Simple Q - *QLearning Framework*

Simple Q is a free modular framework that uses the QLearning Algorithm Process to allow you to build and train AI agents within the Unity App, without the need for additional assets.

Simple Q's reinforcement learning framework integrates advanced techniques such as prioritized learning and experience replay to accelerate the training process and improve learning efficiency.

Trained data is saved with a simple string set on the component (in the Inspector), which allows multi agents (or NPC's) to use the same learned data by accessing the same saved path. This enables continuous experimentation and iteration, for developers to freely create trained AI agents for Games/Apps.

*To access the necessary classes for building your own agent, use the "QLearning" namespace.

Summarisation

Simple Q works as is, out of the box, and can be used to train AI agents right away, with their data saved and accessible for mobile and computer applications.

However, it is also a learning tool, so for the most optimized results and for more advanced bots, it is best built upon - the Prioritized Learning is a perfect example, as it has three ways to prioritize learning (random, state, attribute) but the best way to prioritize the learning is based upon the results the user wants, therefore, these methods can be altered, changed, or added too, dependent on the users desires.

Contents:

The Asset contains examples of different agents and environments (virtual, 2D, 3D). Along with two types of QLearning scripts:

1. **LittleQBrain**, which uses the most basic QLearning process, and is a great learning point for anyone trying to understand ML/AI.
2. **OpenQLearningBrain**, the main framework, which uses a more advanced QLearning process, and far more meta-variables to adjust learning parameters.

And also includes:

- An **AttributesPair** class/script. This is used for the Attribute pairs of the states, and allows us to visualize them in the Inspector (State_And_Actions list).

- The **OpenQLearningSharedData** class/script. Attached to an object, this acts as a hivemind, allowing multiple agents, with multiple instances of the same brain, to access, use, and update a shared data.

The LittleQBrain Class (script) uses the basic process (step 2-5 looping) to learn:

1. Initialize the Brain (QTable) (Call the function)
2. Choose An Action (Call the function)
3. Enact Action (Your code)
4. Reward the Action (Call the function)
5. Update the Brain (QTable) (Call the function)

Due to the addition of an Experience Replay Buffer, we no longer update the QTable immediately after a reward is granted in the OpenQLearningBrain.

Instead, the Replay buffer is updated with the experience and the QTable has the State added if not present. Then after our interaction count has met the amount we set in the Inspector (***choicesBetweenBatchLearning***) we take a batch sample of experiences, and update our QTable according to the batch learning process.

With this, we have a slight change to the usual process, making our last step automated. Meaning, once initialized, the user only need to loop through steps 2-4:

1. Initialize the Brain (QTable) (Call the function)
2. Choose An Action (Call the function)
3. Enact Action (Your code)
4. Reward the Action (Call the function)
5. Update the Brain (QTable) (Called Automatically)

Using Simple Q, outside of the users code for their bots controls (e.g. movement, attacking, animation) we simply call following functions:

1. InitializeQBrain()
2. MakeAChoice()
3. UpdateReward()

The action taken from the choice (the function returns a string action choice), is based upon the user's bot. E.g. An enemy in battle (battle bot), might get the choice to defend, and the user would have it play the defend animation, and go into a defend state (defending bool = true), taking less damage. And of course, they would then get their current state, and update the reward.

When calling the **MakeAChoice()** function, there are two parameters to be passed in: the string state, and the attributes pair list. The attributes pair list is optional. To make it easier, simply create a new StateClass instance, and assign the values to be passed in - this helps to clarify the information being fed to the brain.

When calling the **UpdateReward()** function, there are three parameters to be passed through: the current state, the reward float, and a done bool. The done bool flag is optional. The current state is a StateClass instance, the reward is based upon the users reward logic for different actions/outcomes, and the bool flag informs the Replay Buffer if an action led to the completion of a goal/session.

The QLearning Process in Simple Q:

For the QLearning process, **MakeAChoice()** records our *state before action* and then our *action choice*. When the user then calls **UpdateReward()**, our *state after the action* and *reward for the action*, which is recorded in our Replay Buffer Experiences, and used to update our QTable.

1. MakeAChoice() = state before action, action chosen
2. UpdateReward() = state after action, reward for action choice
(3.) Replay Buffer Updates QTable values periodically

With this, we have the necessary requirements for the QLearning process. And once a brain is sufficiently trained, only the **MakeAChoice()** needs to be called to receive action choices for the different states.

States And Actions:

Simple Q is set up on the basis of States and Actions. The agent is in a state, and it takes the best action available. We have further broken it down into State-Attributes and Actions, however, the agent can run solely on State and Actions, and the Attributes can be left empty.

Attributes have been introduced into Simple Q, to allow for an easier way to access Replay Buffer Experiences when utilizing Prioritised Learning techniques.

E.g. In our dodge bot, we use a bool flag to check whether there is an object to its left and/or right. If it does move there anyway, it is punished. These are placed in the attributes, and, if needed, could be focused upon by accessing sample batches.

Important: A single example state should be given in the **State_And_Actions** list in the Inspector, allowing the bot to understand the states and actions it will be receiving.

Currently, the function **GetCurrentState()** returns the current state, however, if the state is not present in the data, the brain learns the state. In this setup, it simply gets the actions from the example state, and sets that as this new state's actions. This is only optimal if states have the same actions - but can be adjusted to fit any needs in the **GetCurrentState()** function.

Also, for the quicker readers, yes, the fact we are passing in the current state to a function to return the current state is redundant in that sense, but works perfect as a function for ensuring we are using a state in the dictionary (returns the matching state) and as a learning catch for learning new states/experiences.

Learning Variables:

The standard learning variables for an agent are available in the inspector, allowing users to tailor the brain/agent/learning process to their needs. These include:

1. Learning Rate - a value between 0-1, affecting how quickly our agent learns
2. Discount Factor - a value between 0-1, affecting the impact of future rewards on the bots decision making.
3. Replay Batch Sample Size - usually set between 32 - 512. Dependent upon your needs.
4. Choices Between Batch Learning Experience - this value allows us to call the Batch Learning Experience, automatically updating our QTable, after a set amount of interactions have taken place - choices made. An internal counter counts up to this value whenever **MakeAChoice** is called.

These variables can be adjusted to best suit the needs of the agents being trained. If an agent is slowing down the process, consider raising the Choice Between Learning Batch Experience (or removing the automatic save called at the end of the Batch Learning).

