Organic Non-playable characters using neural networks

Student Name: Oscar Johnson
Student Number: 16025481
Supervisor: James Huxtable
Department of Computer Science and Creative Technology
University of the West of England
Coldharbour Lane
Bristol, UK
Oscar2.Johnson@live.uwe.ac.uk

Figure 1. A screenshot of the Project.

Summary

This project uses neural networks to drive an NPC agent that behaves realistically in an RPG setting. Using three neural networks, one for idle behavior, one for combat behavior and one final emotionally driven network, this project simulates a realistic bar brawl setting where NPCs can act on and react to in game events.
This project includes a small framework for developing behaviors through neural networks and training these networks to behave as expected.

Brief Biography

I am a young and aspiring student currently studying Games Technology (BSc).
I have always been interested in AI and wanted to use this project as a way learn more about the current practices used in the industry and how other methods could be applied to better effect. I chose neural networks as they employ machine learning, a skill I have always been infatuated with and one that many employers are seeking as they believe it is at the forefront of current technology.

How to access this project

The project can be accessed from Github at: https://github.com/JohnnersUK/Organic-NPCs.
The accompanying video can be watched here: https://youtu.be/nAxArUUPL_A

After downloading the project, a demo can be launched by running CTP.exe from the /builds/ folder.

Controls: W,A,S and D move the player, Space bar jumps, Shift runs, Left control toggles combat and E attacks.

The network visualizer can be activated by pressing f1 and navigated using the arrow keys.

To view the code, look at the trainer and network customizations options as well as the network visualizer this project can be opened in Unity2018.2.2f1.

The trainer can be found in the scene hierarchy and the agent prefab can be found in the root asset folder as botPrefab.

The tavern assets were purchased from the unity asset store, the robot's model and animations were gathered from mixamo.com and the shader for viewing the player through walls was found on the unity3d wiki. All sources can be found in the appendices under asset sources.

Introduction

This project set out to solve the issue of developing realistic NPCs that allow for greater player immersion as well as system optimization.

Development of AI systems are often restricted as they are less marketable than other game features such as graphical fidelity and gameplay content, therefore they are often lackluster and overly simplified to decrease their overhead and impact on system performance.

This project aims to solve this issue by using neural networks to control NPC behavior. In theory, neural networks have a relatively low impact on performance as once trained, evaluating the network is just a short series of floating-point operations which modern CPUs excel at.

Currently, behavior trees (as seen in the video game Halo 2 *(Bungie, 2004))* are one of the preferred methods of video game AI. However, as discussed in a blog post on Gamasutra *(Rasmussen J, 2016)* they are slow to implement meaning *"Game designers cannot quickly prototype their ideas"* and they are hard to expand on as *"The number of bugs and time to fix them will grow exponentially",* resulting in lackluster AI.

Additionally, in a Comparison of the decision tree, artificial neural network, and linear regression methods *(Kim Y S, 2008)* it was found that neural networks are consistently faster at evaluating than decision trees, the same premise behavior trees are based on.

Therefore, the goals of this project can be summarized as follows:

To create an AI system that acts consistently, believably and as the player would expect. An AI system that has a low impact on system resources as to save overhead for other systems such as physics and rendering. And an AI system that is easy to develop,

that is highly customizable to the developer's needs and that can be prototyped rapidly.

Neural networks are not perfect either. The neural network model can be more difficult to understand than decision trees and as neural networks operate like black boxes, it can be difficult to see how the network is operating making bug fixing difficult. These were all issues that were taken into account throughout development.

Using neural networks, this project has achieved an AI system that has a minimal impact on performance, is easily integrated into rapid development cycles and delivers an immersive experience for player.

Practice

Developing the neural network class

The network class was developed to be modular, easy to modify and have a low overhead. As a class, this network class can be added to any unity mono behavior allowing for network control in any script.
A constructor was created that takes an array of integers along with a string. Each element in the array represents a layer with the value representing the depth of the layer, this allows the networks to be fully modifiable in number of inputs, number and depth of the hidden layer and the number of outputs.
The string is used for identification so that the networks can be sorted later.

