Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2008, Article ID 432365, 18 pages
doi:10.1155/2008/432365

Research Article

A Hybrid Fuzzy ANN System for Agent Adaptation in a First Person Shooter

Abdennour El Rhalibi and Madjid Merabti

School of Computing and Mathematical Sciences, Liverpool John Moores University, James Parsons Building, Byrom Street, L3 3AF, Liverpool, UK

Correspondence should be addressed to Abdennour El Rhalibi, a.elrhalibi@ljmu.ac.uk

Received 31 July 2007; Accepted 1 November 2007

Recommended by Kok Wai Wong

The aim of developing an agent, that is able to adapt its actions in response to their effectiveness within the game, provides the basis for the research presented in this paper. It investigates how adaptation can be applied through the use of a hybrid of AI technologies. The system developed uses the predefined behaviours of a finite-state machine and fuzzy logic system combined with the learning capabilities of a neural computing. The system adapts specific behaviours that are central to the performance of the bot (a computer-controlled player that simulates a human opponent) in the game with the paper's main focus being on that of the weapon selection behaviour, selecting the best weapon for the current situation. As a development platform, the project makes use of the Quake 3 Arena engine, modifying the original bot AI to integrate the adaptive technologies.

1. INTRODUCTION

With graphics at an almost photorealistic level and complex physics systems becoming commonplace, AI [1] is becoming more important in providing realism in games.In the past, game AI has used techniques that are suited to the restricted computational power available to it, but which still produced believable, but limited, nonplayer characters (NPCs) artificial intelligence (AI) technologies such as finite state machines (FSM) and rule-based systems (RBS). These techniques were also used due to their relative simplicity which did not require much development time to implement and were easy to debug, especially as the programmers generally did not specialise in AI.

With the increase in computational power available for AI, more complex techniques can be incorporated into games creating more complex behaviours for NPCs. The increasing importance of AI in games has meant that specialised AI programmers are becoming part of development teams bringing techniques from academia [2–5]. One of the areas of AI which has gathered interest is that of using machine learning techniques to create more complex NPC behaviours.

Most players develop styles of play that take advantage of certain weaknesses inherent in the NPC AI that become apparent as they become more proficient at the game. Once discovered, these deficiencies in the preprogrammed AI mean the competitive edge is lost making the player lose interest in the now all too easy game. If the NPC developed new tactics, adapting to the players style, uncovered their hiding places, or even discovered tactics that exploited weaknesses in the players' play, then this would add immeasurably to the enjoyment and prolong the life of the game [6–8]. The game should be tailored to provide a variety of challenges, and increasing the level of difficulty to deal with NPCs. This should of course be adjusted to the need of the player and be provided as a way to increase difficulty level in the gameplay without introducing unbeatable NPC which could lead to player's frustration. We are aware that it is a difficult issue to adjust the balance gameplay between performance and playability, and it will be the role of the game designer to deal with it. We are just interested in this paper in increasing NPCs' performance and adaptation.

This paper describes a method of implementing a first-person shooter (FPS) bot which uses machine learning [9] to adapt its behaviour to the playing style of its opponent. It

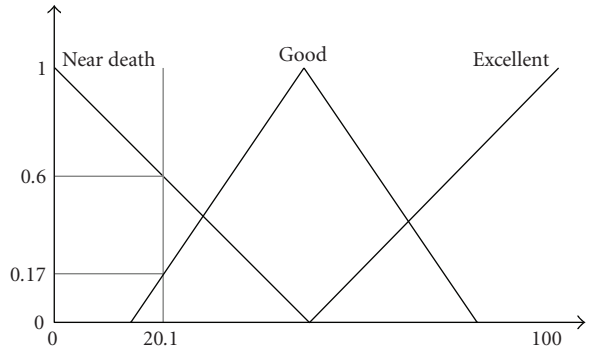

Figure 1: Sets defining the linguistic variable "health."

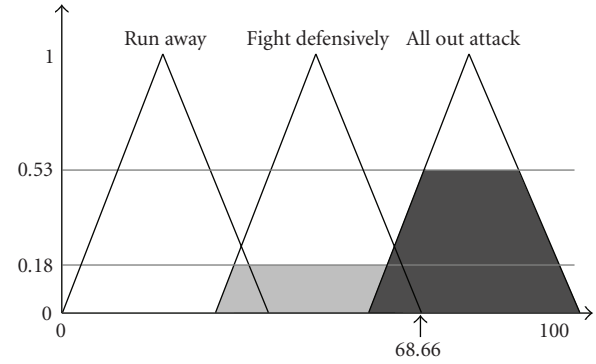

Figure 2: Sets defining the output variable "aggressiveness."

uses a combination of small, focussed, artificial neural techniques and predefined behaviours that allow the bot to exhibit changes in those behaviours to compensate for different player styles. For the purposes of this paper, only a single behaviour is focused on, that of weapon selection, although, in order to significantly adapt the play of the bot, multiple behaviours would use the system during a match.

2. FOUNDATION TECHNOLOGIES

2.1. Fuzzy logic

Whereas traditional logic describes concepts in terms of "true" or "false," fuzzy logic provides a way of describing values by the degree with which they correspond to a certain category within the concept, called DOM (the degree of membership) in a *set*. *Linguistic variables* are collections of sets that represent real concepts, for example the variable *health* could be made up of the sets near death, good, and excellent, as shown in Figure 1 [3, 10, 11].

Fuzzy logic provides a way of combining more than one variable to give a single output value, making decisions based on multiple criteria. For example, the aggression of a game character based on its health and the distance to the enemy.

Using fuzzy logic to derive decisions based on the input values for a number of variables requires that a sequence of steps to be carried out.

(1) Selection of sets that comprise the linguistic variables for the inputs and output. As with the input variables, the output variable consists of a number of sets defined by a range of values (see Figure 2). The difference is in the way they are used to calculate the final out value (see step (6), *Defuzzification*).
(2) Creation of fuzzy rules corresponding to the different combinations of inputs. The rules determine the output set for the different combinations of inputs. Using the previous example, if health is "good" and distance is "close," the rule could be "fight defensively."
(3) *Fuzzification* of the crisp inputs into fuzzy values giving the DOM for the inputs sets. Figure 1 shows the fuzzification of the crisp value 20.1 resulting in the DOM for each set of near death = 0.6, good = 0.17 and excellent = 0.0.
(4) Use *inference* to evaluate which rules are active based on the DOM of the input sets that make up that rule. Each combination of sets (for each input variable) is compared with the rulebase to determine which output sets are active. The DOM of the output set is determined, in this case, using the lowest DOM of the inputs (there are a number of methods for calculating the DOM). This results in a number of possible DOMs for each set of the ouput variable.
(5) Combine the multiple DOMs for each rule into the output sets using *composition*. This results in a single DOM for each of the output sets, as shown in Figure 2.
(6) *Defuzzification* of the output sets to give a single crisp value. This is done by calculating the centre of the area under the graph defined by the DOM in each set. Figure 2 shows an example for the output variable "aggressiveness" with doms of 0.18 for "fight defensively" and 0.53 for "all out attack." There are a number of methods, of varying complexity and accuracy, for determining the output value. One of the least expensive, in terms of computation, is the *mean of maximum* method, the equation for which is shown in (calculation of mean of maximum)

$$\frac{(\mathrm{RA}_{\max} * \mathrm{RA}_{\mathrm{dom}} + \mathrm{FD}_{\max} * \mathrm{FD}_{\mathrm{dom}} + \mathrm{AA}_{\max} * \mathrm{AA}_{\mathrm{dom}})}{(\mathrm{RA}_{\mathrm{dom}} + \mathrm{FD}_{\mathrm{dom}} + \mathrm{AA}_{\mathrm{dom}})}, \tag{1}$$

where

(i) $\mathrm{RA}_{\max}$ is the crisp value for the centre of "run away" set (where fuzzy value = 1);
(ii) $\mathrm{RA}_{\mathrm{dom}}$ is the fuzzy value for DOM for run away set;
(iii) $\mathrm{FD}_{\max}$ and $\mathrm{FD}_{\mathrm{dom}}$ are as above for "fight defensively" set;
(iv) $\mathrm{AA}_{\max}$ and $\mathrm{AA}_{\mathrm{dom}}$ are as above for "all-out attack" set.

2.1.1. Combs method

In traditional fuzzy logic, a rule needs to be defined for every combination of set for all the input variables. This can result in combinatorial explosion as the number of rules required grows exponentially according to the number of fuzzy sets for

each linguistic variable, that is, 2 variables each with 5 sets = 5^2 = 25 rules and 5 variables with 5 sets = 5^5 = 3,125 rules. This can make large systems slow, confusing, and difficult to maintain which, particularly in games, can make fuzzy logic impractical.

The main difference between Combs method [10] and the traditional method is in the way the rule set is defined. It builds rules based on each individual set's relationship to the output, considering one variable at a time, rather than creating rules for every combination of set for all the variables. This reduces the exponential growth of the number of rules into a linear growth, so that a system with 10 variables and 5 sets per variable would have 50 rules as opposed to 9 765 625 with the traditional system.

