Getting started with Unity ML Agents

The Unity ML agents has a pretty comprehensive documentation and I highly recommend to read all of it to understand how unity ml agents work in depth. In this document, I try to give a a brief introduction on how to get started with ML agents by building a simple game in which we will train an agent to dodge falling balls. I assume that the reader has basic knwledge of unity. This is the first time I attempted to develop games with unity and ml agents, so there is always scope of improvement in the project. Lets get started!

Installation

Follow the steps here to install unity ml agents. If you do have trouble finding the "com.unity.ml-agents " subdirectory, switch to master branch of the ml agents repository by executing the command:

```
git checkout master
```

The subdirectory will be within the main ml-agents directory. For this tutorial, I will be using the latest version of unity ml-agents which is version *0.13.2*. The library is still in beta so expect changes with every new iteration. However they provide changelogs so you can go through them to check what has changed!

Note - To train your unity model, make sure your python ml-agents api and ml-agents-env versions are the same. These package are not backward compatible with each other. If you do get an version mismatch error, reinstall the packages with the same version.

What we will be doing

Note - In this tutorial I will only be focusing on ML-Agents functions and properties and not with developing the mechanics of the game such as instantiating objects, scene etc. I assume the reader has basics Unity skills and is familiar with C#.

Create new project on unity

Select 2D as we will be developing a 2D game. Once you have the project running in the editor, open the manifest.json file in the Packages directory of your project. Add the following line to your project's package dependencies:

```
 "com.unity.ml-agents" : "file:<path_to_local_ml-agents_repo>/com.unity.ml-agents"
```

Confirm that ML agents package has been added correctly by checking if ML-Agents option exists under Components.

Methods to Override

To define an ML agent, it is **necessary** to override three methods. They are:

1. `CollectObservations()` - Defines all the data required to train a RL-based model for our agent.
2. `AgentAction(int[] action)` - Defines actions our agent can take.

3. `AgentReset()` - Called to reset the agent.

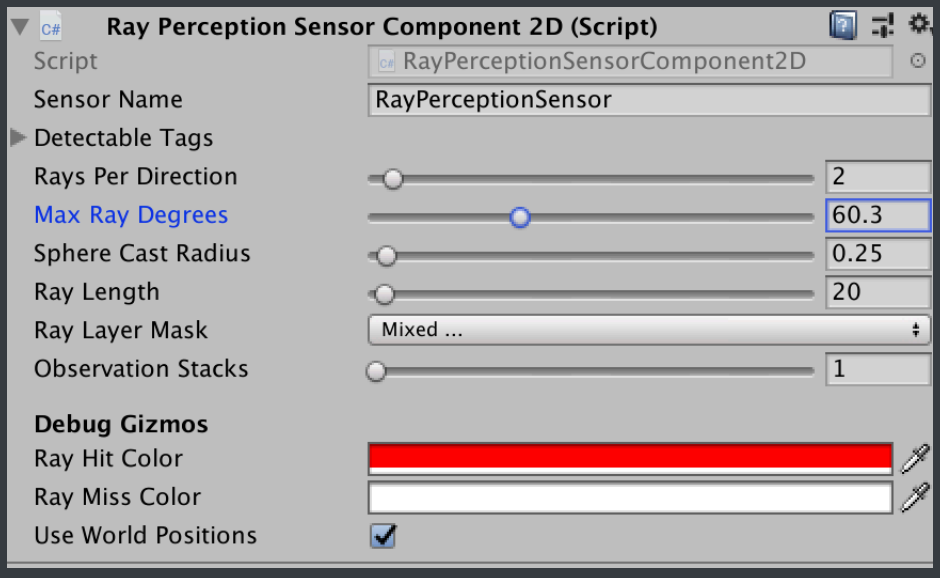
Defining and Collecting Observations

To make decisions, an agent must observe its environment in order to infer the state of the world. A state observation can take the following forms:

- **Vector Observation** — a feature vector consisting of an array of floating point numbers.
- **Visual Observations** — one or more camera images and/or render textures.

When you use vector observations for an Agent, implement the `Agent.CollectObservations(VectorSensor sensor)` method to create the feature vector. When you use **Visual Observations**, you only need to identify which Unity Camera objects or RenderTextures will provide images and the base Agent class handles the rest.

For our agent, we will only be defining the Vector observations. To add vector observation, use `AddVectorObs()`. In our dodgeball example, we will be adding one feature vector that is the distance between our agent and the falling balls. With previous version of ml-agents, you would have to define and add multiple raycasts for object detection in the `CollectObservations` method but with the recent version you can add `Ray Perception Sensor Component` script in the agent inspector window. This automatically collects observations from the environemnt and adds them to the feature vectors of the agent.



