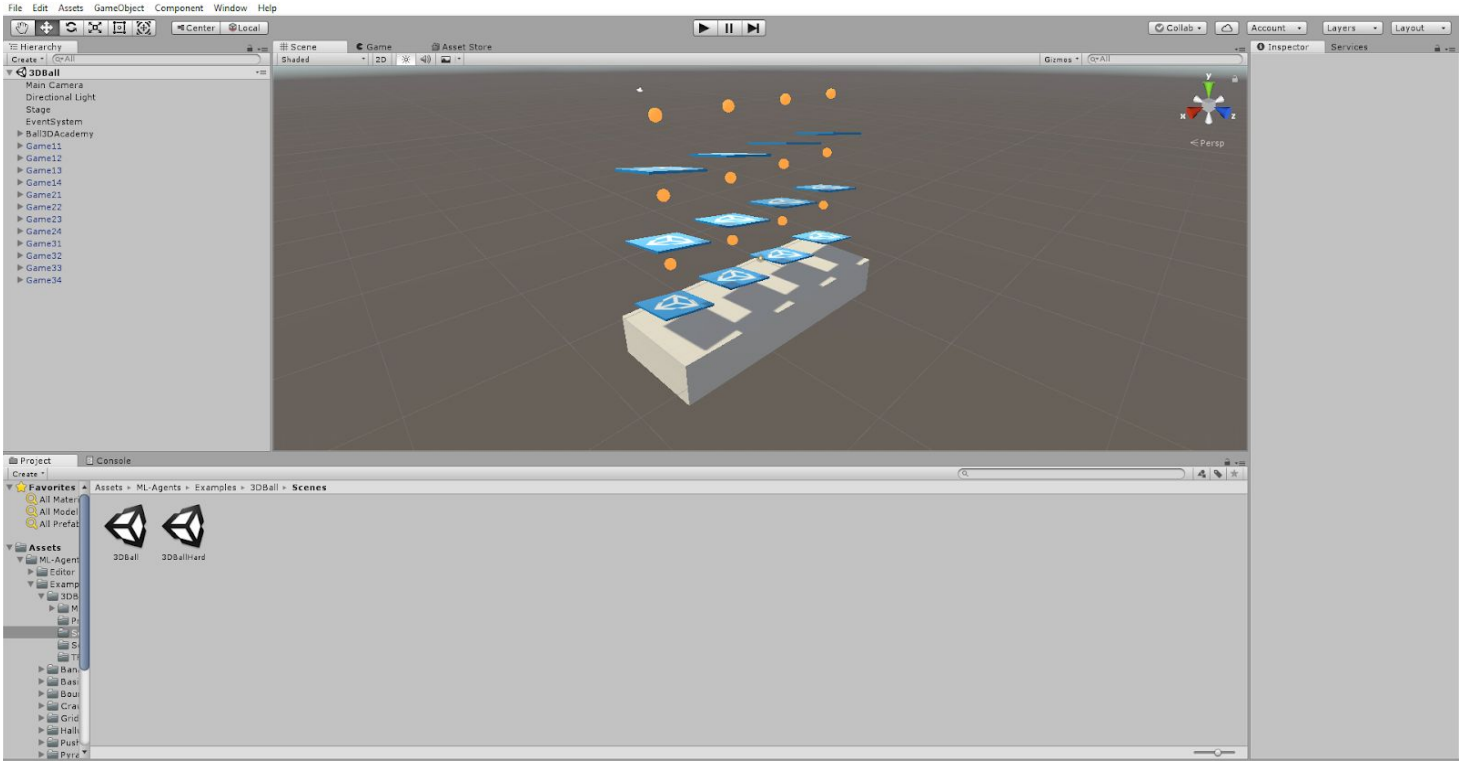
ML_Agents

INTRODUCTION

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 an OS X-exclusive game engine. As of 2018, the engine has been extended to support 27 platforms and has more applications than just game development.

ML-Agents is one such application in the shape of an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents. Agents can be trained using reinforcement learning, imitation learning, neuroevolution, or other machine learning methods through a simple-to-use Python API. The availability of this plugin and the engine itself makes it an interesting option to further delve into more complex topics within AI. And so, the purpose of this document is to show some basic usage of said plugin as well as the some of the theoretical background upon which it's constructed.

PRACTICAL APPLICATION EXAMPLE



Upon starting the engine with the project we will find ourselves in this type of scenario. In here we can access to the different demo scenes to train our agents. However, to do so we have to set some things first. First of all is selecting the scene and within it we have to set the Brain script's type to External. Said Brain the Agents use will be a child of the Academy in the Unity scene hierarchy. After that the scene has to be saved and then a game built from our selected scene.

