DEVELOPING A ML AGENT USING UNITY PLATFORM

1. INTRODUCTION

In this project we would like to create a ML(Machine learning) agent in unity platform and see how useful the real world application is.This project discusses the use of AI and machine learning in Unity game engine. Using unity's own machine learning toolkit, the primary plan is to research the basics of unity machine learning, which makes it possible to train intelligent agents using reinforcement learning (RL) via simple Python API, use it to implement a self-learning agent into a unity simulation, this self learnt agent can be used in creating and solving real world applications and problems.

Fig.1.1: Unity ML-Agent Workflow model

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents. This provides implementations (based on PyTorch) of state-of-the-art

algorithms to enable game developers and hobbyists to easily train intelligent agents for 2D, 3D and VR/AR games. Researchers can also use the provided simple-to-use Python API to train Agents using reinforcement learning, imitation learning, neuroevolution, or any other methods.

2. LITERATURE SURVEY/REVIEW OF LITERATURE

In recent years, there have been a number of simulation platforms developed for the purpose of providing challenges and benchmarks for deep reinforcement learning algorithms. Many of these platforms are based on existing games or game engines and carry with them specific strengths and weaknesses. While not exhaustive of all currently available platforms, below we survey a few of the simulators.

2.1 ARCADE LEARNING ENVIRONMENT

The release of the Arcade Learning Environment (ALE) contributed to much of the recent resurgence of interest in reinforcement learning. This was thanks to the development of the Deep Q-Network, which was able to achieve superhuman level performance on dozens of emulated Atari console games within the ALE by learning only from pixel inputs (Mnih et al., 2015). The ALE provides a Python interface for launching and controlling simulations of a few dozen Atari 2600 games. As such, the ALE falls into the category of environment suite. When considering the simulation criteria described above, the ALE provides visual complexity through pixel-based rendering, task-logic complexity in the form of hierarchical problems within some games such as Montezuma's Revenge, and high-performance simulation with an emulation able to run at thousands of frames per second (Bellemare et al., 2013).

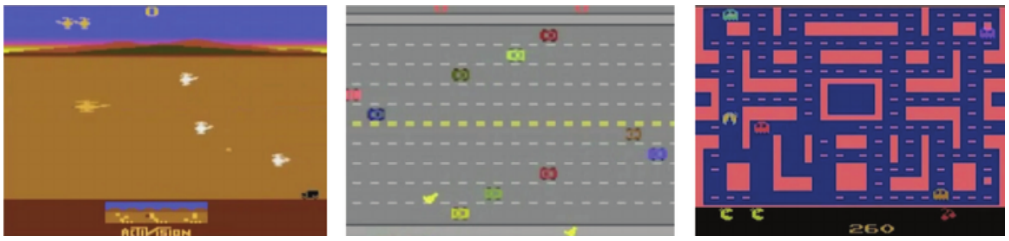


Fig. 2.1: Examples of ALE working models.

Its downsides include deterministic environments, relatively simple visuals, a lack of realistic physics, single agent control, and a lack of flexible control of the simulation configuration. In general, once an environment that is part of the ALE is launched, it is immutable and a complete black box from the perspective of the agent.

2.2 DEEPMIND LAB BUILT FROM THE QUAKE III GAME ENGINE

DeepMind Lab (Lab) was released in 2016 as the external version of the research platform used by DeepMind (Beattie et al., 2016). Designed in the wake of public adoption of the ALE, Lab contains a number of features designed to address the other platform's shortcomings. By using a 3D game-engine, complex navigation tasks similar to those studied in robotics and animal psychology could be created and studied within Lab. The ability to create a set of specific kinds of tasks makes DeepMind Lab a domain-specific platform. The platform contains primitive physics enabling a level of prediction about the quality of the world and allows researchers to define their own environmental variations.

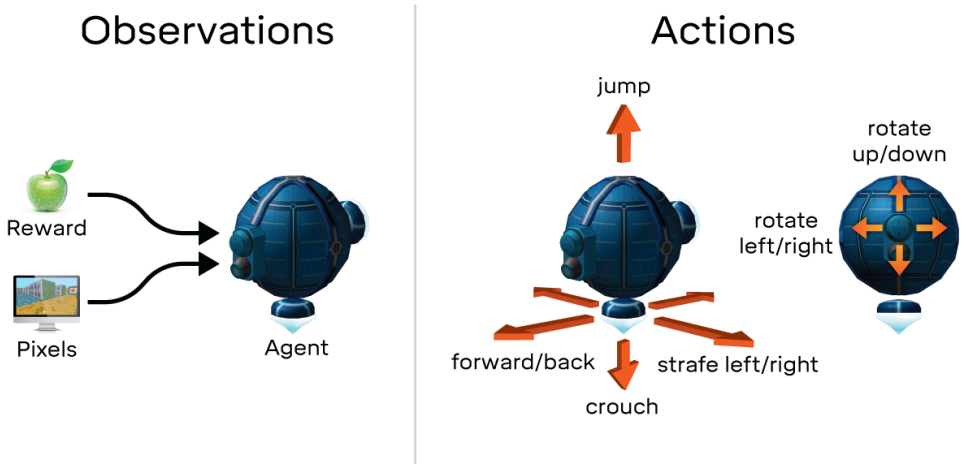


Fig. 2.2: Open Source deep mind lab's first person view ML-Agent built on decade old graphics

The limitations of this platform, however, are largely tied to the dated nature of the underlying rendering and physics engine, which was built using decades-old technology. As such, the gap in quality between the physical world and the simulation provided via Lab is relatively large.

2.3 PROJECT MALMO

Another popular simulation platform is Project Malmo (Malmo) (Johnson et al., 2016). Based on the exploration and building game Minecraft, the platform provides a large amount of flexibility in defining scenarios and environment types making it a domain-specific platform. As a result, there have been a number of research projects exploring multi-agent communication, hierarchical control, and planning using the platform.

Fig. 2.3: Microsoft's Project Malmo

The limitations of the platform, however, are bound tightly with the underlying limitations of the Minecraft engine itself. Due to the low-polygon pixelated visuals, as well as the rudimentary physics system, Minecraft lacks both the visual as well as the physical complexity that is desirable from a modern platform. The platform is also limited to describing scenarios which are only possible within the logic of Minecraft.

2.4 PHYSICS SIMULATORS THE MUJOCO PHYSICS

The MuJoCo physics engine has become a popular simulation platform for benchmarking model-free continuous control tasks, thanks to a set of standard tasks built on top of MuJoCo, provided with OpenAI Gym and the DeepMind Control Suite. High quality physics simulation combined with a number of standardized benchmarks has led to the platform being the primary choice for researchers interested in examining the performance of continuous control algorithms. The nature of the MuJoCo engine, however, poses limitations for more general AI research.

Fig. 2.4: MuJoCo as unity plugin as it is an open standard

The first is around the limited visual rendering capabilities of the engine, preventing the use of complex lighting, textures, and shaders. The second is the restrictions of the physics engine itself and that MuJoCo models are compiled which makes the creation of dynamic “game-like” environments, where many different objects would be instantiated and destroyed in real-time during simulation.

3. SOFTWARE REQUIREMENT ANALYSIS

There are many hurdles while developing a virtual environment. There are many requirements to be met when developing a simulation for an ML-Agent. Generally developing a 3D environment takes a lot of strain on a PC but Training an Agent takes it to another level, without a GPU it is highly impossible to train an agent and render it in Unity. The functionality of the program should meet with requirements in the real world which are possible real world outcomes.

3.1 UNDERSTANDING THE PROBLEM

Simulated environments are constrained by the limitations of the simulators themselves. Simulators are not equal in their ability to provide meaningful challenges to learning systems. Furthermore, it is sometimes not obvious which properties of an environment make it a worthwhile benchmark for research. The complexity of the physical world is a primary candidate for challenging the current as well as to-be-developed algorithms. It is in the physical world where mammals and, more specifically, human intelligence developed, it is this kind of intelligence which researchers are often interested in replicating (Lake et al., 2017).

3.2 SOFTWARE AND LANGUAGE USED IN DEVELOPMENT

The softwares and languages used in this project are very standard when working with Unity platform. Basic programming knowledge and graphic development is needed to replicate this project.

3.2.1 UNITY

Fig. 3.1: Unity logo and 2020 version interface

Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Inc.'s Worldwide Developers Conference as a Mac OS X-exclusive game engine. 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. Prior to C# being the primary programming language used for the engine, it previously supported Boo, which was removed with the release of Unity 5, and a version of JavaScript called UnityScript, which was deprecated in August 2017, after the release of Unity 2017.1, in favor of C#.

Within 2D games, Unity allows importation of sprites and an advanced 2D world renderer. For 3D games, Unity allows specification of texture compression, mipmaps, and resolution settings for each platform that the game engine supports, and provides support for bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects.

Working with Unity is simple and easy to learn for a beginner. As there are many options to choose from we choose unity because of its simplicity, customizability, and live demo options.

