Leveraging Unity ML Agents for Intelligent Game AI: A Practical Implementation

Deac Melinda-Anca

01.05.2025

Contents

Abstract

This paper explores the use of Unity's ML Agents Toolkit to train intelligent agents using reinforcement learning in game environments. By integrating Python-based machine learning with Unity's real-time 3D engine, ML Agents allow developers to create AI systems capable of learning complex tasks through trial and error. This paper provides a comprehensive overview of the ML Agents framework, covering installation, environment configuration, training workflow, and deployment. Through hands-on implementation and testing, this work highlights the powerful capabilities of ML Agents for building adaptive, goal-driven behaviors in games and simulations.

1 Introduction

Artificial intelligence has become a key component in game development, allowing developers to create more immersive and challenging experiences. Traditional rule-based systems are often limited in complexity and adaptability, especially in dynamic or unpredictable environments. Machine learning, and specifically reinforcement learning (RL), offers a promising alternative by enabling agents to learn optimal behaviors through interaction with their environment.

Unity, a popular game engine, provides the ML Agents Toolkit to bridge the gap between modern AI techniques and real-time game development. ML Agents combine Unity's engine with a Python API, enabling training of agents through simulation. This paper aims to present ML Agents as a tool for implementing intelligent behavior without hardcoded logic, emphasizing the benefits of reinforcement learning within Unity.

2 Background & Toolkit Overview

Unity ML Agents is an open-source toolkit developed by Unity Technologies to facilitate machine learning in game-like environments. It enables training of agents using reinforcement learning, imitation learning, and other algorithms via interoperability with Python.

Key Components

- **Unity Environment**: The simulation where the agent lives and interacts. This includes visual components and physics.
- **ML Agents Python Package**: A Python package built on PyTorch that runs the actual machine learning algorithms.
- **C# API**: Used to define agents, actions, and observations within Unity.
- **ONNX Brain Models**: The output of training, these models are loaded into Unity to control agents in inference mode.

Workflow Overview

1. **Observation**: The agent collects data from the environment.
2. **Decision/Action**: Based on observations, the agent chooses an action.
3. **Reward**: The agent receives a numerical reward based on the outcome of its action.
4. **Training**: The data is sent to the Python backend for training.
5. **Inference**: A trained model is imported into Unity for real-time use.

This loop of observe-act-reward-train mimics how animals or humans learn, and it represents the foundation of reinforcement learning.

3 Setup & Installation

ML Agents requires the proper configuration of both Unity and Python environments.

Step-by-Step Setup with Anaconda

1. **Install Anaconda Navigator**: Download and install from `https://www.anaconda.com/`.
2. **Create a Virtual Environment**: Execute the following command in a terminal or Anaconda Prompt:

   ```
   conda create -n mlagents python=3.10.12 && conda activate mlagents
   ```

3. **Install PyTorch**: Follow installation instructions from `https://pytorch.org/get-started/locally/` depending on the operating system. Example:

   ```
   conda install pytorch torchvision torchaudio cpuonly -c pytorch
   ```

4. **Install ML Agents Toolkit**:

   ```
   pip install mlagents
   ```

5. **(Optional) Install CUDA**: For GPU-based training, install the appropriate CUDA and cuDNN versions corresponding to the PyTorch configuration.
6. **Unity Package**: In Unity, access the Package Manager, switch to "Unity Registry," and install the ML Agents package. Enable preview packages if required.

4 Background Theory of Reinforcement Learning (RL)

Reinforcement learning is a branch of machine learning in which an agent learns to act in an environment by performing actions and receiving feedback in the form of rewards. Over time, the agent improves its policy to maximize the cumulative reward.

In the context of Unity ML Agents, this learning cycle involves:

- **Agent**: A GameObject capable of perceiving its environment, taking actions, and receiving rewards.
- **Environment**: The Unity scene representing the problem space in which the agent operates.
- **Reward Function**: A specification of how rewards or penalties are assigned based on the agent's behavior.
- **Policy**: A neural network that maps observed states to actions.

Through iterative simulation (episodes), the agent identifies state-action pairs that yield higher long-term rewards. Unity ML Agents streamlines this process by handling data flow between Unity and the machine learning backend.

