How to use DRL Agents framework

(v6.3.0)

DRL Agents (available on GitHub) is an open-source project that comes as an aid for game developers that want to introduce artificial intelligence for their bots/NPCs without calling for a procedural behavior implementation. We provide this goal by using deep reinforcement learning applied in a easy-to-work-with manner, exempting the user/developer from knowing machine learning domain. This documentation provides a Step-by-Step guide, as well as some thoroughgoing study of the tool specific parts that might also give a perspective of how the tool works itself.

Table of Contents

I. **SHORT TUTORIAL** **1**

II. **STEP-BY-STEP TUTORIAL** **1-5**

III. **AGENT IN DEPTH** **6-7**

IV. **TRAINER IN DEPTH** **7-8**

V. **INTERACTION TYPES** **9-10**

VI. **MANUAL CONTROL** **10**

VII. **SELF CONTROL** **10**

VIII. **HEURISTICS IN DEPTH** **10-11**

IX. **CHOOSING THE NEURAL NETWORK LAYOUT** **12**

X. **USING RAY SENSOR** **13**

XI. **IMPROVEMENT TIPS** **14**

❖ I. SHORT TUTORIAL

- Create the **AI** and add ***Agent.cs*** component. **Override** the Script. Create a new **AI Layer** that doesn't interact with itself and apply to the agent model. **Run** the simulation, **set a brain** and press **SaveBrain** (check the net in *Neural_Networks* folder), then **stop**.
- Create a **Trainer** GameObject and add ***Trainer.cs*** component. [Override the Script]. **Add the Agent Model**. **Add the Network Model**. [Choose the **training environment Type** and add the ***Environment* tag** to it.] [Add *TMP_Text* and *Rect* for statistics.] **Setup the training** settings. **Run the training**, the brain model is overwritten.
- Place the trained network to the Agent and set behavior to **Self**.

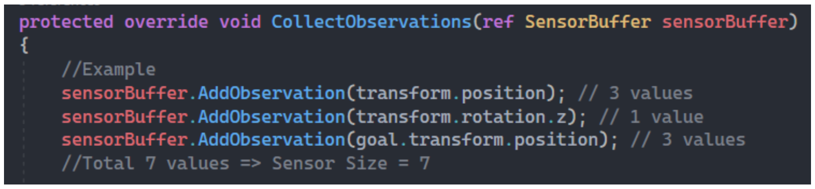
❖ II. STEP-BY-STEP TUTORIAL

This step-by-step tutorial helps simulating the Reinforcement Learning process. For Heuristic Training, check Heuristics in depth.

- ➢ Installation:
  - ✓ Download the .zip file from GitHub or import it from the Assets Store.
  - ✓ If you chose the first method, select the folder with the latest stable version. In the selected folder, you have access to three C# scripts. Upload all in your Scripts folder inside your Unity project, or
  - ✓ From Unity Editor, select **Assets** -> **Import Package -> Custom Package** then choose the package from the zip file.
  - ✓ Create the objects from the following steps:
- ➢ Your own AI agent [prefab] [+**specific layer**] and add *Agent.cs* as component.
- ➢ One empty GameObject [called Trainer] and *Trainer.cs* add as component.
- ➢ [Optional but recommended for training performance] One or more empty GameObjects. Add for each one ***Environment*** tag. [When dealing with more environments, insert in each environment a copy of the agent**,** this way the trainer will now know how to reset agents positions for each different environment]
- ➢ [Optional] A canvas with RenderMode on Screen Space – Camera (and drag your main camera in Render Camera), followed by the following objects:
  - One *TMP_Text* (used for real time statistics)
  - One *RectTransform* (used for evolution performance graph and neural net visualization)
- ➢ Override *Agent.cs* Script:
  - ▪ Override **Heuristic()** and **OnActionReceived()**,and set behavior to *Manual* in order to test your AI behavior by keyboard. Override **CollectObservations()** and set Behavior to Heuristic to train your AI by learning from your actions.

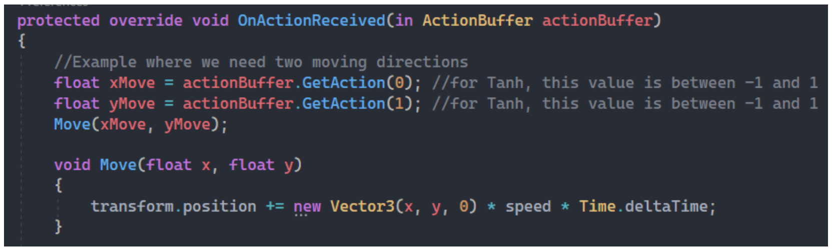
    Tip: Always keep agent Behavior to Static. When training, the behavior is auto set to Self.

