Unity ML Agents Documentation

Installation & Set-up

Install Unity 2017.1 or later (required)

Download link available here.

Clone the repository

Once installed, you will want to clone the Agents GitHub repository.

References will be made throughout to unity-environment and python directories. Both are located at the root of the repository.

Installing Python API

In order to train an agent within the framework, you will need to install Python 2 or 3, and the dependencies described below.

Windows Users

If you are a Windows user who is new to Python/TensorFlow, follow this guide to set up your Python environment.

Requirements

- Jupyter
- Matplotlib
- numpy
- Pillow
- Python (2 or 3; 64bit required)
- docopt (Training)
- TensorFlow (1.0+) (Training)

Installing Dependencies

To install dependencies, go into the python sub-directory of the repository, and run (depending on your python version) from the command line:
```
pip install .
```
or

```
pip3 install .
```
If your Python environment doesn't include pip, see these instructions on installing it.
Once the requirements are successfully installed, the next step is to check out the Getting Started guide.

Installation Help

Using Jupyter Notebook

For a walkthrough of how to use Jupyter notebook, see here.

General Issues

If you run into issues while attempting to install and run Unity ML Agents, see here for a list of common issues and solutions.

If you have an issue that isn't covered here, feel free to contact us at ml-agents@unity3d.com.

Alternatively, feel free to create an issue on the repository. Be sure to include relevant information on OS, Python version, and exact error message if possible.

Making a new Learning Environment

This tutorial walks through the process of creating a Unity Environment. A Unity Environment is an application built using the Unity Engine which can be used to train Reinforcement Learning agents.

Setting up the Unity Project

1. Open an existing Unity project, or create a new one and import the RL interface package:

- [ML-Agents package without TensorflowSharp]
- [ML-Agents package with TensorflowSharp]

2. Rename TemplateAcademy.cs (and the contained class name) to the desired name of your new academy class. All Template files are in the folder Assets -> Template ->

Scripts. Typical naming convention is YourNameAcademy.

3. Attach YourNameAcademy.cs to a new empty game object in the currently opened scene (Unity -> GameObject -> Create Empty) and rename this game object to YourNameAcademy.
   Since YourNameAcademy will be used to control all the environment logic, ensure the attached-to object is one which will remain in the scene regardless of the environment resetting, or other within-environment behavior.
4. Attach Brain.cs to a new empty game object and rename this game object to YourNameBrain1. Set this game object as a child of YourNameAcademy (Drag YourNameBrain1 into YourNameAcademy). Note that you can have multiple brains in the Academy but they all must have different names.

5. Disable Window Resolution dialogue box and Splash Screen.

   i. Go to Edit -> Project Settings -> Player -> Resolution and Presentation.

ii. Set Display Resolution Dialogue to Disabled.
    3.Check Run In Background.
iii. Click Splash Image.
iv. Uncheck Show Splash Screen *(Unity Pro only)*.

6. If you will be using Tensorflow Sharp in Unity, you must:

   i. Make sure you are using Unity 2017.1 or newer.
   ii. Make sure the TensorflowSharp plugin is in your Asset folder.
   iii. Go to Edit -> Project Settings -> Player
   iv. For each of the platforms you target (PC, Mac and Linux Standalone, iOS or Android):
      a. Go into Other Settings.
      b. Select Scripting Runtime Version to Experimental (.NET 4.6 Equivalent)
      c. In Scripting Defined Symbols, add the flag ENABLE_TENSORFLOW
   v. Note that some of these changes will require a Unity Restart

Implementing YourNameAcademy

1. Click on the game object YourNameAcademy.
2. In the inspector tab, you can modify the characteristics of the academy:

- Max Steps Maximum length of each episode (set to 0 if you do not want the environment to reset after a certain time).
- Wait Time Real-time between steps when running environment in test-mode.
- Frames To Skip Number of frames (or physics updates) to skip between steps. The agents will act at every frame but get new actions only at every step.
- Training Configuration and Inference Configuration The first defines the configuration of the Engine at training time and the second at test / inference time. The training mode corresponds only to external training when the reset parameter train_model was set to True. The adjustable parameters are as follows:
    - Width and Height Correspond to the width and height in pixels of the window (must be both

    greater than 0). Typically set it to a small size during training, and a larger size for visualization during inference.
    - Quality Level Determines how mesh rendering is performed. Typically set to small value during training and higher value for visualization during inference.
    - Time Scale Physics speed. If environment utilized physics calculations, increase this during training, and set to 1.0f during inference. Otherwise, set it to 1.0f.
    - Target Frame Rate Frequency of frame rendering. If environment utilizes observations, increase this during training, and set to 60 during inference. If no observations are used, this can be set to 1 during training.
- Default Reset Parameters You can set the default configuration to be passed at reset. This will be a mapping from strings to float values that you can call in the academy with resetParameters["YourDefaultParameter"]

3. Within InitializeAcademy(), you can define the initialization of the Academy. Note that this command is ran only once at the beginning of the training session.
   Do not use Awake(), Start() or OnEn able()
4. Within AcademyStep(), you can define the environment logic each step. Use this function to modify the environment for the agents that will live in it.
5. Within AcademyReset(), you can reset the environment for a new episode. It should contain environment-specific code for setting up the environment. Note that AcademyReset() is called at the beginning of the training session to ensure the first episode is similar to the others.

Implementing YourNameBrain

For each Brain game object in your academy :

1. Click on the game object YourNameBrain
2. In the inspector tab, you can modify the characteristics of the brain in Brain Parameters

- State Size Number of variables within the state provided to the agent(s).
- Action Size The number of possible actions for each individual agent to take.
- Memory Size The number of floats the agents will remember each step.
- Camera Resolutions A list of flexible length that contains resolution parameters
  : height and width define the number dimensions of the camera outputs in pixels. Check Black And White if you want the camera outputs to be black and white.
- Action Descriptions A list describing in human-readable language the meaning of each available action.
- State Space Type and Action Space Type.
  Either discrete or continuous.
  - discrete corresponds to describing the action space with an int.
  - continuous corresponds to describing the action space with an array of float.

3. You can choose what kind of brain you want YourNameBrain to be. There are four possibilities:

- External : You need at least one of your brains to be external if you wish to interact with your environment from python.
- Player : To control your agents manually. If the action space is discrete, you must map input keys to their corresponding integer values. If the action space is continuous, you must map input keys to their corresponding indices and float values.
- Heuristic : You can have your brain automatically react to the observations and states in a customizable way. You will need to drag a Decision script into YourNameBrain. To create a custom reaction, you must :
    - Rename TemplateDecision.cs (and the contained class name) to the desired name of your new reaction. Typical naming convention is YourNameDecision.
    - Implement Decide: Given the state, observation and memory of an agent, this function must return an array of floats

    corresponding to the actions taken by the agent. If the action space type is discrete, the array must be of size 1.
  - Optionally, implement MakeMemory: Given the state, observation and memory of an agent, this function must return an array of floats corresponding to the new memories of the agent.
- Internal : Note that you must have Tensorflow Sharp setup (see top of this page). Here are the fields that must be completed:
  - Graph Model : This must be the bytes file corresponding to the pretrained Tensorflow graph. (You must first drag this file into your Resources folder and then from the Resources folder into the inspector)
  - Graph Scope : If you set a scope while training your tensorflow model, all your placeholder name will have a prefix. You must specify that prefix here.
  - Batch Size Node Name : If the batch size is one of the inputs of your graph, you must specify the name if the placeholder here. The brain will make the

batch size equal to the number of agents connected to the brain automatically.

- State Node Name : If your graph uses the state as an input, you must specify the name if the placeholder here.
- Recurrent Input Node Name : If your graph uses a recurrent input / memory as input and outputs new recurrent input / memory, you must specify the name if the input placeholder here.
- Recurrent Output Node Name : If your graph uses a recurrent input / memory as input and outputs new recurrent input / memory, you must specify the name if the output placeholder here.
- Observation Placeholder Name : If your graph uses observations as input, you must specify it here. Note that the number of observations is equal to the length of Camera Resolutions in the brain parameters.
- Action Node Name : Specify the name of the placeholder corresponding to the actions of the brain in your graph. If the

action space type is continuous, the output must be a one dimensional tensor of float of length Action Space Size, if the action space type is discrete, the output must be a one dimensional tensor of int of length 1.

- Graph Placeholder : If your graph takes additional inputs that are fixed (example: noise level) you can specify them here. Note that in your graph, these must correspond to one dimensional tensors of int or float of size 1.
  - Name : Corresponds to the name of the placeholder.
  - Value Type : Either Integer or Floating Point.
  - Min Value and 'Max Value' : Specify the minimum and maximum values (included) the placeholder can take. The value will be sampled from the uniform distribution at each step. If you want this value to be fixed, set both Min Value and Max Value to the same number.

Implementing YourNameAgent

1. Rename TemplateAgent.cs (and the contained class name) to the desired name of your new agent. Typical naming convention is YourNameAgent.
2. Attach YourNameAgent.cs to the game object that represents your agent. (Example: if you want to make a self-driving car, attach YourNameAgent.cs to a car looking game object)
3. In the inspector menu of your agent, drag the brain game object you want to use with this agent into the corresponding Brain box. Please note that you can have multiple agents with the same brain. If you want to give an agent a brain or change his brain via script, please use the method ChangeBrain().
4. In the inspector menu of your agent, you can specify what cameras, your agent will use as its observations. To do so, drag the desired number of cameras into the Observations field. Note that if you want a camera to move along your agent, you can set this camera as a child of your agent
5. If Reset On Done is checked, Reset() will be called

when the agent is done.
Else, AgentOnDone() will be called.
Note that if Reset On Done is unchecked, the agent will remain "done" until the Academy resets. This means that it will not take actions in the environment.

6. Implement the following functions in YourNameAgent.cs :

- InitializeAgent() : Use this method to initialize your agent. This method is called when the agent is created.
  Do notuse Awake(), Start() or OnEnable().
- CollectState() : Must return a list of floats corresponding to the state the agent is in. If the state space type is discrete, return a list of length 1 containing the float equivalent of your state.
- AgentStep() : This function will be called every frame, you must define what your agent will do given the input actions. You must also specify the rewards and whether or not the agent is done. To do so, modify the public fields of the agent reward and done.
- AgentReset() : This function is called at start, when the Academy

  resets and when the agent is done (if Reset On Done is checked).
- AgentOnDone() : If Reset On Done is not checked, this function will be called when the agent is done. Reset() will only be called when the Academy resets.

If you create Agents via script, we recommend you save them as prefabs and instantiate them either during steps or resets. If you do, you can use GiveBrain(brain) to have the agent subscribe to a specific brain. You can also use RemoveBrain() to unsubscribe from a brain.

Defining the reward function

The reward function is the set of circumstances and event which we want to reward or punish the agent for making happen. Here are some examples of positive and negative rewards:

- Positive
  - Reaching a goal
  - Staying alive
  - Defeating an enemy
  - Gaining health
  - Finishing a level
- Negative

    - Taking damage
    - Failing a level
    - The agent’s death

Small negative rewards are also typically used each step in scenarios where the optimal agent behavior is to complete an episode as quickly as possible.

Note that the reward is reset to 0 at every step, you must add to the reward (reward += rewardIncrement). If you use skipFrame in the Academy and set your rewards instead of incrementing them, you might lose information since the reward is sent at every step, not at every frame.

Organizing the Scene Layout

This tutorial will help you understand how to organize your scene when using Agents in your Unity environment.

