**TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY  
Control Robotics and Machine Learning Laboratory**





**Project A**

**Building Multi Agent RL Baselines in Unity**

**Students:**

Rotem Levinson Shahar Gottlieb

**Advisor:**

Krupnik Orr

Spring 2019

Table of Contents

[Abstract 3](#_Toc30601877)

[Background 4](#_Toc30601878)

[Machine Learning 4](#_Toc30601879)

[Reinforcement Learning 5](#_Toc30601880)

[DDPG 7](#_Toc30601881)

[MARL and MADDPG 10](#_Toc30601882)

[Reinforcement Learning Environments 12](#_Toc30601883)

[Unity 13](#_Toc30601884)

[ML-Agents-Toolkit 14](#_Toc30601885)

[Our project 15](#_Toc30601886)

[General idea 15](#_Toc30601887)

[Goals 15](#_Toc30601888)

[Project in details 16](#_Toc30601889)

[High level architecture 16](#_Toc30601890)

[Architecture in details 16](#_Toc30601891)

[Unity scene 16](#_Toc30601892)

[Unity scripts 17](#_Toc30601893)

[Python 19](#_Toc30601894)

[Test, Train and Main 19](#_Toc30601895)

[Agents 19](#_Toc30601896)

[Results 20](#_Toc30601897)

[Baseline Evaluation 20](#_Toc30601898)

[DDPG 20](#_Toc30601899)

[MDDPG 21](#_Toc30601900)

[MADDPG 22](#_Toc30601901)

[Summery 23](#_Toc30601902)

[Bibliography 24](#_Toc30601903)

[Appendix 25](#_Toc30601904)

[Usage and instructions 25](#_Toc30601905)

# Abstract

Reinforcement Learning is a branch of machine learning where an agent preforms in an environment and gets rewarded for his action. The agent goal is to learn a behavior policy that will maximize his reward over time.

Some of the research done in this field, is about expending RL to multi agent settings. The goal of this research is to solve problems where many agents preforms in the same environment, at the same time, and each of them try to maximize his own reward. This type of problems raises challenges that don't exist in single agent problems and requires different solutions.

Today, the development and testing environments for multi agent reinforcement learning are scarce, and the existing environments are ill-suited for solving some interesting problems.

In this project, we developed a new environment for multi agent reinforcement learning research, where one can implement and test our suggested algorithms or develop his own.

Our environment is a 3D realistic car track with up to 16 cars that can drive simultaneously.

We built our environment in Unity 3D, and implemented both DDPG and MADDPG using PyTorch packages. We provide some base line results for comparison and open source code for implementing other algorithms.

# Background

## Machine Learning

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning focuses on the development of computer programs that can access data and use it learn for themselves.

The process of learning begins with observations or data, such as examples, direct experience, or instruction, in order to look for patterns in data and make better decisions in the future based on the examples that we provide. The primary aim is to allow the computers learn automatically without human intervention or assistance and adjust actions accordingly.

In this project we will work on a branch of machine learning call "Reinforcement Learning" (aka RL).

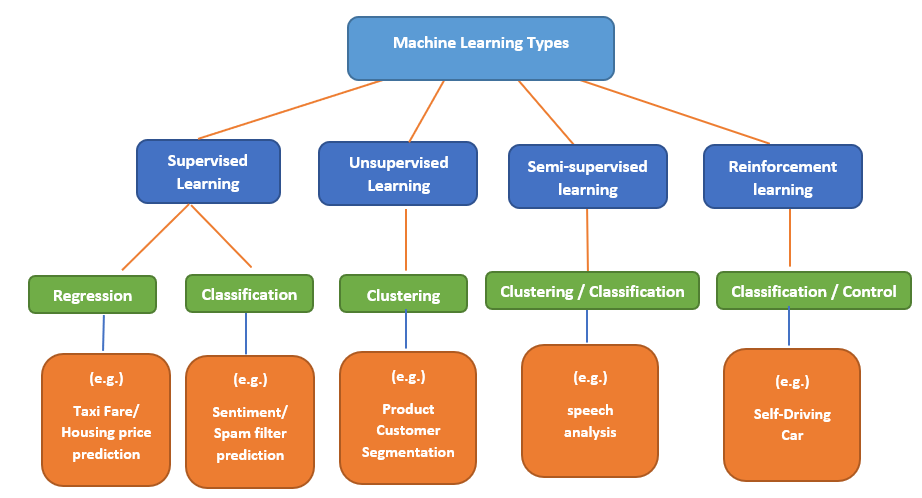


Figure 1- ML categories

## Reinforcement Learning

When it comes to explaining machine learning to those not concerned in the field, reinforcement learning is probably the easiest sub-field for this challenge. RL it’s like teaching your dog (or cat if you live your life in a challenging way) to do tricks: you provide goodies as a reward if your pet performs the trick you desire, otherwise, you punish him by not treating him, or by providing lemons. Dogs really hate lemons.

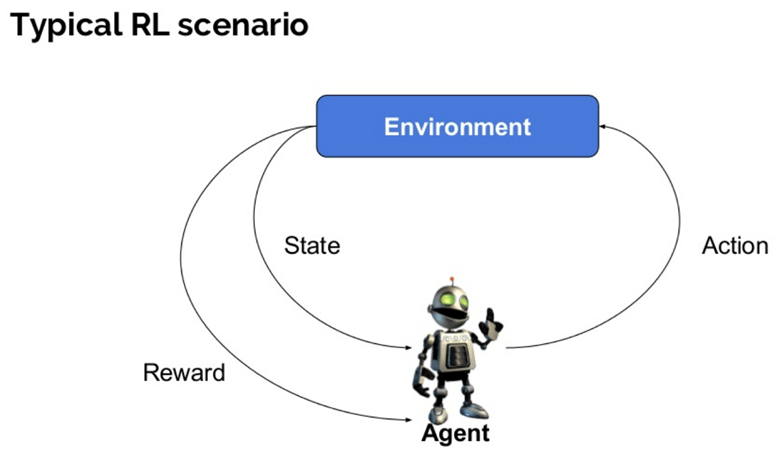


Figure 2- RL scheme

Here are some basic terms we need to use in order to continue:

