UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

3D Augmented Reality

A.Y. 2022/2023

Introduction to Reinforcement Learning & ML Agents

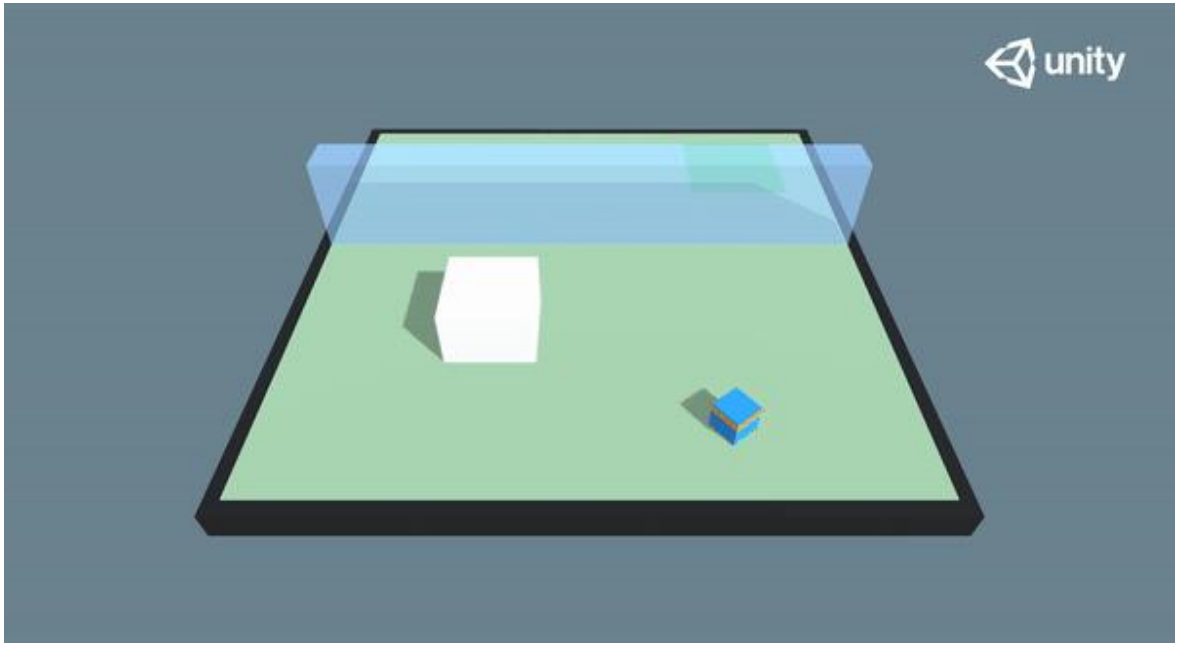
LAB experience 6

Elena Camuffo

elena.camuffo@phd.unipd.it

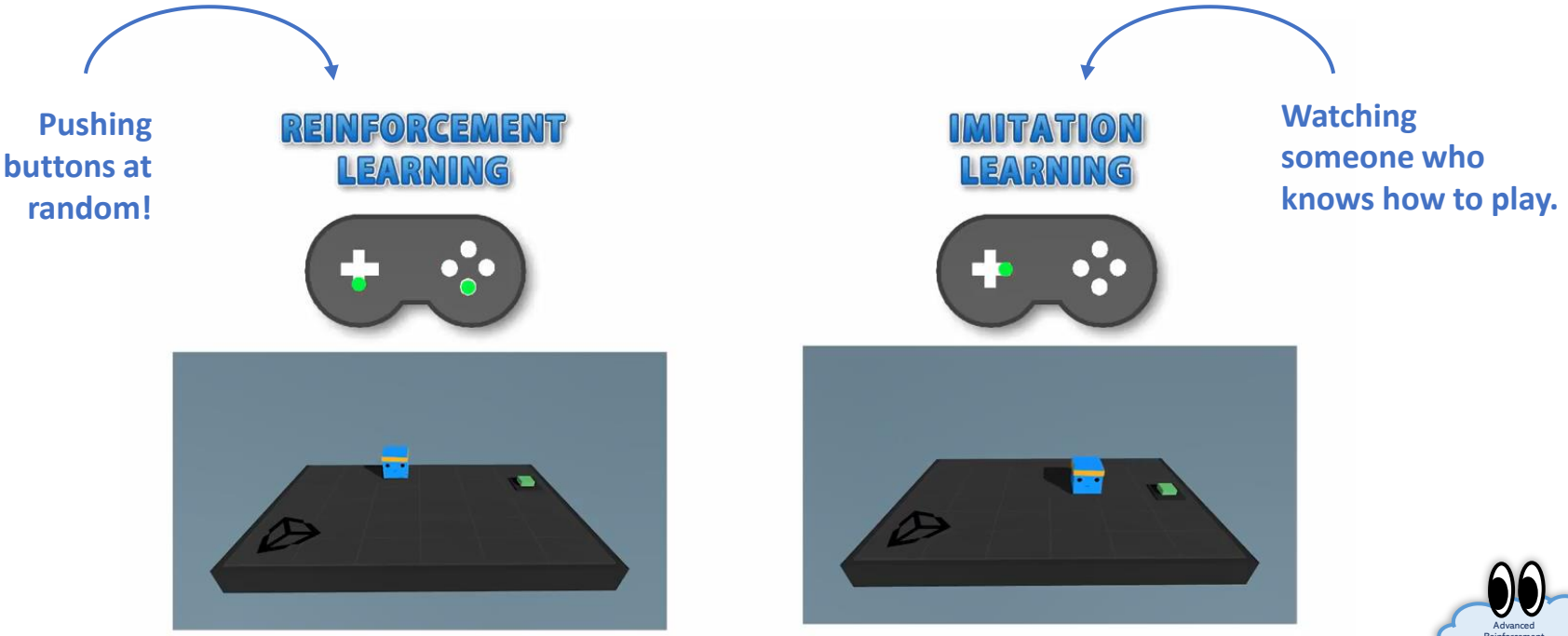
Unity Machine Learning (ML) Agents

- The **Unity Machine Learning Agents Toolkit (ML-Agents)** is an open-source plugin for Unity 3D that enables games and simulations to serve as environments for training intelligent agents.



Learning Methods

- The **Agents** in Unity can learn a *behavior* through 2 different methods:
  - **Reinforcement Learning:** learns by getting rewards.
  - **Imitation Learning:** learns by imitating what the player does.



Pills on Reinforcement Learning

- **Reinforcement Learning (RL)** is an area of machine learning for solving control tasks (also called decision problems) by building *agents* that **interact with the environment** by **performing actions**, learn through trials and errors and **receive rewards** (positive or negative) as unique feedback**.**

Introduction to Deep RL:
https://thomassimonini.medium.com/an-introduction-to-deep-reinforcement-learning-17a565999c0c