ML-Agents Game Objects

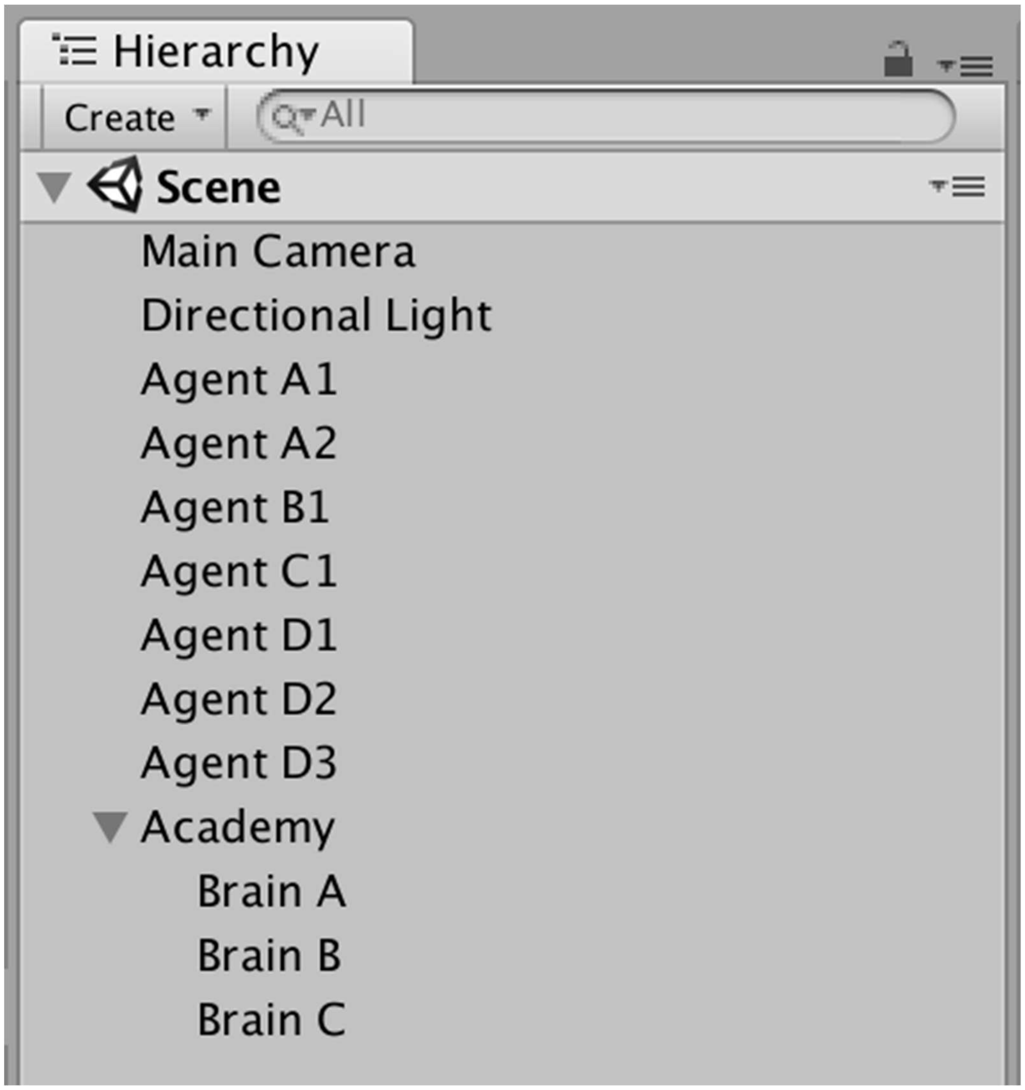
There are three kinds of game objects you need to include in your scene in order to use Unity ML-Agents:

- Academy
- Brain
- Agents

Keep in mind :

- There can only be one Academy game object in a scene.
- You can have multiple Brain game objects but they must be child of the Academy game object.

Here is an example of what your scene hierarchy should look like :



```
Main Camera
Directional Light
Agent A1
Agent A2
Agent B1
Agent C1
Agent D1
Agent D2
Agent D3
Academy
    Brain A
    Brain B
    Brain C
```

Functionality

The Academy

The Academy is responsible for:

- Synchronizing the environment and keeping all agent's steps in pace.

As such, there can only be one per scene.

- Determining the speed of the engine, its quality, and the display's resolution.
- Modifying the environment at every step and every reset according to the logic defined in AcademyStep() and AcademyReset().
- Coordinating the Brains which must be set as children of the Academy.

Brains

Each brain corresponds to a specific Decision-making method. This often aligns with a specific neural network model. The brain is responsible for deciding the action of all the Agents which are linked to it. There can be multiple brains in the same scene and multiple agents can subscribe to the same brain.

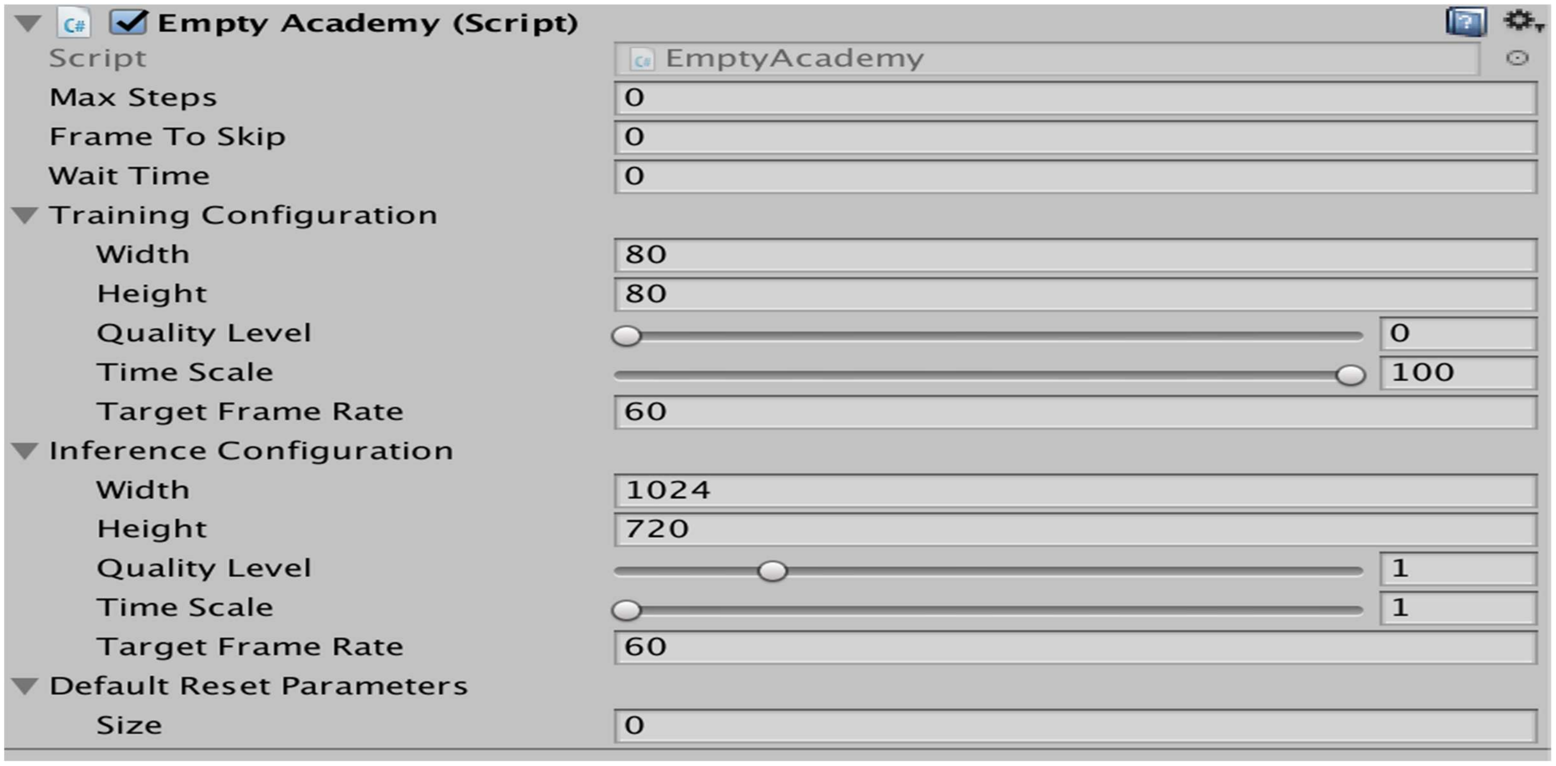
Agents

Each agent within a scene takes actions according to the decisions provided by it's linked Brain. There can be as many Agents of as many types as you like in the scene. The state size and action size of each agent must match the brain's parameters in order for the Brain to decide actions for it.

ML Agents Editor Interface

This page contains an explanation of the use of each of the inspector panels relating to the Academy, Brain, and Agentobjects.

Academy



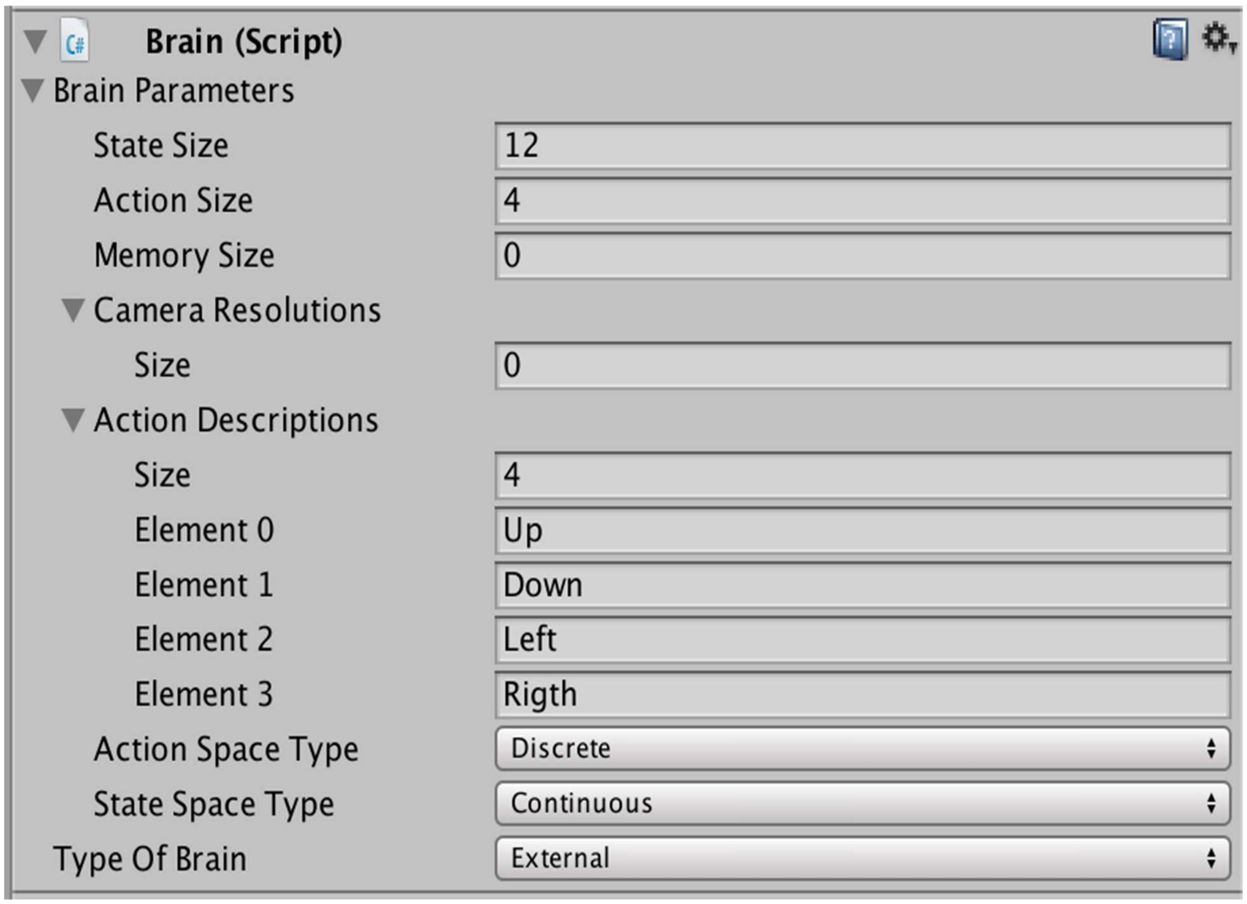
Empty Academy (Script)

Script	EmptyAcademy
Max Steps	0
Frame To Skip	0
Wait Time	0
Training Configuration	
Width	80
Height	80
Quality Level	0
Time Scale	100
Target Frame Rate	60
Inference Configuration	
Width	1024
Height	720
Quality Level	1
Time Scale	1
Target Frame Rate	60
Default Reset Parameters	
Size	0

- Max Steps - Total number of steps per-episode. 0 corresponds to episodes without a maximum number of steps. Once the step counter reaches maximum, the environment will reset.

- Frames To Skip - How many steps of the environment to skip before asking Brains for decisions.
- Wait Time - How many seconds to wait between steps when running in Inference.
- Configuration - The engine-level settings which correspond to rendering quality and engine speed.
  - Width - Width of the environment window in pixels.
  - Height - Width of the environment window in pixels.
  - Quality Level - Rendering quality of environment. (Higher is better)
  - Time Scale - Speed at which environment is run. (Higher is faster)
  - Target Frame Rate - FPS engine attempts to maintain.
- Default Reset Parameters - List of custom parameters that can be changed in the environment on reset.

