Reaction AI - User Guide

V2.00

Overview

Reaction AI is an artificial intelligence engine to be used for game developers that have non-player characters governed by AI or other game components that rely on artificial intelligence for reactions based on actions by the avatar. It can also be used for other systems that require learning systems and provides an intelligent output based on input.

Reaction AI models the human brain and neural networks in its design. It learns over time and uses past events in a memory system to determine the next best reaction. However, it is pseudo-dynamic in the resulting output based on how well the "brain" is functioning at that time based on stimuli. The "brain" is divided into four components, each sized proportionally to the human mind. This includes the Frontal Lobe which performs reasoning and emotion, the Parietal Lobe for movement, the Occipital Lobe for visuals, and the Temporal Lobe for memory.

When the "brain" is created, the number of total neurons must be specified. This will generate an n-ary tree. At each time a new "brain" is created (i.e. for different NPCs), a different brain structure is created dynamically and pseudo-randomly. When stimuli is applied (i.e. an avatar action), Reaction AI will send a random set (within a range) of electrons to each component of the brain (also proportional to the size of the respective lobe) to activate a subset of the neurons. The algorithm uses the active neurons to total neurons to determine the performance of the "brain" in each of the lobes for their respective purpose. Thus, a better functioning "brain" will yield more "intelligent" results. Each time stimuli is applied, the "brain" will function slightly better or worse due to the difference in the number of active neurons. This is done by ranking the results and using data based on whether it can map the avatar action with a calculated emotion and prior reaction data for what the NPC "thinks" is the best next action. If certain data fails to map, fallback mechanisms are performed which still are "intelligent".

There is a memory system where historical data is stored. When a new memory is created, the successfulness of the most recent reaction is also mapped to that memory cell. The data includes what the avatar action was, what was the AI's output for that action, and how successful it was. This data grows over time and the "search" starts from most recent memory to earlier in time memories. However, the "search" also goes as far back in memory as the number of active neurons at that time. The maximum size of the memory is the total number of neurons in the temporal lobe. When a new memory is created, and the maximum size is reached, the oldest memory is removed. When the "brain" is created, the memory is initialized with a user defined input (one time) for the NPC's "personality", which defines the allowed reactions, sub-reactions, and emotions.

The final output of the system is a reaction category, sub-reaction, emotion, visual rank (ranging from 1 -5 where 1 is the lowest rank and 5 is the highest), and a speed value (ranging from 1 – 4 where 1 is the slowest and 4 is the fastest). You can decide how to handle (how the NPC behaves) based on the output, and can cause a different reaction as well based on the values returned for emotion, visuals, and speed. The neural networks involve the structure of the "brain", the stimuli that activate neurons, the memory, and the personality, being passed to different functions in different parts of the "brain" and results in final outputs. The fact that the system is varying each time a "brain" is created, the performance of the "brain" per stimulus, and what the allowed output values can be, allow Reaction AI to be reused any number of times with varying results. Thus, the same engine can be used for any number of different NPCs or AI based elements. The user defined "size" of the "brain" also introduces dynamics. Finally, for extremely large "brain" sizes, and longer periods of learning, Reaction AI can be as complex accordingly.

Setup and Usage

Requires .NET Framework 3.5 or higher

To use Reaction AI, simply add the DLL reference provided to your project. If you are using the Unity3d build, simply add the reference to the assets folder and include (using) it in your C# script. When creating the brain, it needs to be initialized with a personality. This is done with a json formatted input (see the sample JSON provided). For the generic build, the json file uses the .json extension, while when using Reaction AI in Unity3d, the extension is .txt and needs to be placed in a created "/Resources" folder in the assets folder. The path to this file must be provided when creating the brain. If, in Unity3d, if the file is in a subfolder under /Resources, the path must be provided, but the file name provided must NOT include the extension.

The JSON personality file has reactions with its own set of sub-reactions, where the sub-reactions are tied to an emotion. You can have any number of reactions, but the reaction value (integer) must be unique across all reactions, unless you want to map the same sub-reaction to a different emotion. You can have as many sub-reactions per reaction, but the sub-reaction integer value must also be unique across all the sub-reaction for all reactions. The emotions value can be reused but can only be integers from 1 to 10 (you don't need to use 10 total as you can use any value up till 10 (see the provided sample json as an example).

When creating the "brain", any number of neurons can be specified. However, integer values ranging from 10,000 to 100,000 are recommended. When sending the stimuli, provide an integer value that represents the avatar action. After sending the stimuli, create a new memory with the reaction object response, and also the success of the output as a percentage between 0-100. Finally, clear the stimulus which resets the active neurons to inactive neurons.

Flow Example:

```
Brain brain;
Brain.Reaction reaction;

brain = new Brain(50000, "ReactionAI")

//.txt JSON formatted input personality initialization file in /Resources Folder


reaction = brain.sendStimuli(2);//avatar action was 2

//NPC to perform action…


brain.createNewMemory(reaction, 89);//NPC reaction was 89% successful

brain.clearStimuli();

//Repeat sending stimuli, create memory, and clear stimuli
```

SEE THE SAMPLE APPLICATION FOR USAGE

The Reaction_AI.dll exposes the `Brain and its members` class and the `Brain.Reaction` class and its members. You need to instantiate a single instance of the Brain class per NPC. The Brain.Reaction object can be simply declared. This object gets assigned as the return type of the Brain member, sendStimuli.