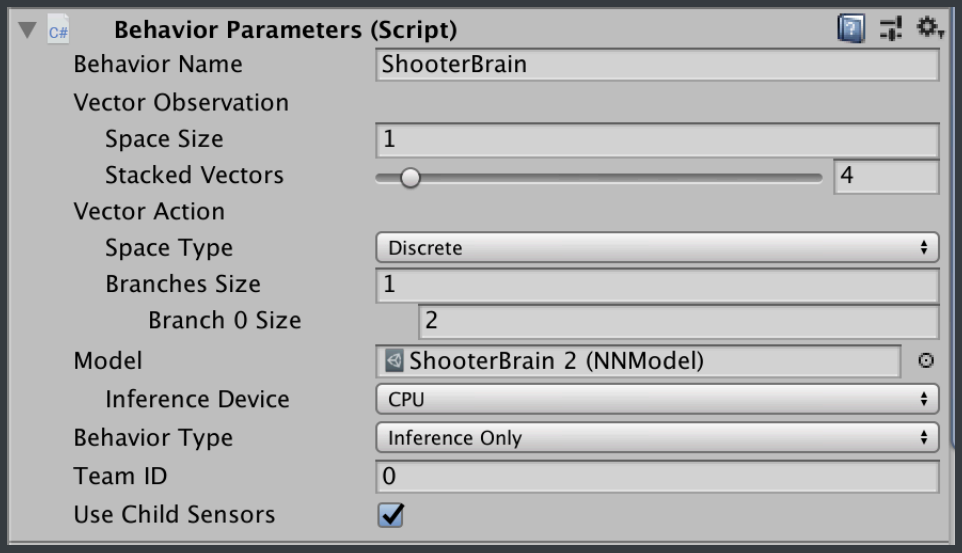
In our example we are adding a total of 5 raycasts to our agent. The agent always has a vertical raycast by default and the rays per direction specify the number of rays on the left and right of the agents Z axis, spread 30 degrees wide. Hence there are a total of 5 raycasts.

Defining Actions

This is one of the most important part of training your model - defining what action your agent can perform in the environment. There are two kinds of action space - Discrete and Continuous. When you specify that the vector action space as Continuous, the action parameter passed to the Agent is an array of control signals with length equal to the Vector Action Space Size property. When you specify a Discrete vector action space type, the action parameter is an array containing integers. In this tutorial, I will be focusing on just Discrete action space. These actions are defined in the **AgentAction()** method.

1. ## Defining Discrete Action Space

   In Discrete action space, an array of integers is passed to the AgentAction Function. The length of this array determines how many **types** of action our agent can take. It is always better to separate out the action behaviour of our agent since it would become increasingly hard to encode all possible actions in just one index of the array. For example, suppose our agent could both walk in 4 directions and also rotate. Instead of defining 7 integer actions in one branch, we split them into two branhces. The first branch would contain 4 possible integers each mapping to a different direction and our second branch would contain 3 possible integers each mapping to a rotation in x,y, z axis. Perhaps, an example would make this clearer. **Note** - These actions are indexed from 0. We define this length as the Branch size in Behaviour Parameters script of our agents inspector.



In the above example we are defining our action space array of length 1 and in its only brach (branch - 0), we define the number of possible actions our agent can take in the first branch - here, 2. Now how do we define what action our agent can take? Thats where the AgentAction function comes into picture. **Note**- We are using the same example below for our project.Check the example below:

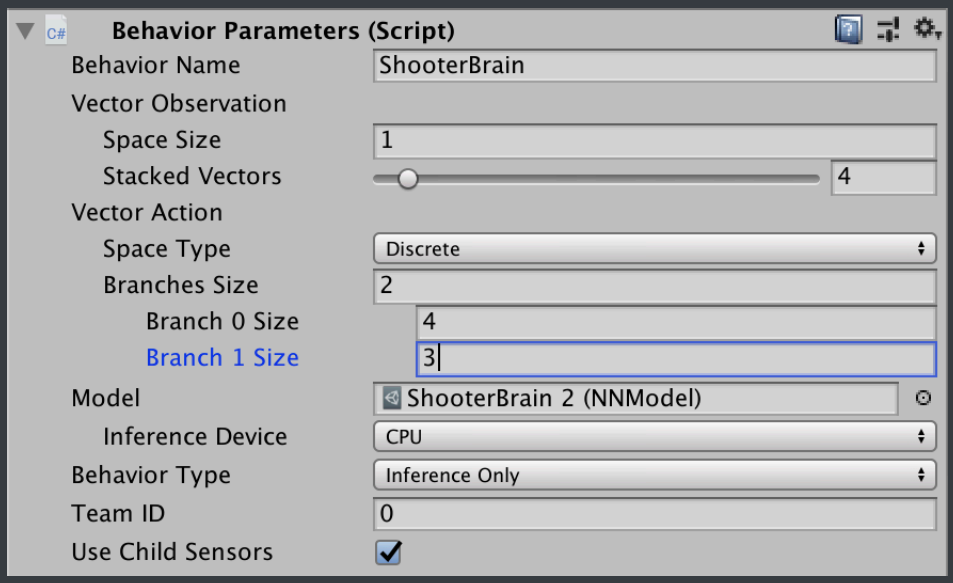
```
``` C#
    public void AgentAction(float[] action)
    {
        //For every frame surived
```
```

```
         AddReward(0.001f);
         addVariableReward();
        int act = Mathf.FloorToInt(action[0]);
        switch (act)
        {
             case 0:
                 transform.Translate(Vector2.right * speed * Time.deltaTime);
                 break;
             case 1:
                 transform.Translate(-Vector2.right * speed * Time.deltaTime);
                 break;
        }
    }
```