3.2.2 C#

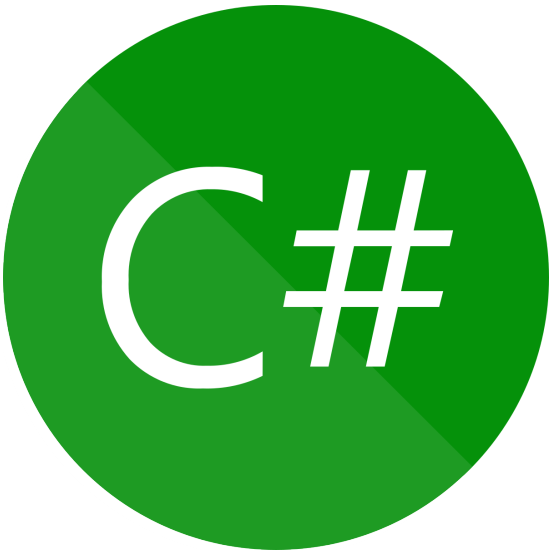


Fig. 3.2: C Sharp logo

C# (pronounced see sharp, like the musical note C♯, but written with the number sign)[b] is a general-purpose, multi-paradigm programming language encompassing static typing, strong typing, lexically scoped, imperative, declarative, functional,

generic, object-oriented (class-based), and component-oriented programming disciplines.

C# was developed around 2000 by Microsoft as part of its .NET initiative and later approved as an international standard by Ecma (ECMA-334) in 2002 and ISO (ISO/IEC 23270) in 2003. It was designed by Anders Hejlsberg, and its development team is currently led by Mads Torgersen, being one of the programming languages designed for the Common Language Infrastructure (CLI). The most recent version is 9.0, which was released in 2020 in .NET 5.0 and included in Visual Studio 2019 version 16.8.

Mono is a free and open-source project to develop a cross-platform compiler and runtime environment (i.e. virtual machine) for the language.

We choose C# because

- The language is intended to be a simple, modern, general-purpose, object-oriented programming language.
- The language, and implementations thereof, should provide support for software engineering principles such as strong type checking, array bounds checking, detection of attempts to use uninitialized variables, and automatic garbage collection. Software robustness, durability, and programmer productivity are important.
- The language is intended for use in developing software components suitable for deployment in distributed environments.
- Portability is very important for source code and programmers, especially those already familiar with C and C++.
- Support for internationalization is very important.

- C# is intended to be suitable for writing applications for both hosted and embedded systems, ranging from the very large that use sophisticated operating systems, down to the very small having dedicated functions.
- Although C# applications are intended to be economical with regard to memory and processing power requirements, the language was not intended to compete directly on performance and size with C or assembly language.

3.2.3 PYTORCH

PyTorch is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab (FAIR). It is free and open-source software released under the Modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.

Fig. 3.3: PyTorch logo

A number of pieces of deep learning software are built on top of PyTorch, including Tesla Autopilot, Uber's Pyro, HuggingFace's Transformers, PyTorch Lightning, and Catalyst.

PyTorch provides two high-level features:

- Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPU)
- Deep neural networks built on a type-based automatic differentiation system

3.2.4 TENSORFLOW

TensorFlow is a free and open-source software library for machine learning. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.

Tensorflow is a symbolic math library based on dataflow and differentiable programming. It is used for both research and production at Google.

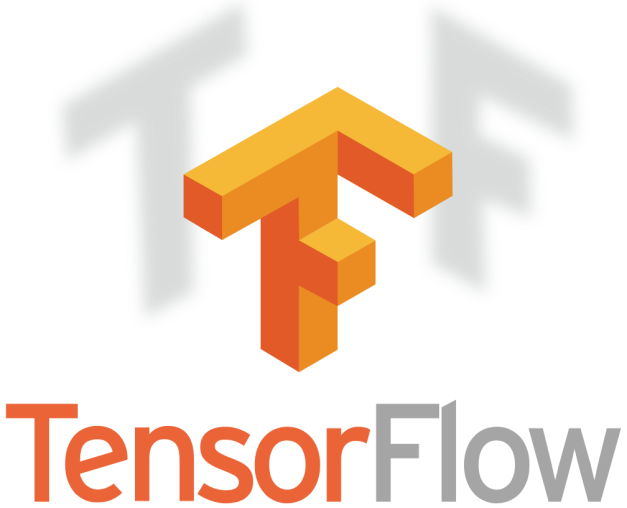


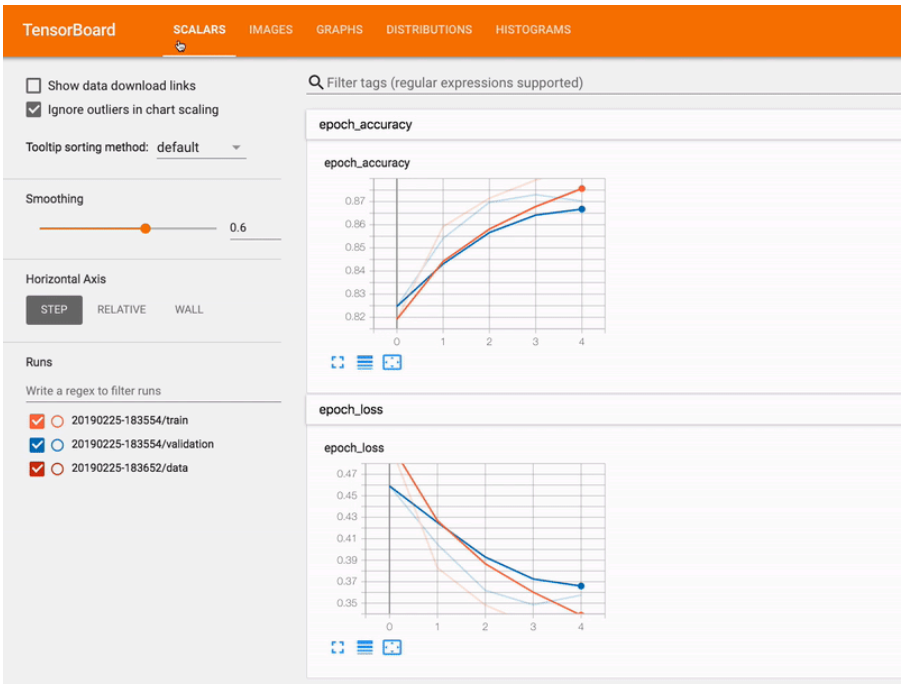
Fig. 3.4: TensorFlow Logo

TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache License 2.0 in 2015.

3.2.5 TENSORBOARD: TENSORFLOW'S VISUALIZATION TOOLKIT

In machine learning, to improve something you often need to be able to measure it. TensorBoard is a tool for providing the measurements and visualizations needed during the machine learning workflow.

Fig. 3.5 :Tensor board visualization of data from python



It enables tracking experiment metrics like loss and accuracy, visualizing the model graph, projecting embeddings to a lower dimensional space, and much more.

3.2.6 PYTHON

Python is an interpreted high-level general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant indentation.

Fig. 3.6: Python logo

Its language constructs as well as its object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural), object-oriented and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.

Guido van Rossum began working on Python in the late 1980s, as a successor to the ABC programming language, and first released it in 1991 as Python 0.9.0. Python 2.0 was released in 2000 and introduced new features, such as list comprehensions and a garbage collection system using reference counting. Python 3.0 was released in 2008 and was a major revision of the language that is not completely backward-compatible and much Python 2 code does not run unmodified on Python 3. Python 2 was discontinued with version 2.7.18 in 2020. Python consistently ranks as one of the most popular programming languages.

4. CONCEPTS AND ALGORITHMS

4.1 UNSUPERVISED LEARNING

Unsupervised learning (UL) is a type of algorithm that learns patterns from untagged data. The hope is that, through mimicry, the machine is forced to build a compact internal representation of its world and then generate imaginative content. In contrast to supervised learning (SL) where data is tagged by a human, e.g. as "car" or "fish" etc, UL exhibits self-organization that captures patterns as neuronal predilections or probability densities.

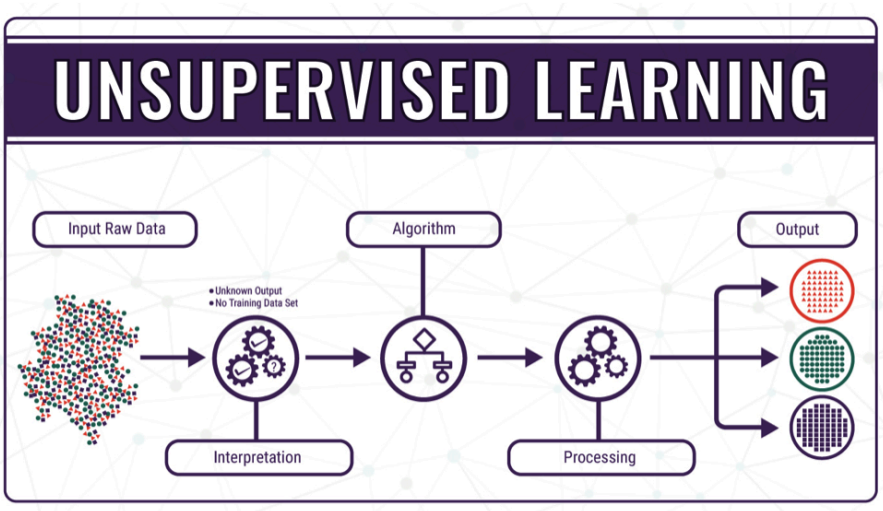


Fig. 4.1: Unsupervised Learning

The other levels in the supervision spectrum are reinforcement learning where the machine is given only a numerical performance score as its guidance, and semi-supervised learning where a smaller portion of the data is tagged. Two broad methods in UL are Neural Networks and Probabilistic Methods.