Brain constructor

```
Brain(<param 1>, <param 2>);
```

Param 1: # of neurons of the entire brain (recommended to use 10,000 to 100,000 neurons). The more neurons there are the more dynamic the brain can be

Param 2: path to JSON (.txt) input file for brain initialization. DO NOT USE THE .txt EXTENSION in the argument.

`Brain` members:

```
sendStimuli(<param>);
```

Param: an integer representing the avatar action just performed

Return type: Brain.Reaction object. Assign the return value of this member to the previously declared reaction object.

Usage: call this every time you want a reaction from the AI.

```
createNewMemory(<param 1>, <param 2>);
```

Param 1: the same and unmodified Brain.Reaction object returned from sendStimuli. This uses the reaction of the AI from sendStimuli to see how well it did against the avatar.

Param 2: an integer from 0-100 denoting (in percent) how well the AI's reaction was against the avatar. For example, "no damage" could equate to 0, and a lethal attack could be 100. The AI will map this success percent to the action it just performed so it "learns" for the subsequent reactions.

Return type: VOID

Usage: call this every time sendStimuli is called so the AI can "learn"

```
clearStimuli();
```

Return type: VOID

Notes: this needs to be called every time sendStimuli is sent to clear the active neurons. If this is not done, the reaction will not work as expected as the results will be aggregated and skewed which would be incorrect.

`Brain.Reaction` members:

SHOULD ONLY BE ACCESSED ONCE sendStimuli RETURNS A RESPONSE

```
getAction()
```

returns the AI action (integer) number (each action number has a set of subreactions). For example, the action value could represent "attack", and the subreaction could be "punch". The action value will need

to be included in the input JSON (.txt) initialization file. Furthermore, the action value could represent different categories of attacks also. For example, value 1 could be melee, value 2 could be weapon based, etc.

`getSubReaction()`

returns the subreaction value (integer) for the action deduced. As stated above, if the action is "attack", the subreaction could represent "punch", "swing sword", "use sniper", etc.

`getEmotion()`

returns one of the most determined emotions of the NPC. This is an integer value that is based on the JSON input file for initialization which is initially tied to a subreaction. If the input file defines 5 emotions (10 max allowed) then emotion 1 could be mapped as aggressive with an attack type subreaction, and emotion 2 could be associated with FEAR for the "flee" action type. The emotion and subreaction over time do not have correlation except for how the AI learns. That means an emotion of FEAR could be also return an action group of ATTACK. You should determine logic of how you want the avatar to respond finally to this combination.

`getVisualRank()`

returns an integer value between 1 and 5. 1 being poorest vision, 5 being, best vision. This can be used in the final NPC action, for example, if the action group is FLEE, 1 will make the NPC at the last minute, while 5 would allow the NPC to have a better chance of escaping by fleeing earlier.

`getSpeed()`

returns an integer value between 1 and 4. 1 being the slowest, and 4 being the fastest. This can be included in, for example, the attack speed with respect to frames per second it takes for the action to complete.

Total continuous flow

For NPC *x*:

Pre-conditions:

Create brain

Declare reaction object

Flow:

Send Stimuli

Create memory

Clear Stimuli

Repeat steps 1-3 per avatar action during NPC vs. Avatar engagement

Personality Initialization Input File

From the sample file provided:

```
{
	"reactions": [
		{
			"reaction" : 1,
			"subreactions" : [
				{"subreaction" : 1, "emotion" : 1},
				{"subreaction" : 2, "emotion" : 1},
				{"subreaction" : 3, "emotion" : 3},
				{"subreaction" : 4, "emotion" : 3}
			]
		},
		{
			"reaction" : 2,
			"subreactions" : [
				{"subreaction" : 5, "emotion" : 2},
				{"subreaction" : 6, "emotion" : 2},
```

```
                {"subreaction" : 7, "emotion" : 1},
                {"subreaction" : 8, "emotion" : 4}
        ]
},
{
        "reaction" : 3,
        …
```

Structure

The JSON file you use a .txt extension with a JSON format. This file is used as a "seed" to the NPC personality and initializes the memory with the file contents. It should exists in a "/Resources" folder and the path specified in the Brain constructor. When the "brain" initializes it will assign pseudo-random success rates to each subreaction. From this "seed" the NPC will learn and modify its memory by averaging the best subreactions. Thus, the longer the NPC "learns", the more accurate the subreactions become over time.

The JSON's parent is an array called "reactions" which holds multiple reaction numbers. You can have as many reactions as you like. Each reaction should have a unique value (integer). Each reaction also has an array of subreactions. You can have as many subreactions per reaction, but overall, each subreaction value must be unique across the file (integer value). A subreaction will have an emotion value associated with it. You can mix and match emotions with subreactions but are limited to 10 emotions (valued from 1-10). You do not, however, need a full 10 emotion values.

Unity Asset Structure

Script

In your C# script, include the dll namespace

```
using Reaction_AI;
```

JSON Input file

In the Brain construction, use the path under the Resources folder to the file without including "Resources" and without the starting forward slash at the beginning of the string for the path.