

**Faculty of Information Technology**

**Department of Mechanics, Information Technology, and Robotics**

Specialization of Robotics

**Patryk Stradomski**

s16716

**Sarsamax implementation**

**with OpenAI Gym and Unity engine**

B. Eng. Thesis

Thesis supervisor  
MSc. Eng. Piotr Gnyś

Warsaw, May 2021



**Wydział Informatyki**

**Katedra Mechaniki, Informatyki i Robotyki**

Robotyka

**Patryk Stradomski**

s16716

**Implementacja algorytmu Sarsamax przy pomocy interfejsu Gym od OpenAi i silnika Unity**

Praca inżynierska

Pod nadzorem:

Mgr. inż. Piotr Gnyś

Warszawa, Maj, 2021

# Abstract in Polish

Praca ta skupia sie na zaimplementowaniu algorytmu uczenia maszynowego znanego jako Sarsamax i wykorzystaniu go do przeszkolenia agenta korzystającego z środowiska zaprojektowanego w silniku Unity. Integracja ta odbywa sie przy pomocy przygotowanego przez Unity Technologies pakietu ML-Agents i interfejsu Gym stworzonego przez firmę OpenAI.

Praca wpierw opisuje zagadnienie uczenia przez wzmocnienie, jego historię, a także teorię potrzebną do zrozumienia algorytmu Sarsamax. Następnie opisane zostają Gym i ML-Agents wraz z firmami, które za nimi stoją oraz misjami jakie im przyświecają. Na koniec przedstawiona jest seria eksperymentów przeprowadzonych na implementacji algorytmu uruchomionej na środowisku “Basic” dostarczonym przez ML-Agents jak i eksperyment polegający na modyfikacji wspomnianego środowiska.

Słowa kluczowe: Uczenie przez wzmacnianie, Q-learning, Unity, OpenAI, Gym, ML-Agents.

# Table of Contents

[Abstract in Polish 3](#_Toc73464867)

[Table of Contents 3](#_Toc73464868)

[1 Introduction 5](#_Toc73464869)

[1.1 Abstract 5](#_Toc73464870)

[1.2 Goals 5](#_Toc73464871)

[2 Reinforcement Learning 6](#_Toc73464872)

[2.1 Introduction 6](#_Toc73464873)

[2.2 Components 6](#_Toc73464874)

[2.3 History 7](#_Toc73464875)

[2.4 Markov Decision Processes 8](#_Toc73464876)

[2.5 Exploration versus Exploitation 10](#_Toc73464877)

[2.5.1 - Greedy policy 11](#_Toc73464878)

[3 Q-learning 12](#_Toc73464879)

[3.1 Value Functions 12](#_Toc73464880)

[3.2 Temporal Difference Learning 13](#_Toc73464881)

[3.3 Sarsamax 13](#_Toc73464882)

[3.4 Q-table 14](#_Toc73464883)

[4 OpenAI 15](#_Toc73464884)

[4.1 Introduction 15](#_Toc73464885)

[4.2 Mission 15](#_Toc73464886)

[4.3 History 15](#_Toc73464887)

[5 OpenAI Gym 16](#_Toc73464888)

[5.1 What is Gym? 16](#_Toc73464889)

[5.2 Environments 16](#_Toc73464890)

[5.2.1 Algorithms 16](#_Toc73464891)

[5.2.2 Classic control 16](#_Toc73464892)

[5.2.3 Atari 17](#_Toc73464893)

[5.2.4 MuJoCo and Robotics 18](#_Toc73464894)

[5.3 Interface 19](#_Toc73464895)

[6 Unity ml-agents 20](#_Toc73464896)

[6.1 Unity Technologies 20](#_Toc73464897)

[6.2 Machine learning with Unity 20](#_Toc73464898)

[6.3 Environments 21](#_Toc73464899)

[6.3.1 Crawler 21](#_Toc73464900)

[6.3.2 Sorter 22](#_Toc73464901)

[6.3.3 GridWorld 22](#_Toc73464902)

[6.4 Robotic simulation 23](#_Toc73464903)

[6.5 OpenAI Gym integration 23](#_Toc73464904)

[7 Q-learning implementation using ML-Agents 24](#_Toc73464905)

[7.1 Basic environment 24](#_Toc73464906)

[7.2 Implementation decisions 26](#_Toc73464907)

[7.3 Experiments 26](#_Toc73464908)

[7.3.1 Run 1 – The first run 26](#_Toc73464909)

[7.3.2 Run 2 – Higher discount factor 28](#_Toc73464910)

[7.3.3 Run 3 – Higher discount factor and learning rate 30](#_Toc73464911)

[7.3.4 Equal distance to the rewards. 32](#_Toc73464912)

[8 Conclusions 35](#_Toc73464913)

[8.1 Further reading 35](#_Toc73464914)

[9 References 36](#_Toc73464915)

[10 Appendix 40](#_Toc73464916)

[10.1 How to create an environment Unity executable 40](#_Toc73464917)

[10.2 Using OpenAI Gym interface with Unity executable 42](#_Toc73464918)

[10.2.1 UnityEnvironment 42](#_Toc73464919)

[10.2.2 UnityToGymWrapper 43](#_Toc73464920)

# Introduction

## Abstract

This paper is focused on presenting a Q-Learning solution to a simple reinforcement learning problem that has been implemented in Unity Game Engine. The method has been written as a python script which communicates with the unity executable with the help of Unity ML-agents package through an interface that was first implemented by OpenAI Gym. The paper begins with familiarizing the reader with reinforcement learning; more specifically, the aspects of it that are necessary to have a clear understanding of the proposed solution. Please note that reinforcement learning is an ever-growing field and not all relating topics are covered in the aforementioned description. Then a brief description of OpenAI, Gym, Unity Game Engine and Unity ML-Agents is given in order to provide more context on how the solution works and how said parts are integrated. After that the titular solution is explained and effects of experiments are shown.

The paper presents the solution implemented using Python and Unity ML-Agents on Unity-made environment as easy to do and straightforward. It is proposing that the framework brings significant value by considerably quickening the modeling process of reinforcement learning solutions.

Keywords: Reinforcement learning, Q-learning, Unity, OpenAI, Gym, ML-Agents.

## Goals

Researchers working on reinforcement learning should be able to spend most of their time and resources on implementing the most efficient solution rather than on preparing environments and tools. The goal of this paper is to present the ease of integration of reinforcement learning solutions with an environment prepared in Unity Game Engine through Gym by OpenAi by implementing a Q-learning algorithm and running it on a prepared executable.

# Reinforcement Learning

## Introduction

Reinforcement learning is a field of machine learning that focuses on maximizing rewards while performing a certain task or navigating an environment.

Reinforcement learning should feel very familiar and intuitive to most people since it is a way of learning most similar to that of a person or animal. Sutton in his book [1] mentions as an example a child learning by interacting with its environment without any tutorage. Lacking guidance said child would have to rely on its actions to determine how the world will react to it and how it affects the child itself. This is a good example of how familiar the trial and error and learning from experience should be. However, this does not fully describe reinforcement learning.

A reinforcement learning problem would consider a certain **agent** performing **actions** that directly influence its **environment** in order to maximize the **reward** that results from that environment. The child from the aforementioned example might be an agent acting in an environment, but to fully set it in the reinforcement learning framework the child would need to have a defined goal. For example, if the child – representing an agent – would desire to learn how to ride a bike on its own, the possible actions could be defined as shifting its balance or steering the wheel whilst the reward would be determined by how far the bike has travelled. For the sake of creating the model of that environment the only things we would like to consider should be the child and its bike since the model needs to only be wide enough to contain the problem. [2]

## Components

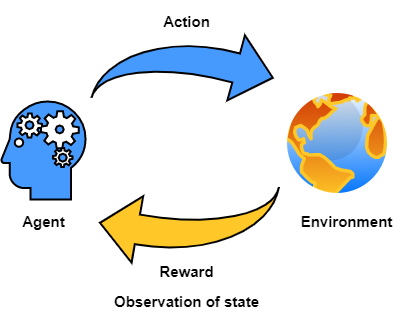


Fig. 1. An illustration of the agent-environment interaction. It shows how an agent performs actions on an environment and in turns receives a reward and an observation of the environment after that action. Then the process repeats.

**Environment** is a certain situation we want to model. It has its own set of rules describing the scope, possible actions and the reward function associated with those actions.

**State** describes a snapshot of the environment, agent, and their relation to each other in a given moment . It includes all parameters of the environment, and it can change through time along with values corresponding to those parameters. For example, in the bicycle model [section 2.1] the state would consider the position of the agent, and all considered physical forces acting on it.

**Observation**  of state is the information that can be accessed from the state. For the purposes of this paper, it will be assumed that . However, it has to be noted that there exist models where this is not true and applying such models directly to real-life scenarios would be difficult if not impossible. [3] In such cases the observable part of the environment can be modeled as probabilistically related to the true state. [4]

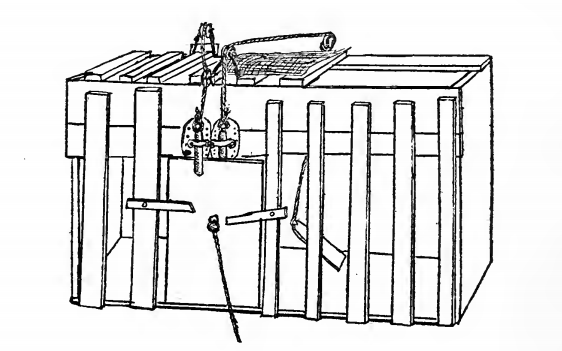
**Action**  can be performed on the environment in a given state. It may alter the state of the environment resulting in different actions being available All possible actions of an environment are called the action space and all actions available at a given state are denoted as . Reward function of an environment provides a specific reward based on the performed action.

We define an **Agent** as an entity capable of performing actions that influence the environment or its position. Each step it is performing an action based on a policy which is an instruction that defines the Agents behavior in a given state. Reinforcement learning aims to provide a policy that maximizes the sum of all rewards after arriving at a terminal state.

## History

Reinforcement learning as a subject is a convergence of two distinctive and -at the first glance – unrelated fields of science. Those fields being [1] [2]:

* Study of psychology of animal learning with an outstanding examples being the well-known Pavlovian conditioning experiments [5] and the research on trial and error conducted by Edward Thorndike. In years 1897-1898 Thorndike made a series of experiments on cats, dogs, and chickens that revolved around placing them inside a box with several mechanisms and providing them with discomfort in a form of hunger and feeling of confinement with the purpose of making the animal try to escape the box. What he has found was that cats and dogs that managed to escape the contraption would generally perform the same task faster depending on which consecutive attempt it was. Reducing time spent in cage for a sample cat from 160 seconds on the 1st try to just 7 seconds on the 24th. [6]

  
Fig. 2. "Puzzle-box" used by Thorndike in experiments on cats.

* Optimal control theory. The field has been greatly influenced by research of Richard Bellman who proposed a functional equation for solving dynamic optimization problems. [7] The function later became known as Bellman equation or dynamic programming equation [1].

The two subjects start interpolating with a rise of the method of Temporal Difference Learning. [8] The procedure was unconventional because instead of updating the policy based on the difference between the predicted and actual outcome, it did so, based on the new, more accurate prediction. This came after a research paper by Sutton and Barto [9] concerning conditioned response in animals appearing before the actual stimuli, very much like the Pavlov’s dogs starting to salivate without receiving the stimulus in the form of food [5].

What those two papers have in common is how both in animal conditioning and TD, the reaction is produced because of the prediction of future reward rather than the actual reward itself. Temporal Difference learning has been later used by Watkins in his PhD thesis to introduce Q-Learning. [10]

The next prominent step in reinforcement learnings history came with popularization of Deep Learning, a type of machine learning methods utilizing artificial neural networks, which in combination with contemporary reinforcement learning methods created Deep Reinforcement Learning. This new field proved invaluable in widening the range of decision-making tasks that were previously out of reach. [11] With major modern achievements including programs outcompeting human world champions in games like Go [12] or Dota 2 [13].

