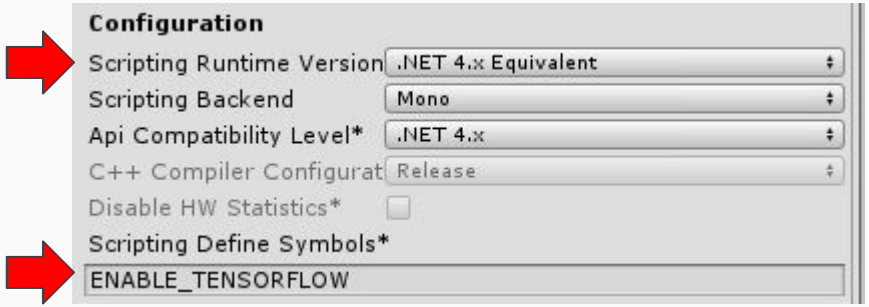
Setting Up a Unity-ML Environment

By Rafael Bayer

Prerequisite

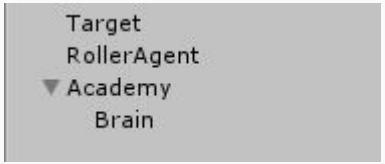
- Requirements
  - Tensorflow
  - Unity3D
  - Tensorsharp plugin
  - Unity ML-Agents asset package
  - Python modules
    - Docopt
    - Pyyaml
    - Pillow

- Project settings
  - Edit -> project settings -> player
  - Other settings



Unity Scene Components

- **Academy**
  - Manager game object, base class of Academy
  - Responsible for managing and configuring the brains and agents
- **Brain**
  - Communicates with the Agents
  - Receives input to feed into neural network (Vector observations)
  - Outputs neural networks response (Vector actions)
- **Agent**
  - Makes Observations and decisions using Brain
  - Manages reward functions
  - Sends input to network (Vector observations)
  - Uses neural network outputs (Vector actions)



Academy Setup

- The academy requires very little setup
- All academy's inherit from the academy base class, and therefore already have many built-in capabilities
- Can be used to reset scene after agent completes/fails tasks

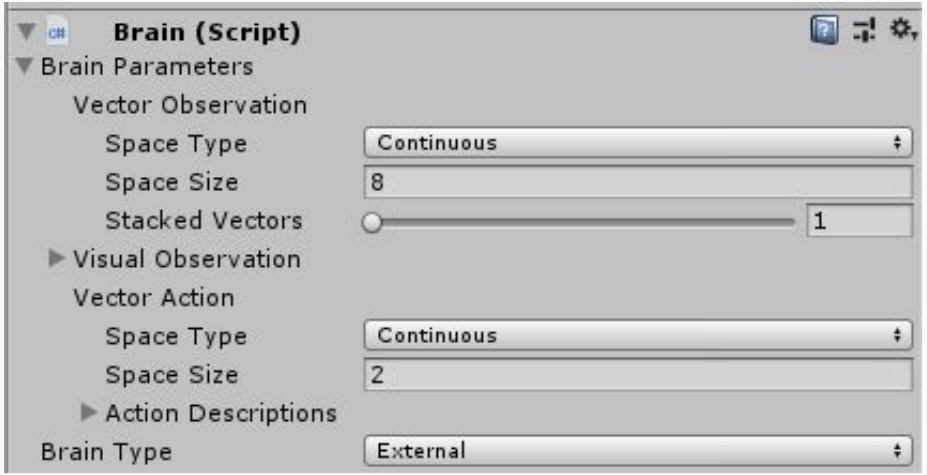
Example Implementation from the RollerBall project

```
1
2   using UnityEngine;
3   using MLAgents;
4
5   public class RollerAcademy : Academy {
6
7   }
```

Brain Setup

- The Brain object is already prebuilt, it is not a subclass you have to write like the academy
- Requires setup within the actual unity editor based on the number and types of expected inputs and outputs to the network
- Brain type can be changed to **external** for training, **internal** to use training data, or **player** to test the responses to inputs

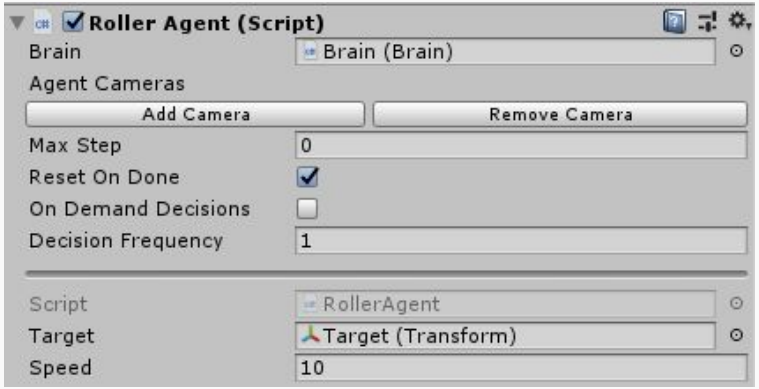
Example brain implementation from the RollerBall project