Brain



Brain (Script)

Brain Parameters

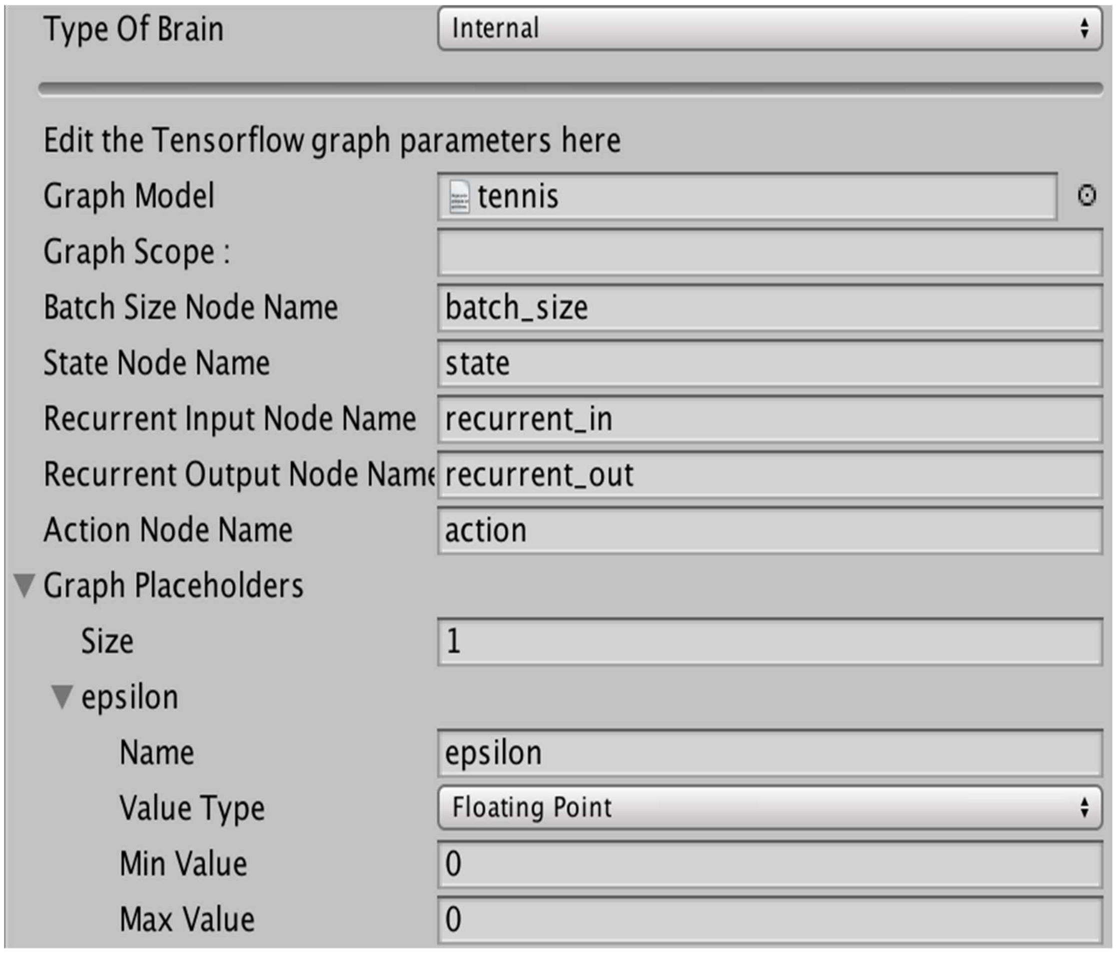
State Size	12
Action Size	4
Memory Size	0
Camera Resolutions	
Size	0
Action Descriptions	
Size	4
Element 0	Up
Element 1	Down
Element 2	Left
Element 3	Rigth
Action Space Type	Discrete
State Space Type	Continuous
Type Of Brain	External

- Brain Parameters - Define state, observation, and action spaces for the Brain.
    - State Size - Length of state vector for brain (In *Continuous* state space). Or number of possible values (in *Discrete*state space).
    - Action Size - Length of action vector for brain (In *Continuous* state space). Or number of possible values (in *Discrete* action space).

    - Memory Size - Length of memory vector for brain. Used with Recurrent networks and frame-stacking CNNs.
    - Camera Resolution - Describes height, width, and whether to greyscale visual observations for the Brain.
    - Action Descriptions - A list of strings used to name the available actions for the Brain.
- State Space Type - Corresponds to whether state vector contains a single integer (Discrete) or a series of real-valued floats (Continuous).
- Action Space Type - Corresponds to whether action vector contains a single integer (Discrete) or a series of real-valued floats (Continuous).
- Type of Brain - Describes how the Brain will decide actions.
    - External - Actions are decided using Python API.
    - Internal - Actions are decided using internal TensorflowSharp model.
    - Player - Actions are decided using Player input mappings.
    - Heuristic - Actions are decided using custom Decision script,

which should be attached to the Brain game object.

Internal Brain



Type Of Brain: Internal

Edit the Tensorflow graph parameters here

Graph Model	tennis
Graph Scope :	
Batch Size Node Name	batch_size
State Node Name	state
Recurrent Input Node Name	recurrent_in
Recurrent Output Node Name	recurrent_out
Action Node Name	action

Graph Placeholders

Size	1

epsilon

Name	epsilon
Value Type	Floating Point
Min Value	0
Max Value	0

- Graph Model : This must be the bytes file corresponding to the pretrained Tensorflow graph. (You must first drag this file into your Resources folder and then from the Resources folder into the inspector)
- Graph Scope : If you set a scope while training your tensorflow

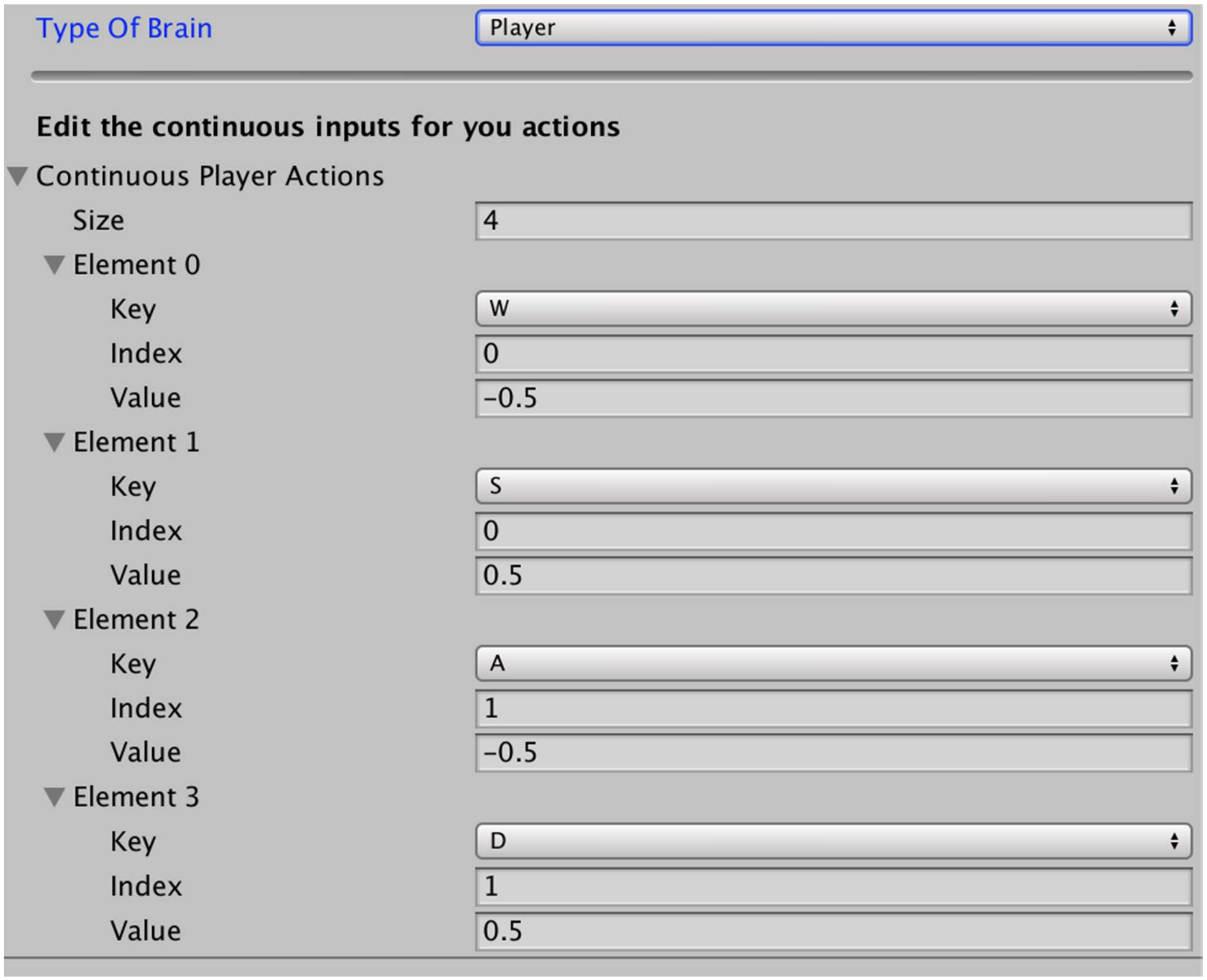
model, all your placeholder name will have a prefix. You must specify that prefix here.

- Batch Size Node Name : If the batch size is one of the inputs of your graph, you must specify the name if the placeholder here. The brain will make the batch size equal to the number of agents connected to the brain automatically.
- State Node Name : If your graph uses the state as an input, you must specify the name if the placeholder here.
- Recurrent Input Node Name : If your graph uses a recurrent input / memory as input and outputs new recurrent input / memory, you must specify the name if the input placeholder here.
- Recurrent Output Node Name : If your graph uses a recurrent input / memory as input and outputs new recurrent input / memory, you must specify the name if the output placeholder here.
- Observation Placeholder Name : If your graph uses observations as input, you must specify it here. Note that the number of observations is equal to the length

of Camera Resolutions in the brain parameters.

- Action Node Name : Specify the name of the placeholder corresponding to the actions of the brain in your graph. If the action space type is continuous, the output must be a one dimensional tensor of float of length Action Space Size, if the action space type is discrete, the output must be a one dimensional tensor of int of length 1.
- Graph Placeholder : If your graph takes additional inputs that are fixed (example: noise level) you can specify them here. Note that in your graph, these must correspond to one dimensional tensors of int or float of size 1.
  - Name : Corresponds to the name of the placeholdder.
  - Value Type : Either Integer or Floating Point.
  - Min Value and Max Value : Specify the range of the value here. The value will be sampled from the uniform distribution ranging from Min Value to Max Value inclusive.

Player Brain



Type Of Brain: Player

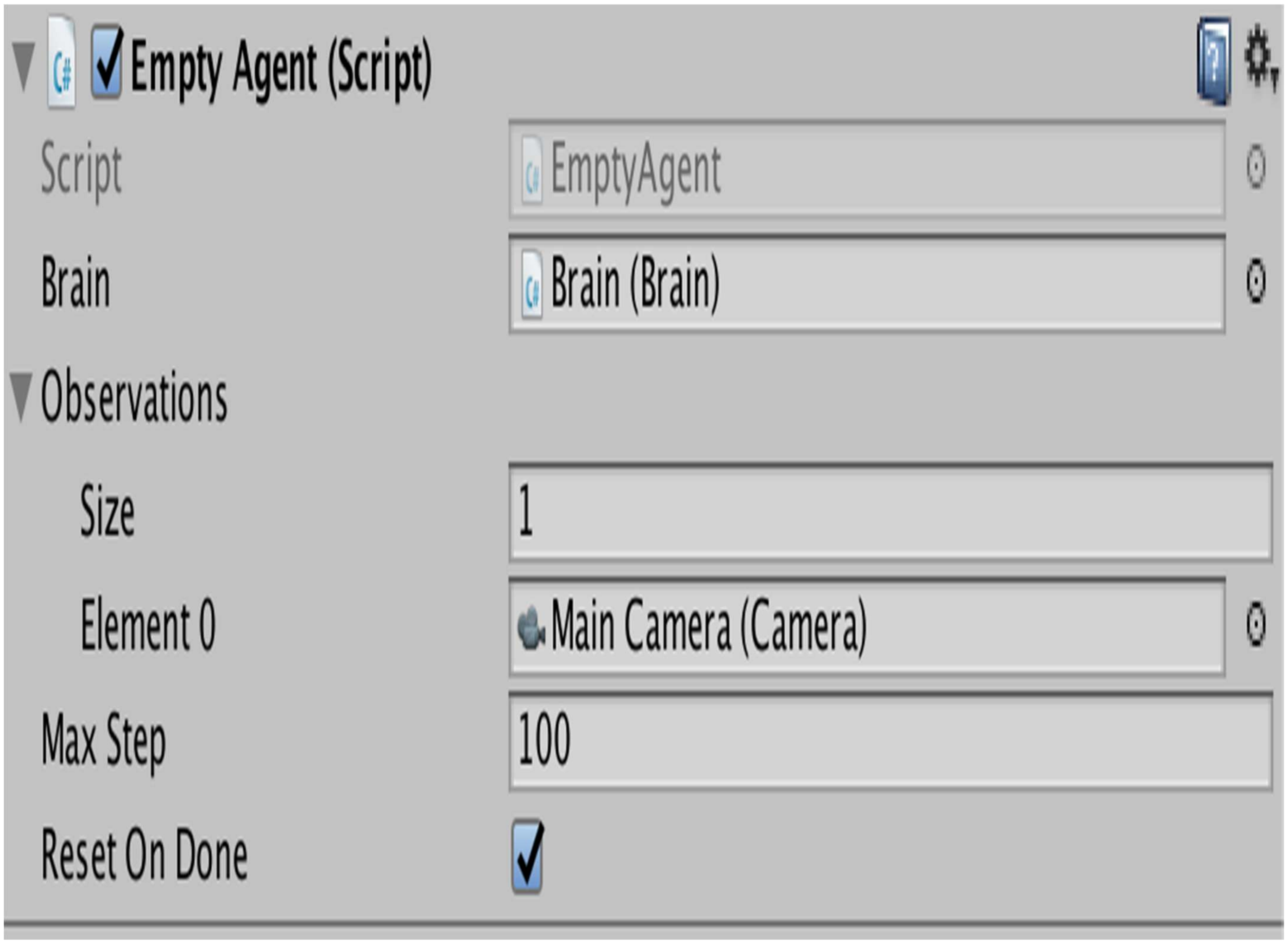
Edit the continuous inputs for you actions

Continuous Player Actions

Size	4
Element 0	
Key	W
Index	0
Value	-0.5
Element 1	
Key	S
Index	0
Value	0.5
Element 2	
Key	A
Index	1
Value	-0.5
Element 3	
Key	D
Index	1
Value	0.5

If the action space is discrete, you must map input keys to their corresponding integer values. If the action space is continuous, you must map input keys to their corresponding indices and float values.