## Markov Decision Processes

Reinforcement learning tasks are modeled using Markov Decision Processes. MDPs are described using the components introduced in section 2.2.. Where is the set of all possible states, is the action space, describes the reward that is associated with transition from to and is the probability of a given action changing the state from to . It needs noting that and not always definite values, but rather often – random variables. Most modeled environments are stochastic, hence the probabilities in the decision process. To expand on a previous example: if an agent controlling the bicycle would decide that the best action to take would be to pull the break, it does not necessarily mean that the bike would stop – if the road were icy, it could start skidding. [2]

For any given state and action Sutton and Barto [1] denote the following *Transition model*:   
.  
The transition model fully describes the dynamics of the environment. Markov decision processes are a natural extension of Markov chains for modelling decision making. The model therefore has to satisfy the Markov Property, meaning that outcome of each particular action is not dependent on the outcome of any of the previous actions. This can be observed in the transition model since the next state and the reward depend solely on the state and action pair *)* [14] [1] [2].

We can take an example of a Martian rover exploring the red desert. For purposes of demonstration, I will provide a simplistic model that does not consider most problems of a space rover. The rover is equipped with wheels, a battery powering the wheels, and a mechanism which charges the battery when resting. The goal of the robot is to travel as far is can, however it needs to consider its battery which depletes when it is traveling. The battery recharges when the rover rests but if the rover decides to travel with a low battery, it may die. We can assume that the rover can be in three states:

,

and for each state it can perform the following actions:

,,.

The dead battery state is a terminal state, meaning that no action can be taken in it.  
If the battery has a high charge and the robot decides to travel there is a probability that the battery will go into low charge. If it decides to rest, then the charge will always stay high. If the battery is already in low charge, it can be recharged on rest with the probability . However, if the rover travels with a low battery charge there is a probability that it will completely run out of energy, rendering it inoperable. The reward should be representative to the number of meters the rover has travelled. I will denote it as and it will be available whenever the rover chooses to travel. If it is resting the reward will be 0 and if the battery dies the reward will be -100. A transition table can be produced in order to present the constructed model:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Table 1. Transition probabilities and expected rewards for the Markov decision process of the travelling rover.* | | | | |
|  |  |  |  |  |
| High charge | Low charge | travel |  |  |
| High charge | High charge | travel |  |  |
| High charge | Low charge | rest | 0 | 0 |
| High charge | High charge | rest | 1 | 0 |
| Low charge | High charge | rest |  | 0 |
| Low charge | Low charge | rest |  | 0 |
| Low charge | High charge | travel | 0 |  |
| Low charge | Low charge | travel |  |  |
| Low charge | Dead battery | travel |  |  |

*Table 1* depicts all valid combination of state , next state and possible action . For each combination there is a probability of transitioning to from given and the corresponding reward for such transition.

A good way of visualizing a Markov decision process is a transition graph. The big nodes are the possible states of the rover, I have decided to mark the terminal state in red. The small black nodes are possible actions from the action space of the state. The lines between small and big nodes are transitions, next to the transition the probability and reward are shown. It only depicts possible transitions; ones where .

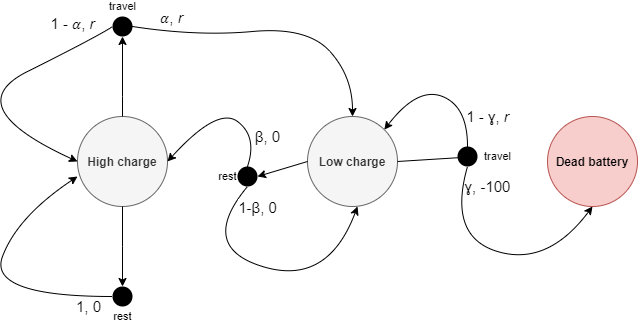


Fig. 3. Transition graph for the Markov decision process of the travelling rover

Markov decision process that also considers the observation of state is called *partially observable Markov decision process*. If action and state spaces of an environment are finite, then the process is called *finite Markov decision process.* This papers deals only with finite, fully observable Markov decision processes. [1]

## Exploration versus Exploitation

As in real life, in reinforcement learning an agent must first learn before it can successfully perform a task. This process of learning the environment is called **exploration**. If an agent is exploring the environment, then it is trying out actions to discover how they affect the environment and possible reward. It also lets the agent discover all the available actions. However, the agent will not achieve the goal of finding the optimal policy by simply stumbling around in the environment. The knowledge accumulated in the process helps the agent in the **exploitation**. In this phase the agent is acting upon its policy, exploiting what it knows about how the actions influence the environment. [1]

The challenge of balancing the exploitation and exploration has been fundamental to the optimization. [15] On one hand if the exploitation is favored without sufficient exploration, the agent will be selecting suboptimal actions without because of not discovering how beneficial the other actions may be. Alternatively, with too much time being spent on exploration the agent may not have the ability to act upon its knowledge in the short term since an explorative action can bring a negative reward.

### - Greedy policy

One of the simplest, commonly used solutions to the exploration/exploitation dilemma is stochastically exploring some of the time. Each timestep the agent would either choose the best-known action according to its policy or – with some probability – choose to perform a random action from the available action space.

A policy where the agent is only selecting the best next action is called ***greedy***, hence the name . This method has been proved to positively affect the agent’s performance [1]

Some argue that in the long run running after the agent has had time to discover all profitable actions is detrimental to the growth rate of the reward sum. Even if the agent is nearing the perfect policy, it still will be forced to choose potentially non-optimal action instead. One of the solutions would be to have decrease over time with a certain rate. This would result in less random actions the more time the agent had to study its environment. [16] [2]

# Q-learning

## Value Functions

From any state in a Markov decision process a value function can be calculated for a given policy . The value function describes the expected discounted cumulative reward an agent would obtain if it started in state and followed the policy until it arrived at a terminal state. The function can be written as follows:

Where is the discount rate – this parameter implies that the rewards received further into the future are less valuable than if they were received immediately. [1]

Similarly, we can define such a function for each state and action pair . A function which describes the expected cumulative reward starting in state , performing action and then following the policy is called the *action-value function for policy*, and is defined as [1]:

As mentioned in previous sections, reinforcement learning is concerned with finding the optimal policy for a given task. To discern which policy is better one just has to compare their value functions. If the expected discounted reward sum of a policy is greater than the expected discounted reward sum of then . A value function following the optimal policy is called the optimal value function. Similarly optimal policy will also have an optimal action-value function . We can then establish that the value function will be equal to the best action value function from that state [1] and with that and the previous equations, it can the following recursive function can be constructed:

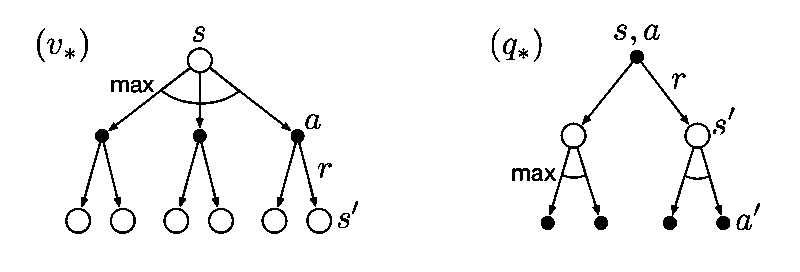


Fig. 4. Backup diagrams of the optimal value and optimal action-value functions. Backup diagrams are used to summarize algorithms and the relation of the parameters described in it. [1]

## Temporal Difference Learning

Temporal difference is in heart of most popular reinforcement learning algorithms. As briefly mentioned in section 2.3 Temporal Difference learning is what connected the two history roots of the field. The general idea behind TD is to rely on new estimates instead of waiting to discover the real value. To describe this further we will have to consider an estimate where is a set of all non-terminal states.

The general form of the update rule for an estimate is:  
 .

If we were to traverse the whole Markov decision process, for example using a Monte Carlo method we could describe the estimate updating as:

Where is the learning rate – parameter determining how much does the new information override the estimation. Please note that, as described in section 3.1, is the discounted sum of all rewards, meaning that the whole chain has to be traversed and all the rewards discovered in order to calculate its value.

In Temporal Difference Learning, instead of using the actual value of , we simply use the estimate of the next state . The formula for the simplest temporal difference method – [17]– is:  
This way the estimate can be updated after the episode ends but instead – every single step [1].

## Sarsamax

Sarsamax or Q-learning is a reinforcement learning algorithm first introduced by Christopher Watkins in his PhD thesis entitled “Learning from delayed rewards.” [10] Q-learning is a Temporal Difference Learning algorithm that utilizes the idea of following the optimal action value function from section 3.1. It follows an update rule similar to that of from section 3.2. Each step the estimate of the value function is being updated using the prediction of the value of the next step. Since the algorithm follows the optimal policy, the next chosen action will always be the one that provides the most valuable [18] [2] [1]

Where:

* – represents the learning rate, it affects the effect of the new prediction on the previous estimation.
* – represents the reward after performing the action in state .
* – is the discount factor. If its smaller than one the then rewards received later are valued less than those received earlier. [19]
* is current, non-terminal state.
* – represents the estimate action value function for the next state with the most optimal action.

Fig. 5. Backup diagram representing the Q-learning. The alternative name, Sarsamax, was later constructed from the elements of the algorithm, visible clearly on this diagram.

## Q-table

In a standard implementation of the Q-learning algorithm a lookup table for the values is created. Such a matrix containing estimates of the action value function for all possible and pairs is called a Q-Table. [20] Values in a q-table can be initialized using 0’s to imply that there is no known value for any pair. Alternatively, it can be filled out with unnaturally high values – those are called “optimistic initial conditions” [1]. The latter method encourages exploration since actions that have already been chosen will always have smaller values than those still in the starting state.

An initial Q-Table for the Martian rover example could look like this:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  | 200 | 200 |
|  | 200 | 200 |
|  | 200 | 200 |

Now the Q-learning formula can be followed. We are assuming that the rover starts in a state and decides to . The reward, selected at random, was equal to 20 and the agent is now in . The Q-table can be updated using that information. We can set the learning rate to 1 and the discount factor to 0.5 for ease of calculation. We can assume that out of 2 available action from state, the rover will choose the first one, since both have an estimated value of 200.

The new estimated value for is 120, we can alter the Q-Table accordingly:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  | 120 | 200 |
|  | 200 | 200 |
|  | 200 | 200 |

Next time the rover is faced with a state action pair it will assume that the best policy is to . The obvious limitation to the Q-table is that it can only hold discrete states and actions. If the process were continuous – either discretization would have to be used, or a different algorithm.

# OpenAI

## Introduction

OpenAi is an organization that focuses on AI research. It was created in December 2015 by a group of AI researchers and entrepreneurs including Elon Musk, Sam Altman, Ilya Sutskever and more. Together, the investors managed to accumulate a funding of 1 billion dollars. [21] It started as a non-profit research company trying to develop an ethical AI solutions that would serve everybody. It has since developed several products both open-source and proprietary.