5 Model Training and Deployment (Overview)

Once the agent and environment are configured, training can be initiated using a command-line interface:

```
mlagents-learn config/your_config.yaml --run-id=experiment_name
```

The YAML configuration file specifies hyperparameters such as learning rate, batch size, and reward strategies.

Training may be accelerated by executing multiple instances of the environment in parallel. Upon completion, a `.onnx` model is generated. This file can be assigned within Unity to an agent operating in inference mode, eliminating the need for Python during execution.

6 Visualization with TensorBoard

TensorBoard provides visual feedback on the training process.

- To start TensorBoard, run:

  ```
  tensorboard --logdir results
  ```

- Navigate to `http://localhost:6006` to access graphical representations of:
  - Cumulative reward trends
  - Average episode length
  - Entropy, learning rate, and loss metrics

These visualizations assist in diagnosing training inefficiencies and verifying learning progression.

7 Implementation

To use Unity's ML Agents, a previously made game was enhanced. The game implies a maze with walls of stone, in which a piglet is trying to find all coins. Originally, the maze was randomly generated and the coins were randomly placed within the maze.

In order to make the learning attainable, the maze is not statically generated, much smaller and with only one coin within it. The goal is for the piglet to find the coin (see Figure1).

On the piglet's components, the ML-Agents specific components were added (Figure2). The **Behavior Parameters** are where the **Space Size**, **Continuous Actions**, **Model** and **Behavior Type** are set. The **Move to Coin Agent** is the script which overrides the actions and rewards system of the agent. **Max Step** ensures that a certain running episode does not take too long.

Listing 1: Agent Script for Coin Collection

```
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
```