Since we only have one branch we just need to look at only one index of action array that is 0. This branch has 2 possible values - 0 and 1 as the branch size is just 2. If the action number is 0, we are moving the agent to the right and when it is one, we are moving it to the left.

Neither the Policy nor the training algorithm know anything about what the action values themselves mean. The training algorithm simply tries different values for the action list and observes the affect on the accumulated rewards over time and many training episodes. Thus, the only place actions are defined for an Agent is in the `AgentAction()` function.

Another example:



In the above script, we have 2 branches. The first one can probably correspond to movements and the second one can be rotation. The mapping of these actions are only defined in the AgentAction Function. Lets take a look at the function:

```c#
    public void AgentAction(float[] action){
        int movement = action[0];
        int rotation = action[1];

        switch(movement){
            case 0:
                transform.Translate(Vector2.right * speed * Time.deltaTime);
                break;
            case 1:
                transform.Translate(-Vector2.right * speed * Time.deltaTime);
                break;
            case 2:
                transform.Translate(Vector2.up * speed * Time.deltaTime);
                break;
            case 3:
                transform.Translate(Vector2.down * speed * Time.deltaTime);
                break;
        }
```

```
        switch(rotation){
            case 0:
                transform.Rotate( 0.0f, 0.0f, 45f);
                break;
            case 1:
                transform.Rotate( 45f, 0.0f, 0.0f);
                break;
            case 2:
                transform.Rotate(0.0f, 60f, 0.0f);
                break;
        }
    }
```

Sample example showing how multiple branches and action space can be defined!

Agent Properties

```c#
* `Behavior Parameters` - The parameters dictating what Policy the Agent will receive.
  * `Behavior Name` - The identifier for the behavior. Agents with the same behavior name
  will learn the same policy.
   this is used as the top-level key in the config.
  * `Vector Observation`
    * `Space Size` - Length of vector observation for the Agent.
    * `Stacked Vectors` - The number of previous vector observations that will
      be stacked and used collectively for decision making. This results in the
      effective size of the vector observation being passed to the Policy being:
      _Space Size_ x _Stacked Vectors_.
  * `Vector Action`
    * `Space Type` - Corresponds to whether action vector contains a single
      integer (Discrete) or a series of real-valued floats (Continuous).
    * `Space Size` (Continuous) - Length of action vector.
    * `Branches` (Discrete) - An array of integers, defines multiple concurrent
```

```
      discrete actions. The values in the `Branches` array correspond to the
      number of possible discrete values for each action branch.
  * `Model` - The neural network model used for inference (obtained after
  training)
  * `Inference Device` - Whether to use CPU or GPU to run the model during
inference
  * `Behavior Type` - Determines whether the Agent will do training,
inference, or use its
  Heuristic() method:
    * `Default` - the Agent will train if they connect to a python trainer,
otherwise they will perform inference.
    * `Heuristic Only` - the Agent will always use the `Heuristic()` method.
    * `Inference Only` - the Agent will always perform inference.
  * `Team ID` - Used to define the team for [self-play](Training-Self-Play.md)
  * `Use Child Sensors` - Whether to use all Sensor components attached to
child GameObjects of this Agent.
* `Max Step` - The per-agent maximum number of steps. Once this number is
  reached, the Agent will be reset.
```

Heuristics Mode

It is always a good idea to test your environment manually before embarking on an extended training run. To do so, you will need to implement the `Heuristic()` method on the Agent's class This will allow you control the Agent using direct keyboard control.

```
    public override float[] Heuristic()
    {
         if (Input.GetKey(KeyCode.D))
        {
            Debug.Log("D");
            return new float[] {0};
        }
        if (Input.GetKey(KeyCode.A))
        {
            return new float[] {1};
        }
        return new float[] {-1};
    }
```

For our dodgeball example, since we only have two possible movements, that is either move left or move right, we just need to provide with two input controls. Make sure that the returned value is between 0 and the branch size. To use an agent in the heuristic mode switch to Heuristics only in the inference type parameter of the agent.