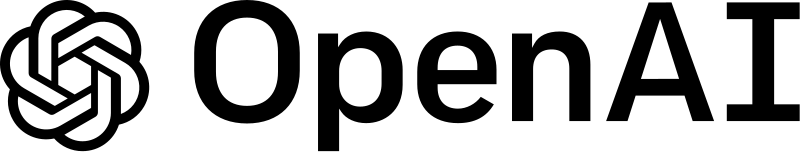


Fig. 6. OpenAI logo as of May 2021

## Mission

OpenAI’s mission is ensuring that artificial intelligence serves all of humanity. To meet this goal the organization intends to develop and help advance beneficial autonomous systems with its long-term safety in mind. The founders were concerned that the AI race may become so competitive that some could be willing to cut corners when it comes to safety precautions. They pledge that if some organization with similar views will come close to creating an artificial general intelligence before OpenAi, the company will stop competing against them and instead start supporting their project to ensure that no conflict of interest would jeopardize the systems’ safety. [22]

## History

OpenAi published its introductory post on December 11th, 2015 [21] and soon after started attracting talents from all over the world. The first project released to the public was the beta of OpenAi Gym. The intent was to help the reinforcement learning community by providing standardization and diversity of environments so that future researchers can reliably compare their results. [23]Through the years they pursued the field with several popular milestones like writing a bot that beats professional players of DOTA 2 [13], a competitive online multiplayer RTS game, or developing a novelty language model based on Transformers and unsupervised learning. The model was later succeeded by well-known GPT-2 [24] and GPT-3 [25], with the latter boasting 175 billion parameters. By this time, however, not all solutions created by OpenAi were open source, in March 2019 OpenAi LP was created with the parent company introducing is as follows:

*“We’ve created OpenAI LP, a new ‘capped profit’ company that allows us to rapidly increase our investments in compute and talent while including checks and balances to actualize our mission.”* [26]

Under the LP they licensed GPT-3 to over 300 companies, including Microsoft. The company assures that nothing it does should interfere with their core vision of bringing about advanced, safe AI [27]. Nevertheless, the decision has been met with a great deal of criticism with many a researcher accusing OpenAI for *“going back on its mission”* [28]by restricting access to technology of GPT-2 to the highest bidders.

# OpenAI Gym

## What is Gym?

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It provides a set of standardized problems which can be solved with the use of reinforcement learning. The problems, referred to as *Environments*, encapsulate a simulation with an agent. The toolkit exposes a concise interface that wraps the environment allowing for an agent to easily influence it. After each tick, the interface can provide us with the following: The current state of the environment – the representation of which varies, the latest reward obtained by the agent, information on whether it is time to end the current episode, and diagnostic information that can be used to further understand what happened during the last step. The agent can make use of the first three to then provide an action that will influence the environment in the next step. [29] [30]

## Environments

Gym boasts a considerable number of premade environments.

The environments registry can be found in *gym.envs.registry*, import gym and run the snippet below to see it.

>>> from gym import envs

>>> print(envs.registry.all())

### Algorithms

Simple computations or logic actions such as addition or reversing of symbols provided by the environment. While such a task would be trivial for a computer, the goal is to have the agent learn by examples, very much like a person would.

An example would be “Reverse-v0”. The task is to take a sequence of symbols from an input tape and output it in reverse. The input tape ends with a special character. The agent has to learn to move the selection right until it hits the special character and then to move left, copying each symbol to the output tape.

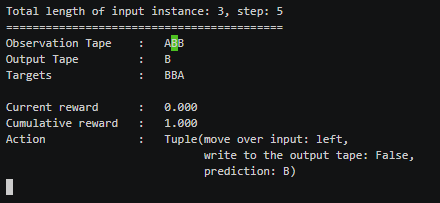
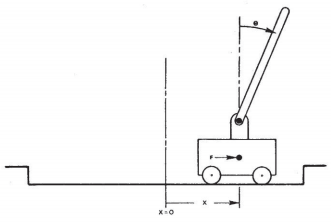


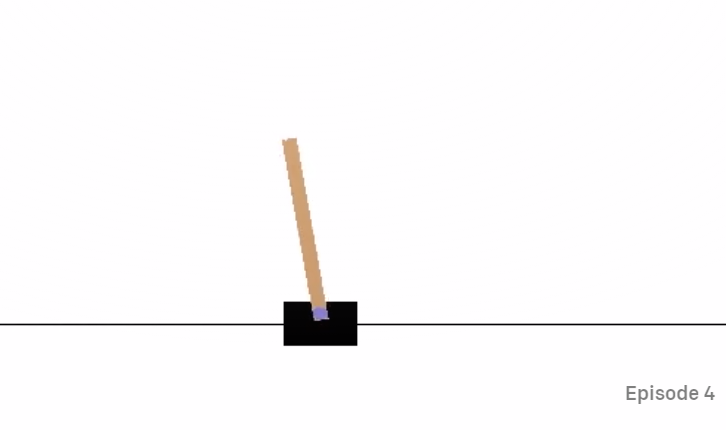
Fig. 7. An example step in the "Reverse-v0" environment.

### Classic control

“Classic control” is set of reinforcement learning problems previously proposed in literature.

Notable examples include the cart-pole problem, also known as the inverted pendulum, which has been used as a benchmark in multiple papers [31]. This particular simulation is based on the version of the cart-pole problem defined in *“Neuronlike adaptive elements that can solve difficult learning control problems”* [32] .

  
Fig. 8. The cart-pole problem as presented in [32].

  
Fig. 9. A visualization of the cart-pole environment in OpenAI Gym.

### Atari

This is an integration of the Arcade Learning Environment [33], numerous Atari 2600 game emulations can be found in this package with all offering the on-screen image as the environment’s observation. An agent performing in such an environment would be no different to a person playing the very same game on the console, that is, having visual observation as an input and pressing appropriate buttons as actions. Every environment comes in two versions, in one the observation is the RGB image of the screen but in the other – the observation of the environment is the actual RAM of the simulated Atari console. The Atari’s ram consisted of 128bytes which is a miniscule amount compared to personal computers we use today. It is still however an imaginable number if we were to imagine bit arrangements as states in a Markov decision process. These tasks are especially interesting for deep learning researchers since several of the emulated games were used in Deep mind researchers’ paper “*Playing Atari with Deep Reinforcement Learning*” [34].

A good example would be the Breakout-v0 environment. The agent controls a pallet that can move either left or right, or not move at all. The goal is to hit the incoming ball in such a way that it returns to one of the blocks on top destroying it and resulting in points. The observation is either an array of shape (210, 160, 3) – representing the on-screen image – or the 128 bytes of ram.

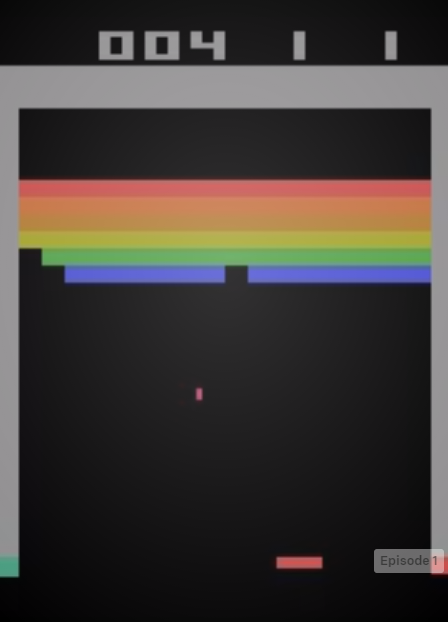


Fig. 10. Breakout-v0's on-screen image.

### MuJoCo and Robotics

Gym lists MuJoCo category as their environments of choice for continuous control tasks. It takes the name after a physics simulation engine created by Emo Todorov for the Movement Control laboratory of the University of Washington. MuJoCo is an acronym of Multi-Joint dynamics with Contact. The engine is now owned by Roboti LLC. [35]

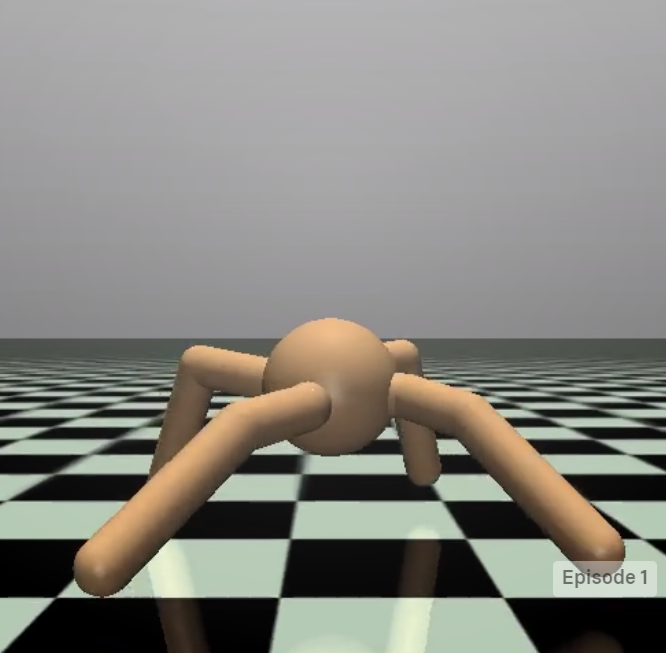


Fig. 11. An image from the 'Ant-2' environment rendered in the MuJoCo engine. The task modeled here has originally appeared in [36].

The environments take advantage of MuJoCo’s fast engine to simulate agents’ limbs as they try to move with the highest velocity. The engine is also used in the Robotics category environments dedicated to, as the name suggests, robot control. The tasks under this category are goal based and include manipulating a robot arm to interact with a block or manipulating a robotic hand to position its digits or objects held within it in a desired position.

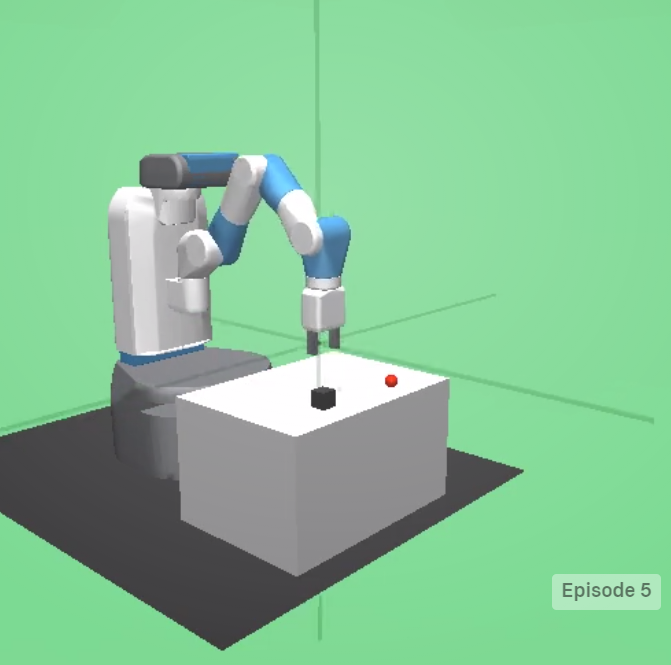


Fig. 12. An image from the 'FetchPush-v1' environment where the robot has to push the black box into a goal position.

While the MuJoCo’s physics engine may be accurate, unlike OpenAI Gym it is not freeware. In order to use the product for research a license has to be bought from the owners [37] [38]. However understandable, this may come as a great hinder to many researchers who might not be able to receive funding for their projects.

## Interface

Gym provides a simple interface for their environment. It is entirely based on the components of reinforcement learning process described in section 2.2.

Having started an environment *env*, an order to perform an action can be issued with the following method:

observation, reward, done, info = env.step(action)

The action is performed, provided that it matches the environment’s action space, and in turn the environment returns four variables:

* Observation: *List* - agent's observation of the current environment
* Reward: *Float* – the value that has been awarded to the agent after the performed action
* Done: *Boolean* – specifies whether the environment is ready to be reset. Returned as true in cases like the agent arriving at its final destination or failing beyond fixing.
* Info: *Dict* – diagnostic information on the environment, irrelevant from the algorithms point of view.

Gym also directly uses the concept of *Spaces.* Those describe the format of input and output of the environment, namely the actions and observations. Each environment has an action space and an observation space, which can be referenced in code by: *env.action\_space* and *env.observation\_space.*OpenAI’s documentation presents the following example. [29]

import gym

env = gym.make('CartPole-v0')

print(env.action\_space)

#> Discrete(2)

print(env.observation\_space)

#> Box(4,)

From the snippet we can learn that environment “CartPole-v0” has two discrete actions. The values are following non-negative number so for this example they will be 0 and 1. The observation space of the cart pole environment is a *Box(4,)*. A *Box* represents an array so in this case the observation will be a list of 4 elements.

