Evolutionary Neural Networks for Non-Player Characters in Quake III

Joost Westra and Frank Dignum

Abstract—**Designing and implementing the decisions of Non-Player Characters in first person shooter games becomes more difficult as the games get more complex. For every additional feature in a level potentially all decisions have to be revisited and another check made on this new feature. This leads to an explosion of the number of cases that have to be checked, which in its turn leads to situations where combinations of features are overlooked and Non-Player Characters act strange in those particular circumstances. In this paper we show how evolutionary neural networks can be used to avoid these problems and lead to good and robust behavior.**

I. Introduction

The behavior of Non Playing Characters (NPC's) in First Person Shooter games has become much more advanced. A downside of these better and more complex behaviors is that they take a lot of time to create and optimize. This problem will only increase in the future because the behaviors will continue to take more features of the environment into account in order to look more natural. Almost all the game studios are using very old techniques for creating intelligent behavior. No learning is used for the creation or the fine-tuning of the behaviors. This is all done using scripting and finite state machines. One of the reasons to use these techniques is to keep tight control over the behavior of the NPC during the game.

The goal of this paper is to show that learning techniques also can be used in commercial computer games, making it easier to create complex behaviors and to improve the level of these behaviors, while keeping control over the behavior of the NPC.
One of the more important decisions to take is where AI learning techniques can fruitfully be employed in the design and implementation of NPC's. In [1] we argued that agent technology can be used to implement complete NPC's. However, this approach means that also the game design should be changed, taking into account the higher level of autonomy of the NPC (and thus less control by the designer). In this paper we look into a less disruptive way of using AI techniques. We want to introduce AI learning techniques without disrupting the overall implementation process of the game.

Very good candidates for applying AI learning techniques are the basic decision points of the NPC where it has to select a certain behavior depending on the environment. Examples are weapon selection, selecting which item to get, selecting a target, etc. The advantage of these types of decisions is that they are quite well isolated and therefore the existing methods can easily be replaced by a different one without having to change the rest of the NPC.

We are not the first ones to incorporate learning techniques in FPS games. Zanetti and Rhalibi [2] use neural networks and supervised learning. Three different tasks of the Quake NPC are explored; fight movement, routing in the map and weapon handling. They use data from human players to give feedback on the desired outputs of the neural network during the training period. Using their approach they were able to evolve bots that did better than the standard bots. Bauckhage, Thurau and Sagerer [3] also use supervised learning in a FPS game from data of recorded demos of expert human players. However, this approach is only feasible in cases where (good) human players of the game are available. Typically this is not the case during the development of the game. Therefore we could not use supervised learning the way they did for our purposes. We use an off-line learning approach combining genetic algorithms [4] and neural networks to evolve neural networks that can be used in the actual game. Cole, Louis and Miles [5] used a genetic algorithm to optimise parameters in the FPS counterstrike. Priesterjahn et al. [6] evolved rule-based controllers for agents in the Quake 3 FPS game. Parker and Bryant [7] evolved neural networks using only visual input to play Quake 2.

We will concentrate on weapon and item selection of the NPC's in Quake III in this paper. The main reason to use these selection points is that they already have a good implementation and thus provide a valid test for new AI techniques. Quake III is used as it is open source and thus easy to change the code of the game, while being a good representative of FPS games.

The rest of this paper is structured as follows. In section 2 we discuss the weapon and item selection problems in Quake III. In section 3 we discuss the techniques that we used to implement the selection functions. In section 4 we present the results of the experiments. Finally we draw some conclusions and give directions for future research in section 5.

II. Selection Tasks

There are a lot of different decision problems in the Quake III NPC, some of them have bigger effects than others. It is possible to try and learn all these different problems at the same time, but it would make the experiments unnecessarily large. Weapon selection and item selection were chosen for a few different reasons. The first one is that these tasks probably have a big influence on the total performance. The

Information and Computing Sciences, Utrecht University, The Netherlands (email: {jwestra,dignum}@cs.uu.nl).

978-1-4244-4815-9/09/$25.00 ©2009 IEEE

combination of the two is chosen because they supplement each other. It would be a pity if the item selection is able to learn to pick up the best weapons but they were not selected by the weapon selection.

In the default behavior [8] of the NPC a function call is made to the fuzzy logic unit to select the best weapon. This function call is modified to use default weapon selection if a normal NPC is used and the new weapon selection if the special neural NPC is used. This is done to make it possible to compare against original NPCs. The moment this function call is used is not affected.

