**Appendix**

Contents

[1. Developing digital environments in Unity 2](#_Toc17360236)

[1.1 Environment 3](#_Toc17360237)

[1.2 Walls & Obstacles 5](#_Toc17360238)

[1.3 Pedestrians 5](#_Toc17360239)

[1.4 Crowded areas 9](#_Toc17360240)

[1.5 Academy 9](#_Toc17360241)

[1.6 Agent 13](#_Toc17360242)

[1.7 Target 18](#_Toc17360243)

[1.8 Training 19](#_Toc17360244)

[1.8.1 Training hyperparameters 19](#_Toc17360245)

[1.9 TensorBoard 20](#_Toc17360246)

[1.10 Tuning of hyperparameters of the environment 20](#_Toc17360247)

[2. Proximal Policy Optimisation explained 21](#_Toc17360248)

[2.1 Reward Shaping 22](#_Toc17360249)

[3. Supplement to section 4 23](#_Toc17360250)

[3.1 Comparisons 23](#_Toc17360251)

[3.1.1 Baseline 24](#_Toc17360252)

[3.1.2 Exploring under certainty 24](#_Toc17360253)

[3.1.3 Exploring under uncertainty 26](#_Toc17360254)

[3.2 Effect of restricting the number of steps for the agent 26](#_Toc17360255)

[3.3 Effect of invoking maximum steps 27](#_Toc17360256)

[3.4 Effect of parallelisation 28](#_Toc17360257)

The first part of this appendix is devoted to the technical aspect of the study, implying a full treatment of the elements of the environment and their configuration.  
The second part of the appendix contains extended content related to the methodology.  
The third part contains supplement material to the analysis.  
The fourth part contains an overview of the scripts developed, related to everything else than the operation of the environment.

# Developing digital environments in Unity

Every application build with Unity is made of *Scenes, GameObject’s, components* and *scripts*. An application can contain an arbitrary number of scenes, and the application shown in figure 14 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.

Developing an environment, formally known as a *Scene* in Unity, to facilitate the possibilities within the ML-agents toolkit requires some standard objects; an actual *environment* to explore, an *academy*, an *agent* and a *target*. Figure 14 is an example of how everything could be organised within a scene.

*Figure 14 – 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 14 is that all necessary elements are contained in a *prefab* named *Area\_EnvX*[[1]](#footnote-1). Prefabs are pre-defined GameObject’s, and the use of prefabs are a neat way of altering similar objects simultaneously. For the environment in question, containing all necessary elements in a prefab, is a way to utilise a parallelised set-up, allowing for faster training.

The following describes key components of the entire environment, divided into the four categories made up by the basic objects required by the ML-Agents toolkit. All components left out of the following description can be found in the appendix, to ensure reproducibility.

## Environment

The walkable area is labelled *ground* in figure 14, and it serves two important purposes. Firstly, it defines the extend of the area through its scale, seen in figure 15. 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[[2]](#footnote-2). Initialising new GameObject’s as children of another GameObject is a way to ensure interaction with intended 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.

***Tags***

Every GameObject within the training area is tagged, as seen in the right side of figure 15[[3]](#footnote-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 (Unity, 2019).

***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 15[[4]](#footnote-4) 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.

*Figure 15 – The ground object  
A screenshot of a cell phone

Description automatically generated*

## Pedestrians

The pedestrian GameObject is a prefab, which is attached to the academy from where it is initialised. The RL agent reset upon collision with them, as it is unacceptable for ADR’s to collide with pedestrians. Figure 16 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 16 – 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 (7), 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 5 shows the values used, and the calculated drag.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 5. 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.

## Academy

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. The academy is a Python API, and bridging is done through the *brain*.  
The academy has only a single component attached, seen from figure 18.

***Brain***

There are two types of brains, learning brains and player trains.   
Learning brains learns the policy based on the net implemented in TensorFlow. See figure 17 for the configuration of a learning brain.   
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.

*Figure 17 – 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 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 determined by the observations collected, with an example using sensor information provided in section 1.6.

Stacking observation vectors equips the agent with short-term memory, which can be beneficial when dealing with dynamic obstacles.

*Figure 18 – 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, discussed at the end of this paper.

***Varying complexity***

An important purpose of the academy is to initialise and alter the complexity of environment. The academy script, except for some helper functions, contains two methods; InitialiseAcademy() and AcademyReset().  
AcademyReset(), for this paper, is used with CL-based training, and the purpose is to update the environment with increased complexity at specific times.   
One could alternatively change location of certain objects from the academy, and thereby specify a maximum number of steps or call *Done* to reset. However, changing location of certain objects, in this study, is done through the agent script.

***Reset parameters***

The reset parameters are the variables that enables changing the environment on reset, and they are input variables to the agent script as well, used when changing locations.   
The values of the reset parameters are being set by the *curriculum[[5]](#footnote-5)*.

***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 50, based on test simulations shown in appendix A showing minimal effect on performance and training time.

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.

