is the ability to replicate real-life physical complexity, and thereby enabling realistic movement patterns as well as interaction between objects. This is an important factor with an underlying interest in enabling the results to generalise to the real-world, as higher similarity between the environment within the results are obtained and the environment in which the results are deployed, increases the likelihood for generalisation. Another feature of Unity, which makes it appealing for modelling urban dynamics, is the possibility to model complex social interaction. Unity comes with two options to add layers of social interaction to the environment. The *NavMesh* class provides the ability to add AI agents to the environment, useful for spatial queries, as pathfinding. As described in detail later, this class is utilised to model pedestrian behaviour in the constructed urban environment for this paper. NavMesh agents can interact with other NavMesh agents, as well as avoid other moving obstacles, enabling a layer of social interaction with relative ease.   
A more challenging, and perhaps are more interesting way to add social interaction to the environment is by utilising the ML-Agents toolkit, a central part of this paper. The ML-Agents toolkit bridges sophisticated machine learning methods with the graphical interface and complex physical engine of the traditional Unity application, enabling a new setting to push the boundaries for DRL research (Juliani, 2018). The toolkit allows researcher to utilise pre-defined algorithms, based on TensorFlow, or define them themselves, via a Python API. In the light of the NavMesh class, the toolkit puts the control of the interaction in the hands of the researcher, which carries a certain responsibility. It requires the researcher to exhibit a greater understanding of the task and modelling at hand, and so reduces the possibility of headless simulation – limiting the risk of another black box appearance.  
With the introduction of ML-agents, can Unity partly be regarded as the new kid in the class of software’s usable for Agent-Based modelling (ABM). The ML-Agents toolkit makes Unity an appealing contender to the traditional software’s used for ABM, by the fact that the scripting languages are C# and Python. The use of C# and Python implies low latency along with a wide variety of options for further data processing, through open source libraries, and user support, from the enormous communities surrounding the two languages.

Consider the ending of this section, if it should be changed/extended a bit.

### The environment

The environment explored in this paper is seen in figure 1, and it is intended to simulate areas of a city which are less trafficated with cars, and more dominated by pedestrians, formally known as pedestrian streets. An example hereof could be the area around Carnaby Street in Soho, London. Simulating a network of pedestrian streets is justified by the current state at which ADR’s are today, i.e. still being in an early stage and facing challenges in the unpredictability of the real world (Nichols, 2019).

*Figure 1 – The environment*  
  
*The environment contains five moving pedestrians (light green), two high-density areas (red), the target (dark green square) and the five static obstacles.*

Developing an environment, formally known as a *Scene* in Unity, to facilitate the possibilities within the ML-agents toolkit requires some basic objects; an actual *environment* to explore, an *academy*, an *agent* and a *target*. With the presence of these objects one gets something similar to the environment seen in figure 2, which on top includes walls to prevent the agent from falling of the surface – yet these are not strictly needed, as to be elaborated on shortly when digging into the set-up of the *agent* component. Figure 2 is an example of how everything could be organised within a scene.

Every application build with Unity is made of *Scenes, GameObject’s, components* and, often at least, *scripts*. An application can contain an arbitrary number of scenes, and the application shown in figure 2 contains one, namely *SensorEnvironment-4.2.* Every scene contains GameObject’s, in which components and scripts are attached, to sustain any form of behaviour imaginable. Components enables the use of all the built-in functionalities in the Unity Engine, and scripts provides the researcher with the option to take full control.

*Figure 2 – A scene in Unity containing the components of the ML-Agents toolkit*A screen shot of a computer

Description automatically generated  
*The scene contains all available elements within the environment.*

One thing to note from figure 2 is that all necessary elements are contained in a *prefab* named *Area\_EnvX*, where X referrers to the current version of the implementation[[17]](#footnote-17). Prefabs are user defined GameObject’s, and the use of prefabs are a neat way of altering similar objects simultaneously. In relation to figure 2, containing all necessary elements in a prefab, is a way to utilise a parallelised set-up, allowing for faster training but more on that later.