2.2. Artificial neural networks

There are many forms of artificial neural nets (ANN) of varying complexity which attempt to mimic the biological operation of the brain artificially by modelling the inter-connected cells that enable the brain to process information. The simplest form of ANN, the one used here, is the perceptron which is modelled as a single neuron with a set of weighted inputs mapping to a single output [7, 12–14].

The inputs (X_1 to X_n) to the perceptron can vary in number and value (binary or real numbers) depending on the application. Each input is multiplied by its corresponding weight (W_1 to W_n) and the weighted inputs are then added together, along with the bias, giving the output value. The bias represents a constant offset and can be treated as another input with a constant value of 1. By adjusting its weights, the perceptron can be trained to recognise specific combinations of inputs and generalise for similar inputs.

2.2.1. Training the perceptron

Initially, the perceptrons use a default value for all of their weights. This, in effect, means that the perceptrons will not have any influence over the effectiveness rating for the weapons, only the characteristics and fuzzy logic will affect the value. Once adaptation has begun, occurring every time there is feedback, the following training procedure is performed.

The training of perceptrons described here uses an incremental approach, computing the adjustments to the weights by way of the steepest descent technique [7]. The *delta rule* algorithm calculates the change required Δw_i for each weight w_i by taking the difference between the actual y and the desired t output and multiplying it by the input value x_i for that weight and by a, typically small, learning rate η; see (2), computation of required adjustment for each weight:

$$\Delta w_i = \eta(t - y)x_i. \tag{2}$$

The new weight for each input can then be found using the *steepest descent technique*, as shown in (3), computation of adjusted weight value, changing the weights as a result of feedback:

$$w_i \longleftarrow w_{i+}\Delta w_i. \tag{3}$$

The incremental nature of the algorithm means that it can be performed as the game is being played using feedback from actions performed.

2.3. Quake 3 arena

In order to implement the adaptable AI, a suitable environment was required that provided all the features of a FPS so that the capabilities of the bot can be tested. The Quake 3 Arena (Q3A) game engine [8, 15] provided the framework for the development of the bot AI. The new AI was integrated with the original AI, reusing many of its features. For more information regarding the Q3A engine, specifically in relation to the interface between the AI and the game engine, [8] provides the most comprehensive documentation.

Using the original bot AI provided the opportunity to be able to define the characteristics of the bot using text files that determine the style of play of the bots within the game. This proved helpful in the evaluation of the new AI, which was carried out in matches against the "standard" Q3A bots. By defining specific characteristics, situations could be set up that required the bot to adapt its behaviour.

3. SYSTEM DESIGN

A number of features are required of the adaptable AI system in order to achieve the aim of a bot that is able to adapt to the play of an adversary:

(i) to be able to play competitively from the first game (out of the box);
(ii) to adapt its behaviour as the game is being played (online);
(iii) to be computationally inexpensive.

The system makes use of the indirect adaptation technique, using a conventional AI layer to control the bot, with the adaptation AI modifying the behaviours of the bot in response to feedback according to its actions. This enables the bot to be competitive immediately by giving it a priori knowledge, as recommended by [1, 3, 5].

The adaptation system incorporates a number of components that combine to rate the effectiveness of a choice within a behaviour and adapt the value to reflect how well the chosen action performs in the game. Figure 4 shows how the separate elements are linked together to calculate the rating and allow adaptation to occur.

The system utilises a hybrid of two AI technologies: fuzzy logic and perceptrons. The fuzzy logic acts as the prior knowledge enabling the bot to perform in the game at a competitive level. The perceptron is used to facilitate the adaptation, acting as a form of memory enabling the bot to "remember" the effectiveness of actions in certain situations, altering its weights based on the feedback it receives from game. By using perceptrons, rather than more complex multilayer networks, the computational requirements are kept as low as possible whilst retaining the basic features of a neural net.

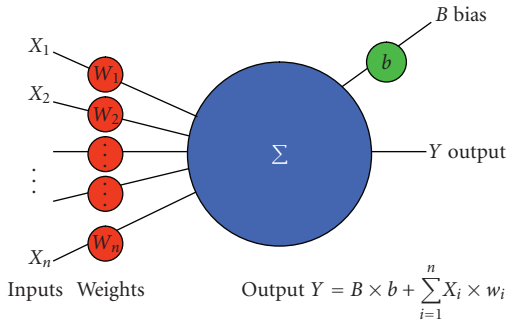

Figure 3: Architecture of a perceptron.

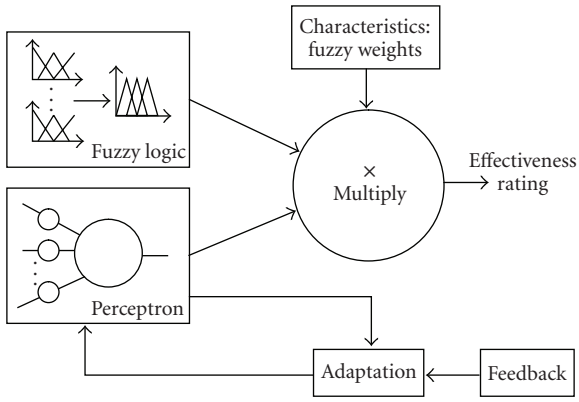

Figure 4: Adaptation system overview.

The system is composed of two main mechanisms:

(i) the effectiveness rating mechanism: used to determine how effective a certain choice is according to the input values;

(ii) the adaptation mechanism: used to change the effectiveness rating according to feedback from the game on how effective it was.

The effectiveness of a choice is predicted using a combination of the characteristics of the bot, defined in the characteristic files, a fuzzy logic component and a perceptron component. This system is used for each of the choices within a behaviour. The effectiveness is calculated by multiplying the outputs from the fuzzy logic component and the perceptron together with the characteristic for the choice.

The adaptation mechanism uses feedback from the game to determine how successful the choice was compared to the perceptron's predicted effectiveness of the choice. The feedback and output of the perceptron are then used to train the perceptron, increasing or decreasing the weight values according to the delta rule training algorithm discussed in Section 2.1.1. Adjusting the weights of the perceptron changes its output impacting on the effectiveness rating for the action, thus making it more or less likely to be used.

3.1. Adaptation of weapon selection behaviour

Modern FPS games, such as Quake 3 Arena, make use of complex 3D environments for their game worlds which, in turn, mean that the NPCs that inhabit them must have complex AI to interact with them, and the player, convincingly. Bots must be able to exhibit a number of behaviours, specialising in particular actions or strategies that contribute to the overall aim of winning the game. Due to the nature of the game, the aim being to kill the opponent more times than they kill you, the behaviours that would benefit most from adaptation are those that relate to combat with opponents, either directly or indirectly. One such behaviour is that of selecting the most effective weapon for the current situation. The rest of the paper will focus on this behaviour to demonstrate how the system can be applied.

The aim of adapting the selection of weapons is to enable the bot to change its weapon preferences depending on its success in particular situations. By changing the "effectiveness" or "fitness" of each weapon, by way of changing the perceptron weights according to the input values, different play styles can be adapted to.

The selection of information used as inputs for the system components is vital to their efficiency at performing actions in the game. The following sections detail the inputs for the fuzzy logic and perceptron components.

3.1.1. Fuzzy logic for weapon selection

Each of the weapons have a set of data defined for the variables (inputs) that represent the range of values that are significant to that weapon. The variables used for the fuzzy logic component are the following.

(i) Distance to the enemy. Each of the weapons available is better or worse at different distances. For example, the Lightning Gun has a maximum range of 768 and the Rocket Launcher risks splash damage when used at close distance. The distance needs to be broken down into fuzzy sets defining the effectiveness of each weapon for the distance range represented by that set.

(ii) Ammunition amount for each weapon. Each of the weapons have different firing rates. For example, the Machine Gun fires a shot every 1/10th of a second whilst the Railgun can only fire a shot every 1.5 seconds.Running out of ammunition in a fight means changing to another weapon, which takes time, reducing the damage that can be inflicted on the enemy. The ammunition level needs to be represented as a number of fuzzy sets spanning the maximum amount of ammunition (200). Each weapon requires a unique collection of set data defining the relative values of ammunition depending on their rates of fire—10 ammunition for the Railgun is different to the same amount for the Machine Gun.

3.1.2. Perceptron for weapon selection

Each opponent and game map have their own set of perceptrons as, for instance, different weapons can be more or less

effective depending on the map being played. Each weapon is represented by a perceptron, each having a unique set of weight values for that weapon. The inputs to the perceptron are the same for each of the weapons, although weapon specific inputs, that is, the amount of ammunition, will result in certain inputs having slightly different values. Some of the variables investigated are the folllowing.

(i) Distance to the enemy. By adapting the distance at which the weapon should be used, the weapons will increase/decrease the range at which they are used. An example of a use for this is if the enemy is very aggressive and continues attacking when low on health. Normally the rocket launcher may not be used at close range due to the danger of splash damage, selecting a less damaging, and less successful, weapon instead. The system could adapt the lower range of the Rocket Launcher so that it is selected over the less useful weapon, incurring damage to the bot but also killing the opponent with one shot.

(ii) Ammunition. The amount of ammunition for each weapon can be adapted to make use of weapons that the opponent is more susceptible to be damaged by. Used by the fuzzy logic component, it has a large influence on the selection of weapons and by adjusting the ranges the bot will be more likely to stick with a successful weapon even though the ammunition is running low in the hope of killing them before the weapon needs to be switched.