***Initialisation of environment***

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

## 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 19.

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. The purpose of the custom scripts is to draw movement trails and to hold the necessary methods needed, to leverage the ML-Agents toolkit.

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 enables the researcher to take more control with the interaction, 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.

The specifications of the agent are based on the specifications available on the current generation of ADR’s. 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).

***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. The rays resemble with LIDAR sensors commonly used for robots (Georgiev and Allen, 2004; Kümmerle et al., 2013; Starship, 2019; FedEx, 2019).

***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 19.

*Figure 19 – 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. Certain methods control the movement, more specifically; CollectObservations, AgentAction and MoveAgent.

CollectObservations serves to provide the agent with a vector of *observations*, sensor information in this case,and is only needed when the brain uses vector observations*[[6]](#footnote-6)*.   
The agent has 180 degrees sensor vision, in steps of 10 degrees, spanning in front with a length of 50 meters.   
The agent is provided with five tags to recognise, and a ray is casted for each of the degrees specified. For each ray is the following returned; A one-hot encoding, whether an object is detected or not (1/0) and the normalised distance to the detected object.  
The one-hot encoding returns 0 or 1 for each detectable object, depending of which object was detected.  
The observation vector has the dimension , which for the specific case here means that the observation vector is .

Describe partial/full observability.

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.  
The action of the agent is degrees to turn, as the agent is moved forward with a constant speed. The speed of the agent is subject to change, via the public *speed* variable.   
The speed of the agent is, by default yet subject for change, set equal to 1 m/s, which is roughly equal to a speed of 3.5 km/t – the average chilled walking speed.

***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 with a wall, a static obstacle, a pedestrian or the target results in Done being called, to reset the agent, and start a new episode.

On reset, not only the position of the agent resets, but also the position of the target and the crowded areas. Re-positioning the crowded areas causes re-drawing the densities.   
For this reason, the option to specify levels and number of possible densities is 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.

***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.

***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.

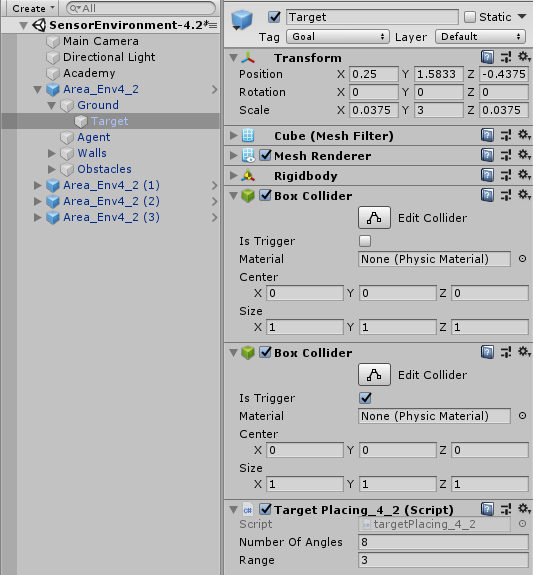
***Second brain***

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.

## Target

The content of the target is seen in figure 20.  
The target contains a custom script which serves two purposes; randomly setting a new position of the target and check for collision with static objects in the environment.

Continuously changing the position of the target increases the likelihood of the learning to generalise to unseen environments.

*Figure 20 – The content of the target object  
*

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, because of the associated cost, and this paper is so limited to the use of multiple training areas.

### Training hyperparameters

As within other areas of machine learning, requires RL algorithms considerations on various hyperparameters. Tuning of the hyperparameters becomes increasingly important as the complexity of the task increases. However, when tuning the parameters to a specific task or environment, some generalisation is lost.   
In this study, the hyperparameters are mostly kept at default values, with some being modified slightly to accommodate continuous actions instead of discrete. The full set of hyperparameters used is seen from figure 21, and for a discussion of the hyperparameters and how they should be set optimally, see (Schulman et al., 2017; Juliani et al., 2019).

*Figure 21 – Training Hyperparameters*  
A black and silver text on a screen

Description automatically generated

## 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).

## Tuning of hyperparameters of the environment

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 is because they naturally neither are part of a curriculum nor determined by the desire of having realistic physics in the simulated environment.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Table 5. 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 | 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 | | | | |  |  |  |  |

# Proximal Policy Optimisation explained

**Re-create in Tex**

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 (8) and the algorithm is shown in figure 22. Review (Shulman et al., 2017b) for a thorough description of PPO and the difference to other PG methods.

*Figure 22 – PPO Algorithm  
A screenshot of a cell phone

Description automatically generated*

PPO relies on an estimate of the advantage function, , and this estimate is the *generalised advantage estimator* by (Schulman et al., 2016). A full treatment is found in the reference and here it is noted that the estimate is the exponentially weighted average of the empirical returns minus the value function baseline given by (9).