#### Environment

The walkable area is labelled *ground* in figure 2, and it serves two important purposes. Firstly, it defines the extend of the area through its scale, seen in figure 3. The extend of the area is used to ensure that random placing, through scripting, of objects happens within the bounds of the traceable area.   
Secondly, it serves as a container for the objects belonging to this training area[[18]](#footnote-18). Initialising new GameObject’s as children of another GameObject is a way to ensure intended interaction with relevant GameObject’s. It allows the researcher to write generic scripts and not instances specific scripts, which are in general good practice, and especially desirable when working with parallelised set-ups.

*Figure 3 – The ground object  
A screenshot of a cell phone

Description automatically generated*

***Tags***

Every GameObject within the training area is tagged, as seen in the right side of figure 3. Tags is an elegant way to differentiate GameObject’s from each other, especially useful in association with collision detection, collecting observations on the state of the environment and random placing of GameObject’s.

***Layers***

Assigning different GameObject’s to different layers is used to either include or exclude certain GameObject’s from some sort of detection. This is useful in the two-brain set-up, ensuring that one brain handles avoidance of dynamic obstacles and one brain takes care of the general navigation towards the target.

***Static Objects***

To the right of the GameObject’s name is the ability to mark a GameObject as static, which is used in connection with NavMesh agents. Static GameObject’s are part of mesh in which a NavMesh agent can navigate, but this will be described in a bit more detail below.

***Geometry of a GameObject***

Any object having a shape contains a Mesh filter, defining the geometry of the object, and a Mesh renderer, which ensures rendering of the object at the position specified in the transform component. Figure 3 shows that the *ground* element is a plane, having a size of 80x80x1, positioned at (0,0). The height (size in the y direction) of the *ground* element is not as such important, if it is above 0, to sustain the plane rendering.

Within the bounded *ground,* not necessarily as child objects, is six types of objects placed, two of them elaborated in individual sections below, and the other four are *walls, obstacles, pedestrians* and *crowded areas.*

##### Walls & Obstacles

The walls as well as obstacles are, as GameObject’s, identical to the *ground* GameObject, with a different tag along with the obviously different size and position. The walls and obstacles have different materials, to indicate that they represent something different. The walls can be thought of as boundaries for a certain pilot area in a given city, and the obstacles represents buildings. Both walls and obstacles serve as a resetting mechanism upon collision with the agent.

##### Pedestrians

The pedestrian GameObject is a prefab, which is attached to the academy from where it is initialised. The initialisation will be described in more detail in the academy section. Figure 4 shows the components attached to the pedestrian prefab, of which four are interesting to elaborate on.

***Collider***  
The collider together with the *rigidbody* component, is what that enables collision detection between the object and another object, with a collider and rigidbody component attached as well. The settings in the collider is irrelevant, as they are standard settings matching the scale of the GameObject.

***Rigidbody***  
The rigidbody is the component that enables the physics engine to take control of the movement of the GameObject. The *mass* of the GameObject is specified in kilograms and is set equal to a reasonable average value for a male.

*Figure 4 – The pedestrian prefab*  
A screenshot of a cell phone

Description automatically generated

***RB: Drag***

The *drag* is a force working in the opposite direction to the movement of the object, specifying at what pace the movement of the object is decreased. The value for the drag of the pedestrian prefab is calculated using (1), to ensure realistic behaviour in the simulation, because the default value is zero which is not in accordance with realistic behaviour.

Where is density of the fluid that the object passes through, air in this case here, is the speed (m/s) at which the object moves, is the drag coefficient (unit less) and A is the cross sectional area related to the movement, which is the area of the object normal to the direction of the movement. Table 1 shows the values used, and the calculated drag.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 1. Drag** |  | Value | Unit |
| Density (air), rho: |  | 1.225 | Kg/m^3 |
| Speed, v: |  | 1 | m/s |
| Drag Coefficient, Cd\*: |  | 1.3 | Unitless |
| Cross sectional area, A: |  | 3.141593 | m^2 |
| *Average of human body in upright position, and at the same time the coefficient of a short cylinder.* | | |  |
|  |
| Drag: |  | 2.501493 |  |

The *angular drag* is how much a rotation is slowed down, and it is kept at standard value, because the default value is within a realistic order.

The rigidbody is marked as being *kinematic*, which implies that the object isn’t influenced by any forces. Why have the rigidbody attached then? Because it ensures better collision properties having both the collider and the rigidbody attached to an object, and all movement is handled by the NavMesh agent component below.