(iii) Visibility of enemy. It would be useful to adapt the weapon selection based on the visibility of the enemy so that areas that contain obstacles, creating cover for the enemy to hide behind, can influence the selection to favour weapons that have splash damage enabling the weapon to inflict damage around corners.

(iv) Height difference. Like the visibility of the enemy, the height difference between the bot and its opponent could be used to influence the use of weapons that have splash damage. If the opponent is below the bot, it can aim at the floor near to the enemy, hitting with radial damage. If the opponent is higher, making it difficult to hit them with splash damage, then Grenades can be launched onto the higher area or more precise weapons can be used. Adapting the relative strengths of weapons when there is a height difference will select the most effective weapons in those situations.

3.2. Feedback for perceptron training

The feedback that is used to train the perceptrons for the weapon selection behaviour is focused on the criteria of causing as much damage as possible whilst avoiding inflicting damage to oneself. This means that it must account for a combination of health lost by the enemy and by the bot itself as a result of its own attack (not damage sustained from enemy attack). A timed aspect is required to allow for the different characteristics of each of the weapons (firing rate and damage per shot) and enable the performance of the weapons to be compared. To reward weapons that have the capability of "finishing off" enemies (e.g., Railguns are very good at one-shot kills) a bonus is also required when the opponent is killed by the current weapon. This increases the overall feedback value thus increasing the weight values when training.

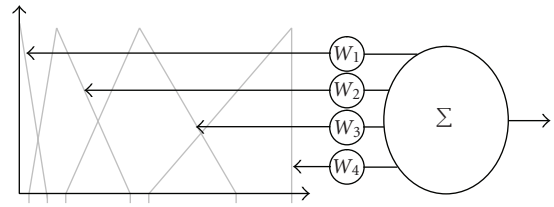

Figure 5: Categorisation of perceptron inputs.

3.3. Categorisation of perceptron inputs

Due to the linear nature of perceptrons (they are unable to handle nonlinear problems) difficulties arise with inputs that can be effective at high and/or low values. One problem is that higher input values will always output higher ratings and so if the lower input values are better (i.e., correspond to a more effective weapon use), these inputs values will not be able to characterize this effectiveness. Another problem is if the weapon is more effective with an input value that is in the middle range, such as the grenade launcher that can cause splash damage close-up but has a limited range. This is compounded by the training mechanism that changes the weight of the input depending on the input value. This means that high values will always be penalised more than low values.

To allow adaptation to occur independently for different levels of the same input, its range of values needs to be categorised into ranges. The fuzzy logic component can be utilised to achieve this. It is able to take a single value and assign a DOM for each of the categories by fuzzifying the input value. Each of the categories represents an input into the perceptron, splitting the single input value into the number of sets that represents that input, as shown in Figure 5. The advantage of this approach is that it will categorise the input into continuous values for each set, rather than the imprecise method of just determining whether the value is in a category or not. It also uses functionality that is already within the system so no new component needs to be developed.

One of the main advantages with using fuzzy logic to categorise the input value is that the fuzzy values will represent the DOM for the set. This means that the low category can have a high input value and the high a low value—0 (100% membership of the low category) could input a 1. When training the perceptron, this will be useful in correctly rewarding or punishing the value range responsible for the action selected. Another advantage is that the maximum membership of a set is 100%, in effect normalising the input values for each set to a value between 0 and 1. Although the input value can be in multiple sets, the combined fuzzy values will approximate 1 (fuzzy values need not add up to 1 but are usually near to this value, depending on the set data).

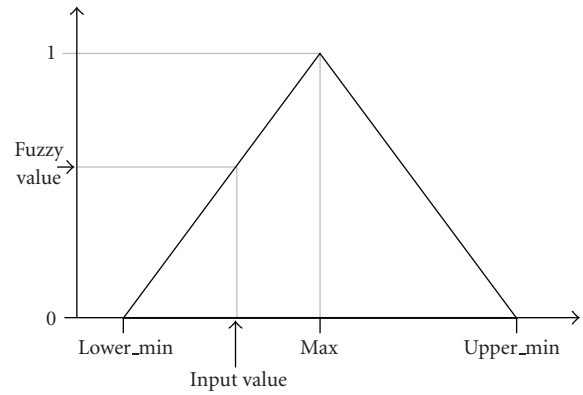

Figure 6: Data definitions for a fuzzy set.

4. DISCUSSION ON IMPLEMENTATION

4.1. Fuzzy logic component design

The fuzzy logic component is comprised of three parts. The first represents the fuzzification process, turning the crisp values of the inputs into fuzzy values of the degree of membership in the sets for that input, along with the rule associated with those sets. The second part, representing the composition process, calculates the degree of membership of the output sets based on the rules and fuzzy values calculated in step 1. The last part, the defuzzification process, determines the output value for the component.

It was decided during the design of the fuzzy logic component that each of the input variables should have four sets. In keeping with Combs method, the output sets should have the same number of sets as the input variables and so they also have four sets. Using four sets to define a variable provides a balance between sufficient detail to describe the inputs properly without making the component overly complicated.

4.1.1. Fuzzification process

The fuzzification of a crisp value into a fuzzy value or values is achieved using the data that defines the membership function for each set in the input variable. The data for the sets is defined in an array that is loaded during initialisation of the game. Figure 6 shows a typical representation of a fuzzy set with the data that represents that set labelled on the x-axis. The two "min" values represent the upper and lower limits of the set at the point where the degree of membership in that set equals 0. In between these two points the degree of membership will be greater than 0 with the maximum fuzzy value, 1, marked by "max."

Using this data, any input value that is within the "min" range can have a fuzzy value calculated for it using linear interpolation. The point at which the input value crosses the line defining the edge of the set (joining lower_min and max) can be determined by finding the difference between the input value and the max or min (depending on which is highest) and dividing it by the difference between the max and min values that the input value bisects. This results in two equations depending on whether the input value is between the lower_min and the max, or the upper_min and the max. Equation (5) presents calculation of the fuzzy value when max < input < upper_min,

$$\text{fuzzy value, fv}_l = \frac{(\text{input} - \text{lower\_min})}{(\text{max} - \text{lower\_min})}. \tag{4}$$

Equation (1) presents calculation of the fuzzy value when max < input < upper_min,

$$\text{fuzzy value, fv}_u = \frac{(\text{upper\_min} - \text{input})}{(\text{upper\_min} - \text{max})}. \tag{5}$$

Along with the fuzzy value, the rule associated with the set is also required so that the fuzzy value can be applied to the correct output set during composition. The rule is represented by the array location of the output set, that is, output [0] is bad, output [1] is average, [2] is good, and [3] is excellent. The fuzzification of the input value will result in 2 or 4 values being returned; the input usually has a degree of membership in 2 sets, 1 set only if the degree of membership is very high (>90%), so the fuzzy value and rules for both sets must be returned.

Pseudocode for fuzzification process

Algorithm 1 shows how the fuzzification process is calculated,calculating the fuzzy value if within a set, 1 if equal to the max value, and 0 if outside (also setting the rule to −1 to mark it as unused).

The fuzzification function is designed so that, as well as being used in the fuzzy logic component, it can also be used for single inputs when categorising input values for use with the perceptron.

4.1.2. Composition process

The composition of the fuzzy values and their associated rules into the degree of membership for each of the output sets is done by taking the MAX fuzzy value associated with each of the output sets calculated for all the inputs. This returns an array of values for each of the output sets that can be used to determine the output value in the next part, defuzzification.

Pseudocode for composition process

The process of composition is quite straightforward, simply putting fuzzy values into an array representing the output sets if the value is greater than the one currently in there. Algorithm 2 shows how this process can be accomplished.

(i) Fuzzy_values[] is an array containing the fuzzy values and rules calculated in the fuzzification process where [0] to [3] is the fuzzy values and rules for an input.
(ii) Output_array[] is an array that contains the MAX values for each of the output sets where [0] is set 1, [1] is set 2, and so forth.

```
for each of the inputs
  for each of the sets
    if set_lower_min value < input_value < set_max_value
      fuzzy_value = (1 /
                  (set_max_value − set_lower_min_value))*
                          (input_value − set_lower_min_value)
      output_rule = set_rule
  end of if
  else if set_max_value < input_value < set_upper_min_value
        fuzzy_value = (1 / (set_upper_min_value −
          set_max_value))*(set_upper_min_value − input_value)
        output_rule = set_rule
  end of if
      else if input_value = set_max_value
            fuzzy_value = 1
            output_rule = set_rule
      end of if
            else fuzzy_value = 0
                 output_rule = −1
            end of else
  end of for
end of for
```

Algorithm 1: Pseudocode for fuzzification process.

```
for number of sets (i)
    if fuzzy_values[i + 1] > −1
      if output_array [fuzzy_values[i + 1]] <
                        fuzzy_values[i]
       output_array [fuzzy_values[i + 1]] = fuzzy_values[i]
      end of if
      increment i by 2
end of for
```

Algorithm 2: Pseudocode for composition process.

```
for number of output sets (i)
    mean of max top + = output_data[i]* output_array[i]
    mean of max bottom + = output_array[i]
end of for
mean of max = mean of max top/mean of max bottom
```

Algorithm 3: Pseudocode for defuzzification process.