# Unity ml-agents

## Unity Technologies

Unity Technologies is a software development company most known by its flag product called Unity – a game engine which it has been developing since 2005. Over the years, the engine has grown from simple 3D rendering tool to cross-platform toolkit capable of producing 2D and 3D games, physical simulations, VR and AR experiences and much more. The Editor that comes with the engine allows for scripting in C#, but also uses custom node-based visual scripting. For these reasons Unity has become one of the most popular tools in the gaming industry, especially for smaller studios [39].

It has over 80 different case studies listed on their site [40] created by companies and organizations from a range of industries including: Gaming, Engineering, Automotive, Film and more.

The Unity Game Engine has several paid licenses with the more expansive ones providing the most benefits. However, Unity lists a ‘personal license’ – a free version of Unity eligible if revenue or funding was less than $100K in the last 12 months. It also proposes a student plan meaning that it is a great tool for unfunded research [41].

The team behind Unity wants to empower everyone with a universal kit that will let them focus on their goal rather than on the essential tools. This is an important idea for the gaming industry, since prior to the popularization of software development kits such as Unity, most game developers had to start by writing their own game engine. This proved more difficult as the industry moved towards the 3D games with more and more people not being able to share their idea through this medium because of their lack of skills, funding, or time. [42]

## Machine learning with Unity

The Unity Machine Learning Agents is a toolkit that empowers the creation of games and simulation for the purpose of using them as environments for intelligent agents. It comes with a number of sample environments prepared with sample machine learning solutions that could utilize them. Alternatively, new solutions can be tested against those either by creating policies for the agents inside the software development kit or by utilizing a provided python API [43]. The toolkit first appeared in Beta on Sep 19, 2017, and as of April 21, 2021 it is on its 16th stable release. The project is open source and has a growing community. The goal behind its creation was to advance AI research and to let game developers utilize AI solutions to further benefit their games.

*“The ML-Agents Toolkit is mutually beneficial for both game developers and AI researchers as it provides a central platform where advances in AI can be evaluated on Unity’s rich environments and then made accessible to the wider research and game developer communities.”* [43]



Fig. 13. Picture from ML-Agent’s documentation, it presents numerous assets used in its example environments.

The creators recognized that the Unity Game Engine would be an outstanding platform for research as it can provide accurate physics simulation, realistic visuals and much more. By using the ML-Agents toolkit researchers get access to an easily configurable platform that has proved to solved problems that many contemporarily used tools face. The comparison and prospects for Unity as a general platform for AI research has been acknowledged in a paper prepared by Unity’s team [44] which has been cited almost 300 times to date.

The toolkit includes a C# software development kit to be used in pair with Unity Editor by importing it through the Unity Package Manager, this makes it ready to use almost out of the box. [45]

## Environments

The ML-Agents package comes with 17 example environments, all of which have their own setups and can be solved in different ways. The examples include motion control, remembering, learning algorithms, multi-agent environments and more. Most researchers should find a benchmark environment here matching their project.

However, the great thing about ML-Agents is that the provided environments present only a part of what can be achieved with the toolkit. The documentation provides a thorough guide on creating new environments [46]. This gives people the freedom to design simulations fit to their research.

Notable examples [47] from the toolkit include:

### Crawler

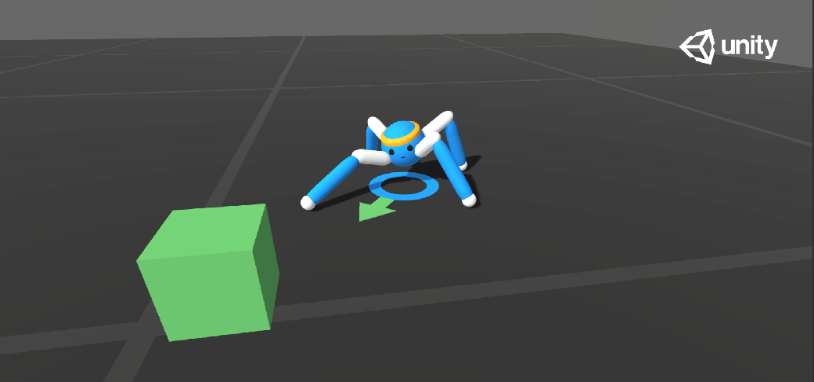


Fig. 14. An image from the Crawler environment showing the agent positioning itself towards the goal.

This environment contains a spider-like agent with 4 sets of limbs connected by two joints. The agent has to move towards a randomly selected goal and receives rewards for a product of its velocity towards the goal and the alignment of its head.

Observation space: 172 values describing the state of agent’s acceleration, its limbs, their velocity, angular velocities etc.

Action space: 20 continuous inputs corresponding to desired rotation of the joints.

### Sorter

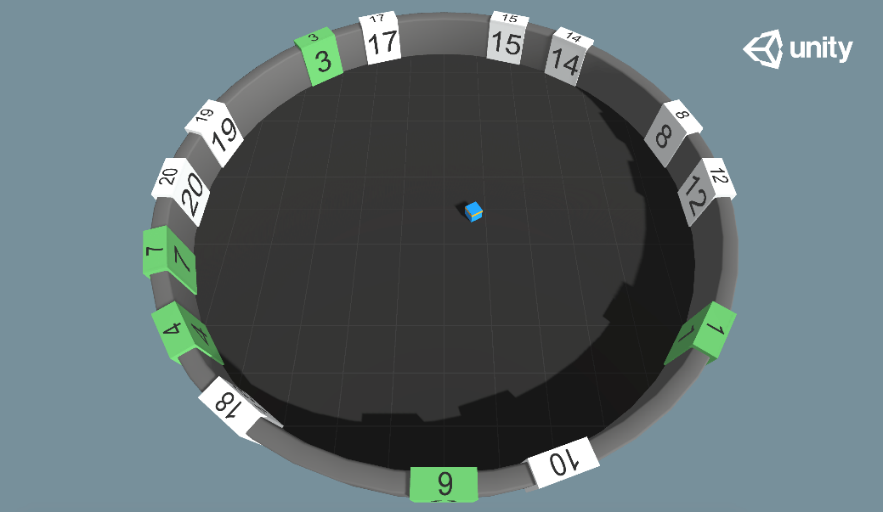


Fig. 15. An image from the Sorter environment with several numbers already marked as visited and the agent pointing towards the next one.

Agent is placed in a room with 20 randomly placed numbers, the goal is for the agent to learn to touch the numbered tiles in an ascending order. The numbers placement is randomized each episode and their tiles change color when touched. The reward is received for touching the correct tile, penalty is given if an incorrect tile was touched, there is also existential penalty.

Observation space: Values describing the agent’s position, the positions of tiles, their numbers and whether they have been touched already.

Action space: Inputs specify the agent’s movement and rotation.

### GridWorld

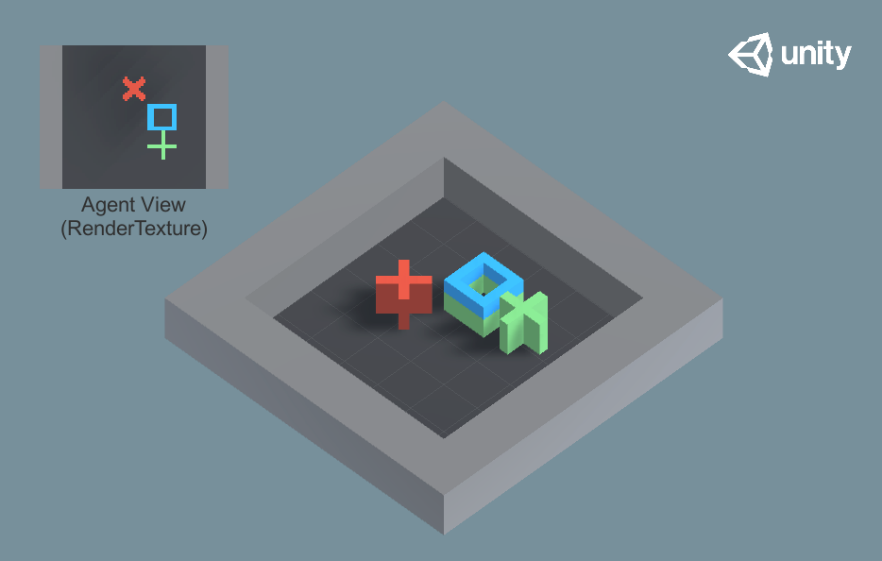


Fig. 16. A representation of the GridWorld environment with the visual observation received by the agent displayed in the top right corner.

This environment has the agent reach the goal without touching the obstacle. The name and the implementation of the environment suggest that the creators were inspired by the classic reinforcement learning task where the agent has to traverse a rectangular grid in the least amount of moves and reach a certain goal [1].

Observation space: An RGB rendering of the plan projection of the environment.

Action space: 5 possible actions representing movement in horizontal/vertical direction or not moving.

## Robotic simulation

Fig. 17 The simulated UR3 robotic arm and the object it is supposed to interact with.

Though not included in the base ml-agents package, the team behind it has created a simulation of Universal Robotics UR3e robot powered by Nvidia's PhysX 4 engine. The demo of the UR3 can be checked out from Unity technologies repository. From there it can be built as a unity executable and after that it should be ready to work with ml-agents like any other environment.

The agent would have the ability to rotate 6 joints using discrete values as well as open or close the pincher. The observation would be the rotations of joints and the position of an object. In the provided demo the goal of the agent is to touch the object.

This simulation is provided under Apache 2.0 license meaning that. unlike MuJoCo engine, this simulation would be free to use for research purposes. [48]

## OpenAI Gym integration

Unity technologies recognize that Gym by OpenAi has been used as benchmark for many years and that machine learning researchers have gotten used to interacting with simulation environments using the Gym interface. Because of that Unity ML-Agents includes a way to integrate their *UnityEnvironment* with the gym wrapper providing an interface for python scripting using the familiar gym nomenclature [49]. The wrapper is available after importing the *gym\_unity* package.

The solution, however, has some shortcomings. When used in pair with the gym wrapper, the environment can only be used by a single agent. This may not be a problem for those used to the OpenAI gym, but it is a big limitation of what the ML-Agents toolkit has to offer. To give another example: the *env.render()* method, which in Gym renders the next frame of the simulation onto the screen, in ML-Agents only returns the latest visual observation. The rendering is done by the environment simulation itself, if needed. This may come as counter intuitive to those only used to using the Gym [49].

# Q-learning implementation using ML-Agents

In order to demonstrate the ease of use of the ML-Agents Gym wrapper this paper addresses an implementation of Sarsamax algorithm written in python and run on an example environment from the ML-Agents toolkit.

## Basic environment

I have chosen the ‘Basic’ environment provided by the ML-Agents framework. The environment contains a single cube, representing an agent, and two green spheres that correspond to terminal states that I will refer to as ‘big reward’ and ‘small reward’. The cube has three possible action in every state, it can either move left, right, or not move at all. There are two rewards in the level: one small but close and one considerably bigger but a little further away.

The agent is placed in a 1-dimensional space that holds 20 positions. The agent starts at the eleventh position from the left while the small and big reward are on the positions 8 and 18, respectively.

