Getting Started With MLAgents

By Cameron Olson

Setting up MLAgents on macOS

Follow these steps to set up MLAgents on macOS:

1. Download and install Python by following the instructions at https://www.python.org/downloads/.
2. Download and install Miniconda by following the instructions at https://docs.conda.io/en/latest/miniconda.html.
3. Create a new conda environment for MLAgents by running the following commands in your terminal:

```
conda create --name $unique_name python=3.7
conda activate $unique_name
```

4. Install MLAgents and its dependencies by running the following commands:

```
pip install mlagents==0.28.0
pip install importlib-metadata==4.4
```

5. Verify that everything is set up correctly by running the following command:

```
mlagents-learn --help
```

   This should display the help menu for the MLAgents learning tool.

6. To use MLAgents in your Unity project, you need to navigate to your project directory in the terminal.

```
cd $UnityProjectPath"
```

Install MLAgents Package in Unity

1. Open your Unity project and navigate to the Package Manager. You can access the Package Manager by clicking on the Window menu in the top bar, and then selecting Package Manager.
2. Once you are in the Package Manager, you can search for MLAgents in the search bar. You should see a package called "com.unity.ml-agents" appear in the search results.

3. Click on the "Install" button next to the MLAgents package. Unity will then download and install the package for you. This may take a few minutes depending on the speed of your internet connection and the size of the package.

4. Once the package is installed, you should see it listed in the Package Manager under the "Installed" tab.

5. To start using MLAgents in your Unity project, you will need to create an MLAgents scene and add some agents to it. You can find detailed instructions on how to do this in the MLAgents documentation.

Introduction to Reinforcement Learning

Reinforcement learning is a type of machine learning in which an agent learns to make decisions in an environment to maximize a reward signal. The agent interacts with the environment by taking actions, and receives feedback in the form of rewards or penalties based on the actions taken. The goal of the agent is to learn a policy, which is a mapping from states to actions, that maximizes the expected cumulative reward over time.

The training loop for reinforcement learning

1. Initialize the environment and the agent.
2. The agent observes the current state of the environment.
3. The agent selects an action based on the current state using its policy.
4. The environment transitions to a new state based on the action taken by the agent.
5. The agent receives a reward signal from the environment based on the new state.
6. The agent updates its policy based on the observed state, action, and reward.
7. Repeat steps 2-6 until the agent has learned a good policy.

During the training process, the agent's policy is updated based on the rewards it receives from the environment. The goal is to maximize the expected cumulative reward over time. The agent's policy is typically represented by a neural network, which is trained using techniques such as Deep-Q-Networks

Creating an MLAgent Script in Unity

To create an ML Agent script in Unity, follow these steps:

1. Open your Unity project and create a new C# script. add namespaces bellow. this will allow you to access the MLAgents libraries in your script.

   ```
   using Unity.MLAgents;
   using Unity.MLAgents.Actuators;
   using Unity.MLAgents.Sensors;
   ```

2. Inherit from the `Agent` class to get access to MLAgents functionality. To do this, replace the class definition with the following:

```
public class YourAgentName : Agent
{
    // Add agent behavior here
}
```

3. Save the script and attach it to a GameObject in your Unity scene. This will create an instance of your agent.

In the next section, we'll cover how to configure your agent's behavior using the Behavior Parameters component.

Behavior Parameters for MLAgents in Unity

The Behavior Parameters Component is an important part of MLAgents in Unity. Here are the different options available in the component:

Space Size

This determines the number of inputs from the observation. Keep in mind that if you're taking in `transform.position`, this counts as three inputs because it's a `Vector3`. You should think about the data coming into the system when deciding on the space size.

Stacking Vectors

Stacking vectors allow the network to observe changes over time. For example, if you're monitoring the position of an entity using three stacked vectors, the network will receive the current position and the two preceding positions. This enables the network to infer the entity's movement direction based on its previous positions.

Continuous Actions

Continuous actions are actions that can take on a range of continuous values instead of discrete values. You should specify the number of continuous actions you want in this box. Keep in mind that these are float inputs in your script.

Discrete Branches & Branch Size

Discrete branches are the different sets of actions that an agent can take. Each branch corresponds to a different category of actions that the agent can perform. For example, if you have an agent that needs to move and shoot, you would have two branches, one for moving and one for shooting. You should set the size to the number of actions in each branch.

Model Input

This is where you should provide the trained network file in Onix format that you want to use for the agent. This file should be the output of your training process, and it contains the trained weights and biases of the neural network. You can think of it as the "brain" of your agent that will be used to make decisions. You should specify the location of the file in the Model Input field.

Inference Device

This allows you to set the device you want to run inference on. The options are DEFAULT, GPU, BURST, or CPU. Keep in mind that "inference" is the word used when referring to using a model.