The activation function tanh was chosen for the network after multiple rounds of testing training effectiveness. The research phase *(Johnson, O. 2018)* discussed using a linear or sigmoid activation function

```
public NeuralNetwork(int[] layout, string n =
"Default")
{
    _layers = new int[layout.Length];
    for (int i = 0; i < _layers.Length; i++)
    {
        _layers[i] = layout[i];
    }

    // Initialize the neurons
    List<float[]> neurons = new List<float[]>();

    for (int i = 0; i < _layers.Length; i++)
    {
        neurons.Add(new float[_layers[i]]);
    }
```

CODE SNIPPET FROM NEURAL NETWORK INITIALIZATION

rather than a step function as it allowed for a greater degree of control over the outputs akin to fuzzy logic.

Upon implementation it was found that after a few rounds of training and mutation the weights became too large, causing the delta between outputs to grow. This made it difficult to train for different behaviors and data sets as small weight adjustments lead to massively differing outputs meaning more rounds of training were required to fix errors.

More research into methods of managing weights such as weight decay to regulate growth *(Krogh A, 1992)* and alternative activation methods was required. This was a short sight of the research phase *(Johnson, O.*

2018) and could easily have been avoided if this topic was covered with more scrutiny.

A paper studying the performance of different activation functions *(Karlik B, 2011)* found that on average, the tanh activation function gave better accuracy overall than a sigmoid function. As discussed in "Activation Functions: Comparison of Trends in Practice and Research for Deep Learning" *(Nwankpa C, 2018),* Tanh skews the range of outputs between -1 and 1 which removes the delta between outputs caused by larger weights.

After implementing Tanh, the training results showed less errors caused by differing outputs and helped improve the overall fitness of the networks in fewer training rounds.

Also discussed in the research report was implementing backpropagation for improved training results. For function, a mutation algorithm was included for evolving weights in-between training sessions, with the intent of removing this and implementing back propagation later on.

This implementation of mutation allowed for rapid training of test behaviors at a low cost. This sped up the development time as the turn around on prototype builds was short

With feedback from the supervisor of this project and the play test results, it was found that through mutation the agents were reaching realistic enough behavior that the use of back propagation seemed like a waste of time which could be used to develop additional behaviors.

Activation Function:	Average fitness after 5 training rounds:
Linear	652
Sigmoid	899
Tanh	1355

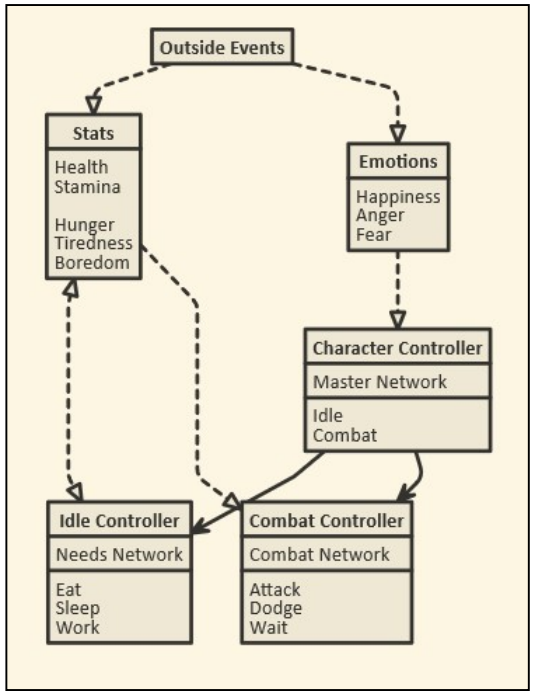
TRAINING DATA BEFORE AND AFTER IMPLEMENTING TANH

Creating behavioral classes

As each behavior required its own network and resulting functions for initializing, storing and running through them, it made sense to create a base class for all agent behavior scripts to inherit from. Not only did this class help streamline the process of creating new behaviors but it also allowed for the polymorphism used in the training script discussed later in this report.