Not only is the weapon selection evolving but also the long term goal ”item selection”. The long term goal ”item selection” is an important part of the long term goal selection. Other parts of the long term goal are fighting and retreating. So item selection is used when the NPC is not in a fighting situation. When the NPC is not in a fighting situation he is gathering different items in the arena to get more weapons, ammo or other powerups. Something to keep in mind is when the NPC is picking up items it can go to strategically better or worse positions.

The name ”long term” is a bit confusing. The idea of the original Quake was to combine long term goals with nearby goals. This would make it possible for the NPC to pick up nearby items while having a different long term goal. These nearby goals are never used, thus the long term goals are actually defining the complete goal of the NPC.

Different items are placed in the arena for the NPC to pick up. These items all have a beneficial effect on the states of the NPC. An item can be a new weapon, ammo or a powerup. Picking up an item is just a matter of walking over it, it is automatically picked up. All items are placed in a fixed place in the arena, and if they are picked up they will ”respawn” after a certain fixed time. Some of the items can also be dropped by players that are killed. These items appear in the location where the player was killed. Because all the items have a positive effect for the NPC, but some more than others, a decision has to be made. The navigation is able to plan a path to all the possible items. The item selection decides which item to go for. How good is a certain item? Should the NPC walk further to pick up a stronger weapon or should he first pick up a weaker one nearby? This is the decision problem we are trying to solve with the item selection.

III. Learning Weapon and Item Selection

As said in the introduction we used (feedforward) neural networks with a bias neuron in the input and the hidden layer to represent the selection functions. The sigmoid transfer function is used for the neurons. All the networks that are created have all the weights randomized at the start, which is important when using an evolutionary algorithm to evolve the weights of the networks. Using an unsupervised learning algorithm for the weights was necessary, because we could not assume that we have a training set available to train the network. I.e. we do not have a human player that could be used as the ”perfect” example of the selections that should be made in each case.

The evolutionary algorithm can also be used to learn a lot of different decision tasks at the same time using only one fitness function. The fitness function is also very easy for most games. This is important because game developers should not have to make too many complicated decisions, for example where the game should be divided in episodes or to come up with complicated reward functions.

Different solutions are tested at the same time to explore different possibilities, this facilitates the separate solutions do not need to explore. This means that the best solution is exploited, giving optimal performance.

No evolution on the structure of the networks is used because it makes it more difficult to ensure the real time aspect of the algorithm and they could produce overly complicated networks not needed for most tasks.

A. Weapon selection

For the weapon selection a neural network is used that evaluates all the weapons in only one pass. This works well because all the scores of the different weapons are evaluated at the same time and we do not have too many outputs. The network has the same amount of outputs as there are weapons.

Every evaluation step the inputs are fed into the network, this will give a score for every weapon that is in the game. But not all the weapons are available for the bot to choose. If the bot did not pick up the item or if he has no ammo for it (not all weapons use ammo) it is not possible or very smart to choose this weapon. Sometimes with systems like this the network is left to learn that it should not choose these weapons in these conditions. But it is a lot easier to filter out the weapons that cannot be chosen, this is useful context information that does not introduce any wrong biased information. From the filtered answers, the answer with the highest score is chosen. This evaluation happens very often especially in a fighting situation where ammo can run out.

A lot of different inputs can be used for this experiment but the inputs are kept close to the inputs of the original fuzzy logic code. The fuzzy logic only uses the amount of ammo of the weapon. With the neural network the ammo of all the weapons are used as input simultaneously. This means that the score of one weapon does not only depend on its own ammo but also on the amount of ammo of all the other weapons. As extra inputs the amount of health and armor is used.

When using neural networks it is a good idea to normalize all the inputs, to make sure that one value does not have a bigger influence and to keep the inputs in the best range for the transfer function. This range is usually between 0 and 1 or between -1 and 1. We only have positive values so a range between 0 and 1 is used. The health and the armor have a range from 0 to 200 so they are divided by 200. For some weapons the maximum amount of ammo is a lot higher than the values that occur in a normal situation. If these values would be divided by the maximum it would result in very

small inputs for the weapons. The solution is to divide the amount of ammo by the maximum realistic value and limit the value to 1.

This gives a neural network with 9 inputs with values between zero and one (7 different ammo types plus health and armor) and 8 (the number of possible weapons) different outputs for all the possible weapons. As a guideline we used around twice the number of hidden neurons compared to the number of inputs.

B. *Item selection*

Because the items are not evaluated at the same time in the item selection, and items can occur more than once, and item specific inputs are used, it is not possible to get a score for all the items in one pass. This is why for every item a separate pass is used and only one output neuron is needed. For every item type there is an input neuron. There can be more than one item for each input type (types are shown in appendix B), these are all evaluated separately because the other inputs (for example the travel time) are different.