- Decide your AI's observations number. Override **CollectObservations()** by fullfiling *SensorBuffer* argument with specific data. Use *AddObservation()* method to add different kind of observations. *Note: every observation might have different size depending on how many float values are inside them. *Example: You decided to have 14 input values, you can add a Transform observation (where observation size is 10, 3 for position, 3 for localScale, 4 for rotation) ,a Vector3 observation (where observation size is 3, 1 for x, 1 for y and 1 for z) ,and an int value observation (where observation size is 1) . 10 + 3 + 1 = 14 input values. Do not let any gaps or inputs empty.*



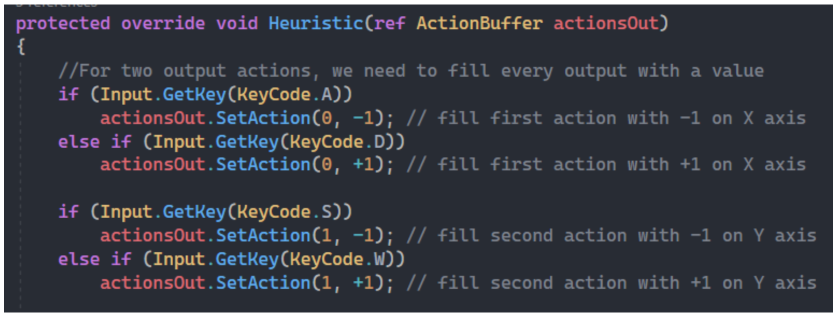
```
protected override void CollectObservations(ref SensorBuffer sensorBuffer)
{
    //Example
    sensorBuffer.AddObservation(transform.position); // 3 values
    sensorBuffer.AddObservation(transform.rotation.z); // 1 value
    sensorBuffer.AddObservation(goal.transform.position); // 3 values
    //Total 7 values => Sensor Size = 7
```

- Decide your AI's actions. Override **OnActionReceived()** by assigning actions depending on values received from *ActionBuffer*. You can access each action individually by using ***GetAction()*** method and specify the index of the action. (outputs are in a range depending on the output activation function, usually (-1,1) if you use Tanh, same as above, this method is called in Update(), use Time.deltaTime if needed) Example:



```
protected override void OnActionReceived(in ActionBuffer actionBuffer)
{
    //Example where we need two moving directions
    float xMove = actionBuffer.GetAction(0); //for Tanh, this value is between -1 and 1
    float yMove = actionBuffer.GetAction(1); //for Tanh, this value is between -1 and 1
    Move(xMove, yMove);

    void Move(float x, float y)
    {
        transform.position += new Vector3(x, y, 0) * speed * Time.deltaTime;
    }
```

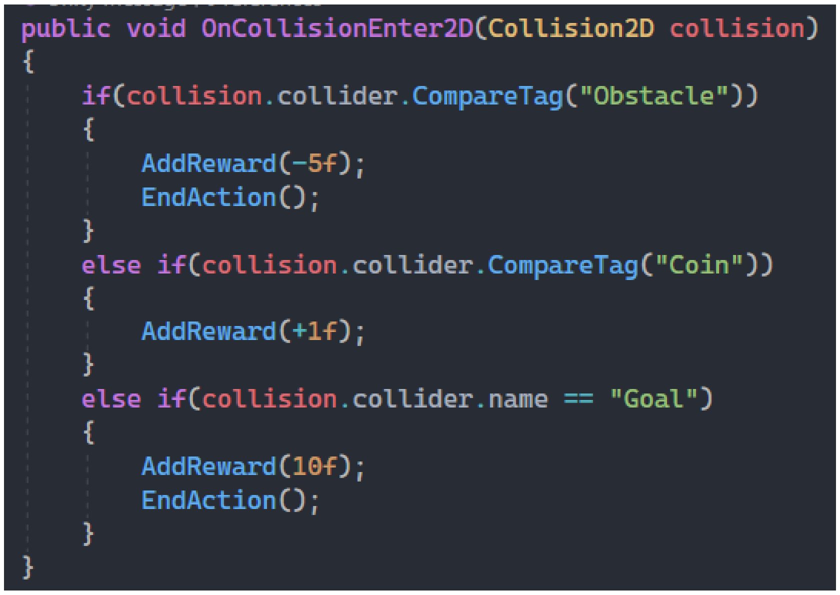
- Fullfill every AI's action with user input in Heuristic() method. Foreach actionBuffer value, set its action by using SetAction() method, specifying the index of the action and the value to set. Set all actions, no more, no less. Make sure your output fits the output activation (usually, if is Tanh, set values between -1 and 1). Example:



```
protected override void Heuristic(ref ActionBuffer actionsOut)
{
    //For two output actions, we need to fill every output with a value
    if (Input.GetKey(KeyCode.A))
        actionsOut.SetAction(0, -1); // fill first action with -1 on X axis
    else if (Input.GetKey(KeyCode.D))
        actionsOut.SetAction(0, +1); // fill first action with +1 on X axis

    if (Input.GetKey(KeyCode.S))
        actionsOut.SetAction(1, -1); // fill second action with -1 on Y axis
    else if (Input.GetKey(KeyCode.W))
        actionsOut.SetAction(1, +1); // fill second action with +1 on Y axis
```

- Create a Rewarding System. Use ***AddReward()*** or ***SetReward()*** to deprive or grant your agents performance.*TIP: divide the reward by *episodesPerEvolution* to normalize the fitness (in case you want to train your agents on different environments/on more episodes, the average fitness will be counted this way).

