Coevolution and turnbased games
A case study

Joakim Långberg
[Coevolution and turnbased games]
Submitted by Joakim Långberg to the University of Skövde as a dissertation towards the degree of B.Sc, by examination and dissertation in the School of Humanities and Informatics. The dissertation has been supervised by Henrik Jacobsson.

[2005-09-20]
I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.
Signature: _______________________________________________
Coevolution and turnbased games

Joakim Långberg (a02joala@student.his.se)

Abstract

Artificial intelligence plays an increasingly important role in modern computer games. As the complexity of the games increase, so does the complexity of the AI.

The aim of this dissertation is to investigate how AI for a turnbased computer game can coevolve into playing smarter by combining genetic algorithms with neural networks and using a reinforcement learning regime.

The results have shown that a coevolved AI can reach a high performance in this kind of turnbased strategy games. It also shows that how the data is coded and decoded and which strategy that is used plays a very big role in the final results.

**Keywords:** Coevolution, turnbased games, genetic algorithms, neural networks
1 Introduction


As the games have a tendency to become more complex, with more and more variables to keep track of, so does the AI. This puts an increasing demand on the people developing the AI.

Most games use a rule-based approach to control the AI. With a simple game that is probably the most cost-effective solution, but when a game can use hundreds of variables or more, writing specific rules to handle them all can be difficult and time-consuming. Therefore, an alternative approach to controlling the AI can be interesting.

Artificial neural nets and genetic algorithms are both techniques that have shown great potential in learning and adaptation. Coevolutionary algorithms have promised several advantages over traditional evolutionary algorithms.

The aim of this dissertation is to investigate how AI for a turn-based computer game can coevolve into playing smarter by combining genetic algorithms with neural networks and using a reinforcement learning regime.

This report will start by describing the background of artificial neural networks and genetic algorithms and how they are related to computer game AI and learning. This is followed by a detailed description of the game that will be used for this case-study.

The problem description explains why these techniques are interesting to use in this context, and the method and implementation chapters describes exactly how they will be used.

The result chapter illustrates how effective the AI became with this approach, and the similarities and differences between the three strategies tested. This is followed up by the conclusion chapter where the final thoughts are formulated.
2 Background

In this chapter all the concepts that are central to this dissertation are described. These include artificial neural networks, genetic algorithms, learning, and the role these concepts have in the context of modern computer game AI. Also, the purpose of this work, and the game that is going to be used in this case study are described.

2.1 Artificial Neural Networks

An artificial neural network (NN) is a collection of communicating units, called artificial neurons. This is a concept inspired by how the biological neural network in a human brain processes information. It’s a very simple abstraction of how the neurons in the brain connects and send information to each other.

An artificial neuron accepts inputs signals from other neurons or from an external source. The signals are first multiplied with associated weights and then a rule is used to combine these signals together and to add a bias. This summation results in a net value that is used as an input for a transfer function. This function is used to calculate an output signal which can then be sent to other neurons. Figure 1 illustrates the artificial neuron concept.

![Artificial Neuron Diagram](image)

*Figure 1. An artificial neuron accepts inputs I₁⁻Iₙ which are multiplied with the associated weights W₁⁻Wₙ. The results are summated and a bias is added to create a net value. This value is used for the transfer function, and the end result becomes the output value.*

A weight can be either positive or negative, which means it can either excite or inhibit the signal that is being communicated. There are several kinds of transfer functions, and each give the neuron a range of possible output values. A commonly used transfer function is the log-sigmoid function (Callan, 1999) illustrated in figure 2.
A single neuron is capable of very simple calculations, but when many neurons are connected together, the network is capable of performing complicated tasks.

There are different architectures used for connecting neurons together, some are extremely complex, some are simpler. One of most commonly used is the feedforward network (Buckland, 2002). This architecture is illustrated in Figure 3. A feedforward network is divided into layers, and each layer of neurons feed their outputs into the next layer until an output is given. No intralayer connections are allowed. A layer can contain any number of neurons. How many neurons that should be used depends on the problem. Using too many hidden neurons can decrease the speed of the network unnecessarily and can result in overfitting, which means that the network has memorized the training data and therefore its generalization capability suffers. Generalization is a measure of how well the network performs on data not seen during training. Training is described below.

