Reaction AI ReInvented

Hourglass Software LLC

Core Features

- Dynamic Non-player character Combat Artificial Intelligence using Machine Learning Multiclass prediction engine (Reaction AI + Microsoft ML.NET)
- A* pathfinding
- Automatic audio detection and following for NPC

Combat Overview

Reaction AI is an artificial intelligence engine to be used for game developers that have nonplayer 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 Mind 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 "mind" is functioning at that time based on stimuli. The "mind" is divided into four components, each sized proportionally to the human mind.

Once there is enough data, it will train an ML .NET multiclassification model and use that model to make predictions of what the NPC action should be. The prediction is based on a feature vector that includes providing a parameter for the desired success value, and then vending the respective action based on that with the other features provided for the training and prediction.

If a model has been trained, it will be saved locally and used for further predictions. The model can continuously get trained with additional data over time. The training data is also saved locally and can be used to train the model upon making an API call. The training data size limit is also configurable and will not exceed the amount specified (to not allow the data size to increase beyond a desired amount).

The multiclass training and predictions use a Multiclass Maximum Entropy training algorithm.

Before the model is trained, the API will utilize the base Reaction AI algorithm for determining the NPC move. As mentioned, once the Machine Learning Model is trained, it will then start making prediction using the trained model. Furthermore, the predictions will also introduce a reaction using the base algorithm (not the model) once every 5 times as a means to introduce "noise" to allow the next iteration of training the model to be more dynamic.

The AI algorithm and Machine Learning model feature vector 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 "mind" is created, the number of total neurons must be

specified. This will generate an n-ary tree. At each time a new "mind" is created (i.e. for different NPCs), a different Mind 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 Mind (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 "mind" in each of the lobes for their respective purpose. Thus, a better functioning "mind" will yield more "intelligent" results. Each time stimuli is applied, the "mind" 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".

For the base Reaction AI algorithm, if the model is not trained, but used in the training data, 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 "mind" 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 "mind", the stimuli that activate neurons, the memory, and the personality, being passed to different functions in different parts of the "mind" and results in final outputs. The fact that the system is varying each time a "mind" is created, the performance of the "mind" 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 "mind" also introduces dynamics. Finally, for extremely large "mind" sizes, and longer periods of learning, Reaction AI can be as complex accordingly.

The training feature vector includes the following data:

1) Last SubReaction value
2) Emotion value
3) Speed valud
4) Avatar action value
5) Number of active neuron and neurons ratio
6) Success Percentage
7) Information about Avatar health and NPC health

8) Time value in between moves

Setup and Usage Requires .NET Framework 4.6 or higher

Unity Version: Unity 2019.2.19f1 (64-bit) or Above

Architecture: x64

To use the API, place both Reaction_AI.dll and Reaction_AI_ReInvented.dll in your project (Assets references folder).

In your script, include the namespaces:

using Reaction_AI_ReInvented;

using Reaction_AI;

For the Machine Learning ML .NET, you will also need the following DLLs in your project's "Plugins" folder:

1) CpuMathNative.dll
2) Microsoft.ML.Core.dll
3) Microsoft.ML.CpuMath.dll
4) Microsoft.ML.Data.dll
5) Microsoft.ML.DataView.dll
6) Microsoft.ML.KMeansClustering.dll
7) Microsoft.ML.PCA.dll
8) Microsoft.ML.StandardTrainers.dll
9) Microsoft.ML.Transforms.dll
10) Microsoft.Win32.Primitives.dll
11) Newtonsoft.Json.dll
12) System.Buffers.dll
13) System.Collections.Immutable.dll
14) System.Memory.dll
15) System.Runtime.CompilerServices.Unsafe.dll
16) System.Threading.Tasks.Dataflow.dll

The Reaction_AI_ReInvented DLL has the "Mind" class for using ML.NET. Reaction_AL.dll has the classes for A* pathfinding and audio follow. Also, you can use "Mind" from Reaction_AL.dll to use the non-ML.NET AI from the original Reaction AI library.