Use ***EndAction()*** to stop your agents from doing action. (use these methods in OnCollisionEnter/OnTriggerEnter when your AI touches a goal or a trap)



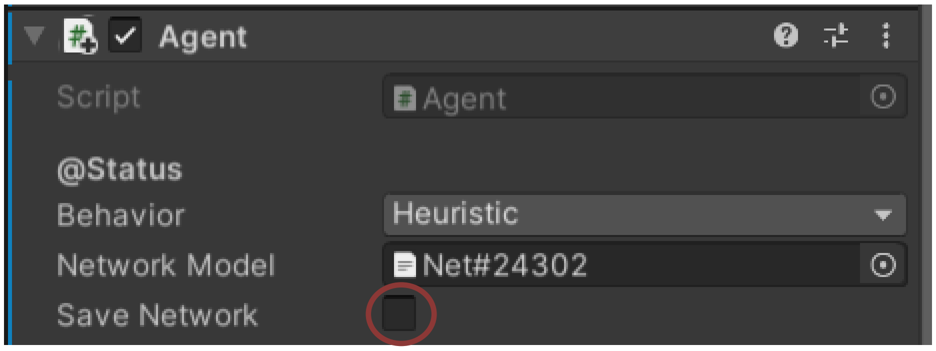
```
public void OnCollisionEnter2D(Collision2D collision)
{
    if(collision.collider.CompareTag("Obstacle"))
    {
        AddReward(-5f);
        EndAction();
    }
    else if(collision.collider.CompareTag("Coin"))
    {
        AddReward(+1f);
    }
    else if(collision.collider.name == "Goal")
    {
        AddReward(10f);
        EndAction();
    }
}
```

Example of adding reward and ending agent's action.

- Use or create variables like speed, rb, etc. under ***===AI Properties===*** header.
  Note: Update and Awake are used in *AgentBase*. Update and Awake are virtual, if you need to use them, override and call their bases.

➢ Create a brain model:

- Go back in UnityEditor and **Play**. Select your agent and take a look on *Agent (Script)* Component. Set ***Network Properties*** (check **IX. CHOOSING THE NEURAL NETWORK LAYOUT**)



Agent

Script: Agent

@Status

Behavior: Heuristic

Network Model: Net#24302

Save Network: ☐

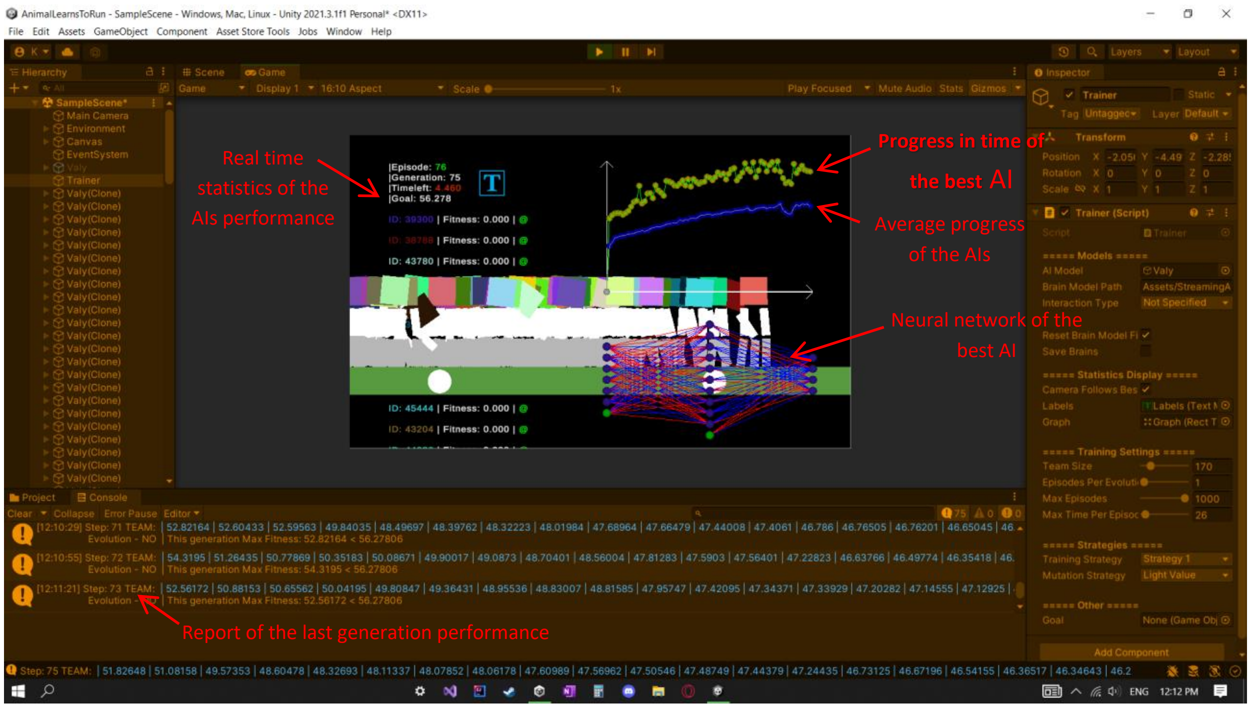
Neural Network settings, according to our previous examples (plus 1 extra hidden layer with 10 neurons)

- Press ***Save Brain*** checkbox once (don't worry if it doesn't modify to check sign, it works like a button) and press **Stop**.
- Look in *Assets\ Neural_Networks* folder. There was created a .txt file with a brain model assigned with randomized weights.

➢ Set your Trainer (for Reinforcement training):

- Drag and drop your prefab (or agentInHierarchy) in ***Agent Model***.
- Drag and drop network file in ***Network Model***.
- [Optional] Specify an Environment Type. The type is used to reset the scene objects and agent's position/rotation/scale after each Episode.