4.2 REINFORCEMENT LEARNING

Reinforcement learning (RL) is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

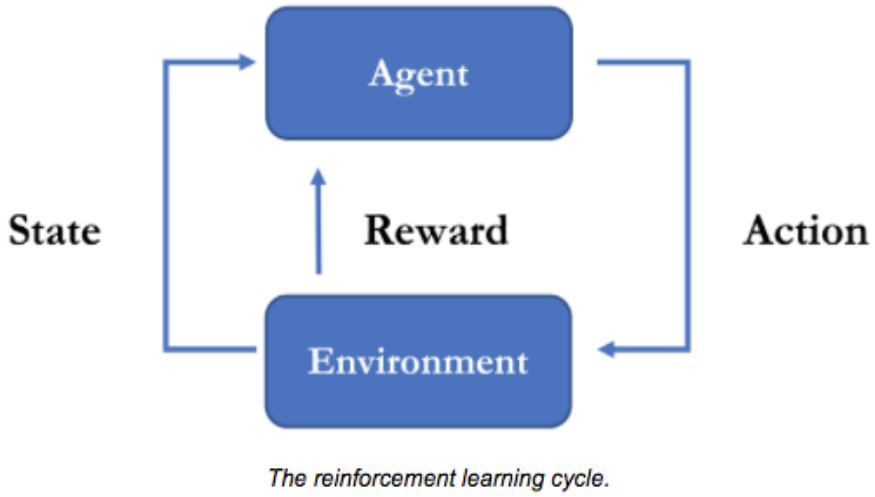


Fig. 4.2: Reinforcement learning cycle

Reinforcement learning differs from supervised learning in not needing labelled input/output pairs to be presented, and in not needing sub-optimal actions to be explicitly corrected. Instead the focus is on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

4.3 PROXIMAL POLICY OPTIMIZATION

Proximal Policy Optimization performs better than state-of-the-art approaches while being much simpler to implement and tune. PPO has become the default reinforcement learning algorithm because of its ease of use and good performance.

It involves collecting a small batch of experiences interacting with the environment and using that batch to update its decision-making policy. Once the policy is updated with this batch, the experiences are thrown away and a newer batch is collected with the newly updated policy.

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)\,]$$

- θ is the policy parameter
- $\hat{E}_t$ denotes the empirical expectation over timesteps
- r_t is the ratio of the probability under the new and old policies, respectively
- $\hat{A}_t$ is the estimated advantage at time t
- ε is a hyperparameter, usually 0.1 or 0.2

  - A policy defines how an agent acts from a specific state.
  - A policy is a network of those successful steps from the state to the successful outcome.

With supervised learning, we can easily implement the cost function, run gradient descent on it, and be very confident that we'll get excellent results with relatively little hyperparameter tuning. The route to success in reinforcement learning isn't as obvious — the algorithms have many moving parts that are hard to debug, and they require substantial effort in tuning in order to get good results. PPO strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small.

4.4 SOFT ACTOR CRITIC

The Soft Actor-Critic (SAC) algorithm has been developed jointly at UC Berkeley and Google. Soft actor-critic is one of the most efficient model-free algorithms available today, making it especially well-suited for real-world robotic learning.

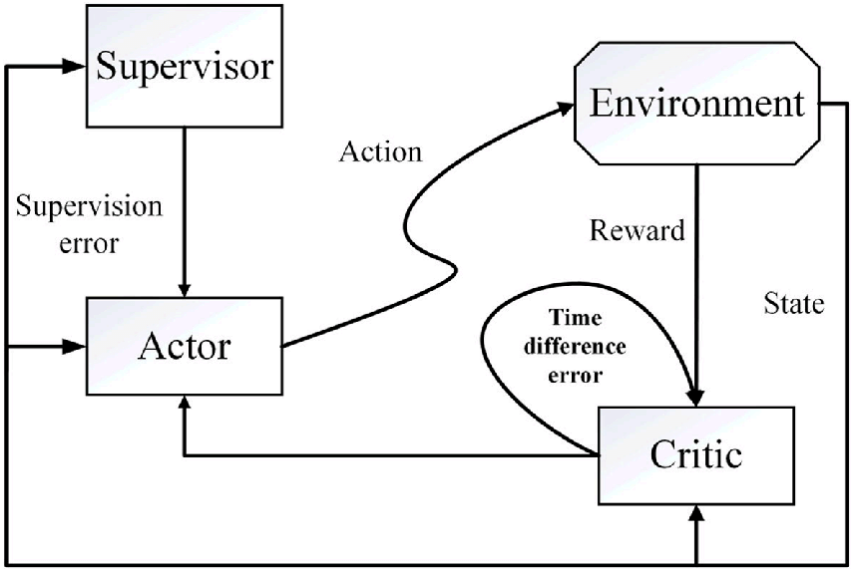


Fig. 4.3: soft actor critic workflow

SAC is based on the maximum entropy RL framework. In this framework, the actor aims to simultaneously maximize expected return and entropy. That is, to succeed at the task while acting as randomly as possible.

Regarding the algorithm, several properties are desirable:

- Sample Efficiency. Learning skills in the real world can take a substantial amount of time. Prototyping a new task takes several trials, and the total time required to learn a new skill quickly adds up. Thus good sample complexity is the first prerequisite for successful skill acquisition.

- No Sensitive Hyperparameters. In the real world, we want to avoid parameter tuning for the obvious reason. Maximum entropy RL provides a robust framework that minimizes the need for hyperparameter tuning.
- Off-Policy Learning. An algorithm is off-policy if we can reuse data collected for another task. In a typical scenario, we need to adjust parameters and shape the reward function when prototyping a new task, and use of an off-policy algorithm allows reusing the already collected data.

Soft actor-critic is based on the maximum entropy reinforcement learning framework, which considers the entropy augmented objective

$$J(\pi)=E\pi[\sum tr(st,at)-\alpha log(\pi(at|st))]$$

where st and at are the state and the action, and the expectation is taken over the policy and the true dynamics of the system. In other words, the optimal policy not only maximizes the expected return (first summand) but also the expected entropy of itself (second summand). The trade-off between the two is controlled by the non-negative temperature parameter α, and we can always recover the conventional, maximum expected return objective by setting α=0. In a technical report, we show that we can view this objective as an entropy constrained maximization of the expected return, and learn the temperature parameter automatically instead of treating it as a hyperparameter.

5. SOFTWARE DESIGN

To explore the concept of virtual and real-world data collection, we created a virtual environment for data collection. Here we go through the step by step process of how we managed to get the virtual data to an ONNX framework.

- Install Unity (2019.4 or later)
- Install Python (3.6.1 or higher)
- Install the com.unity.ml-agents Unity package
- Install the com.unity.ml-agents.extensions Unity package (Optional)
- Install the mlagents Python package
- (Windows) Installing PyTorch
- Installing ML-Agents
- Create separate virtual environment for the project in project folder
- Open unity and add ml agent from package manager
- Create a prefab and objects and agent models(here it a hallway)

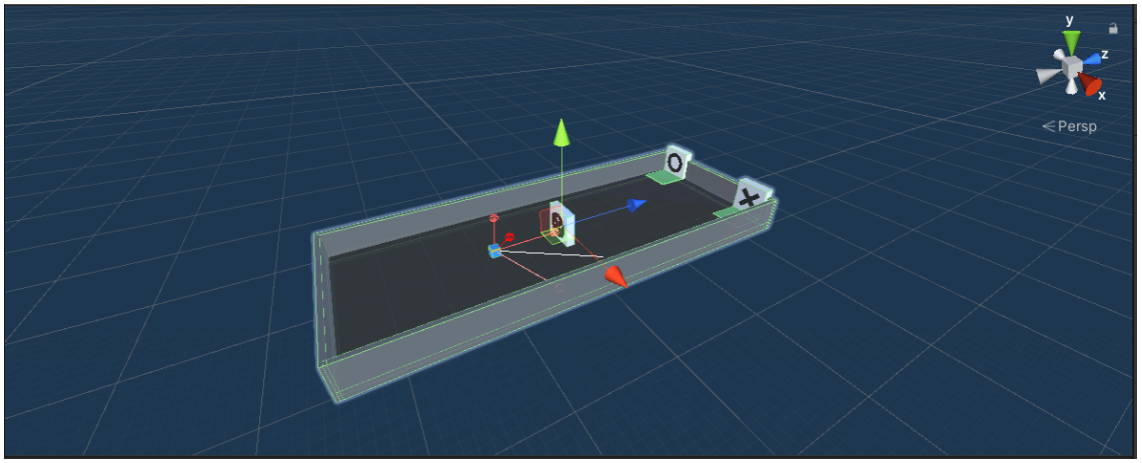


Fig. 5.1: Prefab for our project

- Add GameObjects and ML-Agent components into the agent as well as floor.
    - To do that, Create the Floor Plane
    - Add the Target Tiles
    - Add target agent
    - Group into Training Area
    - Implement an Agent: Add components and scripts to the agent
    - Initialize and reset startup and resetting of environment.
    - Observe the Environment
    - Do Agent Actions and Assigning Rewards
    - Finally setup behaviour parameters
- Run the pre-trained model.
- After it works, now we have to train a new reinforcement learning model.
- Next is training in the environment, if ML-Agent runs correctly we should be able to see in the command prompt that it is running.

