Evolutionary Neural Networks applied in First Person Shooters

Joost Westra
joostwestra@gmail.com

University Utrecht
Thesis number: INF/SCR-06-34

March 27, 2007

Abstract

Computers games are becoming more and more complex. This calls for better artificial behaviors of the computer opponents. Right now creating these solutions is all done by hand, making it a very labor intensive task. Especially if the number of inputs increases this becomes very impractical.

Several learning techniques are created in the scientific world that could be used to make this an automated process. The goal is to show that these techniques could be used in commercial games. A comparison is made between currently used techniques in commercial games and related work using scientific approaches. Quake III is chosen as platform to test the algorithm, this is a first person shooter game. It was selected because first person shooter games are a good test case and the code was completely open source.

From all the tasks of the artificial behavior we concentrated on decision problems. More specific on weapon selection and item selection. The implemented algorithm using evolutionary algorithms on neural networks is explained and different parameter settings are tested.

We successfully created a system using evolutionary neural networks that fully automatically creates solutions for the decision problems. The performance of these solutions outperforms the original hand coded approach.

Acknowledgements

I would like to thank my supervisors dr. Marco Wiering and dr. Frank Dignum from the University of Utrecht for their terrific guidance and support during this whole project.

IDsoftware deserves a compliment for creating this very good game. And making the code of their games open source.

Finally I would like to thank my friends and family for being so patient and giving me lots of support.

Contents

1 Introduction

Computers have become a lot faster since the first computer games. A lot of this computation power is used for the graphics, but more and more resources are becoming available for intelligent behavior. In the last years, games are introduced where artificial players can directly compete against human players on complex problems.

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 become more complicated. 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 by hand.

This in contrast to scientific research, where these algorithms are hardly considered to be part of the field of artificial intelligence and an incredible amount of new algorithms have been developed over the years. In scientific research on artificial intelligence a lot of work is done on different learning techniques. In most of these experiments very artificial environments are used, compared to modern computer games, for testing these different algorithms. Making the game studios very skeptical about the practical use.

The goal of this thesis 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.

In the following sections the research question will be elaborated further.

1.1 What is a First Person Shooter?

A First Person Shooter is a genre of computer games where a player walks around in a virtual 3D world. The point of view is always rendered from the virtual eyes of the character. Figure 1 shows an in-game screenshot from Quake III [4]. There are different environments where the action takes place, these are called arenas or maps.

A big part of all First Person Shooter (FPS) games is the use of handheld weapons. The goal is to stay alive as long as possible and to kill as many opponents as possible. During the game a player can pick up different powerups to improve the abilities of the player. Some different types of team versus team games are also possible.

ID Software with the games Wolfenstein 3D and Doom has had a big influence in the creation of this genre. Quake III is also from ID Software but the game is three full generations newer than Doom. Not all games that

use this first person viewpoint and use weapons are considered first person shooter. The difference with other shooter games is that in an FPS the player has a lot more control over the movement of the character to avoid enemy attacks or collect items.

The first FPS were created as single player games where the character has to complete the level by making it to the end without getting killed. The opponents in these games do not display very intelligent behavior. Most of the time they walk straight to the character without avoiding damage or collecting items.

With newer generations the intelligence of the opponents has become a lot better. This created the possibility to create "bots". These are opponents with identical capabilities as the character of the human player. It is possible to play a multi player game against the bots, which gives an experience very similar to playing a multi player game against other human players. The goal of a multi player game is to score as much points as possible until the game has ended (time has passed, or maximum score is reached by a certain player), points (frags) are scored by killing the other players. The bots are now at a level that they pose a serious challenge against most human players. The aiming ability and the reaction speed surpass that of most human players. But good human players can still win because the bots do not always make the best decisions. Improving this decision making process is what we are going to concentrate on in this report.

1.2 Game choice

1.2.1 Why First Person Shooter?

There are a lot of different types of games where learning can be used. Why was a first person shooter chosen for testing this new approach? First person shooters are considered to already have a high level of intelligent behavior compared to other games. The artificial behavior is very important in the newest first person shooters because the artificial players are representing a single human player. This means that each bot should be intelligent enough to have the same performance as a human.

First person shooters are also a good test to see if an algorithm can work in real time, because these kinds of games have a really high pace. It is not possible to take extra time for a certain timestep, this is possible for certain tasks in more strategic games (chess is an extreme example). The decision time always has to be very low because situations change very fast.

Another reason for choosing these kinds of games is that the best human players are still better than the best artificial opponents, making it a useful

Figure 1: In-game image from Quake III

task to improve.

1.2.2 Why Quake III?

There are several reasons for choosing Quake III as a base for the experiments.

Quake III is a first person shooter with real computer versus human multiplayer; the bots perform the same task as a human player. This is not the case with some of the older first person shooters like Quake II. In these older games the opponents were less intelligent, making it impossible to create opponents for a real equal multiplayer game. These older games used a single player with a lot of less powerful opponents against the human player. Using Quake III already gives much more intelligent bots, making it more useful to expand on and to use for comparison. Quake III is known for the high level of the intelligence of the bots compared to some of the other first person shooters.

Quake III is a well known commercial game, this is important to show game developers that the tested approach will not only work in special conditions or environments.

The most important reason for choosing Quake III is that the code is now completely open source. This is very useful when making big modifications to the game. In some other games it is possible to change certain things by making modifications in separate libraries without changing the source itself. There are two problems with this approach; the first is that it is not possible to really figure out how the original code is implemented, and an even bigger problem could be that certain things that need to be modified can not be modified because these kinds of changes are not foreseen by the programmers. When using a project that is completely open source as a base, there never is a risk that halfway through implementing you become stuck because certain operations are not possible.

1.3 The Quake III bot

The Quake III bot is an artificial player, created to emulate a human player in a multiplayer first person shooter, also known as a bot. The bot needs no human interaction to navigate through the levels, all path planning and reachability calculations are done real-time.

The goal for a bot is to make it act like a human as much as possible. Humans gather information about the game by looking at the rendered graphics (figure 1) and listening to the sounds. Because the bot is directly linked to the game engine the state of the environment can directly be accessed to gather this information. To make the bot a realistic player it is important

that the bot is not cheating in this step. The bot should only know where an opponent is when he is facing the right direction and the vision is not obstructed. The bot is allowed to predict where an opponent is going without using extra information.

The complete layout of the map and all the possible paths through the map are always accessible for the bot. Also the positions of all the different items are known. This may sound like an unfair advantage but experienced players are also able to do this after exploring a map. The bot does not know if an item is available at the moment without "looking" in the right direction.

Outputs of humans are processed via the keyboard and the mouse, bots send actions directly to the game engine. Some of the possible basic actions are moving, shooting, weapon selection, looking direction and using an item.

The bot is built up in several layers (figure 2). In the first layer are the awareness system and the basic actions, these are the direct interactions with the game engine. The area awareness system gives a preprocessed representation of the environment, only "fair" information is created, so the higher layers can use all this information.

The second layer should represent the part of the intelligence of a human player that is subconscious. Weapon and goal selection are handled in this layer. Also the navigation to the selected items is done in this layer. In the third layer different situations and states, the bot can be in, are defined. Here the higher level thinking is defined for these different situations. The behavior of the bot in fighting situations is an example of this. The fourth layer is only for a team leader in a team play situation, here it is decided what the roles of the different bots should be.

The arrows that go up in figure 2 represent the information flow of the environment and the bot's status. The information becomes more abstract when going to a higher level. The arrows that go down represent decisions the bot makes, all the way down to the basic actions. There also is some information flow between different components in one layer. The AI Network receives information from the Miscellaneous AI and the Goals component retrieves information from the Fuzzy component for goal selection.

To make the bots more versatile and to create some variation in the bots different characters can be defined. The characters not only have a different appearance but they also have different strengths and weaknesses. This should also make the bots less predictable because they all behave a little bit different. One of these values is an aiming accuracy parameter for each weapon, so a bot can be more accurate with one weapon but less accurate with another. Also the preference for certain items or weapons is different. The maximum rotation speed is also a parameter for every bot,

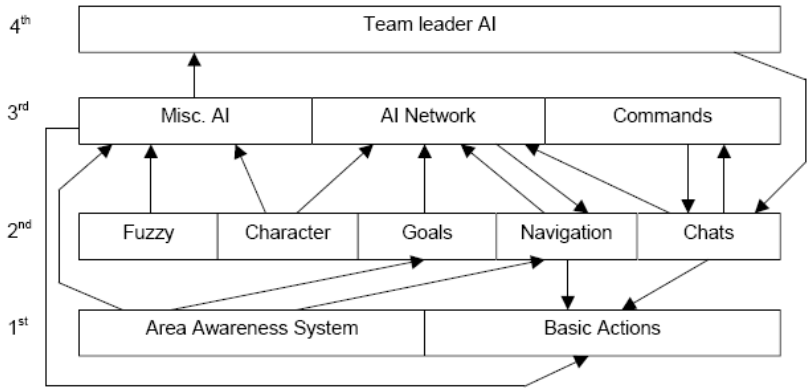


Figure 2: General architecture of the Quake III bot

if this speed is set too fast the bot will turn in an unrealistic speed. There are a lot more parameters defined but not all these different parameters are used.

1.4 Decision problems

In very old computer games like pong it was very easy to figure out what the best action for an artificial opponent would be. There are no decision problems, just performing one simple task as good as possible (stay behind the ball in the pong example).

But as games become more and more complicated it becomes a lot harder to decide what the best action should be. Choices have to be made between several possible options. What movement shall I make? Which weapon should I select? How aggressive should I be? There are many more of these kind of dilemma's in current computer games and they all have an impact on the behavior of the artificial player. There are a lot of different ways to solve these decision problems. Different approaches for making these are discussed and tested in the following chapters.

1.5 Problem statement

As discussed earlier, the goal is to prove the usefulness of using scientific research on artificial intelligence. Now we can be more precise. Can AI techniques be used efficiently for the decision making process of bots in Quake III? More specific; can they be used for the decisions on item selection and weapon selection? To test this we are going to concentrate on decision making in Quake III. The decisions we are going to concentrate on are item selection

and weapon selection.

First in chapter 2 we will show techniques currently used by commercial game developers. In chapters 3 and 4 scientific approaches are explained and related work is discussed. More explanation about the two decision tasks and the setup of the experiments can be found in chapter 5. The results of the experiment are shown in chapter 6. In chapters 7 and 8 the conclusion and possible future work can be found.

2 Currently used techniques

First we are going to discuss a few techniques used by current commercial game companies [10][11], so later on they can be compared to the scientific approaches. Four different kinds of techniques are discussed. First simple hard coding is explained and then scripting as an expansion to that. After that, finite state machines are explained which are able to make different decisions in different situations. And last fuzzy logic which makes it possible to use more fluent value representations for decision problems.

2.1 Hard coding

This is one of the most basic ways to implement AI in games. It is very computationally efficient because most of the time it only takes very basic instructions and it can all be optimized by the compiler. Of course there are many different ways to hard code anything using a fully capable programming language. We shall concentrate on the simple and most used IF THEN constructions. With the hard coding approach the programmer has the most control, he is less bound to a specific API for inputs, outputs or operations. So if an extra input is needed, he can just change the source code to supply that input. Below is an example of using a hard coded approach in Quake III to specify the aggression of the bot.

```
float BotAggression(bot_state_t *bs) {
  //if the bot has quad
  if (bs->inventory[INVENTORY_QUAD]) {
    //if the bot is not holding the gauntlet or the enemy is really nearby
    if (bs->weaponnum != WP_GAUNTLET ||
      bs->inventory[ENEMY_HORIZONTAL_DIST] < 80) {
      return 70;
    }
 }
  //if the enemy is located way higher than the bot
  if (bs->inventory[ENEMY_HEIGHT] > 200) return 0;
  //if the bot is very low on health
  if (bs->inventory[INVENTORY_HEALTH] < 60) return 0;
  //if the bot is low on health
  if (bs->inventory[INVENTORY_HEALTH] < 80) {
    //if the bot has insufficient armor
    if (bs->inventory[INVENTORY_ARMOR] < 40) return 0;
  }
```

```
  //if the bot can use the bfg
  if (bs->inventory[INVENTORY_BFG10K] > 0 &&
      bs->inventory[INVENTORY_BFGAMMO] > 7) return 100;
  //if the bot can use the railgun
  if (bs->inventory[INVENTORY_RAILGUN] > 0 &&
      bs->inventory[INVENTORY_SLUGS] > 5) return 95;
  //if the bot can use the lightning gun
  if (bs->inventory[INVENTORY_LIGHTNING] > 0 &&
      bs->inventory[INVENTORY_LIGHTNINGAMMO] > 50) return 90;
  //if the bot can use the rocketlauncher
  if (bs->inventory[INVENTORY_ROCKETLAUNCHER] > 0 &&
      bs->inventory[INVENTORY_ROCKETS] > 5) return 90;
  //if the bot can use the plasmagun
  if (bs->inventory[INVENTORY_PLASMAGUN] > 0 &&
      bs->inventory[INVENTORY_CELLS] > 40) return 85;
  //if the bot can use the grenade launcher
  if (bs->inventory[INVENTORY_GRENADELAUNCHER] > 0 &&
      bs->inventory[INVENTORY_GRENADES] > 10) return 80;
  //if the bot can use the shotgun
  if (bs->inventory[INVENTORY_SHOTGUN] > 0 &&
      bs->inventory[INVENTORY_SHELLS] > 10) return 50;
  //otherwise the bot is not feeling too good
  return 0;
}
```

As can be seen in this example, these programs can become large very fast when using multiple inputs. This approach also makes it very difficult to combine different inputs. All the different combinations with different inputs that influence each other should be programmed by a very large tree like nested IF THEN structure, this is not very practical because of the amount of time it takes. An example of this can be seen in the code of the Quake example. If the bot has a "Quad Damage" powerup (and the opponent is nearby) the bot should be aggressive and if the bot has a strong weapon it should be more aggressive. If the bot has both it should be even more aggressive (if the weapon can not do damage to himself), this is not the case in the example above. The bot even becomes less aggressive. If the bot has "Quad Damage" and almost any weapon, the aggression is 70, this is lower than if the bot is holding the railgun without "Quad Damage". It would also be better if both the health status and the chosen weapon were combined for all the different weapons and health levels, but this is too much work with this approach (combining health, weapon and item status would

be even better and even more work).

The values also have no form of interpolation; this means that if you want to implement an output without the steps between the outputs being too big, a lot of different possible outputs have to be defined by hand.

There is no separation between the engine and the values, all the values have to be changed in the source code. This also makes it more difficult for the people who did not program this part to find the right place to change the behavior.

2.2 Scripting

Scripting is an extra abstraction layer on top of hard coding. It is a more specialized language to make it easier for the programmers for this specific task.

Scripting is the system that is used in a lot of computer games at this moment. Some of the best known are Unreal script language [7][15] and LUA [5]. The basic principle of a script is defining different basic actions which can be triggered by an event. For example turning on a light when the player activates a switch, or an enemy that starts attacking if the human player passes a certain point.

An advantage of a special scripting language is that people who are implementing the behavior only have to understand this language. The behavior of the bots will always be very predictable. An advantage of this is that a bot will probably do what the programmer expects from it. A disadvantage is that the bot is also very predictable for the player. If there is a certain weakness in the behavior of the bot, the player can exploit this weakness over and over. With scripting it is also possible to write complete scenarios very similar to a movie, this is used very often in modern games to make the agents look intelligent.

But a computer game is not the same as a movie; the player is in control to go where he wants and which action he is going to take. This is where the scripted scenarios often fail, the programmer expects all the players to take certain actions and writes a clever response to that. But if the player takes a different approach, which was not foreseen by the programmer the scripted opponents usually do not respond in an intelligent fashion. For example if an opponent is guarding a room, the programmer expects the player to walk in the room through a certain door and get attacked by the guarding bot that also sets of a bomb. But if the bot enters the room through a window, the whole scenario is not triggered and the player can easily shoot the bot in its back, without its preprogrammed behavior ever getting activated.

Most of the time a scripting language is interpreted in real time, this

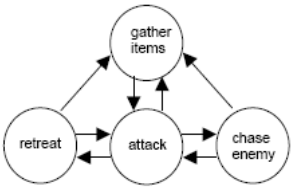


Figure 3: Simple finite state machine

is not very efficient from a speed perspective. The separation between the engine and the behavior is good using a special scripting language. However when a map is changed these kinds of scripts also have to be adapted by hand.

2.3 Finite state machine

Finite state machines are usually not a replacement for the other approaches we discussed, but most of the time they are combined to specify different situations. In these different situations some of the other techniques can be used to make situation specific choices.

A finite state machine always consists of a fixed number of states, these states can represent the current state the agent is in. Only one state at a time can be the active state. The different possible transitions between the states are predefined, some states can not transition to certain other states. A transition to a different state can both be triggered by the internal state of the agent or by an external trigger from the environment. When using finite state machines the behavior of the agent has to be divided in different subtasks. Figure 3 shows a very simple example of a behavior for a bot using a finite state machine.