This process will be done for each of the inputs in turn; finally getting the MAX values for each of the output sets after all the inputs have been processed.

4.1.3. Defuzzification process

The defuzzification process takes the array generated by composition and returns the final crisp value that is used to determine the rating of the action. It uses the mean of maximum method of defuzzification to calculate the single output value, based on the max values of each of the output sets and the fuzzy values for that set.

Pseudocode for defuzzification process

The defuzzification process uses the output array (Output_array[]) and the stored data for the output sets (Output_data[]) to calculate the output value.

4.2. Perceptron component design

The perceptron component does not require a separate function to calculate its output value. A perceptron is a combination of a multiplication for each of the inputs and its associated weight followed by a sum of all the multiplications. The perceptron calculation is incorporated into the functions for each behaviour, as specific information for the inputs is required in each case.

4.2.1. Pseudocode for perceptron component

The code shown in Algorithm 4 shows the design of the perceptron component that is incorporated into each behaviour. A single function is not used for simplicity, as the requirements of each behaviour regarding the number of inputs to the perceptron and the information to get differ enough to warrant separate functions. Each of the functions follows the same design, just using different information.

```
get input values specific to behaviour
for each behaviour action (a)
        for each input to perceptron (i)
            output[a] + = input_value[i]* weight[i]
        end of for
end of for
```

Algorithm 4: Pseudocode for perceptron component.

```
SelectBestWeapon() function
        fzEval = EvalFuzzyWeapons()
        pEval = EvalPerceptWeapons()
        fzVal = GetWeaponFuzzyVals()
        for each weapon
            eval = FzEval* PEval* FzVal
            if weapon is current weapon
               eval *= 1.1
            end of if
            if eval is highest value
                best weapon = eval weapon
               end of if
        end of for
        return eval weapon
end of function
```

Algorithm 5: Pseudocode for weapon selection function.

4.3. Weapon selection behaviour design

The weapon selection behaviour is handled by a single function from which the fuzzy logic and perceptron evaluations are called and all the relevant input data extracted. This function replaces the original Q3A function *trap_BotChooseBestFightWeapon()* for the adaptable bot in the *BotChooseWeapon()* function. Algorithm 5 shows the structure of the function and how the effectiveness for each weapon is determined.

The variables fzEval, pEval, and fzVal are all arrays that are filled with the data for all the weapons stored in the same array locations in each, that is, array location [0] contains all gauntlet data, [1] machine gun data, and so forth. The fzEval array holds the fuzzy logic evaluations for each weapon, pEval the perceptron evaluations, and fzVal the fuzzy weapon weights defined in the characteristic file.

Once all the data has been calculated and collected, the overall evaluation of the weapon is calculated by multiplying all the values from the 3 arrays for each weapon together to determine the effectiveness rating for each weapon. If the weapon is the currently held weapon it is given a bonus so as to prevent circumstances were the evaluations of 2 weapons are very close and slight changes in situation cause constant changing between weapons. Each time the rating of a weapon is calculated, it is compared with the previous weapon and the one with the highest value is recorded. At the end of the calculations, the weapon with the highest rating is returned.

4.3.1. Weapon fuzzy data

The data that defines the fuzzy sets for the input variables, distance and the ammunition level, is integral to the performance of the fuzzy logic component. The values that were used to define the sets were carefully chosen after careful observation of a number of games, with collection of the information displayed on the screen for accurate appraisal.

Distance variable data definition

The set data for the distance variable is the same for all of the weapons; except for the gauntlet which is a special case in that it can only hit an opponent when in direct contact with it. The distance variable is split into 4 categories:

(i) close,
(ii) medium,
(iii) far,
(iv) very far.

It was decided that, with the exception of the gauntlet, each of the weapons generally could be rated according to the same ranges defined by the 4 categories and so just 2 sets of fuzzy sets needed to be used: one for the gauntlet and the other for the rest of the weapons.

The fact that the gauntlet needs to be touching the opponent to make a hit means that only 1 of its sets has to be considered when designating its min, max, and rule values; the *close* set. Anything outside the *close* set means that the weapon cannot damage the opponent and so is given the worst rule (bad). The major consideration was determining the "killing range" of the weapon, when it would be deemed usable. Table 1 shows the value ranges for the sets (far and very far not shown for readability as they also have rule of 0); minL and minH being the lower_min and upper_min values.

The gauntlet data shows that it is only usable within a distance of 5 units from the enemy. At any other distance, it will output the lowest value possible. This means that, combined with its low weapon weight, all other weapons should be chosen before it.

The set data for the other weapons needed to take into account the different ranges of the weapons, some being good at close range but almost useless at long range, others being good in the middle ranges but less so when close or very far away. It was decided that the upper range for distance was 1500 units, anything above this being set to 1500, as this was near the limit of the bot's awareness.

Figure 7 shows the data values used for each of the sets' ranges, the rules needed to be defined for each of the weapons separately as each has its different strengths and weaknesses (which can be seen in Table 2).

The data values chosen for each weapon were derived from observation of games being played and data collection from personal experience from playing the game. Access to the fuzzy logic component of the Q3A AI was not possible, so the values are a "best guess" as to the values used. The rules represent the weapon's ability to damage at each distance range, the lowest being 0 (bad) with the best being 3 (excellent).

Table 1: Distance fuzzy set data for gauntlet.

Close				Medium				Far				Very Far			
minL	minH	max	rule	minL	minH	max	rule	—	···	—	—	—	···	—	—
0	5	0	1	0	200	20	0	—	···	—	—	—	···	—	—

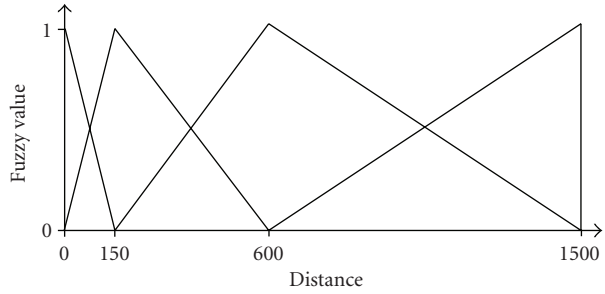

Figure 7: Fuzzy set data for weapon selection distance input.

Table 2: Fuzzy rule set for weapon selection distance.

	Rules			
	Close	Medium	Far	Very far
Machine Gun	1	1	1	1
Shotgun	3	2	1	0
Grenade launcher	1	2	2	1
Rocket launcher	1	3	2	2
Lightning gun	2	3	2	0
Railgun	1	2	3	3
Plasma gun	3	3	2	1
BFG	3	3	3	2

A number of factors were taken into account when defining the data values. For instance, the rocket launcher is given a *close* rule of 1 (average) because it has a large splash damage radius which will cause damage to the bot if used at close range. Its long range effectiveness is marked down due to the relatively slow speed of rockets which means that they are easy to avoid given the time that longer ranges afford. As another example, the railgun is only rated at 1 for close range due to its long reload time between shots, even though a single hit could kill the opponent.

Ammunition variable data definition

Each of the weapons needed to have its own set of data defined for the amount of ammunition input variable. All the weapons have different firing rates that determine the range for each of the ammunition sets. The only data that is constant between all the weapons, except for (again) the gauntlet which does not use ammunition at all, was the maximum amount of ammunition that could be available which is 200.

In the same way as was done for determining the values used for the distance sets, careful observation of the game lead to the selection of the set data, depending on the rate of fire of each weapon and the amount that is available when the weapon is first picked up. The amount of ammunition available on the map is not taken into account for this iteration of the project, although it could be a future improvement.

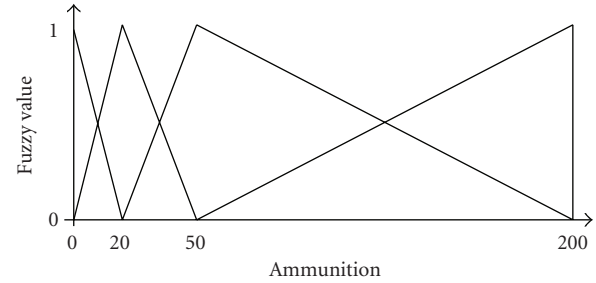

Figure 8: Fuzzy set data for shotgun ammunition.

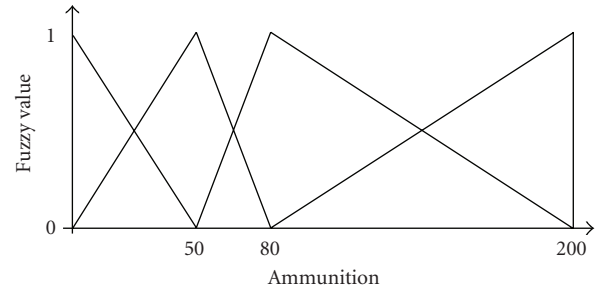

Figure 9: Fuzzy set data for plasma gun ammunition.

The set data for the ammunition for the shotgun and the plasma gun are shown in Figures 8 and 9. the diagrams show how the shotgun and plasma gun differ in the way they use ammunition. The shotgun has a slow reload and so the ranges of the sets are smaller and they are grouped near to 0 to represent the fact that, due to its slow firing rate, smaller amounts of ammunition are considered good. The plasma gun, in contrast, has a high firing rate which can be seen by the way the sets are more spaced out with larger ranges making larger amounts of ammunition more important than for the shotgun.