A command or terminal window has to be opened after this and optionally navigate to the folder in which the copy of the project (not the game we just built) is. From here the next command has to be run:

```
mlagents-learn <trainer-config-path> --run-id=<run-identifier> --train
```

- `<trainer-config-path>` is the relative or absolute file path of the trainer configuration. The defaults used by example environments included in the project can be found in config/trainer_config.yaml.
- `<run-identifier>` is a string used to separate the results of different training runs.
- `--train` tells `mlagents-learn` to run a training session (rather than inference).

This will launch a training session with Reinforcement Learning as it's main method. Said training session can be stopped by pressing Ctrl+C. The trained model will be at `models/<run-identifier>/editor_<academy_name>_<run-identifier>.bytes` where `<academy_name>` is the name of the Academy GameObject in the current scene. This file corresponds to the model's latest checkpoint. This training model can now be embed to an Internal Brain by dragging the trained model to the Graph Model placeholder in the scene's Brain inspector window. To finally witness it in action we only have to press the Play button of the scene.

Of course, some explanations are due to what some of these terms mean and what part they take in the agent training process.

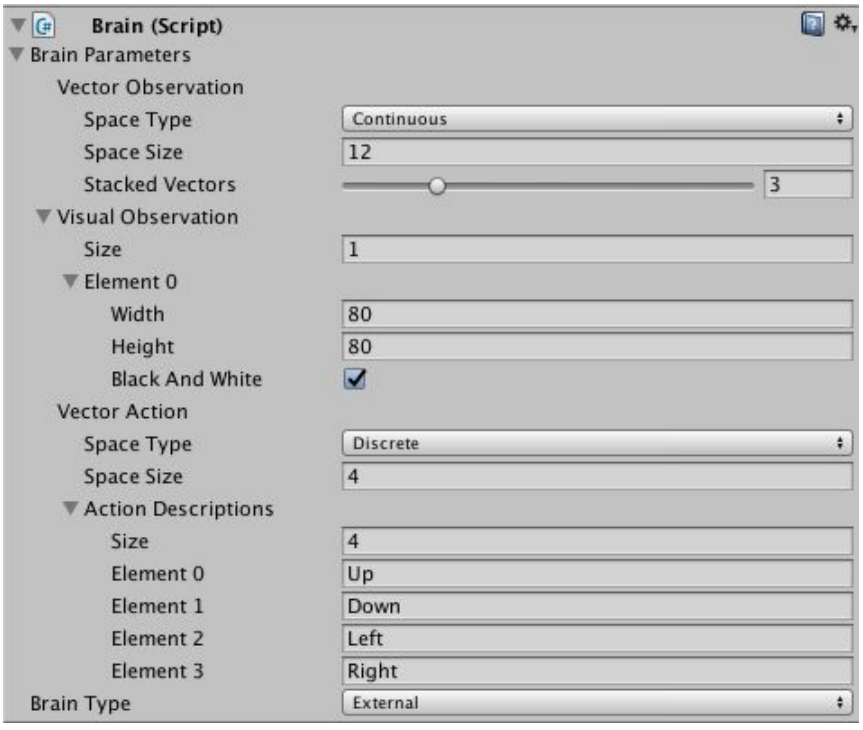
An Academy orchestrates all the Agent and Brain objects in a Unity scene. Every scene containing Agents must contain a single Academy. To use an Academy, a subclass must be created. However, all the methods that can be overridden are optional. The Academy's methods allow to do the following:

- Initialize the environment after the scene loads
- Reset the environment
- Change things in the environment at each simulation step

The Brain encapsulates the decision making process. Every Agent must be assigned a Brain, but the same Brain can be used with more than one Agent. Several Brains can be created, which can be associated to one or more Agents. The toolkit has up to 4 Brain types implemented:

- **External**: Used when training agents, It can also be used to communicate with a Python script.
- **Internal**: Used to make use of a trained model.
- **Heuristic**: Used to hand-code the Agent's logic by extending the Decision class.
- **Player**: Used to map keyboard keys to Agent actions, which can be useful to test the Agent code.

The Brain class has several important properties that can set using the Inspector window, located on the right side of the engine. These properties must be appropriate for the Agents using the Brain. The Brain Inspector window in the Unity Editor displays the properties assigned to a Brain component in this fashion:



These properties include things such as: vector observations, visual observation, and vector actions for the Brain, height, width, and whether to grayscale visual observations for the Brain, action descriptors, arrays of discrete actions, length of action vector for the Brain, etc.