    - [Optional] Drag and drop TMP GameObject from Canvas in *Labels*. (don't forget to adjust the text area – best on left half of the screen)
    - [Optional] Drag and drop RectTransform [from Canvas] in *Graph*. (don't forget to adjust the rectangle area and turn on Gizmos in Scene editor to watch the graph. The Neural Network represents the brain of the best AI in the scene, with the neurons colored in its color, biases in green, positive weights in blue and negative ones in red) *adjust Canvas plane Distance accordingly when in a 3D project
    - Set === *Training Settings* === at your preference. In the beginning, let Training Strategies on their default states.
- [Optional] Override Trainer.cs Script:
    - Add Environment movement by overriding *EnvironmentAction()*
    - Override *OnEpisodeBegin()* if needed.
    - Override *OnEpisodeEnd()* if needed. (this method is called for each individual AI)
    - ***Awake()***, ***Start()*** and ***Update()*** are virtual**,** call **base** if you need to use them.
- Run the simulation:
    - Since is a mono-environmental training, create a new layer that doesn't intersect with itself and assign it to your AI (Edit -> Project Settings -> Physics[2D] -> Layer Collision Matrix) The AI's will start training from their model starting position.
    - Depending of what kind of training you want to simulate, turn on or off the Trainer. *See at chapter VIII how to process **Heuristic Training**.*
- Press **Play**.
- Check console to see the results of each generation.
- Enter in SceneEditor with Gizmos ON to watch the graph.
- Best AI's brain is always overwritten over the .txt file from Trainer ***Network Model***.
- Use the NeuralNetwork post-training:
    - In **Agent.cs**, delete any [*AddReward*(), *SetReward*() or] *EndAction*() calls.
    - Drag and drop network in ***Network Model*** (from === *Network Properties* ===).
    - Control your AI by changing its public Behavior variable between Static or Self.


Real time statistics of the AIs performance
Progress in time of the best AI
Average progress of the AIs
Neural network of the best AI
Report of the last generation performance

```
|Episode: 76
|Generation: 75
|Timeleft: 4.460
|Goal: 56.278
ID: 39300 | Fitness: 0.000 |
ID: 38768 | Fitness: 0.000 |
ID: 43780 | Fitness: 0.000 |
ID: 45444 | Fitness: 0.000 |
ID: 43204 | Fitness: 0.000 |
```

```
[12:10:29] Step: 71 TEAM: | 52.82164 | 52.60433 | 52.59563 | 49.84035 | 48.49697 | 48.39762 | 48.32223 | 48.01984 | 47.68964 | 47.66479 | 47.44008 | 47.4061 | 46.786 | 46.76505 | 46.76201 | 46.65045 | 46.
           Evolution - NO | This generation Max Fitness: 52.82164 < 56.27806
[12:10:55] Step: 72 TEAM: | 54.3195 | 51.26435 | 50.77869 | 50.35183 | 50.08671 | 49.90017 | 49.0873 | 48.70401 | 48.56004 | 47.81283 | 47.5903 | 47.56401 | 47.22823 | 46.63766 | 46.49774 | 46.35418 | 46.
           Evolution - NO | This generation Max Fitness: 54.3195 < 56.27806
[12:11:21] Step: 73 TEAM: | 52.56172 | 50.88153 | 50.65562 | 50.04195 | 49.80847 | 49.36431 | 48.95536 | 48.83007 | 48.81585 | 47.95747 | 47.42095 | 47.34371 | 47.33929 | 47.20282 | 47.14555 | 47.12925 |
           Evolution - NO | This generation Max Fitness: 52.56172 < 56.27806
Step: 75 TEAM: | 51.82648 | 51.08158 | 49.57353 | 48.60478 | 48.32693 | 48.11337 | 48.07852 | 48.06178 | 47.60989 | 47.56962 | 47.50546 | 47.48749 | 47.44379 | 47.24435 | 46.73125 | 46.67196 | 46.54155 | 46.36517 | 46.34643 | 46.2
```

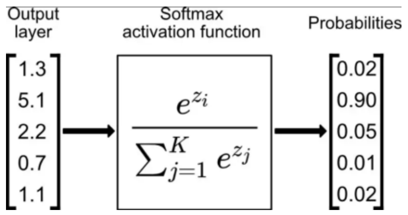
```
Trainer (Script)
===== Models =====
AI Model               Valy
Brain Model Path       Assets/StreamingA
Interaction Type       Not Specified
Reset Brain Model Fi   ✓
Save Brains
===== Statistics Display =====
Camera Follows Bes     ✓
Labels                 Labels (Text M
Graph                  Graph (Rect T
===== Training Settings =====
Team Size              170
Episodes Per Evoluti   1
Max Episodes           1000
Max Time Per Episoc    26
===== Strategies =====
Training Strategy      Strategy 1
Mutation Strategy      Light Value
===== Other =====
Goal                   None (Game Obj
```

GUI example of a Reinforcement training session

III. AGENT IN DEPTH

- **Behavior:** there are 4 types of behavior.
  - **Static:** the AI remains stationary, with no control by neural network or by keyboard.
  - **Manual:** the AI is controller by user. See MANUAL CONTROL.
  - **Self:** the AI is controlled by its own neural network.
  - **Heuristic:** the AI is trained heuristically. See HEURISTICS IN DEPTH.
- **Network Model:** represents the file of the Neural Network. Drag and drop **from Assets** folder the file here (the file is created in StreamingAssets and must be placed in Assets).
- **Save Network:** it [creates or] saves a neural network [in *StreamingAssets/Neural_Networks].*
- **Sensor Size:** the number of inputs/observations.
- **Action Size:** the number of outputs /actions.
- **Hidden Layers:** the number of hidden layers. Each element is a layer. Each element value represents the number of neurons on that hidden layer.
- **[Output] Activation Type:** Function used for neuron activation. *[Output activation function will affect your output values range.]* Tanh returns values in range (-1,1), BinaryStep returns binary values, Sigmoid returns values in range (0,1), ReLU returns

values In range [0, +infinity), SiLU returns same values as ReLU but in a smooth manner and SoftMax outputs a probabilistic result:


Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

- **Initialization Type:** The switch will not make a significant change, is more like a flavor for weights and biases initialization.
- **Heuristic Properties:** check HEURISTICS IN DEPTH chapter.

- **Methods to override:**
  - **CollectObservations(SensorBuffer) ->** AddObservation(observation)
  - **OnActionReceived(ActionBuffer) ->** GetAction(index)
  - **Heuristic(ActionBuffer) ->** SetAction(index, actionValue)
  - HeuristicOnSceneReset() **->** called after EndAction() for Manual/Heuristic behavior.
- **Methods to use:**
  - **AddReward(reward, [bool value]) ->** if the boolean value is true, the reward is added even if the agent's action has ended. (usually used in OnEpisodeEnd())
  - **SetReward(reward)**
  - **EndAction() ->** sets agent's behavior to static until reset the episode.

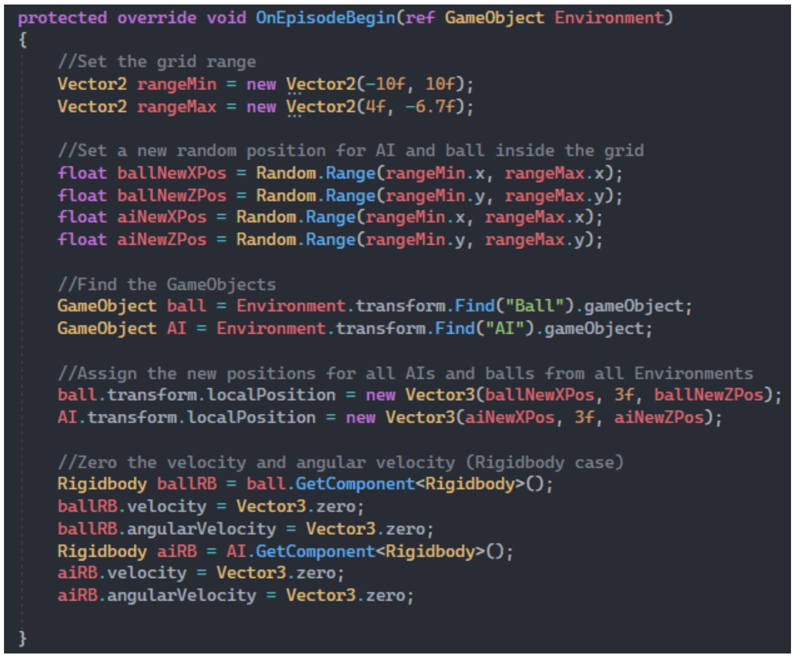
❖ IV. TRAINER IN DEPTH

- **Agent Model:** drag and drop the AI gameobject.
- **Network Model**: Drag and drop the network file here (same as for Agent).
- **Interaction Type:** Choose the training environment option (see INTERACTION TYPES)
- **Reset Brain Model Fitness:** When a new training session begins, the fitness from the network file is not taken in consideration, and the network starts with 0 fitness (the training does not continue from the fitness found in the file). In this situation, the neural network can be affected from the first episode*.
- **Save Networks:** When you see an AI that has good behavior and doesn't manage to get too much fitness, *Save Brains* can be used to save the brains of the best AIs. Then, stop the training and use his brain for another training session. The saves are placed inside of a newly created directory (indexed randomly), situated in **Saves** folder.
- **Camera Follows Best AI:** implicitly finds the first object with Camera component. It follows he AI with the best real-time fitness. **It disables all other custom components* (like custom camera follow scripts).**

- **Labels and Graph:** Drag and drop a TextMeshPro canvas object and a normal Canvas empty object to view the real-time statistics.
- **Team Size:** Use as much AI's as possible while keeping the frame rate stable (around 60)
- **Episodes per Evolution:** Let more episodes to train for one generation. Rewarding is cumulative. The next generation occurs every ***x*** episodes.
- **Max Episodes:** represents the length of the current training session.
- **Episode Length:** Give a limited time to your AI's per Episode. **Because some AI's might never end their action the episode will run forever.*

- **Training Strategy:** At the beginning of training, start with *Strategy 1*. When you see an AI that is quite good since his behavior is close to what you expect and he managed a good fitness, switch to *Strategy 2* (this way the best brain will be inserted in 1/3 of the AI's and mutated every Episode). If your AI is ready you can go for a training with Strategy 3, where only the best brain is reproduced, this might be good to find a better AI with the same behavior.
- **Mutation Strategy:** Use Classic *MutationStrategy* mostly. You might switch to Light/Strong Percentage in combination with *Strategy 3* to fine-tune your agent abilities.

