

Letting our agents make decisions

We want our agent to learn which actions to take given a certain state of the environment — e.g. if the ball is on our side of the court, our agent should get it before it hits the floor.

The goal of reinforcement learning is to learn the **best policy** (a mapping of states to actions) **that will maximise possible rewards**. The theory behind how reinforcement learning algorithms achieve this is beyond the scope of this series, but the courses I shared in the [series introduction](#) will cover it in great depth.

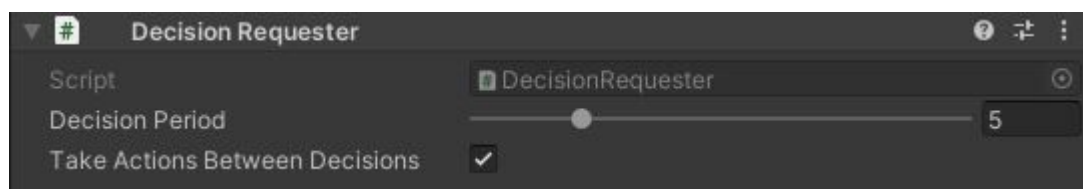
While training, the agent will either take actions:

1. At random (to explore which actions lead to rewards and which don't)
2. From its current policy (the optimal action given the current state)

ML-Agents provides a convenient **Decision Requester** component which will handle the alternation between these for us during training.

To add a Decision Requester:

1. Select the **PurpleAgent** game object (within the **PurplePlayArea** parent).
2. Add Component > Decision Requester.
3. Leave decision period as 5.



Defining the agent behavior

Both agents are already set up with the VolleyballAgent.cs script and **Behavior Parameters** component (which we'll come back to later).

In this part we'll walk through VolleyballAgent.cs. This script contains all the logic that defines the agents' actions and observations. It contains some helper methods already:

●Start() — called when the environment is first rendered. Grabs the parent Volleyball environment and saves it to a variable `envController` for easy reference to its methods later.

●Initialize() — called when the **agent** is first initialized. Grabs some useful constants and objects. Also sets `agentRot` to ensure symmetry so that the same policy can be shared between both agents.

●MoveTowards(), CheckIfGrounded() & Jump() — from ML-Agents sample projects. Used for jumping.

●OnCollisionEnter() — called when the Agent collides with something. Used to update `lastHitter` to decide which agent gets penalized if the ball is hit out of bounds or rewarded if hit over the net.

Adding an agent in Unity ML-Agents usually involves extending the base `Agent` class, and implementing the following methods:

●OnActionReceived()

●Heuristic()

●CollectObservations()

●OnEpisodeBegin() (**Note:** usually used for resetting starting conditions. We don't implement it here, because the reset logic is already defined at the environment-level in `VolleyballEnvController`. This makes more sense for us since we also need to reset the ball in addition to the agents.)

Agent actions

At a high level, the Decision Requester will select an action for our agent to take and trigger `OnActionReceived()`. This in turn calls `MoveAgent()`.

MoveAgent()

This method resolves the selected action.

Within the `MoveAgent()` method, start by declaring vector variables for our agents direction and rotation movements:

```
var dirToGo = Vector3.zero;
var rotateDir = Vector3.zero;
```

We'll also add a 'grounded' check to see whether its possible for the agent to jump:

```
var grounded = CheckIfGrounded();
```

The actions passed into this method (actionBuffers.DiscreteActions) will be an array of integers which we'll map to some behavior. It's not important which order we assign them, as long as they remain consistent:

```
var dirToGoForwardAction = act[0];
var rotateDirAction = act[1];
var dirToGoSideAction = act[2];
var jumpAction = act[3];
```

In Unity, every object has a transform class that stores its position, rotation and scale. We'll use it to create a vector pointing to the correct direction in which we want our agent to move.