Fig. 5.2: Example of training running

- We can see the mean reward here.
- We can observe the training process graphically on the tensorboard platform.
- Once the training is complete we can export the ONNX file into unity and play the sequences.
- We can take the same ONNX file and add it to a real world Agent and configure it to work.

5.1 THINGS TO LEARN BEFORE STARTING THE PROJECT

Academy - Singleton object which controls timing, reset, and training/inference settings of the environment.

Action - The carrying-out of a decision on the part of an agent within the environment.

Agent - Unity Component which produces observations and takes actions in the environment. Agents actions are determined by decisions produced by a Policy.

Decision - The specification produced by a Policy for an action to be carried out given an observation.

Editor - The Unity Editor, which may include any pane (e.g. Hierarchy, Scene, Inspector).

Environment - The Unity scene which contains Agents.

Experience - Corresponds to a tuple of [Agent observations, actions, rewards] of a single Agent obtained after a Step.

External Coordinator - ML-Agents class responsible for communication with outside processes (in this case, the Python API).

FixedUpdate - Unity method called each time the game engine is stepped. ML-Agents logic should be placed here.

Frame - An instance of rendering the main camera for the display. Corresponds to each Update call of the game engine.

Observation - Partial information describing the state of the environment available to a given agent. (e.g. Vector, Visual)

Policy - The decision making mechanism for producing decisions from observations, typically a neural network model.

Reward - Signal provided at every step used to indicate desirability of an agent's action within the current state of the environment.

State - The underlying properties of the environment (including all agents within it) at a given time.

Step - Corresponds to an atomic change of the engine that happens between Agent decisions.

Trainer - Python class which is responsible for training a given group of Agents.

Update - Unity function called each time a frame is rendered. ML-Agents logic should not be placed here.