Agent



Empty Agent (Script)

Script	EmptyAgent
Brain	Brain (Brain)
Observations	
Size	1
Element 0	Main Camera (Camera)
Max Step	100
Reset On Done	✓

- Brain - The brain to register this agent to. Can be dragged into the inspector using the Editor.

- Observations - A list of Cameras which will be used to generate observations.
- Max Step - The per-agent maximum number of steps. Once this number is reached, the agent will be reset if Reset On Done is checked.

Best Practices when training with PPO

The process of training a Reinforcement Learning model can often involve the need to tune the hyperparameters in order to achieve a level of performance that is desirable. This guide contains some best practices for tuning the training process when the default parameters don't seem to be giving the level of performance you would like.

Hyperparameters

Batch Size

batch_size corresponds to how many experiences are used for each gradient descent update. This should always be a fraction of the buffer_size. If you are using a continuous action space, this value should be large (in 1000s). If you are using a discrete action space, this value should be smaller (in 10s).
Typical Range (Continuous): 512 - 5120
Typical Range (Discrete): 32 - 512

Beta (Used only in Discrete Control)

beta corresponds to the strength of the entropy regularization, which makes the policy "more random." This ensures that discrete action space agents properly explore during training. Increasing this will ensure more random actions are taken. This should be adjusted such that the entropy (measurable from TensorBoard) slowly decreases alongside increases in reward. If entropy drops too quickly, increase beta. If entropy drops too slowly, decrease beta.
Typical Range: 1e-4 - 1e-2

Buffer Size

buffer_size corresponds to how many experiences should be collected before gradient descent is performed on them all. This should be a multiple of batch_size. Typically larger buffer sizes correspond to more stable training updates.
Typical Range: 2048 - 409600

Epsilon

epsilon corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating. Setting this value small will result in more stable updates, but will also slow the training process.
Typical Range: 0.1 - 0.3

Hidden Units

hidden_units correspond to how many units are in each fully connected layer of the neural network. For simple problems where the correct action is a straightforward combination of the state inputs, this should be small. For problems where the action is a very complex

interaction between the state variables, this should be larger.
Typical Range: 32 - 512

Learning Rate

learning_rate corresponds to the strength of each gradient descent update step. This should typically be decreased if training is unstable, and the reward does not consistently increase.
Typical Range: 1e-5 - 1e-3

Number of Epochs

num_epoch is the number of passes through the experience buffer during gradient descent. The larger the batch size, the larger it is acceptable to make this. Decreasing this will ensure more stable updates, at the cost of slower learning.
Typical Range: 3 - 10

Time Horizon

time_horizon corresponds to how many steps of experience to collect per-agent before adding it to the experience buffer. In cases where there are frequent rewards within an episode, or episodes are prohibitively

large, this can be a smaller number. For most stable training however, this number should be large enough to capture all the important behavior within a sequence of an agent's actions.
Typical Range: 64 - 2048

Max Steps

max_steps corresponds to how many steps of the simulation (multiplied by frame-skip) are run durring the training process. This value should be increased for more complex problems.
Typical Range: 5e5 - 1e7

Normalize

normalize corresponds to whether normalization is applied to the state inputs. This normalization is based on the running average and variance of the states. Normalization can be helpful in cases with complex continuous control problems, but may be harmful with simpler discrete control problems.

Number of Layers

num_layers corresponds to how many hidden layers are present after the

state input, or after the CNN encoding of the observation. For simple problems, fewer layers are likely to train faster and more efficiently. More layers may be necessary for more complex control problems.
Typical range: 1 - 3

Training Statistics

To view training statistics, use Tensorboard. For information on launching and using Tensorboard, see here.

Cumulative Reward

The general trend in reward should consistently increase over time. Small ups and downs are to be expected. Depending on the complexity of the task, a significant increase in reward may not present itself until millions of steps into the training process.

Entropy

This corresponds to how random the decisions of a brain are. This should consistently decrease during training. If it decreases too soon or not at all, beta should be adjusted (when using discrete action space).

Learning Rate

This will decrease over time on a linear schedule.

Policy Loss

These values will oscillate with training.

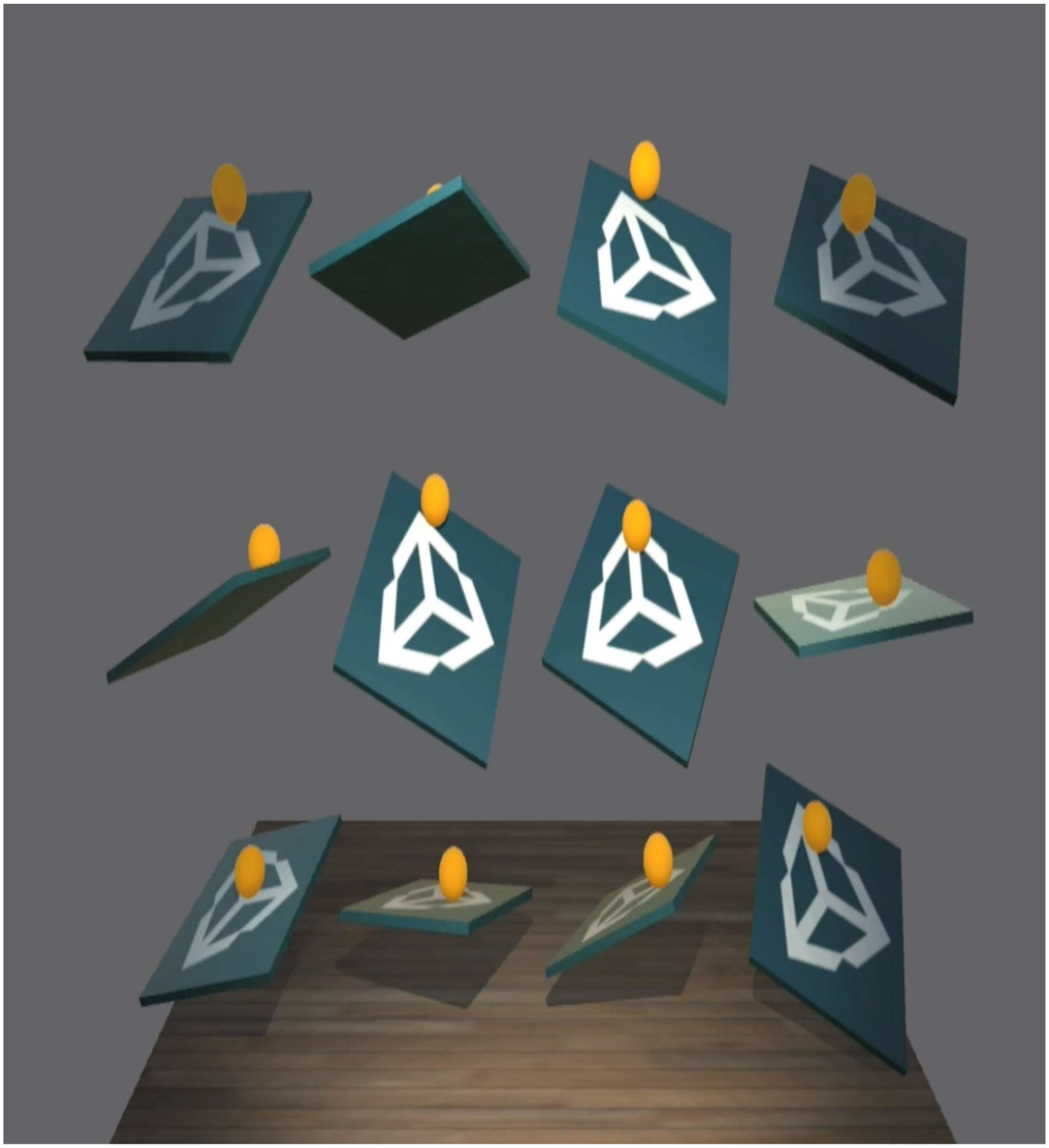
Value Estimate

These values should increase with the reward. They corresponds to how much future reward the agent predicts itself receiving at any given point.

Value Loss

These values will increase as the reward increases, and should decrease when reward becomes stable.

Getting Started with the Balance Ball Example



This tutorial will walk through the end-to-end process of installing Unity Agents, building an example environment, training an agent in it, and finally embedding the trained model into the Unity environment.

Unity ML Agents contains a number of example environments which can be used as templates for new environments, or as ways to test a new ML algorithm to ensure it is functioning correctly.

In this walkthrough we will be using the 3D Balance Ball environment. The environment contains a number of platforms and balls. Platforms can act to keep the ball up by rotating either horizontally or vertically. Each platform is an agent which is rewarded the longer it can keep a ball balanced on it, and provided a negative reward for dropping the ball. The goal of the training process is to have the platforms learn to never drop the ball.

Let's get started!

Installation

In order to install and set-up the Python and Unity environments, see the instructions here.

Building Unity Environment

Launch the Unity Editor, and log in, if necessary.

1. Open the unity-environment folder using the Unity editor. *(If this is not first time running Unity, you'll be able to skip most of these immediate steps, choose directly from the list of recently opened projects)*
    - On the initial dialog, choose Open on the top options
    - On the file dialog, choose unity-environment and click Open *(It is safe to ignore any warning message about non-matching editor installation)*
    - Once the project is open, on the Project panel (bottom of the tool), navigate to the folder Assets/ML-Agents/Examples/3DBall/
    - Double-click the Scene icon (Unity logo) to load all environment assets
2. Go to Edit -> Project Settings -> Player
    - Ensure that Resolution and Presentation -> Run in Background is Checked.
    - Ensure that Resolution and Presentation -> Display

Resolution Dialog is set to Disabled.

3. Expand the Ball3DAcademy GameObject and locate its child object Ball3DBrain within the Scene hierarchy in the editor. Ensure Type of Brain for this object is set to External.
4. *File -> Build Settings*
5. Choose your target platform:
   - (optional) Select "Development Build" to log debug messages.
6. Click *Build*:
   - Save environment binary to the python sub-directory of the cloned repository *(you may need to click on the down arrow on the file chooser to be able to select that folder)*

Training the Brain with Reinforcement Learning

Testing Python API

To launch jupyter, run in the command line:

```
jupyter notebook
```

Then navigate to localhost:8888 to access the notebooks. If you're new to

jupyter, check out the quick start guide before you continue.
To ensure that your environment and the Python API work as expected, you can use the python/Basics Jupyter notebook. This notebook contains a simple walkthrough of the functionality of the API.
Within Basics, be sure to set env_name to the name of the environment file you built earlier.

Training with PPO

In order to train an agent to correctly balance the ball, we will use a Reinforcement Learning algorithm called Proximal Policy Optimization (PPO). This is a method that has been shown to be safe, efficient, and more general purpose than many other RL algorithms, as such we have chosen it as the example algorithm for use with ML Agents. For more information on PPO, OpenAI has a recent blog post explaining it.

In order to train the agents within the Ball Balance environment:

1. Open python/PPO.ipynb notebook from Jupyter.

2. Set env_name to the name of your environment file earlier.
3. (optional) In order to get the best results quickly,
set max_steps to 50000,
set buffer_size to 5000, and
set batch_sizeto 512. For this exercise, this will train the model in approximately ~5-10 minutes.
4. (optional)
Set run_path directory to your choice.
5. Run all cells of notebook with the exception of the last one under "Export the trained Tensorflow graph."

Observing Training Progress

In order to observe the training process in more detail, you can use Tensorboard. In your command line, enter into pythondirectory and then run :

```
tensorboard --logdir=summaries
```

Then navigate to localhost:6006.
From Tensorboard, you will see the summary statistics of six variables:

- Cumulative Reward - The mean cumulative episode reward over all agents. Should increase during a successful training session.

- Value Loss - The mean loss of the value function update. Correlates to how well the model is able to predict the value of each state. This should decrease during a successful training session.
- Policy Loss - The mean loss of the policy function update. Correlates to how much the policy (process for deciding actions) is changing. The magnitude of this should decrease during a successful training session.
- Episode Length - The mean length of each episode in the environment for all agents.
- Value Estimates - The mean value estimate for all states visited by the agent. Should increase during a successful training session.
- Policy Entropy - How random the decisions of the model are. Should slowly decrease during a successful training process. If it decreases too quickly,
  the beta hyperparameter should be increased.

Embedding Trained Brain into Unity Environment *[Experimental]*

Once the training process displays an average reward of ~75 or greater, and there has been a recently saved model (denoted by the Saved Model message) you can choose to stop the training process by stopping the cell execution. Once this is done, you now have a trained TensorFlow model. You must now convert the saved model to a Unity-ready format which can be embedded directly into the Unity project by following the steps below.

Setting up TensorFlowSharp Support

Because TensorFlowSharp support is still experimental, it is disabled by default. In order to enable it, you must follow these steps. Please note that the Internal Brain mode will only be available once completing these steps.