***Nav Mesh Agent***

The Nav Mesh Agent component is what turns an empty GameObject into a Nav Mesh agent. At least initially, is every parameter herein kept default, except speed and angular speed. The speed of the agent is set equal to 1 m/s, the same as the ML agent, which is roughly equal to a speed of 3.5 km/t – the average chilled walking speed.  
The angular speed is set equal to 150, a bit above the default value of 120, as it is equal to the angular speed used in the ML agent.

***Walking around***

The final component is a custom script, written to ensure that the pedestrian walks around the training area continuously doing an episode. The script has two public variables, the radius and frequency. The radius is the distance from the agent, that a new target point is draw within. One target point is drawn within the frequency specified, and there should therefore strike a balance between the radius and the time, such that the agent has time to travel that distance within the specified time.

##### Crowded areas

The crowded area is a prefab and is initialised from the academy object. The crowded area object represents an area with a lot of potential challenging obstacles, which could be a public square or an area on a pedestrian street, where some entertainment unfolds etc. It is an area where the agent can pass through, but it will require a significant amount of additional effort, which implies that it potentially is beneficial for the agent to avoid interaction. How to learn the agent that, is a real challenge as will be clear later, and different things are tried to motivate the agent to learn. For now, just focus on the fact that the crowded area can attain different densities, and later it will be specified how the densities are used to shape the reward function, which ultimately determines the success of the learning. The crowded area is represented as a circle, which has a certain *density*, see figure 5. The size of the area is set through the academy and is therefore as such fixed.

*Figure 5 – The crowded area prefab*A screenshot of a cell phone

Description automatically generated

***Density***

The density values that a crowded area can possess are based on empirical population density estimates for 2019 from London, see table 3 and figure 6 & 7. The actual values of an area if randomly drawn from the list of possible values, and the density as well as the location of the area is changed every time a new episode begins.

|  |  |
| --- | --- |
| **Table 3 - Population densities, London.** | |
| Max Ward: | 0.0289 |
| Max Boroughs: | 0.0164 |
| Overall: | 0.0058 |

*Figure 6 – Population density, Boroughs, London  
A close up of a piece of paper

Description automatically generated*

*"Contains National Statistics data © Crown copyright and database right [2015]" and "Contains Ordnance Survey data © Crown copyright and database right [2015]"*

Figure 6 and 7 show the population density in London, at borough and ward level respectively. It is no surprise that the densest areas are close to the city centre, and the densest ward is located within Westminster borough, namely Church Street.

*Figure 7 – Population density, Wards, London   
A close up of a map

Description automatically generated  
"Contains National Statistics data © Crown copyright and database right [2015]" and "Contains Ordnance Survey data © Crown copyright and database right [2015]"*

#### Academy

Up until now, none of the outlined parts of the environment been specific to the ML-Agents toolkit. The academy is one of three cornerstones of the toolkit, and it serves to bridge actions and observations to the TensorFlow-based models in Python. To be right, it is an instance referred to as the *brain* that serves as the bridge, but one cannot have a brain without an academy. The academy has only a single component attached, and the content of the academy script is seen in figure 9.

***Brain***

The brain is what makes determines the action, based on the observed observations in the current state, and the academy facilitates the training of the brain. There are two types of brains, learning brains and player trains. Learning brains learns the policy based on the neural network, implemented in TensorFlow and is to be elaborated on in a following section. The player brains allow the researcher to test before invoking the learning brain, by giving the researcher the option to control the agent with keys on the keypad. See figure 8 for the configuration of a brain.

*Figure 8 – A learning brain*  
A screenshot of a cell phone

Description automatically generated

A brain can take *vector observations* as well as *visual observations* as input, and it outputs an action vector. The size of both the observation vectors and the action vector is specified by the researcher, and it is highly problem specific.  
The learning brain used this case takes in vector observations, as visual observations requires more computational power than available for this paper. The size of the vector is 126 and the size is determined by specifications in the agent. Describe partial/full observability.  
When the control checkbox is ticked, next to the attached learning brain in figure 9, the agent learns otherwise is uses a pre-trained model, which should be attached to the brain, see figure 8.

*Figure 9 – The academy*  
A screenshot of a cell phone

Description automatically generated

It is possible to add more than one brain to a single agent; however, it is not possible to train multiple brains on a single agent – yet. Training multiple brains on an agent would present some interesting possibilities, which will be discussed at the end of this paper.