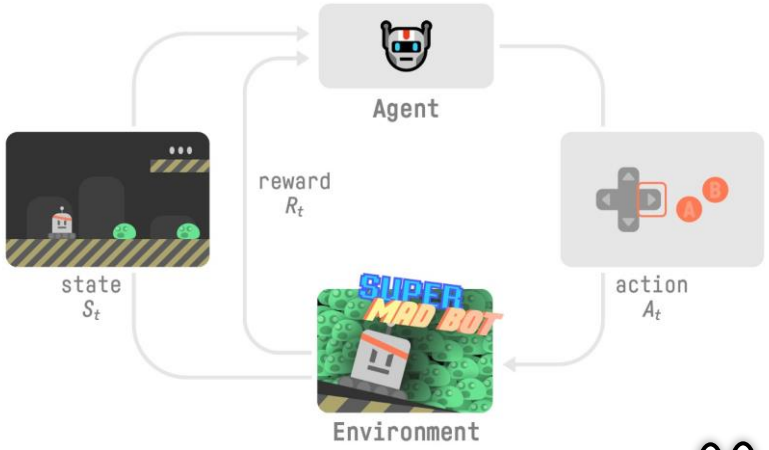
Pills on Reinforcement Learning

- The **goal** of the agent is to **maximize its cumulative reward**, called *the expected return.*

- The agent needs only the **current state** to make its decision about what **action** to take.
- He **does not need the history** of all the states and actions he took before (*Markov Decision Property).*

Formal Definitions

- **Action (A):** All the possible moves that the agent can take.
- **State (S):** Current situation returned by the environment.
- **Reward (R):** An immediate return send back from the environment to evaluate the last action.
- **Policy (π):** The strategy that the agent employs to determine next action based on the current state.
- **Value (V):** The expected long-term return with discount, as opposed to the short-term reward R. $V_\pi(s)$ is defined as the expected long-term return of the current state under policy π.
- **Q-value or action-value (Q):** like Value, except that it takes an extra parameter, the current action a. $Q_\pi(s, a)$ refers to the long-term return of the current state s, acting a under policy π.