1. Make sure you are using Unity 2017.1 or newer.
2. Make sure the TensorFlowSharp plugin is in your Assets folder. A Plugins folder which includes TF# can be downloaded here. Double click and import it once

downloaded. You can see if this was successfully installed by checking the TensorFlow files in the Project tab under Assets -> ML-Agents -> Plugins -> Computer

3. Go to Edit -> Project Settings -> Player
4. For each of the platforms you target (PC, Mac and Linux Standalone, iOS or Android):
    i. Go into Other Settings.
    ii. Select Scripting Runtime Version to Experimental (.NET 4.6 Equivalent)
    iii. In Scripting Defined Symbols, add the flag ENABLE_TENSORFLOW. After typing in, press Enter.
5. Go to File -> Save Project
6. Restart the Unity Editor.

Embedding the trained model into Unity

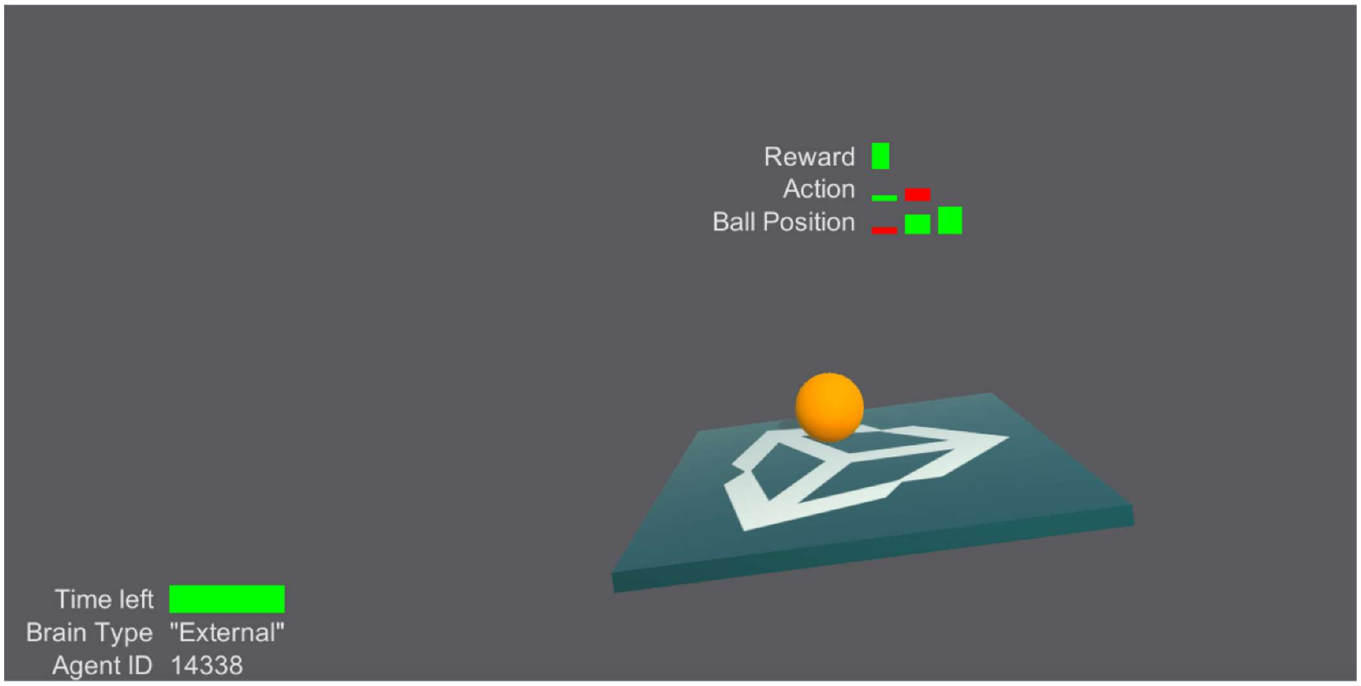
1. Run the final cell of the notebook under "Export the trained TensorFlow graph" to produce an <env_name >.bytes file.
2. Move <env_name>.bytes from python/models/ppo/ into unity-environment/Assets/ML-Agents/Examples/3DBall/TFModels/.

3. Open the Unity Editor, and select the 3DBall scene as described above.
4. Select the Ball3DBrain object from the Scene hierarchy.
5. Change the Type of Brain to Internal.
6. Drag the <env_name>.bytes file from the Project window of the Editor to the Graph Model placeholder in the 3DBallBrain inspector window.
7. Set the Graph Placeholder size to 1 (*Note that step 7 and 8 are done because 3DBall is a continuous control environment, and the TensorFlow model requires a noise parameter to decide actions. In cases with discrete control, epsilon is not needed*).
8. Add a placeholder called epsilon with a type of floating point and a range of values from 0 to 0.
9. Press the Play button at the top of the editor.

If you followed these steps correctly, you should now see the trained model being used to control the behavior of the balance ball within the Editor itself. From here you can re-build the

Unity binary, and run it standalone with your agent's new learned behavior built right in.

Using the Monitor



The monitor allows visualizing information related to the agents or training process within a Unity scene.

You can track many different things both related and unrelated to the agents themselves. To use the Monitor, call the Log function anywhere in your code :

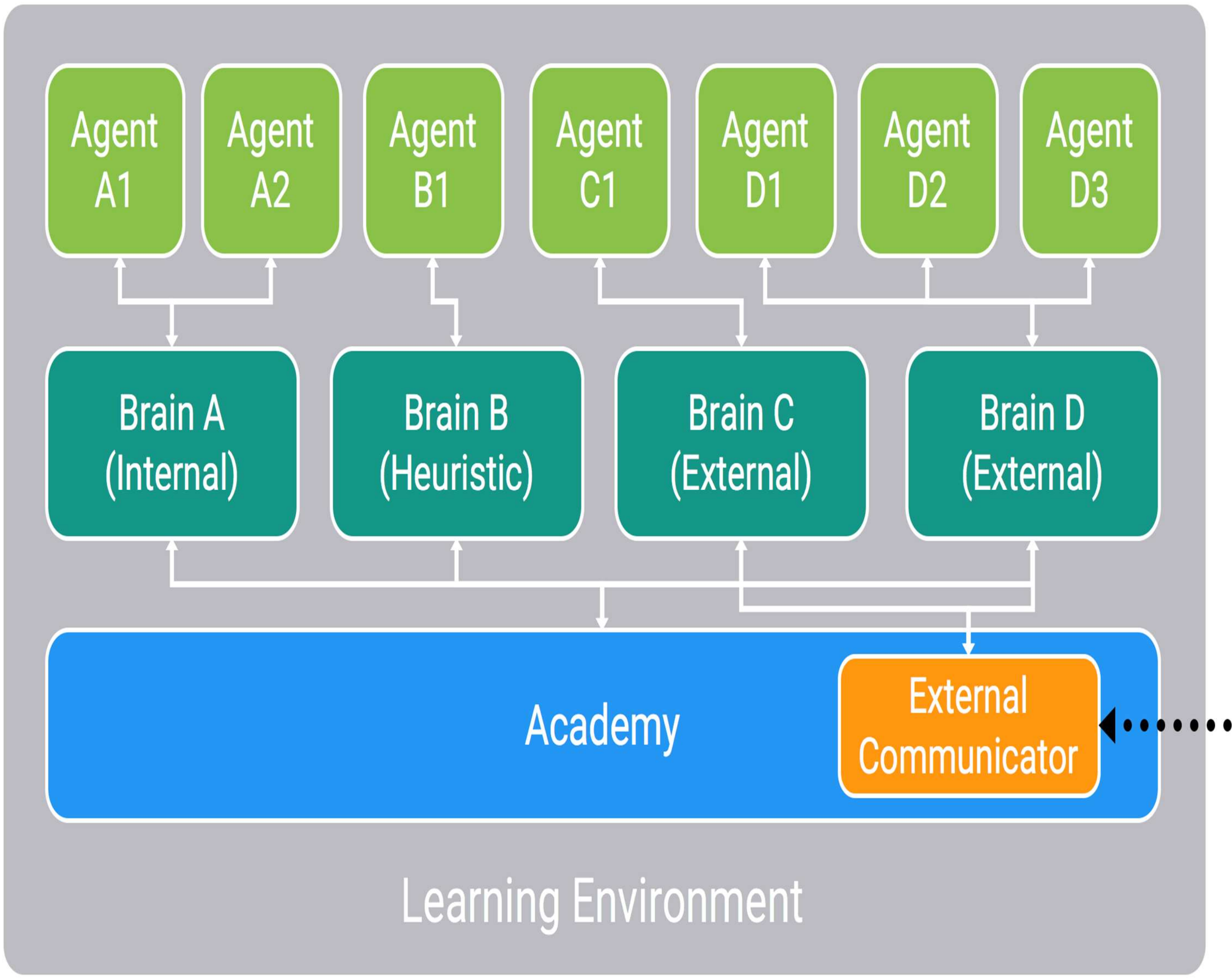
```
   Monitor.Log(key, value, displayType
, target)
```

- *key* is the name of the information you want to display.
- *value* is the information you want to display.
- *displayType* is a MonitorType that can be
  either text, slider, bar or hist.
  - text will convert value into a string and display it. It can be useful for displaying error messages!
  - slider is used to display a single float between -1 and 1. Note that value must be a float if you want to use a slider. If the value is positive, the slider will be green, if the value is negative, the slider will be red.
  - hist is used to display multiple floats. Note that value must be a list or array of floats. The Histogram will be a sequence of vertical sliders.
  - bar is used to see the proportions. Note that value must be a list or array of positive floats. For each float in values, a rectangle of width

    of value divided by the sum of all values will be show. It is best for visualizing values that sum to 1.

- *target* is the transform to which you want to attach information. If the transform is null the information will be attached to the global monitor.

Learning Environments Overview



A visual depiction of how a Learning Environment might be configured within ML-Agents.

The three main kinds of objects within any Agents Learning Environment are:

- Agent - Each Agent can have a unique set of states and observations, take unique actions within the environment, and can receive unique rewards for events within the environment. An agent's actions are decided by the brain it is linked to.

- Brain - Each Brain defines a specific state and action space, and is responsible for deciding which actions each of its linked agents will take. Brains can be set to one of four modes:
  - External - Action decisions are made using TensorFlow (or your ML library of choice) through communication over an open socket with our Python API.
  - Internal (Experimental) - Actions decisions are made using a trained model embedded into the project via TensorFlowSharp.
  - Player - Action decisions are made using player input.
  - Heuristic - Action decisions are made using hand-coded behavior.
- Academy - The Academy object within a scene also contains as children all Brains within the environment. Each environment contains a single Academy which defines the scope of the environment, in terms of:

- Engine Configuration - The speed and rendering quality of the game engine in both training and inference modes.
- Frameskip - How many engine steps to skip between each agent making a new decision.
- Global episode length - How long the the episode will last. When reached, all agents are set to done.

The states and observations of all agents with brains set to External are collected by the External Communicator, and communicated via the Python API. By setting multiple agents to a single brain, actions can be decided in a batch fashion, taking advantage of the inherently parallel computations of neural networks. For more information on how these objects work together within a scene, see our wiki page.

Flexible Training Scenarios

With the Unity ML-Agents, a variety of different kinds of training scenarios are possible, depending on how agents, brains, and rewards are connected. We are excited to see what kinds of novel and fun environments the community creates. For those new to training intelligent agents, below are a few examples that can serve as inspiration. Each is a prototypical environment configurations with a description of how it can be created using the ML-Agents SDK.

- Single-Agent - A single agent linked to a single brain. The traditional way of training an agent. An example is any single-player game, such as Chicken. Video Link.
- Simultaneous Single-Agent - Multiple independent agents with independent reward functions linked to a single brain. A parallelized version of the traditional training scenario, which can speed-up and stabilize the training process. An example might be training a dozen robot-arms to each open a door simultaneously. Video Link.
- Adversarial Self-Play - Two interacting agents with inverse reward functions linked to a single brain. In two-player games, adversarial self-play can allow an agent to become increasingly more skilled, while always having the

perfectly matched opponent: itself. This was the strategy employed when training AlphaGo, and more recently used by OpenAI to train a human-beating 1v1 Dota 2 agent.

- Cooperative Multi-Agent - Multiple interacting agents with a shared reward function linked to either a single or multiple different brains. In this scenario, all agents must work together to accomplish a task than couldn’t be done alone. Examples include environments where each agent only has access to partial information, which needs to be shared in order to accomplish the task or collaboratively solve a puzzle. (Demo project coming soon)
- Competitive Multi-Agent - Multiple interacting agents with inverse reward function linked to either a single or multiple different brains. In this scenario, agents must compete with one another to either win a competition, or obtain some limited set of resources. All team sports would fall into this scenario. (Demo project coming soon)

- Ecosystem - Multiple interacting agents with independent reward function linked to either a single or multiple different brains. This scenario can be thought of as creating a small world in which animals with different goals all interact, such a savanna in which there might be zebras, elephants, and giraffes, or an autonomous driving simulation within an urban environment. (Demo project coming soon)

Additional Features

Beyond the flexible training scenarios made possible by the Academy/Brain/Agent system, ML-Agents also includes other features which improve the flexibility and interpretability of the training process.

- Monitoring Agent’s Decision Making - Since communication in ML-Agents is a two-way street, we provide an Agent Monitor class in Unity which can display aspects of the trained agent, such as policy and value output within the Unity environment itself. By providing these outputs in real-time, researchers and developers can more easily debug an agent’s behavior.