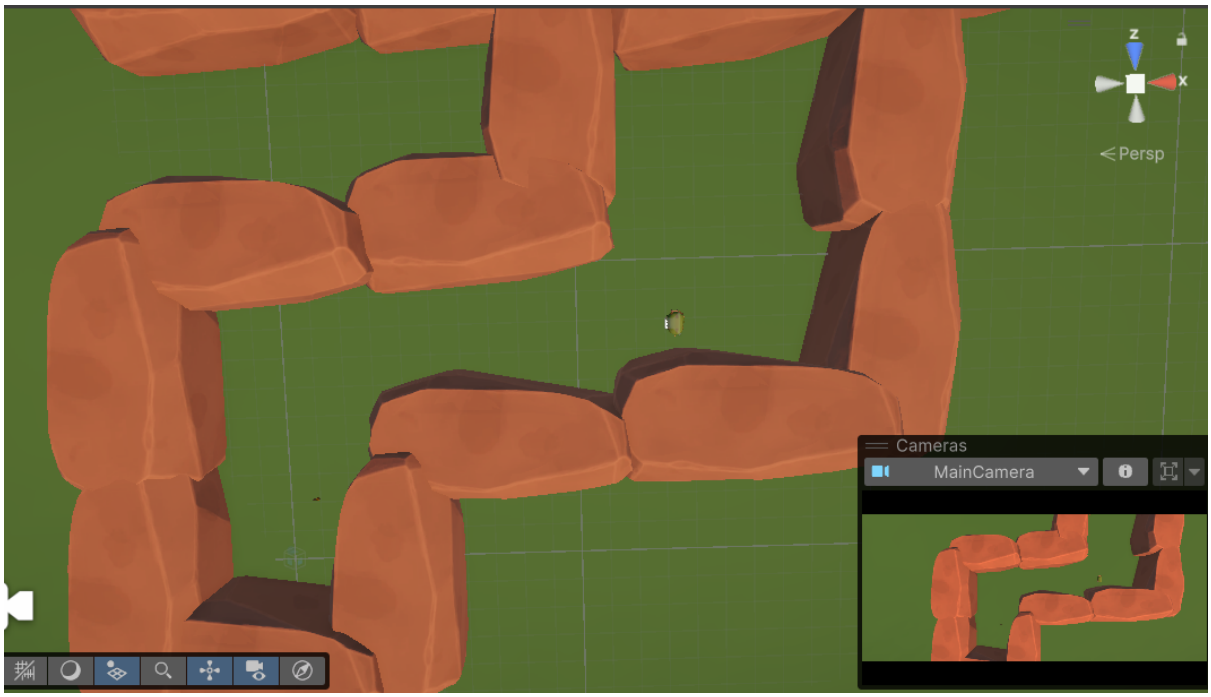

Figure 1: Game From Above

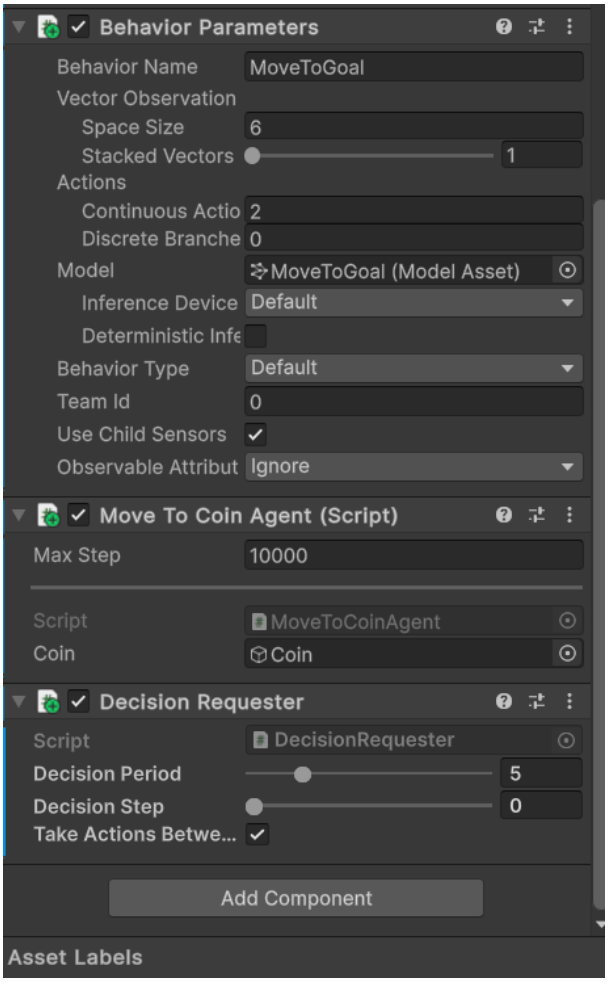

Figure 2: ML Agents components on the piglet

```
using Unity.MLAgents.Sensors;

public class MoveToCoinAgent : Agent
{
    [SerializeField]
    private GameObject coin;

    private Vector3 GetRandomPosition(float y)
    {
        Vector3 newPos = new Vector3(0, y, 0);
        int zone = Random.Range(0, 3);

        switch (zone)
        {
            case 0:
                newPos.x = Random.Range(-1.5f, 1.8f);
                newPos.z = Random.Range(0.5f, 7f);
                break;
            case 1:
                newPos.x = Random.Range(-1.3f, 16.5f);
                newPos.z = Random.Range(8f, 7f);
                break;
            case 2:
                newPos.x = Random.Range(11.8f, 16.6f);
                newPos.z = Random.Range(8f, 14f);
                break;
        }

        return newPos;
    }

    public override void OnEpisodeBegin()
    {
        transform.position = GetRandomPosition(0.25f);
        coin.transform.position = GetRandomPosition(1.5f);
    }

    public override void CollectObservations(VectorSensor sensor)
    {
        sensor.AddObservation(transform.position);
        sensor.AddObservation(coin.transform.position);
    }

    public override void OnActionReceived(ActionBuffers actions)
    {
        float moveX = actions.ContinuousActions[0];
        float moveZ = actions.ContinuousActions[1];
```

```
        float moveSpeed = 1f;
        Vector3 move = new Vector3(moveX, 0, moveZ).normalized *
                moveSpeed * Time.deltaTime;

        transform.position += move;
    }
}
```

In Listing1 is depicted the script for the piglet agent. It can be seen that on the beginning of a new episode, a random position for the piglet and the coin is chosen. Also, when a new continuous action is chosen, the piglet is moved depending on the received values. When finding the coin, a new episode is started.

The reward system was integrated within the piglet's manager. It can be observed in Listing2 that a positive reward is added when the piglet is closer to the coin than before and a negative reward is added when the piglet is further from the coin than before or when it hits a wall.