Pills on Reinforcement Learning

- **Observation *o*:** is a **partial description of the state**, in a partially observed environment [b].
- **State *s*:** is a **complete description of the state of the world** (there is no hidden information), in a fully observed environment [a].

a.

b.

c.

- The **Action space** is the set of all possible actions in an environment. It can be:
  - *Discrete* if the number of possible actions is *finite* [b].
  - *Continuous* if the number of possible actions is *infinite* [c].

Installation

- Execute the following commands to create a **virtual environment,** activate it, and install the mlagents package together with pytorch.

```
py -m venv venv                          # create virtual environment
venv\Scripts\activate                    # activate it
python -m pip install --upgrade pip      # upgrade pip (tool to install packages)
pip install torch                        # install pytorch
pip install mlagents                     # install mlagents package
mlagents-learn –help                     # show the list of commands available
```

- Install *ML Agents package* from package manager. For this project we select **version 1.9 preview** (go to the package manager settings to enable preview packages if they are not).

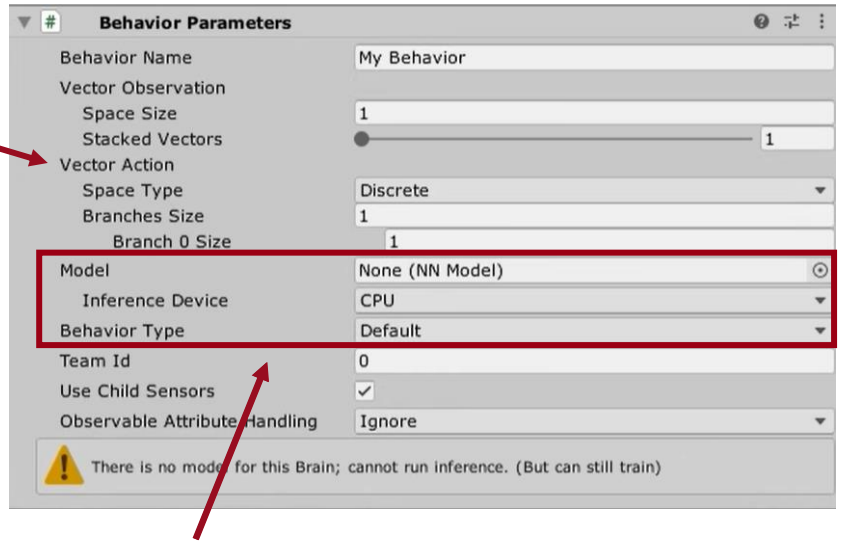
Pay attention to the ***version of python and pytorch*** that you install, as each version is compatible with a different version of ML agents package.

Agent's Behavior Parameters

- The ML agents package has a set of fundamental **components** that are required to create your agents.

The **Vector Action** parameters define the type and components for the action**.**

The **Model** variable holds the Neural Network model you want to use when you make *inference.*



- **Behavior Parameters** is the fundamental component that you must add to a Game Object to make it act as an *Agent*.
- In this component you define the specifications of your RL algorithm and you can also include a *pretrained model* to infer the agent behavior.

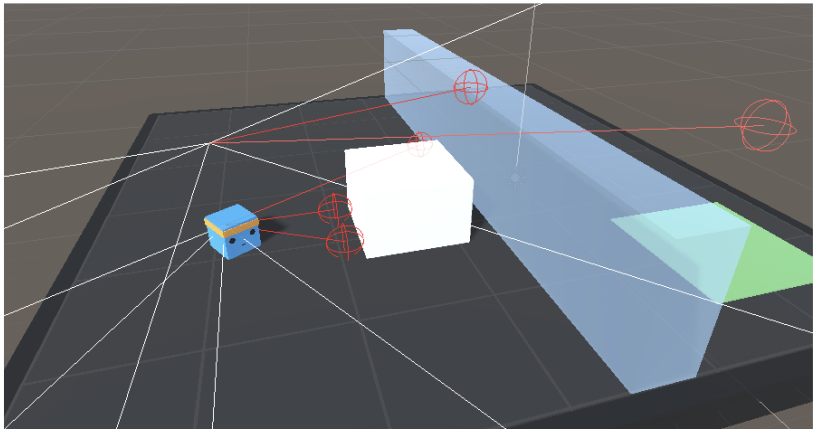
The **Behavior type** tells the agent if:

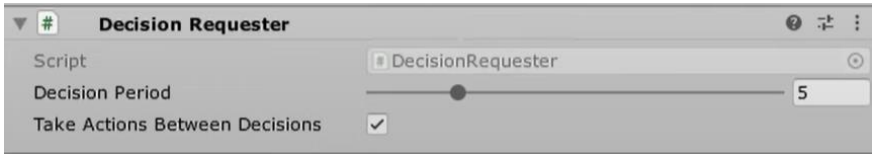
1. He has to move alone following RL (**default**).
2. He has to move guided by the user (**heuristic**).
3. He has to move alone following a trained model (**inference**).

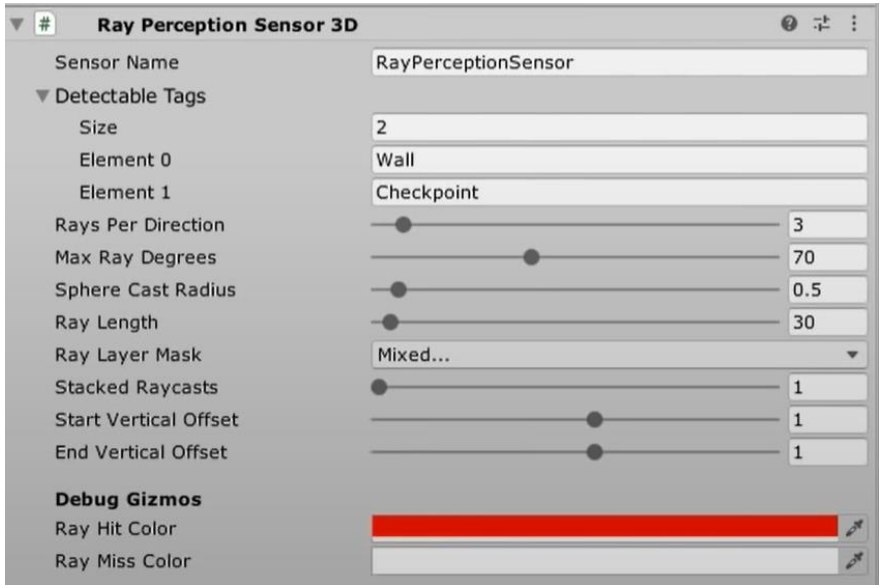
Example from: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Create-New.md

Other main modules

- To train a model from scratch you need to include the **Decision Requester** module in the agent, to allow it deciding which action to take.
- To make the agent perceive the world with raycasting, you can use the **Ray Perception Sensor 3D** module.







Agent Script Example

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

public class RollerAgent : Agent
{
    Rigidbody rBody;
    void Start()
    {
        rBody = GetComponent<Rigidbody>();
    }

    public Transform Target;
    public override void OnEpisodeBegin()
    {
        // If the Agent fell, zero its momentum
        if (this.transform.localPosition.y < 0)
        {
            this.rBody.angularVelocity = Vector3.zero;
            this.rBody.velocity = Vector3.zero;
            this.transform.localPosition = new Vector3(0, 0.5f, 0);
        }

        // Move the target to a new spot
        Target.localPosition = new Vector3(Random.value*8-4, 0.5f, Random.value*8-4);
    }

    public override void CollectObservations(VectorSensor sensor)
    {
        // Target and Agent positions
        sensor.AddObservation(Target.localPosition);
        sensor.AddObservation(this.transform.localPosition);