There are a lot of different items that can possibly be located in a certain level, but most of the time not all possible items are present in the level. This is why for every level we check which items are present thus limiting the number of inputs. There are also inputs for the amount of health and armor similar to the weapon selection.

In the original code there are three different modifiers that have a very big influence on the values of the items. The effects of these modifiers are so strong that the original values of the items almost become secondary to these modifiers. The first one is that all weapons that are dropped got a very large (so large that it immediately is the highest of all items) fixed value (1000) added to it. In this project instead of this bonus an extra input is created to represent if an item is dropped. Secondly, the original fuzzy logic approach divided all the scores by the traveltime, the result is that the bot is very likely to go to a nearby item even if the original score was relatively low. In the neural network approach this dividing by traveltime is left out and the traveltime is passed as an extra input, leaving it to the network to figure out how important the traveltime is for the current item.

And finally there is a similar situation with the ”avoid-time”. The avoidtime is the time the bot does not go to that specific item, this can be calculated because the bot knows how much time it takes for the item to respawn. Staying away from an item which still has to respawn is probably not always the best approach because it can be useful to guard a certain item or position. In the neural network we also disabled this fixed time to stay away, but we do give it as an extra input to the network.

Similar to the weapon selection all inputs are scaled to have values between 0 and 1. The result is a neural network with a number of binary inputs equal to the number of different items available in the arena, plus the three inputs of the weights modifiers and two inputs for the amount of health and armor. This makes around 15 inputs for most levels, using the same guideline as before 30 hidden neurons are created. All items in the levels are tested and the item with the highest output is selected.

C. *Experimental set up*

A big problem is that there is a big variation in performance of the bots due to luck and environmental factors. For example it happens quite often that a bot scores a lot of kills because he acquires a superior weapon when the game just started, because he was placed a lot closer to that item. The problem with this variation is that it is difficult to make an accurate ranking of all the bots. When 6 equal bots are created the difference can be as much as 15 points against 0. An obvious solution to this is to do the ranking over a longer period of time. The longer this period is the higher the chance that the best bot will come out on top. 500 frags still gives some fluctuation but the luck factor is significantly decreased. Of course longer testing periods require much more time for the overall experiments. So, we tried to find a feasible balance between accurate ranking and experiment time and found 500 frags to give dependable results in a decent amount of time.

The ”Sarge” bot is used in all experiments because he gives good overall performance making it a good benchmark to test against.

1) population: The fastest way to evolve a genetic algorithm in first person shooters is to rank all the bots at the same time. We also want to continuously compare the evolved solutions against the original Quake bots. This is needed because we want to see how good the new bots are working and if they continue to improve. If only learning bots are used it is very difficult to know how the bots are performing. If you benchmark them against each other it is possible to rank them, and thus possible to use the genetic algorithm. But it is not possible to evaluate if they are all improving or that they all stay at the same level. This is why we use a combination of learning bots and original bots. The original bots have exactly the same skills and parameters as the learning bots only the decision making process is different.

There is a limitation to how many bots can compete against each other at the same time in an arena. It is possible to put 50 bots in one arena but this is not a good representation of normal gameplay, the arenas are created for around 5 players for the small arenas and 10-15 for the bigger arenas. If too many players are added in a single arena then it becomes almost impossible to acquire the needed items and thus the decision making process cannot be learned. Arena pro-q3dm13 is chosen. This is one of the biggest arenas, making it possible to use 12 bots without any problems. To make a fair comparison we use an equal amount of learning and non-learning bots. So we use 6 learning and 6 non-learning bots. A population size of 6 is very small for a genetic algorithm but we have to make a compromise between speed, realistic gameplay and optimal population size for the genetic algorithm. Using the original bots might also help the learning process because in the beginning the

original will perform better and will eliminate bad solutions faster.

2) fitness function: The fitness function uses the number of kills the bot made in a generation. First a fitness with both the number of kills (points scored) made by the bot as well as a smaller penalty if the bot dies was considered. However, if a bot dies the score remains the same, but the bot loses all its items. Therefore this disadvantage of dying is already part of the fitness because fewer points are scored. Opponents do get extra points if the bot is killed and thus get a higher fitness as well. So, in the end dying already is reflected strongly when comparing the number of points in a game. This is actually a strong point of using genetic algorithms, if the goal of the task is clear, the best thing to do is maximizing the score on this task. No subtasks have to be defined and all sub learning tasks can be learned at the same time using only one fitness function. The number of kills for every bot is saved in the bot state and is reset every generation.

3) selection: There are very few options that can successfully be used for selection because of the small population size. Truncation selection is used to ensure that individuals who are just created and performed below average are not selected. With most other solutions these individuals would have a relatively high change of being selected.