Default, Heuristic, and Inference Modes

The Behavior Parameters Component provides three modes that you can use to control the agent's behavior:

- Default Mode: This is the recommended mode, as it automatically picks the best option for you based on your configuration. You don't need to make any additional changes to your script.
- Heuristic Mode: This mode allows you to implement custom decision-making logic or to support manual control of the agent using keyboard, mouse, or game controller input. You should add the heuristic function to your script to provide this custom logic.
- Inference Mode: This mode runs the trained model to make decisions. You should specify the trained model in the Model Input field, and the agent will use the model to make decisions based on the observations it receives.

Observation Collection with MLAgents

In MLAgents, observations are the inputs given to an agent that enable it to perceive its environment. To collect observations in your Unity project, you need to override the `CollectObservations` method in your agent's script.

To override the `CollectObservations` method, add the following code to your agent's script:

```
public override void CollectObservations(VectorSensor sensor)
{
    // Add observations here
}
```

The `CollectObservations` method takes in a `VectorSensor` object, which is used to collect the observations for the agent. To add an observation to the `VectorSensor`, you can use the `AddObservation` method. For example:

```
sensor.AddObservation(transform.position);
```

This code adds the current position of the agent's transform as an observation, so that the ML Agent can "see" where it is in the environment.

Using `AddObservation`, you can add any information that you want the agent to observe, such as its velocity, the color of nearby objects, or the distance to the goal. By adding more observations, you give your agent more information to make decisions with.

Once you've added all the necessary observations, the `CollectObservations` method builds the agent's observation vector, which is used as input to the neural network that controls the agent's behavior.

MLAgent Decision-Making

To enable an agent to make decisions, you need to add the `DecisionRequester` component to it. The `DecisionRequester` component is a crucial component in the ML Agents toolkit, as it enables agents to make decisions based on their current observations.

1. Navigate to the GameObject that represents the agent in the Unity Editor.
2. Click on the "Add Component" button.
3. Select `ML Agents` and then `DecisionRequester` from the drop-down menu.

The `DecisionRequester` component works by automatically requesting decisions for an agent at regular intervals. This ensures that the agent always has the most up-to-date information to work with when making decisions.

MLAgent Action Handling

To specify the actions an agent can perform, we override the `OnActionReceived` method provided by MLAgents. This method receives an `ActionBuffers` object that contains the continuous or discrete actions the agent should perform.

Continuous Actions

Continuous actions are actions that can be performed with a continuous range of values, such as movement speed, rotation angle, or acceleration. To get the continuous actions, we use the `ContinuousActions` property of the `ActionBuffers` object.

For example:

```
float moveX = actions.ContinuousActions[0];
float moveY = actions.ContinuousActions[1];
```

Discrete Actions

Discrete actions are actions that can be performed with a finite set of possible values, such as jumping, firing a weapon, or changing direction. To get the discrete actions, we use the `DiscreteActions` property of the `ActionBuffers` object.

For example:

```
int actionIndex = actions.DiscreteActions[0];
```

Movement Example

Here's an example of how we can use the `OnActionReceived` method to update the agent's position based on the continuous actions it receives:

```
public override void OnActionReceived(ActionBuffers actions)
{
    float moveX = actions.ContinuousActions[0];
    float moveZ = actions.ContinuousActions[1];
    float moveSpeed = 5f;
    transform.position += new Vector3(moveX, 0, moveZ) * Time.deltaTime *
moveSpeed;
}
```

Rewarding Agent Behavior

In Reinforcement Learning, it's crucial to reward your agent for good behavior and discourage it from bad behavior. There are two main ways to give an agent a reward: setting a reward for a specific amount or incrementing the current reward.

You can set a reward for a specific amount using the `SetReward()` method:

```
SetReward(rewardValue);
```

Alternatively, you can increment the current reward using the `AddReward()` method:

```
AddReward(rewardValue);
```

Reinforcement Learning Training Loop Setup

In Unity, the training loop for your agent is referred to as an episode. An example of an episode is when the agent hits the ball, receives a reward, and then is reset to the starting position.