Based on the previous assignment order, this is how we'll map our actions to behaviors:

- 1.dirToGoForwardAction: Do nothing=0 | Move forward=1 | Move backward=2
- 2.rotateDirAction: Do nothing=0 | Rotate clockwise=1 | Rotate anti-clockwise=2
- 3.dirToGoSideAction: Do nothing=0 | Move left=1 | Move right=2
- 4.jumpAction: Don't jump=0 | Jump=1

Add to the MoveAgent() method:

```
if (dirToGoForwardAction == 1)
    dirToGo = (grounded ? 1f : 0.5f) * transform.forward * 1f;
else if (dirToGoForwardAction == 2)
    dirToGo = (grounded ? 1f : 0.5f) * transform.forward * volleyballSettings.speedReductionFactor * -1f;

if (rotateDirAction == 1)
    rotateDir = transform.up * -1f;
else if (rotateDirAction == 2)
    rotateDir = transform.up * 1f;
```

```

if (dirToGoSideAction == 1)
    dirToGo = (grounded ? 1f : 0.5f) * transform.right * volleyballSettings.speedReductionFactor * -1f;
else if (dirToGoSideAction == 2)
    dirToGo = (grounded ? 1f : 0.5f) * transform.right * volleyballSettings.speedReductionFactor;

if (jumpAction == 1)
{
    if (((jumpingTime <= 0f) && grounded))
    {
        Jump();
    }
}

```

Note: volleyballSettings.speedReductionFactor is a constant that slows backwards and strafe movement to be more 'realistic'.

Next, apply the movement using Unity's provided Rotate and AddForce methods:

```

transform.Rotate(rotateDir, Time.fixedDeltaTime * 200f);
agentRb.AddForce(agentRot * dirToGo * volleyballSettings.agentRunSpeed,
    ForceMode.VelocityChange);

```

Finally, add in the logic for controlling jump behavior:

```

// makes the agent physically "jump"
if (jumpingTime > 0f)
{
    jumpTargetPos =
        new Vector3(agentRb.position.x,
            jumpStartingPos.y + volleyballSettings.agentJumpHeight,
            agentRb.position.z + agentRot * dirToGo);

    MoveTowards(jumpTargetPos, agentRb, volleyballSettings.agentJumpVelocity,
        volleyballSettings.agentJumpVelocityMaxChange);
}

// provides a downward force to end the jump
if (!jumpingTime > 0f && !grounded)

```

```

{
    agentRb.AddForce(
        Vector3.down * volleyballSettings.fallingForce, ForceMode.Acceleration);
}

// controls the jump sequence
if (jumpingTime > 0f)
{
    jumpingTime -= Time.fixedDeltaTime;
}

```

Heuristic()

To test that we've resolved the actions properly, let's implement the Heuristic() method.

This will map actions to a keyboard input, so that we can playtest as a human controller.

Add to Heuristic():

```

var discreteActionsOut = actionsOut.DiscreteActions;
if (Input.GetKey(KeyCode.D))
{
    // rotate right
    discreteActionsOut[1] = 2;
}
if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow))
{
    // forward
    discreteActionsOut[0] = 1;
}
if (Input.GetKey(KeyCode.A))
{
    // rotate left
    discreteActionsOut[1] = 1;
}
if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow))
{
    // backward
    discreteActionsOut[0] = 2;
}