Exploration Vs Exploitation:

Simple Q currently uses two different methods for Exploration Vs Exploitation, which can be set by the user in the Inspector:

- Epsilon-Greedy
- Boltzmann Exploration

Both methods have three dynamic decay methods for their respective values (Epsilon, Temperature):

- Annealing Decay (current episode / amount of episodes in a session)
- Episode Decay (current choice(step) / amount of choices(steps) in an episode)
- Experience Fluctuation (the mean of experience values * age / max experience value)

Again, as with Prioritized Learning and the Replay Buffer Removal Policy, these are basic methods for controlling Exploration Vs Exploitation and Dynamic Decay.

These methods are simply called from the **ChooseAnAction()** function, using an if/else statement, allowing users to easily build upon/add/remove these methods.

(**“Experience Fluctuation”** is a new, simple method created for Simple Q, demonstrating how non-linear variable control can affect the Exploration vs Exploitation of a bot, and how different techniques can be used to suit the different needs of different agents.)

Prioritized Learning:

Simple Q uses the Replay Buffer to prioritize learning through specific experiences - we decide which types of experiences we want to sample during the batch learning process.

Currently, Simple Q offers three basic examples of prioritized learning through the Replay Buffer, with the following methods:

1. Random Experiences - this randomly selects a batch of experiences to learn from.
2. State String Experience - this selects experiences that match a specific state.
3. Attributes Pair Experience - this selects experiences that match a specific attribute.

These are basic examples of how an agent learning can be manipulated to prioritize certain experiences over others.

For example, an agent could have more than one layer to its prioritized learning parameters, like a certain state and attribute to match, or a primary priority experience (e.g. state) and a secondary priority experience (e.g. done flag). Or a priority experience (e.g. state) and an exclusion attribute (e.g. got hit), which would only select the specified states if they did not have the exclusion attribute.

Replay Buffer Removal Policy:

Simple Q has four simple examples of Replay Buffer Removal methods, with the variables in the inspector for setting the size of the Replay Buffer and the amount to remove when the function is called.

Due to the FIFO and LIFO examples included in the Framework, a Sort() function is used in the **ReplayBufferExperienceRemoval()** function for each example method, which can cause a noticeable lag when removing. If the user does not intend to use these types of numbered removal examples keep the **experienceRemovalIdSorted** bool **false.**

The currently methods included in the Framework are:

1. Random Removal - Remove a random batch of experiences.
2. FiFO Removal - First in, first out (oldest experiences removed first).
3. LIFO Removal - Last in, first out (newest experiences removed first).
4. Attributes Removal - Remove a batch of experiences with a certain attribute.

Once more, these are working example methods that demonstrate how the user can control what data is removed from the Replay Buffer when freeing up space. Method 4 demonstrates

how a user can further specify the types of experiences they or least in need of (Attributes is an example, as states, rewards, and other parameters could be used).

Persistent Data (Saving and Loading)

Simple Q uses a basic saving and loading implementation. With a string variable set in the Inspector, the same QTable and Replay Buffer data can be accessed, loaded, manipulated, and saved.

The framework uses Streamwriter and Unity's JSON to save and load text files. Loading happens in the brain's initialization stage. Saving can be called by the user with simple functions, or left to the automatic saving of the Experience Replay.

Currently, Simple Q has a separate class for saving, with a new instance created in the main class, named **Save_Manager**. All automatic save points have been removed (commented out) except for the in UpdateQTable() function, which uses batch learning to update, and then saves, using the **Save_Open_QBrain()** function.

A **sortedOutput** bool (default: false) can be set to true before saving, to sort the output and promote better readability. However, this should be used at the end of a session, as calling it constantly throughout can slow the system down considerably (the Sort() function called has a lot of overhead).

This system works out-of-the box, but can be easily adjusted to fit users desires. The data saving has a Sort() function, but this is best set (programmatically) at the end of a session save. For more details on how the saving and loading work, read the *further details* section below.

Shared Data:

Simple Q utilizes concurrent data structures to allow multiple agents to manipulate the same data sources, allowing for the use of multi-threads for dispatching/training agents. However, Simple Q has no multithreading methods coding into the framework, so as to keep the code focused more solely on ML/AI, and the promotion of its creation. But, as stated, it does have the capabilities, so users can still utilize this method if intended.

Agents can still be created (like in the battle bot example) as instances in a class, and train on the same data parallel to each other.

Adding the same brain script (an **OpenQLearningBrain** script with the same save_path directory) to different game objects allows them to use the same learned data when making choices.

However, adding the same script to different objects creates different instances of that same data. This is fine for a trained brain, e.g. given to different NPC's to use in a game setting. But if the agents were training, then the saved data would be made up of the experiences of the last agent to save.

For this reason the class/script **OpenQLearningSharedData** has been included. By adding this script to a hive object (an object the agents can all reference), we can set the bool, **sharedData**, in the Inspector to "true", and add the hive object to the slot available.

Two bools, and a function have been added to the **OpenQLearningSharedData** class, to allow only one agent to load up the data, and save the data manipulated by the masses. (This ensures that multiple agents are not calling saves and loads, creating bottlenecks).

The bools added are: **QTableUpdateCalled** (saved yet) and **dataLoadedYet.** And the function is **UpdateQTableCalledFlag()** which sets the **QTableUpdateCalled** to false are a set amount of time (current 3f), allowing the save to be called again.

Now the script is set up to use a shared data source. Any other agent using the script with the bool set and the same hive object, will use the shared data source also, instead of the data source in their scripts. Allowing them to all share the same experiences (greatly speeding up the process).

The choice of whether the brain uses the local data or the shared data, is set in the **InitializeQBrain()** function. If no longer sharing, simply set the bool to "false" and re-initialize the QBrain (either through exiting Play Mode and turning off the bool in the Inspector, or programmatically setting the bool, and the calling the initialize function again).

Simple Setup

The simple setup takes into consideration that you may have no prior knowledge of how to create an agent and an environment, and will use a simple example to show how simple it is to set up.

This setup will include optional ("*") steps that can make creating the agent/environment easier.