* **Agent:** It is an assumed entity which performs actions in an environment to gain some reward (e.g. a dog).
* **Environment (e):** A scenario that an agent has to face (e.g. doggy playground).
* **Reward (R):** An immediate return given to an agent when he or she performs specific action or task (e.g. goodies for the dog, or lemons). The reward depends on the current state, and action: .
* **State (s):** State refers to the current situation the environment returns (e.g. dog's positions in time, weather, owner's mood etc.).
* **Observation (o)**: some partial data from the state, as observed by the agent (e.g. dog's view of the world).
* **Action (a)**: a move that the agent can make inside the environment (e.g. do can run, flip, jump…)
* **Policy (π):** It is a strategy which the agent applies to decide the next action based on the current observation (e.g. when to run, when to jump, when to flip…). The policy can be either deterministic or stochastic.
* **Value (V):** It is the expected long-term return with discount, as compared to the short-term reward. The value function takes the current 'state' as parameter.
* **Discount ():** the 'importance' of future rewards compared to current rewards.
* **Q value or action value (Q):** Q value is quite similar to value. The only difference between the two is that it takes an additional parameter as a current action. The connection between Q value and Value is:

In RL, the scenario is modeled as a Markov Decision Process (MDP):

* is the probability of transition (at time ) from state to state under action .
* is the immediate reward after transition from to with action .

A RL agent interacts with its environment in discrete time steps. At each time t, the agent receives an observation , which typically includes the reward . It then chooses an action from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state and the reward , associated with the transition , is determined. The goal of a reinforcement learning agent is to collect as much reward as possible. The agent can (possibly randomly) choose any action as a function of the history.

Question – we said that a machine learning algorithm learns from **data**. What data is used here?

Answer – A RL algorithm, will often use some "trial and error" of the agent. The agent will run around in the environment, try different actions, and get different rewards. For every timestep the RL algorithm will save . This will be the data in each the algorithm learns from.

There are many ways and algorithms used to solve this problem, but we will discuss the model free method of DDPG.

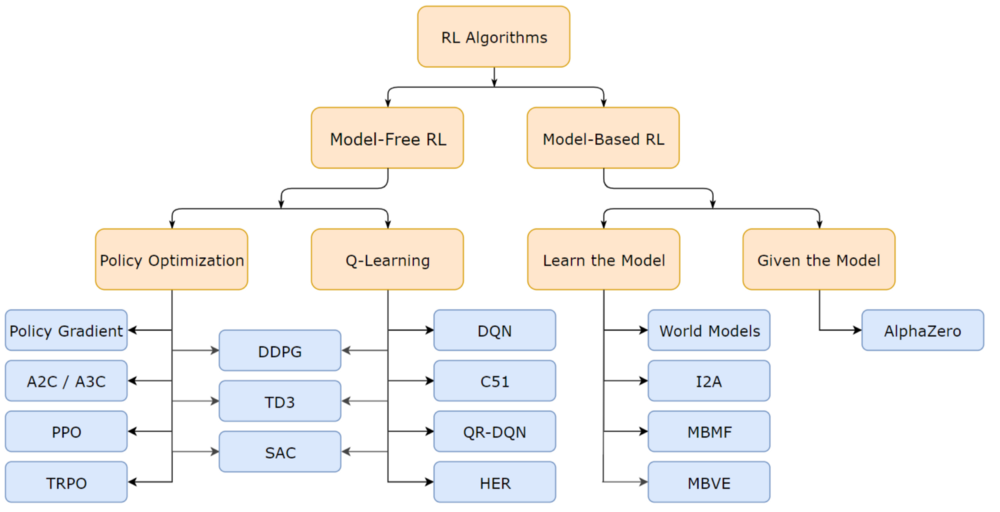


Figure 3- RL algorithm types

## DDPG

“Continuous Control with Deep Reinforcement Learning” (Lillicrap et al, 2015)

Deep Deterministic Policy Gradient (DDPG) is a **model-free** algorithm which concurrently learns a **Q-function** and an **optimal policy** using a **Deep Neural Network**. It uses **off-policy** data and the **Bellman equation** to learn the Q-function, and uses the Q-function to learn the **deterministic policy**. DDPG is used for **Continuous Control**.

Now let's go through these terms one by one:

* **Model free** - mean that the RL algorithm doesn't need an explicit statistical model of the environment (don't need to know ). Instead, DDPG tries to model the Q-value.
* **Continuous Control** – means that the actions a, are a continuous value (also - making them differentiable).
* **Off-policy -** basically means that the data (saved ) the algorithm uses in the learning process, does not necessarily come from the current policy the agent uses.
* **Deterministic Policy** – mean that the policy function is deterministic
  + The Trajectory (– a sequence of states and actions observed in the environment:

.

* + Since is deterministic, .
* **Optimal Policy ():**
  + The Return (R) – the overall discounted reward, determined by states and actions in time.
  + The optimal policy () – the policy function that maximizes the expected Return for every state.
* **Q-function in DDPG:**
  + The Optimal Q-value function () - gives the expected return if you start in state , take an arbitrary action , and then forever after act according to the optimal policy in the environment.

* **Bellman equation -** the Q-function obeys a special self-consistency equation called the Bellman equation. The basic idea behind the Bellman equations is this:

The value of your starting point is the reward you expect to get from being there, plus the value of wherever you land next.

Where s' is the next state, a' is the best next action (since is optimal), and represents the probability of s' to be the next state of the environment, given s and a. . The proof for the bellman equation comes from the fact that we are dealing with an MDP (without it, everything falls apart).

* **Deep Neural Network** **(DNN)**- an artificial neural network with multiple layers between the input and output layers. The DNN finds the correct mathematical manipulation to turn the input into the output. Each layer has a set of weights (parameters ) that are tuned (using optimization methods). After tuning the weights, a DNN can represent extremely complex mathematical functions. In DDPG, the algorithm uses a DNN to represent the optimal policy function and Q-function.

The Idea of DDPG is too 'learn' concurrently both and , when 'learn' means to tune the DNN's parameters . When the networks are trained, the agent can use as the policy to act of – thus solving the RL problem (maximizing its reward).