***Varying complexity***

The main purpose of the academy, besides acting as a container for the brain, is to initialise and alter the complexity of environment. This is revealed by taking a look at the code underlying the academy script, which except for some helper functions contains just two methods; InitialiseAcademy() and AcademyReset().  
AcademyReset(), for this paper, is only used if *curriculum learning* is utilised, and the purpose is to add increased complexity to the environment at specific times. More on that in the coming section. One could alternatively change location of certain objects from the academy and thereby either using the option to specify a maximum number of steps call *Done* to reset, but that is not how it is done in this paper. Changing location of certain objects is done through the agent script, which is described shortly.

Before going over the configuration specifications, it is natural to touch the reset parameters as they are related to the above paragraph. The reset parameters are the variables that changes when the environment is being reset, and they are a major part of the agent script, to be described. In this paper here, they take input from the specified curriculum.

Below the reset parameters are the two prefabs, *pedestrian* and *crowded area*, attached for initiation of the environment.

***Configuration***

The width and the height determine the size of the application window when training is done outside the editor.

The quality level is the quality of the camera input, if visual observations are provided to the brain, and is so not relevant for this paper.

The time scale is the speed at which the simulation is carried out – 1 is real time and 100 is 100 times faster than real time. The actual level of time scale does not as such affect the performance of the training, only the training time, yet some physics calculations gets inaccurate with a too high time scale, and so affecting the performance. This should only be relevant if one has objects that travels at high speed, which isn’t the case in this paper.

The time scale is set to 100, based on test simulations shown in appendix A showing no effect on performance, yet ensuring as fast as possible training.

The final configuration parameter is the targeted frame rate, which is the rate at which Unity aim at rendering the frames, which shouldn’t be altered unless one is using visual inputs.

#### Agent

The agent object is by far the most complex, in terms of the number of components and methods contained in the attached script. The content of the agent object is seen figure 8.

The agent GameObject contains a ray perception component along with two custom scripts, other than the familiar components as of the mesh, the rigidbody and the collider components.

Other than the mesh components as well as the rigidbody and collider component, which are previously described, contains the agent a ray perception component along with two custom scripts; one to draw trails and one to hold the necessary methods needed for any agent, to leverage the ML-Agents toolkit.

***Ray perception***

The ray perception component enables the agent to cast rays in specified length and direction and are here used to collect observations about the state of the environment.

Simulates the use of LIDAR, used in real-life robots.

***Agent Script***

Any ML agent needs an agent script, to hold the agent-specific methods, just as the academy needed an academy script. An agent script contains some default variables and options, listed above the grey line in figure 10.

*Figure 10 – The content of the agent object  
A screenshot of a cell phone

Description automatically generated*

***Brain***

An agent needs a brain to control the movement of the agent, and it is the same brain as specified in the academy, at least doing training. Certain methods are needed to control the movements, more specifically; CollectObservations, AgentAction and MoveAgent. To have a closer look at all methods, the interested reader should open the agent script using ones preferred text editor.

The CollectObservations method entitles the agent with vision, and here the agent has 180 degrees sensor vision, in steps of 10 degrees, spanning in front and with a length of 50 meters. These observations are what makes up the vector observations, mentioned in the description of the brains. The agent is provided with five tags to recognise, and so it casts a ray for each of the tags to recognise, for each of the degrees specified. Furthermore, it keeps track of distance to the objects, and if an object has been missed. The implication is that the resulting observation vector has the dimension , which for the specific case here means that the observation vector is .

The agent chooses an action, based on the observations about the current state of the environment. The action/-’s is chosen by the brain, and facilitated to the agent through the AgentAction method, in which the action signal is translated to actual movement via the MoveAgent method. One example is, that the agent is moved forward with a constant speed, and the action of the agent is degrees to rotate, to sustain the desired navigation around the environment. This is how it is done in this paper, and the level of the constant speed is subject to change by the researcher via the **speed** variable, below the grey line.

By default, the speed of the agent is equal to 1, with the same argument as found in the outline of the *Nav Mesh Agent*.

***Camera/Render textures***

The agent script contains the option to specify camera/-s and/or render textures, if the brain attached to the agent uses visual observations.