Proportional selection methods, like roulette wheel selection, should not be used because differences in fitness values are relatively small.

In all experiments the best 50% is selected as parent, this gives three parents. Selecting less than three parents would make it very difficult to create different new offspring and if selecting more than three parents below average performers are selected.

4) recombination: Normal real valued recombination is used. For real valued recombination the position of the weights makes no difference at all it is only important that the corresponding weights are matched. This can easily be done by using nested FOR loops. Because the population is relatively small it is important that offspring is created not too different from the parents. This is why not all weights are recombined but a pair of weights is only combined with a certain chance, else the weight from the first parent is kept. Two different recombinations are used; both are a form of line recombination [9]. The first is the most basic one, when weights are to be combined the average of the two parent weights is used.

The second one uses line recombination where a random number a, chosen between -d and 1+d defines the amount the weights of a parent counts. The amount of the importance of the weights of the parent is randomly chosen for every new crossover. We are still also using the chance if a weight should be recombined. If d is zero the values of the weights are always somewhere between the values of both parents. If d is bigger than 0 there is a chance that a is bigger than 1 or smaller than 0. If this happens the values of the weights are adapted further away from the other parent, creating a solution that is even more different than the other parent. This is done to keep the variance of the individuals big enough. If the weights are always averaged the offspring becomes more and more the same. When using a value of 0.25 for d the variance statistically stays the same. When using a value higher than 0.25 the variance grows over time.

5) mutation: We are working with real valued numbers so it is possible to make small changes to all the weights. The used approach is adding a small Gaussian pseudo-random number to all the weights. Random numbers created with software algorithms are always pseudo-random but are good enough for most tasks.

6) Reinsertion: With such a small population it is difficult to test enough new individuals and to make sure not to throw away good solutions. Especially with this learning task we also have such a big variation in performance that it is possible that the best solution does not have the best fitness. This is why elitist reinsertion is used, all the other types give a very high probability of losing the best solution (because of the small population size). Every generation 3 new individuals are created to replace the three worst parents.

Using any form of fitness based reinsertion is not a reasonable option because it would take a lot of time to evaluate the fitness of the offspring.

IV. Results

In this section we present the results of the experiments described in the previous section. Different parameter settings are tested because sometimes they have a big influence when using evolutionary algorithms. We will also look at learning the decision problems separately and combined.

For all the different parameters, series of experiments are performed. Every series is a complete experiment with six learning and six non-learning bots. It is very important to test multiple series because there can be a lot of difference between series, due to the randomness in the environment and the algorithm. If only one experiment is used it could happen for example that one of the individuals is randomly initialized with a very good or very bad solution, this could make a lot of difference in the final result.

To measure the performance of the six learning bots they are compared to six bots that use the original code for the decisions problem but are identical in all the other parameters. In the results we compare the ratio between the average performance of the learning bots against the average performance of the non-learning bots. This is a slight disadvantage for the learning bots because the non-learning bots are all using their optimal solutions and from the learning bots some are using solutions worse than the best learning bot because new solutions are tested in these individuals. This is why in all the graphs with the average performance there is a line with the average performance of the best learning bot compared to the average of the best non-learning bot and line compared to average of all non-learning bots. The final performance after picking the best solution for usage should be somewhere between these lines.

We will not only look at the average performance of the series, but also at the separate series. This is done because

Parameter	Setting
Population size	6
Selection	truncation selection 50%
Mutation; standard deviation	0.05
Crossover type	Average with chance
Crossover chance	10%
Number of hidden neurons weapon	20
Number of hidden neurons items	30

TABLE I
DEFAULT PARAMETERS

it gives more insight in the behavior of the algorithm. It is very difficult to make statistically sound conclusions from all the data because of the fluctuations in the results. Sometimes the score of the best bot does not keep rising but might even drop a bit. This can partially be explained by the variations in score due to environmental factors. But also because the best bot is always competing with the other bots that are learning. Because of the way the genetic algorithm works it is very likely that another learning bot will be created that uses a similar strategy to the best bot. The result will be that these bots want e.g. to go to the same positions to pick up certain items, but meet each other in the process and hinder each other from getting the items.

We first show the results with all the parameters and settings set to the default values we discussed earlier. They are the first results and a benchmark to test other parameter settings against. A recap of all the default parameters can be found in table I.

A. Weapon only

The first experiment used the default parameters. Only the weapon selection is learned. The item selection is done using the original Quake approach. Because the weapon selection in the original Quake bot is quite good this will be a good test to see how easy it will be to learn the decisions using the neural network.