DDPG Implementation:

The DDPG architecture is said to be an "Actor Critic" model. The Actor is (deciding which actions to take) and the Critic is (deciding is the actor's action was good or not).

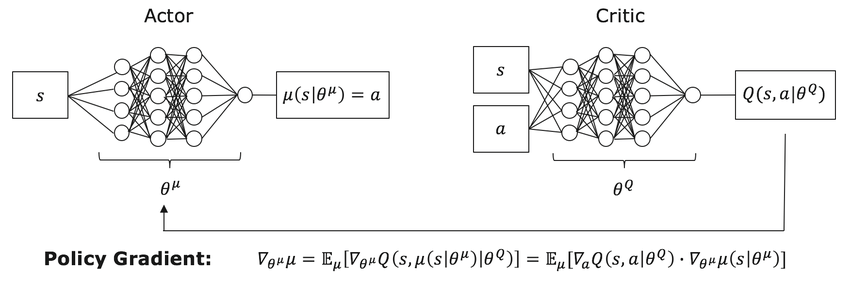


Figure 4 - DDPG architecture

**Learning :**

The Bellman equation is the starting point for learning an approximator to . Suppose the approximator is a neural network , with parameters , and that we have collected a set of transitions (where indicates whether state is terminal). We can set up a mean-squared Bellman error (MSBE) function, which tells us roughly how closely comes to satisfying the Bellman equation:

The Critic parameters are learned by: .

The Actor parameters are learned by

**Implementation Tricks:**

* Replay Buffers - All standard algorithms for training a deep neural network to approximate make use of an experience replay buffer. This is the set of previous experiences. In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep everything. If you only use the very-most recent data, you will overfit to that and things will break; if you use too much experience, you may slow down your learning. This may take some tuning to get right.
* Target Networks - Q-learning algorithms make use of target networks. The term

is called the target, because when we minimize the MSBE loss, we are trying to make the Q-function be more like this target. Problematically, the target depends on the same parameters we are trying to train: . This makes MSBE minimization unstable. The solution is to use a set of parameters which comes close to , but with a time delay—that is to say, a second network, called the target network, which lags the first. The parameters of the target network are denoted .

In DDPG-style algorithms, the target network is updated once per main network update by 'soft update'

where is a hyperparameter between 0 and 1 (usually close to 0). (This hyperparameter is called TAU in our code).

* Exploration vs. Exploitation - DDPG trains a deterministic policy in an off-policy way. Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make DDPG policies explore better, we add noise to their actions at training time. The authors of the original DDPG paper recommended time-correlated OU noise. To facilitate getting higher-quality training data, we reduce the scale of the noise over the course of training.

At test time, to see how well the policy exploits what it has learned, we do not add noise to the actions. (in our code we implemented this in our class OUNoise).

## MARL and MADDPG

* **MARL** - Multi-Agent Reinforcement Learning (MARL) is a growing and interesting field of RL.

As we know, RL has been applied to solve challenging problems like gameplaying, robotics, industrial applications and more. In the sections above we mentioned single agent domains. however, there are a number of important applications that involve interaction between multiple agents where RL agents co-evolve together. For example, multi-robot control, the discovery of communication and language, multiplayer games, and even analysis of social dilemmas - all operate in a multi-agent domain. Successfully scaling RL to environments with multiple agents is crucial to building artificially intelligent systems that can productively interact with humans and each other.

* **Challenges of MARL** - Unfortunately, traditional reinforcement learning approaches such as Q-Learning or policy gradient are poorly suited to multi-agent environments. One issue is that each agent’s policy is changing as training progresses, and the environment becomes non-stationary from the perspective of any individual agent (in a way that is not explainable by changes in the agent’s own policy).

Some explanation – when an agent "plays" around in the environment he "thinks" that his actions (and his actions alone) will lead to the rewards he gets along the way. From that perspective, an agent can't learn anything if his rewards are not solely depended on his actions. For example, if a tennis player plays a game thinking that he's playing against a brick wall – he will be very confused when the wall gets better every game.

This presents learning stability challenges and prevents the straightforward use of past experience replay, which is crucial for stabilizing deep Q-learning. Other RL methods, on the other hand, usually exhibit very high variance when coordination of multiple agents is required. In general, it is hard to train multiple (sometimes competing) agent, as evidenced by the notorious instability of adversarial training methods.

* **MADDPG** – one of the algorithms used to face these issues is the Multi-Agent Deep Deterministic Policy Gradient (MADDPG), presented in:

*Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., and Mordatch, I. (2017). Multi-agent actor-criticfor mixed cooperative-competitive environments. Inthe Annual Conference on Neural Informa-tion Processing Systems (NIPS).*

<https://arxiv.org/abs/1706.02275>

They used an extension to the Markov decision processes (MDPs), called Markov Games.

These Markov games, separate the State and Action into partial observations and actions that each agent takes. The architecture of the MADDPG agent is a set of multiple agents and critics (one for each agent), this opposed to other ideas of a central critic for all agents. Each critic receives as parameters all actions and partial observations from all of the agents, and that way the agents can learn better. Using the previous example, now the tennis players will not think of the opponent as a static wall – but as another player which gets better at the same time. The neural networks used in the MADDPG algorithm look as follows:

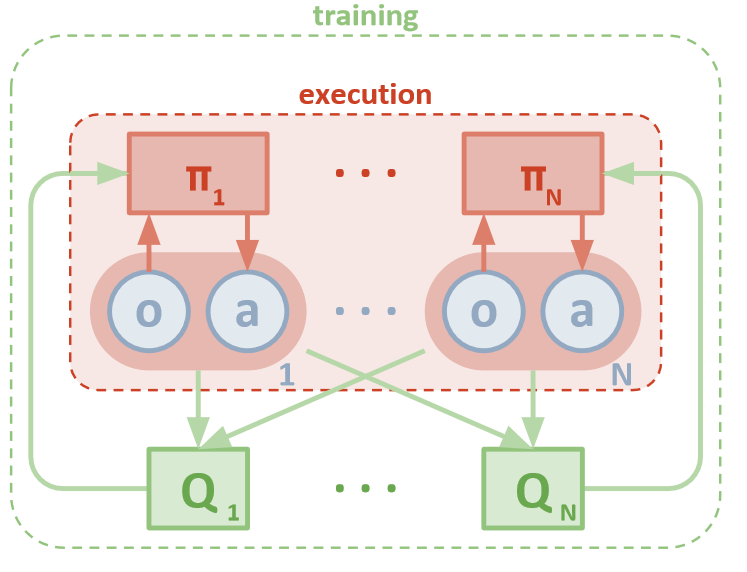
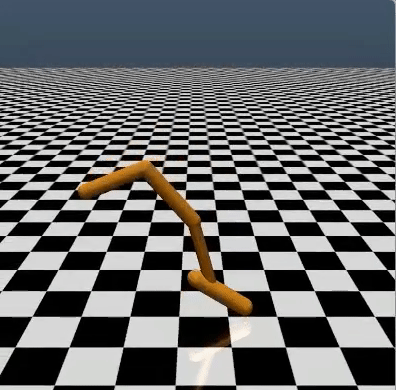