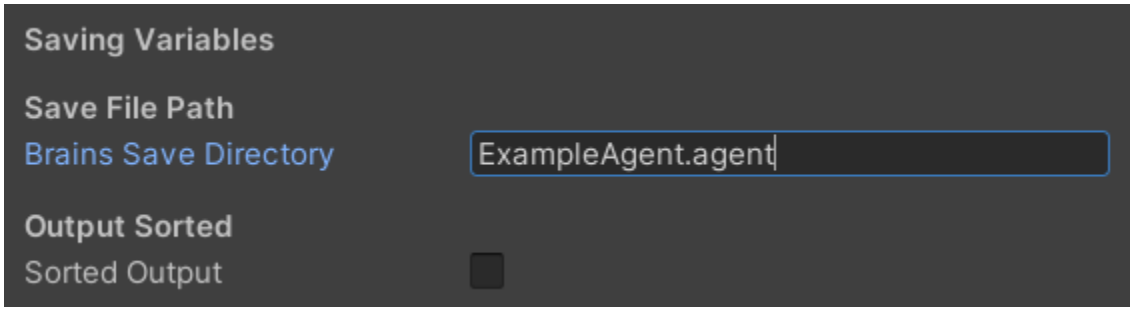
We will be using the example of a battle bot, using only a string state, and no attributes. Our state will be "agent health, enemy health" (e.g: 10,8). With three actions: Attack, Defend, Flee.

This setup will include using Annealing Decay and Episode Decay for Epsilon-Greedy exploration vs exploitation.

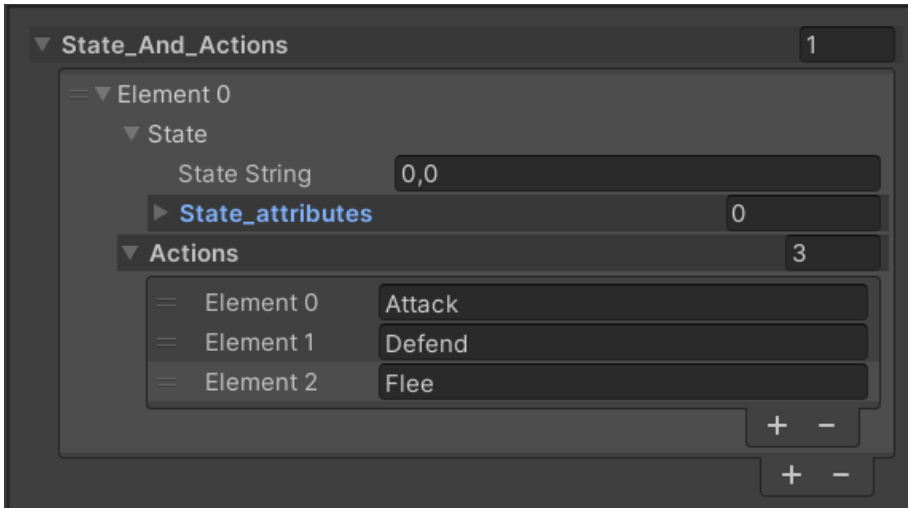
Step 1 - Inspector Setup

1. Create an object (we'll call it "Agent Controller") and add an **OpenQLearningBrain** script component to the object.

2. In the **OpenQLearningBrain** component (in the Inspector) we will set our "Brains Save Directory" to whatever we choose. For this example, I will input: *ExampleAgent.agent*



3. Next, in the same script, in the Inspector, we will add a new item to the "States And Actions" list. This will be our example state.
    - In the "State String" we will input an example of what our states will look like, 0,0.
    - We will leave the attributes empty
    - In the actions list we will add our three actions



4. We will create an “Agent Controller” script for our agent (the same object which has the brain - although this is optional), and open that script.

Step 2 - Simple Script Setup

1. First, to access the classes for building our OpenQLearning brain, we need to reference the namespace for the QLearning:

```
using UnityEngine;
using QLearning;
```

2. Next we will create some variables to hold our values:

    - A public **OpenQLearningBrain** variable
    - *A list of action strings to compare against
    - *Two State_Class variables for our old and new state
    - Two int variables for player and enemy health

```
Unity Script | 0 references
public class AgentController : MonoBehaviour
{
    //Instance of our brain
    public OpenQLearningBrain ourBrainScript;

    //List of our actions - OPTIONAL
    public List<string> actions = new List<string>()
    {
        "Attack",
        "Defend",
        "Flee"
    };

    //State Classes to save us time creating creating new vars every time we use them - OPTIONAL
    private State_Class currentState;
    private State_Class newState;

    //Variables for health
    private int playerHealth;
    private int enemyHealth;
```

3. We will then create two helper functions:

    - **GetOurState()** - we will use this function to return a **State_Class**. Inside the function, we will simply get the playerHealth and the enemyHealth, and create a string from them to assign to the **State_Class** stringState.

```
//Function fore returning our current class
2 references
public State_Class GetOurState()
{
    State_Class state = new State_Class();

    //Update the string state according to how we want the data laid out
    state.stateString = playerHealth + "," + enemyHealth;


    return state;
}
```

    - **RewardForActions() -** this function will return a float value, rewarding/punishing the agent accordingly. For this example, we are simply checking if the agent or enemy are dead (health below 0), but any variables checks could be made - using information before and after a choice - to alter the reward/punishment values.

```
//Function for getting our reward
1 reference
public float RewardForActions()
{
    //We are using the player and enemy alive state for this reward
    //However, we could infact use any variables for rewards
    //Health before and after a choice could be tracked e.t.c

    float reward = 0f;

    if(playerHealth <= 0)
    {
        //Player died
        reward -= 1f;
    }
    else if(enemyHealth <= 0)
    {
        //Enemy died
        reward += 1f;
    }

    return reward;
}
```

- **Some Examples Functions for Calling** - these are just examples of functions that could be called for the battle bot to work.

```
//Example Functions
1 reference
public void AgentAttack()
{
    //Attack logic (or anims even)
}
1 reference
public void AgentDefend()
{
    //Defend Logic
}
1 reference
public void AgentFlee()
{
    //Flee logic
}
1 reference
public void EnemyAttack()
{
    //Enemy Attack Logic
}
```

4. Lastly, we bring everything together in a single function (for this example, actually agents could have several functions for this to span over), and call the two functions needed to make everything work; MakeAChoice(), UpdateReward(). Our function is:

    - **ExampleRound()** - this function takes the following steps to accomplish a successful Q Learning loop; Get our current state, Get a choice string, Act on the Choice, Get our new state, Get our Reward, Update our Rewards

```
0 references
public void ExampleRound()
{
    //Get our current state using the function we created
    currentState = GetOurState();

    //Get a choice string from the brain by feeding in the state (or in this example, only the string)
    string choice = ourBrainScript.MakeAChoice(currentState.stateString);

    //Our Code for acting
    if(choice == actions[0])
    {
        //Attack
        AgentAttack();
    }
    else if(choice == actions[1])
    {
        //Defend
        AgentDefend();
    }
    else if(choice == actions[2])
    {
        //Fless
        AgentFlee();
    }

    //Of course our enemy would also make his move next
    EnemyAttack();

    //Now we use the get state function to create our new state after everything has been donw
    newState = GetOurState();

    //We get our reward for the outcome
    float reward = RewardForActions();

    //And update our Rewards (Updating our Exeperiences) with the new state and our reward
    ourBrainScript.UpdateReward(reward, newState);
```

Finished

And that is everything needed (and a little extra) on how to set up Simple Q (it called simple for a good reason). If you have read all the documentation, and looked over the example, you are likely more than ready to begin training and dispatch ML agents and AI bots using Simple Q's framework.