Output data definition

The output sets are defined by a single value that represents the max value of the set. This is the only value that needs to be specified due to the defuzzification method used; that of mean of maximum which only uses the max value to calculate the output value.

As can be seen in Figure 10, the output sets are not equally spaced from 0 to 100, each set is assigned a value to bias the output for that set. By doing this, the better rated sets produce much higher output values than the *bad* set, which can offset weapons with much higher characteristic

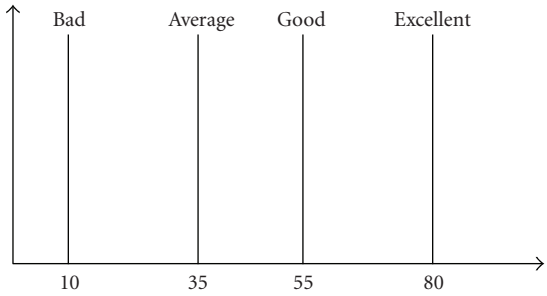

Figure 10: Fuzzy set data for weapon selection output.

```
EvalFuzzyWeapons()
        GetinputValues()
        for each weapon
                if have weapon and ammo
                        Fuzzify()
                        Composition()
                        Defuzzify()
                end of if
        end of for
        return array of weapon evaluations
end of function
```

Algorithm 6: Pseudocode for weapon selection fuzzy logic evaluation function.

preferences. This was done so that there would be very little chance of selecting weapons in circumstances were they are useless, such as when the enemy is out of range and so no damage can be inflicted.

4.3.2. Weapon selection fuzzy logic evaluation function

This function is called from the *SelectBestWeapon()* function in order to get the fitness values for all the weapons according to the fuzzy logic component. From this function, the three parts of the fuzzy logic component are called, as can be seen in Algorithm 6.

First, the input values for the inputs are extracted and then, if the bot has the weapon in its inventory and also has ammunition for the weapon, the fuzzy logic component is run on them (calling the three parts in turn—*Fuzzify, Composition* and *Defuzzify*). An array is returned at the end that holds the evaluation of all the weapons by the fuzzy logic component.

4.3.3. Weapon selection perceptron evaluation function

The function that evaluates the weapons using the perceptron component simply uses the code explained in Section 4.2. For the weapon selection behaviour, like the fuzzy logic evaluation function, the evaluation is only run if the weapon is in the bot's inventory and there is an ammunition available for it.

The output of the perceptron is normalised to a value between 0 and 1 by dividing it by the number of inputs into the perceptron. Each of the inputs is normalised also and by taking the maximum number of inputs that can be used at one time, the perceptron output can be scaled appropriately. The maximum number of inputs takes into account the categorisation of some inputs, which means that the four inputs resulting from categorisation actually represent 1 input as, at most, there will only be two active at one time and their combined values will always be approximately 1.

Perceptron inputs for weapon selection

The inputs used for the perceptron are the following:

(i) distance to enemy: categorised to 4 inputs;
(ii) ammunition: categorised to 4 inputs;
(iii) health;
(iv) visibility: categorised into 2 inputs, visible or not visible;
(v) height difference: categorised into 2 inputs, above or below;
(vi) aggression.

The inputs used were determined from assessment of those specified in the analysis section during development. Some of the input variables proposed were discovered to be superfluous to the adaptation process and others had to be tailored to the limitations of the Q3A source code. The quad input was discarded due to the test map not including the power-up, although this could be implemented if other maps were to be used. The visibility input was originally intended to be a measure of obstructions in the area but it was discovered that the function that returns the visibility of the enemy only returns the values 0 and 1 on the test map (although on maps that include fog it returns values between 0 and 1). This meant that it is now only used to determine whether the bot can see its opponent or not and is used choosing weapons that have splash damage that can still hit the enemy even when they are hidden.

The visibility and the height difference inputs are categorised simply into 2 inputs, each being either a 1 or 0 depending upon whether they are visible or not or whether the enemy is above or below. If in one category that input value is 1 and the other 0 and vice versa. This enables the perceptron to gain positive or negative reinforcement for situations when the enemy is not visible (1 in not visible input) or when they are (1 in visible input).

Fuzzy set data for input categorisation

The categorisation of the inputs distance and ammunition are needed to scale the inputs,so that each weapon would have the same output from the perceptron for the same relative inputs. This presented no problems for the distance input as the distance to the enemy is the same for all the weapons. The ammunition input required the amounts to be scaled appropriately for each of the weapons individually so as to represent the characteristics of each, and give

roughly the same perceptron output for the same relative level of ammunition. In order to do this, the input set data used for the ammunition by the fuzzy logic component was used.

4.3.4. Feedback for weapon selection

The feedback for the weapon selection behaviour required careful thought on how to deal with the varying characteristics of the different weapons. The feedback needed to be a measure of the weapon that dealt the most damage to the enemy, whilst also taking into account any damage done to the bot. This meant that a timing element needed to be introduced to account for the different firing rates of the weapons. Another consideration was that some projectiles take time to travel to their target whilst others strike immediately. The weapons fall into one of two categories that affect the feedback.

(i) Instant shot: those weapons whose projectiles impact immediately upon firing (having a speed of 0). Weapons in this category are
   (a) gauntlet (W_1),
   (b) machine gun (W_2),
   (c) shotgun (W_3),
   (d) lightning gun (W_4),
   (e) railgun (W_5).

(ii) Missile: weapons that fire projectiles that have a finite speed and take time to impact. Weapons in this category are
   (a) grenade launcher (W_6),
   (b) rocket launcher (W_7),
   (c) plasma gun (W_8),
   (d) BFG (W_9).

The method developed to record the input data (distance, ammunition, etc.) when a firing event occurs (when the weapon is fired). When the projectile impacts, hitting either opponent or environment, it calls the feedback function which determines the time between impacts and the damage inflicted on the enemy and sustained by the bot. This deals with the problems of calling feedback just after a fire event (would not know damage for missiles) or calling feedback when the missile impacts (need to know inputs when projectile fired). To implement this method, each fired projectile needs to be tracked after it is fired, so that it can be related to the correct inputs when it impacts. The time between impacts is only measured when in combat with the enemy, not including time between combat when navigating the map. The previous impact time is reset each time the bot finds a new enemy.

Based on the requirements of the feedback, (6) (weapon selection feedback equation) shows how the feedback value is calculated from the damage to the enemy, the damage to the bot, the time between projectile impacts and a bonus. The bonus is given when the enemy is killed by the current weapon:

$$\text{feedback} = \frac{\text{damage2enemy} - \text{damage2self}}{\text{last\_impact\_time} - \text{prev\_impact\_time}} + \text{bonus}. \tag{6}$$

The feedback value is a representation of the damage/second inflicted by the weapon. In order to be able to directly compare this value with that of the perceptron output value for the training it needs to be normalised to the range [0, 1].

Normalisation of weapon selection feedback

The normalisation of the feedback requires that it represent how well the weapon is performing. Every time the weapon is fired it produces feedback whether it hits or not. To fairly judge the performance of the weapons, the characteristics of each weapon was analysed so that a good measure of a good performance could be established. Table 3 shows the firing characteristics of each weapon.

Using this data, the average damage per second, taking into account all the weapons, was estimated to be 150 (rounding down to the nearest 10). This figure assumes that the weapons are 100% accurate. The average accuracy level of a "standard" bot, at skill level 4, was calculated to be approximately 25%. Taking this into account, the average damage the weapons inflict could be estimated at 150/4 = 37.5. This value could then be used as an "average" feedback score, one which should produce a normalised value of 0.5.

In keeping with observed accuracy, most of the time the weapon is going to miss, on average hitting only 1 in 4 times. This means that when training the perceptron with the missed shots, the reduction in value of the weights inflicted by misses should be compensated for when the enemy is hit inducing damage. The time between shots also has a bearing on the performance measure, but an average damage per second of 37.5, taking into account misses and hits, should train the perceptron to output 0.5 in those situations.

The upper and lower limits of the feedback value could also be estimated from the weapon data, using the max splash damage to determine the lower value and max damage per second to determine the upper. After investigating the source code it was discovered that splash damage only inflicts 0.5 of the damage on the attacker, so the minimum level is 50 (100*0.5 with no hit on enemy). The maximum damage per second is capped at 250 due to the excessive damage of the BFG, which is available only on a few maps and usually has little ammunition available for it.

In order to adequately normalise the feedback to give a value that could be used for training of the perceptron, fuzzy logic is used. This enables input values to output specific values, used to output 0.5 from an input of 37.5, and replaces what could be a mathematically complex function with a simple process.

Figure 11 shows the input fuzzy sets that are used to normalise the feedback value. The 4 sets span the range of values from −50 to 250 that the feedback falls between with set 1 centred on 0, the feedback for a miss, and set 2 centred on

Table 3: Weapon firing characteristics.