The project includes a Python API that allows direct interaction with the Unity game engine as well as a collection of trainers and algorithms to train agents in Unity environments. This package contains 2 components: a low level API which allows the direct interaction with a Unity Environment (mlagents.envs) and an entry point to train (mlagents-learn) which allows the training of agents in Unity Environments using implementations of reinforcement learning or imitation learning.

THEORY

Machine learning

Machine learning, a branch of artificial intelligence, focuses on learning patterns from data. The three main classes of machine learning algorithms include: unsupervised learning, supervised learning and reinforcement learning. Each class of algorithm learns from a different type of data.

Unsupervised learning

In this type of learning you only have input data (X) and no corresponding output variables. The goal for unsupervised learning is to model the underlying structure or distribution in the data in order to learn more about the data. These are called unsupervised learning because there is no correct answers and there is no 'teacher'. Algorithms are left to their own devises to discover and present the interesting structure in the data.

Unsupervised learning problems can be further grouped into clustering and association problems.

- **Clustering**: A clustering problem is where you want to discover the inherent groupings in the data, such as grouping customers by purchasing behavior.
- **Association**: An association rule learning problem is where you want to discover rules that describe large portions of your data, such as people that buy X also tend to buy Y.

Some popular examples of unsupervised learning algorithms are k-means and Apriori algorithms.

Supervised learning

In supervised learning, it isn't wanted to just group similar items but directly learn a mapping from each item to the group (or class) that it belongs to. This method will have some input variables (x) and an output variable (Y) and you use an algorithm to learn the mapping function from the input to the output. It is called supervised learning because the process of an algorithm learning from the training dataset can be thought of as a teacher supervising the learning process. We know the correct answers, the algorithm iteratively makes predictions on the training data and is corrected by the teacher. Learning stops when the algorithm achieves an acceptable level of performance. Supervised learning problems can be further grouped into regression and classification problems.

- **Classification**: A classification problem is when the output variable is a category, such as "red" or "blue" or "disease" and "no disease".
- **Regression**: A regression problem is when the output variable is a real value, such as "dollars" or "weight".

Some common types of problems built on top of classification and regression include recommendation and time series prediction respectively. Some popular examples of machine learning algorithms are the following: Linear regression, Random forest and Support vector machines.

For both supervised and unsupervised learning, there are two tasks that need to be performed: attribute selection and model selection. Attribute selection (also called feature selection) pertains to selecting how we wish to represent the entity of interest. Model selection, on the other hand, pertains to selecting the algorithm (and its parameters) that perform the task well. Both of these tasks are active areas of machine learning research and, in practice, require several iterations to achieve good performance.

Reinforcement learning

Reinforcement learning can be viewed as a form of learning for sequential decision making that is commonly associated with controlling robots. Consider an autonomous firefighting robot that is tasked with navigating into an area, finding the fire and neutralizing it. At any given moment, the robot perceives the environment through its sensors, processes this information and produces an action. In other words, it is continuously making decisions about how to interact in this environment given its view of the world and objective. Teaching a robot to be a successful firefighting machine is precisely what reinforcement learning is designed to do. The goal of reinforcement learning is to learn a policy, which is essentially a mapping from observations (what the robot can measure from its environment) to actions(is a change to the configuration of the robot).

One common aspect of all three branches of machine learning is that they all involve a training phase and an inference phase. While the details of the training and inference phases are different for each of the three, at a high-level, the training phase involves building a model using the provided data, while the inference phase involves applying this model to new, previously unseen, data.