5.2 LET'S LOOK INTO SOME ACTION CLASS UML DIAGRAMS AND ARCHITECTURE OF THE PROJECT

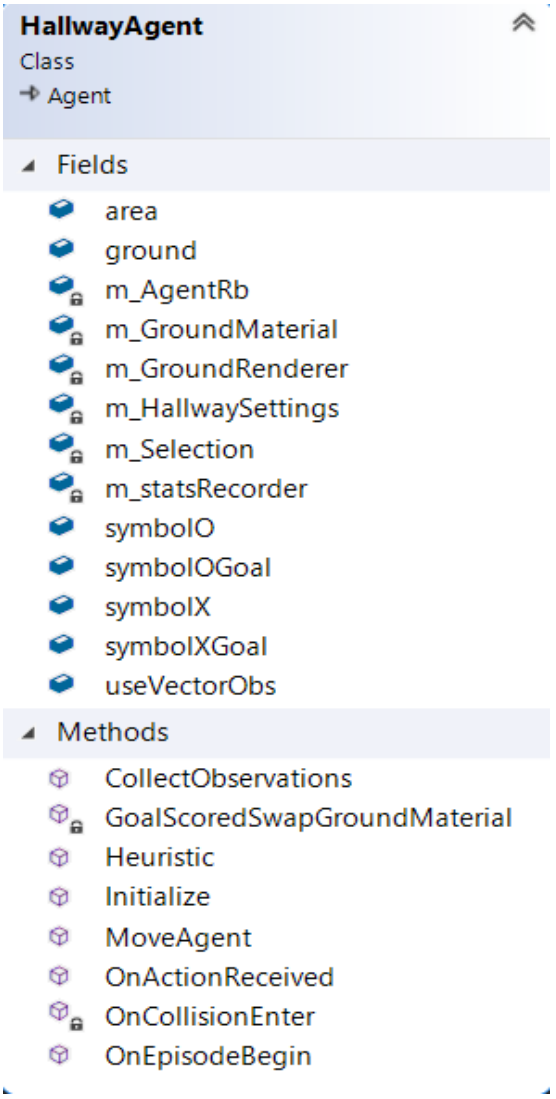


Fig. 5.3: Class diagram of hallway agent

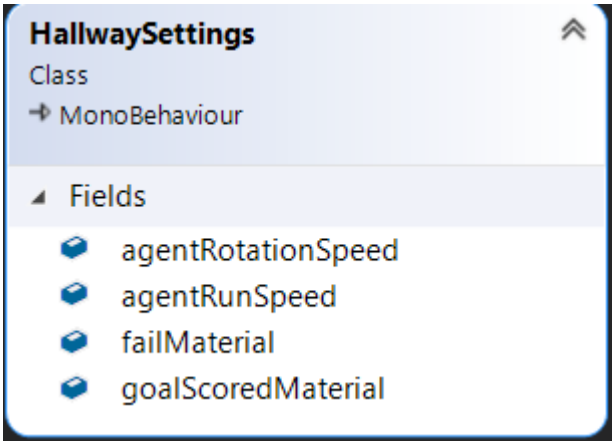


Fig. 5.4: Class diagram of hallway setting

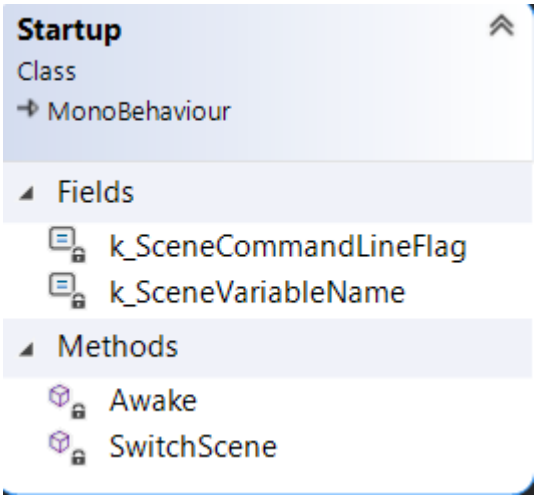


Fig. 5.5: Class diagram of startup of program

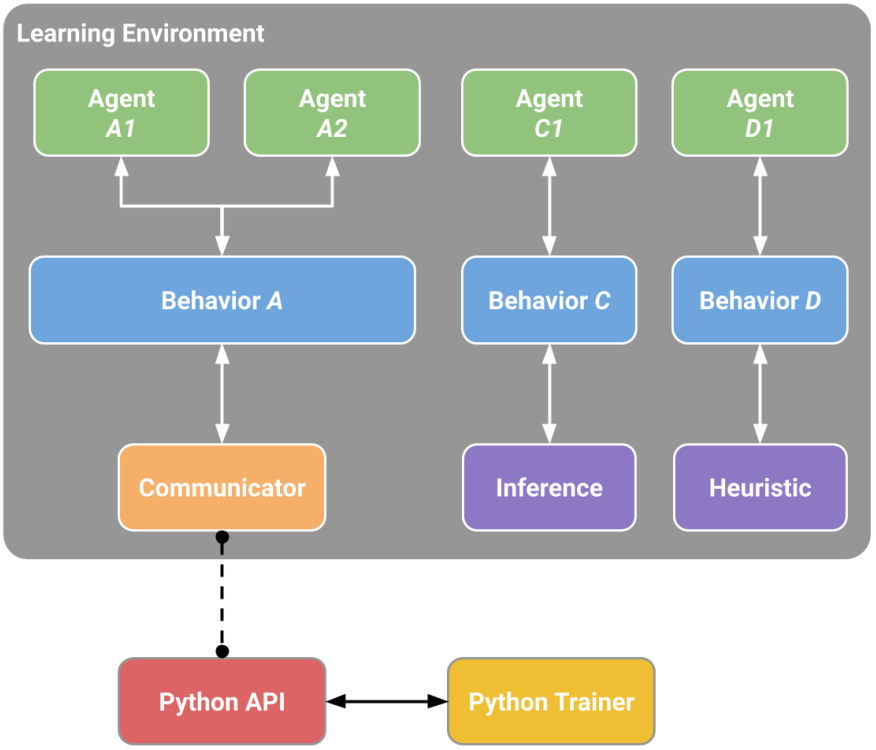


Fig. 5.6: Workflow of the program

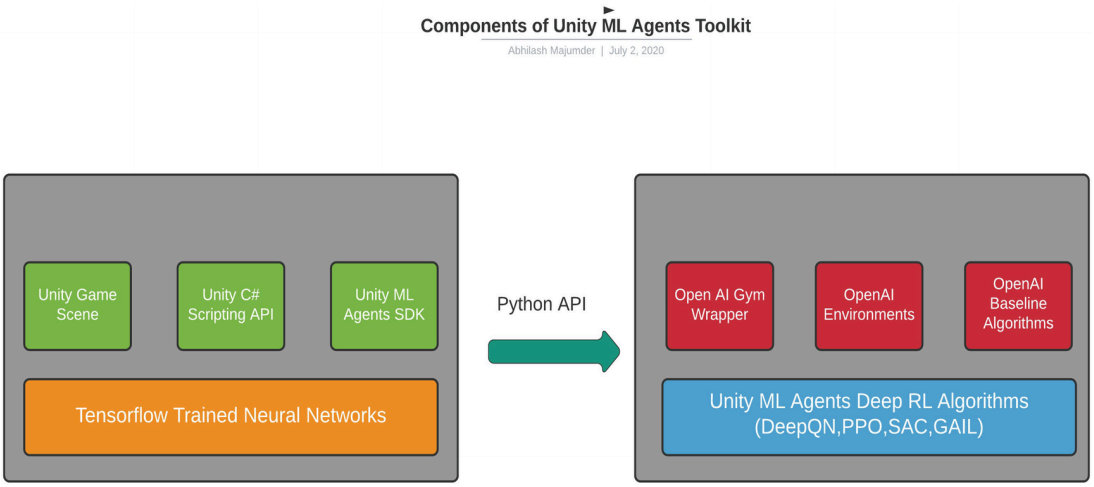


Fig. 5.7: Block Diagram of Unity ML-Agent toolkit

6. SOFTWARE AND HARDWARE REQUIREMENTS

System		Minimum requirements
Desktop		
	Operating system	Windows 7 SP1+ macOS 10.12+ Ubuntu 16.04+
	CPU	SSE2 instruction set support.
	GPU	Graphics card with DX10 (shader model 4.0) capabilities.
iOS		iOS 9.0 or higher.
Android		OS 4.1 or later ARMv7 CPU with NEON support or Atom CPU
WebGL		Any recent desktop version of Firefox, Chrome, Edge or Safari.
Universal Windows Platform		Windows 10 and a graphics card with DX10 (shader model 4.0) capabilities. OpenGL ES 2.0 or later.

Table 6.1

Machine learning is a very CPU intensive program-esque thing to be running. Here are some system requirements to adhere to Quad core Intel Core i7 Skylake or higher (Dual core is not the best for this kind of work, but manageable)

- 16GB of RAM (8GB Depending on task)
- M.2 PCIe or regular PCIe SSD with at least 256GB of storage, though 512GB is best for performance. The faster you can load and save your applications, the better the system will perform. (SATA III will get in the way of the system's performance)
- Premium graphics cards, so things with GTX 980 or 980 MX would be the best for a laptop, and 1080s or 1070s would be the best for the desktop setup.

7. CODING /CODE TEMPLATES

HALLWAYAGENT.CS

```
using System.Collections;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;


public class HallwayAgent : Agent
{
    public GameObject ground;
    public GameObject area;
```

```
public GameObject symbolOGoal;
public GameObject symbolXGoal;
public GameObject symbolO;
public GameObject symbolX;
public bool useVectorObs;
Rigidbody m_AgentRb;
Material m_GroundMaterial;
Renderer m_GroundRenderer;
HallwaySettings m_HallwaySettings;
int m_Selection;
StatsRecorder m_statsRecorder;

public override void Initialize()
{
    m_HallwaySettings = FindObjectOfType<HallwaySettings>();
    m_AgentRb = GetComponent<Rigidbody>();
    m_GroundRenderer = ground.GetComponent<Renderer>();
    m_GroundMaterial = m_GroundRenderer.material;
    m_statsRecorder = Academy.Instance.StatsRecorder;
}

public override void CollectObservations(VectorSensor sensor)
{
    if (useVectorObs)
```

```
    {
        sensor.AddObservation(StepCount / (float)MaxStep);
    }
}

IEnumerator GoalScoredSwapGroundMaterial(Material mat, float time)
{
    m_GroundRenderer.material = mat;
    yield return new WaitForSeconds(time);
    m_GroundRenderer.material = m_GroundMaterial;
}

public void MoveAgent(ActionSegment<int> act)
{
    var dirToGo = Vector3.zero;
    var rotateDir = Vector3.zero;
    var action = act[0];
    switch (action)
    {
        case 1:
            dirToGo = transform.forward * 1f;
            break;
        case 2:
            dirToGo = transform.forward * -1f;
```

```
            break;
        case 3:
            rotateDir = transform.up * 1f;
            break;
        case 4:
            rotateDir = transform.up * -1f;
            break;
    }
    transform.Rotate(rotateDir, Time.deltaTime * 150f);
                m_AgentRb.AddForce(dirToGo * m_HallwaySettings.agentRunSpeed,
ForceMode.VelocityChange);
}

public override void OnActionReceived(ActionBuffers actionBuffers)
{
    AddReward(-1f / MaxStep);
    MoveAgent(actionBuffers.DiscreteActions);
}

void OnCollisionEnter(Collision col)
{
    if(col.gameObject.CompareTag("symbol_O_Goal")||col.gameObject.CompareTa
    g("symbol_X_Goal"))
    {
```

```
        if ((m_Selection == 0 && col.gameObject.CompareTag("symbol_O_Goal")) ||
           (m_Selection == 1 && col.gameObject.CompareTag("symbol_X_Goal")))
        {
                SetReward(1f);
        StartCoroutine(GoalScoredSwapGroundMaterial(m_HallwaySettings.goalScore
        dMaterial, 0.5f));
                m_statsRecorder.Add("Goal/Correct", 1, StatAggregationMethod.Sum);
        }
        else
        {
                SetReward(-0.1f);
                StartCoroutine(GoalScoredSwapGroundMaterial(m_HallwaySettings.fai
        lMaterial, 0.5f));
                 m_statsRecorder.Add("Goal/Wrong", 1, StatAggregationMethod.Sum);
        }
        EndEpisode();
    }
}

public override void Heuristic(in ActionBuffers actionsOut)
{
    var discreteActionsOut = actionsOut.DiscreteActions;
    if (Input.GetKey(KeyCode.D))
    {
```

```
        discreteActionsOut[0] = 3;
    }
    else if (Input.GetKey(KeyCode.W))
    {
        discreteActionsOut[0] = 1;
    }
    else if (Input.GetKey(KeyCode.A))
    {
        discreteActionsOut[0] = 4;
    }
    else if (Input.GetKey(KeyCode.S))
    {
        discreteActionsOut[0] = 2;
    }
}

public override void OnEpisodeBegin()
{
    var agentOffset = -15f;
    var blockOffset = 0f;
    m_Selection = Random.Range(0, 2);
```

```
if (m_Selection == 0)
        {
           symbolO.transform.position =
            new Vector3(0f + Random.Range(-3f, 3f), 2f, blockOffset +
                    Random.Range(-5f, 5f)) + ground.transform.position;
           symbolX.transform.position =
           new Vector3(0f, -1000f, blockOffset + Random.Range(-5f, 5f))
                    + ground.transform.position;
        }
        else
        {
           symbolO.transform.position =
              new Vector3(0f, -1000f, blockOffset + Random.Range(-5f, 5f))
              + ground.transform.position;
           symbolX.transform.position =
              new Vector3(0f, 2f, blockOffset + Random.Range(-5f, 5f))
              + ground.transform.position;
        }

        transform.position = new Vector3(0f + Random.Range(-3f, 3f),
           1f, agentOffset + Random.Range(-5f, 5f))
           + ground.transform.position;
        transform.rotation = Quaternion.Euler(0f, Random.Range(0f, 360f), 0f);
        m_AgentRb.velocity *= 0f;
```

```
        var goalPos = Random.Range(0, 2);
        if (goalPos == 0)
        {
                symbolOGoal.transform.position = new Vector3(7f, 0.5f, 22.29f) +
area.transform.position;
                symbolXGoal.transform.position = new Vector3(-7f, 0.5f, 22.29f) +
area.transform.position;
        }
        else
        {
                symbolXGoal.transform.position = new Vector3(7f, 0.5f, 22.29f) +
area.transform.position;
                symbolOGoal.transform.position = new Vector3(-7f, 0.5f, 22.29f) +
area.transform.position;
        }
        m_statsRecorder.Add("Goal/Correct", 0, StatAggregationMethod.Sum);
        m_statsRecorder.Add("Goal/Wrong", 0, StatAggregationMethod.Sum);
    }
}
```

HALLWAYSETTING.CS

```
using UnityEngine;
public class HallwaySettings : MonoBehaviour
```

```
{
    public float agentRunSpeed;
    public float agentRotationSpeed;
     public Material goalScoredMaterial; // when a goal is scored the ground will use this material for a few seconds.
     public Material failMaterial; // when fail, the ground will use this material for a few seconds.
}
```