From the results we see that all the series reach a reasonable stable performance after around 15 generations already. The performance is a little bit better than the original code (figure 10)! When looking at the individual series (figure 1) they all have very similar performance. Using the inputs that are available to the neural network it is probably not possible to get much better performance when only changing the weapon selection. An extra indication for this is that the performance is even a little bit better compared to using reinforcement learning on almost exactly the same task [10].

From this experiment we can conclude that even well performing decisions can be learned equally well in relatively few generations.

B. Item selection only

Subsequently we performed an experiment where only item selection is learned and the weapon selection is handled by the original code.

All the series (figure 3) reach an almost stable performance in only 10 generations. There is quite a large variation in the results from the different series but they all reach a level better than the original code. The average performance (figure 2) is a lot better than the original approach.

So, we can conclude that our approach is very well suited for learning this decision.

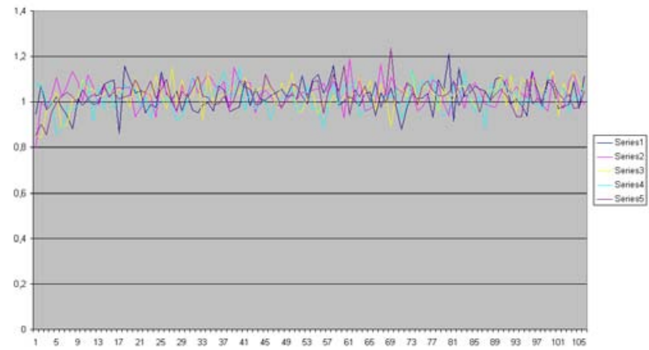

Fig. 1. Only weapon selection

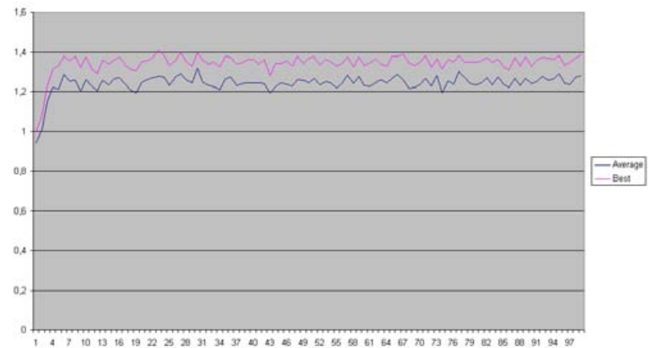

Fig. 2. Only item selection

C. Combining weapon and item selection

When looking at the separate series (figure 5) they all reach a stable level in only a few generations. But there is quite a big difference between the final results of the different series. This is probably caused by some of the series getting stuck in a local maximum and not being able to generate offspring to jump out of these local maxima. Still all the

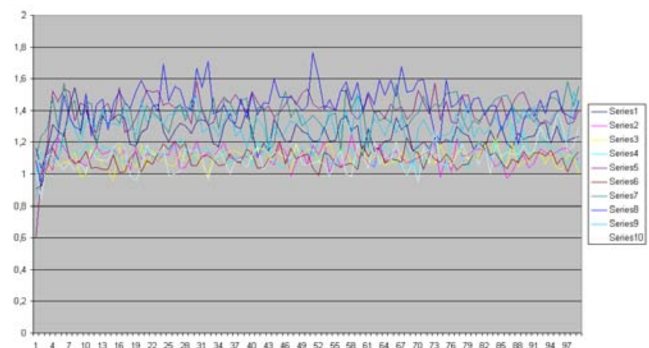

Fig. 3. Only item selection

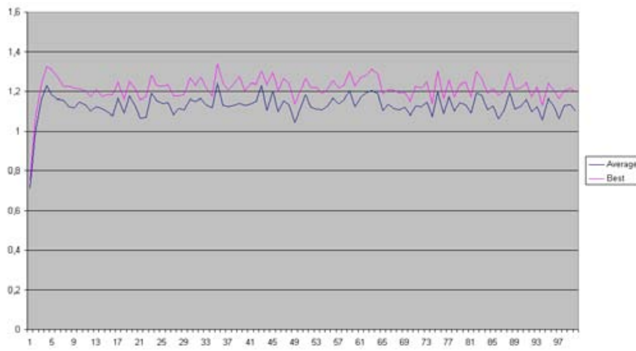

Fig. 4. Combined with default parameters

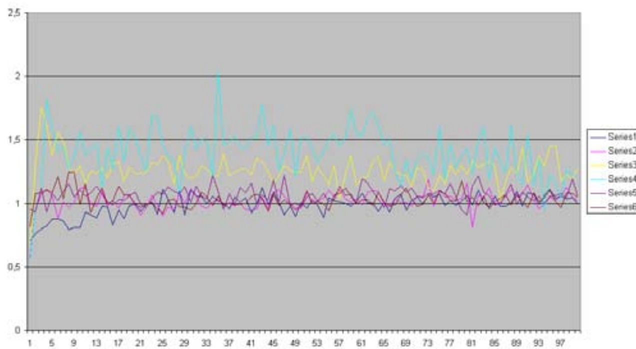

Fig. 5. Default parameters

series perform at least as good, or better, than the original approach. When looking at the average of all the series (figure 4) the performance is 10% to 20% better than the original code.