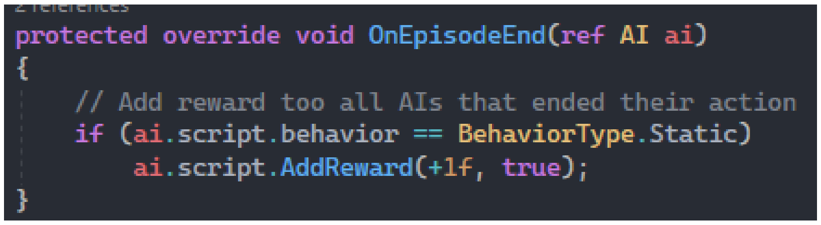
- **Methods to override:**
  - **OnEpisodeBegin(ref GameObject Environment) -> called at the beginning of every episode (called even if interaction type set to *Not Specified*, but ref parameter is not used). *Called in the 1st episode too*.***



```
protected override void OnEpisodeBegin(ref GameObject Environment)
{
    //Set the grid range
    Vector2 rangeMin = new Vector2(-10f, 10f);
    Vector2 rangeMax = new Vector2(4f, -6.7f);

    //Set a new random position for AI and ball inside the grid
    float ballNewXPos = Random.Range(rangeMin.x, rangeMax.x);
    float ballNewZPos = Random.Range(rangeMin.y, rangeMax.y);
    float aiNewXPos = Random.Range(rangeMin.x, rangeMax.x);
    float aiNewZPos = Random.Range(rangeMin.y, rangeMax.y);

    //Find the GameObjects
    GameObject ball = Environment.transform.Find("Ball").gameObject;
    GameObject AI = Environment.transform.Find("AI").gameObject;

    //Assign the new positions for all AIs and balls from all Environments
    ball.transform.localPosition = new Vector3(ballNewXPos, 3f, ballNewZPos);
    AI.transform.localPosition = new Vector3(aiNewXPos, 3f, aiNewZPos);

    //Zero the velocity and angular velocity (Rigidbody case)
    Rigidbody ballRB = ball.GetComponent<Rigidbody>();
    ballRB.velocity = Vector3.zero;
    ballRB.angularVelocity = Vector3.zero;
    Rigidbody aiRB = AI.GetComponent<Rigidbody>();
    aiRB.velocity = Vector3.zero;
    aiRB.angularVelocity = Vector3.zero;

}
```

Example of overriding OnEpisodeBegin() method, where the agent and a ball are placed randomly within the defined ranges.

- **OnEpisodeEnd(AI ai) -> called immediately after episode reset for each agent**



```
protected override void OnEpisodeEnd(ref AI ai)
{
    // Add reward too all AIs that ended their action
    if (ai.script.behavior == BehaviorType.Static)
        ai.script.AddReward(+1f, true);
}
```

Example of overriding OnEpisodeEnd() method, where the agents that finished their action before episode ends gets 1 reward

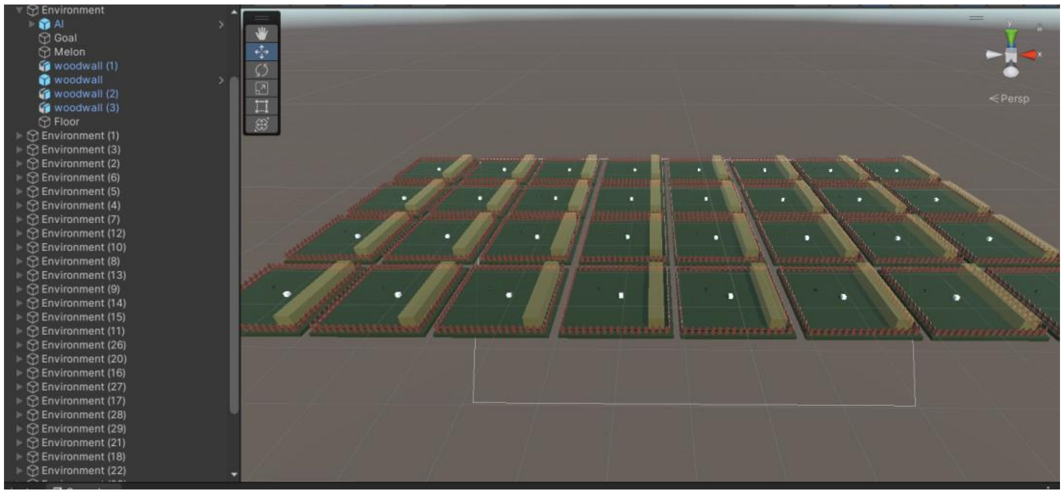
AI struct has the following fields:

- **agent**: access agent GameObject
- **script**: access agent's Script
- **fitness**: access agent's current fitness (read-only, modifications do not affect)

❖ V. INTERACTION TYPES

- ✓ Use *Not Specified* if <u>not needed</u>.
- ✓ Add tag **Environment** for all your environments you want to use (at least 1).
- ✓ Add a copy of your agent in each environment. Adjust their positions and rotations (even of its children) at your preference. You can rename the copies as you wish to avoid ambiguities, but is not necessary.

- **More Agents Per Environment**: In this situation, the agents are trained together in the same environment by overlapping each other. When there is more than 1 environment, is necessary for each of them to have a model of the agent inside (representing the starting position), otherwise the main model is used as a start.
- **One Agent Per Environment**: This is mainly used when you need to let just 1 agent to interact with the environment. To properly use this, take your normal environment and clone it for several times (**with the agent inside**), positioning them apart from each other. (team size is adjusted automatically to the number of environment clones). Every object from all environments will remain with references to the agent from the same environment. Trainer still requires the Agent Model. See the following example:



Original Environment has all objects inside including the Agent. It was cloned several times, and all Environments all placed next to each other, such way they do not intersect. TIP: when cloning the environments, also clone the clones, this way you double them each cloning.