As you can see the agent can transition from the retreat state to the gather items state, but not the other way around. The finite state machine is most often used on top of for example a scripting or hard coded approach, so different scripts are used in different states. The goal of a finite state machine is to represent the thinking process of humans, but humans often are able to combine different goals. For example a human would try to pick up a nearby powerful weapon while chasing an opponent, this is not possible using a finite state machine approach with separate states for chasing and gathering. In Quake III finite state machines very similar to Figure 3 are used to switch between different subbehaviors.

2.4 Fuzzy logic

Fuzzy logic is a way to express a non-linear function of something. This can be used for things where we do not want to know if something is true or not, but when you want to know how good (or even how likely) something is.

This can then be used to decide which item the bot should go for. In the Quake III code it is used to calculate how much the bot wants a certain item or which weapon is preferred. Different criteria can be used as an input to evaluate the value. There are different types of fuzzy logic. We are going to concentrate on the tree like structure used in Quake III. This is the structure that is used:

```
value"name"
{
  switch([criteria1])
  {
    case [smaller than a certain value]
      switch([criteria2])
      {
        case [smaller than a certain value]:return [matching output value]
        case [smaller than a certain value]:return [matching output value]
        default:return [matching output value]
      }
    }
    case [smaller than a certain value]:return [matching output value]
    case [smaller than a certain value]:return [matching output value]
    default:return [matching output value]
  }
}
```

This would give a very jagged outcome if the numbers of input values are limited. This is why interpolation between the two closest values is used to give a better estimate for values between inputs.

An advantage of using this structure is that it is relatively easy to understand what is happening. When something unexpected happens it is not a big problem to find the node responsible for the problem.

In this example only two criteria are used and there are already a lot of values that have to be added. Adding extra inputs can make the tree grow exponentially. When this tree becomes too big the advantage that it is easy to understand also disappears. Another problem is that these values all have

to be added by hand and most of the time the values are completely arbitrary and have to be tuned using experiments.

The separation between the engine and the variables is good. The fuzzy logic structure is defined in the engine and the values are supplied in separate bot configuration files.

Usage in Quake III When reading the documentation of the Quake III bot [16] it looks like a reasonable solution for some of the preference problems in the game. This is an implementation example for the value for one of the weapons:

```
weight "Lightning Gun"
{
  switch(INVENTORY_LIGHTNING)
  {
    case 1: return 0;
    default:
    {
      switch(ENEMY_HORIZONTAL_DIST)
      {
        case 768:
        {
          switch(INVENTORY_LIGHTNINGAMMO)
          {
            case 1: return 0;
            case 50: return 70;
            case 100: return 77;
            case 200: return 80;
            default: return 80;
          }
        }
        case 800: return 0;
        default: return 0;
      }
    }
  }
}
```

This would result in the output seen in figure 4. This looks like a pretty nice good thought out solution. But how do we know if these values are any good? They filled in the values using "common sense", but how do you

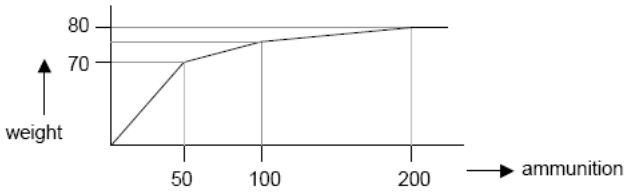


Figure 4: Fuzzy logic lightning gun

know which weapon is better given the abilities of the bot, and how do you know how much impact the amount of ammo has? After looking in the code it becomes clear that they did not know. The solution they used was only using one criteria and only one value. The complete fuzzy logic is reduced to checking if the item and ammo are present. Only one value is returned so no interpolation is possible. This leaves a system that uses fixed preferences for the weapons. This gives a very predictable behavior. If a player figured out the weapon preferences of a bot, he knows if the bot is going to select a certain weapon if he picked it up.

Learning The main problem here is not that fuzzy logic is a bad system, but it is a bad idea to figure out the complex interaction between all the values by hand. Using fuzzy logic in combination with learning (for example evolutionary algorithms) might give some interesting results. But other systems are probably better when using multiple inputs. In the next chapter different learning techniques are discussed, some of these learning techniques could be used in combination with fuzzy logic.

3 Scientific representations

We are going to implement learning on tasks using a lot of different inputs. Tables and treelike structures become too large, therefore another representation is needed. In this chapter we will look at some techniques from the field of artificial intelligence that can be used to solve this problem.

3.1 Function approximators

What are Function approximators? Function approximators are a way to compress a large number of state actions pairs. Unfortunately with a lot of problems this is not possible without losing any accuracy. This is why it is called an approximation. There are different types of function approximators for example course coding and tile coding. In this thesis we will concentrate on neural networks.

Why Function approximators? Genetic algorithms can be used on a tabular representation. And with tabular representations very complex (non continuous) functions can be exactly represented. But there are some problems with using a tabular representation. The most important problem is that for every extra input or output the size of the table grows exponentially. This grows too big very fast.

This problem becomes even greater in games because most inputs have a big range of different values. A lot of values are in the hundreds range and some even use decimal values making the number of inputs too big for a table.

Another benefit of using function approximators is that they can be used to generalize. Without function approximators, situations that do not happen very often only play a small role and probably get very little feedback while learning. This will probably result in very different solutions for very similar states, and the states that get very little feedback probably get very poor solutions. When using function approximators, situations that do not happen very often, benefit from situations that are similar but happen more often.

3.2 Neural networks

Neural networks are a type of function approximator. They are good at generalizing which is very important in the tasks we are trying to learn here.

Artificial neural networks are inspired by the way brains of humans and animals work, a brain contains many billions of interconnecting cells (neu-

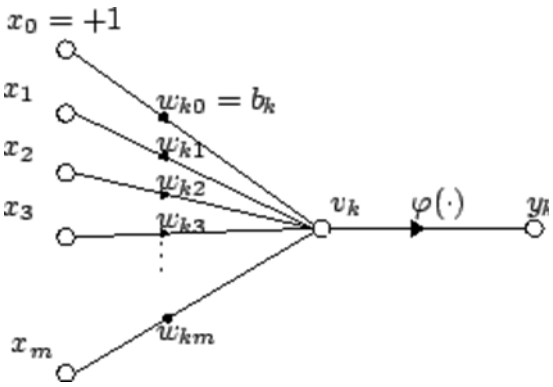


Figure 5: artificial neuron

rons). These cells can pass signals to each other that are not merely on or of but they can have varying strength. It is debatable if an artificial neural network is a good approximation of a brain, but it is nice to use a system that is inspired by brains when trying to replicate human behavior.

Artificial neural networks are often referred to as just neural networks. A neural network consists of an interconnected network of artificial neurons. An artificial neuron processes one or more inputs to one or more outputs (Figure 5). All artificial neurons can be defined by this formula:

$$y_k = \varphi\left(\sum_{j=0}^{m} w_{k_j} x_j\right)$$

Thus for each neuron k all the inputs (x_j) are multiplied by their corresponding weights w_{k_j} and then everything is added up (the summation).

After the summation we still have the transfer function (φ) to limit the output range and to enhance the capabilities of the whole neural network. The inputs can be outputs from other artificial neurons or they can be inputs from the task we are trying to solve.

The outputs can be inputs for other neurons or they can be outputs for the task. The part of the formula without the transfer function (φ) shall be referred to as x. To change the mean value of the transfer functions an optional bias value can be added.

Step function This transfer function only has two different possible outcomes, namely 1 and 0. If the output before the transfer function is higher or equal than a certain threshold (θ) the output is 1 else the output is 0.

$$\varphi(x) = \begin{cases} 1 & if\ x \geq \theta \\ 0 & if\ x < \theta \end{cases}$$

Linear combination Linear combination is just the summation plus an optional bias.

$$\varphi(x) = x$$

Neural networks with neurons using only this kind of transfer function are called perceptrons.

Sigmoid This is the one that is used most often. A nice property is that the derivative is not too complex, this is only important if reinforcement learning or supervised learning is used not when using genetic algorithms. It is a non-linear function. The range is fixed between zero and one. Sometimes a bias of 0,5 is subtracted to get negative outputs when the output before the transfer function was negative, and positive outcomes when the output was positive. In figure 6 there is a plot of the sigmoid function, as you can see changes in values close to zero have a much larger effect than the same changes on very large values.

$$\varphi(x) = \frac{1}{1 + e^{-x}}$$

Guassian The Gaussian function is the probability function of the normal distribution. This non-linear function also has the property that changes of values close to zero have a larger effect (figure 6). Equivalent positive and negative numbers give the same output, this property makes it unsuitable for tasks where positive and negative inputs should give different results. It is mostly used for tasks where deviation is important.

$$\varphi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)}$$

with mean μ and variance σ^2

Non-linear is more powerful A multi layer neural network using linear transfer function can always be replaced by an equivalent single layer network. This is not the case with a non-linear transfer function, here the multiple layers add reasoning power.

Network structure Figure 7 shows a diagram of a simple 3 layer feed-forward neural network, this is the type of network that is used most often. The first layer is usually called the input layer, the second is called hidden

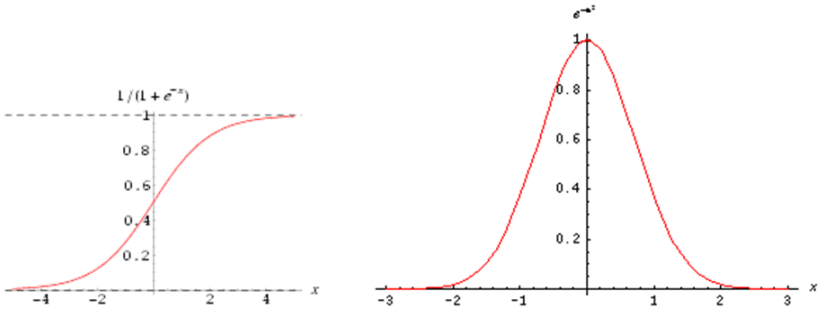


Figure 6: Left: plot of sigmoid function Right: plot of Gaussian function

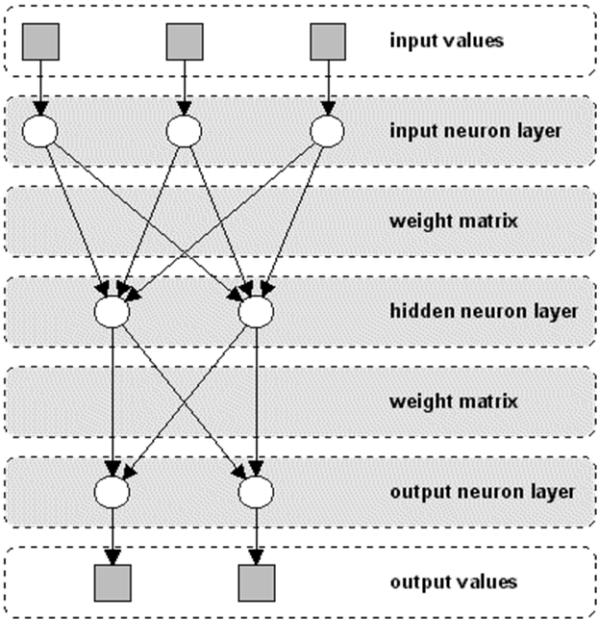


Figure 7: neural network

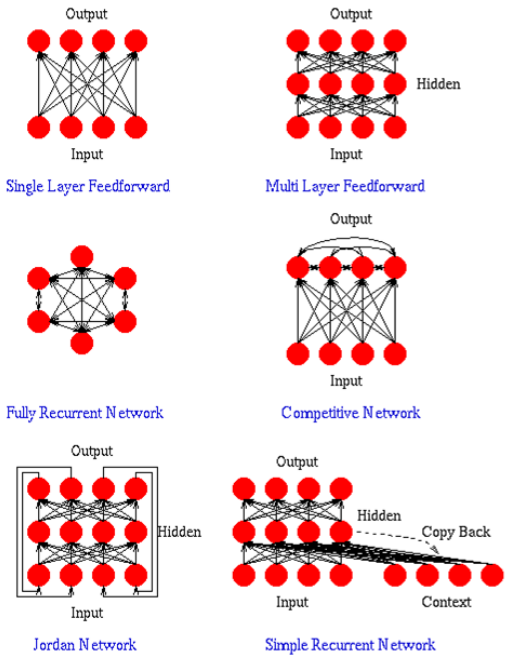


Figure 8: Different types of neural networks

layer and the third is called the output layer. On the top are the input neurons with the corresponding inputs from the task. From this first layer there are connections to all the neurons in the second layer, these connections are outputs for the neurons from the input layer and inputs for the hidden layer. The situation from the hidden layer to the output layer is similar. Because for every neuron there is a connection to all neurons of the next layer this is a fully connected feedforward neural network.

For every connection there is a separate weight, these weights define how much positive or negative effect the output from the above neuron has. These weights have to be adapted to optimize the network for the task.

There are more types of neural networks than only feed-forward as show in figure 8. An interesting category are recurrent neural networks, these networks have connections going back to the previous layers. These are not just the outputs from that neuron used as an input for the previous one, but it is the output from the previous evaluation of the network.

4 Scientific techniques

There are a lot of different approaches that can be used for solving the kind of decision problems we are trying to solve here. The ones that are used most often can be divided in three different main categories; namely supervised learning, reinforcement learning, and evolutionary algorithms. These shall be explained in this chapter and if there is related work using these algorithms, they are also discussed.

Extra attention is given to evolutionary algorithms because this is the approach used in the experiments of this thesis. For all the other algorithms related work is discussed to show how these algorithms could be used for similar tasks. The discussed articles are selected to be related as much as possible to the learning tasks of this thesis.

4.1 Evolutionary algorithms

An evolutionary algorithm is a stochastic search method that improves using a technique very similar to biological evolution.

When using evolutionary algorithms [9] we start with a population of a fixed size. Each individual from this population represents a possible solution to the problem. The representation can be very different depending on the task. The initial population has to be randomly generated to get diversity.

All these individuals have to compete to make it to the next generation. The idea is that better individuals are more likely to make it to the next generation, this is similar to survival of the fittest in nature.

The population needs to stay a fixed size for every generation. This means that individuals that did not make it to the next generation need to be replaced by new individuals. These new individuals are created using information from the better individuals that did make it. Hopefully these new individuals retain information of good solutions or maybe represent an even better solution. Most of the time these new individuals are mutated a bit to increase the diversity of the individuals.

Every generation we keep good solutions compared to the rest, throw away solutions with below average performance and create new promising solutions. This process can be repeated an infinite number of times, usually the improvements over every generation flatten of and the algorithm can be terminated. In figure 9 you can see a schema of the whole process.

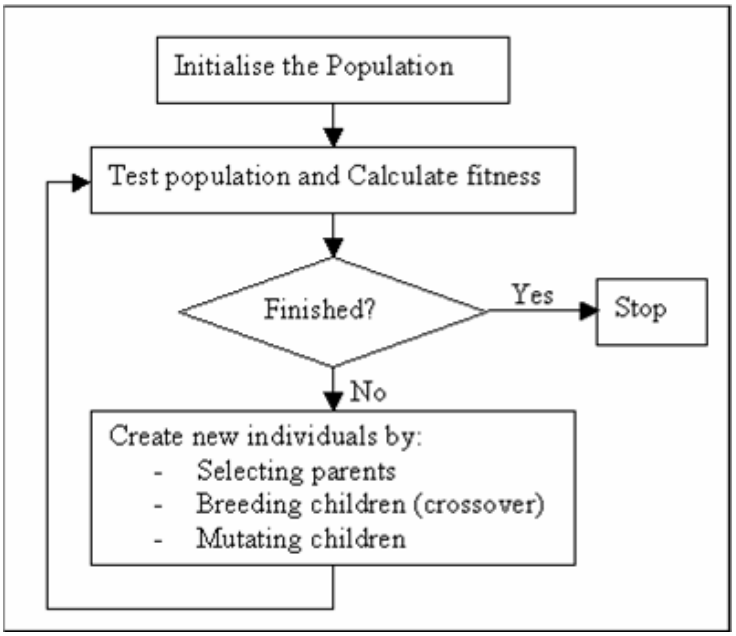


Figure 9: Schema of an Evolutionary Algorithm

4.1.1 Selection

In the selection phase the offspring producing individuals are chosen. There are many different types of selection but they all have in common that better individuals have a higher chance of being selected. Selective intensity is the expected average fitness value of the population after applying a selection method to the normalized Gaussian distribution.

Truncation selection Truncation selection is a selection process that is also used by real life breeders. A certain truncation percentage has to be chosen when using truncation selection. This truncation specifies the percentage that goes through to the next generation. Only the best individuals are selected. So if we have 100 individuals and the truncation percentage is 10% then the 10 best individuals are selected.