In this experiment the performance is a bit less than the experiment where only item selection is learned. This could possibly be explained by a certain chance factor of a series getting stuck in a local maximum. However, it is also likely that learning several tasks together has a slightly negative influence on the combined behavior. This is particular the case here because the default weapon selection seems to be quite optimal already and thus any improvement in another part will only improve the bot. So, we did not find a positive influence of learning weapon selection and item selection at the same time. However, as said before the resulting bot still outperformed the original one by a margin of around 10% to 20%.

After the satisfying results of our experiments we varied a number of the parameters to check whether we could improve our results even more.

D. Stronger mutation

In the first experiment we made a significant change in the standard deviation of the Gaussian mutation on all the weights of the offspring. We changed this from 0,05 to 0,25 to see what happens with lots of mutation on the weights. The higher mutation should result in more variation of the networks between generations and thus more chance to find

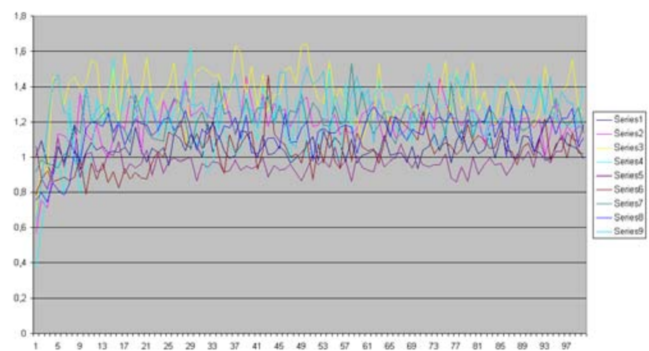

Fig. 6. Extra mutation

better performing networks in a short time. Of course there is also a higher chance of creating very bad performing networks, but these will be dropped in the next generation anyway. For the rest the same settings are used as in the previous experiment, so both weapon selection and item selection are learned. All the series (figure 6) fluctuate a bit more using this setting but they all reach a reasonably stable level. It takes a lot longer for all the series to stabilize compared to the default settings. The average performance (figure 10) is also a bit better, this could mean that the extra mutations help to prevent the algorithm from getting stuck in local maxima.

A bit surprising is that the difference between the average performance and the best performance is about the same as in the experiment with the default mutation (figure 10). This means that newly created offspring with a lot of mutation performs (almost) as good as newly created offspring with a lot less mutation. This might be an indication that the networks are not very sensitive to the exact weights, but more to relative strengths of the weights.

E. Stronger recombination

In this experiment we test the effect of the probability a weight is averaged during crossover. The default setting for this is 10%, in this experiment we are going to test a probability of 30%.

The effect can best be seen in the individual series (figure 7). All the series stay at a certain level and sometimes the performance suddenly improves. This is caused by the too large changes of the recombination, making it very hard to make small improvements every generation. The big jumps in performance are probably caused by lucky big changes in the recombination.

Even though the final results are good (figure 10), it probably is not a good idea to set this probability too high because information of the parents is not represented very well by the offspring, this usually is an important requirement when using genetic algorithms.

F. Less hidden neurons

In this experiment only the number of hidden neurons is lowered for both decision problems. The number of hidden