The final project contains three network driven scripts, they can be broken down as follows:

DIAGRAM OF SYSTEMS DRIVING AGENT BEHAVIOR



The agent controller script acts as a top-level class containing a master network. This network chooses what behaviors to run based off results from the two other networks as well as the agents list of emotions. This integration of the emotion system allows the network to drive the agent's actions off their emotions much like humans tend too, in an attempt to create more realistic actions.
Currently the emotional system contains three emotions: happiness, anger and fear which all play an important role in how the agent makes decisions. The research report details the use of a more complex emotional system of the likes of Robert Plutchiks emotional wheel *(2001)* however, this was cut back as the amount of behaviors currently implemented removed the need for such a complex system. Further development of more behaviors would require more emotions and therefore with more development time a more complex emotional system would be more successful.

The needs network drives the agent's idle behavior, as chosen by the master network. The network evaluates their current needs (such as hunger, boredom, etc.) and a prioritized list of actions is created.

Initially, the network would only choose one action to complete and would then wait until that action was completed to choose another. As highlighted by the supervisors during the mid-progress demo, this caused issues with expandability as if the agents couldn't complete this action, say there was nowhere free to eat, they couldn't complete the task and would stand idle until they could. This was a major hurdle in the development of this project as the system needed to be flexible if it ever were to be implemented into an actual game. To resolve this issue, instead of evaluating the best result from the network, each output was given a priority based off the value of each node. A simple function was made to check if the most prioritized action was available, If it was it would be added to the agents queue of actions, if not then the next most important action would be evaluated. This queue could be interrupted by manually inserting actions of a higher priority at the front to allow flexibility for other behaviors, much like how system interruptions work on modern computers. This queue also allows for greater optimization as unlike other networks, this network isn't constantly feeding through inputs if the queue isn't empty, saving system resources.

The final network controls the agents combat. This network is the simplest due to the need for rapid reactions in a combat scenario, it takes a list of agent's stats including health and stamina and decides whether they need to attack, dodge or wait for stamina to recover.

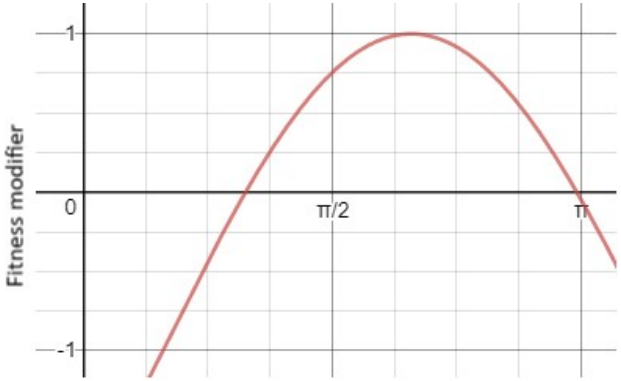
Training the networks

To train networks, a script was developed that allows the developer to select the behavior they wish to train using an enum and polymorphic behavior, how long to train for and how fast to simulate game time. At the end of each training round the results are then evaluated, with the best fitted network being serialized and saved as a binary file.

The project ran into major issues training the needs controller as the expected values were more abstract, bots were expected to fulfill their needs but the order and importance placed on each need was largely up to the network. This caused a large delay in the

development of the project as a heap of development time was spent simply figuring out how to train each network.

Lots of different methods for adding fitness were explored such as job completion rate, average of needs, most active etc. but eventually it was found that by calculating a weighted happiness function, where hunger was awarded points based off a sine wave, and then adding that to the networks fitness the training results showed lower error rates.



Hunger as Pi

This was later mitigated by the inclusion of an intensive training option where the developer can spawn a near infinite amount of networks to be trained in parallel, however it doesn't account for how difficult it is to see where errors lie in a network even with the inclusion of the network visualizer.

In retrospect, developing a more fleshed out training script before attempting to train networks would have saved a considerable amount of time and further time should have been allocated developing a better method of visualizing networks.

Developing additional behaviors

A small portion of development time was spent developing additional behaviors to help improve player immersion whilst playing.

An event system was added using a web of events and delegates controlled by an event manager. This manager alerts bots when certain actions have taken place near them such as a fight or death.