Below I have added some additional steps to create a better version of this bot through multiple training sessions, Annealing Decay, and Episode Decay. Check it out, and play around. The script for this setup can be found in the example folder. It hasn't been attached to anything (it's Inspector variables set) and is ready to be loved and cared for!!!

Optional Additions

The next steps will illustrate additions that can be made to the example.

Multi-Runs:

1. First we will create a small helper function to reset the player (agent) and enemies health.

```
//Function for resetting health parameters
2 references
public void ResetHealth()
{
    playerHealth = 10;
    enemyHealth = 10;
}
```

2. Next we add in two variables, one for the amount of sessions we want, and one to count the sessions so far.

```
//Amount of sessions
public int amountOfSessions;
private int sessionsSoFar;
```

3. Then lastly, we create a coroutine to check our session count, then our player and enemy health, and call the ExampleRound() function while we are alive, or update the session count and reset the healths if there is a death.

```
0 references
public IEnumerator RunTrainingSessions()
{
    //Reset any variables first
    sessionsSoFar = 0;
    ResetHealth();

    //While we have yet to meet the numbe of sessions, keep checking health and calling the battle round
    while(sessionsSoFar < amountOfSessions)
    {
        if(playerHealth <= 0 || enemyHealth <= 0)
        {
            //Update our run for a new session
            ResetHealth();

            //Update session loop
            sessionsSoFar++;
        }
        else
        {
            //Continue to call our attack loop
            ExampleRound();
        }

        yield return new WaitForSeconds(0.1f);
    }

    //OPTIONAL - save when the whole session has finished
    ourBrainScript.Save_Open_QBrain();


    yield break;
}
```

With this, we can run multiple sessions, and optionally, - as I have included - we can save the entire brain at the end of the session (although it may have already been saved periodically through the Replay Buffer, it is best practice to save at the end of the session to ensure all learnt data is captured).

Annealing Decay:

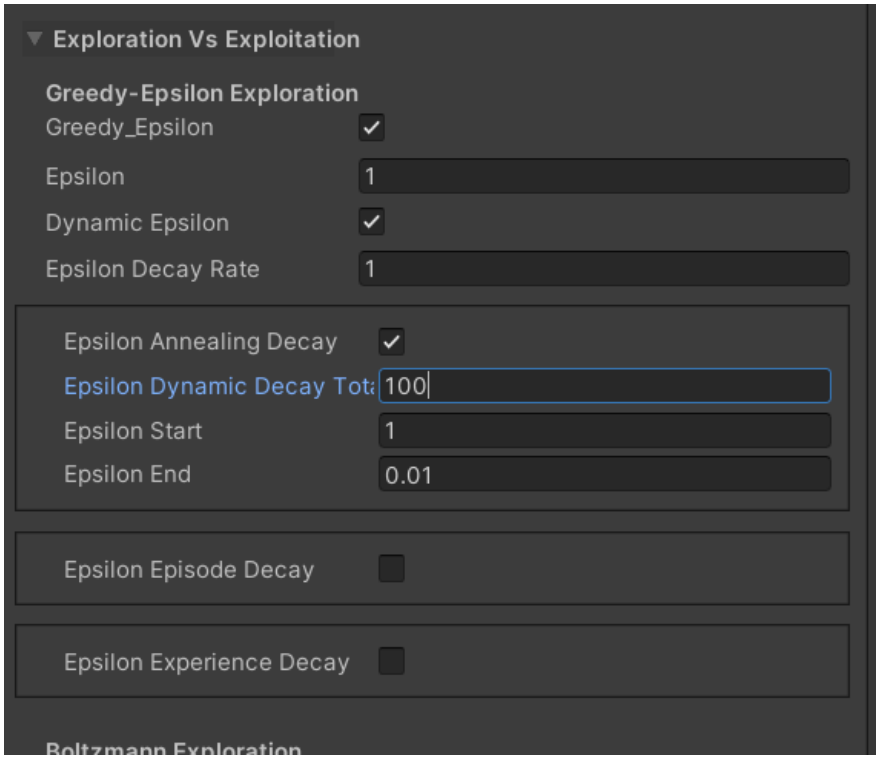
Since Epsilon-Greedy is the default, we will proceed with that for this example.

1. Setting the Inspector Up:

    - Head to the Inspector, ensure Epsilon Greedy is selected, and set the Dynamic Decay to “true”. This will give you the methods to choose from.

    - Set the Annealing Decay to “true”. Within you will find the start and end variables for the epsilon, and the training session total.

    - Input the amount of episodes are in your session - for our example, this would be our *amountOfSessions* variable. This value could also be calleds set through code, as it is set to “public”.



2. Adding to the Agent Script:

    In our coroutine, we will make a call to two functions (we could also just set these values ourselves through the brain script instance - but this helps keep it clearer for users).

- Add a **ResetAnnealingEpisodeDecayCount()** call to the start of our coroutine, to reset the count before starting the session.
- Add a **UpdateEpisodeForSessionAnnealingDecay()** call to the coroutine whenever we update the session counter.

Updating our coroutine would look like this:

```
0 references
public IEnumerator RunTrainingSessions()
{
    //Reset any variables first
    sessionsSoFar = 0;
    ResetHealth();

    //For Annealing Decay
    ourBrainScript.ResetAnnealingEpisodeDecayCount();

    //While we have yet to meet the numbe of sessions, keep checking health
    while(sessionsSoFar < amountOfSessions)
    {
        if(playerHealth <= 0 || enemyHealth <= 0)
        {
            //Update our run for a new session
            ResetHealth();

            //Update session loop
            sessionsSoFar++;

            //For Annealing Decay
            ourBrainScript.UpdateEpisodeForSessionAnnealingDecay();
        }
        else...

        yield return new WaitForSeconds(0.1f);
    }
```

With this, our example's epsilon value will now decay over a session, according to it's episodes.