***Max step***

As with the academy, the option to specify a maximum number of steps is present. The agent will be reset when the number of steps surpasses the specified number, which is useful to break unfavourable movement patterns, as is shown later.

***Reset on Done***

Another way to reset the agent is to call *Done*at some point, which usually is after colliding with an object in the environment.

Both *max step* and *reset on Done* is used in this paper. Collisions are detected using the OnCollisionEnter method, and if the agent collides with either a wall, a static obstacle or a pedestrian, the agent is rewarded with a large negative reward (-1) and Done is called, to reset the agent, and start a new episode. If the agent collides with the target, a large positive reward (1) is rewarded and *Done* is called as well.

On reset, not only the position of the agent resets, but the position of the target and the sensor clouds, along with the associated density, changes as well. Therefore, is the option to specify the levels and number of possible **densities** available under the agent script.

***Decisions***

Decisions can be done either at a specific interval or on demand, and this paper here uses decision at a specific interval. The decision interval (DI) should be chosen with the complexity of the environment and the speed at which the agent moves in mind.

As with many of the other parameters of the environment, it is of interest to choose the level at a level which generalises. The default level of DI is chosen to be every fifth step but can be subject to change. It seems natural that greater complexity/speed should benefit from lower DI.

***Draw trails***

The agent script contains the option to enable drawing, and specify the number of trails drawn, which serves to visualise changes in learning patterns doing training. A custom implementation is used, contained in the second custom script of the agent, because the default implementation, *trail renderer,* does not consider the resetting of the agent. *Trail renderer* draws the jump from where one episode ends to the start position of the agent, which minimises the information obtained by visualising the trails.

The custom implementation simply draws a line for each episode, as either a new child element of the agent, if the number of trails is less than the specified number of trails, or by modifying the oldest trails.

***Name of file***

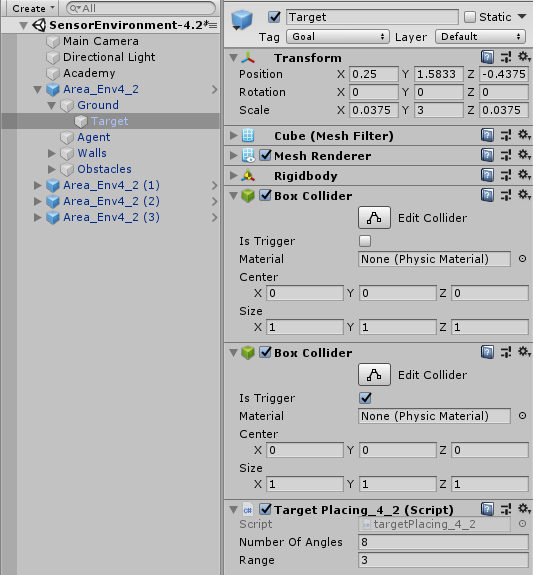
If desired, by checking ***verbose***, the agent collects additional information, compared to the information provided via TensorBoard and writes it to specified files. The additional information provides deeper insights into the progress of the training and highlights potential shortcomings. The additional information is information on the number of collisions with pedestrians and crowded areas, the steps used to locate the goal and the steps taken in the crowded area. However, this can be changed to suite the environment and the need of the researcher.

***Second brain***

The agent script is provided with an option to add a second brain, and potentially many more, which is useful to investigate the effect of separating tasks on individual brains. This is investigated at the later point.

#### Target

The target is an important element of the environment, as it is the sparse positive reward signal, which the agent searches for. In this paper, the target is a cube randomly positioned in the environment. As irrelevant as the geometry of the target is, as relevant is the size, relative to the environment. The relative size of the target affects how easily the target is located and can be thought of as the difference in locating a building on a street compared to a specific brick on a specific building, at that same street. The size of the target is fixed here, at a relative size of 3.75%, chosen arbitrary and the effect of the relative size of the target could be a topic for future investigation. 3.75% is equivalent to size of 3 meters.   
The content of the target is seen in figure 11.

*Figure 11 – The content of the target object  
*

The target contains, besides the custom target script, only familiar components. The target script has two box colliders .. Check if that is even necessary

The custom target script serves two purposes; randomly setting a new position of the target and check for collision with static objects in the environment. The public parameters of the script are input to the method used to check for collisions. By default, 4 rays are casted with a length of 3, covering all sides.