A faction system was then layered on top of this system to control how each bot responds. There are 3 factions, neutral, guards and barbarians and as expected guards will attempt to keep peace in the tavern, barbarians will be more sensitive to and more likely to cause violence and neutral will be more likely to observe from a distance or run away.

Factions also dictate what interactable a bot may use, this allows the developer to assign different jobs and hobbies to different factions.

Evaluation of user feedback

Throughout the development of this project, 5 play test sessions were carried out with the same participants to help gauge user immersion and guide further development.

Overall these playtests were unsuccessful due to the small sample size, as the participants needed to remain the same, and as participants were more eager to focus on smaller details irrelevant to the project, rather than focus on NPC behavior.

Each playtest saw a measurable increase in player immersion as key milestones in development were hit such as adding player interaction, adding bot on player interactions etc. however each round came with a new series of bug fixes, minor tweaks and suggestions that, overall distracted the development from the NPCs.

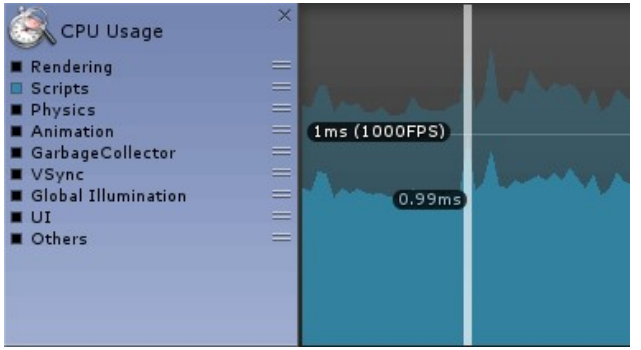
If anything, this playtest helps to demonstrate how player immersion can't be solved by developing one robust system but comes from how the many complex systems in a game interact with each other. Players tended to be more forgiving towards unrealistic NPC behavior yet stated all immersion was lost when bugs or other annoyances got in the way such as character controls or hitbox issues that reminded them that what they were experiencing was just some game.

Discussion of outcomes

This project proves that neural networks are a viable option for developing functional video game NPCs at a low system cost. Whilst the current implementation of the agents lacks polish and additional behaviors, all the core behaviors are present. The bots can react to player actions adequately and can also behave realistically whilst idle.

Tested on 2 systems, a desktop pc and a mid-tier gaming laptop; With 10 agents in a unity scene the networks take up a fraction of the system resources, which were mostly allocated to the physics and rendering segments.

With 500 training nodes all running through the networks once a frame the script time was low enough, 6.2ms on the desktop, to push framerates over 100fps. These framerates are well over the standard 30 to 60fps perused by current video game developers and leave plenty of headroom for other game systems.



FRAME TIME FOR 10 AGENTS WITH 3 BEHAVIOR SCRIPTS

The framework allows for a streamlined process of developing networked behaviors quickly. New behaviors can be created by inheriting from the AiBehavior.css base class, the network can be customized completely to the developers needs from the unity inspector and from there the outputs can be evaluated and relevant actions can be coded.

Beginning to train new behaviors is just as easy, provided the developer implements a robust strategy for applying fitness. Once a new behavior is made it can be added to the enumerator of trainable scripts and after the developer has coded a method of evaluating the network it can be trained for a customizable length with any number of agents.

Once trained these behaviors can be introduced to the system by adding them as an output to the master network and then training the master network how to

use these new behaviors. This allows developers to easily add and remove behaviors without impacting the rest of the system where as if one big network was used it would be impossible to do so without causing major problems.

Splitting up the neural network into separate networks and controlling them with one master network proved more effective with training and debugging as well. If a behavior needed more training it could be done without the complexity of training a deep network and if the agents started to show errors it was easier to isolate where the error was happening.

Given more development time, it would be better if this project featured its own API that supported scripting, via CSScript, instead of relying on the unity inspector and a good understanding of c#. This would allow a custom build to be shipped to designers that is more light weight than unity and allows for behavior creation in a more controlled environment through a more user-friendly language. This was out of the projects scope however, and the framework implemented is a step in the right direction towards accessible neural networks.