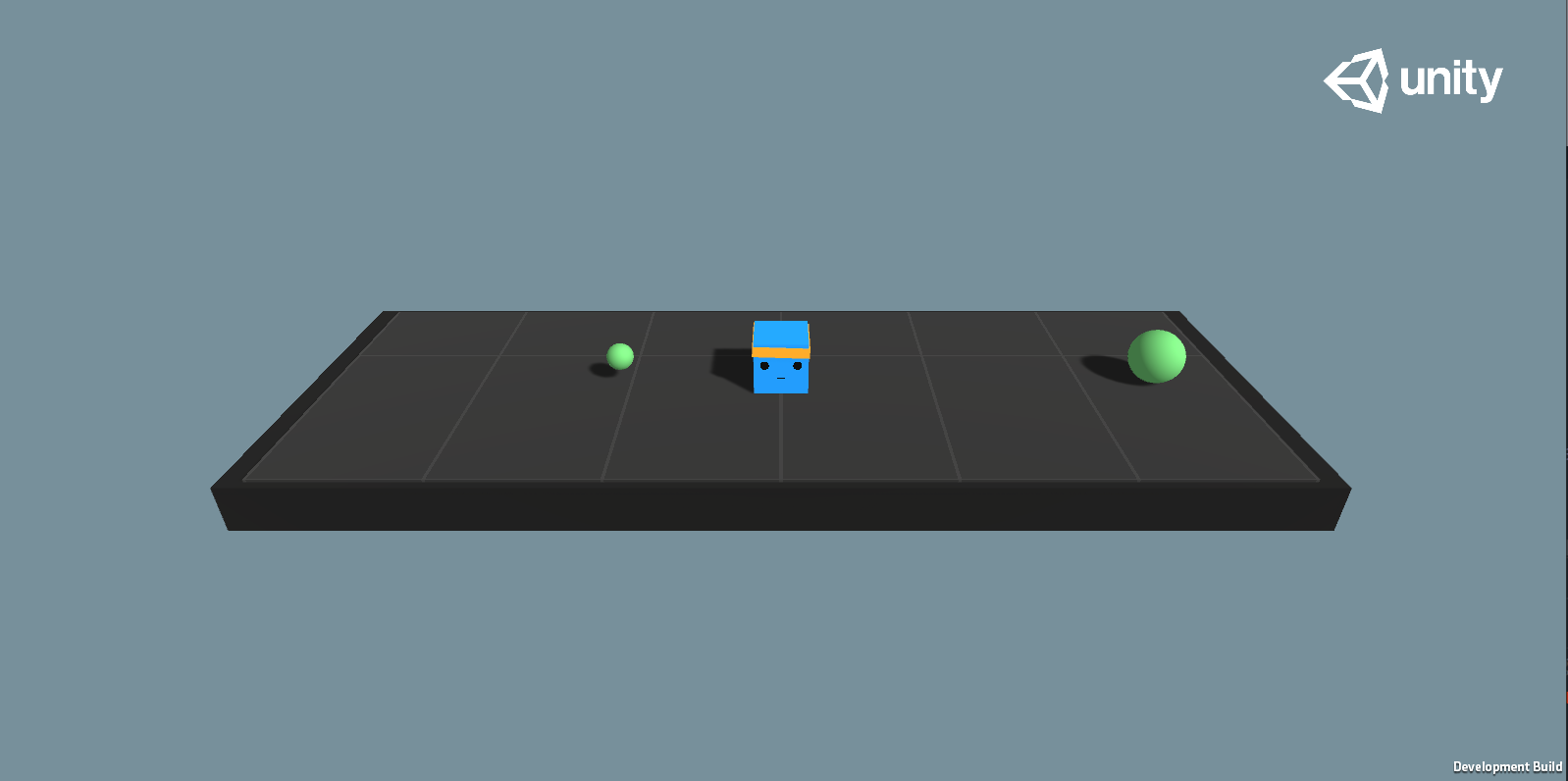


Fig. 18. View of the Basic environment in its initial state.

Reward function for the environment:

* +1.0 on touching the big reward.
* +0.1 on touching the small reward.
* -0.01 on every action.

The documentation [47] lists the benchmark mean reward as 0.93. According to this benchmark we should expect the agent to arrive at the bigger reward state after 7 steps.  
To achieve the lower reward however, the agent needs to only perform 3 steps. The path is more than 2 times shorter, but the reward is ten times smaller. This simple composition demonstrates one of the more challenging aspects of reinforcement learning – taking actions that will maximize long term rewards even if a different action seems better in the short term. [50]

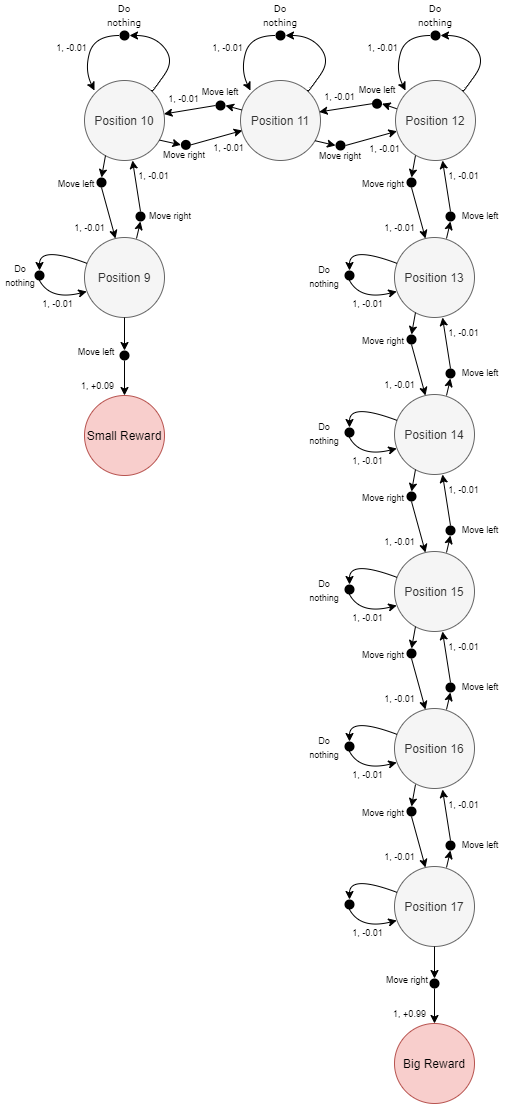


Fig. 19. Markov decision process representation of the Basic environment

## Implementation decisions

I have decided to use the one-step Q-learning with Q-table initialized with ones. As the benchmark sum of rewards is lower than 1, this optimistic initialization will encourage the agent to access all fields and should stop it from falling into the obvious local maximum of going for the near, small reward.

The script has been created using Python as it is the language that both OpenAI Gym and the ML-Agents gym wrapper were written in. This should come as no surprise since Python is commonly used in the field of machine learning due to its comfortable syntax and extensive list of libraries and packages prepared with machine learning and data science in mind [51].

## Experiments

After implementation I decided to run a series of experiments inclined towards tuning the algorithm to obtain the optimal policy for the task. In all runs I have decided to set the epsilon to decay with a rate of 0.975 between 1 and 0.01 and to run the experiment for 100 episodes. I assumed that the epsilon would be decreasing slowly enough for the agent to be able to test out all possibilities in the first half of the experiment and then confirm its policy by adjusting to random deviations in the second half of the run. Ultimately the epsilon should hit the minimum at the end of the run.

### Run 1 – The first run

For the first run I decided to set the discount factor and the learning rate arbitrarily, to their median values. This was more of an exploratory run to see how the agent will behave.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Table 2. Parameter summary of the 1st run.* | | | | | |
| Discount factor | **Learning rate** | **Beginning epsilon** | **Epsilon decay** | **Minimum epsilon** | **Number of episodes** |
| 0.5 | 0.5 | 1 | 0.975 | 0.01 | 100 |

The result of the first run was unsatisfactory. The Agent did not learn that in order to maximize the reward it should always go for the big reward to the right.

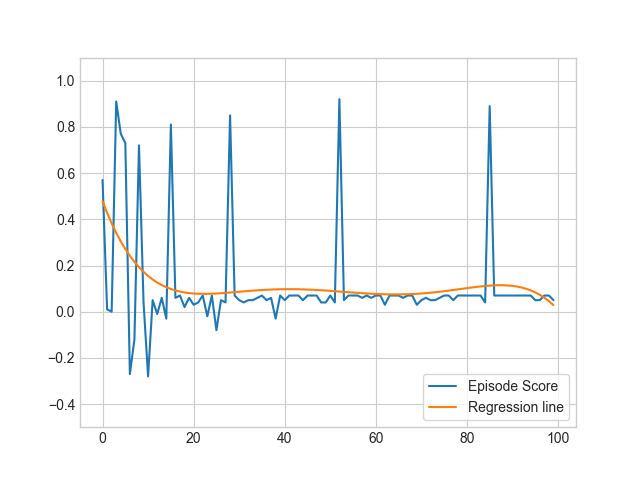
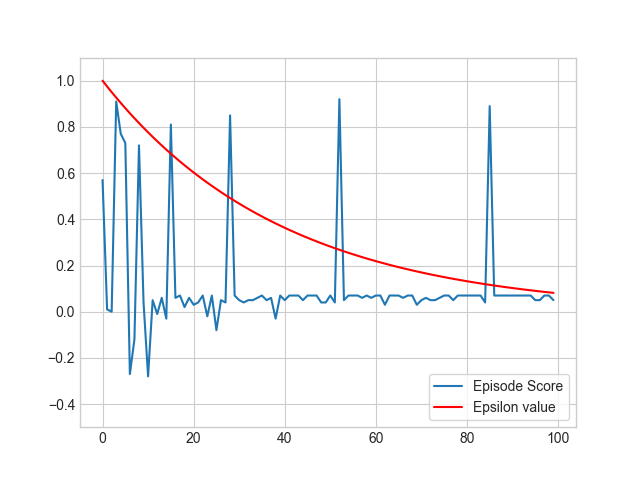


Fig. 20 Plot of episode scores for the 1st run.

We can see that there are occasional spikes in the scores around the 50th and 85th episode but the regression line shows us that the general trend is around the (0, 0.1) values, meaning that the agent was consistently going for the small reward.

|  |  |  |  |
| --- | --- | --- | --- |
| *Table 3. The Q-Table for the 1st run after 100 episodes.* | | | |
| State | **Action** | | |
| Do nothing | Move left | Move right |
| 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 |
| 3 | 1 | 1. | 1 |
| 4 | 1 | 1 | 1 |
| 5 | 1 | 1 | 1 |
| 6 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 |
| 9 | 0.28508811 | 0.59 | 0.13258446 |
| 10 | 0.13250502 | 0.285 | 0.05625006 |
| 11 | 0.05625018 | 0.1325 | 0.01812575 |
| 12 | 0.03845533 | 0.05625024 | 0.02967271 |
| 13 | 0.06011542 | 0.03401082 | 0.07315883 |
| 14 | 0.08611124 | 0.0509184 | 0.16479025 |
| 15 | 0.18033051 | 0.07904708 | 0.35336701 |
| 16 | 0.39244141 | 0.18670899 | 0.73364045 |
| 17 | 0.73945313 | 0.37557953 | 1.48904298 |
| 18 | 1 | 1 | 1 |
| 19 | 1 | 1 | 1 |
| 20 | 1 | 1 | 1 |

Fig. 21. Estimated values of state-action pairs for the 1st run after 100 episodes.

As we can see on the chart, the agent seems to think that going right is the optimal strategy only after reaching the 13th position. Since the cube is initially placed on the 11th position, if there is no exploitation involved, the agent will always arrive in the small reward terminal state. This seems to suggest that the agent has fallen for the local maximum of going to the closer reward.

### Run 2 – Higher discount factor

To solve this problem, we might try raising the value of the discount factor closer to 1 as we know that the higher the discount factor the more valuable the next estimated reward will seem to the agent [19].

For the 2nd run I have raised the value of the discount factor to 0.9, it is much higher than in the 1st one but there’s still room to raise it if needed. With higher discount factor, the estimated value of the predicted next state-action pair will be higher, meaning that the agent will be more ‘trusting’ towards its estimations.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Table 4. Parameter summary of the 2nd run.* | | | | | |
| Discount factor | **Learning rate** | **Beginning epsilon** | **Epsilon decay** | **Minimum epsilon** | **Number of episodes** |
| 0.9 | 0.5 | 1 | 0.975 | 0.01 | 100 |

After the second run it was clear that, even with an adjusted discount factor, the agent was still falling for the smaller reward.