        // Agent velocity
        sensor.AddObservation(rBody.velocity.x);
        sensor.AddObservation(rBody.velocity.z);
    }
```

OnEpisodeBegin()

Overridden, this method is called each time a new episode begins. The process of training involves running **episodes** where the Agent attempts to solve the task. Each episode lasts until the Agents solves the task, fails or times out.

CollectObservations()

Overridden method. The Agent sends the information we collect to the Brain, which uses it to make a decision.

Agent Script Example

OnActionReceived()

Overridden, this method receives actions and assigns rewards. The rewars are assigned through method **SetReward()** as positive if the target is reached and negative or null if not.
Once the reward is assigned begin a new epidsode with method **EndEpisode()**.

Heuristic()

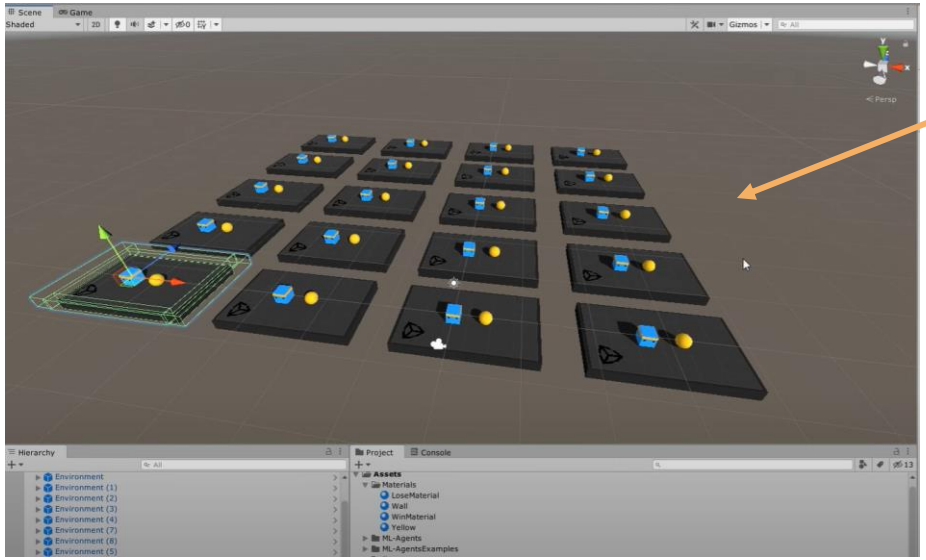
Overridden method. It provides the commands needed to generate the actions that correspond to the actions of the Agent when performing RL.

```
    public override void OnActionReceived(ActionBuffers actionBuffers)
    {
        // Actions, size = 2
        Vector3 controlSignal = Vector3.zero;
        controlSignal.x = actionBuffers.ContinuousActions[0];
        controlSignal.z = actionBuffers.ContinuousActions[1];
        rBody.AddForce(controlSignal * forceMultiplier);

        // Rewards
        float distanceToTarget =
Vector3.Distance(this.transform.localPosition, Target.localPosition);

        // Reached target
        if (distanceToTarget < 1.42f)
        {
            SetReward(1.0f);
            EndEpisode();
        }

        // Fell off platform
        else if (this.transform.localPosition.y < 0)
        {
            EndEpisode();
        }
    }

    public override void Heuristic(in ActionBuffers actionsOut)
    {
        var continuousActionsOut = actionsOut.ContinuousActions;
        continuousActionsOut[0] = Input.GetAxis("Horizontal");
        continuousActionsOut[1] = Input.GetAxis("Vertical");
    }
}
```

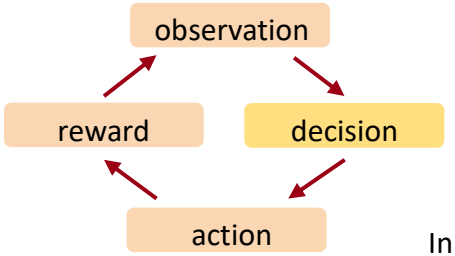
Train your model

- To train your agent you need to use the **terminal** with the *mlagents-learn command*.