```

```

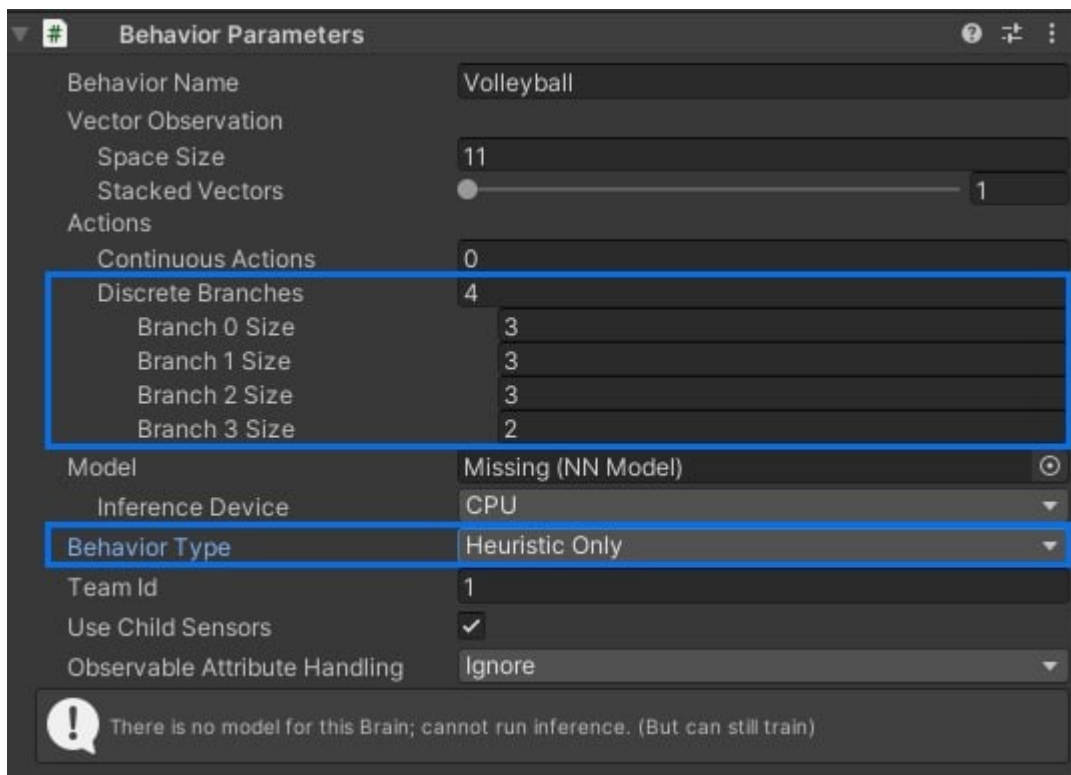
}
if (Input.GetKey(KeyCode.LeftArrow))
{
    // move left
    discreteActionsOut[2] = 1;
}
if (Input.GetKey(KeyCode.RightArrow))
{
    // move right
    discreteActionsOut[2] = 2;
}
discreteActionsOut[3] = Input.GetKey(KeyCode.Space) ? 1 : 0;


```

Save your script and return to the Unity editor.

In the Behavior Parameters component of the PurpleAgent:

1. Set Behavior Type to Heuristic Only. This will call the Heuristic() method.
2. Set up the Actions:
 - i. Discrete Branches = 4
 - a. Branch 0 Size = 3 [No movement, move forward, move backward]
 - b. Branch 1 Size = 3 [No movement, move left, move right]
 - c. Branch 2 Size = 3 [No rotation, rotate clockwise, rotate anti-clockwise]
 - d. Branch 4 Size = 2 [No Jump, jump]



Press  in the editor and you'll be able to use the arrow keys (or WASD) and space bar to control your agent!

Note: It might be easier to playtest if you comment out the `EndEpisode()` calls in `ResolveEvent()` of `VolleyballEnvController.cs` to stop the episode resetting.

Observations

Observations are how our agent 'sees' its environment.

In ML-Agents, there are 3 types of observations we can use:

- **Vectors** — "direct" information about our environment (e.g. a list of floats containing the position, scale, velocity, etc of objects)
- **Raycasts** — "beams" that shoot out from the agent and detect nearby objects
- **Visual/camera input**

In this project, we'll implement **vector observations** to keep things simple. **The goal is to include only the observations that are relevant for making an informed decision about how to act.**

With some trial and error, here's what I decided to use for observations:

- Agent's y-rotation [1 float]
- Agent's x,y,z-velocity [3 floats]
- Agent's x,y,z-normalized vector to the ball (i.e. direction to the ball) [3 floats]
- Ball's x,y,z-velocity [3 floats]

This is a total of **11 vector observations**. Feel free to experiment with different observations. For example, you might've noticed that the agent knows nothing about its opponent. This ends up working fine for training a simple agent that can bounce the ball over the net, but won't be great at training a competitive agent that wants to win.

Also note that selecting observations depends on your goal. If you're trying to replicate a 'real world' scenario, these observations won't make sense. It would be very unlikely for a player to 'know' these direct values about the environment .