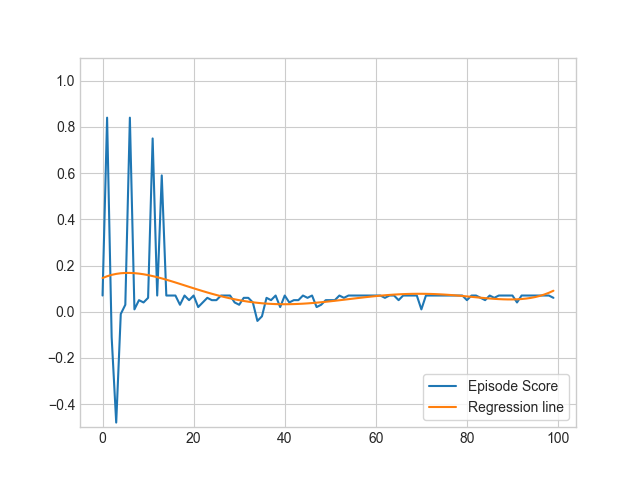
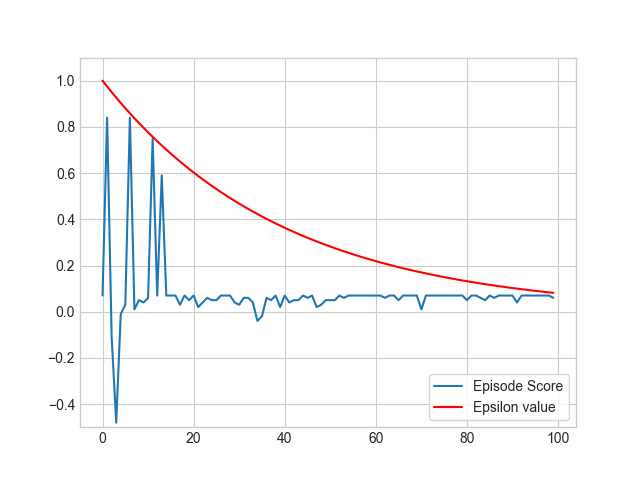


Fig. 22. Plot of episode scores for the 2nd run.

The trend shows that – after initial randomness caused by a high epsilon – the data settles to show a suboptimal average score of around 0.7.

|  |  |  |  |
| --- | --- | --- | --- |
| *Table 5. The Q-Table for the 2nd run after 100 episodes.* | | | |
| State | **Action** | | |
| Do nothing | Move left | Move right |
| 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 |
| 3 | 1 | 1. | 1 |
| 4 | 1 | 1 | 1 |
| 5 | 1 | 1 | 1 |
| 6 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 |
| 9 | 0.88100036 | 0.99 | 0.7829 |
| 10 | 0.78290003 | 0.881 | 0.694611 |
| 11 | 0.69461234 | 0.7829 | 0.615153 |
| 12 | 0.61603149 | 0.69461 | 0.637935 |
| 13 | 0.71320439 | 0.632919 | 0.690988 |
| 14 | 0.72388343 | 0.668002 | 0.913176 |
| 15 | 0.76279316 | 0.767574 | 1.217957 |
| 16 | 1.15191954 | 0.825375 | 1.584832 |
| 17 | 1.42253126 | 1.116313 | 1.862188 |
| 18 | 1 | 1 | 1 |
| 19 | 1 | 1 | 1 |
| 20 | 1 | 1 | 1 |

Fig. 23. Estimated values of state-action pairs for the 2nd run after 100 episodes.

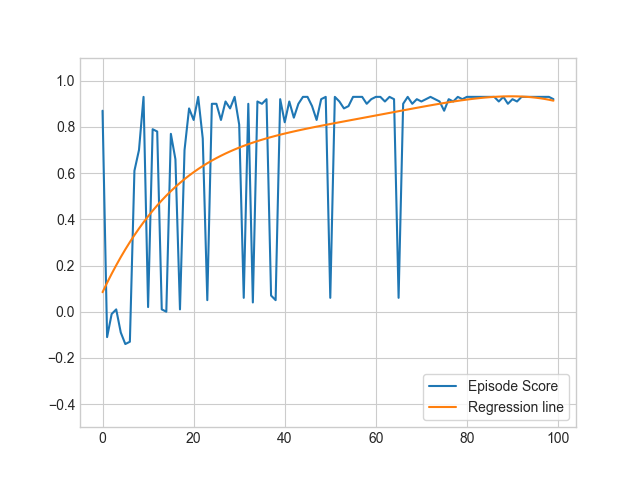
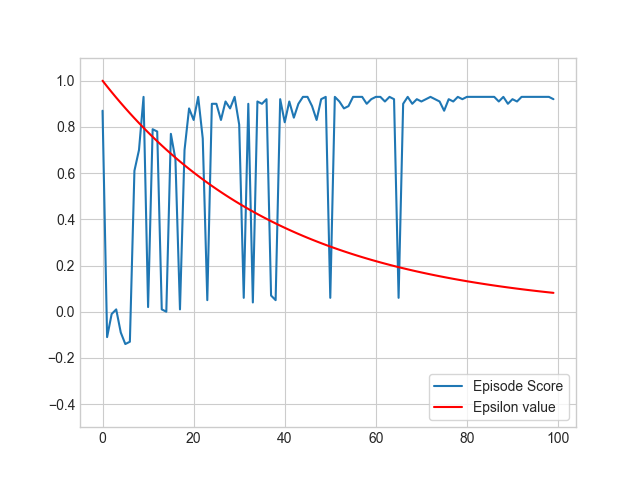
The plot shows a similar problem as in the 1st run. The agent starting at the position 11 still believes that going left is the best course of action. The switch between 12th and 13th position is less distinct this time with the agent for some reason considering not doing anything at the 13th spot.

### Run 3 – Higher discount factor and learning rate

In this experiment we will try altering the learning rate parameter. The learning rate is responsible for the dynamics of the learning process, putting more emphasis on the new information. I am hoping that this will make the big rewards impact more visible for the agent and that it will be propagated towards the initial state in the following episodes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Table 6. Parameter summary of the 3rd run.* | | | | | |
| Discount factor | **Learning rate** | **Beginning epsilon** | **Epsilon decay** | **Minimum epsilon** | **Number of episodes** |
| 0.9 | 0.9 | 1 | 0.975 | 0.01 | 100 |

This run has succeeded in creating the optimal policy for the ‘Basic’ environment.



The graphs presesnt that despite a turbulent start the mean score settles around the 0.97 benchmark. There seems to be a lot more of spikes caused by randomly deviating from the optimal path. This may be due to there being a shorter path to the small reward, in turn causing the agent to go for it after only a single step left in the beginning.

Fig. 24. Plot of episode scores for the 3rd run.

|  |  |  |  |
| --- | --- | --- | --- |
| *Table 7. The Q-Table for the 3rd run after 100 episodes.* | | | |
| State | **Action** | | |
| Do nothing | Move left | Move right |
| 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 |
| 3 | 1 | 1. | 1 |
| 4 | 1 | 1 | 1 |
| 5 | 1 | 1 | 1 |
| 6 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 |
| 9 | 0.881 | 0.99 | 0.78290378 |
| 10 | 0.7829 | 0.881 | 0.83905252 |
| 11 | 0.85181084 | 0.7829 | 0.9575676 |
| 12 | 0.9575676 | 0.85181084 | 1.07507511 |
| 13 | 1.07507511 | 0.9575676 | 1.20563901 |
| 14 | 1.20563901 | 1.07507511 | 1.35071001 |
| 15 | 1.35071001 | 1.20563901 | 1.51190001 |
| 16 | 1.51190001 | 1.35071001 | 1.69100001 |
| 17 | 1.69100001 | 1.51190001 | 1.89000001 |
| 18 | 1 | 1 | 1 |
| 19 | 1 | 1 | 1 |
| 20 | 1 | 1 | 1 |

Fig. 25. Estimated values of state-action pairs for the 3rd run after 100 episodes.

The graph presents a clear propagation of the estimated reward from the big reward step towards the initial, 11th position. With such estimates and agent following this policy in a deterministic ‘Basic’ environment should always choose the action moving it right.

It should be noted that if an agent would go from the 9th position towards the right most position, the sum of accumulated rewards would still far exceed the sum of rewards that the agent would achieve by going for the small reward. The truly optimal policy for any step in the ‘Basic’ environment would be to always take the action that moves the agent to the right. The possible explanation for current behavior could be that during the exploration the agent would always end up with the small reward after reaching the 9th position meaning that it did not have the ability to update its estimation for that position. Advised solution would be to extend the exploration possibilities of the agent.

### Equal distance to the rewards.

In the ‘Basic’ environment provided by Unity the agent has to learn to choose the reward even though it is twice as far as the small reward. This can lead to a situation where the agent gets stuck on the local maximum and decides to always go for the small reward.

However, if the goal were to check the very obvious assumption that the actor will always go for the bigger of two rewards then it could be more reasonable to prepare an environment that will check just that.

Unity provides us with a way to easily create new environments and alter existing ones. Using the intuitive UI, I was able to quickly alter the position of the rewards, their color, and add a description of this experiment.

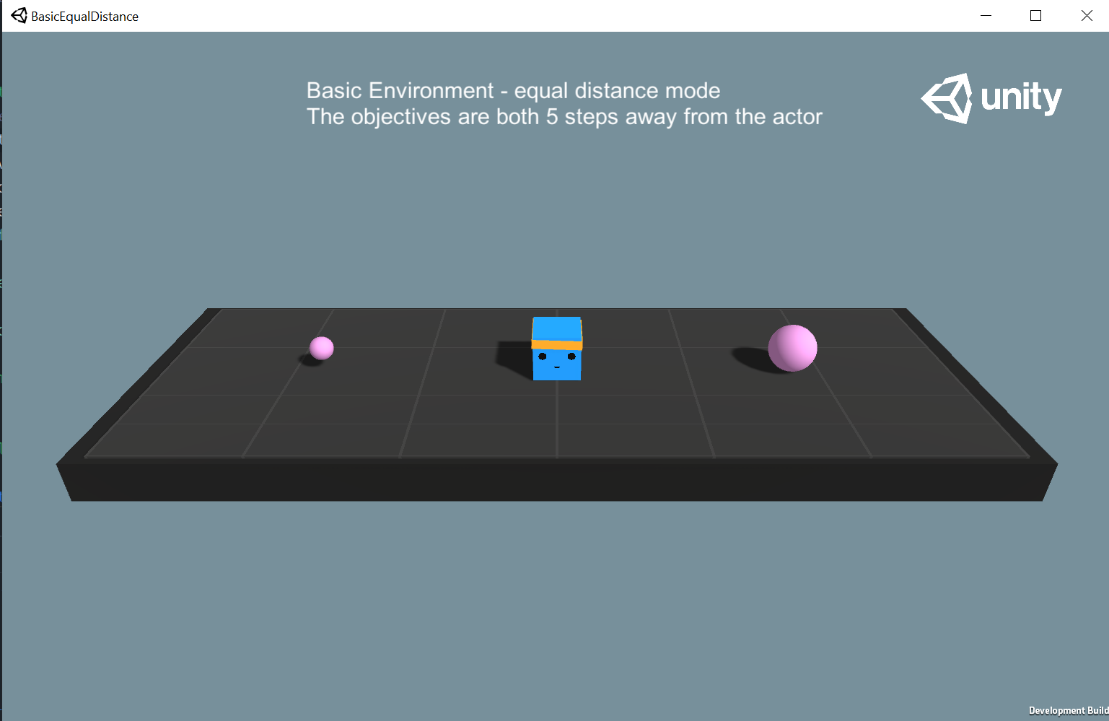
After building the executable I performed the Sarsamax training with the attributes from the initial run. The Q-table has been initialized with all ones and the training run for 100 episodes.