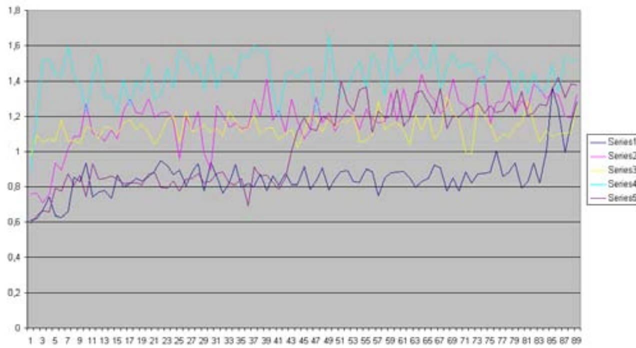

Fig. 7. Stronger recombination

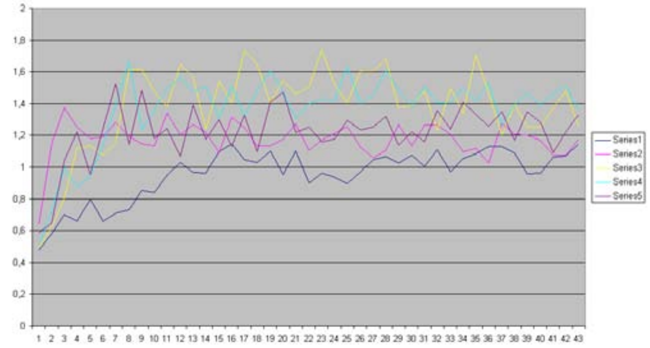

Fig. 9. Advanced line recombination

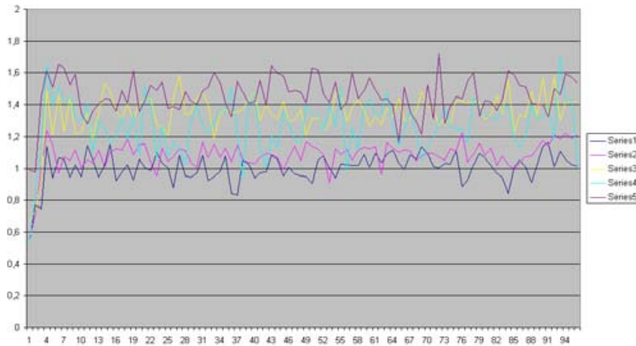

Fig. 8. Less neurons

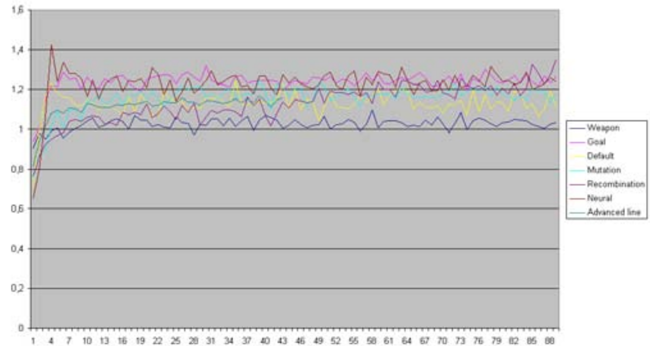

Fig. 10. Average results of all experiments

neurons for weapon selection is changed from 20 to 10 and the number of hidden neurons for item selection is changed from 30 to 15.

The variation between the series (figure 8) is quite large but they all stabilize on a performance equal or better than the original code. This gives a reasonable average score (figure 10). Overall the behavior is very similar to the default parameters, which means that this task can be solved just as good with this amount of neurons. Taking less than this amount of neurons did not improve the result and thus it seems that taking the number of hidden neurons equal to the number of input neurons is a reasonable heuristic.

G. Line recombination

In this experiment all the parameters are the same as in the combined experiment except a different recombination operator is used. The line recombination with a crossover chance for the weights is used. The crossover chance is 10%, just as the default using average crossover. Using this operator the increase in performance continues for more generations in all the series (figure 9), but the increase in the beginning is a bit slower. A value of 0.25 is chosen for the ”d” parameter, keeping the variance stable. The overall performance (figure 10) is not much better than the default recombination. Maybe the performance of the worst series is a bit better than the default series because the population is kept more diverse, but it is hard to know for sure when looking at the data.

H. Combined results

In figure 10 the average performance of all experiments are shown together to make it easier to compare and to compress the information due to space constraints. When only weapon selection is used the performance is only marginally better than the original code. In all the other experiments the performance is significantly better than the original code.

V. Conclusions

From the experiments that we performed we can tacitly conclude that it might be beneficial to replace decision modules in FPS bots by neural networks that are trained using an evolutionary algorithm. We have shown that already using a limited number of generations the networks performed significantly better than the solutions provided in the original bots.

When using this learning approach on different bots they all try to find the optimal solution for the weapon and item selection. This does not mean that all bots will learn to make the same decision because the parameters for the abilities of the different bots are still different, changing the preferences for the different weapons and items. The improved skill level of the bots is not a problem against beginning human players because it is very easy to decrease the performance of the bots (for example making the aiming less accurate).

There were some indications that already there is a real need for solutions like this. The fuzzy logic approach for the weapon selection used in Quake III is reduced to using

fixed values, probably because it would take too much time for the developers to define all the fuzzy relations. However, using learning for subtasks that can be solved by using simple calculations or other logical algorithms (for example aiming or navigation) would overreach the goal. One should use these techniques only for decisions where complex relations play a role and potentially many inputs could be used (leading to an explosion of combinations in traditional approaches). When using neural networks it is very easy to add extra inputs for all the different decisions problems without large penalties in complexity or speed. This is useful in making the behaviors more complex and less predictable.