W#	Damage/projectile	Speed of projectile (0 = instant)	Splash damage	Splash radius	Fire delay (1/10 sec)	Damage/second
W_1	50	0	0	0	400	125
W_2	7	0	0	0	100	70
W_3	90	0	0	0	1000	90
W_4	100	700	100	150	800	125
W_5	100	900	100	120	800	125
W_6	16	0	0	0	200	80
W_7	100	0	0	0	1500	67
W_8	20	2000	15	20	100	200
W_9	100	2000	100	120	200	500

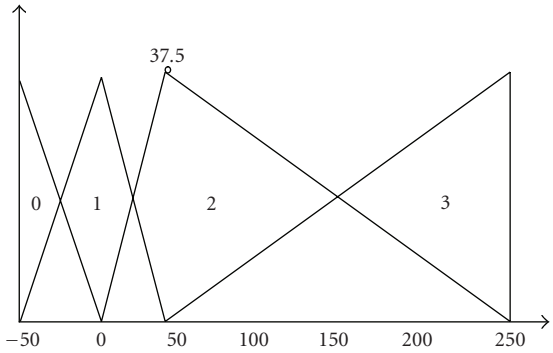

Figure 11: Feedback normalisation input fuzzy set data.

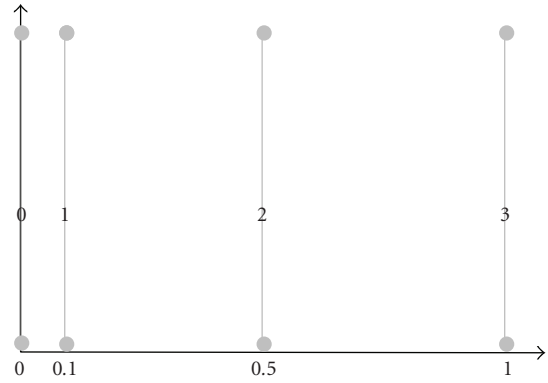

Figure 12: Feedback normalisation output fuzzy set data.

37.5, the feedback for an average hit. Each of the sets directly map to the output sets, shown in Figure 12, by way of their rule.

The output sets are set up so that they output specific values for specific input values. An input value of 37.5 will have a 100% membership in set 2 which maps to the output set 2, producing an output of 0.5. An input of 0 will have 100% membership in set 1, mapping to output set 1 giving an output of 0.1. Inputs of −50, the minimum feedback, and 250, the maximum, will output 0 and 1, respectively.

```
weaponFeedback()
  last_impact_time = current_time
  damage_enemy = damage to enemy
  damage_self = damage to self from splash from own
weapon
  if enemy killed
    bonus = 80
end of if
feedback_value = (damage_enemy − damage_self)/
                 (last_fire_time − prev_fire_time) + bonus
feedback_value = normaliseFeedback(feedback_value)
 trainPerceptron(feedback_value)
 prev_impact_time = last_impact_time
 last_impact_time = 0
end of function
```

Algorithm 7: Pseudocode for weapon selection feedback function.

Pseudocode for weapon selection feedback

The feedback function is called after every projectile impact, extracting the information required and then calling the function that trains the perceptron. Algorithm 7 shows pseudocode for the operation of the function.

First, the time is recorded so that the time since the previous impact occurred can be calculated. The damages done to the enemy and self are extracted and a bonus given if the enemy died from this projectile. The feedback value is then calculated and normalised before being passed to the perceptron training function to adjust the weights. Finally, the previous impact time is made equal to the current time and the last impact time is set to 0.

4.3.5. *Training for weapon selection*

The training of the perceptron is simply involved using the delta rule algorithm to update the weights of the perceptron based on the output of the perceptron when the projectile was fired and the feedback gathered from the projectiles impact.

```
trainPerceptron()
        for each input to perceptron (i)
            weight[i] + = learning_rate*
                        (feedback_value − output_value)*
                          input_value[i]
        end of for
end of function
```

Algorithm 8: Pseudocode for weapon selection training function.

Pseudocode for weapon selection training function

The function for training the perceptron is quite simple in operation as can be seen in Algorithm 8. The learning rate is set at a low value, which will be altered during evaluation in order to find a good balance between adapting fast enough to influence a game and slow enough so that isolated events do not interfere with the appropriate learning.

The function simply loops through the inputs to the perceptron, calculating the change required to the weight by multiplying the difference between the perceptron output and the feedback by the learning rate and the input value. This adjustment is then added to the weight, increasing or decreasing its value accordingly.

4.4. System development

The design section discussed the methods that are used to implement the adaptation system. This section will show how the designs for the components were realised using the Q3A engine [8, 15].

The adaptation system is composed of a number of functions relating to a specific component or behaviour within the system. All of the functions that were specially written for this system use the same prefix "*adapt*" in order to easily find and identify them within the many functions that make up the Q3A source code. The majority of the adaptation system functions and data structures are defined in the *ai_main.h* and *ai_main.c* files of Q3A engine [8, 15]. This provides access to and from the Q3A AI functions and data structures that this system integrates with.

The functions developed and a description of their purpose are listed below.

(i) *Fuzzy logic component functions:*

  (a) *AdaptFuzzify* fuzzifies the input data;
  (b) *AdaptComposition* calculates the degree of membership for the output sets;
  (c) *AdaptDefuzzify* calculates the output of the fuzzy logic component.

(ii) *Weapon selection behaviour:*

  (a) *AdaptSelectBestWeapon* is a main function that evaluates the weapons and selects the best available.

(iii) *Weapon selection behaviour fuzzy logic functions:*

  (a) *AdaptLoadWeaponFuzzyVals* loads the weapon fuzzy values from the characteristic files;
  (b) *AdaptAllocWepDOMS* loads the fuzzy set data;
  (c) *AdaptEvalFuzzyWeapons* main function that is called to evaluate the weapons using fuzzy logic.

(iv) *Weapon selection behaviour perceptron functions:*

  (a) *AdaptInitWPercept* initialises the perceptron weights and loads the set data for categorisation of inputs;
  (b) *AdaptEvalPerceptWeapons* is a main function that is called to evaluate the weapons using the perceptron;
  (c) *AdaptGetWinputvalues* extracts the data for the perceptron inputs;
  (d) *AdaptWeaponFeedback* calculates the feedback for the weapon selection behaviour;
  (e) *AdaptTrainWPerceptron* trains the perceptron based on the feedback.

(v) *Utility functions:*

  (a) *AdaptFlagFireEvent* records a firing event;
  (b) *AdaptOutputWepEvaluation* outputs the perceptron data to file.

4.5. Fuzzy logic component development

4.5.1. Data structures

The fuzzy logic component was required to fulfil a number of roles in the system: evaluation of actions, categorisation of inputs to the perceptron, and normalisation of values. In order to meet these requirements, two data structures were created to contain the information used to calculate the output from the fuzzy logic component.

(i) *adapt_DOM_t*: this structure contains a 2-dimensional array to hold the data for each set of an input variable.
(ii) *adapt_FL_t*: this structure contains an array of adapt_DOM_t structures for each input variable and an array containing the output set data. It also has variables for the number of sets and the number of input variables.

By splitting the 2 structures up, it allowed the use of a single adapt_DOM_t structure for categorisation of inputs, in which case the output set is not needed.

4.5.2. Loading fuzzy set data

The set data for the fuzzy logic component comprises a large amount of information; each input variable having a number of sets each requiring 4 values to define it (MINL, MINH, MAX, and RULE). For the weapon-selection behaviour, this meant 9 weapons each with 2 inputs each of which had 4 sets defined by 4 values, resulting in 288 (+4 for output set) values that needed storing. In order to facilitate the use of this

```
0 20 0 1 0 200 20 0 20 1500 200 0 200 1500 1500 0
0 10 0 0 5 100 50 0 50 150 100 0 100 200 200 0
0 150 0 1 0 600 150 2 150 1500 600 1 600 1500 1500 1
0 50 0 1 0 100 50 1 50 200 100 2 100 200 200 2
...
```

Algorithm 9: Extract from fuzzy set data file for weapon selection.

```
line_start = 0
read file and put all data into input_string
for each weapon
        for each input variable
            while newline character not encountered in
        input_string
                        line_length++
            end of while
           copy data between line_start and line_length to
        line_string
        scan line_string for values and put them into
        adapt_FL_t structure
          line_start + = line_length+1
          line_length = 0
         end of for
end of for
```

Algorithm 10: Pseudocode for function to parse fuzzy set data.

amount of data, it is loaded in from a text file on initialisation of the game.

The data is stored in a particular format, as shown in Algorithm 9. The structure of the data can be seen in Table 4 (data missing for clarity); the first two lines of the data representing the input variables for a single weapon, each line representing all the data for each input. A line is made up of four groups of four data items, representing the data for each fuzzy set in ascending order.

In order to load the data, the function *AdaptAllocWepDOMS* needed to be written which extracted the data from the file and put it into the appropriate place in the *adapt_FL_t* structure used to hold the information. The file utilities incorporated into QuakeC [15] are basic, much of which is tailored to specific purposes within the game engine, especially when parsing the loaded data.

The Q3A *trap_FS_Read()* function enables data to read in from the file and be placed into an array of characters. The data then needs to be parsed and placed into the data structure. Algorithm 10 shows the pseudocode for how the data is parsed from the string read in from file.