Figure 5 - MADDPG architecture

## Reinforcement Learning Environments

Modeling real life problems is hard. Especially when the problems we're dealing with are physical. As we wrote, Reinforcement Learning is a study of an agent learning through interactions with the environment. You can't have a computer learn to drive a real car, At least not at the beginning… Even learning to move a robotic arm can be challenging because of the financial cost of owning a robot. Thus, to test and compare results of different reinforcement learning algorithms, we need testbed environments. By far the most commonly used testbed has been 57 Atari 2600 games. However, different environments require different exploration schemes and different algorithms. Thus, it is important to have various environments. Here are some famous RL environments:

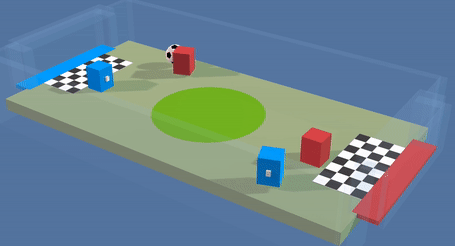
* **OpenAI Gym** – Perhaps the most famous toolkit for developing and comparing reinforcement learning algorithms. The gym library is a collection of environments that makes no assumptions about the structure of your agent. Gym comes with a diverse suite of environments, ranging from classic video games such as [Atari 2600](https://www.endtoend.ai/envs/gym#atari-2600) and [continuous control tasks](https://www.endtoend.ai/envs/gym#mujoco).

Figure 6 - OpenAI

* **The Unity Machine Learning Agents Toolkit** (ML-Agents) is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents.
* **PySC2** - [DeepMind](http://deepmind.com/)’s Python component of the StarCraft II Learning Environment (SC2LE). It exposes [Blizzard Entertainment](http://blizzard.com/)’s [StarCraft II Machine Learning API](https://github.com/Blizzard/s2client-proto) as a Python RL Environment. This is a collaboration between DeepMind and Blizzard to develop StarCraft II into a rich environment for RL research. PySC2 provides an interface for RL agents to interact with StarCraft 2, getting observations and sending actions.

Figure 7- Unity

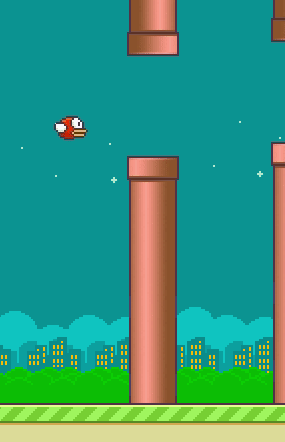
* **PyGame** **Learning Environment (PLE)** - A learning environment, mimicking the Arcade Learning Environment interface, allowing a quick start to Reinforcement Learning in Python. The goal of PLE is allow practitioners to focus design of models and experiments instead of environment design.

Figure 8 - PySC2

Figure 9 - PyGame

* **MARL environments** - even though there are many RL environments, there are not many built for multi-agent scenario. Especially environments that are modular in the sense of the number of agents. The well-known existing environments are the unity tennis environment, and OpenAI's "Multi-Agent Particle Environment" (which was used in the MADDPG paper). These environments are either not scalable (tennis) or not very interesting (particles).

Figure 10 - OpenAI Particle enviornments as used in MADDPG article

## Image result for made with unity logoUnity

Figure 10 - Tennis in Unity

Unity is a cross-platform game engine developed by Unity Technologies. As of 2018, the engine had been extended to support more than 25 platforms. The engine can be used to create three-dimensional, two-dimensional, virtual reality, and augmented reality games, as well as simulations and other experiences. The engine has been adopted by industries outside video gaming, such as film, automotive, architecture, engineering and construction.

Unity gives users the ability to create games and experiences in both 2D and 3D, and the engine offers a primary scripting API in C#, for both the Unity editor in the form of plugins, and games themselves, as well as drag and drop functionality. In addition, unity own the “Unity asset store” which is an online store for buying assets such as code, 3D modules, affects, and more.