Something that could hinder the acceptance of this approach in commercial games could be that the programmers lose a bit of the control that they did have before. A neural network always is a bit of a black box; it is difficult to understand exactly what relations are learned by the network. Especially when there are a lot of inputs. But if a lot of inputs are used in the old approaches it also becomes less clear what all the relations are and it becomes too complicated and labor intensive to make them manually.

In order to really substantiate our claim that learning techniques could support the development of complex decision making bots in FPS games we should of course not only show that the learning bots can in the end outperform the original bots, but we should show that these bots could indeed be implemented quicker and easier than the original ones. However, in order to perform this experiment we would need to cooperate with a game developer during the development of a new game and have a kind of contest between developers using different techniques. At the moment this is practically not feasible, but it would be interesting if a game developer would be open to such a challenge.

Some other points for future research are the following. The original behavior of the bots is created in such a way that they keep running in circles (because of the traveltime and avoidtime modifiers). When the bot is learning to use the new representation, the bot can learn that it sometimes is better to wait for a moment at a certain location, because of strategic advantages. For example if a certain weapon is much better than all the other weapons in the current arena, then it is a good idea to pick up that weapon and to prevent the other bots from acquiring it even if you do not need it yourself. This needs an awareness of where your enemies are. In the Quake III version this is a problem, because the bot is not learning to face likely directions of enemies when it is standing still (it might very happily stand looking at a wall instead of along a corridor).

For the reported task it was possible to test all the individuals of a generation at the same time by playing against each other, but the population size already was very minimal. In other games this is not always possible or maybe a bigger population is needed. Different solutions can be used for different games. Sometimes the best approach will be to just test every individual separately in different runs of the game. But more solutions are possible. A nice way to expand the population size would be to use a kind of tournament approach; the different individuals can be divided in small groups that compete against each other and then the best individuals from the subgroups can be selected. Or if needed let these best individuals compete against each other to get a ranking order of all the good individuals.

Because it is easy to add extra input parameters it would be useful to see what happens when certain extra inputs are added. For example the weapon selection could use the distance to the enemy and the height difference between himself and the enemy. And for the item selection it would be nice to see what happens if the coordinates in the arena are also used as an input.

In the performed experiments only one arena was used for testing and learning. It would be nice to see what happens if learning is performed in different arenas. After learning on multiple arenas tests could be performed to see if the knowledge learned can be applied in a new arena. Because the learning can be done without too much human effort it would probably be better to learn specific neural networks for different arenas making it possible for the bots to adapt their decisions to that specific arena.

Finally we should note that not only the performance of the bots is important, but they should also be more fun to play against. It would be nice to test if people perceive the bots as more natural or intelligent.

ACKNOWLEDGMENTS

This research has been supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO) and the Netherlands ICT Research and Innovation Authority (ICT Regie).

REFERENCES

[1] F. Dignum, J. Westra, W. van Doesburg, and M. Harbers, "Games and Agents: Designing Intelligent Gameplay," *International Journal of Computer Games Technology*, 2009.

[2] S. Zanetti and A. E. Rhalibi, "Machine learning techniques for FPS in Q3," in *Advances in Computer Entertainment Technology*. ACM, 2004, pp. 239–244. [Online]. Available: http://doi.acm.org/10.1145/1067343.1067374

[3] C. Bauckhage, C. Thurau, and G. Sagerer, "Learning human-like opponent behavior for interactive computer games," *Lecture notes in computer science*, pp. 148–155, 2003.

[4] M. Mitchell, *An introduction to genetic algorithms*. Bradford Books, 1996.

[5] N. Cole, S. Louis, and C. Miles, "Using a genetic algorithm to tune first-person shooter bots," in *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 1, 2004.

[6] S. Priesterjahn, O. Kramer, A. Weimer, and A. Goebels, "Evolution of human-competitive agents in modern computer games," in *IEEE Congress on Evolutionary Computation, 2006. CEC 2006*, pp. 777–784.

[7] M. Parker and B. Bryant, "Visual Control in Quake II with a Cyclic Controller," *CIG'08: IEEE Symposium on Computational Intelligence and Games*, 2008.

[8] J. M. P. van Waveren, "The Quake III Arena bot," *Master's thesis, University of Technology Delft*, 2001.

[9] H. Pohlheim, "Geatbx: Genetic and evolutionary algorithm toolbox for use with matlab," 1996.

[10] B. Jacobs, B. Kemperman, M. Kous, T. Slijkerman, and M. Vuurboom, "AI in Quake III," 2006.