This ability to generalize has made the neural networks useful in many different areas, like face recognition, speech recognition, medical diagnosis, bot navigation and others.

A neural network is created with random weights. The purpose of learning is to replace these initial weights with weights that are close to optimal. There are different kinds of learning regimes used for this:
- Supervised training – an input pattern and a desired output pattern is given to the network for learning. By using a weight-adjusting algorithm the output is made to match the input.
- Unsupervised training – no training patterns are given to the network. Instead it has to analyse the data to find correlations and cluster the patterns.
- Reinforcement training – when the output can only be classified as good or bad. Rewards are given for good behaviour and penalties for bad. The weights are adjusted to maximize awards and to minimize penalties.

Another differentiation between learning regimes are offline versus online, which here is explained in a computer game context:

- Offline learning – means that the network is trained before a game is shipped. After that, its weights are locked and it has effectively become a fixed set of rules. This, naturally, hinders any further adaptation.
- Online learning – allows the network to learn within the game. Adaptation to the player is possible.
- Hybrid schemes – is a combination of the previous two approaches. Offline training is used to reach reasonable performance, and after shipping the weights can still be adjusted.

2.2 Genetic Algorithms

Genetic algorithms (GA) are inspired by the natural selection process described in Darwin’s theory of evolution. In a similar way that species evolve over many generations to become more successful at survival and reproduction, GA evolves over time to produce near-optimal solutions to complex problems.

2.2.1 Biological Evolution

Each member of a species has its own properties, specified by their genes. When two members reproduce, each of them transmits half of their genetic material to the new member which then will contain a completely new set of genes. This process is called crossover and is illustrated by figure 4.

Parents:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Children:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*Figure 4: The crossover process using binary strings*

This means that an offspring with entirely new combinations of genes can appear. It may be more or less fit than both of the parents. Fitness is a measure of a member’s success, and the main concept in Darwin’s theory of evolution. The more fit a member
is, the more likely it is to produce offspring and thereby passing on its own genes. As generations pass by, the tendency shown is that the average fitness level increases.

During the crossover process there is a small probability that an error will occur. When this happens it results in genes that are marginally changed. This change is called a mutation and might mean a disadvantage, no difference, or a slight advantage over the other members. Over many generations, the difference might be significant and bring birth to completely new features. This is only possible because new genes are introduced to the gene pool.

### 2.2.2 Digital Evolution

A genetic algorithm can be used on any problem where the variables to be optimized can be encoded to form a vector. This means that the string can be decoded to represent a solution to the problem. The solution might be poor, or it might be perfect, but every single string represents a possible solution.

The initial population is usually created with totally random chromosome values. This means that the first generation usually performs poorly.

A fitness score is assigned to each chromosome to reflect its performance within the problem domain. When the entire population has been tested, the performance of each chromosome relative to the others can be identified.

A selection process is used to determine which chromosomes should be used for reproducing. The probability for a chromosome to be selected depends on its fitness score. The higher fitness score, the higher probability for being selected.

![Roulette wheel selection of chromosomes based on their fitness score. The chromosomes, f₁ to f₆ each get a slice that is proportional to their fitness score relative to the others. The higher the fitness score, the bigger the slice. The slice represents their chance of being selected for reproducing.](image)

The crossover rate is the probability that a pair of chosen chromosomes will swap their genes to produce new offspring. Experimentation has shown that a good value for this is typically around 0.7 (Buckland, 2002), although some problem domains may require much higher or lower values.

The mutation rate is the probability that a gene within a chromosome will be changed. This is usually a very low value.