The selection pressure is not dependent on the variance of the fitness values when using truncation selection. It makes no difference if the best 10% is much better than the rest or if they are just marginally better. Only the order of the individuals matters. The selection intensity is higher if the percentage is smaller.

Roulette wheel selection For roulette wheel selection all the individuals are mapped to a certain range. This range is proportionally divided over all the individuals according to their fitness. So if an individual has a fitness twice as high as another individual, its part on the range is twice as big. This can be compared to a roulette wheel where the sections of the wheels are sized according to the fitness of the individuals. Then a random number between the beginning and the end of the range is generated and the individual where this random number is mapped to is then chosen. With this selection it is possible that one individual is selected more than once. This process is repeated until the desired number of individuals for the next generation is selected.

The selection pressure with roulette wheel selection is very dependent on the fitness values. If the fitness of all the individuals is very similar, then the selection intensity is very close to zero. If one or a small number have fitness values that are much greater than the rest, then the selection intensity is very high making it very likely that these individuals are selected more than once.

Tournament selection In tournament selection a group of individuals is randomly selected. From this group the individual with the highest fitness is selected. This process is repeated for the desired number of individuals. The size of a tournament is a parameter that can be defined by the user. If the size of a tournament is 1 then it is random selection and when the size of the tournament is the size of the total population than the best individual is selected over and over again.

Similar to the truncation selection the variance of the fitness values are not important only the order of the individuals. The selection intensity is higher when the tournament size is larger.

4.1.2 Recombination

Recombination is the production of new individuals using the information from the parents. The way parents need to be recombined is very dependent on the representation that is used. Most of the time two parents are used for recombination but with some recombination types it is possible to use more.

Discrete recombination

$$Var_i^O = Var_i^{P1} * a_i + Var_i^{P2} * (1 - a_i) \quad i \in (1, 2, ..., Nvar)$$

With random $a_i \in \{0, 1\}$ for every i.

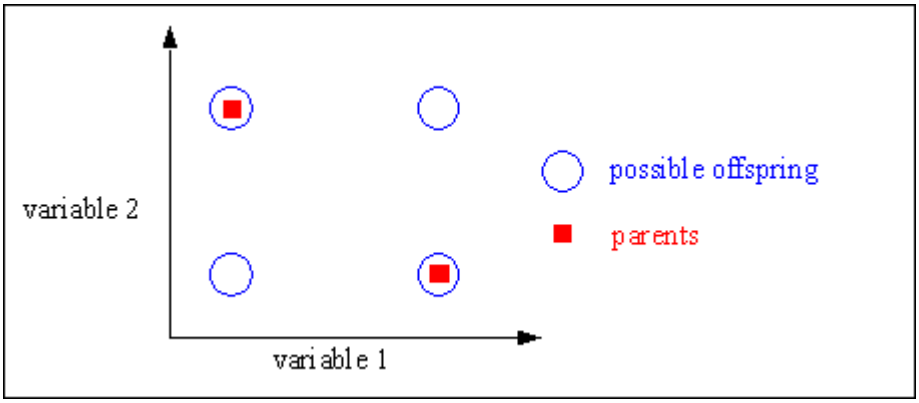


Figure 10: Discrete recombination

For every variable either the value of parent 1 is chosen or the value of parent 2 is randomly chosen as the value for the offspring. It can be used on any type of representation using separate values because there is no need to define a merging operator for the values. This type of recombination is also known as uniform crossover. Figure 10 shows an example of discrete recombination in 2 dimensions.

Real valued recombination With real valued we mean representations using real numbers that stay numbers in the end result, they can not be references. When using real valued numbers there are a lot of possibilities, most of them involve a form of averaging between two corresponding values. The most simple would be to average all the values. A lot of different variations are possible, for example only average a certain percentage of the values.

Intermediate recombination This is a recombination operator for individuals using real valued variables.

$$Var_i^O = Var_i^{P1} * a_i + Var_i^{P2} * (1 - a_i) \quad i \in (1, 2, ..., Nvar)$$

With random $a_i \in [-d, 1 + d]$ for every i and d a fixed value.

This recombination combines the values from the two parents. If d=0 this is called standard intermediate recombination. If d is greater than zero not only values in between the values of the parents are created, but also values more extreme in the direction of one of parents is chosen. This is to prevent the shrinking of the variance every generation. Figure 11 shows an example of intermediate recombination in 2 dimensions.

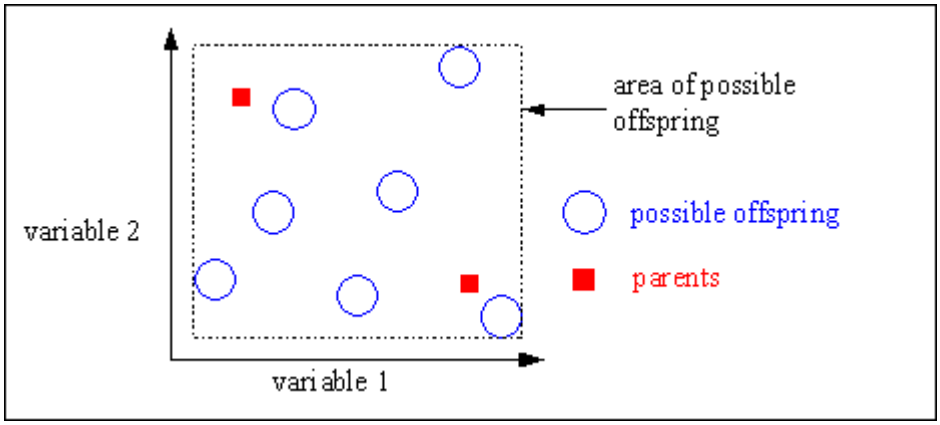


Figure 11: Intermediate recombination

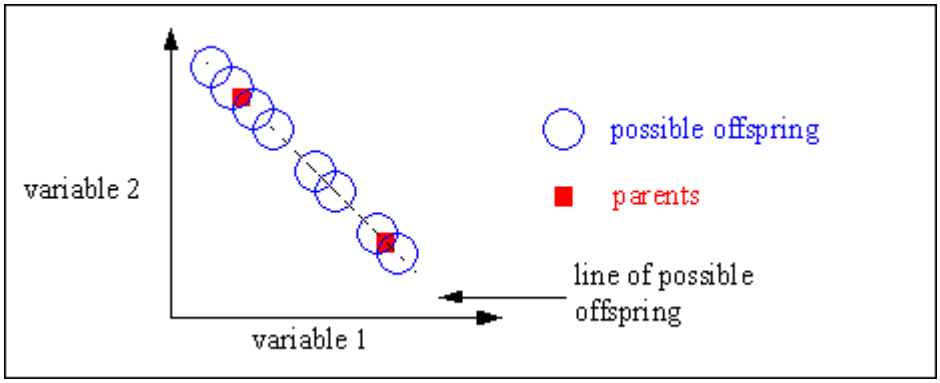


Figure 12: Line recombination

Line recombination Line recombination is almost the same as intermediate recombination but with the main difference that a_i stays the same for all i:

$$Var_i^O = Var_i^{P1} * a_i + Var_i^{P2} * (1 - a_i) \quad i \in (1, 2, ..., Nvar)$$

With random $a_i \in [-d, 1 + d]$ for all i and d a fixed value.

This gives the behavior that all the values of the newly created individual are created in the same direction. Figure 12 shows an example of line recombination in 2 dimensions, this shows both values moving along the same line.

Multi-point crossover This is a very simple crossover operator. Multi-point crossover can also be used for all possible representations of the individuals because no values are merged. Over the whole length of the variables

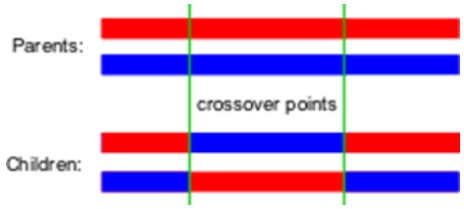


Figure 13: Two point crossover

a predefined number of fixed crossover points are randomly chosen. The easiest is one point crossover; one random crossover point is chosen, then the offspring is created by making one new individual from the first part (before the crossover point) of one parent and the last part (after the crossover point) from the other parent. Two different new individuals are created by also switching the order of the two parents. Figure 13 shows an example of multi-point crossover using two crossover points.

4.1.3 Mutation

Binary mutation Binary representations are used a lot, the default mutation for this is to flip one or more random bits. This can be done by defining a fixed number of bits that need to be flipped. Then select a random bit to be flipped and repeat this process until the correct number of bits is flipped. Or a certain chance can be defined for a bit to flip.

Replace by random values This type of mutation can be used for a lot of different representations, the only requirement is that a random value can be created. Then similar to the binary mutation a certain amount of values are replaced by a random value.

Add a random value This can only be used on real valued representations. When adding values, all the values can be mutated because the value that is added can be chosen small enough to not completely destroy the solution represented by the values. A random value between fixed limits can be added. Or even better it is possible to add a random Gaussian value, this has the property that on average the changes are small enough but with a small chance a bigger mutation happens making it possible to escape from local maxima.

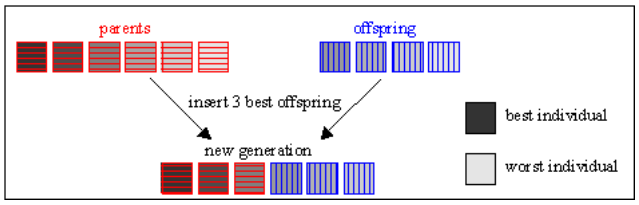


Figure 14: Elitist fitness-based reinsertion

4.1.4 Reinsertion

The method of reinsertion greatly depends on the number of offspring created, because the total amount of individuals must be constant.

Pure reinsertion When using pure reinsertion the complete generation of parents is replaced by a new completely new equally large generation of offspring. A disadvantage is that it is very likely that a very good solution from the previous generation is replaced by less performing offspring, thus throwing away very good solutions.

Uniform reinsertion When using uniform reinsertion less offspring than parents are created, all these individuals replace randomly selected parents. The advantage of this approach is that it introduces a bit of randomness in the search direction. A disadvantage again is (but a lot less likely than with pure reinsertion) that good solutions can be replaced by less performing offspring.

Elitist reinsertion Less offspring than parents are created and these individuals replace the worst parents. The best solutions are never lost using this type of reinsertion.

Fitness-based reinsertion This is not actually a new type of reinsertion but it is a solution that can be combined with the other types. When using fitness-based reinsertion more offspring is created than that is needed for insertion, then these created individuals are tested using the fitness function. From these tested individuals the best are chosen for insertion. An advantage is that better individuals are inserted. A disadvantage is that a lot of extra time is used every generation, especially if it takes a lot of time to evaluate the fitness. Figure 14 shows a schema of a combination of elitist reinsertion with fitness-based reinsertion.

4.1.5 Evolutionary neural networks

One possible representation that can be used with evolutionary algorithms are neural networks. The most basic representation is a neural network with a fixed structure and use the evolutionary algorithm to change only the weights. These weights can be represented as a list of real valued variables, making it possible for normal real valued crossover operators to be used. There is however a bit of a disadvantage using evolutionary algorithms on neural networks, different neural networks can learn the same relation between the same inputs and outputs but using different connections. When using crossover on two of these networks it is not guaranteed that this knowledge is kept.

There are also evolutionary systems that not only evolve the weights but also the number of neurons and the connections between them. One of them is NEAT [13] which will be discussed later (others are "Enforced sub-populations" [3] and "Symbiotic adaptive neuron evolution" [8]).

4.2 Supervised learning

When using supervised learning [14][1] the desired solution of a subset of the input states is used as feedback for the learning algorithm. The algorithm should then be able to generalize using this information to return good solutions on all the possible states. In games it is often very difficult to know the optimal solution. But it is much easier to record actions from a human player given different input states. The goal is not to learn the optimal solution but learn to take similar actions as an expert human player. This should give the bot a more human like behavior.

An obvious requirement is that a human expert is needed. This probably is not a problem for a game that is on the market because gamers play ranking games and tournaments making it easy to find very good players (and even recorded matches that can be used for data). But when the goal is to have the AI finished before the game is released it can be a problem because programmers are not always the best gamers.

It is difficult to say if a bot using supervised learning could surpass the level of the human expert. It could be possible that the bot learns not to make some of the small mistakes the human player makes, this is only possible if the human expert does not make the same mistakes every time or else the bot will learn to make the same mistakes. But it is also possible that the bot is not able to learn the subtle differences between certain input states.

Input	Output
Enemy position (vector)	Movement Direction X
Enemy direction (vector)	Movement Direction Z
Enemy in sight	Jump
Enemy last seen timer	Crouch
Enemy active weapon	
Enemy is shooting	
My Position (vector)	
My Direction (vector)	
My Ammo	
My Active weapon	
My Energy	
My Armour	

Table 1: Fight movement representation

4.2.1 Machine learning techniques for FPS in Q3

Zanetti and Rhalibi [18] use a combination between genetic algorithms and supervised learning. Three different tasks of the Quake bot are explored; fight movement, routing in the map and weapon handling. Because this is a supervised learning approach they use data from human players.

For each separate task a different neural network is used. Simple multi-layer feedforward networks are used. A genetic algorithm is used to evolve the weights of the neural networks. One point crossover is used to create offspring. The population size is 30.

Fight movement In this part the goal is to learn the tactical movements of an expert player during a fighting situation. A fixed time interval is used to check for suitable state action pairs. These samples are only recorded when there is a visible enemy, or short after the enemy disappeared (to also record situations where the enemy disappears and reappears again). The used inputs and outputs are shown in table 1.

As can be seen from these outputs the complete control of all the actions is handled by the neural network, nothing from the original code is used for planning or path finding. In 15 minutes of play from the expert, around 5000 samples were recorded. 3000 of these samples are used for learning.

In table 2 the level of precision (not specified) is shown. After only 100 generations the precision is more than 90%. The resulting behavior learned to strafe left and right during enemy fire and to jump. But it did not learn to make any other decisions based on the position in the field. This is a bit

Generation	Fitness
100	90.16
500	90.44
1000	90.52
1500	90.56
2000	90.58

Table 2: Fight movement results

Input	Output
Last enemy position (vector)	Goal checkpoint
Last enemy direction (vector)	
Enemy last seen timer	
Last Goal checkpoint	
My Position X	
My Position Z	
My Position Y	
My Ammo	
My Energy	
My Armour	
Hold Rocket	
Hold Rail	
Hold Plasma gun	
Hold Shotgun	
Hold Grenade launcher	
Hold Lightening	

Table 3: Routing representation

disappointing after seeing the 90% precision, but is probably caused by lack of useful spatial information from the inputs.

Routing The routing behavior is the movements the bot makes when it is in a non fighting situation. The goal is to learn the paths the expert player takes in a certain situation. To simplify the problem checkpoints are created instead of all possible positions in the field. A checkpoint can be a position of an item or an importing routing point. The input of a sample is recorded when reaching a checkpoint, the output is the next checkpoint that is visited. The original Quake III navigation code is used to navigate to a checkpoint. This time examples are recorded when the enemy is not in sight. The used inputs and outputs are shown in table 3.

Generation	Fitness
100	83.05
500	84.78
1000	85
1500	85.31
2000	85.76

Table 4: Routing results

Because of the way this event driven sample recording works a lot less samples are recorded, only 300 in 15 minutes. This should however be very relevant information. The precision on the training again is pretty high (table 4).

There were different resulting networks with a precision around 85% but some of them learned different behaviors. Some learned to pick up health items when low on health. Others learned to collect weapons when the bot was without them. But none of them succeeded in both.

Weapon handling In this network the aiming, shooting and weapon selection is handled. The samples are recorded continuously using the inputs and outputs shown in table 5.

Because samples are recorded during fighting and non fighting situations around 9000 samples were recorded in 15 minutes. Again the precision shown in table 6 is above 90%.

The behaviors resulting from these networks however do not replicate the behavior of the human player. The bot did not learn to aim at the opponent at all. It sometimes did learn to switch weapons but not always to the stronger weapons. And it did not learn to shoot at all. The results were a bit better for the shooting task when learning samples from only fighting situations were used.

Results & comments The evolutionary algorithm in combination with the neural networks gave an accurate approximation on the training data. The resulting behavior however was not a competitive playing bot. Testing on only the training data is never a good idea when using supervised learning. It will probably result in overfitting; if the algorithm is stopped earlier, the performance on data not in the training data could be better represented even when the performance on the training data is worse. Some tasks worked better than others. It is difficult to say if the disappointing results are because of the use of the supervised learning or because the tasks were too difficult

Input	Output
Enemy position (vector)	Aim at H
Enemy direction (vector)	Aim at V
Enemy in sight	Shoot
Enemy last seen timer	Choose weapon
Enemy active weapon	
My Position (vector)	
My Direction (vector)	
My Ammo	
My Active weapon	
My Aim at H	
My Aim at V	
Hold Rocket	
Hold Rail	
Hold Plasma gun	
Hold Shotgun	
Hold Grenade launcher	
Hold Lightening	

Table 5: Weapon handling representation

Generation	Fitness
100	87.6
500	89.9
1000	90.6
1500	91.23
2000	91.35

Table 6: Weapon handling results

to solve for the neural networks with the used inputs and outputs. For some basic tasks it is probably better to use a simple predefined system. For example shooting when there is a clear line of sight and the angle to the bot is within a certain threshold (possible to learn this) directly at the opponent. And aiming at the opponent should probably work a lot better using simple calculation using the angles.