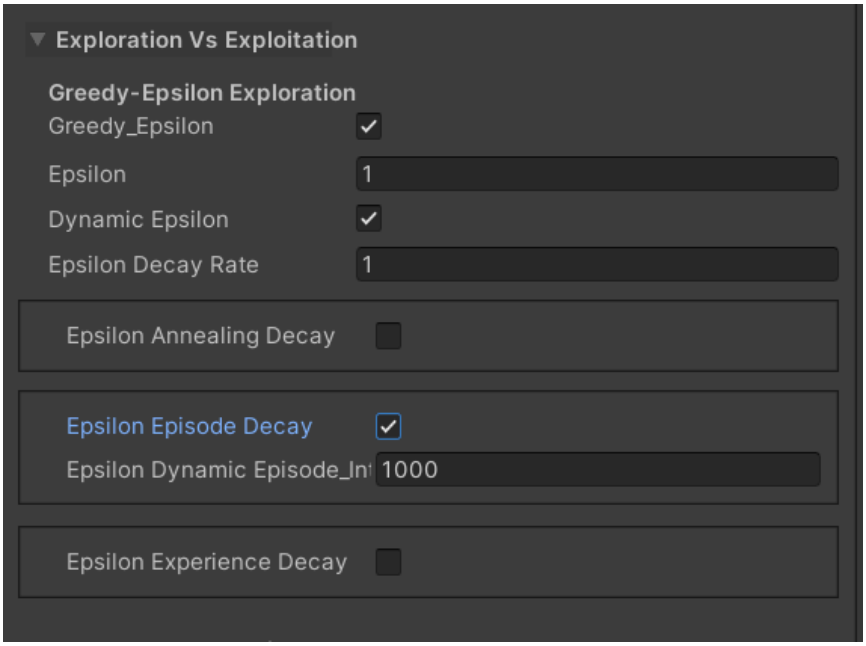
Episode Decay:

Again, Greedy-Epsilon is already our default, so we will proceed with that.

1. Setting up the Inspector:

    - Ensure the Greed-Epsilon is set "true", and set the Dynamic Decay to "true".

    - Set the Episode Decay to "true", and input the amount of interactions will be in a single episode (if, like our battle bot, there is a level of uncertainty, just make a simple guess).



2. Adding to the Agent Script:

    - Head into the agent script, and add a call to **ResetDecayInteractionCounter()** in the coroutine when we update the session count. This way we will reset the interaction count whenever an episode has ended.

The updated code would look something like this:

```
public IEnumerator RunTrainingSessions()
{
    //Reset any variables first
    sessionsSoFar = 0;
    ResetHealth();

    //For Annealing Decay
    ourBrainScript.ResetAnnealingEpisodeDecayCount();

    //While we have yet to meet the numbe of sessions, keep checking health and calling the battle
    while(sessionsSoFar < amountOfSessions)
    {
        if(playerHealth <= 0 || enemyHealth <= 0)
        {
            //Update our run for a new session
            ResetHealth();

            //Update session loop
            sessionsSoFar++;

            //For Annealing Decay
            ourBrainScript.UpdateEpisodeForSessionAnnealingDecay();

            //Here//
            //For Episode Decay
            ourBrainScript.ResetDecayInteractionCounter();

        }
        else
```

And with this the agents epsilon will now decay with his interactions (yes it is just one function call - the counter is updated internally when Episod Decay is selected). The Annealing Decay function calls have been left in, as they have basically no overhead, and if we switch back to Annealing, it is ready to go.

Further Details

In this section, we will grow into greater detail on how the different processes of the Simple Q Framework work, and the functionalities present.

Classes

The Simple Q Framework contains the following classes (of note):

1. ***OpenQLearningBrain:***

   This is the main class containing the data containers and the Machine Learning functions.

2. ***OpenQLearningBrainSaveManager:***

   The save manager class contains our save and load functions.

3. ***Open_QBrain_Class***

   This is the class we use for collecting information for filling our QTable with information. It contains:
   - StateClass - A state class containing the state sting and attributes
   - Actions - A list of actions for the state

4. ***State_Class (Important)***

   This contains the state string and the attributes pair. This class is what we use when calling the functions for our QLearning brain, since it contains the current state and after the new state, needed for learning.

   The class also contains an Equals override function, and a GetHasCode override function. Both functions are used to enable users to make simple comparisons, and key look-ups using only the class instance - they do the logic of checking if variables inside two classes match.

5. ***Replay_Buffer_Key*:**

   This class is used as the Key for our Replay Buffer data dictionary. It contains a State_Class state, an Action (taken), and an experience ID (this is set programmatically).

The class inherits from the IComparable<>, and has a compareTo function, allowing users to quickly, and easily access information matching this class's values.

6. ***Replay_Buffer_Values*:**
   This class contains the other values needed for the Replay Buffer; the Reward, the Next State (State_Class), and the Done flag.

   This, along with the Replay_Buffer_Key, make up the Open_QBrain_Replay_Buffer dictionary.

7. ***AttributesPair<TKey, TValue>:***

   This class, in its own script, is basically just a KeyValuePair, with the added benefit of being viewable in the inspector.

   It has the typical key and value setting functions. And is used for the Attribute pairs of a state.

8. ***OpenQLearningSharedData:***

   This class, again, in its own script, has two variables; a container for our QTable, and a container for our Replay Buffer.

   With this class, and the bool for sharedData set true on an agent - an agent's *OpenQLearningBrain* class will no longer use the local data containers of their class, instead using the shared data containers.

9. ***QTable_Data_Holder:***

   This class is used for storing data from the QTable memory, and then saving (serializing) the class instance as a JSON string. With this, we can easily load up the data, by passing the string back into an instance of this class (deserialize), before applying the variables to the relevant data structures and adding the data to the QTable.

10. ***RB_Data_Holder:***

    This class, much the same as the one above, is simply used for serializing and deserializing the Replay Buffer data.

Brain Initialization (Starting Up)

Simple Q has, as the name suggests, a very simple initialization process, which can be called from inside or outside the class using **InitializeQBrain()**.

Using the **State_And_Actions** list in the Inspector, a new brain can be created with specific States, Attributes, and Actions.

Functions:

- **InitializeQBrain():**

  The function for initialization (loading/creating) the QTable and Replay Buffer. This function also checks to see if we wish to use **sharedData** to use a shared QTable and Replay Buffer.

- **InitialzeANewOpenBrain()**

  The function for creating a new QTable from the variable set in the Inspector for the **State_And_Actions** list.