Fig. 26. The view of BasicEqualDistance environment in starting position.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Table 8. Parameter summary of the equal distance run.* | | | | | |
| Discount factor | **Learning rate** | **Beginning epsilon** | **Epsilon decay** | **Minimum epsilon** | **Number of episodes** |
| 0.5 | 0.5 | 1 | 0.975 | 0.01 | 100 |

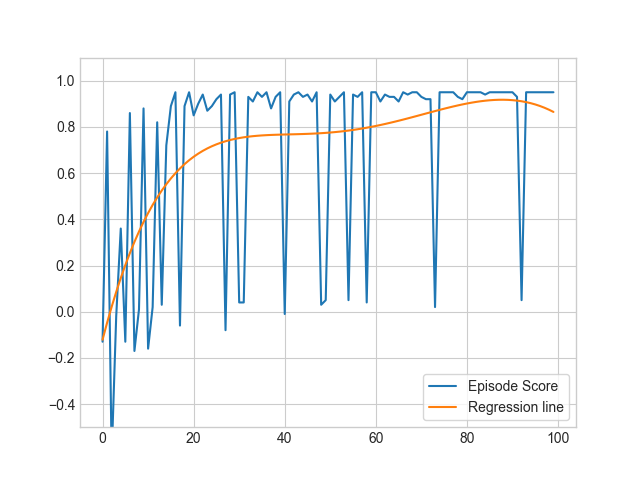
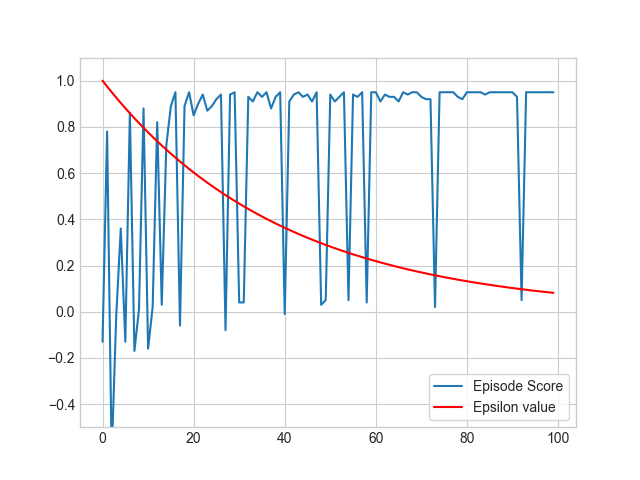


Fig. 27. Plot of episode scores for the equal distance experiment.

The learning seems to have been successful, the trend suggests that the agent is leaning towards the greater reward. However, we clearly see a lot of more big spikes in comparison to the standard version of the environment. This may suggest that even a single random action may cause the agent to set course for the small reward.

|  |  |  |  |
| --- | --- | --- | --- |
| *Table 9. The Q-Table for the equal distance experiment after 100 episodes.* | | | |
| State | **Action** | | |
| Do nothing | Move left | Move right |
| 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 |
| 3 | 1 | 1. | 1 |
| 4 | 1 | 1 | 1 |
| 5 | 1 | 1 | 1 |
| 6 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 |
| 9 | 0.29358216 | 0.59000039 | 0.14625148 |
| 10 | 0.13439615 | 0.28500186 | 0.05686854 |
| 11 | 0.05851969 | 0.13250375 | 0.03480714 |
| 12 | 0.01989113 | 0.05626456 | 0.03531049 |
| 13 | 0.02718097 | 0.01815523 | 0.074375 |
| 14 | 0.07393204 | 0.02717562 | 0.16875 |
| 15 | 0.16873376 | 0.07410427 | 0.3575 |
| 16 | 0.35741854 | 0.16868508 | 0.735 |
| 17 | 0.7349982 | 0.35747709 | 149000001 |
| 18 | 1 | 1 | 1 |
| 19 | 1 | 1 | 1 |
| 20 | 1 | 1 | 1 |

Fig. 28. Estimated values of state-action pairs for equal distance run.

The agent still starts at the 11th position, therefore if it were following the policy laid out in the Q-table presented in *Table 9* it would always arrive at the big reward state. Worryingly, but consistently with the score chart in *Fig. 27*, if the agent deviates from the path by taking a step left in the initial position it will end up on a straight course to the small reward.

# Conclusions

The implementation of the Sarsamax algorithm has proved successful and – after fine tuning the parameters through a series of experiments – the agent finds the optimal policy for solving the ‘Basic’ example environment form the ML-Agents toolkit. The implementation also worked as intended with an altered version of the environment, though there is room for improvement.

The integration of the Python implementation with the environments was nothing but straightforward. ML-Agents seems to be a great tool and the example presented in this paper does not do it enough justice. The Gym interface created by OpenAI gives a clear way of interacting with the environment; one that is familiar to people working with reinforcement learning theory. Finally, the Unity Editor, with which the ‘BasicEqualDistance’ environment was prepared (*section 7.3.4*), is easy to learn and fun to use, enabling swift development of models, simulations, and visual interpretations of environments.

The toolkit could grow into something even more useful once the developers provide full support of Robotic simulation. UR3 demo described in *section 6.4* is a great start but if it were integrated as an example environment and given more attention it could yield more benefit to the researchers. However, the greatest perk of the ML-Agents toolkit is the ease with which new environments can be created. Reinforcement learning researchers should definitely consider getting acquainted with the technology and using it to model their problems. A compelling argument might be the fact that ML-Agents is an open-source project, and the Unity engine has free license plans. I strongly believe that those qualities welcome more people into the field, especially from universities that may not be in position to fund projects requiring costly licenses.

## Further reading

For those curious about the ML-Agents toolkit I would recommend reading the “Unity: A General Platform for Intelligent Agents", 2018. In their paper the developers and researchers behind Unity and ML-Agents consider the problems ailing the reinforcement learning community and compare the abilities of Unity with other popular simulators such as MuJoCo or the Arcade Learning Environment. At the same time the paper goes into details on how the ML-Agents toolkit works and how it can be used effectively.

# References

|  |  |
| --- | --- |
| [1] | R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed., Cambridge, Massachusetts: The MIT Press, 2015. |
| [2] | P. Winder, Reinforcement Learning, O'Reilly Media, Inc., 2020. |
| [3] | K. J. Åström, "Optimal Control of Markov Processes with Incomplete State Information," *Journal of Mathematical Analysis and Applications,* vol. I, no. 10, pp. 174-205, 1965. |
| [4] | R. D. Smallwood and E. J. Sondik, "The Optimal Control of Partially Observable Markov Processes Over a Finite Horizon.," *Operations Research,* vol. 21, no. 5, p. 1071–1088, 1973. |
| [5] | I. Pavlov, Conditioned reflexes., London: Oxford University Press, 1927. |
| [6] | E. L. Thorndike, "Animal intelligence: An experimental study of the associative processes in animals.," *The Psychological Review,* vol. II, no. 4, 1898. |
| [7] | R. E. Bellman, Dynamic Programming, Dover, 1957. |
| [8] | R. S. Sutton, "Learning to Predict by the Method of Temporal Differences," *Machine Learning,* vol. 3, pp. 9-44, 1988. |
| [9] | R. S. S. Andrew G. Barto, "Simulation of anticipatory responses in classical conditioning by a neuron-like adaptive element," *Behavioural Brain Research,* vol. 4, no. 3, pp. 221-235, 1982. |
| [10] | C. Watkins, Learning from Delayed Rewards, King's College, 1989. |
| [11] | V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare and J. Pineau, "An Introduction to Deep Reinforcement Learning," *Foundations and Trends® in Machine Learning,* vol. 11, no. 3-4, 2018. |
| [12] | D. Silver, A. Huang, C. Maddison and e. al., "Mastering the game of Go with deep neural networks and tree search," *Nature,* p. 484–489, 2016. |
| [13] | OpenAI, Dota 2 with Large Scale Deep Reinforcement Learning, 2021. |
| [14] | W. Feller, "The Exponential Density," in *An Introduction to Probability Theory and Its Applications, vol 2*, New York, John Wiley & Sons Inc., 1971, pp. 9-19. |
| [15] | M. Tokic, "Adaptive ε-greedy Exploration in Reinforcement Learning Based on Value Differences," *Lecture Notes in Artificial Intelligence,* pp. 203-210, 2010. |
| [16] | P. Auer, N. Cesa-Bianchi and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine Learning,* vol. 47, no. 2-3, pp. 235-256, 2002. |
| [17] | I. H. Witten, "An adaptive optimal controller for discrete-time Markov environments," *Information and Control,* vol. 34, no. 4, pp. 286-295, 1977. |
| [18] | H. v. Hasselt, Reinforcement Learning: State-of-the-Art, Springer Science & Business Media, pp. 207-251. |
| [19] | V. François-Lavet, R. Fonteneau and D. Ernst, "How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies," 2015. |
| [20] | C. Gaskett, D. Wettergreen and A. Zelinsky, "Q-Learning in Continuous State and Action Spaces," in *Advanced Topics in Artificial Intelligence*, 1999. |
| [21] | OpenAI, "Introducing OpenAI," 11 December 2015. [Online]. Available: https://openai.com/blog/introducing-open. [Accessed 25 May 2021]. |
| [22] | OpenAI, "Charter," 9 April 2018. [Online]. Available: https://openai.com/charter/. [Accessed 25 May 2021]. |
| [23] | OpenAI, "OpenAI Gym Beta," 27 April 2016. [Online]. Available: https://openai.com/blog/openai-gym-beta/. [Accessed 25 May 2021]. |
| [24] | A. Radford, J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, Language Models are Unsupervised Multitask Learners, 2019. |
| [25] | OpenAI, "Language Models are Few-Shot Learners," 28 May 2020. |
| [26] | OpenAI, "OpenAI LP," 11 March 2019. [Online]. Available: https://openai.com/blog/openai-lp/. |
| [27] | OpenAI, "GPT-3 Powers the Next Generation of Apps," 25 March 2021. [Online]. Available: https://openai.com/blog/gpt-3-apps/. [Accessed 25 May 2021]. |
| [28] | MIT Technology Review, "The messy, secretive reality behind OpenAI’s bid to save the world," 17 February 2020. [Online]. Available: https://www.technologyreview.com/2020/02/17/844721/ai-openai-moonshot-elon-musk-sam-altman-greg-brockman-messy-secretive-reality/. [Accessed 27 May 2021]. |
| [29] | OpenAI, "OpenAI Gym Documentation," 2016. [Online]. Available: https://gym.openai.com/docs/. [Accessed 24 March 2021]. |
| [30] | OpenAI, "OpenAI Gym whitepaper," 2016. [Online]. Available: https://arxiv.org/abs/1606.01540. |
| [31] | M. Riedmiller, J. Peters and S. Schaal, "Evaluation of Policy Gradient Methods and Variants on the Cart-Pole Benchmark," *IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning,* pp. 254-261, 2007. |
| [32] | A. G. Barto, R. S. Sutton and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics,* Vols. SMC-13, no. 5, pp. 834-846, 1983. |
| [33] | M. G. Bellemare, Y. Naddaf, J. Veness and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *Journal of Artificial Intelligence Research,* vol. 47, 2013. |
| [34] | DeepMind Technologies, "Playing Atari with Deep Reinforcement Learning," 2013. |
| [35] | Roboti LLC, "MuJoCo advanced physics simulation," Roboti LLC, 2018. [Online]. Available: http://www.mujoco.org/index.html. [Accessed 26 May 2021]. |
| [36] | J. Schulman, P. Moritz, S. Levine, M. Jordan and P. Abbeel, "High-Dimensional Continuous Control Using Generalized Advantage Estimation," *ICLR,* 2015. |
| [37] | OpenAI, "mujoco-py," 2020. [Online]. Available: https://github.com/openai/mujoco-py#obtaining-the-binaries-and-license-key. [Accessed 21 May 2021]. |
| [38] | RobotiLLC, "MuJoCo license," 2018. [Online]. Available: https://www.roboti.us/license.html. [Accessed 27 May 2021]. |
| [39] | gameindustry.biz, "What is the best game engine: is Unity right for you?," 2020. [Online]. Available: https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you. [Accessed 27 May 2021]. |
| [40] | Unity Technologies, "Unity case studies," [Online]. Available: https://unity.com/case-study. [Accessed 27 May 2021]. |
| [41] | Unity Technologies, "Choose the plan that is right for you," 2021. [Online]. Available: https://store.unity.com/compare-plans. [Accessed 27 May 2021]. |
| [42] | Unity Technologies, "Make games - not tools," Unity Technologies, 2017. |
| [43] | Unity Technologies, "ML-Agents," 2021. [Online]. Available: https://github.com/Unity-Technologies/ml-agents. [Accessed 27 May 2021]. |
| [44] | A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar and D. Lange, "Unity: A General Platform for Intelligent Agents," 2018. |
| [45] | Unity Technologies, "Announcing ML-Agents Unity Package v1.0!," 2020. [Online]. Available: https://blog.unity.com/technology/announcing-ml-agents-unity-package-v1-0. [Accessed 27 May 2021]. |
| [46] | Unity Technologies, "Making a New Learning Environment," 2021. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Create-New.md. [Accessed 27 May 2021]. |
| [47] | Unity Technologies, "Example Learning Environments," 2021. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Examples.md. [Accessed 27 May 2021]. |
| [48] | Unity Technologies, "Articulations Robot Demo," 29 July 2020. [Online]. Available: https://github.com/Unity-Technologies/articulations-robot-demo/tree/mlagents. [Accessed 25 May 2021]. |
| [49] | Unity Technologies, "Unity ML-Agents Gym Wrapper," 2021. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/gym-unity/README.md. [Accessed 27 May 2021]. |
| [50] | S. Levine, "Deep Reinforcement Learning lectures," [Online]. Available: http://rail.eecs.berkeley.edu/deeprlcourse-fa19/. [Accessed 21 05 2021]. |
| [51] | Springboard, "Best language for Machine Learning," 31 August 2020. [Online]. Available: https://in.springboard.com/blog/best-language-for-machine-learning/. [Accessed 29 May 2021]. |
| [52] | T. Nieuwdorp, "Dare to Discover: The Effect of the Exploration Strategy on an Agent's Performance," Radboud University Nijmegen, 2017. |