4.3 Reinforcement learning & Back-propagation

Reinforcement learning is learning from interaction [14]. This is very similar to how humans and animals learn. When using reinforcement learning the goal is to map the different states to good actions. The learner is not told which action to take. In interactive problems it is often impractical to obtain examples of desired behavior, most of the time it is not possible to try all the different possibilities and the best solution is not known.

When using reinforcement learning for every executed action a reward is received, this is a measure for the effectiveness of the action. But there is no feedback what the best action was or what the reward would be for the best action.

Because only feedback is received from actions that were performed there is a need to try different actions to see if they give a better reward. This is called exploration, a disadvantage is that trying random actions usually results in bad performance. The dilemma of trying new actions to learn better solutions or to use actions that gave the highest feedback in the past is called the exploration versus exploitation dilemma.

There are many different algorithms for reinforcement learning, they can be divided in on-policy and off-policy algorithms. On-policy means that only feedback is used from the action that is used for the task. Off-policy means that the algorithm gives one action that should be executed but uses feedback from another action.

Back-propagation can also be used with learning systems, like for example supervised learning, using neural networks. It is placed in this chapter because it is used in the related article, so it can be explained with an example.

4.3.1 Reinforcement learning in Quake III

This project [6] also uses neural networks to learn weapon selection in Quake III. But instead of evolutionary algorithms it uses reinforcement learning.

The structure of the weapon selection is very similar to the main experiment in this thesis because it is a spin-off project from that. The neural

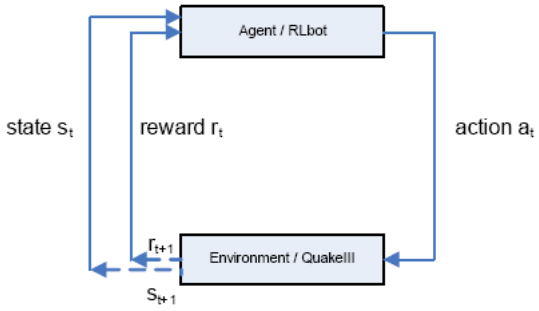


Figure 15: Interaction between the agent and the environment

network has 11 input neurons, 15 hidden layer neurons and 8 output neurons. Both input and output neurons use an extra bias neuron. One neuron stands for the health, one for the armor, one for the distance to the current enemy and 8 neurons for the amount of ammo of the weapons. The output neurons represent the 8 different weapons (the gauntlet is left out). The initial weights are randomized.

The Monte Carlo algorithm was used for learning. An episodic environment is needed, by default there are no good episodes to learn from. Episodes were chosen to be as short as possible, so they chose the time the bot is holding the same weapon. More formal; an episode starts when the bot is placed in the level and lasts until the bot dies, the bot selects another weapon or when the game ends. When an episode ends a new episode is immediately started. The short episodes are important when using Monte Carlo learning, because the longer the episodes the more time it takes to process an episode. The total time to process all the states stays the same if the episodes are longer, but the time per episode is longer. This time per episode is critical in a real time game situation, how often it happens is less of a problem.

A special level was created containing all the possible weapons to simplify the task and to speed up the learning process.

Every time the engine calls the weapon selection function, the neural network is used to determine the best weapon with chance $1 - \epsilon$. Were ϵ is a user defined chance for how often a random weapon is chosen. Every weapon selection results in a state action pair with reward for the current episode (figure 15). This is the Monte Carlo update function:

$$Q_{k+1}(s, a) = Q_k(s, a) + \alpha[R_{k+1} - Q_k(s, a)]$$

The update formula is used to process all the states of an episode directly when an episode is finished. An important part of using reinforcement

learning is choosing the right reward function. This is the formula that was used.

$$Reward = DamageDone - max(0, ((f1 * AmmoSpent) - DamageDone)) - f2 * DamageSelf)$$

With $f1 = 3$ and $f2 = 0.5$.

Back-propagation Back-propagation is used to update the neural network. When using the Monte Carlo update steps the values of the network are updated using back-propagation. The difference between the output of the network and the Monte Carlo algorithm is calculated and the weights of the network are adjusted to lower this error. First the delta for the output neuron is calculated:

$$\delta_j = (t_j - a_j) f'(I_j)$$

With t_j the desired output of output node j, a_j the actual output of output node j, I_j the weighted input to output node j and $f'(I_j) = a_j(1 - a_j)$ given that f is the sigmoid function.

Then the delta of the hidden neurons is calculated using the data from the output delta:

$$\delta_j = \sum_k \delta_k w_{kj} f'(I_j)$$

With δ_k the delta of output unit k and w_{kj} the weight between output neuron k and hidden neuron j.

Using these deltas all the weights can be adjusted:

$$\Delta w_{ij} = \mu \delta_j a_i$$

With δ_j the delta of neuron j, a_i the actual output of node i, w_{ij} the new weight between neurons i and j and μ the learning rate set to 0.35.

Conclusions and remarks The results are shown in figure 16. Overall the performance was about equal to the original. The learning went very fast, only 250 frags are needed. It is a pitty that they used a custom level, this makes it hard to guess how fast learning would go in a real world scenario.

A difficulty with using such short periods is that there is a small chance that a reward is counted in the wrong period (travel time of the ammo), but this apparently was not a big problem for the algorithm. Because the bot keeps exploring the performance will never be perfect, maybe it would be better to decrease the exploration over time or use the algorithm to make a pre-learned behavior and use it without the exploration.

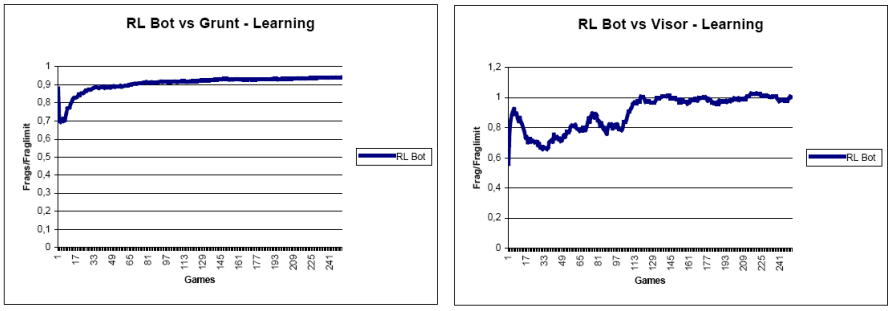


Figure 16: Results from RL weapon selection

4.4 NEAT

NEAT [13] stands for Neural-Evolution of augmenting topologies. It is a form of evolutionary algorithm on neural networks. Not only the weights of the neural network are adapted but also the structure of the network. This can be helpful because it is very difficult to know how complex the network should be to solve the task efficiently.

Individuals are placed in sub populations (species) of similar individuals. The idea behind this is that individuals should compete with similar individuals from their own species. This should protect topological innovation.

Network representation A network is defined by node genes and connect genes. In a normal neural network a connection is just a weight, when using NEAT they also contain a weight but there are some extra parameters. Every connection has an input node and an output node to specify from which to which node the connection goes. There is also a parameter to specify if the connection is active, this makes it possible to remove or to replace certain connections, but keep the information that they were once created. Every connection also stores an innovation number to specify when it was created. The innovation number is a global number, for every new gene created in any network the global innovation number is updated by one and is assigned to that gene. Figure 17 shows how these two types of genes represent a network. As can be seen in this example also recurrent networks can be created using NEAT.

4.4.1 Network operators

Add-connection The add-connection operator adds a connection between two existing nodes that did not have a connection yet. The weight of the connection is set to 1 by default. Figure 18 shows an example. The innovation

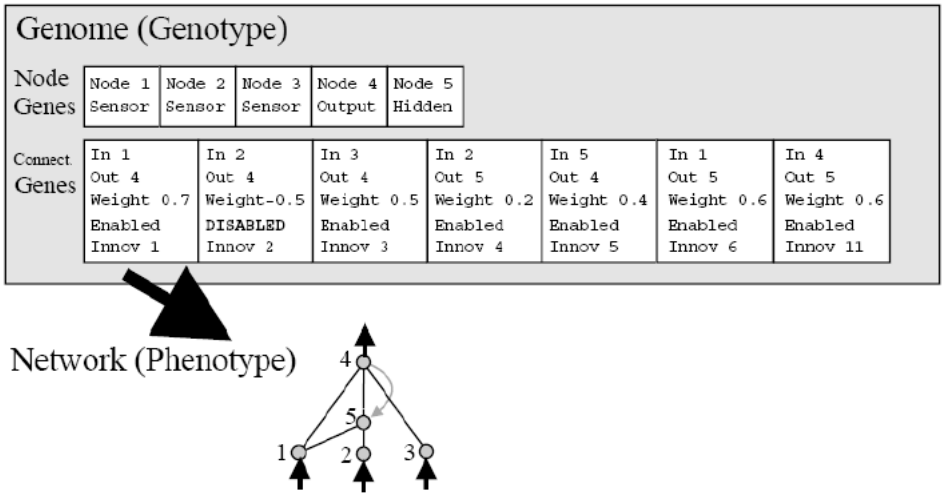


Figure 17: Representation of a network using NEAT

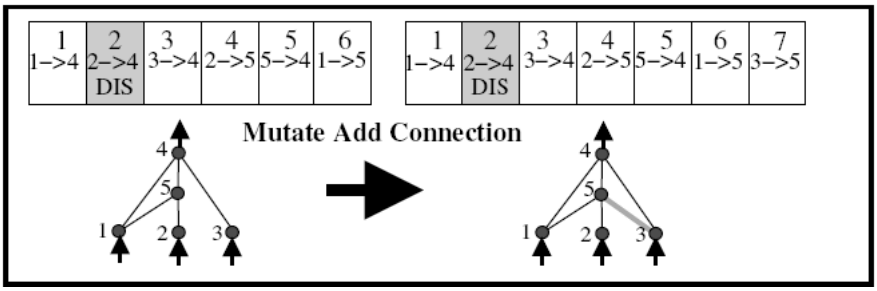


Figure 18: Adding an extra connection in NEAT

number is only one higher than the highest existing innovation number this is because this is the first gene created since the creation of a previous node.

Add-node The add-node operator creates a new node between two existing nodes that already have an existing connection. The existing connection is disabled and two new connection genes are created to connect the old nodes with a newly created node. Figure 19 shows how this process works. The innovation number is two higher than the highest existing innovation number; this means that before this operation a gene is added to another network (in this case the connection gene in the add-connection example).

Recombination The recombination operator can be used to create a new network from two parents. For the crossover operator the innovation numbers become important. Figure 20 gives an example of recombination. When

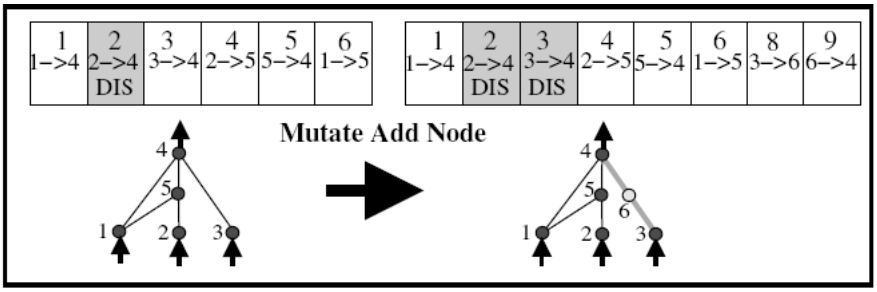


Figure 19: Adding an extra node in NEAT

looking at the networks they appear quite different. But when lining up the connection nodes, using the innovation numbers, it becomes clear that several of the genes match up directly. Genes that do not match are either disjoint (inside the range of the other parent) or excess (outside of the range of the other parent). For genes with the same innovation number, one of the genes of the parents is randomly chosen. Genes that do not match up are only inherited from the parent with the highest fitness. If both parents have equal fitness the disjoint and excess genes are randomly added. The example in figure 20 is obviously from two parents with equal fitness because disjoint and excess genes are used from both parents. During crossover a chance can be defined to enable disabled connection genes. No new genes are created, thus the innovation number remains the same.

Comments A possible advantage of using NEAT is that operators allow the network to expand instead of only changing the weights. This can be good because this makes the network able to learn new things without losing old information. A disadvantage is that no guarantee on the runtime of the algorithm can be given, because the networks can keep growing. The use of sub populations makes it easier to escape local maxima.

4.4.2 rtNEAT

This an extension on NEAT to make it possible to be used in real time learning. Normally when using evolutionary algorithms a large part of the population is replaced every generation. This is not really suited for real time learning because changes in the behavior can be too dramatic and the level of play is not optimal because most of the time not all the new individuals perform as good as the current good solutions.

The solution to this is rtNEAT [12]. When using rtNEAT most of the population is kept the same and only the worst individual is replaced every

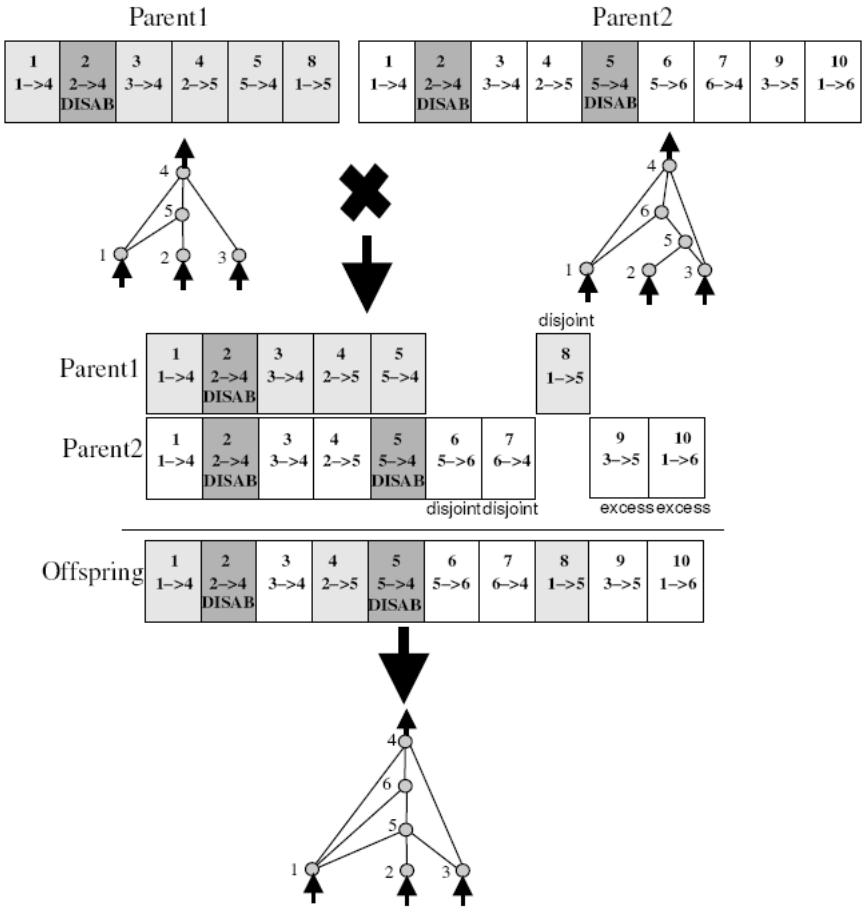


Figure 20: Crossover of two parents with equal fitness using NEAT

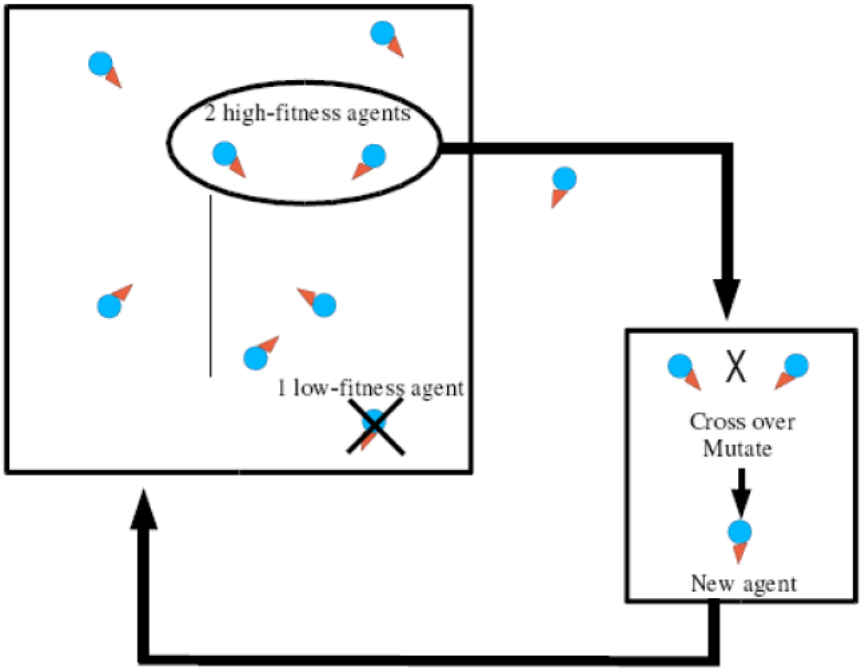


Figure 21: rtNEAT

generation. The flow of this algorithm can be seen in figure 21.