When creating the mind, 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 Mind. 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 subreactions 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 "mind", 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:

```
Mind mind;

Mind.Reaction reaction;

mind = new Mind(50000, 30000, 1, "SampleJSONPersonalityInputFile");  //.txt JSON formatted input personality initialization file in /Resources Folder . Also specify the NPC ID value.  In this case it is 1.

 //NPC to perform action…


reaction = mind.sendStimuli(avatarAction, avatarHealth, npcHealth, deltaTime, 89);

//the success percent value should be the desired success percent of the reaction as part of the feature vector to make a prediction . if the model is created, this will automatically switch from the base AI to use the Multiclass predictions from the model.

npcAction = reaction.getAction();

Debug.Log(reaction.getConfidence().ToString());

Debug.Log(reaction.getSubReaction().ToString());
```

```
mind.createNewMemory(reaction, avatarHealth, npcHealth, delta, successPercent);//success percent is how well the reaction was when making the move

mind.clearStimuli();


//if in the update loop can save and train the mode as so.  Will retrain the model every 60 moves

if (moveCount > 0 && isTrained==false)

    {

      if (moveCount % 60 == 0)

      {

        isTrained = true;

        if (mind.save())

        {

          mind.trainModel();

        }


      }

    }


    if (moveCount % 60 != 0)

    {

      isTrained = false;

    }
```

(SEE THE SAMPLE APPLICATION FOR USAGE)

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

Mind constructor

Mind(<param 1>, <param 2>, <param 3>, <param 4>);

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

Param 2: memory which should be equal to or less than number of neurons. When saving training data, the number of records will not exceed this amount

Param 3: NPC ID

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

Mind members:

sendStimuli(<param 1>, <param 2>, <param 3>, <param 4>, <param 5>);

Param 1: an integer representing the avatar action just performed

Param 2: Avatar Health

Param 3: NPC Health

Param 4: (long) time since last move in milliseconds

Param 5: (integer) desired success percentage

Return type: Mind.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 3>, <param 4>, <param 5>);

Param 1: the same and unmodified Mind.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: avatar health

Param 3: npc health

Param 4: time since last move in milliseconds (long)

Param 5: 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.

save()

saves the training data to a file locally. Returns TRUE for success and FALSE otherwise.

trainModel()

trains a Multiclass prediction machine learning model based on the training data. You must save() successfully before this is called. Once trained, sendStimuli automatically will use the model. Created by a separate thread so will not block main thread. If starting a new session, and this model exists for the NPC ID, it will automatically load when instantiating a “Mind” for the NPC ID given. You should save and train every X moves so the predictions are accurate and current.

Mind.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.

getConfidence()

Returns the confidence value for the most recent Reaction/Subreaction predicted by Machine Learning

Total continuous flow

For NPC x:

Pre-conditions:

Create Mind

Declare reaction object

Flow:

Send Stimuli

Create memory

Clear Stimuli

After every X moves, save the data and train the model

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 Mind constructor. When the "mind" initializes it will assign pseudorandom 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.

JSON Input file

In the Mind 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.

A* Pathfinding Overview

The asset contains demo scenes, grid files and scripts.

You can have as many NPCs use A* (limited by performance) with their own AStar object. The pathfinding uses a text file that describes the map/level that you must create first.

You can also call the pathfinding method on the same AStar object if the "goal" has changed by calling the setGoalLocation method and setNPC method and then call getShortestPath after that. To follow a different path after this, simply reset, set the new Goal, update the NPC position with updateNPCPos, and then get the shortest path again. To follow the path, call followPathGoal.

Using Reaction AI allows you to path-find to the Avatar, and then use the combat support to enter combat with the Avatar. Furthermore, you can design the NPC with the Reaction AI to have some of its reactions map to follow/find the avatar.

For example, the Mind can use a 'reaction' 4 which maps to 'follow/find' if the avatar flees and the 'Mind' for the NPC has learned this.

There are only four API calls required to implement A* within Reaction AI.

Methods Constructor:

public AStar(float startXCoordinate, float startZCoordinate, float endXCoordinate, float endZCoordinate, int numGridRows, int numGridColumns, String gridMapFile)

Parameters

startXCoordinate

This parameter specifies the starting X-coordinate of the bounded map for a level/room and is the X position coordinate in Unity's scene.

startZCoordinate

This parameter specifies the starting Z-coordinate of the bounded map for a level/room and is the Z position coordinate in Unity's scene.

endXCoordinate

This parameter specifies the ending X-coordinate of the bounded map for a level/room and is the X position coordinate in Unity's scene.

endZCoordinate

This parameter specifies the ending Z-coordinate of the bounded map for a level/room and is the Z position coordinate in Unity's scene.

numGridRows

The number of rows in the grid file (not including EOF row)

numGridColumns

The number of columns in the grid file (not including newline).

However, the map reader is expecting CRLF (\r\n) at the end of each row. If your system does not use 2 characters for the newline, but only 1 instead (just \n), then this value will be the number of columns excluding the newline minus 1. If this is the case, for example, if your number of columns (filled with '1' or '#') is 40, use 39.

gridMapFile

The name of the map grid file that you will create in the /Resources folder without the .txt extension.