Initialization Logic:

- **Process:**

  When the initialisation function is called, if any previous data is existent (data saved previously) then the **Open_QBrain_Container** and **Open_QBrain_Replay_Buffer** will be loaded with the saved data. Else, our brain initialises a new container space using **InitialzeANewOpenBrain()**.

  InitialzeANewOpenBrain() loops through the **State_And_Actions** list in the Inspector, adding the values to the new brain. (This list does not actually need to be filled with loads of data, instead, a simple example state and actions will suffice to show the brain what it will be learning and doing.)

QTable Functionality - Brain System

Simple Q uses a QTable system in line with reinforcement learning to train agents. The QTables values are not normalized by the Framework, except when doing certain calculations - though this does not affect in any way the value the user has created from rewards. This is done to allow the user greater freedom in how the values of their QTables are structured.

The simple process for updating the QTable is as follows (after every interaction):

1. Get Q-values for the current state and action
2. Calculate the maximum Q-value for the next state (consider all possible actions)
3. Update Q-value using the Bellman equation
4. Set the updated Q-value in the Q-table

With the buffer replay process of batch learning included, updating the QTable is as follows (after a set period):

1. Sample a minibatch of experiences from the replay buffer
2. Loop through our experiences and get their QTable state
3. Calculate target Q-value using Bellman equation (with experience replay)
4. Update Q-value for the state in the QTable
5. (Additionally) Save the new data - again, might have to take this out

To do this, we have an automatic function for counting the interactions (action choices) made whenever **MakeAChoice()** is called. Once this counter hits the threshold set in the Inspector (choicesBetweenBatchLearning) the **UpdateQTable()** function will automatically be called, updating our QTable with the learnt experiences of the Replay buffer.

The framework contains the following functions to get this done:

1. **UpdateQTable** - the function for updating the QTable (and which calls the next functions)
2. **GetQValue** - returns the Q value for the state and action we pass in
3. **GetMaxQValue** - returns the max Q value for the state
4. **SetQValue** - set the Q value using state, action, target Q value and the current Q value, and use these with the learning rate to set the new Q value for that state.

Exploration Vs Exploitation

Simple Q utilizes the basic Epsilon-Greedy method and the more advanced Boltzmann exploration method, to tackle the exploration vs exploitation of our agent's action selection strategies.

Other methods (UCB, Thompson Sampling, e.t.c) can and should be explored to help users best optimize their agents for their needs.

Along with these two action selection strategies, Simple Q uses three methods for dynamically altering the Epsilon/Temperature variables of the two strategies:

- Annealing Decay (current episode / amount of episodes in a session)
- Episode Decay (current choice(step) / amount of choices(steps) in an episode)
- Experience Fluctuation (the mode of experiences)

With these dynamic changes, we can run longer sessions with more specific types of EvE values - even if those specific types are to favor more random choices. And, as with most things here, these are simply examples, and can be altered/removed/added to freely.
Annealing, and Episode decay are common methods for altering the choice value during a training session. These are the basic implementations of these methods.

The Experience Fluctuation is controlled automatically in the framework; by weighing up a sample batch of experiences, the Epsilon/Temperature are adjusted according to a simple measure of how “good” the choice made by the agent were:

If the agent has made loads of terrible choices, maybe a bit of randomness is needed to explore, if the agent has made a lot of good choices, give it more freedom to choose. (This of course is not the best method for promoting exploration, but does create good results, and demonstrates a different method for affecting an agent’s selection strategy).

For the Annealing Decay, we have two method that can be called, giving users full freedom over what they define as an episode and when it’s counter should rise:

1. **ResetAnnealingEpisodeDecayCount()**
   This sets the variable **decayAnnealing_current_episode_counter** to 0.

2. **UpdateEpisodeForSessionAnnealingDecay()**
   This function is called to increase the **decayAnnealing_current_episode_counter** variable.

For the Episode Decay there's only one function to reset the count, as an automatic counter will run with every interaction in the episode:

1. **ResetDecayInteractionCounter()**
   This resets the value of the Epsilon and Temperature, while resetting the variable **decayEpisode_current_action_counter** to 0.

For the **Experience Fluctuation** there are no methods to be called, but an ageWeight can be adjusted in the Inspector (value: 0-1).

“Experience Fluctuation” is a basic technique developed for Simple Q, which utilizes a constantly varying (fluctuating) number, over a linear-decay, to produce greater exploration in early stages, and less (hopefully if the bot has learnt well) in the later stages - more exploitation.

Experience Fluctuation works on the premise that if the bot has made a lot of what could be considered “bad choices”, it should be more explorative. If the bot has taken more “good choices”, then it can be trusted to be more exploitative. This is done with the following steps:

It takes a sample of experiences, and weighs the values against each other for a mean value, (experiences with negative outcomes vs experiences with positive outcomes), this value is passed through an ageDiscount (age * ageWeight), before being compared against the max value of the experiences. (Weighing if our the outcomes were overall positive or negative, and adjusting accordingly).

An age weight (it influence controlled by the **ageWeight** variable in the Inspector) has been introduced, on the logic that; an older model should be more displeased with it’s negative outcomes and more pleased with it’s positives). The **age** variable is calculated by the current amount of experiences in the memory vs the max amount of memories - if the dataset is nearly full, it should have a wide range of memories, i.e. it’s mature.

This of course is just an experimental method, developed whilst building this framework, so results using this method are not guaranteed.

Data Persistence (Saving/Loading)

Simple Q uses a basic saving and loading implementation. With a string variable set in the Inspector, the same QTable and Replay Buffer data can be accessed, loaded, manipulated, and saved. This is all done with Unity’s JSON utility.

The framework uses Streamwriter to save and load text files. Loading happens in the brain’s initialization stage. Saving can be called by the user with simple functions, or left to the automatic saving of the Experience Replay.

When the **sharedData** bool is set true, only one agent will load and save the data being used in the **sharedDataSource** variable.

This system works out-of-the box, but can be easily adjusted to fit users desires. The data saving has a Sort() function, but this is best set (programmatically) at the end of a session save.

Classes:

- **OpenQLearningBrainSaveManager** (located within the **OpenQLearningBrain** script).

Functions:

OpenQLearningBrain Save Functions:

- **Save_Open_QBrain()**
  This function calls the **OpenQLearningBrainSaveManager**'s two saving functions for the main data and the replay buffer data. It passes in the local container references. (This is the function users should call at the intervals they wish to save.)