- Curriculum Learning - It is often difficult for agents to learn a complex task at the beginning of the training process. Curriculum learning is the process of gradually increasing the difficulty of a task to allow more efficient learning. ML-Agents supports setting custom environment parameters every time the environment is reset. This allows elements of the environment related to difficulty or complexity to be dynamically adjusted based on training progress.
- Complex Visual Observations - Unlike other platforms, where the agent’s observation might be limited to a single vector or image, ML-Agents allows multiple cameras to be used for observations per agent. This enables agents to learn to integrate information from multiple visual streams, as would be the case when training a self-driving car which required multiple cameras with different viewpoints, a navigational agent which might need to integrate aerial and first-

person visuals, or an agent which takes both a raw visual input, as well as a depth-map or object-segmented image.

- Imitation Learning (Coming Soon) - 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. In a future release, ML-Agents will provide the ability to record all state/action/reward information for use in supervised learning scenarios, such as imitation learning. By utilizing imitation learning, a player can provide demonstrations of how an agent should behave in an environment, and then utilize those demonstrations to train an agent in either a standalone fashion, or as a first-step in a reinforcement learning process.

Using TensorFlowSharp in Unity (Experimental)

Unity now offers the possibility to use pretrained TensorFlow graphs inside of the game engine. This was made possible thanks to the TensorFlowSharp project.

Notice: This feature is still experimental. While it is possible to embed trained models into Unity games, Unity Technologies does not officially support this use-case for production games at this time. As such, no guarantees are provided regarding the quality of experience. If you encounter issues regarding battery life, or general performance (especially on mobile), please let us know.

Supported devices :

- Linux 64 bits
- Mac OSX 64 bits
- Windows 64 bits
- iOS (Requires additional steps)
- Android

Requirements

- Unity 2017.1 or above
- Unity Tensorflow Plugin (Download here)

Using TensorflowSharp with ML-Agents

In order to bring a fully trained agent back into Unity, you will need to make sure the nodes of your graph have appropriate names. You can give names to nodes in Tensorflow :

```
variable= tf.identity(variable, name="variable_name")
```

We recommend using the following naming convention:

- Name the batch size input placeholder batch_size
- Name the input state placeholder state
- Name the output node action
- Name the recurrent vector (memory) input placeholder recurrent_in (if any)
- Name the recurrent vector (memory) output node recurrent_out (if any)
- Name the observations placeholders input placeholders observation_i where i is the index of the observation (starting at 0)

You can have additional placeholders for float or integers but they must be placed in placeholders of dimension 1 and size 1. (Be sure to name them)

It is important that the inputs and outputs of the graph are exactly the one you receive / give when training your model with an External brain. This means you cannot have any operations such as reshaping outside of the graph. The object you get by calling step or reset has fields states, observations and memories which must correspond to the placeholders of your graph. Similarly, the arguments action and memory you pass to step must correspond to the output nodes of your graph.
While training your Agent using the Python API, you can save your graph at any point of the training. Note that the argument output_node_names must be the name of the tensor your graph outputs (separated by a coma if multiple outputs). In this case, it will be
either action or action,recurrent_out if you have recurrent outputs.

```
from tensorflow.python.tools import freeze_graph

freeze_graph.freeze_graph(input_graph = model_path +'/raw_graph_def.pb',
              input_binary = True,
              input_checkpoint = last_checkpoint,
              output_node_names =
"action",
              output_graph =
model_path +'/your_name_graph.bytes' ,
              clear_devices = True,
initializer_nodes = "",input_saver =
"",
              restore_op_name =
"save/restore_all",
filename_tensor_name = "save/Const:0")
```

Your model will be saved with the name your_name_graph.bytes and will contain both the graph and associated weights. Note that you must save your graph as a bytes file so Unity can load it.

Inside Unity

Go to Edit -> Player Settings and add ENABLE_TENSORFLOW to the Scripting Define Symbols for each type of device you want to use (PC, Mac and Linux Standalone, iOS or Android).
Set the Brain you used for training to Internal.
Drag your_name_graph.bytes into Unity and then drag it into The Graph Model field in the Brain. If you used a scope when training you graph, specify it in the Graph Scope field.
Specify the names of the nodes you

used in your graph. If you followed these instructions well, the agents in your environment that use this brain will use you fully trained network to make decisions.

iOS additional instructions for building

- Once you build for iOS in the editor, Xcode will launch.
- In General -> Linked Frameworks and Libraries:
  - Add a framework called Framework.accelerate
  - Remove the library libtensorflow-core.a
- In Build Settings->Linking->Other Linker Flags:
  - Double Click on the flag list
  - Type -force_load
  - Drag the library libtensorflow-core.a from the Project Navigator on the left under Libraries/ML-Agents/Plugins/iOS into the flag list.

Using TensorflowSharp without ML-Agents

Beyond controlling an in-game agent, you may desire to use TensorFlowSharp for more general computation. The below instructions describe how to generally embed Tensorflow models without using the ML-Agents framework.

You must have a Tensorflow graph your_name_graph.bytes made using Tensorflow's freeze_graph.py. The process to create such graph is explained above.

Inside of Unity

Put the file your_name_graph.bytes into Resources.
In your C# script : At the top, add the line

```
using TensorFlow;
```

If you will be building for android, you must add this block at the start of your code :

```
#if UNITY_ANDROID
TensorFlowSharp.Android.NativeBinding.Init();
#endif
```

Put your graph as a text asset in the variable graphModel. You can do so in the inspector by making graphModel a public variable and dragging you asset in the inspector or load it from the Resources folder :

```
TextAsset graphModel = Resources.Load
(your_name_graph) as TextAsset;
```

You then must recreate the graph in Unity by adding the code :

```
graph = new TFGraph ();
graph.Import (graphModel.bytes);
session = new TFSession (graph);
```

Your newly created graph need to get input tensors. Here is an example with a one dimensional tensor of size 2:

```
var runner = session.GetRunner ();
runner.AddInput (graph
["input_placeholder_name"] [0], new
float[]{ placeholder_value1,
placeholder_value2 });
```

You need to give all required inputs to the graph. There is one input per TensorFlow placeholder.

To retrieve the output of your graph run the following code. Note that this is for an output that would be a two dimensional tensor of floats. Cast to a long array if your outputs are integers.

```
runner.Fetch
(graph["output_placeholder_name"][0]);
```

```
float[,] recurrent_tensor = runner.Run
() [0].GetValue () as float[,];
```

Python API

Environment Permission Error

If you directly import your Unity environment without building it in the editor, you might need to give it additionnal permissions to execute it.

If you receive such a permission error on macOS, run:

```
chmod -R 755 *.app
```

or on Linux:

```
chmod -R 755 *.x86_64
```

On Windows, you can find instructions here.

Environment Connection Timeout

If you are able to launch the environment from UnityEnvironment but then recieve a timeout error, there may be a number of possible causes.

- *Cause*: There may be no Brains in your environment which are set to External. In this case, the environment will not attempt to

communicate with python. *Solution*: Set the brain you wish to externally control through the Python API to External from the Unity Editor, and rebuild the environment.
- *Cause*: On OSX, the firewall may be preventing communication with the environment. *Solution*: Add the built environment binary to the list of exceptions on the firewall by following instructions here.

Filename not found

If you receive a file-not-found error while attempting to launch an environment, ensure that the environment files are in the root repository directory. For example, if there is a sub-folder containing the environment files, those files should be removed from the sub-folder and moved to the root.

Communication port {} still in use

If you receive an exception "Couldn't launch new environment because communication port {} is still in use. ", you can change the worker number in the python script when calling

```
UnityEnvironment(file_name=filename,
worker_id=X)
```

Mean reward : nan

If you recieve a message Mean reward : nan when attempting to train a model using PPO, this is due to the episodes of the learning environment not terminating. In order to address this, set Max Steps for either the Academy or Agents within the Scene Inspector to a value greater than 0. Alternatively, it is possible to manually set done conditions for episodes from within scripts for custom episode-terminating events.

Best Practices when training with PPO

The process of training a Reinforcement Learning model can often involve the need to tune the hyperparameters in order to achieve a level of performance that is desirable. This guide contains some best practices for tuning the training process when the default parameters don't seem to be giving the level of performance you would like.

Hyperparameters

Batch Size

batch_size corresponds to how many experiences are used for each gradient descent update. This should always be a fraction of the buffer_size. If you are using a continuous action space, this value should be large (in 1000s). If you are using a discrete action space, this value should be smaller (in 10s).
Typical Range (Continuous): 512 - 5120
Typical Range (Discrete): 32 - 512

Beta (Used only in Discrete Control)

beta corresponds to the strength of the entropy regularization, which makes the policy "more random." This ensures that discrete action space agents properly explore during training. Increasing this will ensure more random actions are taken. This should be adjusted such that the entropy (measurable from TensorBoard) slowly decreases alongside increases in reward. If entropy drops too quickly, increase beta. If entropy drops too slowly, decrease beta.
Typical Range: 1e-4 - 1e-2

Buffer Size

buffer_size corresponds to how many experiences should be collected before gradient descent is performed on them all. This should be a multiple of batch_size. Typically larger buffer sizes correspond to more stable training updates.
Typical Range: 2048 - 409600

Epsilon

epsilon corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating. Setting this value small will result in more stable updates, but will also slow the training process.
Typical Range: 0.1 - 0.3

Hidden Units

hidden_units correspond to how many units are in each fully connected layer of the neural network. For simple problems where the correct action is a straightforward combination of the state inputs, this should be small. For problems where the action is a very complex

interaction between the state variables, this should be larger.
Typical Range: 32 - 512

Learning Rate

learning_rate corresponds to the strength of each gradient descent update step. This should typically be decreased if training is unstable, and the reward does not consistently increase.
Typical Range: 1e-5 - 1e-3

Number of Epochs

num_epoch is the number of passes through the experience buffer during gradient descent. The larger the batch size, the larger it is acceptable to make this. Decreasing this will ensure more stable updates, at the cost of slower learning.
Typical Range: 3 - 10

Time Horizon

time_horizon corresponds to how many steps of experience to collect per-agent before adding it to the experience buffer. In cases where there are frequent rewards within an episode, or episodes are prohibitively

large, this can be a smaller number. For most stable training however, this number should be large enough to capture all the important behavior within a sequence of an agent's actions.
Typical Range: 64 - 2048

Max Steps

max_steps corresponds to how many steps of the simulation (multiplied by frame-skip) are run durring the training process. This value should be increased for more complex problems.
Typical Range: 5e5 - 1e7

Normalize

normalize corresponds to whether normalization is applied to the state inputs. This normalization is based on the running average and variance of the states. Normalization can be helpful in cases with complex continuous control problems, but may be harmful with simpler discrete control problems.

Number of Layers

num_layers corresponds to how many hidden layers are present after the

state input, or after the CNN encoding of the observation. For simple problems, fewer layers are likely to train faster and more efficiently. More layers may be necessary for more complex control problems.
Typical range: 1 - 3

Training Statistics

To view training statistics, use Tensorboard. For information on launching and using Tensorboard, see here.

Cumulative Reward

The general trend in reward should consistently increase over time. Small ups and downs are to be expected. Depending on the complexity of the task, a significant increase in reward may not present itself until millions of steps into the training process.

Entropy

This corresponds to how random the decisions of a brain are. This should consistently decrease during training. If it decreases too soon or not at all, beta should be adjusted (when using discrete action space).

Learning Rate

This will decrease over time on a linear schedule.

Policy Loss

These values will oscillate with training.

Value Estimate

These values should increase with the reward. They corresponds to how much future reward the agent predicts itself receiving at any given point.

Value Loss

These values will increase as the reward increases, and should decrease when reward becomes stable.

Python API

Notice: Currently communication between Unity and Python takes place over an open socket without authentication. As such, please make *sure that the network where training takes place is secure. This will be addressed in a future release.*

Loading a Unity Environment

Python-side communication happens through UnityEnvironment which is located in python/unityagents. To load a Unity environment from a built binary file, put the file in the same directory as unityagents. In python, run:

```
from unityagents import UnityEnvironment
env = UnityEnvironment(file_name=filename, worker_id=0)
```

- file_name is the name of the environment binary (located in the root directory of the python project).
- worker_id indicates which port to use for communication with the environment. For use in parallel training regimes such as A3C.

Interacting with a Unity Environment

A BrainInfo object contains the following fields:

- observations : A list of 4 dimensional numpy arrays. Matrix n of the list corresponds to the nth observation of the brain.
- states : A two dimensional numpy array of dimension (batch size, state size) if the state space is continuous and (batch size, 1) if the state space is discrete.
- memories : A two dimensional numpy array of dimension (batch size, memory size) which corresponds to the memories sent at the previous step.
- rewards : A list as long as the number of agents using the brain containing the rewards they each obtained at the previous step.
- local_done : A list as long as the number of agents using the brain containing done flags (wether or not the agent is done).
- agents : A list of the unique ids of the agents using the brain.

Once loaded, env can be used in the following way:

- Print : print(str(env))
  Prints all parameters relevant to the loaded environment and the external brains.

- Reset : env.reset(train_model=True, config=None)
  Send a reset signal to the environment, and provides a dictionary mapping brain names to BrainInfo objects.
  - train_model indicates whether to run the environment in train (True) or test (False) mode.
  - config is an optional dictionary of configuration flags specific to the environment. For more information on adding optional config flags to an environment, see here. For generic environments, config can be ignored. config is a dictionary of strings to floats where the keys are the names of the resetParameters and the values are their corresponding float values.
- Step : env.step(action, memory=None, value = None)
  Sends a step signal to the environment using the actions. For each brain :
  - action can be one dimensional arrays or two dimensional arrays if you have multiple agents per brains.