The process of genetic algorithms can be summarized like this:
• Find a way to encode the potential solution as digital chromosomes.
• Generate the initial population with random chromosomes.
• Loop
  o Assign a fitness score to each chromosome.
  o Select chromosomes from the population to reproduce.
  o Perform crossover.
  o Perform mutation.
  o If solution not found, start new generation
• Until a solution is found

Genetic algorithms do not guarantee to find the best solution, or even to find a solution at all. But when used correctly they are generally able to create a solution that performs well. Knowing how to solve a problem is not required as long as it can be encoded in a way that the genetic algorithm mechanism can utilize and there is a proper fitness function. This is a main advantage with the technique, which has proven to be excellent at optimization problems.

2.3 Artificial intelligence in computer games

Most people would probably agree that computer game AI has come a long way since the early days of *Pong* and *Pac-Man*. At this point in time, a small number of techniques have come to dominate the game AI architecture. These are the so called rule-based AI techniques, more specifically the finite state machine (FSM) and the fuzzy state machine (FuSM). Every game on the market uses rule-based AI to some degree or another (Woodcock, 1998, 1999). The reasons for this seem to be:

• They are familiar, building on principles that are already well understood.
• They are predictable, easy to test, and straightforward to debug
• They are more familiar than exotic technologies such as neural nets and genetic algorithms.

All put together, the rule-based techniques are used because they work. Often they are combined with scripting, which provides more flexible ways to build and tune them. (Woodcock, 2001). But the predictability aspect mentioned above is a double-edged sword, since one of the biggest complaints from players is predictability of the game AI (Woodcock, 1998).
2.4 Learning

Learning has by many been considered the Holy Grail of computer game AI. In practice, this can involve both letting the AI learn from the player and from the surrounding environment. So far, very few games have successfully implemented it. A recent exception to this is Black & White (2001). In this game the player takes on the role as a god who raises a group of creatures which worships him. To form their behaviours he can either reward or penalise them for certain actions. For example, if a creature is punished every time he gets close to a horse, he will learn that horses are bad for him and go to great lengths avoiding them further on. Reinforcement learning has here been made part of the gameplay. Developers have discussed whether this paradigm could be generalized to other games (Woodcock, 2003). They thought that this kind of interaction would make games far more interesting for players, as well as providing more realistic environments.

Most developers thought it would be great to have an AI that could learn from the player, but few have tried to implement it. The reason for this seems to be that there’s no guarantee that the AI won’t learn anything stupid.

Offline learning is much more common. It’s generally used during development and testing to tweak existing parameters over the course of hundreds of runs. These parameters are then locked before the game is shipped and no more adjustment takes place afterwards.

2.5 GA and NN in computer games

Both genetic algorithms and neural networks are interesting techniques which have shown to have great potential in areas like optimization and pattern recognition. They are also very popular in academic research but have found little use in the game industry. Since these techniques are great for learning, and learning is highly regarded in the game industry, this fact might seem a bit strange. According to the game developers themselves, the main reasons for this are the following (Woodcock, 2002, 2003):

- High development and testing costs
- Difficult to understand the reasons for the resulting behaviour
- Difficult to tune or change something you don’t understand
- Difficult to test thoroughly
- Unpredictable, the AI might learn something stupid
- Support problems - you can’t necessarily recreate any problems the user might report
- Difficult to evolve an AI that is more fun to play against, as opposed to just smarter

Although there is a resistance in the gaming industry for these exotic techniques, a few games do use them. The previously mentioned Black & White (2001) uses neural networks as part of its AI. During the development of Colin McRae Rally 2 (2001) neural nets were also used to let the computer AI learn how to drive (Buckland, 2002). Creatures (1996) is another example. Producer Toby Simpson described Creatures AI architecture like this: (Woodcock, 1997):
It used neural nets, genetic algorithms and fuzzy state machines, built up over the course of several years by a number of British researchers before being adapted into the Creatures game as a practical demonstration of its capabilities. The AI is self-modifying and can adapt itself over time as it interacts with the players, passing down various traits and behaviours genetically to offspring from generation to generation.