STARTUP.CS

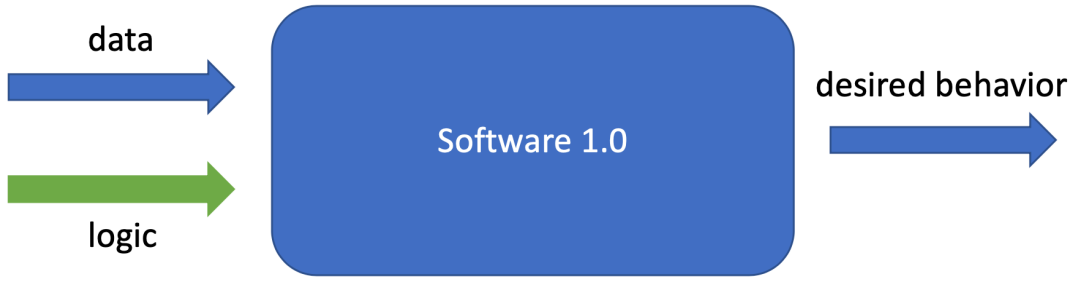
```
using System;
using UnityEngine;
using UnityEngine.SceneManagement;

namespace Unity.MLAgentsExamples
{
    internal class Startup : MonoBehaviour
    {
        const string k_SceneVariableName = "SCENE_NAME";
        const string k_SceneCommandLineFlag = "--mlagents-scene-name";

        void Awake()
        {
            var sceneName = "";
```

```
        // Check for the CLI '--scene-name' flag.  This will be used if
        // no scene environment variable is found.
        var args = Environment.GetCommandLineArgs();
            Console.WriteLine("Command line arguments passed: " + String.Join(" ",
args));
        for (int i = 0; i < args.Length; i++)
        {
           if (args[i] == k_SceneCommandLineFlag && i < args.Length - 1)
           {
              sceneName = args[i + 1];
           }
        }
 var sceneEnvironmentVariable =
Environment.GetEnvironmentVariable(k_SceneVariableName);
        if (!string.IsNullOrEmpty(sceneEnvironmentVariable))
        {
           sceneName = sceneEnvironmentVariable;
        }

        SwitchScene(sceneName);
    }

    static void SwitchScene(string sceneName)
    {
```

```
            if (sceneName == null)
            {
                Console.WriteLine(
                    $"You didn't specify the {k_SceneVariableName} environment variable or
the {k_SceneCommandLineFlag} command line argument."
                );
                Application.Quit(22);
                return;
            }
            if (SceneUtility.GetBuildIndexByScenePath(sceneName) < 0)
            {
                Console.WriteLine(
                    $"The scene {sceneName} doesn't exist within your build."
                );
                Application.Quit(22);
                return;
            }
            SceneManager.LoadSceneAsync(sceneName);
        }
    }
}
```

8. TESTING

In traditional software systems, humans write the logic which interacts with data to produce a desired behavior. Our software tests help ensure that this written logic aligns with the actual expected behavior.



However, in machine learning systems, humans provide desired behavior as examples during training and the model optimization process produces the logic of the system. How do we ensure this learned logic is going to consistently produce our desired behavior?

A typical software testing suite will include:

Unit tests which operate on atomic pieces of the codebase and can be run quickly during development.

Regression tests replicate bugs that we've previously encountered and fixed.

Integration tests which are typically longer-running tests that observe higher-level behaviors that leverage multiple components in the codebase.

Follow conventions such as:

- Don't merge code unless all tests are passing,
- Always write tests for newly introduced logic when contributing code,
- When contributing a bug fix, be sure to write a test to capture the bug and prevent future regressions.

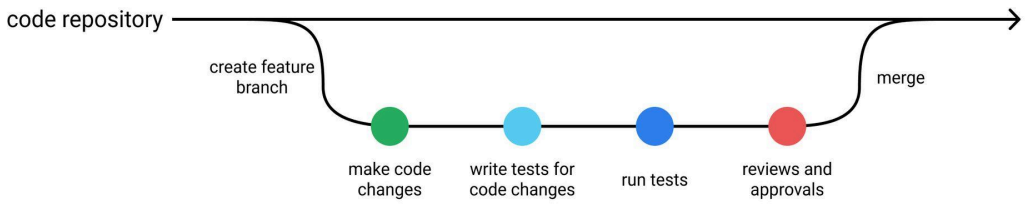


Fig. 8.1: A typical workflow for software development.

When we run our testing suite against the new code, we'll get a report of the specific behaviors that we've written tests around and verify that our code changes don't affect the expected behavior of the system. If a test fails, we'll know which specific behavior is no longer aligned with our expected output. We can also look at this testing report to get an understanding of how extensive our tests are by looking at metrics such as code coverage.

Let's contrast this with a typical workflow for developing machine learning systems. After training a new model, we'll typically produce an evaluation report including:

- performance of an established metric on a validation dataset,
- plots such as precision-recall curves,
- operational statistics such as inference speed,
- examples where the model was most confidently incorrect,

Follow conventions such as:

- save all of the hyper-parameters used to train the model,
- only promote models which offer an improvement over the existing model (or baseline) when evaluated on the same dataset.

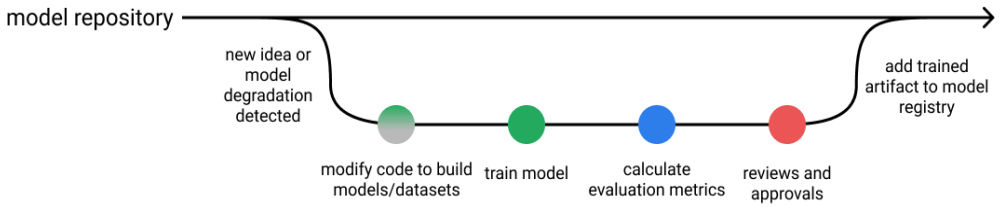


Fig. 8.2: A typical workflow for model development.

When reviewing a new machine learning model, we'll inspect metrics and plots which summarize model performance over a validation dataset. We're able to compare performance between multiple models and make relative judgements, but we're not immediately able to characterize specific model behaviors. For example, figuring out where the model is failing usually requires additional investigative work; one common practice here is to look through a list of the top most egregious model errors on the validation dataset and manually categorize these failure modes.

8.1 MODEL TESTING AND MODEL EVALUATION

While reporting evaluation metrics is certainly a good practice for quality assurance during model development, I don't think it's sufficient. Without a granular report of specific behaviors, we won't be able to immediately understand the nuance of how behavior may change if we switch over to the new model. Additionally, we won't be able to track and prevent behavioral regressions for specific failure modes that had been previously addressed.

This can be especially dangerous for machine learning systems since oftentimes failures happen silently. For example, you might improve the overall evaluation metric but introduce a regression on a critical subset of data. Or you could unknowingly add a gender bias to the model through the inclusion of a new dataset during training. We need more nuanced reports of model behavior to identify such cases, which is exactly where model testing can help.

For machine learning systems, we should be running model evaluation and model tests in parallel.

8.2 PRE-TRAIN TESTS

There's some tests that we can run without needing trained parameters. These tests include:

- check the shape of your model output and ensure it aligns with the labels in your dataset
- check the output ranges and ensure it aligns with our expectations (eg. the output of a classification model should be a distribution with class probabilities that sum to 1)

- make sure a single gradient step on a batch of data yields a decrease in your loss
- make assertions about your datasets
- check for label leakage between your training and validation datasets

The main goal here is to identify some errors early so we can avoid a wasted training job.

8.3 POST-TRAIN TESTS

However, in order for us to be able to understand model behaviors we'll need to test against trained model artifacts. These tests aim to interrogate the logic learned during training and provide us with a behavioral report of model performance.

8.4 ORGANIZING TESTS

In traditional software tests, we typically organize our tests to mirror the structure of the code repository. However, this approach doesn't translate well to machine learning models since our logic is structured by the parameters of the model.

The authors of the CheckList paper linked above recommend structuring your tests around the "skills" we expect the model to acquire while learning to perform a given task.

For example, a sentiment analysis model might be expected to gain some understanding of:

- vocabulary and parts of speech,
- robustness to noise,
- identifying named entities,
- temporal relationships,

- and negation of words.

For an image recognition model, we might expect the model to learn concepts such as:

- object rotation,
- partial occlusion,
- perspective shift,
- lighting conditions,
- weather artifacts (rain, snow, fog),
- and camera artifacts (ISO noise, motion blur).

8.5 MODEL DEVELOPMENT PIPELINE

Putting this all together, we can revise our diagram of the model development process to include pre-train and post-train tests. These tests outputs can be displayed alongside model evaluation reports for review during the last step in the pipeline. Depending on the nature of your model training, you may choose to automatically approve models provided that they meet some specified criteria.

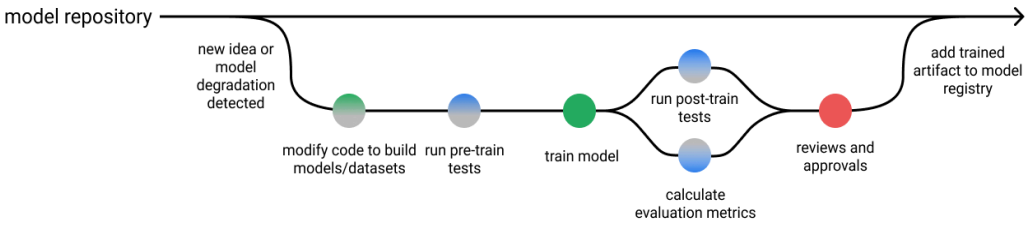


Fig. 8.3: A proposed workflow for developing high-quality models.