OpenQLearningBrainSaveManager Functions:

- ***SaveOQLBrain(ConcurrentDictionary<State_Class, ConcurrentDictionary<string, float>> bQtable)***
  The function called to save our QTable data, with the current QTable data passed in as a parameter

- ***LoadOQLBrain(refConcurrentDictionary<State_Class,ConcurrentDictionary<string , float>> bQtable)***
  The function called to save our QTable data, with the table to be filled passed in using the "ref" keyword in the parameter. This function is called only once, when the brain is initialized.

- ***SaveOQLReplay(ConcurrentDictionary<Replay_Buffer_Key,Replay_Buffer_Values > bufferData)***
  The function for saving our Replay Buffer data, with the current replay buffer data holder passed in as a parameter.

- ***LoadOQLBReplay(ref ConcurrentDictionary<Replay_Buffer_Key, Replay_Buffer_Values> bufferData)***
  The function for loading the Replay Buffer data, with the current container ready to be filled, is passed in through the parameter with the "ref" keyword.

The Logic For Saving/Loading:

- **Variables:**

The string set in the Inspector **brainsSaveDirectory** is combined with a persistent path to create our directory string variable **path**. This is the path used for saving and loading our QTable.

For the Replay Buffer's path, **path_replay_buffer**, the string variable **replay_path_addition = "replay_Buffer_";** is combined with the **brainsSaveDirectory** string and a persistent path.

- **Automatic Saving:**

  All automatic saving calls have been commented out, due to the different needs of different agents. Some agents can run thousands of sessions, saving every interaction with no problem, whilst it slows down others.

  The only exception is in the **UpdateQTable()** function, which is called after the amount of interactions (action choices made) meets the max set in the Inspector (variable: ***choicesBetweenBatchLearning***), which Updates the QTable and then saves the QTable and Replay Buffer

  Again, it is advised that users call the **Save_Open_QBrain()** when they wish to save

- **Strings Variables:**

  Yes, string variables slow down the process, and using Streamwriter (with JSON Utility) is likely not the best method for very large models (QTables/Replay Buffers containing hundreds of thousands of experiences).

  However, for demonstration purposes, and AI bots up to a mid-range of experiences to be used in games, it suits the purpose perfectly. Making things easier for users to read, learn, and then adapt if they wish. (Anything over 100,000, should likely use a better saving and loading system.)

  By using custom classes, we can easily serialize and deserialize the data (with Json.Net this isn't necessary) to and from these classes.

- **User Saving:**

  The main method of saving is to call **Save_Open_QBrain()**. This function calls the save instance QTable and Buffer Replay functions, passing in the relevant data containers through.

Single save functions can be called from the save manager instance (e.g. Save_Manager.SaveOQLReplay(MyReplayBufferDataHolder);).

Important Notes:

- The variable **brainsSaveDirectory** should always be set to whatever you wish in the Inspector (is use .brain just for fun, use whatever you please). If you don't set it, you risk using the default value and overwriting if you create over bots.

- If the system is running slow try increasing the ***choicesBetweenBatchLearning*** in the inspector, as the internal counter **choices_taken_count_batch_learning** counts every choice made (action) until it reaches the number of batch learning which calls the UpdateQTable() function. At the end of this function we make a save call. This is our only automatic save - you can comment this save out if not needed.

- **sortedOut** is a bool that, when set to true, will order the data before saving. This helps with readability. However, use this with caution, as, if set true and the save function is called semi-frequently, the whole system will slow to a crawl. By default, it is set to false.

User Functions

When using the Simple Q Framework, alongside the functions that need to be called to use the brain, users have several functions they can call to help further specify certain aspects of the brain.

The main functions users will need to call when using Simple Q are:

1. **InitializeQBrain():**

   This is the function users call when starting up the brain framework. This function can simply be placed in the Start/Awake function of an agent, allowing them to Initialize the brain at start.

   For more on this function, see the "Brain Initialization" section above.

2. **MakeAChoice()**

   This function takes in a string state, and a list of Attributes pairs, and returns a string action choice from the brain. Only the string state needs to be passed into this function, as the Attributes Pairs list is optional.

This function increases our **choices_taken_count_batch_learning** counter for Replay Batch Learning.

This function also increases our **decayEpisode_current_action_counter** if we are using an episode decay method.

Calling this function essentially sets up the start of a learning loop, and allows users to create game objects that can think independently (i.e. this function can be called, and the result acted upon, and then nothing done thereafter, just a simple use of the brain - not a training exercise.)

3. **UpdateTheReward()**

   This is the final step in our loop, called once the result of our choice is evident (e.g. wouldn't want to call it halfway through an attack).

   This function takes in its parameter, a State_Class, and a reward float. These are used by the internal framework to add to the Replay Buffer as an experience, and later train the brain (the state is also added to the QTable if not yet learnt).

   Reward of course are calculated by the users, in whatever way they wish to train their bot (for more on rewards, check the example scripts, or the Setup below).

4. **Save_Open_QBrain()**

   This function saves both the QTable and Buffer Replay, giving users an easy method for saving all the data at once. Though this can be called at any time, it is best used sparingly on large datasets, as it can cause bottlenecks (especially if using sorted output).

   For more information on saving/loading, check out the Data Persistence section above.

5. **ResetAnnealingEpisodeDecayCount()**

   This function simply sets the Episode Decay Count for Annealing Decay to zero.

6. **UpdateEpisodeForSessionAnnealingDecay()**

   This function updates the counter for episodes when using Annealing Decay (the counter mentioned above).

7. **ResetDecayInteractionCounter()**

This function resets the interaction counter when using Episode Decay.

Example Agents Results:

I have detailed here the example agents/environments I have included in the asset, along with the results I got from them, and a conclusion about how they worked out.

Dodge Bot:

Agent/Environment:

The dodge bot was created in part to demonstrate Simple Q's ability to be used in a 3D setting, and to show how the agent can take in visual information through raycasts (currently taking in objects presence to the left, right, and ahead, along with the speed and distance of objects ahead) and act accordingly.

The dodge bot is set on a platform, with moving objects coming towards it. The agent chooses whether to move left, right, or stay in position. Jumping between a left, center, and right lane. Objects slowly increase in speed and amount, creating a greater level of difficulty.

An episode ends when the agent is hit by an obstacle, and we set the episodes to run in the Inspector.

The dodge bot helped to reveal the framework's need for a shared data component. Instancing the virtual battle bot multiple times inside the single class was fine, as they all used the same data. Duplicating the dodge bot agent/environment was creating an individual experience (fine in some cases, or when an already trained brain is being dispatched into a game) was creating different instances of the same brain. This would lead to individual experiences, and the last bot to call the save function overwriting the QTable with his experiences.