TensorFlow

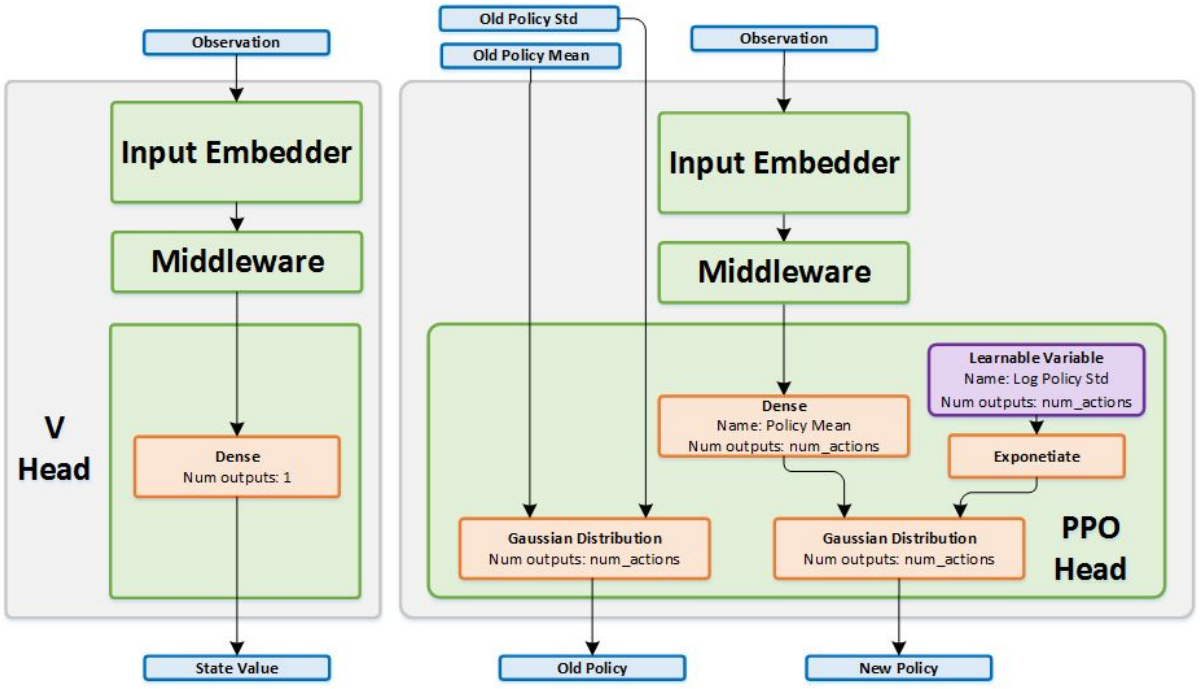
Many of the algorithms that are provided in the ML-Agents toolkit leverage some form of deep learning. More specifically, the implementations are built on top of the open-source library TensorFlow. This means that the models produced by the ML-Agents toolkit are in a format only understood by TensorFlow.

TensorFlow is an open source library for performing computations using data flow graphs, the underlying representation of deep learning models. Thanks to it’s flexibility It facilitates training and inference on CPUs and GPUs in a desktop, server, or mobile device. Within the ML-Agents toolkit, when you train the behavior of an agent, the output is a TensorFlow model (.bytes) file that you can then embed within an Internal Brain. Unless a new algorithm is implemented, the use of TensorFlow is mostly abstracted away and behind the scenes.

Types of training

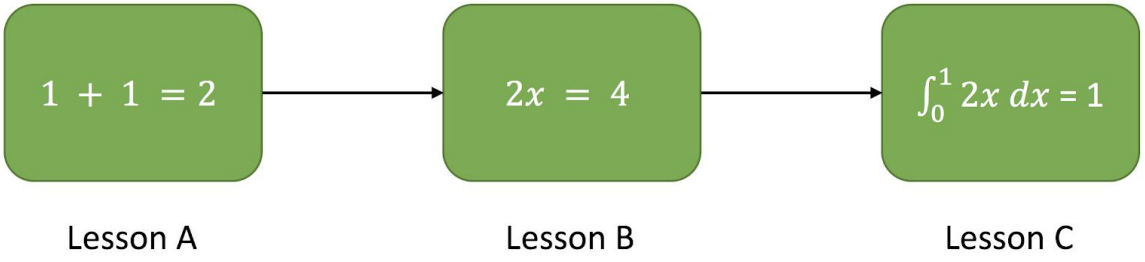
Proximal Policy Optimization:

PPO uses a neural network to approximate the ideal function that maps an agent's observations to the best action an agent can take in a given state, actualizing it's policies as it learns.



Curriculum training:

Curriculum learning is a way of training a machine learning model where more difficult aspects of a problem are gradually introduced in such a way that the model is always optimally challenged.



Imitation training:

It is often more intuitive to simply demonstrate the behavior we want an agent to perform, rather than attempting to have it learn via trial-and-error methods. The imitation learning algorithm will use the pairs of observations and actions from the human player to learn a policy. The idea of teaching by imitation has been around for many years; however, the field is gaining attention recently due to advances in computing and sensing as well as rising demand for intelligent applications.

Refferences:

https://github.com/Unity-Technologies/ml-agents
https://arxiv.org/abs/1809.02627

https://machinelearningmastery.com
https://coach.nervanasys.com/
https://www.tensorflow.org/