9. OUTPUT SCREENS

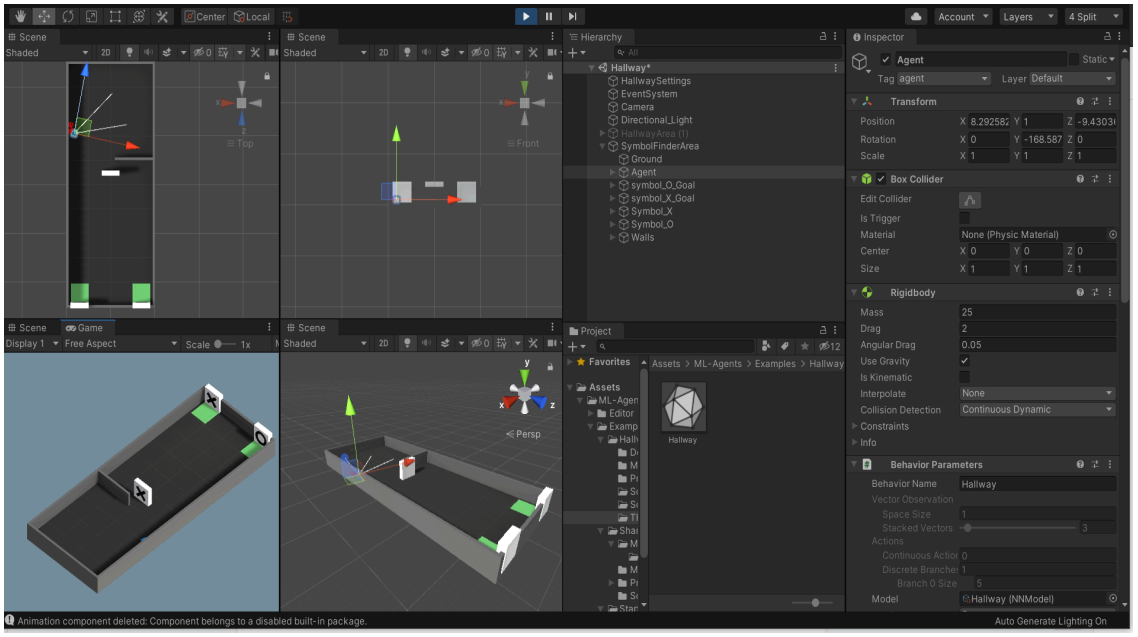


Fig. 9.1: Output screen of agent searching for “X” Mark in the Hallway

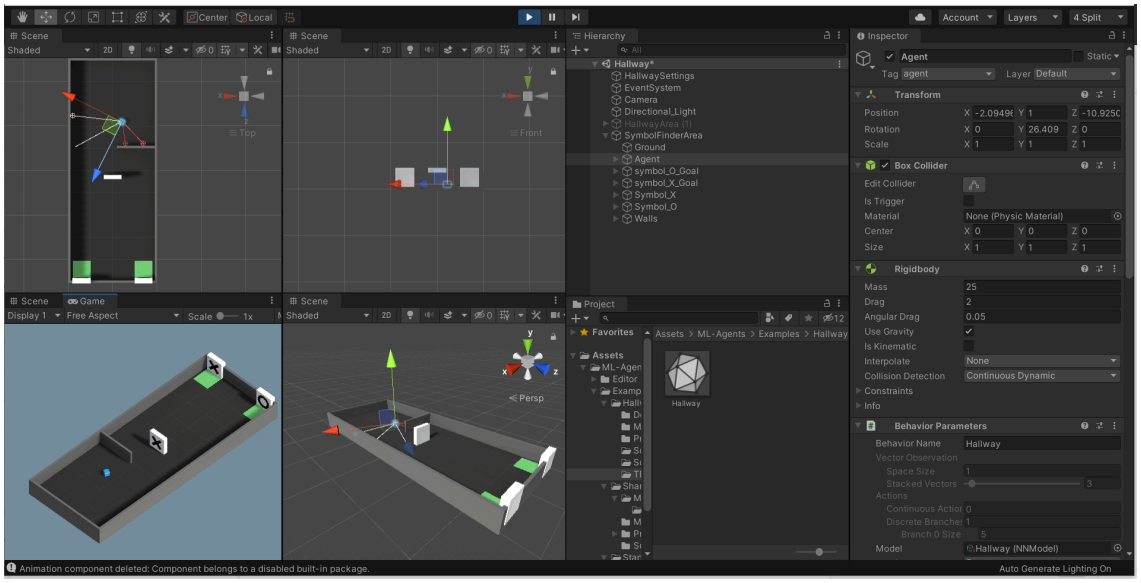


Fig. 9.2: Output screen of agent recognising obstacles in the path

Fig. 9.3: Command prompt running virtual environment within folder for python API flexibility