The function sets up loops for each weapon and each input variable and searches the string for a newline character. The data from the start of the line to the end of the line is then copied to another string, using the Q3A *strncpy()* function, and the Q3A function *sscanf()* is then used to scan through the copied string for the individual data values contained within. Finally, the variable holding the position of the start of the line is set to the start of the next line and the new line length set to 0.

4.5.3. Loading weapon characteristic data

Part of the requirements of the system is that it takes into account the fuzzy preferences that are defined in the characteristic files. Access to the fuzzy component of Q3A is not available in the source code and the data loaded into it could not be extracted for use by the adaptable AI system. This meant that a function needed to be written to load the data from the characteristic file, so that the weapon weights could be used.

The weapon preference file is formatted in a particular way. In a similar way to that of the fuzzy set data, the data needed to be parsed so that the values could be placed into an array. The process of extracting the data required a different technique, as the values are linked to a key representing the weapon that the value applies to. Algorithm 11 shows the pseudocode for parsing the weapon preference data from the file string.

The function goes through the string, character by character, looking for prefix of a key, "*W_*." When it finds this combination of characters, it finds the end of the line and copies that line to another string which is scanned for the key and a value. The key is compared with the weapon names and when a match is found, the value is placed in an array at the appropriate location for that weapon (the weapon number).

4.6. Weapon selection perceptron component development

4.6.1. Data structure

The simple architecture of the perceptron means that the data structure required to hold the data for it is also simple. The structure created, called *adapt_P_t*, is made up of two arrays and an integer variable. The variable simply holds the number of inputs into the perceptron, for use when looping through perceptrons with varying numbers of inputs. Due to the limitations in allocating memory imposed by the Q3A engine, the arrays need to set up to the size of the largest number of inputs into the perceptron. Therefore, the arrays can be of any size so the *numinputs* variable is used when determining the size of the arrays.

One array is used to store the perceptron weights for each of the inputs. The other array stores the input values for the last time the output of the perceptron was calculated. This is required for the training of the perceptron, which needs the value for each input in order to calculate the adjustment for the weights.

4.7. Weapon selection behaviour development

The weapon selection behaviour is controlled through a single function that calls the separate components, calculates the best weapon, and returns the weapon number. It replaces the Q3A function *trap_BotChooseBestFightWeapon()* in the *BotChooseWeapon()* function for the adaptable bot. The function is called *AdaptSelectBestWeapon()*.

Table 4: File format for weapon selection fuzzy set data.

		Set 1				···	···	Set 4		
Weapon	Input	MINL	MINH	MAX	RULE	···	···	MINH	MAX	RULE
Gauntlet	Distance	0	20	0	1	···	···	1500	1500	0
	Ammo	0	10	0	0	···	···	200	200	0
MachGun	Distance	0	150	0	1	···	···	1500	1500	1
	Ammo	0	50	0	1	···	···	200	200	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

```
while not at end of input_string
        if input_string[character_num] = "W" and
          input_string[character_num+1] = "–"
                line_start = character_num+2
                while newline character not encountered
                        line_length++
                end of while
                copy data between line_start and
          line_length to line_string
                scan line_string for key and value
                if key = weapon_name
                        array[weapon] = value
                end of if
        end of if
        character_num++
end of while
```

Algorithm 11: Pseudocode for function to parse weapon preference data.

4.7.1. Information gathering

The weapon selection process requires a number of details about the bot, its enemy and the environment to be gathered. This is done by making use of the data structures and functions defined in Q3A. The distance to the enemy and the ammunition level are common to both the fuzzy logic and perceptron components. Both of these pieces of information can be found in the *inventory* array from the *bot_state_t* data structure. This array contains a large amount of useful information, the ones used for the weapon selection being as follows.

(i) *bs->inventory[INVENTORY_MACHINEGUN]* returns a 1 if the bot currently has the weapon in its inventory. The names of each weapon can be substituted for MACHINEGUN.

(ii) *bs->inventory[INVENTORY_BULLETS]* returns the amount of ammunition type. The names of each type of ammunition can be substituted for BULLETS.

(iii) *bs->inventory[INVENTORY_ARMOR]* returns the amount of armour the bot currently has.

(iv) *bs->inventory[ENEMY_HEIGHT]* returns the difference in height between the bot and its current enemy.

(v) *bs->inventory[ENEMY_HORIZONTAL_DISTANCE]* returns the distance (horizontally) between the bot and its enemy.

Other variables within the *bot_state_t* structure also contain information used for the inputs to the perceptron. The health level of the bot is found using

$$\text{bs- > lastframe\_health} \tag{7}$$

while it can be determined if the bot is directly in combat with another bot or player using

$$\text{bs- > enemy} \tag{8}$$

which returns a -1 if not in combat and the entity number of the enemy when it is.

In order to find the visibility of the enemy and the aggression of the bot, functions need to be called that return the level of each. To get the visibility of the enemy, the function

$$\text{BotEntityVisible()} \tag{9}$$

is called. This returns a floating point number in the range [0, 1], although values other than 0 or 1 are only returned when in fog or water. Otherwise, the value returned simple represents whether the enemy is visible or not. The aggression level of the bot is found using

$$\text{trap\_Characteristic\_BFloat()} \tag{10}$$

and passing CHARACTERISTIC_AGGRESSION as a parameter. This returns a floating point number in the range [0, 1] giving the aggression value defined in the characteristic file.

```
if weapon fired
search through wep_fire_event array for empty slot (i)
if it is not allocated
        weapon_fire = i
        wep_fire_event[i] = event_type
end of if
```

Algorithm 12: Pseudocode for flag fire event function.

4.7.2. Projectile tracking for feedback

The majority of the code for this application was confined to the AI sections of the engine as it purely deals with the behaviour of the bots within the game. Due to the way the feedback is calculated, recording information after a shot and when the projectile impacts, the weapons needed to signal when they fired and the projectile emitted from the weapon needed to be tracked until it impacted on either the environment or the enemy.

In order to achieve this, two different sections of the game engine, the AI (files prefixed with "*ai_*") and the game sections (files prefixed with "*g_*"), were required to communicate with each other using a common data structure that was available to both. The *playerState_t* data structure seemed to provide the answer as it was accessible from *bot_state_t*, which the AI used, and from *gclient_s* which was accessible from the game section. This proved to be problematic in practice due, what appeared to be the same structure in fact being different versions of one another so, data stored in one version from the game section would be copied to the one available to the AI section, but changes made in the AI section were not available from the game section.

This meant changing the location of the data from *player_State_t* to *gclient_t*, which was accessible to the AI section by the way of a global structure that makes data available across the whole server side of the game engine.

When a weapon is fired, the projectile is tracked using an array stored in *gclient_t*, as shown in Algorithm 12. The *FlagFireEvent()* function is in the file *g_weapon.c* and is called from the functions that fire the weapons, also in the same file.

The projectile entity, if a missile, stores the array location of the event, in order that it can be identified upon impact, and stores the type of event (missile or impact) in the *wep_fire_event* array. The missile event is used for the delayed impacts of rockets, grenades, plasma, and the impact event is used for instant shot projectiles and when the missile projectiles impact. An array is used for cases when multiple missiles are active at the same time, for instance when the plasma gun fires a volley or the grenade launcher launches a number of grenades at once. The missiles can impact in any order so the array must search for unallocated slots.

When a missile entity impacts, calling either the *G_MissileImpact()* or *G_ExplodeMissile* functions in *g_missile.c*, it sets the event in the wep_fire_event array to impact and resets the weapon_fire variable. At the end of an AI cycle, the wep_fire_event array is checked for impact events and if one is found, the feedback function is called, resetting the wep_fire_event array to free the slot for other projectiles. It is also checked for missile events and, if a new event is found, a copy of the perceptron input values are put into a wep_event_inputs array that is stored in the *bot_state_t* structure.

5. EVALUATION

For the purposes of testing the adaptation system, the fuzzy logic component was designed to mimic the selections made by the original Q3A AI as closely as possible. This was done so that the changes in behaviour of the bot due to adaptation during a match could be directly compared with the behaviour exhibited by the original AI.

5.1. Adaptation of weapon selection

In order to test whether the bot is able to change weapon preferences within the game, its preferences were set up so that it had a high preference for a certain weapon but also low accuracy. By assigning another weapon a high accuracy but normal preference, the system's ability to change preferences was tested. The adaptable bot's preferences and accuracy levels were set up as follows:

(i) *plasma gun:* accuracy = 0.1, preference = 300;
(ii) *rocket launcher:* accuracy = 0.9, preference = 200;
(iii) *grenade launcher:* accuracy = 0.8, preference = 100;
(iv) *shotgun:* accuracy = 0.7, preference = 150.

The significant weapon numbers in Figures 6 and 7 are 2-machine gun, 3-shotgun, 4-grenade launcher, 5-rocket launcher, and 8-plasma gun.

Figure 13 shows the output of the weapon selection choices, comparing the Q3A AI with the adaptable AI. The graph shows how the adaptable AI and Q3A AI make very similar selections at the start of the match, with only slight variations in the choice of weapon. Towards the end of the match the differences of choice become more evident with regards to the plasma gun in particular; seen clearly in Figure 14 which shows a close up of the last part of the match.