To add observations, you'll need to implement the Agent

class CollectObservations() method:

```
public override void CollectObservations(VectorSensor sensor)
{
    // Agent rotation (1 float)
    sensor.AddObservation(this.transform.rotation.y);

    // Vector from agent to ball (direction to ball) (3 floats)
    Vector3 toBall = new Vector3((ballRb.transform.position.x - this.transform.position.x)*agentRot,
    (ballRb.transform.position.y - this.transform.position.y),
    (ballRb.transform.position.z - this.transform.position.z)*agentRot);

    sensor.AddObservation(toBall.normalized);

    // Distance from the ball (1 float)
    sensor.AddObservation(toBall.magnitude);

    // Agent velocity (3 floats)
    sensor.AddObservation(agentRb.velocity);

    // Ball velocity (3 floats)
```



```
sensor.AddObservation(ballRb.velocity.y);  
sensor.AddObservation(ballRb.velocity.z*agentRot);  
sensor.AddObservation(ballRb.velocity.x*agentRot);  
}
```

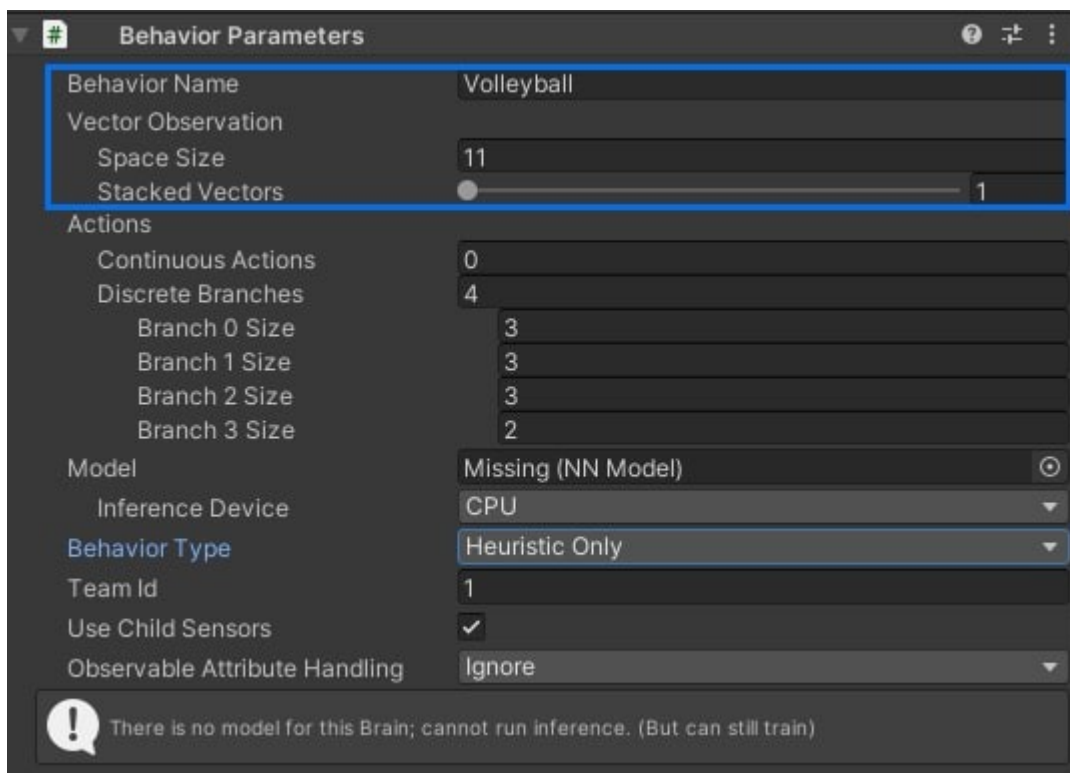
Now we'll finish setting up the Behavior Parameters:

1. Set **Behavior Name** to 'Volleyball'. Later, this is how our trainer will know which agent to train.

2. Set Vector Observation:

i. Space Size: 11

ii. Stacked Vectors: 1



Wrap-up

You're now set up to train your agents using reinforcement learning.

If you get stuck, check out the pre-configured BlueAgent, or see the full source code in the [Ultimate Volleyball project repo](#).

In the next section, we'll train our agents using [PPO](#) — a state of the art RL algorithm provided out-of-the-box by Unity ML-Agents.

[Part 4: Training agents using PPO](#)