NOTE: Reaction AI will determine the cell size based on the bounding box specified by the start and end coordinates and the number of columns/rows specified.

public void setNPC(GameObject npc)

sets the location of the NPC based on the Unity scene position coordinates and uses this object for future position updates. This must be set before finding the shortest path.

public void setGoalLocation(float xCoordinate, float zCoordinate)

sets the location of the GOAL based on the Unity scene position coordinates. This must be set before finding the shortest path.

public float[,] getShortestPath()

Executes the A* algorithm and returns the x and z coordinates of the shortest path from the NPC to the goal. The first dimension of the returned array is the x-coordinate of the Unity position value and the second dimension is the z-coordinate. You can set any y-coordinate you choose. From these locations, you will then need to transform the NPC to follow the path returned (see included samples).

public void followPathGoal(float [,] path, float totalTime, float incrementMultiplier)

Follows the path returned by getShortestPath with the total duration and speed specified as parameters. This should go in the Update loop, but only called to start following the path as for as long until the path following is complete and the NPC has reached the Goal

public bool isFollowingPathComplete()

If a path has started, this will return whether or not the path is complete. True for completed and False otherwise.

public void reset()

This must be called if and before traversing a new path.

public void updateNPCPos()

updates the location of the NPC. Call this before getting a new shortest path.

The Grid File The grid files need to be created to represent your map. It is a top down view of which areas can be traversed and which cannot (obstacles). The files must have the .txt extension.

The grid files MUST be exactly proportional to your maps!

It is recommended to scale proportionally so each cell in the grid file is about the size of the bounding box of the NPC (but does not have to be). Smaller cells will result in more precise locations for traversing. The actual size of the cell is calculated automatically based on the bounding box coordinates and the number of columns/rows.

Use the character '1' to represent a cell that can be traversed, and '#' for those that are not due to obstacles.

The top left of the grid file data is the start x and z coordinates, and the bottom right is the end x and z coordinates. Each row must end with a newline, including the last row. The next row first character is EOF.

You do not need to specify the NPC or goal locations in this file.

Example (testmap3.txt, cell size is 10 units based on testmap3 sample Unity scene):

11111111111111111111

11111111111111111111

11111111111111111111

```
11111111111111111111
11111111111111111111
11111111111111111111
11111111111111111111
1111111111####11111
11####11111####11111
11####11111####11111
11####11111####11111
11####11111111111111
11111111111111111111
11111111111111111111
11111111111111111111
11111111111111111111
11111111####11111111
11111111####11111111
11111111####11111111
11111111####11111111
```

Automatic Audio Detection and Following

Leverages the A* pathfinding implementation where the NPC can detect audio playing and then automatically go to that location.

It combines some of the logic for the previous section (A* Pathfinding) with its own specific requirements and features.

The audio that plays for detection must be non-looping. This is to ensure the NPC does not track music based audio sources which are not involved in the gameplay. Two other requirements, which are configured via the API, are setting the maximum radius from the NPC for which it can detect audio playing from audio sources, and a time interval where the NPC will only detect sound and follow it once the time interval has elapsed. This is so that the NPC does not follow all audio at any moment and creates more realistic gameplay as AI. For example, if the interval is set to 20 seconds, the audio will only be detected and followed if it occurs after 20 seconds. Once this occurs, the timer is reset to 20 seconds again, so the NPC will only detect the new audio once the new 20 second timer also elapses.

This feature can be enabled or disabled with the API.

Furthermore, the API is part of the A* class for the same A* object created mentioned in the previous section with:

public AStar(float startXCoordinate, float startZCoordinate, float endXCoordinate, float endZCoordinate, int numGridRows, int numGridColumns, String gridMapFile)

It shares the same APIs as the A* class but with its own specific members. This allows automatic audio detection and following with other cases to follow either the Avatar or other NPCs.

Place the method, followClosestAudio, in the Update method. If audio is detected, the NPC will automatically go to the location of the audio source. However, unlike the base pathfinding implementation, this does not require getting the shortest path first as it is automatically determined and set the audio source as the Goal, and as long as the NPC position is updated, and the reset method is called before each new detection and traversal.

The audio locations are automatically determined, and if there are multiple audio clips playing at the same time, the NPC will go only to the closest one, again, simulating real-world behavior.

Methods (in addition to AStar methods above)

public void enableFollowAudio()

enables the feature

public void disableFollowAudio()

disable the feature

public void setAudioDetectRadius(float radius)

sets the maximum distance from the NPC where audio can be detected

public void setAudioCheckInterval(float interval)

audio will only be detected after this time interval elapses. Restarts once the previous path completes.

public void followClosestAudio(float time, float increment)

should be placed in the Update loop method. If this is in Update, once audio is detected given the constraints, the NPC will automatically go to the source with the total time and speed specified.