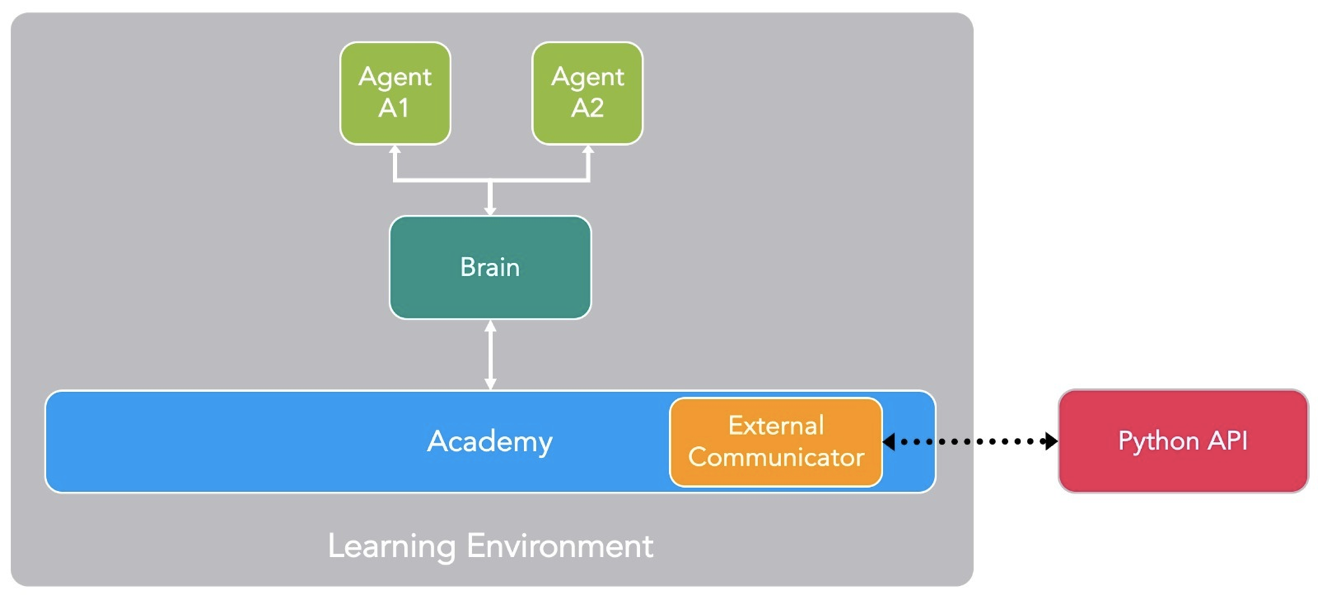
Some well-known games developed by Unity includes: Assassin’s Creed, Temple Run, Battlestar Galactica, Deus Ex: The Fall and more.

## ML-Agents-Toolkit

An open source add-in to Unity which allows to simulate games for training RL agents. The add-in provides an API for communicating with Python code.

The main objects in this toolkit are:

* Brain – A main object that pipes data from the agents to the application and back. Its inputs are the agent observations and rewards, and its output is the actions to those agents.
* Academy – manages the python-Unity communications and initializes the environment from user parameters. At run time, the academy manages the Python-Unity communication socket.
* Agent – collects observations and rewards, passes them to the brain, and acts on the signals that come back from the brain. The agent controls the agent instance in the game (the player in the game).



# Our project

## General idea

we decided to contribute to the MARL research by developing an environment where one could test and implement MARL algorithms.

We chose car race environment, where each car is an agent.

As we mentioned, MARL environments today are limited and not easy scalable.

## Goals

We wanted to develop an environment that will give value to developers and researchers in this field. To achieve this, we set some requirements for the environment:

* Realistic – the environment will be as realistic as possible, both with physics laws and with high graphic quality.
* Scalable – scaling up the number of agents or the environment complexity will be easy.
* User friendly – all the configuration could be made with one-line instruction. The project will be well documented and have a user guide for setup and usage.
* Modularity – the user could implement his own algorithm or any other part of the code without modifying other parts.
* Performance base line – providing weights file and basic results for other users to compare as base line for their algorithms.

# Project in details

## High level architecture

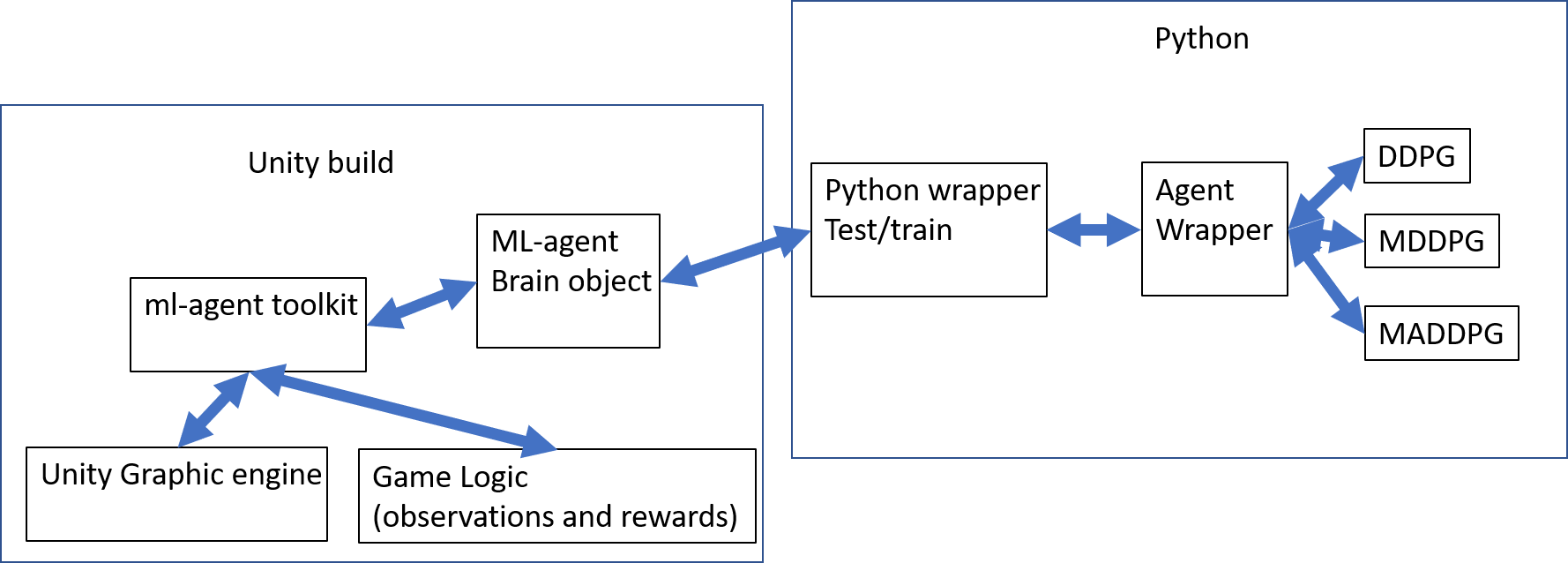


Figure 11- high level architechture

In the project, we used different development environments what need to communicate with each other at run time.

The environment is built in a modular way to provide future users the ability to modify each section independently of the others.