Agent Reset

When the agent performs an unintended action such as not dodging or getting hit by ball, we reset the agent and start from the initial point. Each restart is called an episode that is part of a training sequence. You can reset the agent by calling `Done()` . This automatically calls the `AgentReset` method. Below is the `AgentReset` method for our example:

```
        public override void AgentReset()
        {
            speed = 6f;
            angle = 15f;
            score = 0f;
            text.text = "Score: ";
            targetTime = maxTime + Time.time;
            transform.position = new Vector3(0, -Camera.main.orthographicSize/2,
    0);
            script.Reset();
        }
```

This method performs all the necessary actions to re-initialize the environment back to the initial state.

Rewards

In reinforcement learning, the reward is a signal that the agent has done something right. Rewards are given in the `AgentAction` function. Rewards are given by calling the `AddReward()` or `SetReward()` method. It is advisable to keep the reward between [-1, 1]. For our example, we give an reward of 0.001f for every decision made. Additionally, when the agent survives long enough(target time), we give it the ultimate reward of 1f and resent the environment. Thus, the main goal of the agent is to dodge the ball and survive for a certain period of time. A reward of -1f is given when it is collides with a ball and the environment is ultimately reset.

Variable Rewards

For this project, I try to award variable rewards as well to see impact on the agent decision and training. From our example, it is obvious to see that we need to stay out of the path of a falling block. So every time the agent's raycast hits no ball, it is out of any falling ball and we give it a certain positive reward. Similarly, whenever the agent is in the path of a falling block, we given it a negative reward, which is a function of the distance between the ball and agent. Check the code below:

```
        private void addVariableReward(){
            float distance = rayCast(upRay, hit);
            if(distance == 0f)
                AddReward(0.01f);
            else{
                if(distance < 3f)
                    AddReward(-1f);
                else{
                    float inverse = -1/(distance/raycastDistance);
                    AddReward(-inverse * 0.1f);
                }
            }
        }
```

Training

The ML-Agents toolkit conducts training using an external Python training process. During training, this external process communicates with the Academy to generate a block of agent experiences. These experiences become the training set for a neural network used to optimize the agent's policy (which is essentially a mathematical function mapping observations to actions). In reinforcement learning, the neural network optimizes the policy by maximizing the expected rewards.

The output of the training process is a model file containing the optimized policy. This model file is a TensorFlow data graph containing the mathematical operations and the optimized weights selected during the training process. You can set the generated model file in the Behaviors Parameters under your Agent in your Unity project to decide the best course of action for an agent.

On Agent's inspector window, add the DecisionRequester script (from ML Agents package). The decision period states how often your agent requests a decision from the external python API (your Tensorflow model). The smaller this is, the more intensive the training is.

Before we dive into commands into training the agents, we need to first define the configuration of our training, that is the neural network architecture and model. This is done through a YAML file. Unity ML agents provides with a default trainer_config.yaml file which we will be using to train our agent. But first lets go through some basics of the configuration. Below is the trainer_config file we will be using to train our agent.

```yaml
    default:
        trainer: ppo
        batch_size: 1024
        beta: 5.0e-3
        buffer_size: 10240
        epsilon: 0.2
        hidden_units: 64
        lambd: 0.95
        learning_rate: 6.0e-4
        learning_rate_schedule: linear
        max_steps: 5.0e7
        memory_size: 512
        normalize: false
        num_epoch: 3
        num_layers: 3
        time_horizon: 64
        sequence_length: 64
        summary_freq: 10000
        use_recurrent: false
        vis_encode_type: simple
        reward_signals:
            extrinsic:
                strength: 1.0
                gamma: 0.4
```

Lets go through some of the important paramters. The rest is explained in detail in unity-ml agents documentation.

1. trainer - The algorithm to use. There is PPO and SAC
2. Reward_signals - Types of rewards for the agent.
3. Batch_size - Training samples required to perform one back-propogation in the neural network.
4. Lambd - Regularization Loss parameter
5. Num_layers - no of hidden layers in the neural network
6. Hidden_units - no of units in one hidden layer.

The basic command for training the agent in unity editor is:

```
mlagents-learn <trainer-config-file> --run-id=<some-id> --train
```

You will be prompted with a message in the terminal, that says to click on play button in the unity editor. On click, the agent begins to train. The unity editor interacts with the tensorflow model through the python API. You can terminate the training process any time by hittin Ctrl+C. The command will save your model.

Inference

Locate the model you just trained and drag it into the unity editor assets folder. To use this model for inference, drag this same model to the model option in the agents behavior paramters script and change from default to inference only. This loads the model and now unity will use this model to run our agent. Click on play button to proceed!