## Reward Shaping

The idea of exploring reward shaping (RS) stems from (Mirowski et al., 2018) and is theoretically justified by (Ng et al., 1999; Schulman et al., 2016).  
The basis of RS is that early rewarding of favourable actions increases the likelihood of future actions being in favour of the target, as a sort of guidance of the agent (Ng et al., 1999).  
RS could help to improve the efficiency surrounding episodes with TLDA, discussed in section 4.2.2, by continuously motivating correct actions towards the goal.

(Mirowski et al., 2018) shapes the reward function by providing early rewards to the agent, which are proportional to the distance to the goal within some buffer[[7]](#footnote-7). Furthermore, early rewards are only provided for actions leading to decreasing distance, to prevent the agent form orbiting around the target harvesting early rewards.

This study uses a simpler set-up, based on the lower scale of the environment considered compared to the environment in (Mirowski et al., 2018), which is to provide the agent with an early reward equal to step penalty.

Figure X – Effect of reward shaping

# Supplement to section 4

## Comparisons

This section contains graphs supporting the tables from section 4.2.2, 4.3 and 4.4. The tables serve to summaries the insights from the graphs found here.

ACR/AEL are smoothed to better visualise, by calculating a moving average of the past three observations.

Weighted average growth rates are reported for the ACR curve, weighted by the time spent in each lesson. The growth rates are estimated by fitting a linear regression to observations in each lesson. The reported growth rates are per 2000 step, which is the summary frequency used doing training, see figure 21.  
The standard deviation is calculated on the detrended series.

The mean episode length and the standard deviation, calculated after 100,000 steps, are reported for the AEL curve.

### Baseline

*Figure 9 – Baseline comparison*A screenshot of a cell phone

Description automatically generated

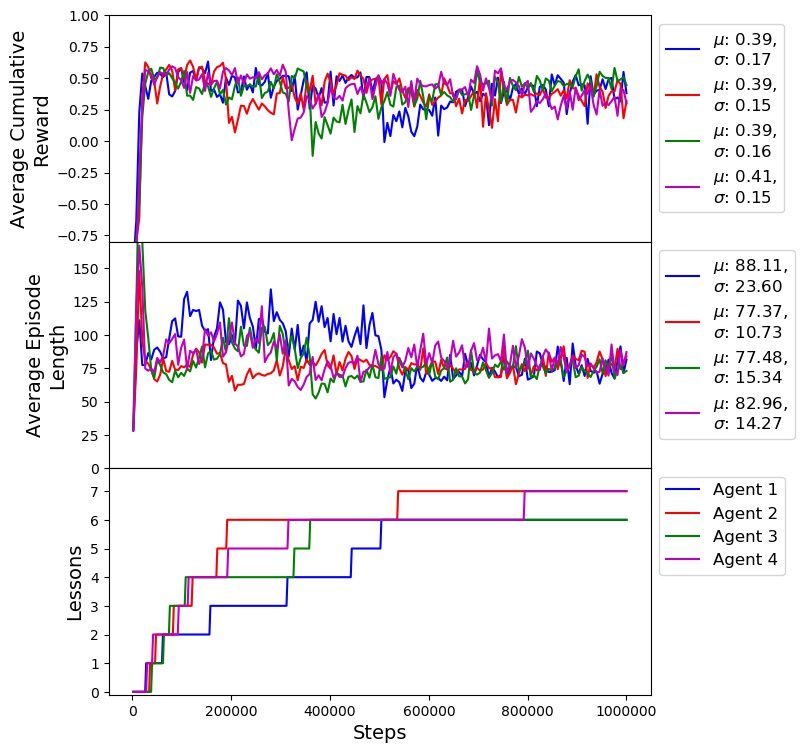
### Exploring under certainty

*Figure 14 – Full set-up under certainty*

A screenshot of a cell phone

Description automatically generated

### Exploring under uncertainty

*Figure 16 – Full Set Up Under Uncertainty*

## Effect of restricting the number of steps for the agent

Equivalent to figure 9 and 10 – yet the worst case seen.

*Figure 23 – The Effect of Restricting the number of steps*  
A screenshot of a cell phone

Description automatically generated

*Figure 24 – Number of Steps Within Each Episode*A screenshot of a social media post

Description automatically generated

## Effect of invoking maximum steps

Restricting the number steps allowed for the agent to take within one episode, reduces the skewness of the step distribution, which in turn results in a lot more stable training. This is seen from figure 9.

*Figure 22 – Distribution of steps taken by the agent*A screenshot of a cell phone

Description automatically generated

## Effect of parallelisation

Accompanying graph to section 4.2.3.

*Figure 26 – The Effect of Parallelisation  
A screenshot of a cell phone

Description automatically generated*

1. X referrers to the current version of the implementation. [↑](#footnote-ref-1)
2. Made up of the *Area* prefab. [↑](#footnote-ref-2)
3. ,29 P. 32 [↑](#footnote-ref-3)
4. [↑](#footnote-ref-4)
5. See section 2.4. [↑](#footnote-ref-5)
6. Visual observations are provided directly to agent. [↑](#footnote-ref-6)
7. See (Mirowski et al., 2018) section 5.3. [↑](#footnote-ref-7)