## Architecture in details

### Unity scene

The strong graphic engine of Unity allows us to use advanced physics laws and high-level graphics which makes our environment to behave and look realistic. Using the unity editor (and some downloaded assets), we created our game scene. Our scene includes basic race track, car objects, and cameras. The online assets we used are:

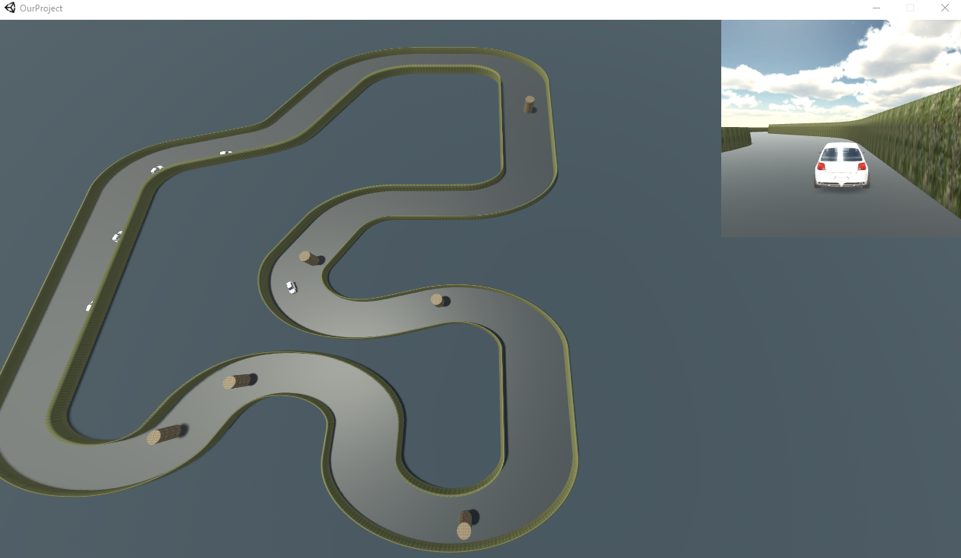
* Unity standard assets – Racetrack and Skybox.
* Low Poly Destructible 2 Cars no. 8 – "Classic 16" frame.
* Yughues Free Stone and Grass Materials – stone and grass materials.

Figure 12- Our scene

### Unity scripts

All the Unity scripts are in C#. these scripts manage the game objects at run time.

The scripts we wrote are:

* Race Academy – derived from the ml-agents-toolkit Academy class. This class communicates with our python code, and initializes the environment with the parameters given by the user.
* Agent manager – initiates all in-game agents, in our case cars.
* Race agent – derived from the ml-agents-toolkit Agent class. Each car has an instance of this class. This class handles the logic regarding on the car's observations, rewards, and actions.
  + The optional actions of an agent are:
    - Throttle – continuous value in the range [-1, 1].
    - Wheel angle – continuous value in the range [-1, 1].
  + The collected observations from an agent are:
    - Speed – current speed normalized by the car max speed (1 dimension).
    - Driver field of view – 11 different angles of observations to simulate real driver point of view. These observations can tell the distance from nearby objects, as well as distinguish between different object types (44 dimensions).
    - Direction of movement – A binary variable that tells if the car is moving in the right direction (1 dimension).