The target is not allowed to be located within another static object in the environment, to prevent conflicting collisions. Having the target located separately from other static objects, simulates the idea about locating a position outside a building for delivery by the ADR.

### Training

Training can be done either in the editor or by running a build application of the environment. How to carry out training from the command line is described neatly by (Juliani et al., 2019), and is so not described here.

Running training from a build application of the environment provides some desired possibilities, where two of them are running concurrent runs and concurrent environments.

Concurrent runs are independent, which is beneficial for benchmarking.   
Concurrent environments are equivalent to having multiple training areas within one environment, which implies more experience being sampled, which should result in improve learning (Teng et al, 2019). Running multiple environments is a way to speed up training beyond have multiple training areas. It is computationally demanding, and it is usually used together with cloud computing.

Cloud computing is not a possibility here, and this paper is so limited to the use of multiple training areas.

### TensorBoard

TensorBoard is used to visualise a wide range of statistics related to the training conducted, covering environment, policy and learning statistics. Each of the statistics available is described in detail in (Juliani et al, 2018b).

## Curriculum Learning

Specifying curriculums in Unity is done in a json file, seen in figure 3, and contains five different parameters

An interesting aspect is how the curriculum should be designed, in order to facilitate optimal learning of the agent, and this is almost an area for research itself. The aim of this paper is not to shed light on optimal construction of curriculums, and the curriculum used here is chosen such that it addresses three of the four main challenges equally, namely the static obstacles, the crowded areas and the dynamic obstacles.

# Exploration/Exploitation trade-off

Coming of configuring the set-up, it is time to explore the environment, address the challenges and hopefully solve the environment.  
When is the environment *solved*? The environment is solved when the agent can cope with the different type of challenges that the environment processes. As mentioned in section X, the environment of this paper processes four main challenges; difficult areas, static obstacles, crowded areas and dynamic obstacles.

## The look of learning

Learning can be expressed in many ways (Juliani et al, 2018b), however, the most intuitive statistics is the average cumulative reward (ACR) and the average length of an episode (ALE), examples are seen from figure 13.

The ACR is expected to converge to the maximum obtainable level of reward within a single episode, when learning has taken place and the agent has solved the environment. The ALE should converge to the minimum number of steps needed to obtain the reward. If a small negative reward is provided at each step, the ALE and ACR develop almost inversely, because the number of steps used bound the maximum obtainable reward.   
The speed of which the convergence takes place is a direct consequence of the complexity of the environment and the degree of assistance provided to the agent, as will be clear doing this section.

*Figure 12 – The simplest version of the environment.*

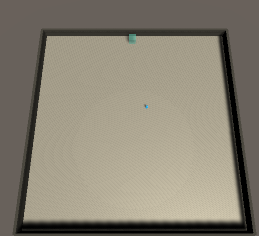
   
*The simple environment contains only a target (darker green square) and the agent (light blue).*

Figure 12 shows the simplest possible environment, namely without any obstacles. The target is here in the upper middle of the environment, yet it changes position every episode. Learning in this environment is straight forward, and this is seen from figure 13. Figure 13 shows how ACR and ALE converges rapidly to their optimal levels, which are obtained after around 75.000 steps in this case.

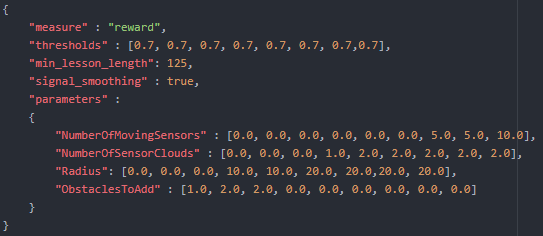
*Figure 13 – ACR and ALE for the environment from fig.3*  
A screenshot of a cell phone

Description automatically generated *The average cumulative reward and average episode length obtained from learning in the simple environment from figure 12.*

The shapes of ACR and ALE seen in figure 13 are ideals, and far from trivial to obtain with the slightest degree of complexity present. The road to meaningful learning requires careful design of the aid provided to the agent to ensure generalisation and so providing the agent with the ability to handle unseen environments, which is one of the aims of this paper. Figure 14 shows the same output, as in figure 13, for the far more complex environment seen in figure 1.