First the worst individual is removed. This is done using the adjusted fitness; adjusted fitness is the fitness of the individual returned by the fitness function, divided by the number of individuals in the species of the individual. This helps small species to survive.

Second the average fitness of all the species is calculated (not using the removed individual). Using these average fitness values a parent species is chosen:

$$Pr(S_k) = \frac{F_k}{T_{tot}}$$

The probability of choosing a parent species S_k is proportional to its average fitness divided by the total of all species average fitness.

The compatibility threshold of the adapted species is also adjusted. The compatibility threshold stabilizes the number of individuals of a species.

Then a new individual is created to replace the one that is removed, using the crossover operator on two parents of the selected parent species. (No information is given how these parents are selected from the species, nor is there any reference to any other mutations or operations that are used on the networks).

A disadvantage of using rtNEAT is that every generation only one new individual is introduced and compared to the other individuals. This makes the learning process a lot slower.

4.4.3 Adaptive behavior using rtNEAT

This report [17] is about using rtNEAT for learning in Quake III. The task that is being learned here is role assignment in a capture the flag game.

Capture the flag is a game variant where two teams compete against each other. The goal is to steal the flag of the opponent and bring it back to your own base. Of course it is also important to defend your own flag, so that the opponent can not capture your team's flag.

There are three different roles that an agent can have. The agent can be aggressive; this type of agent concentrates on capturing the enemy flag more than on killing opponents or defending the team's own flag. The agents can be role defensive; this type of agent concentrates on defending the team's own flag more than on killing opponents or capturing the opponents' flag. And there is the roaming role; this agent concentrates on killing other opponents.

In the original code the team leader decides which role an agent uses. The roaming role is not used in the original code. A tabular representation with three binary (gives a table of 8 long) input variables is used for deciding the number of attackers and defenders. There are two inputs for the states of both flags and one input indicates if the team is winning or losing. In the adapting case also no roaming agents are used. The total number of agents is fixed and there are only two options, this means only the number of agents using one option has to be learned (the rest are using the other option).

Only the number of attackers (and thus the number of defenders) is adapted in this experiment. The input is changed to only use the states of the flags, leaving 4 possible inputs. Five bots are in a team so there are five possible outputs.

The actual decision of which agents become an attacker is not changed, only the number of attackers. The behavior of the bots using a certain behavior is also kept original.

The game is played using the individual with the best fitness value, only one network can be evaluated at the same time. The following fitness function is used:

$$F_t = (Score_f - Score_o)/TimePlayed$$

With $Score_f$ the score of the own team and $Score_o$ the score of the opponents. A generation lasts from one flag status (own or enemy) change to the next. To make sure all species are tested this formula is used:

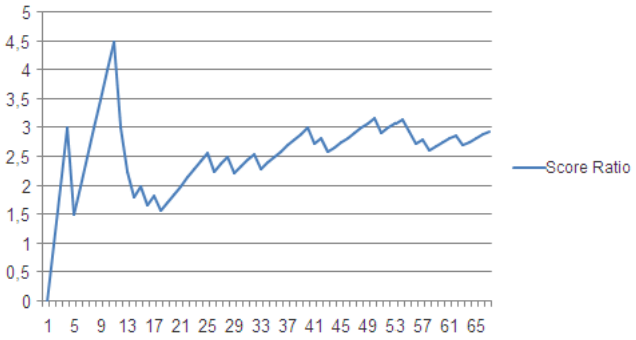


Figure 22: Dynamic vs Static

$$F_o = F_t / O_{evaluations}$$

This means that a species that only has been evaluated a few times has a much higher chance of being selected. The selection is always greedy using this fitness function.

Results Three different experiments were run. First an adapting team against a default static team, the results are shown in Figure 22. The dynamic team performs a lot better than the original after a few generations, showing that learning can really boost the performance of the team play.

The second experiment is of a dynamic team against a similar dynamic team, shown in Figure 23.

No special occurrences here, just very similar performance as you might expect. It does show that the solution is pretty stable. The last experiment is of a pre-learned team using the same method against a dynamic team, results are in Figure 24. There is not much difference between the pre-learned team and the dynamic team, a small edge to the dynamic team.

Conclusions and remarks The use of evolutionary algorithms for learning on this task gives a significant improvement on the performance. After a few generations the behavior is pretty stable even when learning against another dynamic team. There is not much difference between dynamic learning and using a pre-learned approach.

The actual task that is being learned is actually very simple, the problem is reduced to only four different inputs and 5 possible outcomes. Using NEAT seems an overkill for this problem. It can easily be represented using a tabular representation or a very simple neural network. The pre-learned

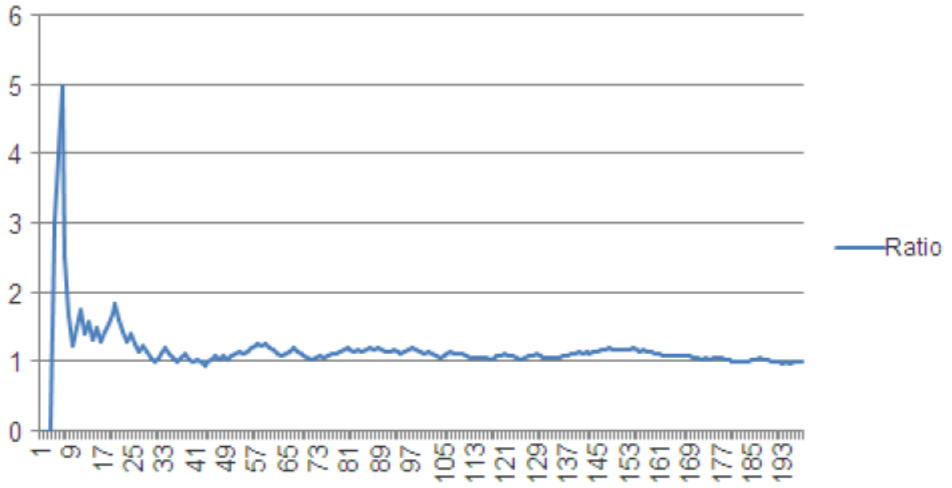


Figure 23: Dynamic vs Dynamic

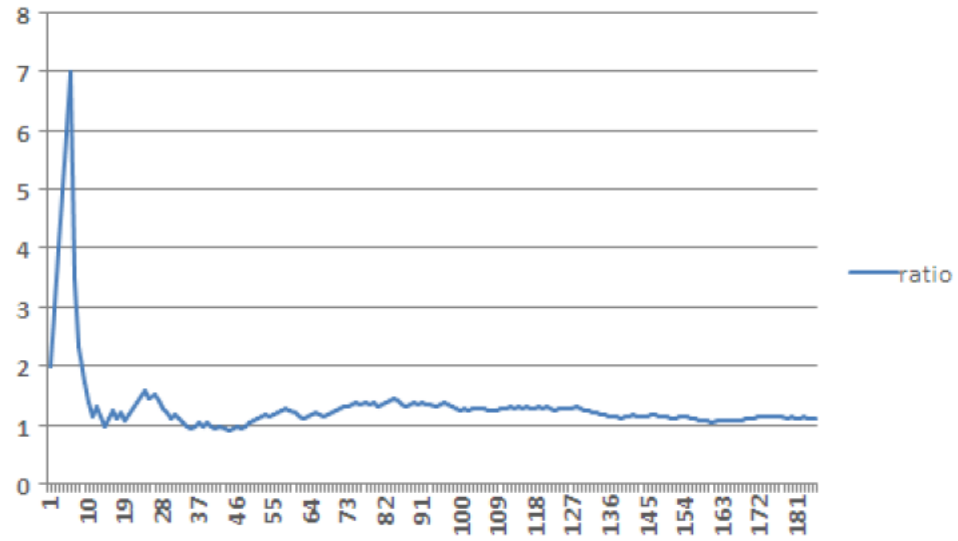


Figure 24: Dynamic vs pre-learned

results can very easily be translated to a (even simpler than the original) tabular representation. The pre-learned approach is actually a static approach using even less inputs, but it performs a lot better than the original code, this means that the original is far from optimal using this setup and level.

It looks like the recombination operator is the only used operator to modify the individuals, if this is really the case no extra connections or nodes are created, making the use of NEAT useless.

5 Experiments

5.1 Algorithm selection

Evolutionary algorithms are used for the experiments because they have certain advantages. First of all they can be used without the need of a human expert. And they can be used to learn at 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 makes it possible that the separate solutions do not need to explore, 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.

5.2 Notes about using Quake III

Quake III is not created as a software platform to do these kinds of experiments. This gives some problems. The first one is the implementation; there is a fixed structure of method calls and parameters. This is not ideal if new implementations need to be created, but it is always possible because the complete source code is available.

A bigger 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. A 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.

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

5.3 Learning tasks

There are a lot of different decision problems in the Quake III bot, 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 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. More information about the weapons and the items can be found in appendix B.

5.3.1 Weapon selection

In the default behavior of the bot 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 bot is used and the new weapon selection if the special neural bot is used. This is done to make it possible to compare against original bots. The moment this function call is used is not affected.

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 can not 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

Ammo of Weapon	Scaling factor
Machine gun	200
Shotgun	25
Grenade launcher	25
Lightning gun	200
Rail gun	20
Plasma gun	100
BFG	40

Table 7: Scaling factors weapon selection

the score of one weapon can 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. The used values can be found in table 7.

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.

5.3.2 Long term item selection

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 bot is not in a fighting situation. When the bot 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 bot 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 bot 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 bot.

Different items are placed in the arena for the bot to pick up. These items all have a beneficial effect on the states of the bot. An item can be a new weapon, ammo or a powerup; more info about the items can be found in appendix B. 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 bot, 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 is the part where is decided which item to go for. How good is a certain item? Should the bot walk further to pick up a stronger weapon or should he first pick up a weaker nearby item? This is the decision problem we are trying to solve with the item selection.

The implementation of this part was a lot harder because there were a lot of nested function calls that all needed to be modified to keep the original code working in combination with the new solution. Even assembler files not included in the project needed to be modified. The structure of the different function calls is shown in appendix A.

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 is checked 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

Input	Scaling factor
Health	200
Armor	200
Dropped	1000
Travel time	2000
Avoid time	50

Table 8: Scaling factors item selection

(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 "avoidtime". 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 scaling factors can be found in table 8.

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.

5.4 Representation

The neural networks in both tasks are basic feedforward neural networks with a bias neuron in the input and the hidden layer. The sigmoid transfer function is used. All the networks that are created have all the weights randomized, this is important when using an evolutionary algorithm.

There is a good separation between al the different parts of the algorithms. For each task a transfer function is created to use the weights of the

networks with the correct inputs and outputs. The evolution of the weights is a separate part, where the weights for all tasks can be evolved simultaneously. The weights of the networks can be stored anywhere as long as they are accessible by both the evolutionary algorithm and the decision function.

5.5 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 arena. 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 can not be learned. Arena pro-q3dm13 (figure 25) 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.

5.6 Fitness function

The fitness function uses the number of kills the bot made in a generation. First was thought to use a fitness with both the number of kills (points scored) made by the bot as well as a smaller penalty if the bot dies. But this would probably make the bots too defensive, because the goal of the game is to get as many points as possible. If a bot dies the score remains the same,

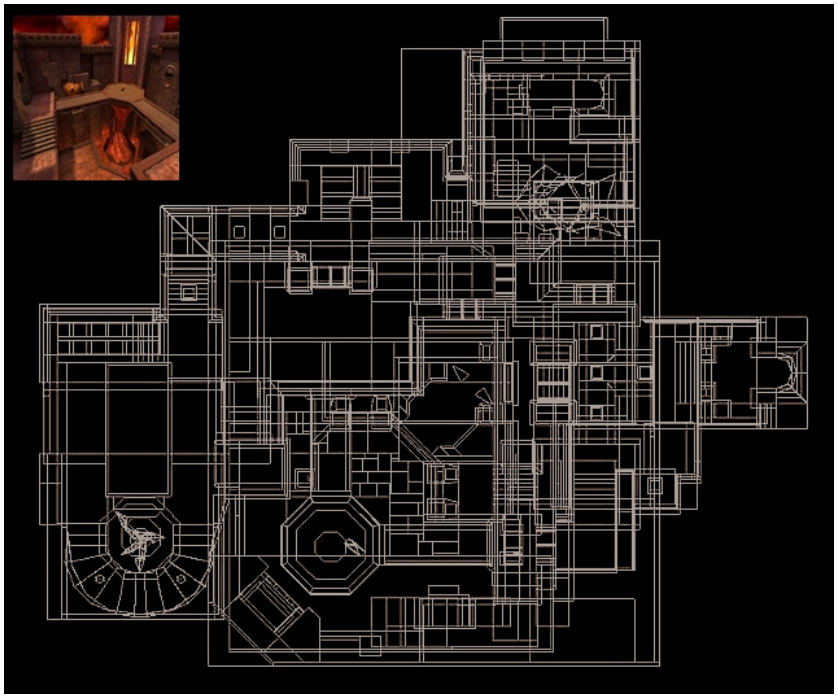


Figure 25: Arena pro-q3dm13

but it is a disadvantage because 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 but this also is not a problem because when comparing the fitness of the bots this is all used. 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.

The accuracy of the fitness function is dependent on the time the bots compete against each other. The longer a generation takes the higher the chance that the best bots will get the best score. Getting an accurate fitness is also important when comparing the results of the bots, now not only the order is important but it would be nice to see how much better or worse a bot performs.

As a comprise between accuracy and runtime a generation lasts 500 points of the best player, this is already very time consuming and the variation is still noticeable.

5.7 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.

5.8 Recombination

The weights of the neural networks are stored as two two-dimensional arrays for each network. 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. The first is the most basic one, when weights are to be combined the average of the two parent weights is used:

```
for ( h=0; h<HIDDEN_NO; h++)
{
  for ( i=0; i<INPUT_NO+1; i++)
  {
    if (random() < AVERAGE_CHANCE)
    {
      child->hiddenWeights[i][h]=
      (parent1->hiddenWeights[i][h] + parent2->hiddenWeights[i][h])/2
    }
    else child->hiddenWeights[i][h]= parent1->hiddenWeights[i][h]
  }
  for ( o=0; o<OUTPUT_NO+1; o++)
  {
```

```
    if (random() < AVERAGE_CHANCE)
    {
      child->outputWeights[h][o]=
       (parent1->hiddenWeights[h][o] + parent2->outputWeights[h][o])/2
    }
    else child->outputWeights[h][o]= parent1->outputWeights[h][o]
  }
}
```

The second one uses line recombination where a random number 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.

```
a = random value between -d and 1+d
for ( h=0; h<HIDDEN_NO; h++)
{
  for ( i=0; i<INPUT_NO+1; i++)
  {
    if (random() < AVERAGE_CHANCE)
    {
      child->hiddenWeights[i][h]=
      (parent1->hiddenWeights[i][h]*a) + (parent2->hiddenWeights[i][h]*(1-a))
    }
    else child->hiddenWeights[i][h]= parent1->hiddenWeights[i][h]
  }
  for ( o=0; o<OUTPUT_NO+1; o++)
  {
    if (random() < AVERAGE_CHANCE)
    {
      child->outputWeights[h][o]=
       (parent1->outputWeights[h][o]*a) + (parent2->outputWeights[h][o]*(1-a))
    }
    else child->outputWeights[h][o]= parent1->outputWeights[h][o]
  }
}
```

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.

5.9 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.

A Gaussian random number generator creates values with a certain chance given by the normal distribution. Values close to the mean are most likely to be generated. Around 68% of the values created are within one standard deviation from the mean and around 95% are within two standard deviations from the mean. The property of small values occurring most of the time and big values happening less is very good for genetic algorithms, local maxima are found by the small changes and the big changes can help the algorithm to try different solutions.

The creation of the random numbers has to be done very efficiently because every generation for every weight of each child that is created a new random number is generated. This all has to be done in real-time. This is why the polar form Box-Muller [2] transform is used; it is a very efficient way of creating Gaussian random numbers.