With the inclusion of the shared data component, agents can now be copied, and can reference the same shared data object to use instead of the local one in their instance.

Results:

Update: *After more testing, it appears, that the dodge bot can successfully begin dodging after a Replay Buffer Experience of 500 samples. Meaning, he can begin functioning optimally after around 3-10 training sessions! This test was run using Boltzmann Exploration and a static variable of 0.146.*

Due to the incredibly large amount of states the agent can be in (a distance that is 0.0001f different from another state's distance is classed as a new state, and that is only one of the variables that could mean we have a different state), and of course the dataset that creates, the

agent took a while to learn (at first).

Using the shared data component, I was able to cut the training time considerably (having around 29 agents all running the sessions together).

After around 3000-4000 training sessions with the bots (1000 at a time), using a mix of annealing and episode decay with the Epsilon-Greedy, the agent would stay mostly in the right lane. And would rarely dodge the obstacles in that lane - this I put down to the bot simply not having enough experience.

So, I ran the bot for another 1000 episodes, this time using Boltzmann, however, this time the episode's counter wasn't climbing (at least not rapidly) as soon as the Temperature had dropped below 0.15f, as the bot had begun successfully avoiding the obstacles.

Conclusion:

From the results of the agent, as can be seen from the results, is that with enough training time, the dodge bot has the ability to learn how to successfully gather visual information, make a decision, and act upon it in an optimal and timely manner, using the Simple Q framework.

It also has to be noted, that right before the bot began its transition from sticking to the right wall, to actually avoiding the obstacles, I did increase the Replay Buffer size variable from 5000 to 8000. This size increase (allowing for more experiences to be held) may also, and in my believe it highly likely, to be a contributing factor to the agent's success.

2D Navigation Bot:

Agent/Environment:

We generate a map for the agent (its environment), containing gold squares (gold coin rewards), red squares (obstacles), white squares (normal positions) and a blue square for the finish point. The maps have a seed function, placed purposefully to generate the same 10 maps every time (map changes are random - but it is the same 10 maps).

(This is where I discovered System.Random would be better for the framework, as the Unity seed function does not affect it, meaning users can call the InitState(int seed) without affecting the framework's code.)

The agent is given 20 steps per episode, to find the finish point. The amount of episodes is given in the Inspector. The session (of episodes) is exited out if the agent finds the finish point.

Currently, the console Debug.Logs how many sessions (episodes) the agent ran before getting to the finish point (if at all).

Result:

After setting the bot to train two-sets of 1000 episodes (approx 2-4 minutes), with an Epsilon-Greedy exploration, dynamic episode decay, the agent began finding the finish point in quicker succession.

Using only a string state that contains the map id, positions e.t.c of the bot. I trained the agent around 1000-2000 times (training sessions) on the same map, using a simple Epsilon-Greedy for exploration vs exploitation, and episode annealing (according to the steps for the each episode - 20).

After which, I changed the exploration vs exploitation to Boltzmann Exploration (preferring the probability selection over epsilon-greedy), and set the Temperature to a small value (not dynamic).

With this the agent's runs suddenly decreased rapidly, each session finishing far quicker than the last (less episodes before the finish line was found). Until finally, the agent finished every session in 1 episode, taking the quickest route that also passed through a gold square (earning that extra reward for it).

Conclusion:

When the 2D navigation bot was built, Simple Q was in a simpler condition, just having the Replay Buffer included, but no attribute pairs yet. I believe that the success of the agent to learn the map can be placed down to a few variables:

- Amount of episodes/sessions it had to gather data.
- The size of the Replay Buffer (which was very large at the time)
- The fact that only one map had been learned at that point (the map it was on), giving it access to only that map's values (why we introduced attributes for prioritized learning).
- The balance of rewards for the agent.

Though the agent's ability to learn is not random, like all ML agents, it is the right meta-variables and learning techniques that need to be used to produce the result required. Another map I tried didn't seem to work-out the same with episode decay, but worked better with annealing decay when learning it - however, with the latest updates, the agent can be trained to specifically find the fastest route through better prioritized learning parameters.

Virtual Battle Bot:

Agent/Environment:

The virtual battle bot was the first agent/environment built, originally using the Little Q build - meaning, it has the most basic of functionalities concerning the framework. Like the 2D navigation bot, it only uses a string state and actions.

The virtual battle bot can however work in groups on the same data, since the bot is virtual, we can create multiple instances of it on the same class, and run them through a large training loop.

The setup is simple. The agent makes a decision, it attacks, the enemy attacks back, and the rounds continue until someone has won, the agent has fled, or the agent has timed out. All detailed with Debugs in the console.

There is an option for a single battle (with all moves debugged) or a multi-battle training, where you can set the amount of battles and the amount of agents battling.

Results (The Cowardly Battle Bot):

The results were incredibly disappointing to say the least. The agent, though named a battle bot, had taken tp fleeing as the best approach. At first, it was an error with the multiple battle bots learning strategies and copying one-another. Once that was resolved, it just deteriorated into an agent who farms the defend and heal actions, and flees at the last moment. Finding and utilizing an exploit.

This was resolved, with the agent finding a better rhythm, and only fleeing when actually necessary - winning most of the fights, losing only a few, and timing out rarely. However, the solution was ctrl-z'd out, and the agent returned to his exploitative fleeing state.

Conclusion:

The Virtual Battle Bot is likely fleeing due to an exploit the agent has found in the reward system. Though not intentional at first, the Virtual Battle Bot has been left in this state - if you train it, it is highly likely to choose fleeing again as the most optimal option.

Challenge: If you read all the way down to here, or have happened across this, then a good challenge, in learning how to train agents, would be to get the virtual battle bot agent into a battle ready state - ready to be placed on an actual enemy or warrior in a 3D setting (so that object can use the brain to fight!).

Conclusion

The Simple Q framework is built by a newly enthused developer, who has begun learning ML/AI and would like to share the fundamentals with other developers starting their path in Machine Learning.

Simple Q is free to use, and will always be so, allowing developers to (hopefully) create amazing and inventive agents for their games.

It can be concluded from the results of the example bots, that agents, given the right data, with the right rewards system (and other meta-variables) can quickly learn how to competently perform tasks using Simple Q's simple framework.

Future Roadmap:

Simple Q will be frequently updated, with plans for further example methods, policies, and example agents.

Eventually Simple Q will utilize a flexible neural network (DQN), pushing the current version back to a simpler version for learning (much like Little Q, which can also be used to train and dispatch agents, but the current version dethroned it, as it will later be dethroned by Simple Q's neural network).