*Figure 14 – ACR and ALE for full environment from fig. 1.*

The first step towards improving learning is to introduce a curriculum, for the agent to learn from, to gradually add complexity to the environment, as the agent learn to cope with novel challenges. The difficult areas are a separate task, which will be addressed shortly.  
The initial curriculum used in this paper is seen from figure 15, and the reader should revisit section 2.4 for a full discussion of each of the parameters.

*Figure 15 – The initial curriculum used to facilitate improved navigation of the robot.*

Starting out with the parameters specified in the curriculum, it is noticed that there is at least one parameter for each of the obstacles for the robot to avoid. The curriculum is designed such that the obstacles are introduced relative to their degree of complexity. The first challenge is the static obstacles, then the semi-static crowded areas and finally the dynamic pedestrians. Each type of obstacle is introduced over three lessons, and the same threshold to be passed is required. The threshold is at this point not fixed, as it should be set based on the two parameters of the environment, which will be addressed below. Signal smoothing and the minimum number of episodes, in which the threshold is surpassed, are here chosen with the same objective, to sustain generalisation. Signal smoothing is enabled to ensure robustness of the signal. The minimum number of episodes are chosen such that the number ensures robustness, yet not to such a degree that it becomes a burden. The minimum number of episodes should be chosen with the threshold in mind. Too high a threshold compared with too high a requirement on the robustness of the signal, will likely prevent any progress, depending on the complexity of the environment. Choosing too low a threshold and too low a level of the robustness, can imply that progress occurs too soon. A common misconception when working with reinforcement learning is the amount of data needed to sustain meaningful learning, the planning fallacy as of (Irpan, 2018). Figure 16 shows ACR and ALE, along with the changes in lessons[[19]](#footnote-19), for a training session with the initial curriculum.

Figure 16 - *ACR and ALE for first training session using curriculum learning*

As discussed in section X, is the ability of the agent to generalise to unseen environments one of the

Figure 4 illustrates the aim at the end of this section, however, the challenges of the environment will blur these graphs

## Insights:

*Randomization of rewards usually poses a problem for the agent - it doesn't know what to do because it doesn't know what reward to expect.*

*.. the agent can't learn a perfect model of how observations correspond to rewards (i.e., they're "noisy")*.

Source: <https://github.com/Unity-Technologies/ml-agents/issues/1680>

## Stuff for the discussion:

As RL is nothing more than a function mapping, either exact or approximative, it follows sort of trivially that randomness makes the mapping blurry, which is the reason why the agents has a hard time dealing with the sensor clouds.

## Thoughts:

If it turns out that, no matter the actions taken, it seems impossible for the agent to learn avoiding the sensor clouds, presumably because of the randomness in the penalties of the steps (as derivative of above), one idea can be to let the penalty be fixed but use the density to change timescale/speed.

# Policy Evaluation

# Appendix

#### Tuning of hyperparameters

There are three parameters that qualifies as hyperparameters of the environment, when looking back at the outlined environment. These three are; the *speed* at which both the agent and the pedestrians move, the *decision interval* of the agent and the *time scale* at this the simulations are carried out. The reason for these three parameters are because they naturally neither are part of a curriculum nor determined by the desire of having realistic physics in the simulated environment.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Table 2. Environment configuration (200.000 steps)** | | | | | | | | |
| Speed | Decision Interval | Time Scale | Training time | Deviation\* | ACR: Mean\*\* | ACR: Std. dev. | AEL: Mean | AEL: Std. Dev. |
| 2 | 5 | 60 | 1650 | 22.2% | 0.663 | 0.65 | 40.32 | 19.41 |
| 1 | 5 | 60 | 1500 | 11.1% | 0.920 | 0.59 | 39.55 | 15.59 |
| 1 | 3 | 60 | 1350 | 0.0% | 0.916 | 0.80 | 101.31 | 198.75 |
| 1 | 7 | 60 | 1750 | 29.6% | 0.896 | 0.63 | 34.07 | 15.86 |
| 1 | 10 | 60 | 1800 | 33.3% | 0.812  **Re-create in Tex** | 0.56 | 26.22 | 9.32 |
| 1 | 5 | 100 | 1350 | 0.0% | 0.908 | 0.62 | 43.03 | 19.58 |
| 1 | 5 | 20 | 2020 | 49.6% | 0.908 | 0.57 | 44.02 | 21.15 |
| \*: Relative to the fastest, \*\*: Mean of converged path | | | | |  |  |  |  |

# References

Juliani, A. et al. (2018a). Unity: A General Platform for Intelligent Agents.