- ***Whenever you don't want to use an environment, remove its Tag or make it inactive.**

❖ VI. MANUAL CONTROL

- ***It REQUIRES a neural network to be instantiated before use.***
- **Heuristic()** and **OnActionReceived()** methods must be overridden in order to control the AI.
- Call **EndAction()** to reset the AI's position.
- Position reset applies when this behavior is on.

❖ VII. SELF CONTROL

- ***It REQUIRES a neural network to be instantiated before use.***
- **CollectObservations()** and **OnActionReceived()** methods must be overridden in order for the AI to self-control.
- **EndAction()** calls from the Agent script should be removed when this behavior is used in its final state and is used in a project.
- Position reset does **not** apply when this behavior is on. (excepting when the Trainer is used)

❖ VIII. HEURISTICS IN DEPTH

- ✓ Load a network to the Agent. Turn OFF the *Trainer (if exists).* Set behavior to Heuristic. Set *Module* to **Collect** (it will write training data in the ***DataSet#*** file; if the file doesn't exist, it will create one in *Neural_Networks*).
- ✓ Start your simulation and control your agent consistently by doing it's job. **Do not stop** the simulation until it ends, otherwise data collecting will collapse.
- ✓ After enough collected data, go to the newly created Directory *Heuristic_Samples* and and move the file found there in Assets, then drag and drop the file in **Training Data File** field. Set the Module to **Learn**.
- ✓ Run the simulation again, this time deep learning is processed using the data inside the file. Do not stop until it finishes. If you really need to stop the simulation, set the Epochs number to 0 using the slider).
- **Training Data File** the file where training data is stored or used. It is auto created if doesn't exist when collecting.
- **Collect Passive Actions** button is used to collect training samples where the user didn't introduced inputs *(all elements in the action vector are equal to 0).* Turning it off heads for a more optimized training (samples number is reduced). If the AI needs to "wait" (it means no actions) at some point, this feature must be turned ON.

- **Mode** represents the interaction mode. If is set to **Collect**, the user gives training data through the keyboard or mouse, data is stored in a *txt* file in ***Neural_Networks*** directory. If is set to **Learn**, the machine reads the file loaded and trains.
- **Epochs** are the number of times the data is reprocessed. A high number of Epochs may cause overfitting, a low number may cause underfitting. ********Reducing the number of epochs while training is the only way to stop safely the simulation.***
- **Session Length** represents the time you spent on giving data to the training file. Longer the session, more divert the data will be.
- **Current Error** shows the error between the network outputs and user outputs. The heuristic training process must decrease this value as much as possible. When the *Current Error* is approaching to 0, you can end the Heuristic Training and pursue with the Trainer.
- **Error Graph** shows the progression of the error. To get the best results for your agent, watch the graph carefully. When the error reaches a *plateau* you can stop the learning session by moving the epochs slider to 0.
- **Environment** works in the same way, but it doesn't need a specific tag, just drag and drop your Environment gameObject here to reset the environment every time the agent action ends. Supports only *mono-environment*.
- Override **HeuristicOnSceneReset()** from *Agent* script to modify the environment or agent position differently for each Scene (*get reference through the agent script to the individual objects or to the Environment gameObject*).

- **Learn Rate** can be modified depending on the learn impact for your agent. Keeping it too low may not cause major impact on the network itself, keeping it to high doesn't show any beneficial results, and can even shatter the old parameters performance. You can modify this value at runtime.
- **Momentum** and **Regularization** are advanced hyper-parameters. [Do not modify them]
- **Loss Function** is the function used to calculate the total error of the outputs comparing to the user controls (desired outputs).
  - ***Cross Entropy*** is used **only** for SoftMax outputActivation function.
  - When **Cross Entropy**, training data *ActionBuffer* values **must contain only one action with value 1 and other actions value 0**.

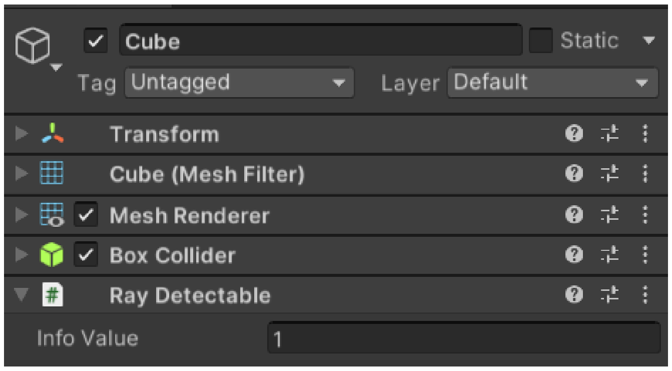
    (Cross Entropy is used for images object recognition, and theoretically only 1 out of all must be true)
- **Batching** is a float value in range **(0,1]** that indicates how the data set is divided in mini-batches. If set on 1, the mini-batch consists of all samples, so is equivalent with ***Full Batch*** mode, otherwise is **Mini Batching**; if is set on 0.1, there will be 10 mini-batches and so on. Cannot be modified at runtime.