To set up the training loop, you'll need to override the `OnEpisodeBegin()` method. This method is called at the beginning of each episode and allows you to define what happens when an episode starts:

```
public override void OnEpisodeBegin()
{
    // What you want to happen after EndEpisode() is called.
}
```

To reset the state for training, you can call the `EndEpisode()` method. This method triggers the `OnEpisodeBegin()` method and starts a new episode:

```
EndEpisode();
```

MLAgents Model Training

Training a model in MLAgents is a straightforward process that can be done through the command line interface (CLI). Follow the steps below to start training:

1. Open your terminal and navigate to your project directory.
2. Use the `mlagents-learn` command to begin training. Below are some useful options for this command:

- `--train`: Indicates that you want to train the model.
- `--run-id`: Sets a unique identifier for the current run, which is useful for keeping track of multiple runs.
- `--num-runs`: Specifies the number of training runs to perform.
- `--resume`: Resumes training from a previous run.
- `--force`: Overrides previous training data and starts training from scratch.
- `--load`: Training code loads trained model as initialization for the weights.
- `--initialize-from=`: Pass in past run id to use as initialization

```
    mlagents-learn --run-id=<unique_id>
```

To start the training process, press Enter in the command line interface (CLI) and then play your Unity environment. As training progresses, you'll see updates printed in the terminal. Once the training is complete, an ONNX file containing the trained model will be generated.

Sure, I'd be happy to help! Here's a revised version of the section that includes more details and explanations:

Hyperparameter Configuration MLAgents

Hyperparameters are variables that determine the behavior and performance of an ML model during training. With MLAgents in Unity, you can directly control hyperparameters for your training through a config file in YAML format. This allows you to fine-tune your model for optimal performance.

Creating a YAML Config File

To create a YAML config file, you'll need to specify the hyperparameters for your training run. Here's an example of a simple YAML config file:

```
behaviors:
  NameOfMLAgentScript:
    trainer_type: ppo
    hyperparameters:
      batch_size: 10
      buffer_size: 100
      learning_rate: 0.0003
      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
    max_steps: 400000
    time_horizon: 64
    summary_freq: 10000
```

Running a Config Training Run

To run a training session using your YAML config file, you'll use the `mlagents-learn` command followed by the path to your config file and a unique ID for the training run. Here's an example command:

```
mlagents-learn $path_to_config.yaml --run-id=<unique_id>
```

Once you run this command, MLAgents will start the training process with the hyperparameters specified in your YAML file.

It's important to note that hyperparameter tuning can be an iterative process. You may need to experiment with different values to find the optimal configuration for your specific use case.

Visualizing and Interpreting Training Metrics with TensorBoard

Visualizing your training progress is an important step in developing your ML model. With MLAgents, you can use TensorBoard to visualize the training metrics in real-time.

Launching TensorBoard

To launch TensorBoard, navigate to your project directory in a terminal and run the following command:

```
tensorboard --logdir results
```

This will launch TensorBoard and start reading the log files generated by your training run. By default, TensorBoard runs on port 6006.

Viewing the TensorBoard Interface

To view the TensorBoard interface, open a web browser and go to `http://localhost:6006`. This will display the main dashboard, which provides an overview of the training metrics.

Interpreting the Training Metrics

The training metrics displayed in TensorBoard can provide valuable insights into the performance of your ML model. Here are some of the key metrics you should pay attention to:

- **Reward**: This metric measures the reward signal received by the agent during training. Ideally, you want to see this metric increasing over time as the agent learns to perform the task.
- **Entropy**: This metric measures the degree of exploration/exploitation in the agent's actions. Higher entropy means the agent is exploring more, while lower entropy means it's exploiting more. You typically want to see a balance between the two.
- **Policy Loss**: This metric measures how much the policy has changed during training. A high policy loss could indicate that the agent is not learning effectively.
- **Value Loss**: This metric measures the difference between the predicted value and the actual value of the agent's actions. A high value loss could indicate that the agent is not accurately predicting the value of its actions.
- **Learning Rate**: This metric measures the learning rate used by the optimizer during training. You typically want to see this value decreasing over time as the agent gets closer to the optimal policy.

By monitoring these metrics during training, you can gain insights into how well your ML model is performing and adjust your hyperparameters as needed.

Deploying Your Trained Model in Unity

After training your model, you can use it in your Unity Project by locating the ONNX file that was generated, This file is typically located in the `results` directory under the `name-id` of your run. your model will have `.onnx` as its file extension.

Next, add the ONNX file to your project's assets and then attach it to your agent by placing it in the model input field in the `Behavior Parameters` component of your agent. You can now run inference on your model.

References

MLAgents Unity Documentation: https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.html

Github MLAgents: https://github.com/Unity-Technologies/ml-agents

GitHub MLAgents docs: https://github.com/Unity-Technologies/ml-agents/tree/develop/docs

Introduction too Reinforcement Learning: https://youtu.be/TCCjZe0y4Qc

Pytorch Reinforcement Learning Deep Q Net: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

RL Cheatsheet: https://colab.research.google.com/drive/1eN33dPVtdPViiS1njTW_-r-IYCDTFU7N

Introduction to Reinforcement Learning Part 1: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html

Introduction to Reinforcement Learning Part 2: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

Introduction to Reinforcement Learning Part 3:
(https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html