Juliani et al. (2018b). *Using TensorBoard to Observe Training*. Online resource, url: <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Using-Tensorboard.md>. Visited: 14/06/2019.

Juliani et al. (2019). Training ML-Agents. Online resource, URL: <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-ML-Agents.md>.

Teng et al (2019). Faster training on real games. Online resource, URL: <https://blogs.unity3d.com/2019/04/15/unity-ml-agents-toolkit-v0-8-faster-training-on-real-games/>,

Irpan, Alex. (2018). Deep Reinforcement Learning Doesn’t Work Yet. Online resource, URL: <https://www.alexirpan.com/2018/02/14/rl-hard.html>. Visited: 23/07/2019.

Merrit, Tom. (2019). Top 5 Delivery Robots. TechRepublic, URL: <https://www.techrepublic.com/article/top-5-delivery-robots/>.

Nichols, Greg. (2019). Delivery Wars: Amazon’s new delivery robot vs. Starship’s college munchie robot. ZDNet, URL: <https://www.zdnet.com/article/amazons-new-delivery-robot-vs-starships-college-munchie-robot/>.

FedEx. (2019). Delivering the Future: FedEx Unveils Autonomous Delivery Robot. FedEx newsroom, URL: <https://about.van.fedex.com/newsroom/thefuturefedex/>.

Starship. (2019). The Self-Driving Delivery Robot. Starship/About, URL: <https://www.starship.xyz/business/>.

Scott, Sean. (2019). Meet Scout. The Amazon blog; dayone, URL: <https://www.starship.xyz/business/>.

1. All sets of the MDP are finite [↑](#footnote-ref-1)
2. This is not entirely true, as it is common practice to reward the agent with a minor negative reward at each state (also called *step*), to incentive fast learning. [↑](#footnote-ref-2)
3. A reference to the fact that this class of methods is the most used today (Karpathy, 2016). [↑](#footnote-ref-3)
4. see (Sutton and Barto, 2018) for a description of both methods. [↑](#footnote-ref-4)
5. A greedy policy is a policy, that chooses the action resulting in the maximum outcome. The alternative is -greedy policies, which with probability chooses a random action and with probability chooses the greedy action. [↑](#footnote-ref-5)
6. See (Sutton and Barto, 2018), Chapter 9, section 4 for a full description. [↑](#footnote-ref-6)
7. See (Lecun et al., 2015) for a detailed description, of the architecture and each of the components. [↑](#footnote-ref-7)
8. N previous observations are stored, over many episodes, from which T *experiences* are randomly sampled from . Updates are done on the sampled experiences, and the agent chooses an action according to an -greedy policy, i.e. off-line learning, (Mnih et al., 2015). [↑](#footnote-ref-8)
9. Page 5, (Silver et al., 2016). [↑](#footnote-ref-9)
10. The first derivative of a function. [↑](#footnote-ref-10)
11. See (Lecun et al., 2015: P. 436, 2nd paragraph towards the bottom) for a straightforward explanation. [↑](#footnote-ref-11)
12. A piece of software, integrating The Arcade Learning Environment (Bellemare et al., 2013) and the preferred script editor of the researcher. [↑](#footnote-ref-12)
13. Height, width and channels. [↑](#footnote-ref-13)
14. The first matrix of weights has the size (pixels x neurons in the hidden layer); and the second weight matrix has a size 5x1, implying a total of million parameters. [↑](#footnote-ref-14)
15. Training of nets using gradient of the objective function, see (Rojas, 1996: Chapter 7) and footnote 11. [↑](#footnote-ref-15)
16. See (Lidoris et al., 2009) for a description. [↑](#footnote-ref-16)
17. It is good code practice to create new versions every time major changes are implemented, to minimise the risk of lost functionality and malfunctioning. [↑](#footnote-ref-17)
18. Made up of the *Area* prefab. [↑](#footnote-ref-18)
19. Each lesson is an entity in the arrays of the parameters in the curriculum. [↑](#footnote-ref-19)