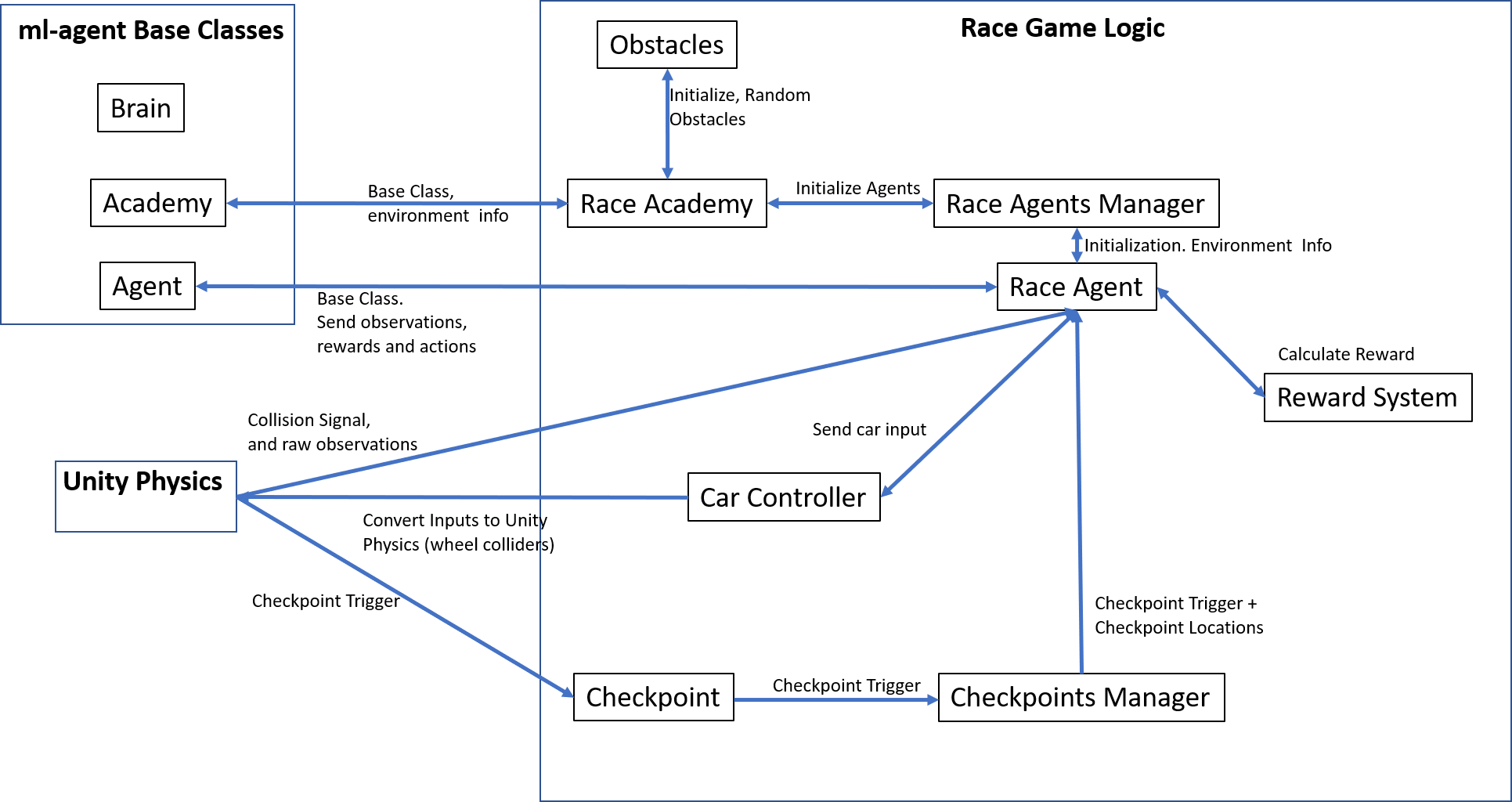


Figure 13- Unity in-game architechture

* Reward System – part of Race Agent. This script handles the logic for achieving rewards. For changing the rewards mechanism, one can edit this class. The rewards we use in our implementation are:
  + Checkpoint pass – along the track we have checkpoints. By passing each of them in the right direction, the agent gets a positive reward.
  + Speed – reward in the sign of the movement direction, and in magnitude of the normalized speed – positive high reward for high speed in the right direction, and a high penalty for high speed in the wrong direction.
  + Standing in place – negative reward for standing still (speed very close to zero)
  + Hitting another object – negative rewards for hitting other cars, walls, or obstacles.

For most of the autonomous vehicle implementations (RL and classic Control), the strategy is usually to follow a line along the race track. We deliberately did not give the car any track to follow since we wanted the car to learn on its own what is the best path to drive in.

* Obstacles Manager – handles the track obstacles. In each episode a random N out of 16 possible obstacles are selected. N is given by the user in the command line.
* Checkpoint and Checkpoint Manager – handles the checkpoint that spread along the track, the checkpoints are used to determine the direction of movement of the agent, and give positive rewards when passing the next checkpoint on track.

## Python

The Python work can be split into two: the “test and train” scripts and the “agents” scripts.

### Test, Train and Main

The test.py and train.py are both wrapper files that communicate with the Unity environment. These files run the training and testing of the models. They are independent of the algorithm which the agents are using and of the Unity build (the game program).

By making these files independent we provide the user the ability to change both the agents and the Unity build, and then use the same command line instructions for test and train.

The main.py scripts provides the user interface usage guide (main.py –h) for running our project. The script parses the user command line and initiates the relevant environment configuration. All setup options can be used with one-line command, by using the optional flags.

### Agents

We implemented three different basic agent types based on articles and previous implementations we found online. We made adaptation to these algorithms in order to fit them into our environment, and test them, all to provide future users a performance baseline for their work. All agents are dependent on PyTorch library but one can use any other library he wishes, since the wrapper functions are independent of the implementation details.

To make the project user friendly to future developers, we chose to create an abstract agent class which all others agents derive from. Train and test are aware only of this abstract class. When initializing the agents, the scripts verify that the agent was indeed derived from Agent abstract class. By doing this, we avoid “Duck Type” and make sure that adding an agent to the project complies to our expected behavior of it.

Our agents:

* DDPG – single agent with Actor Critic architecture. This agent acts alone in the environment. in case there are more than one car while using DDPG, all the cars are under the same network and effectively act as one instance.
* MADDPG – multi agent DDPG. In this architecture each car is an agent and have her own Actor Critic instances. All agents Critics sharing information. This algorithm is shown in the main article we rely in this project.
* MDDPG – we created this architecture as compromise between DDPG and MADDPG. In this mode each car as her own agent, but each agent is independent DDPG agent and there is no communication between the agents.

All of the code is fully documented, making the implementation tidy and easy to understand.

# Results

## Baseline Evaluation

Although our project is not result-oriented, we wanted to provide a solid baseline. We tested our implementations and improved it until it was sufficient. We Implemented 3 RL agent types, using 2 different algorithms – DDPG and MADDPG.

### DDPG

single agent algorithm. Using a single agent gives the best average score, the car finishes the track nicely, avoiding all obstacles the achieving high score. When used in multi-agent environment (the purpose of this project) the DDPG agent's score falls behind the other implementations.

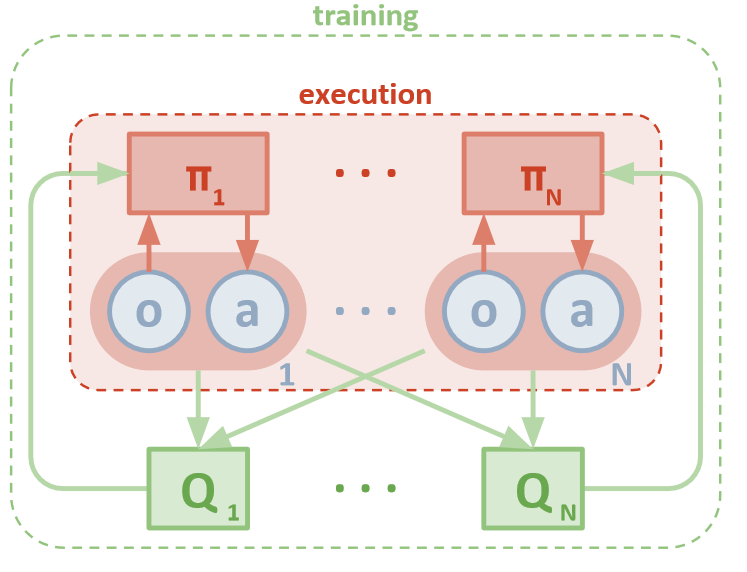
### MDDPG

This agent type is actually a regular DDPG agent, but each car has its own NN – multiple DDPG. MDDPG gave us the best base line we could get for this problem. For a few agents (3-5) we got decent results, but for higher number of agents (6-8) the results are far worse. This is actually good, since we know that our environment is challenging enough for future algorithms (one of the issues of the existing environments is that they're too easy).

In the graph below we can see the average score for the MDDPG agent, as a function of the number of epochs passed. The Average score is taken with a "moving window", taking the mean of the last 20 scores. As we can see, even after averaging the progress is still noisy – indicating the "noisy" nature of RL algorithms.