The implemented algorithm looks like this:

```
{
  static y2
  static use_last = 0
  m = mean
  s = standard deviation

  if (use_last)         /* use value from previous call */
  {
    y1 = y2
    use_last = 0
  }
  else
```

```
  {
    do {
      x1 = 2.0 * random() - 1.0
      x2 = 2.0 * random() - 1.0
      w = x1 * x1 + x2 * x2
    } while ( w >= 1.0 )
        w = sqrt( (-2.0 * log( w ) ) / w )

    y1 = x1 * w
    y2 = x2 * w
    use_last = 1
  }
  return( m + y1 * s );
}
```

A test of this algorithm was done by plotting more than 50000 random numbers (figure 26). The randomly generated values are rounded to the nearest value with one precision digit and for all the values with one precision digit the number of occurrences is counted. The mean for this test was zero and the standard deviation was one.

When using this mutation to modify weights the mean should always be zero or else the mutation will be biased to increase or decrease the values of the weights. A value of 0.05 is chosen for the standard deviation, this should be small enough compared to the values of the weights that are initialized with values between -0.5 and 0.5.

5.10 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.

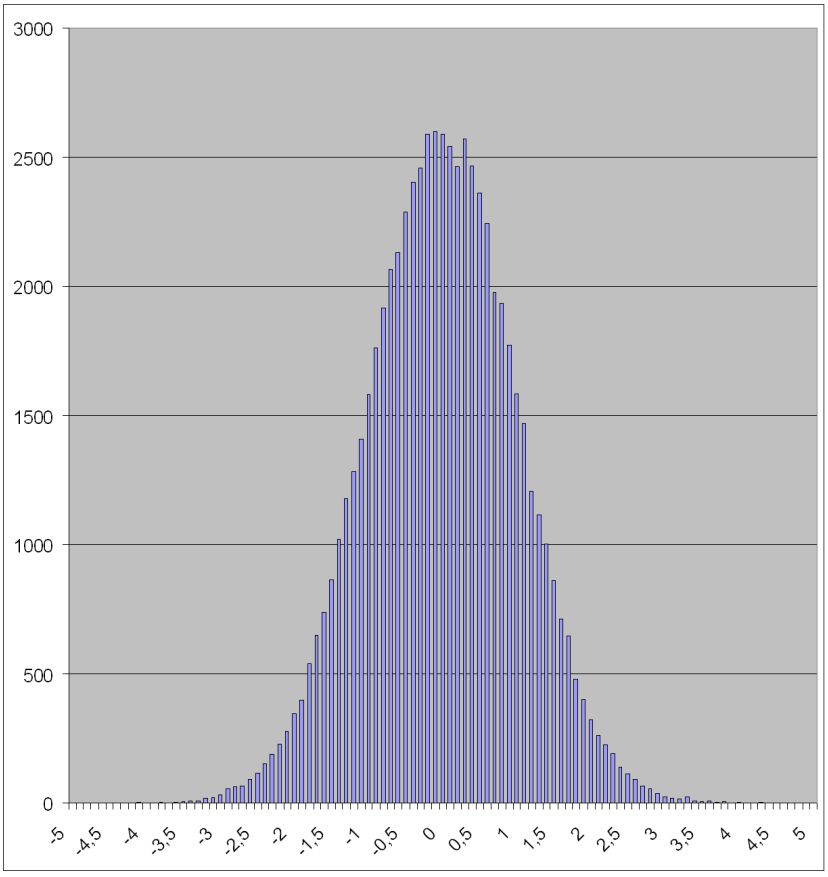


Figure 26: Polar form Box-Muller test

6 Results

In this chapter we will look at the results of the algorithm described in the previous chapters. 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. Also some observations are discussed.

For all the different parameters, multiple experiments are performed, these are called series. 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. The reason is 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 of all the series there also is a line with the average performance of the best learning bots from all the series. 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 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 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.

Parameter	Setting
Population size	6
Selection	truncation selection 50%
Mutation; standard deviation	0.05
Mutation; mean	0
Learning weapon selection	Enabled
Learning item selection	Enabled
Crossover type	Average with chance
Crossover chance	10%
Number of hidden neurons weapon	20
Number of hidden neurons items	30

Table 9: Default parameters

6.1 Default parameters

These are 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 9.

When looking at the separate series (figure 28) 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 are not able to generate offspring to jump out of these local maxima. Still all the series perform at least as good or better than the original approach. When looking at the average of all the series (figure 27) the performance is 10% to 20% better than the original code.

6.2 Weapon only

The same parameters as in the default experiment (section 6.1) are used but only the weapon selection is learned and the item selection is done using the original Quake approach.

All the series reach a reasonable stable performance after around 15 generations. The performance is a little bit better than the original code (figure 29). When looking at the individual series (figure 30) 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

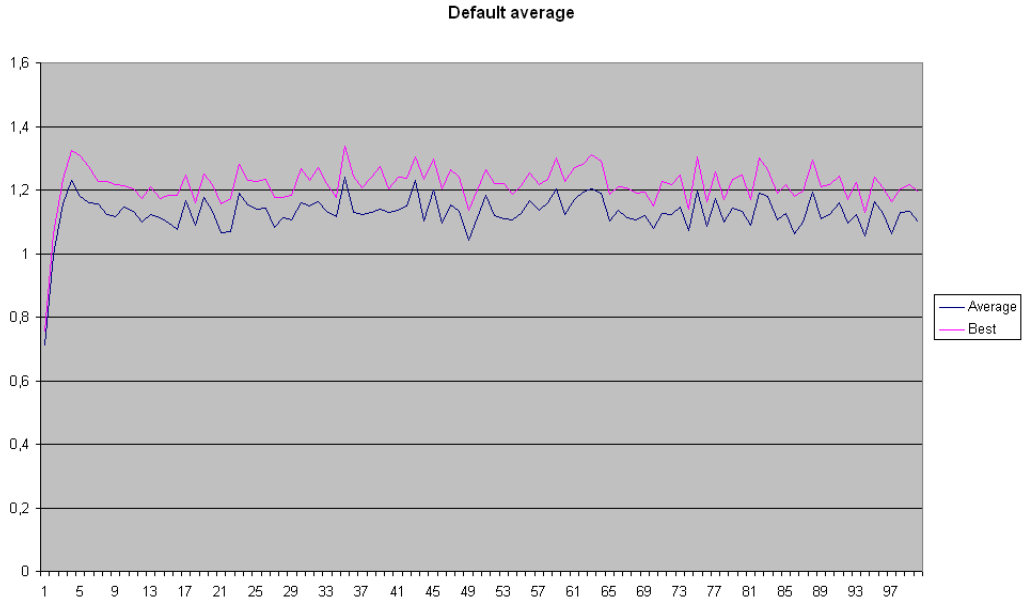


Figure 27: Default parameters

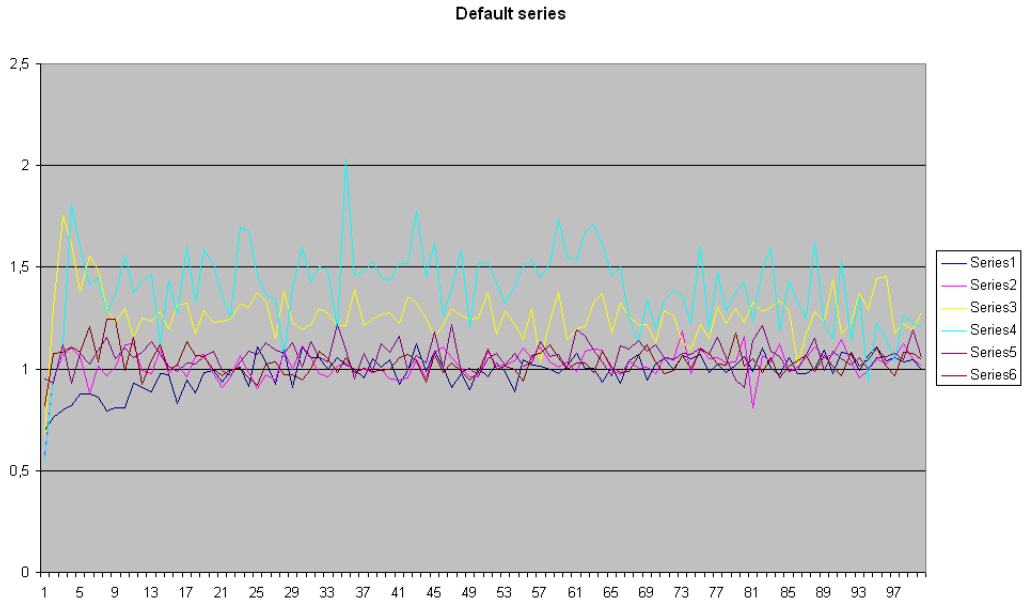


Figure 28: Default parameters

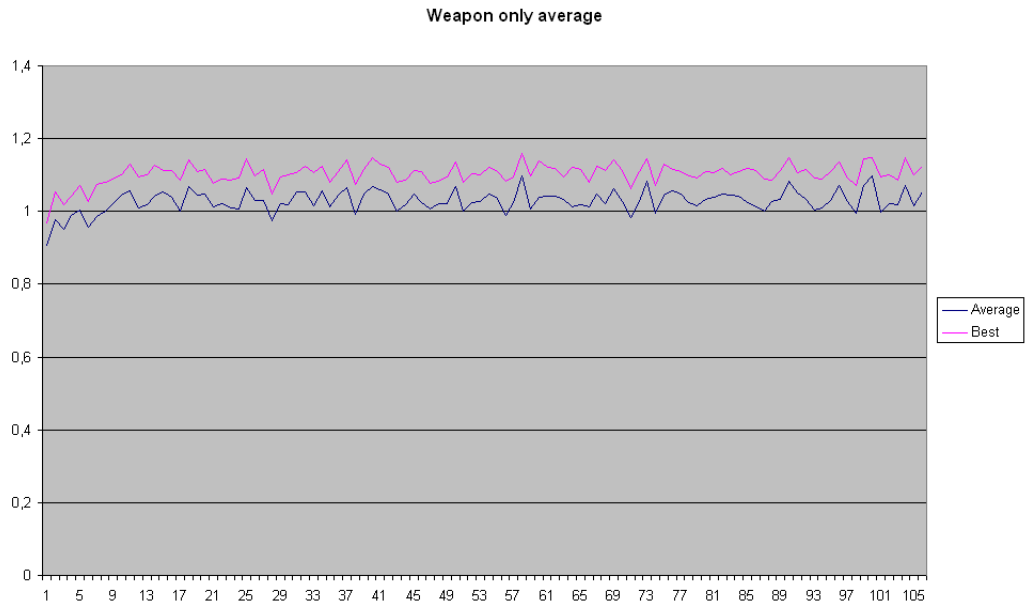


Figure 29: Only weapon selection

on almost exactly the same task (Section 4.3.1).

6.3 Goal only

In this experiment only goal selection is learned and the weapon selection is handled by the original code.

All the series (figure 32) 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 31) is a lot better than the original approach.

A bit surprising is that in this experiment the performance is a bit better than the default experiment(section 6.1). This could possibly be explained by a certain chance factor of a series getting stuck in a local maxima. Or maybe learning multiple decision tasks all together gives slightly worse results. We did not find a positive influence of learning weapon selection and goal selection at the same time. An important factor in this is that the original weapon selection probably already is a good match with the learned goal selection.

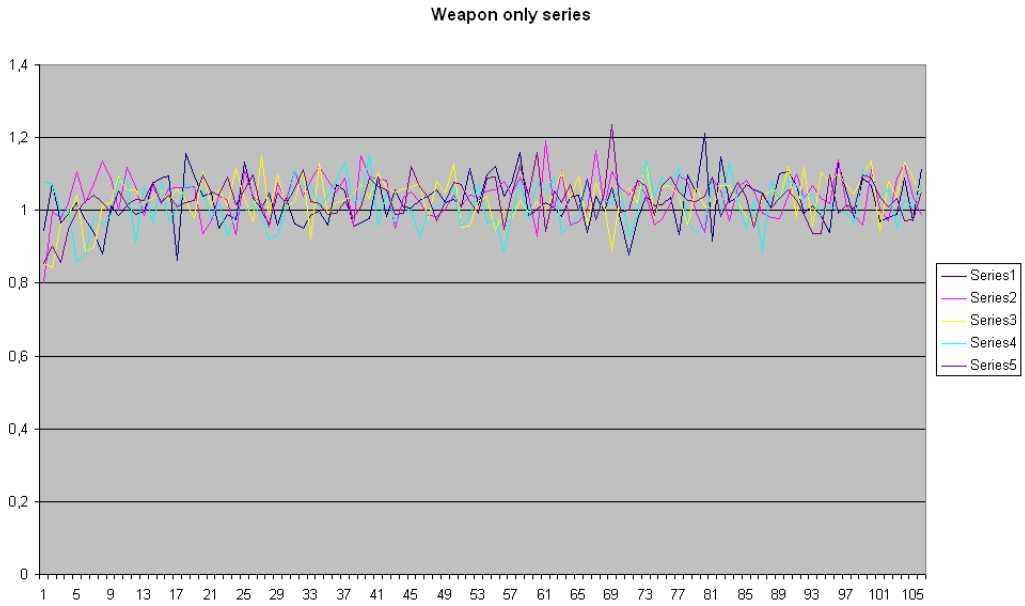


Figure 30: Only weapon selection

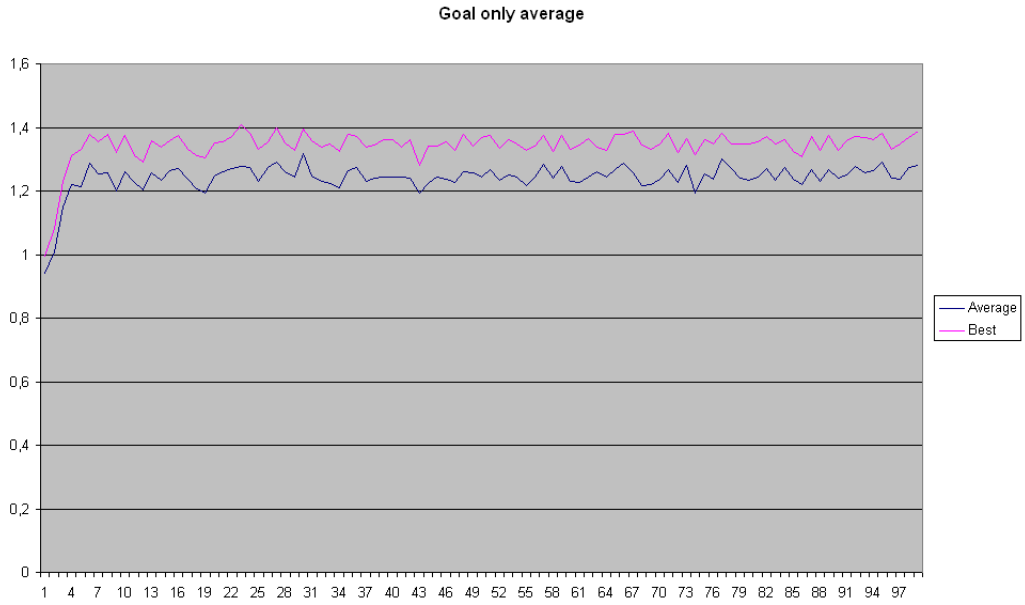


Figure 31: Only goal selection

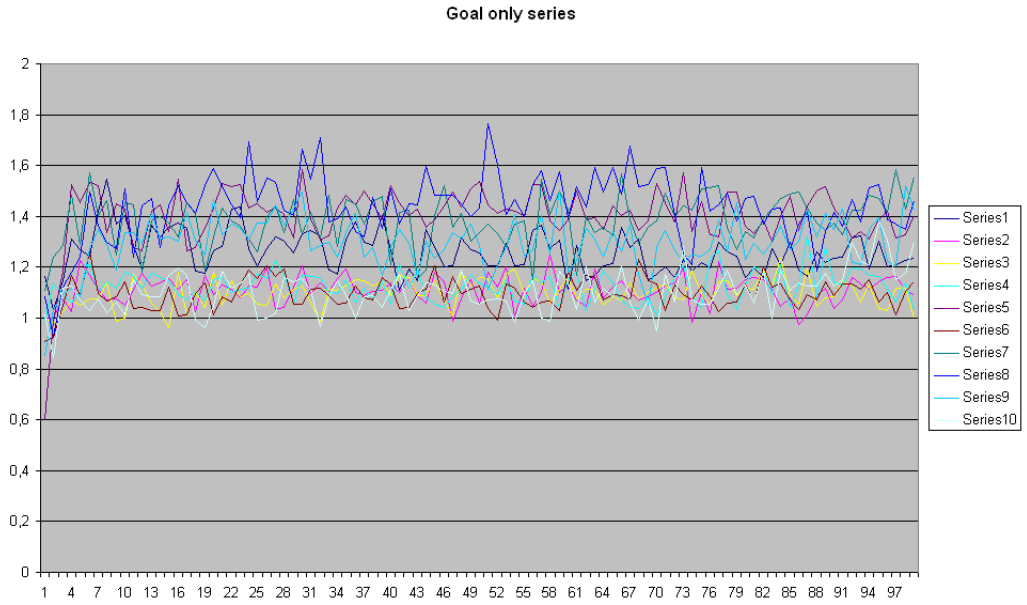


Figure 32: Only goal selection

6.4 Stronger mutation

In this experiment we made a significant change in the standard deviation of the Gaussian mutation on all the weights of the offspring. For the rest the same settings are used as in the default experiment, so both weapon selection and item selection is learned. In all the next experiment this is also the case. We changed this from 0,05 to 0,25 to see what happens with lots of mutation on the weights. All the series (figure 34) 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 33) is also a bit better, this could mean that the extra mutations helps 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 27). This means that newly created offspring with a lot of mutation performs (almost) as good as newly created offspring with a lot less mutation.