The network visualizer is helpful for understanding how the networks are working and can be used to help identify where errors are happening in evaluation. However, without a robust understanding of the network to begin with it can still be difficult to figure out what is causing the error and how to fix it. Even with the network visualizer, adjusting the networks size and fitness is still carried out by trial and error which is a problem this project was unable to solve.

Training the networks is still a major hurdle. Whilst the scripts provided can get close to the desired behavior there was still some erroneous behavior that couldn't be solved. This can cause problems with staging events as well as immersion as in very few occasions the networks can still act in unpredictable ways.

The video game Halo 2 *(Bungie, 2004)* has been praised largely for its AI and in a report by Robert Valdes on how Halo 2's AI works, Chris Butcher is quoted saying "What you want is an AI that is consistent" and "We try to go for predictable actions but unpredictable consequences".

This error, coupled with the extended development time this project had to undergo, could be a couple of reasons why video game developers have stayed away from neural networks in the past.

Conclusion and recommendations

In its present condition, the system is fully functional within the original scope. However, more time with this project to develop additional behaviors would allow the agents to act in a wider variety of ways. This would help both with gameplay and immersion.

Developing better training practices is highly recommended, such as the introduction of backpropagation and weight decay, as this would be beneficial to the system and overall development flow. Narrowing the error to as little as possible is a crucial step towards getting consistent behavior and doing so with the techniques implemented here would be to time consuming.

The use of one network controlling multiple other networks worked well and should be investigated further within different applications as it allowed finer control over the network as a whole.

Neural networks have proven themselves to be a resource effective model for video game AI and should be a serious consideration for developers looking for an alternative to behavior trees and finite state machines. With the growing interest in deep learning amongst developers, this could be an opportune time for such developments as they can deliver good results.

References

Bungie. (2004). *Halo 2*. [Video Game] Microsoft Corporation. Available: http://halo.bungie.net/projects/halo2/ Accessed: 06/04/19.

Johnson, O. (2018). *Organic Non-Playable Characters using neural networks: research report.* Unpublished, University of the west of England.

Kim, Y.S. (2008). *Comparison of the decision tree, artificial neural network, and linear regression methods based on the number and types of independent variables and sample size*. Expert Systems with Applications, 34(2), pp.1227-1234.

Krogh, A. and Hertz, J.A. (1992). *A simple weight decay can improve generalization*. In *Advances in neural information processing systems* (pp. 950-957).

Nwankpa, C E. Ijomah, W. Gachagan, A. Marshal, S. (2018). *Activation Functions: Comparison of Trends in Practice and Research for Deep Learning.* arXiv.org. arXiv:1811.03378. Available: https://arxiv.org/abs/1811.03378 Accessed: 06/04/19

Rasmussen, J. (2016) *Are behavior trees a thing of the past?* [Online] Gamasutra. Available: http://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php. Accessed: 07/04/19

Plutchik, R. (2001). *The nature of emotions: Human emotions have deep evolutionary roots, a fact that may explain their complexity and provide tools for clinical practice*. American scientist, 89(4), pp.344-350.

Valdes, R. (2004). *The Artificial intelligence of Halo 2*. [online] HowStuffWorks.com. Available: https://electronics.howstuffworks.com/halo2-ai.htm Accessed: 06/04/19

Vehbi O, A. Karlik, B. (2011). *Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Networks*. International Journal of Artificial Intelligence And Expert Systems. 1. 111-122. Available: https://www.researchgate.net/publication/228813985_Performance_Analysis_of_Various_Activation_Functions_in_Generalized_MLP_Architectures_of_Neural_Networks Accessed: 06/04/19

Appendixes

Asset sources

The assets for the agent and their animations can be found at https://www.mixamo.com/#/ and are free for use.

The assets for the tavern can be found at https://assetstore.unity.com/packages/3d/environments/fantasy/simpoly-tavern-120464 and were purchased by the author of this project.

The script for the character shader can be found on the unity wiki here: http://wiki.unity3d.com/index.php/Silhouette-Outlined_Diffuse

Log Sheet

The log sheet can be found accompanying this document, or inside the github repository

Questionnaire results

The questionnaire results spreadsheet can be found accompanying this document, or inside of the github repo.