### MADDPG

This complex algorithm took some time to implement, and the results didn't come easy. This algorithm was very sensitive to hyper-parameter tuning, and it took us a while to get meaningful results.



The best scores we got was by using "pre-training".

* + "Pre-Training": the idea was to first train the agent to drive by itself, and only after this "pre-training" phase, we let it be trained with knowledge of the other agents in the scene. We do this be zeroing the input from other agents – for the critic of agent j, we zero the states and actions arriving from all other agents in the scene.

When the "pre-training" is completed, we save the replay-buffer and weights of the agent and load them for the second phase of training. With this method we managed to achieve decent results.

Figure 14- pre-train illustration

Since this was not the purpose of our project, we did not allow this method to be used in our final version. This pre-train idea also may also be a direction of study to pursue in the future in regarding to RL "Transfer Learning".

We can see in the graphs that without using this "trick" the results we got were worse than the MDDPG Agent's results.

# Summery

In this project we got into a new area of machine learning. We learned about reinforcement learning and about the challenges of the field. We developed a 3D game environment from scratch, and learned how to use all the tools we needed and how to integrate them together.

In our learning process, we first create a simple system and made it work. We worked on it until we had a simple game and simple algorithm, that showed good performance and convergence rate. Later, we implemented more complex algorithms (MADDPG) and scaled up the number of agents. The scaling and the MADDPG algorithm didn’t worked as expected and we struggled to get sufficient results and convergence. This forced us to research the algorithm deeper and try to improve its performance. One method we thought of was the "pre-training" of an MADDPG agent. We hope that this idea will be explored in future work since it worked for us, and can potentially improve MARL algorithms.

To sum up the implementation, we go back to the project goals:

* Realistic – using Unity 3D packages we provided real life physical behavior to the cars and environment. We added an extra camera to focus on one agent and see it perform from a different angle.
* Scalable – the environment can support up to 8 agents and up to 16 obstacles right now. If this scale isn’t enough, a future user can create more objects easily and connect them to the code.
* User friendly – we created command line interface with usage guide (main –h) where the user can find all the different options for running and testing. In addition, we provided detailed text file with setup instructions and running examples. Any user can theoretically run this project without any background knowledge.
* Modularity – The code modules communicate with each other in way that the inner implementations don't affect. The user can change any block of the project while preserving the input and output of the block and the environment will work. The part that is most likely to be changed by future users is the Agent class where the RL algorithm is implemented. To make this generic as possible we created an abstract Agent class that the user can inherit from, and implement his own algorithm.
* Performance Base line – we trained many agents with different configuration options and recorded the result. Any future user can look and compare his own results to our base result.

Our final version is a working environment, which any user can download for free from our GitHub repository. We provide base line for three different agent types, Unity game build and weights files ready to use.

This project can contribute as a tool to any researcher in the field of reinforcement learning.

# Bibliography

Links:

<https://www.codeproject.com/Articles/5245488/Introduction-to-Machine-Learning-and-ML-NET-Part-1>

<https://www.guru99.com/reinforcement-learning-tutorial.html>

<https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/>

<https://www.geeksforgeeks.org/what-is-reinforcement-learning/>

<https://towardsdatascience.com/q-learning-54b841f3f9e4>

<https://en.wikipedia.org/wiki/Reinforcement_learning>

<https://en.wikipedia.org/wiki/Q-learning>

<https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c>

<https://www.endtoend.ai/envs/>

<https://expertsystem.com/machine-learning-definition/>

[https://spinningup.openai.com/en/latest/algorithms/ddpg.html#](https://spinningup.openai.com/en/latest/algorithms/ddpg.html)

<https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#the-optimal-q-function-and-the-optimal-action>

<https://www.researchgate.net/figure/Basic-structure-of-the-DDPG-actor-critic-agent-Actor-with-parameters-th-m-state-s-as_fig3_322879739>

<https://arxiv.org/pdf/1706.02275.pdf>

<https://www.endtoend.ai/envs/>

<https://assetstore.unity.com/>

# Appendix

## Usage and instructions

First thing you'll have to do is clone our repo from <https://github.com/ShaharGottlieb/RL_Multy_Agent_Unity>. In this repo you'll find:

* Python source code
* Unity Assets
* Unity ready executables
* Some post-training weights ready to run

Our project consists of 2 parts – the Unity game, and the python project.

If you want to just run/play with our python code, implement new agents etc., you can use the provided executables in the repo (windows, linux, and Mac). If you want to change more (reward systems, observations, other in-game changes) – download Unity, modify our game, and build your own version.

All software versions specified are the version we used when we created this project. These are the versions we know for sure to work with each other. You can install different versions, but you'll have to make sure they work together.

Python part –

* In order to run our code please install **python 3.6.8**. this was the latest version to work with ml-agents repository when we created this project. Requested libraries are:
  + Pytorch 1.1 (and all dependencies)
  + mlagents 0.7 (and all dependencies)

these libraries can be installed easily using pip.

Unity part (optional) –

the following steps are to build easily our Unity project, corresponding to the following versions:

* Unity **2018.3.7f1**
* Ml-agents release 0.7.0 (28.02.2019)

If you want to build latest version of ml-agents, refer to instructions in <https://github.com/Unity-Technologies/ml-agents>. Otherwise, the ml-agents unity files are already inside our repo, and no need to clone ml-agents repo at all.

* Download Unity:
* Download and install Unity. You can download it from <https://unity.com/>.

The version of unity that this project was created with, is **2018.3.7f1.** any other version is not promised to work.

* If working on windows, and you want to modify the scripts in unity, it is recommended to download *Microsoft Visual Studio Tool for Unity.* This tool made the C# scripting a lot easier for us.
* Setup the build (open our project):
  + Open a new project in unity
  + A folder "Assets" was created for this new project. Replace it (or create a soft link) with the "Assets" folder from our repo (Base\UnityEnvs\Assets).
  + Inside the Unity editor open our scene "Assets/ML-Agents/Examples/MyRace/MyRace.unity"
  + Build the project: in file/build\_settings select the target platform, press "Add Open Scenes" and press Build.