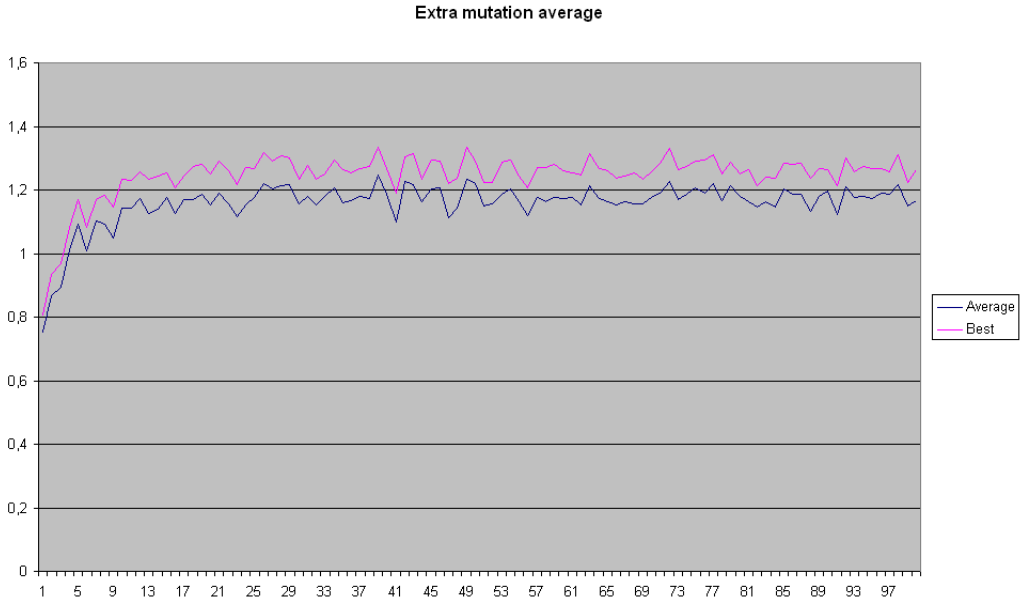


Figure 33: Extra mutation

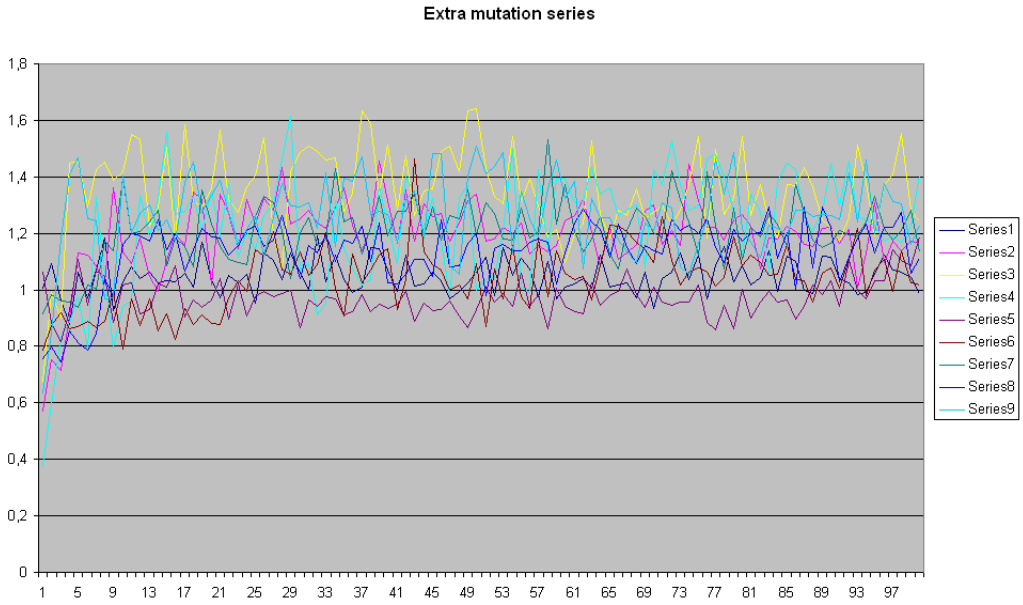


Figure 34: Extra mutation

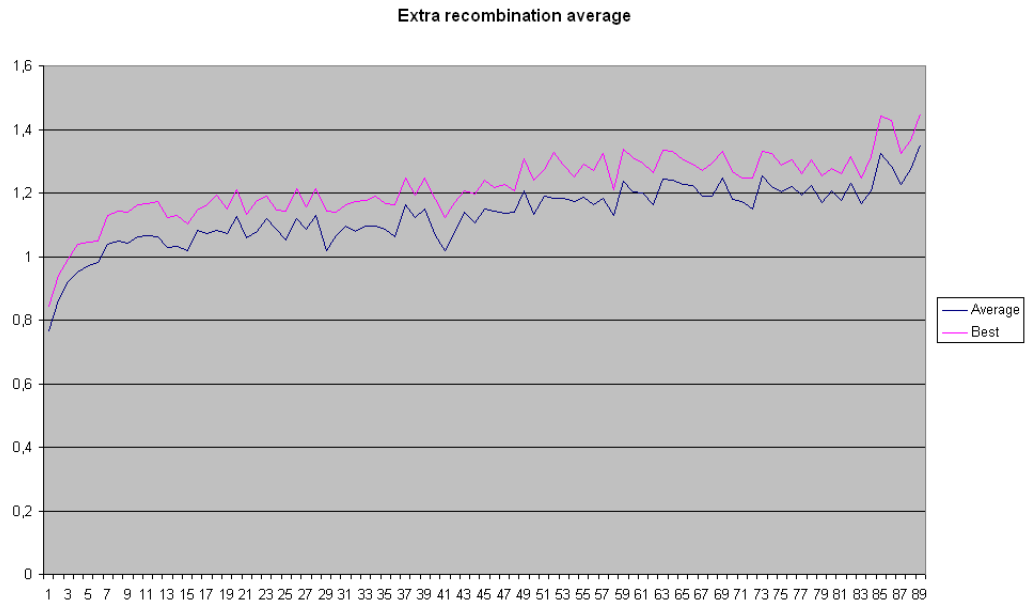


Figure 35: Stronger recombination

6.5 Stronger recombination

In this experiment we are going to 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 36). 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 35) on the problems we are trying to solve. 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.

6.6 Less hidden neurons

In this experiment only the number of hidden neurons is lowered for both decision problems. The number of hidden neurons for weapon selection is changed from 20 to 10 and the number of hidden neurons for item selection

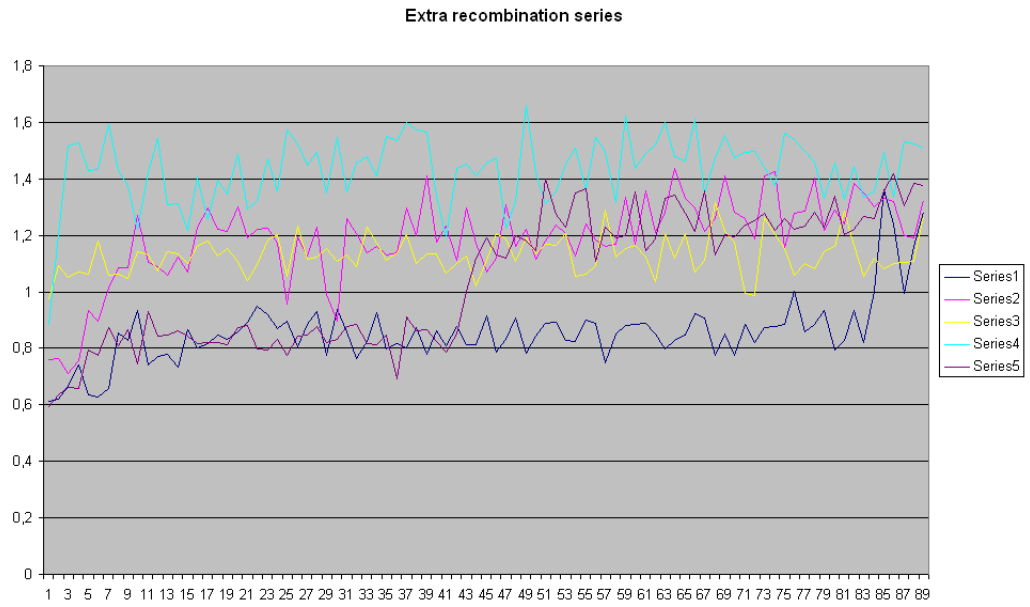


Figure 36: Stronger recombination

is changed from 30 to 15.

The variation between the series (figure 38) is quite large but they all stabilize on a performance equal or better than the original code. This gives a reasonable average score (figure 37). 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.

6.7 Line recombination

In this experiment all the parameters are the same as in the default experiment except a different recombination operator is used. The line recombination with a crossover chance (section 5.8) for the weights is used. The crossover chance is 10%, just as with the default experiment using average crossover. Using this operator the performance increase continues for more generations in all the series (figure 40), 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 39) 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.