    - memory is an optional input that can be used to send a list of floats per agents to be retrieved at the next step.
    - value is an optional input that be used to send a single float per agent to be displayed if and AgentMonitor.cscomponent is attached to the agent.

Note that if you have more than one external brain in the environment, you must provide dictionaries from brain names to arrays
for action, memory and value. For example: If you have two external brains named brain1 and brain2 each with one agent taking two continuous actions, then you can have:

```
action = {'brain1':[1.0, 2.0], 'brain2':[3.0,4.0]}
```

Returns a dictionary mapping brain names to BrainInfo objects.

- Close : env.close()
  Sends a shutdown signal to the environment and closes the communication socket

Environment Design Best Practices

General

- It is often helpful to start with the simplest version of the problem, to ensure the agent can learn it. From there increase complexity over time. This can either be done manually, or via Curriculum Learning, where a set of lessons which progressively increase in difficulty are presented to the agent (learn more here).
- When possible, it is often helpful to ensure that you can complete the task by using a Player Brain to control the agent.

Rewards

- The magnitude of any given reward should typically not be greater than 1.0 in order to ensure a more stable learning process.
- Positive rewards are often more helpful to shaping the desired behavior of an agent than negative rewards.
- For locomotion tasks, a small positive reward (+0.1) for forward velocity is typically used.
- If you want the agent to finish a task quickly, it is often helpful

to provide a small penalty every step (-0.05) that the agent does not complete the task. In this case completion of the task should also coincide with the end of the episode.
- Overly-large negative rewards can cause undesirable behavior where an agent learns to avoid any behavior which might produce the negative reward, even if it is also behavior which can eventually lead to a positive reward.

States

- States should include all variables relevant to allowing the agent to take the optimally informed decision.
- Categorical state variables such as type of object (Sword, Shield, Bow) should be encoded in one-hot fashion (ie 3 -> 0, 0, 1).
- Besides encoding non-numeric values, all inputs should be normalized to be in the range 0 to +1 (or -1 to 1). For example rotation information on GameObjects should be recorded as state.Add(transform.rotation.eulerAngles.y/180.0f-1.0f); rather

than state.Add(transform.rotation.y);. See the equation below for one approach of normalization.

- Positional information of relevant GameObjects should be encoded in relative coordinates wherever possible. This is often relative to the agent position.

$$normalizedValue = \frac{currentValue - minValue}{maxValue - minValue}$$

Actions

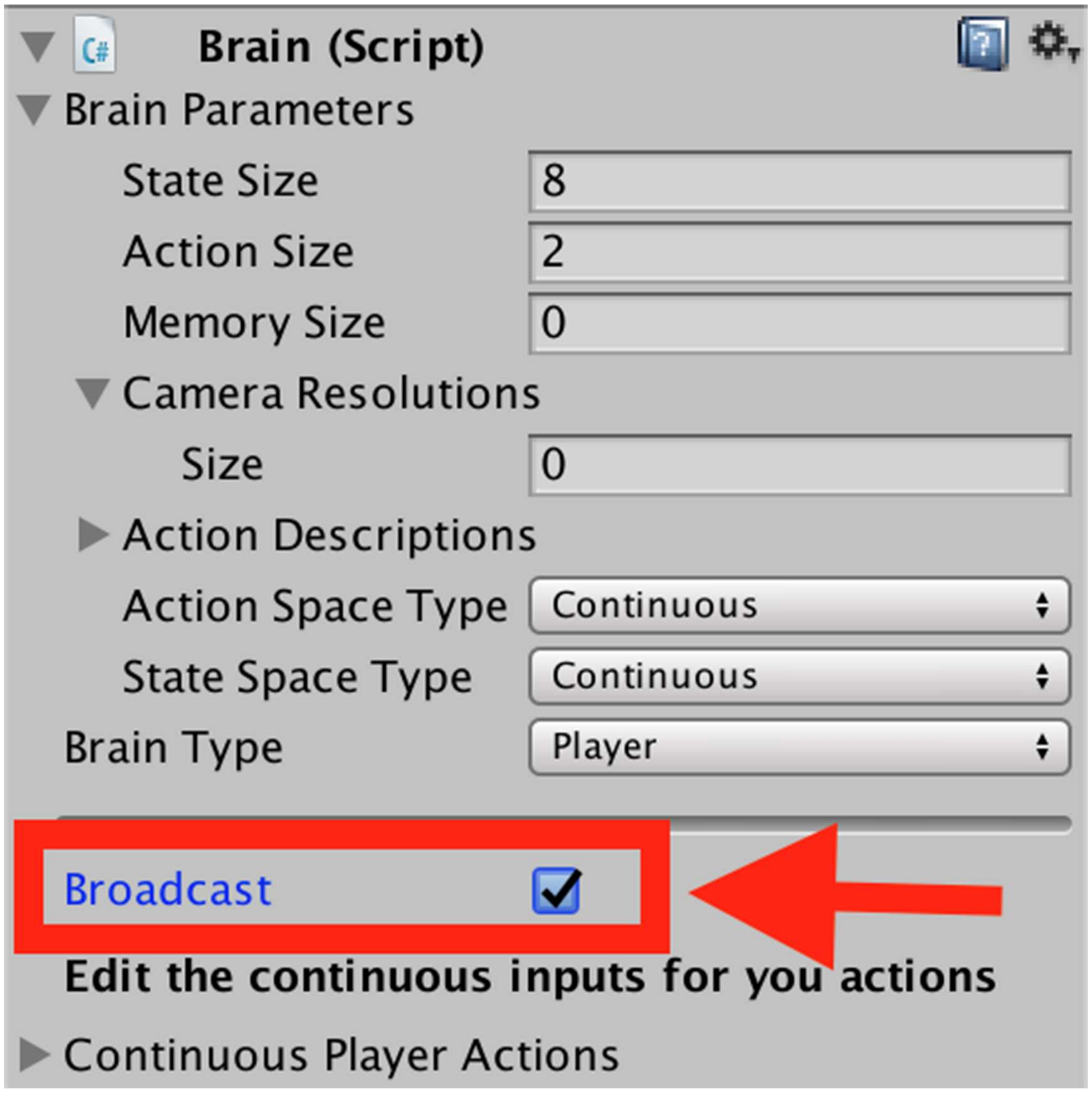
- When using continuous control, action values should be clipped to an appropriate range.
- Be sure to set the action-space-size to the number of used actions, and not greater, as doing the latter can interfere with the efficency of the training process.

Using the Broadcast Feature

The Player, Heuristic and Internal brains have been updated to support broadcast. The broadcast feature allows you to collect data from your agents in python without controling them.

How to use : Unity

To turn it on in Unity, simply check the Broadcast box as shown bellow:



Brain (Script)

Brain Parameters

State Size: 8

Action Size: 2

Memory Size: 0

Camera Resolutions

Size: 0

Action Descriptions

Action Space Type: Continuous

State Space Type: Continuous

Brain Type: Player

Broadcast: ✓

Edit the continuous inputs for you actions

Continuous Player Actions

How to use : Python

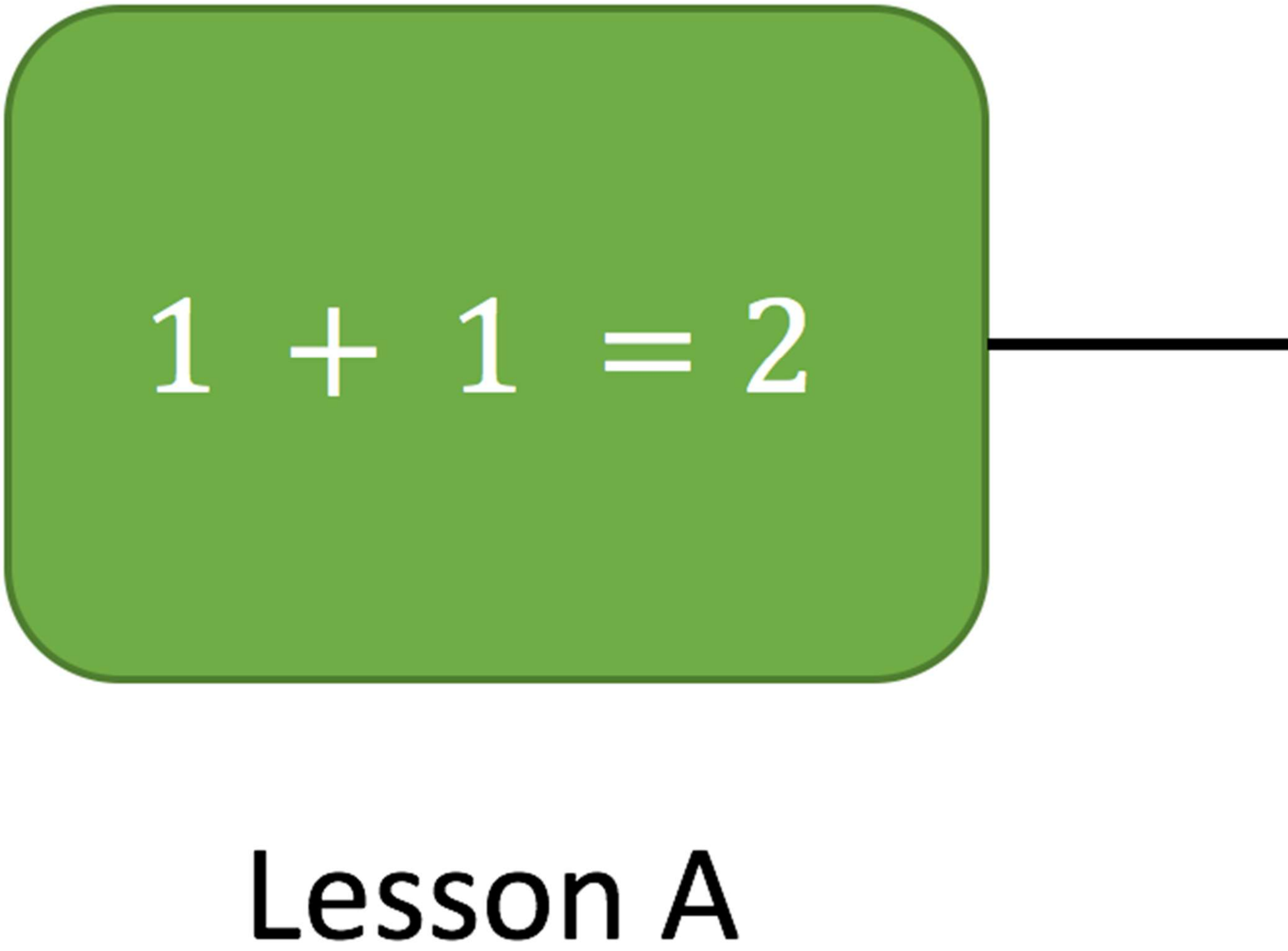
When you launch your Unity Environment from python, you can see what the agents connected to non-external

brains are doing. When
calling step or reset on your environment, you retrieve a dictionary from brain names to BrainInfo objects. Each BrainInfo the non-external brains set to broadcast.
Just like with an external brain, the BrainInfo object contains the fields
for observations, states, memories,rewards, local_done, agents and previous_actions. Note
that previous_actions corresponds to the actions that were taken by the agents at the previous step, not the current one.
Note that when you do a step on the environment, you cannot provide actions for non-external brains. If there are no external brains in the scene, simply call step() with no arguments.
You can use the broadcast feature to collect data generated by Player, Heuristics or Internal brains game sessions. You can then use this data to train an agent in a supervised context.