This demonstrates the adaptation occurring on the plasma gun's use as, due to its very low accuracy, the negative feedback lowers its effectiveness rating over the course of the match until the other weapons effectiveness scores make them a better choice. The rating of the plasma gun falls so low that the grenade launcher, with only a third of the preference rating of the plasma gun, is preferred over it in some situations. The rocket launcher is shown to be preferred to the plasma gun in almost all situations, and those times when it is not can be accounted for by the rocket launcher running out of ammunition.

The graphs showing the adaptation of the perceptrons for the plasma gun (Figure 15) and the rocket launcher (Figure 16) illustrate how rating of the plasma gun drops and the rocket launcher rises to a point were the preferences change for the weapons. Whereas, the plasma gun's *medium* range drops to around 0.2, the rocket launcher's rises, albeit

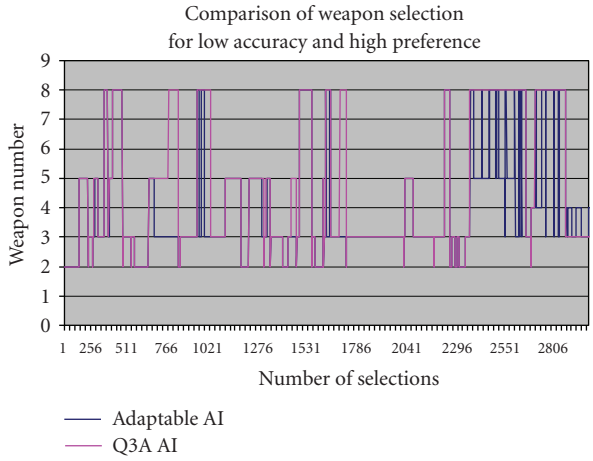

Figure 13: Graph of weapon selection comparison between adaptable ai and q3a ai for low accuracy and high preference of plasma gun.

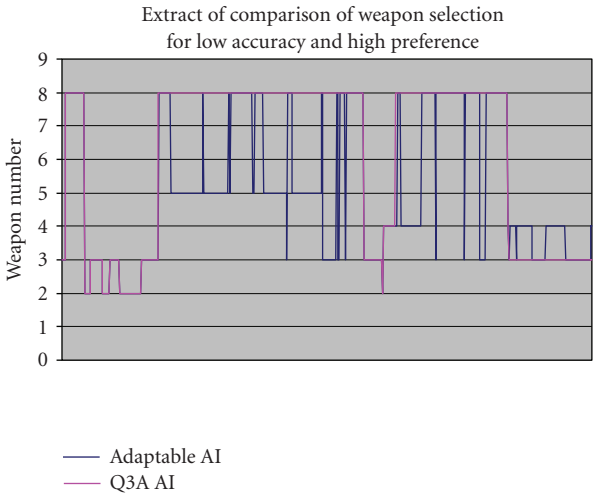

Figure 14: Extract of comparison of weapon selection for low accuracy and high preference, showing difference due to adaptation.

only slightly, to 0.55. This is enough of a variation to cause the change in weapon selection to occur.

5.2. Validity of input choices for perceptron

The inputs chosen for the perceptron resulted in varying degrees of success in their ability to affect the selection of weapons due to adaptation. The distance input was successful in reflecting the feedback of the weapon's strengths and weaknesses in the adaptation of its weights. Trends can be identified from the adjustments made during training that relate to the performance of the weapon in the game.

Another input that demonstrated an effect on the selection process was the height difference, although not to the extent of the distance input. It showed a higher effectiveness for when the enemy is below the bot and lower for enemies above.

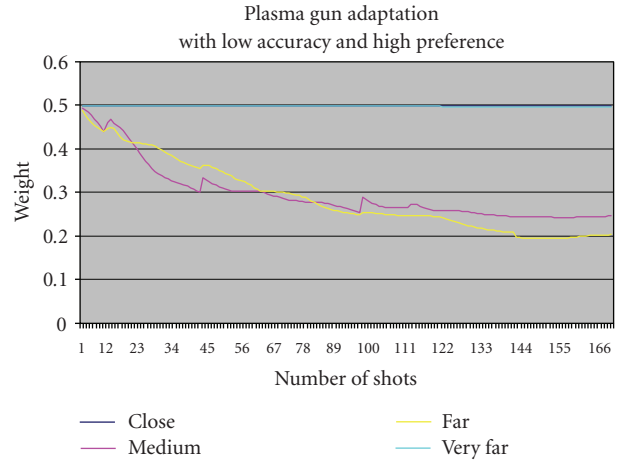

Figure 15: Adaptation of plasma gun distance due to low accuracy and high preference.

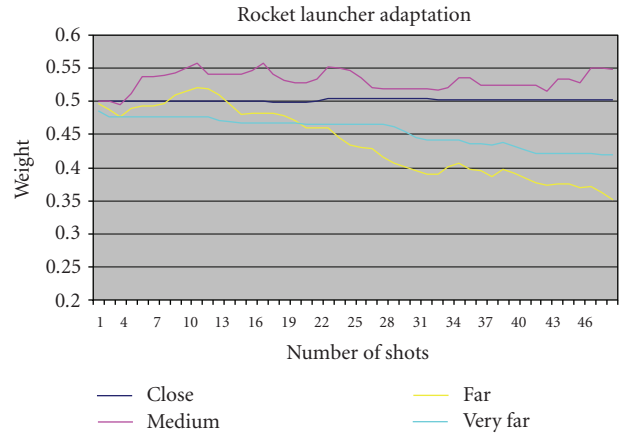

Figure 16: Adaptation of rocket launcher distance due to high accuracy and medium preference.

The ammunition input showed little influence over determining the correct weapons by adapting its values. This is because there is no direct link between the effectiveness of the weapon and the amount of ammunition, therefore the feedback could not influence the ammunition training. The only direct influence of the ammunition on the weapon selection came when the level fell to 0, causing the weapon to be changed to another. All other levels had no bearing on how the weapon performed, indicating that categorisation was not required. Possibly, restricting the ammunition input to a "low ammunition" input would better serve the selection of weapons.

The evaluation of the inputs shows that certain types of input lead to better performance of the adaptation of the perceptron while others contribute little. Generally, the most effective inputs:

(i) directly influence the behaviour; the ammunition input had no direct influence over the effectiveness of the weapon, whereas the distance changed how well it performed;

(ii) are reflected in the feedback; there was no feedback that reflected the effect of ammunition at levels other than 0, when the weapon needed to change. The amount of damage that could be inflicted was not affected by larger or smaller amounts of ammunition.

6. CONCLUSIONS

This paper has shown how, by combining traditional AI techniques, a system can be developed that enables the choices made by a conventional AI layer to be altered in response to feedback from the actions selected. Although the development was limited to just the weapon selection behaviour, so limiting the effect on the game that is visible from testing, evidence was found that the system is capable of adapting to feedback by a significant enough amount to change the actions that prove unsuccessful to those that are successful. The results showed interesting trends that indicate that, with more development and testing to determine optimum settings, the system developed could form the basis of a useable adaptable AI system.

ACKNOWLEDGMENT

The authors would like to express their appreciation to the anonymous reviewers for their very helpful and constructive comments and recommendations.

REFERENCES

[1] J. Manslow, *Learning and Adaptation*, AI Game Programming Wisdom, Charles River Media, Rockland, Mass, USA, 2002.
[2] J. E. Laird, "Game AI: the state of the industry 2000, part two," *Game Developer Magazine*, 2000.
[3] M. McCuskey, *Fuzzy Logic for Video Games*, Game Programming Gems, Charles River Media, Rockland, Mass, USA, 2000.
[4] S. Russell and P. Norvig, *Artificial Intelligence 'A Modern Approach'*, Prentice-Hall, Upper Saddle River, NJ, USA, 1995.
[5] S. Rabin, *AI Programming Wisdom*, Charles River Media, Rockland, Mass, USA, 2002.
[6] N. Palmer, "Machine Learning in Games Development," 2003, http://ai-depot.com/GameAI/Learning.html.
[7] A. J. Champandard, *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviours*, New Riders, Indianapolis, Indiana, 2004.
[8] J. M. P. van Waveren, "The Quake III Arena Bot," University of Technology Delft, Faculty ITS, San Diego, Calif, USA, 2003.
[9] T. Alexander, *An Optimized Fuzzy Logic Architecture for Decision Making*, AI Game Programming Wisdom, Charles River Media, Rockland, Mass, USA, 2002.
[10] M. Zarozinski, *Imploding Combinatorial Explosion in a Fuzzy System*, Game Programming Gems 2, Charles River Media, Rockland, Mass, USA, 2001.
[11] R. C. Berkan and S. L. Trubatch, *Fuzzy Systems Design Principles: Building Fuzzy IF-THEN Rule Bases*, Wiley-IEEE, New York, NY, USA, 1997.
[12] L. Fausett, *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*, Prentice-Hall, Upper Saddle River, NJ, USA, 1994.
[13] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Prentice-Hall, Upper Saddle River, NJ, USA, 2nd edition, 1998.
[14] M. R. Medsker and J. And Liebowitz, *Design and Development of Expert Systems and Neural Computing*, Macmillan College, New York, NY, USA, 1994.
[15] Quake 3 Arena, Id Software, Dallas, Tex, USA, 1999, http://www.idsoftware.com.