Agent Setup

- The neural network interacts with the game through the agent
- Subclass of Agent
- Responsible for sending inputs to the network via **AddVectorObs()** in overridden **CollectObservations()** method
- Responsible for processing outputs with overridden AgentAction method and **float[] vectorAction** parameter

- Responsible for adding and removing reward in response to events in the game via **addReward();**
- Responds to success or failure of task via **Done();** method



AgentReset();

- Method of Agent class
- Called every time agent is marked **Done();**
- Used to reset the agent and scene to its default state at the start of a training episode
    - Often involves resetting agents position, velocity, and other variables
    - Reset environment, reset the obstacles/objectives if needed

```
public override void AgentReset()
```

CollectObservations();

- Method of Agent class
- Used to give observations to the neural network
- Variables are sent using **AddVectorObservation(variable);**
- Observations can either be continuous or discrete
  - **Continuous**
    - Float, float[], Vector2, Vector3, etc,
  - **Discrete**
    - Bool, int, int[], etc,

The same number of Observations must be sent every time **CollectObservations();** is called, and that number must match the vector observation space size in the Brain's inspector window

```
public override void CollectObservations()
```

AgentAction();

- Method of Agent class
- Used to assign reward with **addReward(float);** and make use of network output
- Receives **float[] vectorAction** parameter from the brain, the size of vectorAction depends on the action space size set in the brains inspector window

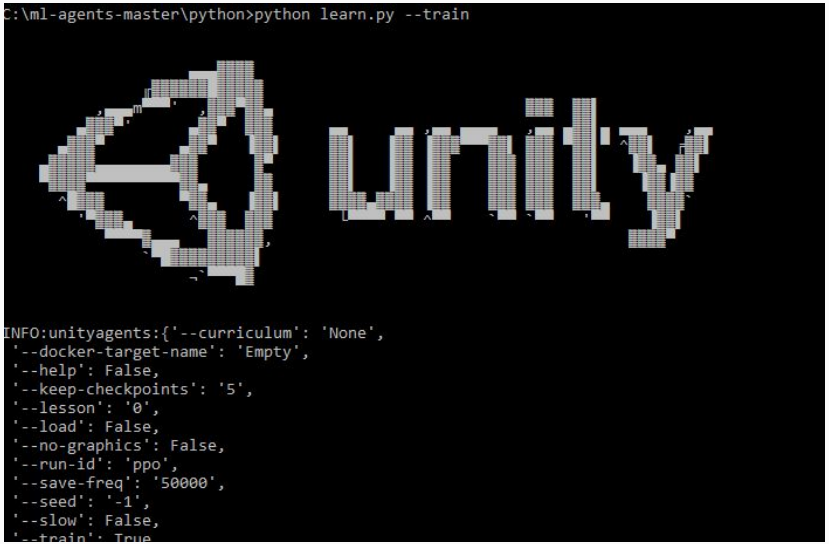
Example of using vectorAction values

```
Vector3 controlSignal = Vector3.zero;
controlSignal.x = vectorAction[0];
rb.AddForce(controlSignal * speed);
```

Starting the training process

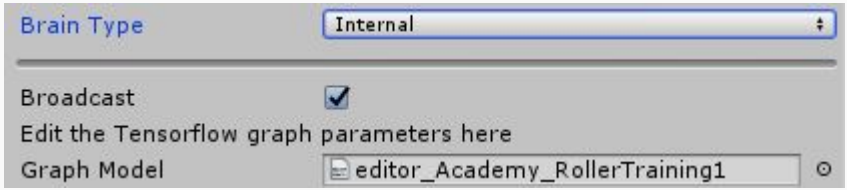
- Once the environment is setup training can begin
- Switch the Brain type to “**external**”
- Open up cmd and change directory to the ML-Agents python folder
- Use the command “**python learn.py --run-id=<your id> --train**” to begin the training process (replace <your id> with a name for this training

- Press **play** in the Unity Editor and the training will begin!



Using your training data

- Once train is completed, a graph model will be created and added to the models folder in your ML-agents folder
- Switch the brain back to **internal**
- Import the graph model file into your Unity Projects assets folder



- Press play and the Agent will now use the trained neural network to play the game!

Issues along the way

- If the training session isn't long enough for adequate results, the settings can be configured in the trainer_config.yaml file in the ML-agents folder
- During the training process summaries will be given every n number of steps (specified in .yaml file), the reward given should gradually increase over time
- If it doesn't seem to be improving check your inputs, try to normalize them between [-1, 1]
- Consider increasing the number and types of inputs
- Consider parallel training, in my experience it greatly increased the speed and effectiveness of training

My next steps

- Improving the infinite runner
- Figuring out a more definite way to measure the improvement of a network after training (tensorboard?)
- Figuring out how to add multiple training sessions of data together, instead of just increasing the number of steps in a single session
- Better inputs to network
- More consistent reward and punishment
- Neurovevolution?