# Appendix

## How to create an environment Unity executable

In the Equal Distance experiment described in *section 7.3.4* I have used a modified version of the ‘Basic’ environment. I will try to briefly describe how it was done.

The ML-Agents GitHub repository [43] contains the example environments under ‘*ml-agents/Project/Assets/ML-Agents/Examples/*’. The folders within contain *scenes* which can be opened in the Unity Editor. Scenes are basically collections of assets that work together in a single instance.

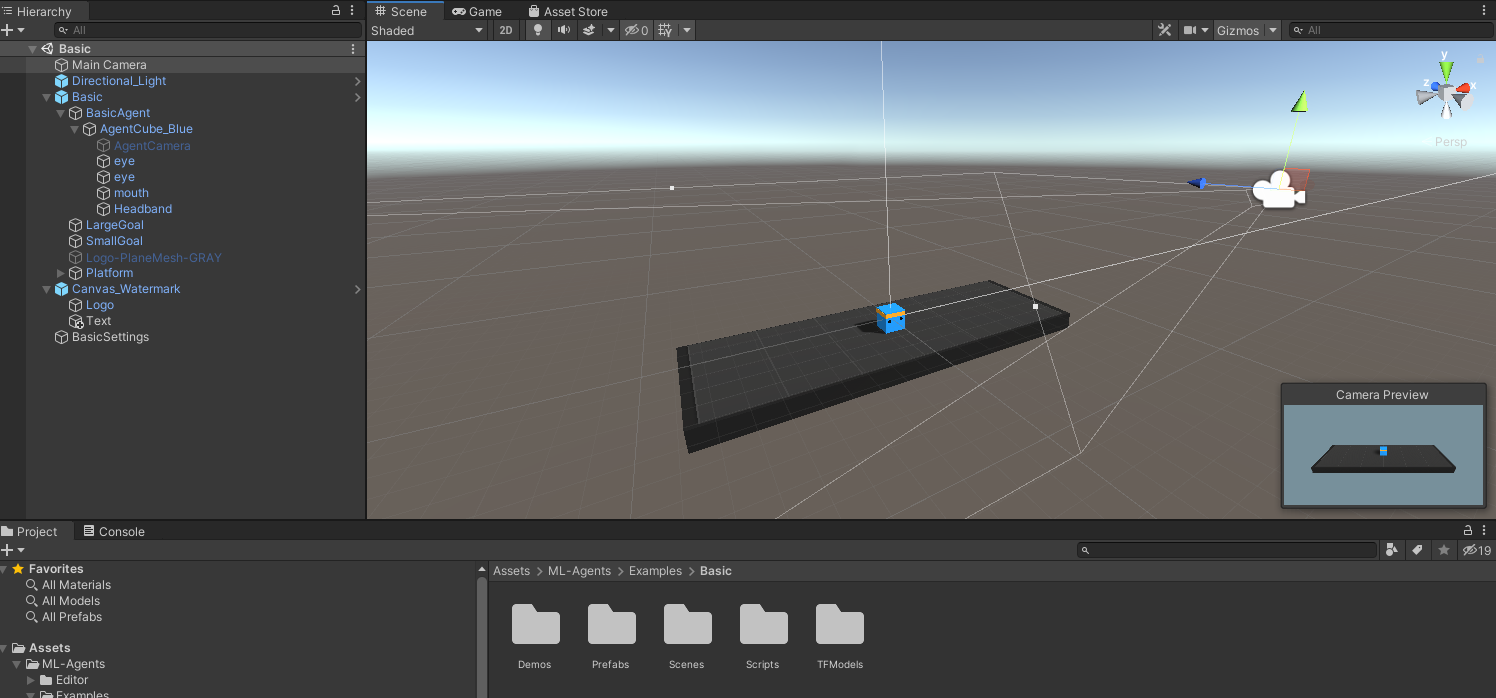


Fig. 29. A view of the scene for the 'Basic' environment in the Unity editor.

The behavior of the environment is described in three scripts:

* ‘*BasicActuatorComponent.cs’* – it is responsible for translating integer actions to actual movement of the agent.
* ‘*BasicSensorComponent.cs*’ – it creates the observation that is returned to the agent in a form of an array describing each position as either empty (0) or non-empty (1). Since there is only one actor, only one position will be marked with a 1.
* ‘*BasicController.cs*’ – this script is the backbone of the environment; it describes the actual behaviour of the agent and the rewards.

In order to have the rewards be an equal distance from the starting position of the agent, I needed to modify the ‘*BasicController.cs*’. The positions of the rewards are set as a class variables and the actual reward objects are placed in the *onEnable()* function.

public class BasicController : MonoBehaviour

{

...

const int k\_SmallGoalPosition = 5;

const int k\_LargeGoalPosition = 15;

public GameObject largeGoal;

public GameObject smallGoal;

...

The parameters for small and large goals positions are hardcoded into the script. I have changed them to 5 and 15, respectively. It should be noted that those are the 6th and 16th positions as the numbering starts at 0.

public void OnEnable()

{

position = 10;

transform.position = new Vector3(position - 10f, 0f, 0f);

smallGoal.transform.position = new Vector3(k\_SmallGoalPosition - 10f, 0f, 0f);

largeGoal.transform.position = new Vector3(k\_LargeGoalPosition - 10f, 0f, 0f);

}

This method is run at the start of the environment, it sets the agent’s position and places the small and large rewards in their specified spots. The placement of the *smallGoal* and *largeGoal* here is strictly visual. The actual check if the agent reached either of the rewards is done with the class variables.

public void MoveDirection(int direction)

{

position += direction;

...

if (position == k\_SmallGoalPosition)

{

m\_Agent.AddReward(0.1f);

m\_Agent.EndEpisode();

ResetAgent();

}

if (position == k\_LargeGoalPosition)

{

m\_Agent.AddReward(1f);

m\_Agent.EndEpisode();

ResetAgent();

}

}

Ultimately the only change needed was that to the *k\_SmallGoalPosition* and *k\_LargeGoalPosition*. If I wanted to change the rewards associated with the goals, I could have done it in the *MoveDirection* method.

The whole process was very simple and straightforward. The code for this example is not generously documented but with such a simple environment it is easy to understand using the short comments above some of the methods or classes.

Once we are satisfied with our environment, in order to run it outside of the editor it needs to be built. To build an environment we simply need to click *File > Build Settings*, mark the scene with our environment, select any additional needed settings and click ‘*Build’*.

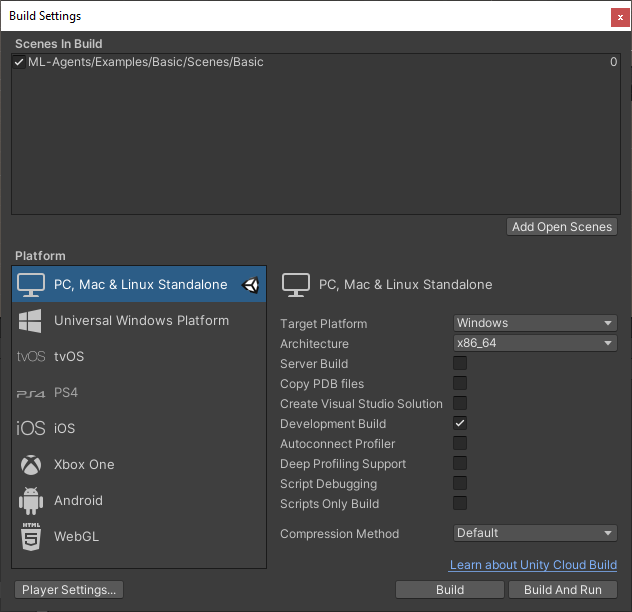


Fig. 30. The Build Settings window in Unity Editor.

The build directory for a project called ‘Basic’ looks as follows:

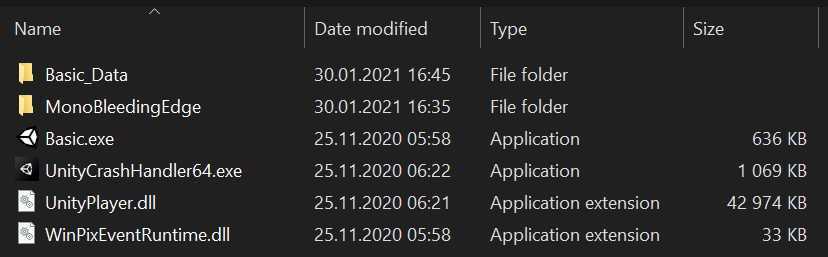


Fig. 31. The build directory containing all necessary files to run an executable with the scene.

*Basic\_Data* contains the assets and scripts that the scene consists of, as well as other resources necessary to run the *Basic.exe* – our environment executable. The other folder and files are required for Unity to run correctly and can be shared between different executables.

## Using OpenAI Gym interface with Unity executable

After creating a unity executable with our environment, we are able to import it into our runtime by creating an object of *mlagents\_envs.environment.UnityEnvironment* and wrapping it into a Gym with the help of *UnityToGymWrapper*from *gym\_unity.envs*.

### UnityEnvironment

On initiation, this object runs the unity environment executable under a provided path and establishes a connection between python runtime and the environment through an unsecured socket. By default, the connection is established on port 5005.

### UnityToGymWrapper

This is the actual method for wrapping the UnityEnvironment into a Gym interface. The utilization would look like this:

from mlagents\_envs.environment import UnityEnvironment

from gym\_unity.envs import UnityToGymWrapper

def main():

unity\_env = UnityEnvironment(<path-to-environment>)

env = UnityToGymWrapper(

unity\_env,

uint8\_visual = True,

flatten\_branched = True,

allow\_multiple\_obs = False)

* *unity\_env* – The unity executable containing the environment.
* *uint8\_visual –* A Boolean that specifies whether the visual observations should be provided in the form of integer values (0-255) like in the Gym environments, or if *False* - as float values (0.0-1.0). The latter is the default behavior.
* *flatten\_branched –* A Boolean that enforces flattening of a branched discrete action space into one that is seen in OpenAI Gym. Defaults to *False.*
* *allow\_multiple\_obs –* Boolean for enabling returning of a list of observations.