❖ IX. CHOOSING THE NEURAL NETWORK LAYOUT

- **Space size** represents the number of agent's observations (or the ***input layer*** neurons). More observations number require more training data for better performance (to cover as many observation situations as possible).
- **Action size** represents the number of agent's actions (or the ***output layer*** neurons). There can be any number of actions, and is recommended to use the negative value (if using *Tanh*) to compress the number of neurons. For instance, instead of using 2 neurons for moving left or right (by choosing the highest value), you can use only 1 neuron; if the *value < 0* move left, else move right.
- **Hidden layers** represent layers of neurons between input and output layer. One element represents a layer, element's value is the number of neurons.
    - For **Reinforcement Learning**, the training is done by randomly mutating the model over time. So, larger the network, longer it will take until it finds a decent weights and biases configuration. It is recommended to use **0** or **1** hidden layers. In case you opt for the last, set its neurons number around the sum of *input layer neurons* + *output layer neurons.* (Examples: [4 | 6 | 2], [8 | 12 | 4], [7 | 14 | 7])
    - For **Heuristic Learning**, the layout is chosen depending on your preference. You can use any number of hidden layers (*usually all of them having more or equal neurons than the output layer*). The difference comes due to the training target. If you need fast training with mediocre performance you can go for a small number of hidden neurons. If you look for good performance, you should use more hidden neurons, but also expect a longer training time (also a higher chance of *overfitting*). This happens due to network's limitations. Over training time, the network will reach an error plateau, where the error does no longer decrease. Larger neural networks allow the error to decrease more at the cost of longer learning time.
(Examples: This network [20 | 15 | 10 | 2] managed to have a minimum error of ***0.45***, the other network [20 | 30 | 15 | 2] managed a minimum error of ***0.38***; both trained by the same training data; first network training x**2** times faster than the second)
- For *Compound Learning* (both learning methods applied on the same agent), follow the *Reinforcement Learning* recommendation.

❖ RAY SENSOR

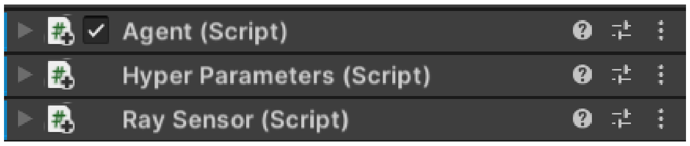
- **RaySensor** is a special component used to cast multiple rays in any direction wanted by adding it to the agent gameObject. Working both in 3D and 2D, it provides a simple to adjust interface in order to produce observations to the agent about the objects in the environment.
- Info means what type of information should the agent receive from each ray. If you don't want the ray distance, you can set it to value. In this situation, special object in the environment are detected by the rays. These objects must contain the component **Ray Detectable** in order to be observed, and every object might have a different info value. To get both the distance and value, use 2 identical RaySensors on the same agent.



Cube
Static
Tag Untagged
Layer Default
Transform
Cube (Mesh Filter)
Mesh Renderer
Box Collider
Ray Detectable
Info Value 1

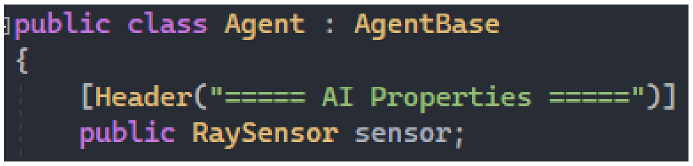
(Cube triggers the rays -> ray value is *Info Value*, other objects that do not have this component -> ray value is 0)

- The number of **rays** represent the number of observations feed into the neural network. RaySensor observations can be added inside ***SensorBuffer*** using the method AddObservation(), inserting either the RaySensor object as a parameter, or **RaySensor**.**observations** array[].



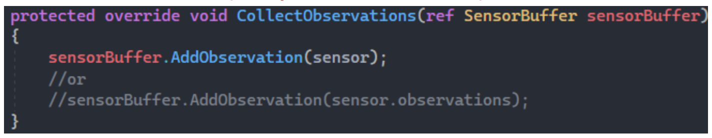
Agent (Script)
Hyper Parameters (Script)
Ray Sensor (Script)

(RaySensor is added along Agent script)



```
public class Agent : AgentBase
{
    [Header("===== AI Properties =====")]
    public RaySensor sensor;
```

(a script reference is needed)



```
protected override void CollectObservations(ref SensorBuffer sensorBuffer)
{
    sensorBuffer.AddObservation(sensor);
    //or
    //sensorBuffer.AddObservation(sensor.observations);
}
```

(can be added in SensorBuffer in 2 ways, make sure agent Action Size >= raysNumber)

❖ IMPROVEMENTS TIPS

- Don’t feed to the agent inconsequential observations. For examples, if you are in a 2D scene, do not add **transform.position** as an observation; instead add **transform.position.x** and **transform.position.y** separately (z position is irrelevant).
- Use relative positions to objects to reduce input neurons and generalize the situations. For example, instead of adding **agent.transform.position** then **goal.transform.position**, add the difference between the two: agent.**transform.position – goal.transform.position**. This reduce the observations from 6 down to 4.
- Use both positive and negative values for **Ray Detectable.**
- Try with different neural networks layouts in order to find the best layout that suits your agent. Usually the neurons number makes the difference, too few neurons do not let agents to perform relative complex tasks, and too many neurons in a neural network makes is harder to train and cause the error to not drop sufficiently.