Training with Curriculum Learning

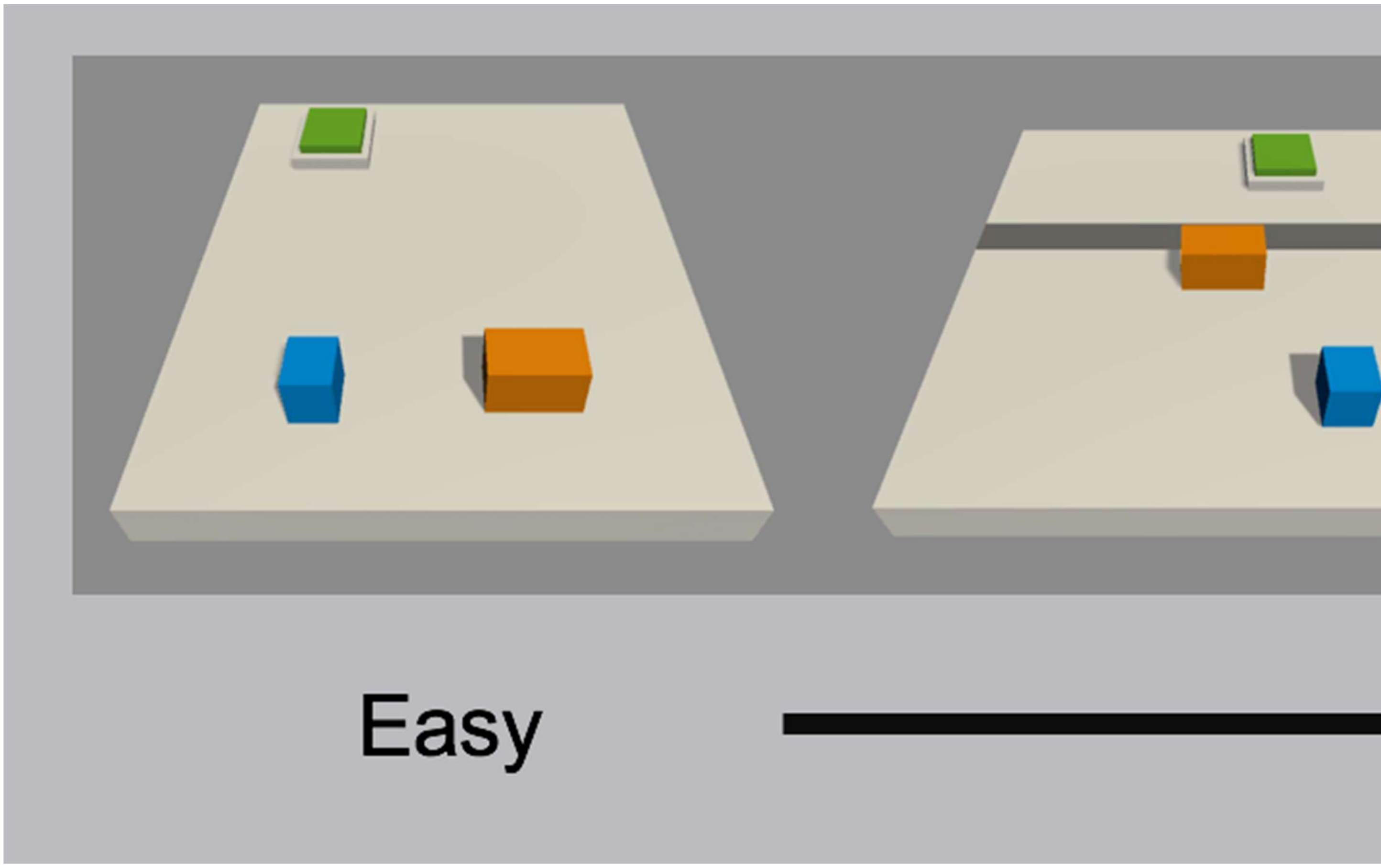
Background

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. Here is a link to the original paper which introduces the ideal formally. More generally, this idea has been around much longer, for it is how we humans typically learn. If you imagine any childhood primary school education, there is an ordering of classes and topics. Arithmetic is taught before algebra, for example. Likewise, algebra is taught before calculus. The skills and knowledge learned in the earlier subjects provide a scaffolding for later lessons. The same principle can be applied to machine learning, where training on easier tasks can provide a scaffolding for harder tasks in the future.



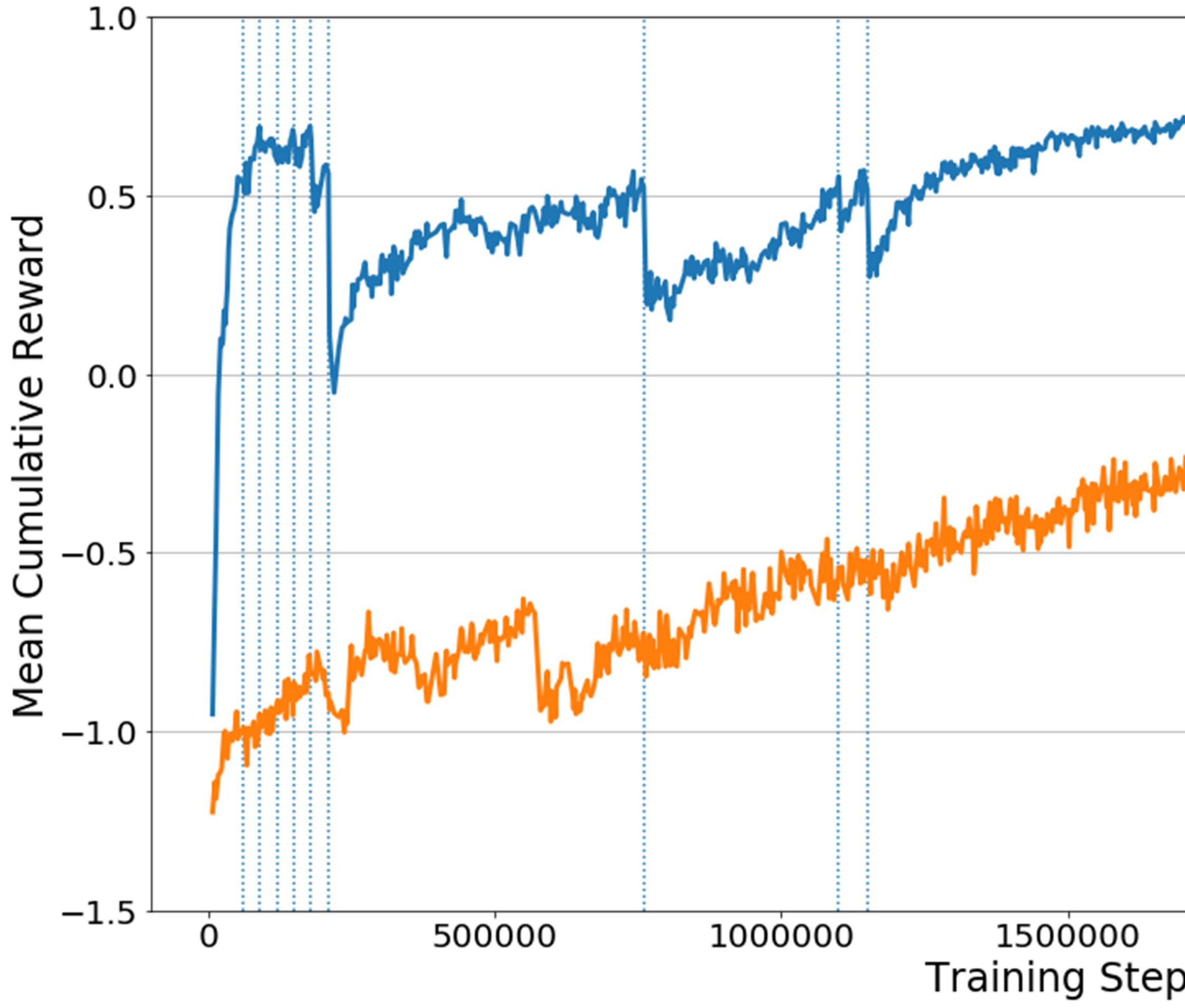
Example of a mathematics curriculum. Lessons progress from simpler topics to more complex ones, with each building on the last.

When we think about how Reinforcement Learning actually works, the primary learning signal is a scalar reward received occasionally throughout training. In more complex or difficult tasks, this reward can often be sparse, and rarely achieved. For example, imagine a task in which an agent needs to scale a wall to arrive at a goal. The starting point when training an agent to accomplish this task will be a random policy. That starting policy will have the agent running in circles, and will likely never, or very rarely scale the wall properly to the achieve the reward. If we start with a simpler task, such as moving toward an unobstructed goal, then the agent can easily learn to accomplish the task. From there, we can slowly add to the difficulty of the task by increasing the size of the wall, until the agent can complete the initially near-impossible task of scaling the wall. We are including just such an environment with ML-Agents 0.2, called Wall Area.



Demonstration of a curriculum training scenario in which a progressively taller wall obstructs the path to the goal.

To see this in action, observe the two learning curves below. Each displays the reward over time for an agent trained using PPO with the same set of training hyperparameters. The difference is that the agent on the left was trained using the full-height wall version of the task, and the right agent was trained using the curriculum version of the task. As you can see, without using curriculum learning the agent has a lot of difficulty. We think that by using well-crafted curricula, agents trained using reinforcement learning will be able to accomplish tasks otherwise much more difficult.



How-To

So how does it work? In order to define a curriculum, the first step is to decide which parameters of the environment will vary. In the case of the Wall Area environment, what varies is the height of the wall. We can define this as a reset parameter in the Academy object of our scene, and by doing so it becomes adjustable via the Python API. Rather than adjusting it by hand, we then create a simple JSON file which describes the structure of the curriculum. Within it we can set at what points in the training process our wall height will change, either based on the percentage of training steps which have taken place, or what the average reward the agent has received in the recent past is. Once these are in place, we simply launch ppo.py using the –curriculum-file flag to point to the JSON file, and PPO we will train using Curriculum Learning. Of course we can then keep track of the current lesson and progress via TensorBoard.

```
{
    "measure" : "reward",
    "thresholds" : [0.5, 0.5, 0.5,
0.5, 0.5, 0.5, 0.5, 0.5, 0.5],
    "min_lesson_length" : 2,
    "signal_smoothing" : true,
    "parameters" :
    {
        "min_wall_height" : [0.0, 0.5,
1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0,
4.5],
        "max_wall_height" : [1.5, 2.0,
2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5,
6.0]
    }
}
```

- measure - What to measure learning progress, and advancement in lessons by.
    - reward - Uses a measure received reward.
    - progress - Uses ratio of steps/max_steps.
- thresholds (float array) - Points in value of measure where lesson should be increased.
- min_lesson_length (int) - How many times the progress measure should be reported before incrementing the lesson.
- signal_smoothing (true/false) - Whether to weight the current progress measure by previous values.

    - If true, weighting will be 0.75 (new) 0.25 (old).
- parameters (dictionary of key:string, value:float array) - Corresponds to academy reset parameters to control. Length of each array should be one greater than number of thresholds.

Training on Amazon Web Service

This page contains instructions for setting up an EC2 instance on Amazon Web Service for use in training ML-Agents environments. Current limitations of the Unity Engine require that a screen be available to render to. In order to make this possible when training on a remote server, a virtual screen is required. We can do this by installing Xorg and creating a virtual screen. Once installed and created, we can display the Unity environment in the virtual environment, and train as we would on a local machine.

Pre-Configured AMI

A public pre-configured AMI is available with the ID: ami-30ec184a in the us-east-1 region. It was created as a modification of the Amazon Deep Learning AMI.

Configuring your own Instance

Instructions here are adapted from this Medium post on running general Unity applications in the cloud.

1. To begin with, you will need an EC2 instance which contains the latest Nvidia drivers, CUDA8, and cuDNN. There are a number of external tutorials which describe this, such as:
    - Getting CUDA 8 to Work With openAI Gym on AWS and Compiling Tensorflow for CUDA 8 Compatibility
    - Installing TensorFlow on an AWS EC2 P2 GPU Instance
    - Updating Nvidia CUDA to 8.0.x in Ubuntu 16.04 – EC2 Gx instance
2. Move python to remote instance.
3. Install the required packages with pip install ..

4. Run the following commands to install Xorg:

```
5. sudo apt-get update
6. sudo apt-get install -y xserver-
   xorg mesa-utils
7. sudo nvidia-xconfig -a --use-
   display-device=None --
   virtual=1280x1024
```

8. Restart the EC2 instance.

Launching your instance

1. Make sure there are no Xorg processes running. To kill the Xorg processes, run sudo killall Xorg. Note that you might have to run this command multiple times depending on how Xorg is configured.
   If you run nvidia-smi, you will have a list of processes running on the GPU, Xorg should not be in the list.

2. Run:

```
3. sudo /usr/bin/X :0 &
4. export DISPLAY=:0
```

5. To ensure the installation was succesful, run glxgears. If there are no errors, then Xorg is correctly configured.

6. There is a bug in *Unity 2017.1* which requires the uninstallation of libxrandr2, which can be removed with :

```
sudo apt-get remove --purge libwxgtk3.0-0v5
sudo apt-get remove --purge libxrandr2
```

This is scheduled to be fixed in 2017.3.

Testing

If all steps worked correctly, upload an example binary built for Linux to the instance, and test it from python with:

```python
from unityagents import UnityEnvironment

env = UnityEnvironment(your_env)
```

You should receive a message confirming that the environment was loaded succesfully.

Limitations and Common Issues

Unity SDK

Headless Mode

Currently headless mode is disabled. We hope to address these in a future version of Unity.

Rendering Speed and Synchronization

Currently the speed of the game physics can only be increased to 100x real-time. The Academy also moves in time with FixedUpdate() rather than Update(), so game behavior tied to frame updates may be out of sync.

macOS Metal Support

When running a Unity Environment on macOS using Metal rendering, the application can crash when the lock-screen is open. The solution is to set rendering to OpenGL. This can be done by navigating: Edit -> Project Settings -> Player. Clicking on Other Settings. Unchecking Auto Graphics API for Mac. Setting OpenGL Core to be above Metal in the priority list.

Unity ML Agents Documentation Origin

About

- Unity ML Agents Overview

- Example Environments
-

Tutorials

- Installation & Set-up
- Getting Started with the Balance Ball Environment
- Making a new Unity Environment
- How to use the Python API

Features

- Agents SDK Inspector Descriptions
- Scene Organization
- Curriculum Learning
- Broadcast
- Monitor
- Training on the Cloud with Amazon Web Services
- TensorflowSharp in Unity [Experimental]

Best Practices

- Best practices when creating an Environment
- Best practices when training using PPO

Help

- Limitations & Common Issues