```
mlagents-learn --help                          # prints all the commands
mlagents-learn --force                         # overwrites the last run
mlagents-learn --run-id=test                   # change the run name to keep it separate
mlagents-learn config/ConfigFile.yaml          # include config file ConfigFile.yaml
mlagents-learn –initialize-from=LastRun        # continue run LastRun
```



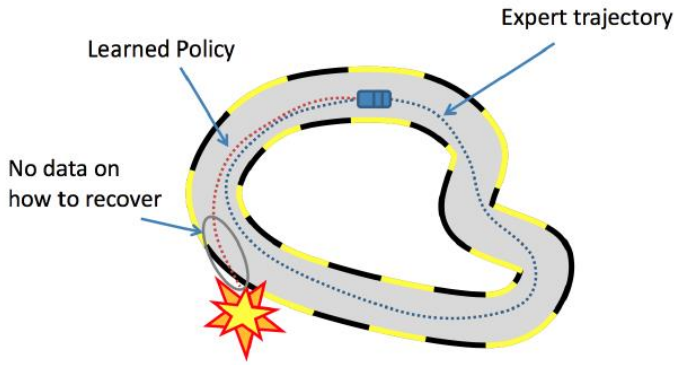
To make the training faster, you can **duplicate your environment** (agent, goal and everything the agent can interact with) several times.



In training loop, decisions are taken by the **Brain.**

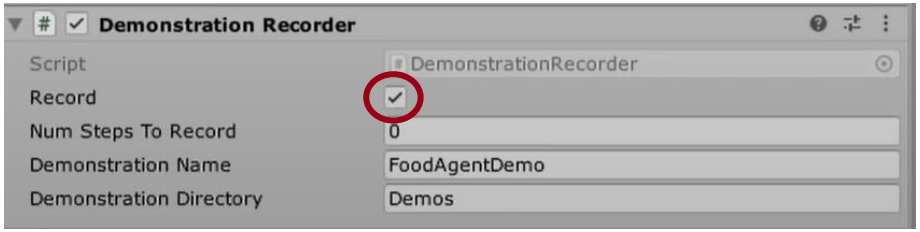
Imitation Learning

- Usually **Imitation Learning** is used when *Reinforcement Learning* is **not enough** to make the training converge to your desired behaviour (it needs a little help!).



- The **techniques** used are:
  - *Generative Adversarial Imitation Learning (GAIL):* A discriminator **guesses** on the trajectory and tells if it comes from the Agent or from the demo.
  - *Behavioral Cloning:* Repeat the path provided by the demo.

- **Demonstration Recorder** is used when performing *imitation learning*: enable it when you want to record your actions to be *imitated*.



Test and improve your model

- Once your model is **trained** you can integrate it inside the *behavior parameters:* the model is the **ONNX file** you can find in the folder with the name of your training inside your Assets.
- Change to *inference* mode and play: the agent will make in practice what he has learnt durning training. However, sometimes it can be **not sufficient** and further training is needeed.
- If you want to set some parameters for the training you can integrate a **configuration file .yaml**.

Parameters used for **imitation learning**

config/RollerBall.yaml

```
behaviors:
  RollerBall:
    trainer_type: ppo
    hyperparameters:
      batch_size: 10
      buffer_size: 100
      learning_rate: 3.0e-4
      beta: 5.0e-4
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
      beta_schedule: constant
      epsilon_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 128
      num_layers: 2
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
      gail:
        strength: 0.5
        demo_path: /demo
    behavioral_cloning:
      strength: 0.5
      demo_path: /demo
max_steps: 500000
    time_horizon: 64
    summary_freq: 10000
```

You can find a lot of examples here:
https://github.com/Unity-Technologies/ml-agents

Look at the results

The time to reach the goal is progressively shorter.

Loss is decreasing,
Value is increasing:
Agent is learning.

Policy is the deep learning method that make agent learn.

- While *training* you can check how it is going so far via **Tensorboard.**
- Tensorboard is a component of Deep Learning frameworks that provides **statistics of some parameters** during training.
- Start tensorboard with:

```
tensorboard --logdir=mylogdir –port=8080
```

- And open it going to: https://localhost:8080/

TensorBoard

EXERCISE: Xmas Agents

1. **Install and setup Unity ML agents**

   Install the required python stuff and ML agents' package (Unity ML agents 1.9 preview).

2. **Complete the Agent Script**

   Fill the missing functions in the agent's script.

   Suggestion: Look at the example script in these slides to complete the methods.

3. **Set up the environment**

   Test the *heuristic* configuration to ensure everything works fine. Create a prefab of your environment and duplicate it several times to set up the training.

4. **Train your model**

   Start the training from command line and press play. Open another terminal and open tensorboard to see the progresses in your training.

5. **Test your model**

   Test your trained model including it inside the agent's parameters.

6. **(OPTIONAL) Imitation Learning**

   Record the behavior of your agent and add the necessary parameters in a *config* file to try the training via imitation learning (you can make your task more complex adding obstacles or other goals).