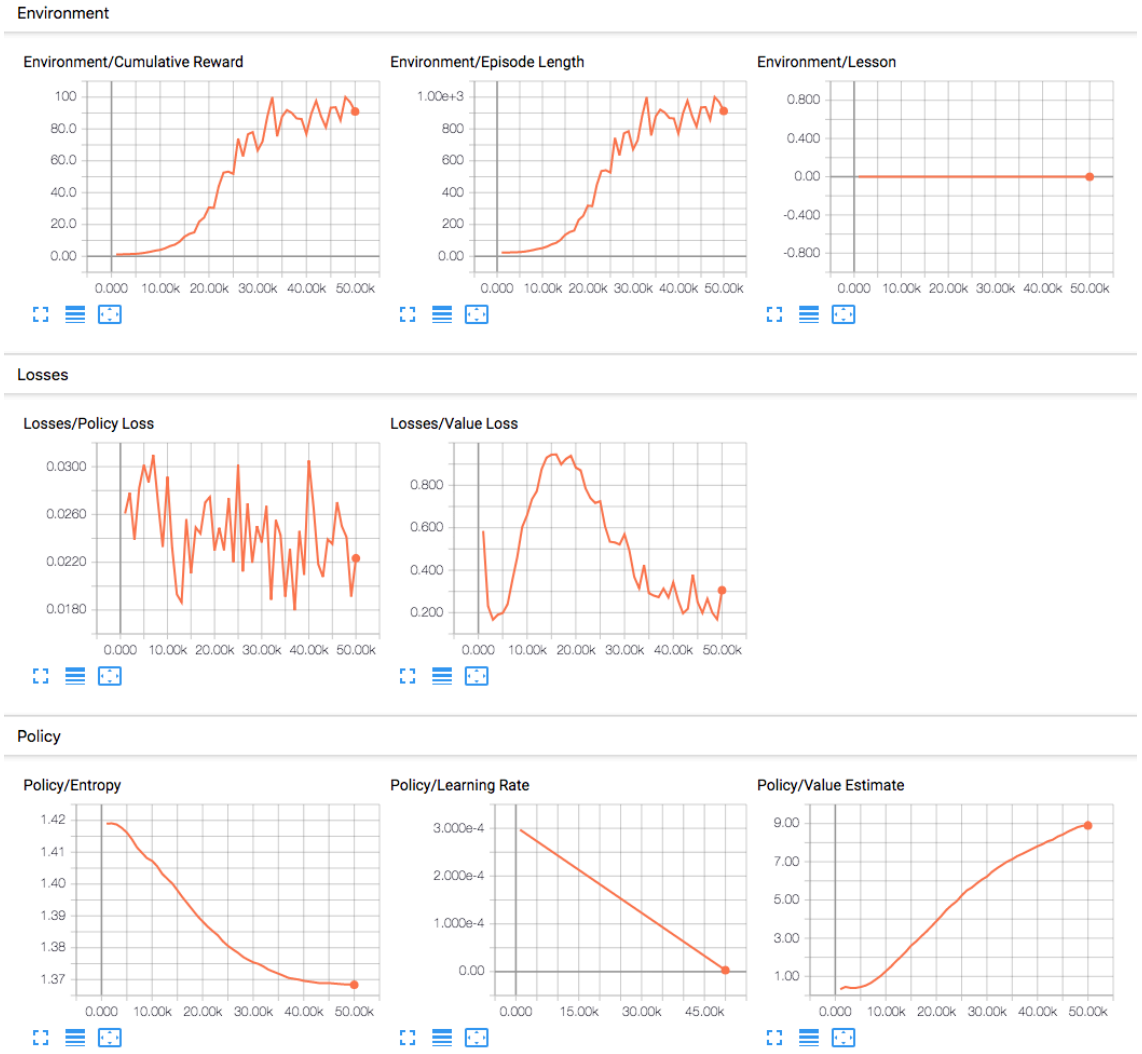


Fig. 9.4: Tensor board visualization of data

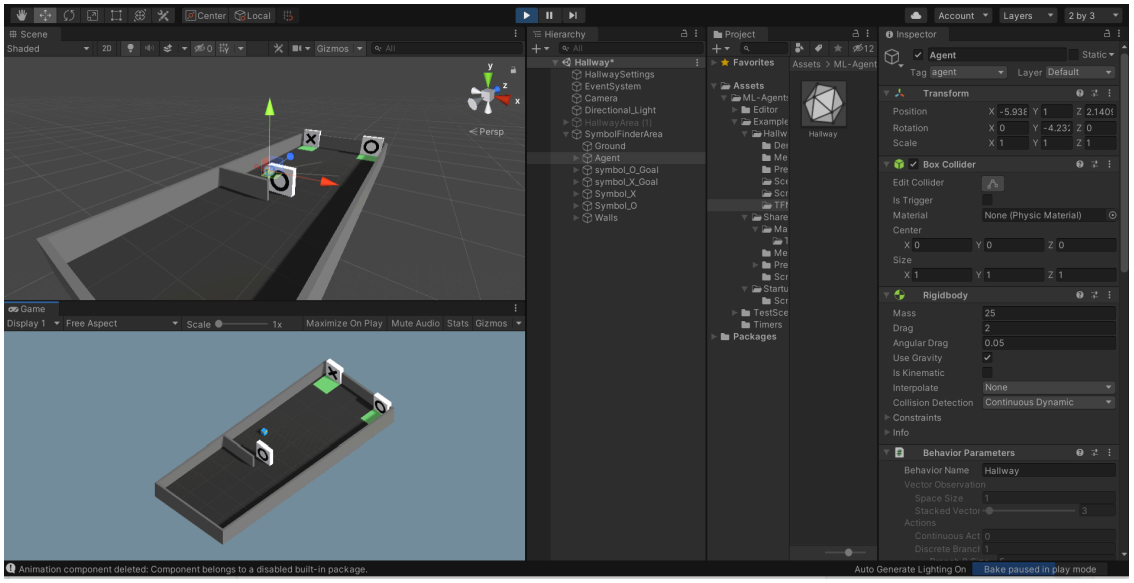
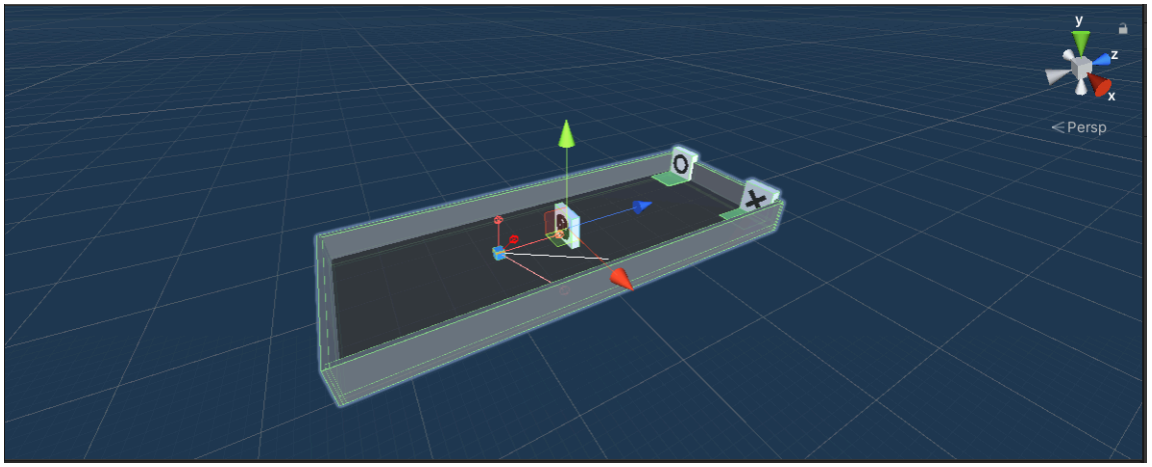


Fig. 9.5: Agent seeking for “O” after recognizing the first slab



.Fig. 9.6: Prefab of the hallway

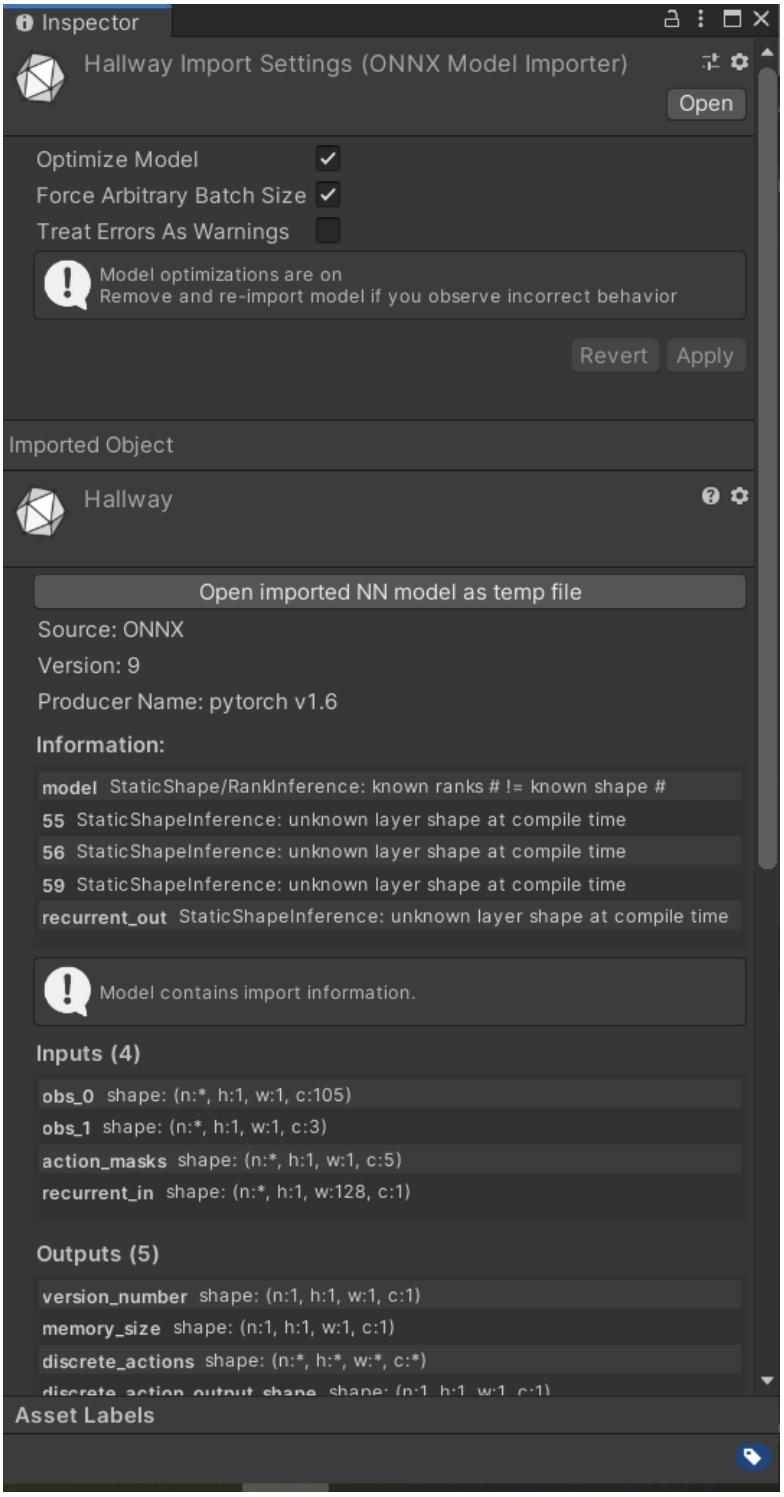


Fig. 9.7: ONNX file data

10. CONCLUSION

In this project, we trained a ML-Agent on virtual data generated from Unity and we saw that Unity's ML-Agent can be used in the real world by taking a virtual environment's training to perform the task. Robot's battery life, human fatigue, and safety considerations are among the major challenges for manual data collection. With the current settings we can minimize a great deal of labour and also minimize time. Robots may then be virtually trained to navigate in a terrain which is hard to access or that are dangerous, including the novel terrains that are currently impossible to access and to collect the data(e.g., Mars) without ever being exposed to those environments. With this approach we can train the agent on many terrains and task autonomous responses with robotics as key for development. In future this model shall be used for autonomous data collection and in unpredictable situations.

REFERENCES

[1] D. Szafir, B. Mutlu, and T. Fong, “Designing Planning and Control Interfaces to Support User Collaboration with Flying Robots,” vol. 36, no. 5–7, 2017, pp. 514– 542.

[2] J. M. Peschel and R. R. Murphy, “On the human– machine interaction of unmanned aerial system mission specialists,” IEEE Transactions on Human-Machine Systems, vol. 43, no. 1, pp. 53–62, 2013.

[3] M. A. Hsieh, A. Cowley, J. F. Keller, L. Chaimowicz, B. Grocholsky, V. Kumar, C. J. Taylor, Y. Endo, R. C. Arkin, B. Jung et al., “Adaptive teams of autonomous aerial and ground robots for situational awareness,” Journal of Field Robotics, vol. 24, no. 11-12, pp. 991– 1014, 2007.

[4]https://towardsdatascience.com/ultimate-walkthrough-for-ml-agents-in-unity3d-5603f76f68b

[5]Unity: A General Platform for Intelligent Agents, Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, Danny Lange

[6]Virtual-to-Real-World Transfer Learning for Robots on Wilderness Trails, Michael L. Iuzzolino, Michael E. Walker, Daniel Szafir