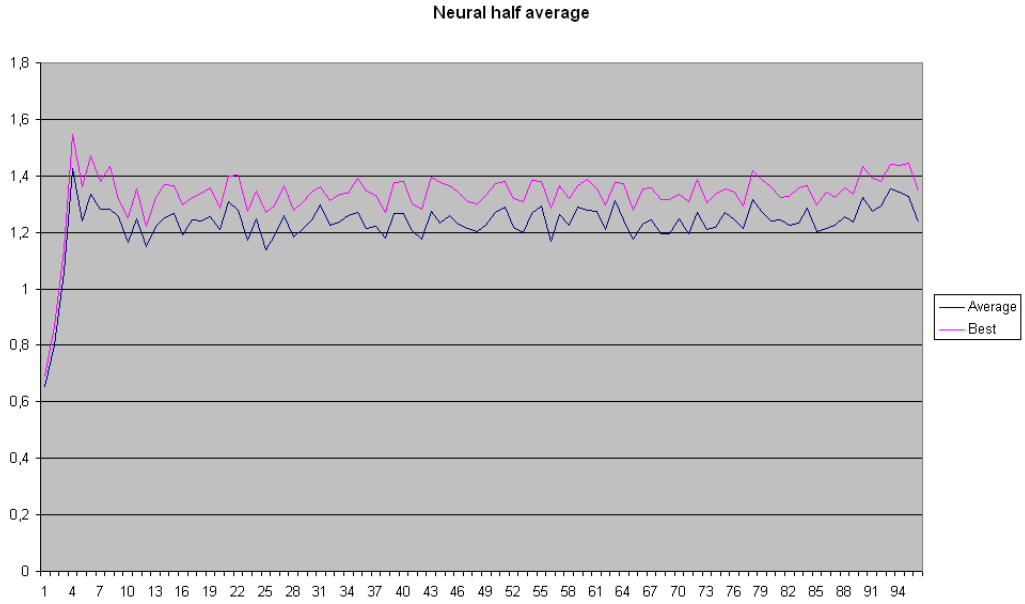


Figure 37: Less neurons

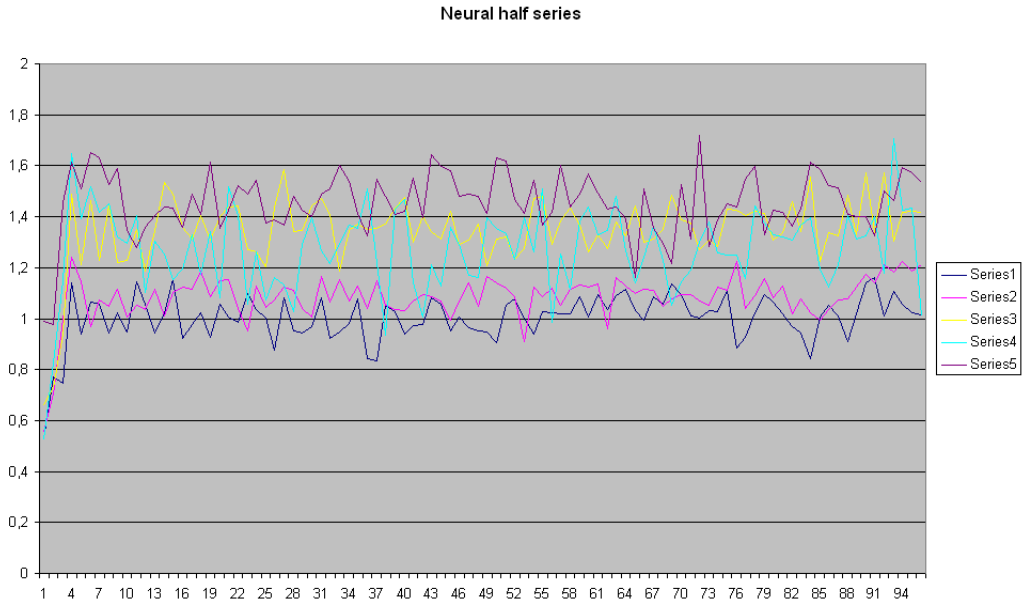


Figure 38: Less neurons

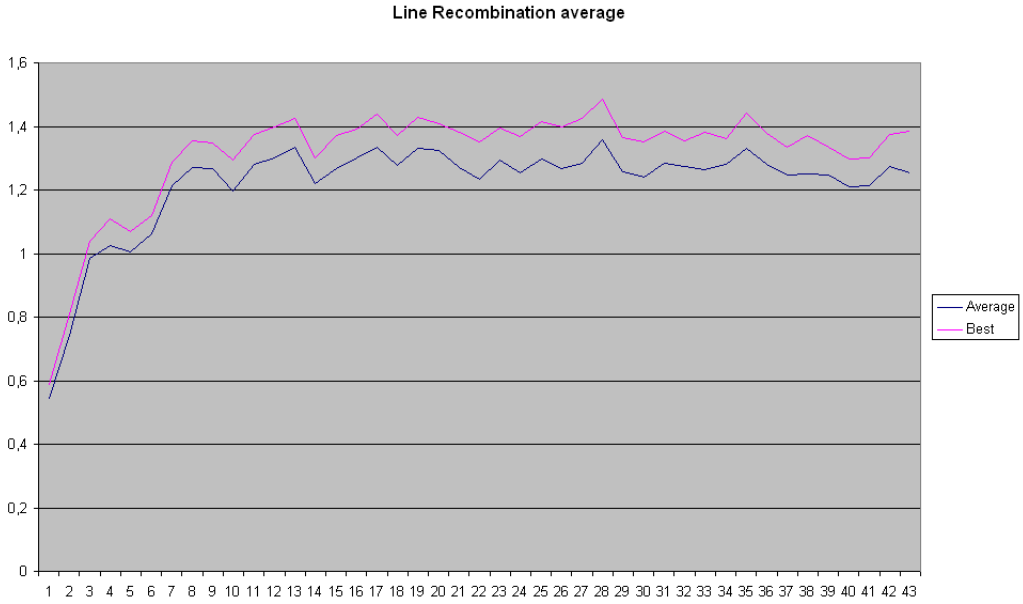


Figure 39: Advanced line recombination

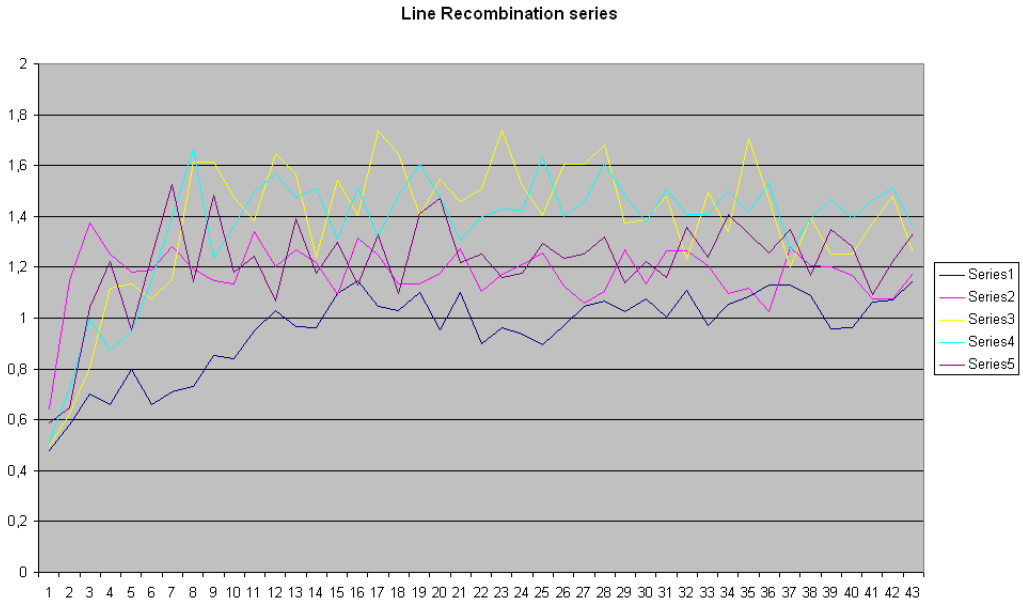


Figure 40: Advanced line recombination

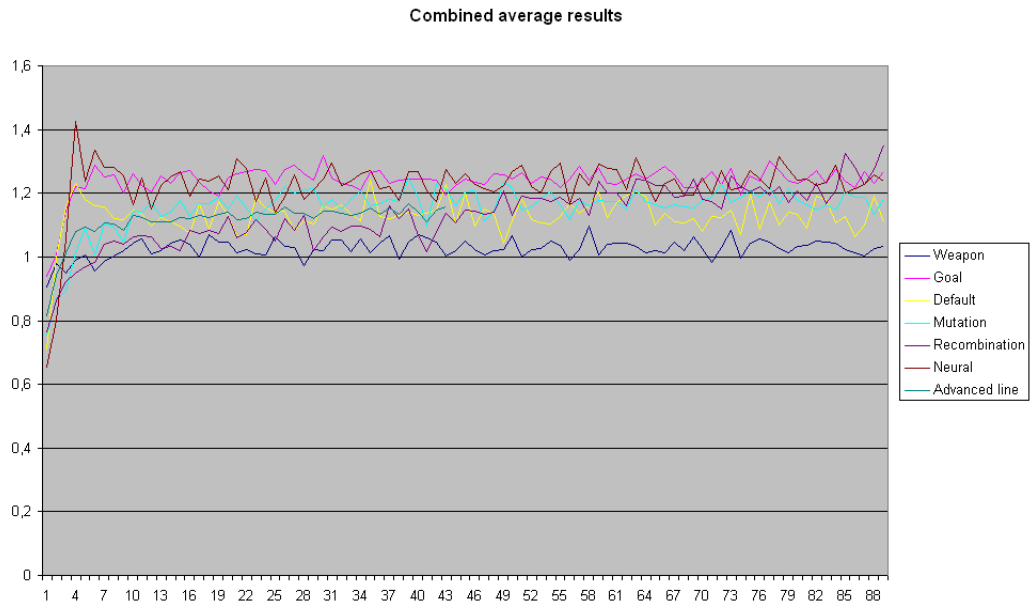


Figure 41: Average results of all experiments

6.8 Combined results

In figure 41 the average performance from all experiments are shown together to make it easier to compare. 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.

6.9 Observations

It is not only possible to measure the score of the bots but it is also possible to observe the bots, this can be done from a special flying "observer" viewpoint or a viewpoint from the perspective of a bot.

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. Or when, from a certain location, the bot can have a significant advantage of scoring more frags.

A problem with this right now is that the bot is not yet learning in which direction it should look when reaching an item. A lot of times it happens that a bot is standing still, looking directly at a wall. When the bot sees or hears an enemy, he switches to "battle mode" and then he will face the opponent bot. But it would be better if he was facing in a more likely direction to spot the opponents. This is a factor that can probably make a big difference.

7 Conclusions

The goal was to test if scientific approaches can be used to solve decision problems in modern commercial computer games and to make it easier to solve these problems. We succeeded in this by creating a system that automatically creates and tunes solutions for solving these decision problems. The resulting solutions performed significantly better than the original hand created and hand tuned solutions.

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).

As shown, this approach can be used in real commercial games, if this system is used in newly created games it would be relatively easy to implement and to use.

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.

There where 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. And in the manual role assignment task that was discussed in section 4.4.3 the whole team would have performed up to three times better if other values were chosen using learning. Also the performance increase in the goal selection is significantly better using learning.

Using learning for subtasks that can be solved by using simple calculations or other logical algorithms (for example aiming or navigation) is not always a good idea as can be seen in section 4.2.1.

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.

8 Future work

The discussed learning approach is already usable for commercial games but there are other things that could be tested or improved.

Population size For this 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.

More decision problems Only two specific decisions problem were tested. There are a lot more decisions problems in most games that could be replaced by a learning approach. More experiments can be done with other decision problems and it would be interesting to see what happens if a great amount of different decision problems are learned simultaneously. In the performed experiments it was not clear if learning multiple tasks at the same time gives better or worse performance, this is something that should be researched further.

Other game genres First Person Shooters were chosen for the experiments, but the techniques that are used can be used on a lot of other games. Most newer games have complex decision problems which could also be solved by a similar learning approach.

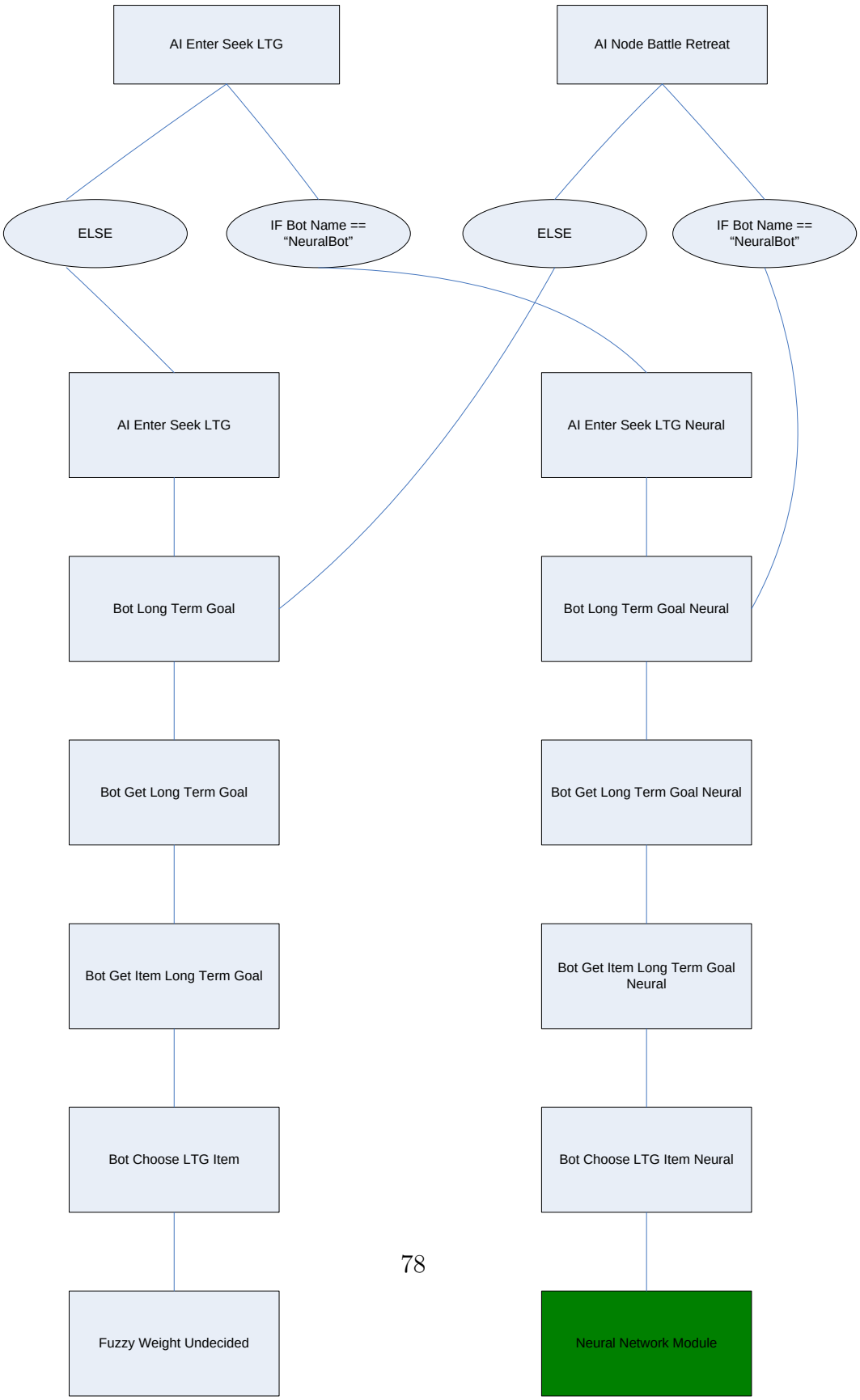
Extra inputs Because it is easy to add extra input parameters it would be useful to see what happens when certain extra inputs are added. For example the weapon selection could use the distance to the enemy and the height difference between himself and the enemy. And for the goal selection it would be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

Generalization and specialization 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.

Behavior analysis More research can be done on analyzing the resulting behaviors. Not only the performance of the bots are important, but it is also nice to test if they are also more fun to play against. It would be nice to test if people perceive the bots as more intelligent. Maybe learned solutions could also be used to come up with new strategies for the games.

A Appendix A


AI Enter Seek LTG
AI Node Battle Retreat
ELSE
IF Bot Name == "NeuralBot"
ELSE
IF Bot Name == "NeuralBot"
AI Enter Seek LTG
AI Enter Seek LTG Neural
Bot Long Term Goal
Bot Long Term Goal Neural
Bot Get Long Term Goal
Bot Get Long Term Goal Neural
Bot Get Item Long Term Goal
Bot Get Item Long Term Goal Neural
Bot Choose LTG Item
Bot Choose LTG Item Neural
Fuzzy Weight Undecided
Neural Network Module

B Appendix B

B.1 Weapons

Gauntlet The gauntlet is an oversized metallic glove with a spinning blade. The weapon does not require any ammunition. A player will have to get up real close to an enemy to inflict damage. However a successful hit inflicts 50 points of damage.

Machine gun The machine gun is the most effective default weapon. This instant hit weapon inflicts between 5 and 7 points of damage per bullet. The rate of fire is quite high. The machine gun is pretty accurate but at large distance quite a few bullets will miss their target. The default amount of ammunition for this weapon is 100. The player can pick up and carry around up to 200 bullets of ammunition.

Shotgun The double-barreled shotgun is a very lethal weapon at close range. Though the weapon appears double-barreled it only uses one shell of ammunition per discharge with a spread of 11 pellets. Each pellet inflicts approximately 10 points of damage. The pellets disperse in a spread pattern, which makes the weapon less effective at a larger range. When a player acquires the weapon it has 10 rounds of ammunition. The player can pick up and carry around up to 200 rounds of ammunition.

Plasma gun The plasma gun shoots hot blobs of plasma at a pretty high rate. A burst of hot plasma will be quite difficult to avoid for an enemy. Each plasma blob inflicts approximately 20 points of damage. The impact of a blob near an enemy can also inflict splash damage. However the radius of the splash damage is tight. When a player acquires the weapon it has 50 cells of ammunition. The player can pick up and carry around up to 200 cells.

Grenade Launcher The grenade launcher fires grenades that bounce around for about three seconds before they explode. The grenade explodes immediately upon striking a player. Littering the floor with grenades on a retreat can be very helpful when trying to shake an enemy of your tail. Each grenade can inflict up to 100 points of damage. The grenades also inflict splash damage when the enemy is within range. When a player acquires the grenade launcher it has 10 rounds of ammunition. The player can pick up and carry around up to 200 grenades.

Rocket Launcher The rocket launcher is one of the most lethal weapons in the game. It fires rockets that inflict around 100 points of damage on impact. When the rocket explodes near a player it will still inflict splash damage. When a player acquires the rocket launcher it has 10 rounds of ammunition. The player can pick up and carry around up to 200 rockets.

Lightning gun The lightning gun shoots a beam of lightning that inflicts about 80 points of damage for each second a player is hit by the beam. Enemies are ensured of a certain death when the beam is held onto them for several seconds. The beam is limited in range so enemies that are far away can get away unharmed. When a player acquires the lightning gun it has 60 rounds of ammunition. The player can pick up and carry around up to 200.

Railgun The railgun is the most powerful instant hit weapon. A slug that inflicts 100 points of damage on a direct hit is ejected at an extremely high speed. However the rate of fire is rather low. The railgun is very accurate

even over large distances. When a player acquires the railgun it has 10 rounds of ammunition. The player can pick up and carry around up to 200 slugs.

BFG10K The BFG10K fires green blobs at high speed that inflict 100 points of damage on impact. The impact of the blob also causes splash damage to nearby enemies. The rate of fire is relatively high. Higher than for instance with the rocket launcher. The BFG10K is only found in very few maps and is often hard to acquire. In return the player gets one of the most powerful weapons in the game. When a player acquires the BFG10K it has 20 rounds of ammunition. The player can pick up and carry around up to 200 rounds of ammunition.

B.2 Items

5 health The player can pick up as many of these health items as can be found. Each green health will add 5 to the player's health.

10 health The yellow health increases the player's health with 10. This item cannot be picked up when the player has 100 or more health.

25 health This item adds 25 to the player's heath. Also this item can only be picked up when the player's health is below 100.

Armor shard When picked up the armor shard increases the player's armor by 5. Armor can be picked up while the player has less than 200. All armor above 100 slowly drains away.

Yellow armor The yellow armor adds 50 to the player's total armor.

Red armor Picking up a red armor adds 100 to the player's total armor.

Mega health This powerup gives the player 100 points of additional health. After a few seconds all health the player has above 100 will slowly drain away.

Battle Suit Wearing this powerup the player is only affected by direct projectile hits. All direct hits will also only apply half the damage. The player can also stay underwater while using this powerup.

Quad damage A player wearing this powerup delivers four times the normal damage with weapons.

Haste When picked up the player can run and shoot 1.3 times faster for half a minute.

B.3 Holdable items

Holdable items are not used immediately when picked up. The player has to press a special use key to use the item. A player can only carry one holdable item at any time.

Med kit Using this powerup will fill your health back up to a hundred. The powerup cannot be used when you have 100 or more health.

Personal Teleporter When this powerup is used the player teleports to a random spawn location in a map. This powerup is ideal to teleport out of a battle if you are not doing all that well.

References

[1] C.[Christian] Bauckhage, C. Thurau, and G. Sagerer. Learning human-like opponent behavior for interactive computer games. In *German Pattern Recognition Symposium*, pages 148–155, 2003.

[2] G. E. P. Box and M. A. Muller. A note on the generation of random normal deviates. *Annals. Math. Stat.*, 29:610–611, 1958.

[3] F. Gomez and R. Miikkulainen. 2-D Pole Balancing Benchmark with Recurrent Evolutionary Networks. *Proceedings of the International Conference on Artificial Neural Networks (ICANN–98).–Skovde, Sweden*, 1998.

[4] Id Software. Quake III Arena. *Computer game, http://www.idsoftware.com/games/quake/quake3-arena/*, 1999.

[5] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[6] Bas Jacobs, Berend Kemperman, Maarten Kous, Teun Slijkerman, and Michiel Vuurboom. AI in Quake III. *Course Report, University Utrecht*, 2006.

[7] R.J. Martin. Unrealscript syntax. 2002.

[8] David E. Moriarty and Risto Miikkulainen. Hierarchical evolution of neural networks. In *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, pages 428–433, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

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

[10] Steve Rabin. *AI Game Programming Wisdom 3 (Game Development Series)*. Charles River Media, Inc., Rockland, MA, USA, 2006.

[11] Pieter Hubert Marie Spronck. Adaptive game AI. *SIKS dissertation series*, no. 2005-06, 2005.

[12] Kenneth O. Stanley, Bobby D. Bryant, Igor Karpov, and Risto Miikkulainen. Real-time evolution of neural networks in the NERO video game. In *AAAI*. AAAI Press, 2006.

[13] Kenneth O. Stanley and Risto Miikkulainen. Competitive coevolution through evolutionary complexification. *J. Artif. Intell. Res. (JAIR)*, 21:63–100, 2004.

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, March 1998.

[15] T. Sweeney. Unrealscript language reference. 2002.

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

[17] S.C.A.M van Weers. Adaptive behavior in quake 3 using neuro-evolution of augmenting topologies. *Master Thesis, University Utrecht*, 2006.

[18] Stephano Zanetti and Abdennour El Rhalibi. Machine learning techniques for FPS in Q3. In *Advances in Computer Entertainment Technology*, pages 239–244. ACM, 2004.