Although GA and NN were used successfully in Creatures, other developers viewed it more as an interesting anomaly than the beginning of a trend. (Woodcock, 1998)

2.6 Description of the game

For this dissertation a subset of the game Frontline: Europe (2005) is used. More specifically it’s some of the unit types and their values. These unit types are combined with a custom made playing board made for this case study. This playing board is illustrated in figure 6.

![Figure 6. The playing board. Each side contains 8 spots where unitstacks can be placed.](image)

As shown in figure 6, the playing board is divided into four different zones. The two left ones belong to the attacking country, and the two right ones belong to the defending country. The two inner zones are called frontline zones, and the two outer ones are called support zones. Each frontline zone contains 5 spots available for unitstacks, and each support zone contains 3 spots. A unitstack contains either one or several units of the same type, described below. For example, a unitstack can contain 2 infantry units or 3 anti air units or some other unit type. But unit types can’t be mixed, which means a unitstack can’t contain infantry units and anti air units.

Altogether there are 11 unit types, divided into three groups.

**Ground units**

*Infantry* – Basic ground units. Cannot attack sea units. Attacks against air units are very weak.

*Tanks* – Stronger ground units. Can withstand more damage, and inflict more damage.

*Artillery* – These long-range units plays a very significant role in combat. They can fire and reach all units on the playing board. Although, if the range is 3 they get a decreased attack value. See figure 7 for illustration.
**Coast Artillery** – These units specialize on attacking sea units.

**Anti Air** – These units specialize on attacking air units.

**Sea units**

*Destroyer* – Basic ship units. Ineffective against ground and air units.

*Battleship* – The king of the sea. Effective against ground and air units. Very effective against other ships. Withstand a lot of damage.

*Carrier* – These big ships are not meant for direct combat. But they can sustain a big amount of damage. Their purpose is to carry air units.

*Transport ship* – These units cannot attack other units. Their purpose is to carry ground units.

**Air units**

*Bomber* – These heavy planes are effective against sea units, and very effective against ground units. But they are ineffective against other air units.

*Fighter* – Used for attacking other planes. Cannot attack ground units or sea units.

### 2.6.1 Specification of unit types

#### Table 1. *Ground units. The different values are explained below the tables.*

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Type</th>
<th>Range</th>
<th>Health</th>
<th>Attack</th>
<th>Defence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ground</td>
<td>Sea</td>
<td>Air</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ground</td>
<td>Sea</td>
<td>Air</td>
</tr>
<tr>
<td>IF</td>
<td>Infantry</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>AR</td>
<td>Artillery</td>
<td>3</td>
<td>2</td>
<td>7*</td>
<td>2*</td>
</tr>
<tr>
<td>TA</td>
<td>Tanks</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>CO</td>
<td>Coast Artillery</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>AA</td>
<td>Anti Air</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*If range = 3 then attack value -2*

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Type</th>
<th>Range</th>
<th>Health</th>
<th>Attack</th>
<th>Defence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ground</td>
<td>Sea</td>
<td>Air</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ground</td>
<td>Sea</td>
<td>Air</td>
</tr>
<tr>
<td>DE</td>
<td>Destroyer</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>BA</td>
<td>Battleship</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>CA</td>
<td>Carrier</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>TS</td>
<td>Transport ship</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

*Range* – How far away this unit can attack an enemy unit. See figure 7 for illustration.

*Health* – How much damage this unit can take before being destroyed.

*Attack* – Shows the attack values this unit has against respectively a ground unit, a sea unit and an air unit.

*Defence* – Shows the defence values this unit has protecting from respectively a ground unit, a sea unit and an air unit.
Figure 7. How far a unit can shoot is controlled by the range property.

When a game is played, it’s divided into three phases:

- **Deployment phase** – where each player place its units in the spots available in its two zones. This is done once.

- **Tactics phase** – where each player choose what actions their unitstacks are about to do. When both players have decided what each of their unitstacks is going to do, the combat phase begins.

- **Combat phase** – the results are calculated and presented. All unitstacks perform their actions at the same time. If the game is not finished, another round of tactics phase follows, where remaining units can choose again what to do during next combat phase.