Listing 2: Reward System

```
private void OnCollisionEnter(Collision collision)
{
    if(collision.gameObject.CompareTag("Coin"))
    {
        coins++;
        moveToCoinAgent.SetReward(10f);
        moveToCoinAgent.EndEpisode();
        audioSource.Play();
    }
    if(collision.gameObject.CompareTag("Wall"))
    {
        moveToCoinAgent.AddReward(-1f);
    }
}

private void UpdateRewards()
{
    var newDistance = Vector3.Distance(
        coin.transform.position,
        moveToCoinAgent.transform.position);

    if (newDistance >= distanceToCoin)
    {
        moveToCoinAgent.AddReward(-1);
    }
    else
    {
        moveToCoinAgent.AddReward(1);
    }
    distanceToCoin = newDistance;
}
```

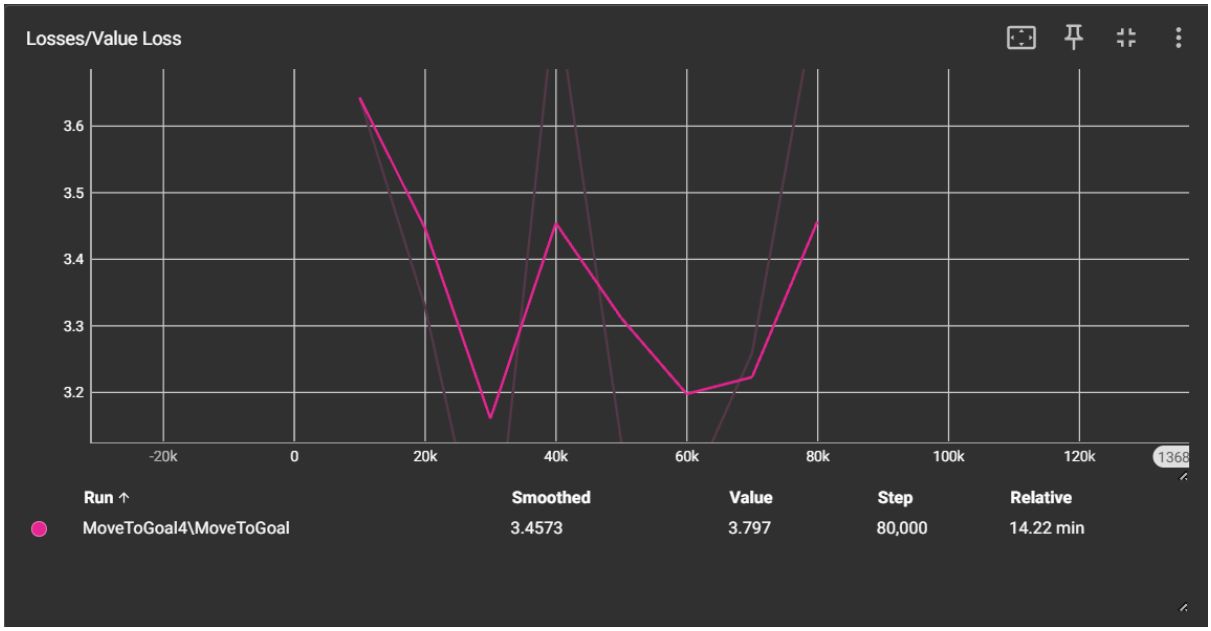

Figure 3: Value Loss

```
private void Update()
{
    inputManager.HandleAllInputs();

    UpdateRewards();
}
```

8 Results

After a few tryouts, the final model was left to train for approximately 30 minutes. In Figure3 is the graphical representation of the value loss of the final model. It is to be noted that the final model was trained starting from the previously trained model and that model was trained starting from the previously trained model as well. So the learning actually started two sessions before this. There are strong reasons to believe that leaving the training process for a longer session, will result in much better results.

9 Conclusion

Experimenting with Unity's ML Agents is no hard science and can prove to be both extremely fun and a real curiosity sparker. All steps of this project were highly enjoyable and it can be safely stated that many things, concepts and approaches have been learned.

10 Discussion

In the further development of the integration of this project with AI, could be some better enforced rewards systems, imitation learning or adding more randomness to the whole environment (maybe returning to randomly generating the maze as well).