Available actions for each unitstack are:

- **Attack** – Possible against targets within range, if the attacking unit has an attack value against the defending unit. Each unitstack targets another unitstack.

- **Defend** – Increases the units defence values with 2 against any attack. Also decreases the maximum damage it can receive from 2 to 1.

- **Move** – The unit can move to a bordering zone even if it belongs to the enemy, as long as there are unoccupied spots.

There are several possible outcomes of a game:

- All units belonging to a country (or both) are eliminated.
- One country retreats from the battle.
- The attacking country doesn’t have any unit’s left that’s able to attack.

**2.6.2 Example of deployment phase**

The players each have 3 infantry units, 2 artillery units and one tank unit. Figure 8 illustrates different ways to deploy them before combat.
Figure 8. Example of deployment phase. The left player has chosen to place both artillery units in the support zone, which often is preferable since most enemy units can’t reach them there while they still have the capability to attack due to their long range. The infantry and tank units have shorter range so they are placed in the frontline zone. The right player has chosen a slightly different strategy, placing one of the artillery in the frontline zone. This makes the unit getting a better attack value against the left player’s artilleries, but also makes it more vulnerable to attacks.

2.6.3 Example of tactics phase

One way to perform tactics is illustrated by figure 9.

Figure 9. Example of tactics phase. Each unitstack gets to chose if it wants to attack, defend or move. In this case all unitstacks have chosen to attack.

2.6.4 Example of combat phase

In this phase the result of all the attacks are calculated. This occurs by looping through all units and let each perform their attack individually. In this case the left player uses an artillery unit (AR) for attacking an infantry unit (IF). What happens is that the artillery units attack value against ground units (7) and the infantry units defence value against ground units (4) are used to find out the hit percentage. This is done by comparing the values by using Table 4. In this case there is a 65 % hit-probability. If the AR hits the IF, then the health of the IF is reduced with one. There is always the chance for a perfect hit, which is 2/10 of the hit-percentage. In this case it’s 65 %* 0.2 = 13 %. A perfect hit means damage given is two. So in this case, when a random
number 1-100 is used: 1-13 means damage = 2. 14-65 means damage = 1. 66-100 means no damage. Even if the infantry unit is destroyed in the attack, it still gets to perform its attack, since everything happens at the same time.

Table 4. *Hit-probability of attacks in percent.* An attacking unit’s attack value is measured against the defending unit’s defence value. The result is the probability that the attacking unit will hit the defending unit. An attack value of 5 and a defence value of 9 give a 30% hit-probability.

<table>
<thead>
<tr>
<th>DEFENCE VALUE</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 1</td>
<td>50</td>
<td>45</td>
<td>40</td>
<td>35</td>
<td>30</td>
<td>25</td>
<td>20</td>
<td>15</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>T 2</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td>40</td>
<td>35</td>
<td>30</td>
<td>25</td>
<td>20</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>T 3</td>
<td>60</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td>40</td>
<td>35</td>
<td>30</td>
<td>25</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>A 4</td>
<td>65</td>
<td>60</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td>40</td>
<td>35</td>
<td>30</td>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>C 5</td>
<td>70</td>
<td>65</td>
<td>60</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td>40</td>
<td>35</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>K 6</td>
<td>75</td>
<td>70</td>
<td>65</td>
<td>60</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td>40</td>
<td>35</td>
<td>30</td>
</tr>
<tr>
<td>V 7</td>
<td>80</td>
<td>75</td>
<td>70</td>
<td>65</td>
<td>60</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td>40</td>
<td>35</td>
</tr>
<tr>
<td>A 8</td>
<td>85</td>
<td>80</td>
<td>75</td>
<td>70</td>
<td>65</td>
<td>60</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td>40</td>
</tr>
<tr>
<td>C 9</td>
<td>90</td>
<td>85</td>
<td>80</td>
<td>75</td>
<td>70</td>
<td>65</td>
<td>60</td>
<td>55</td>
<td>50</td>
<td>45</td>
</tr>
<tr>
<td>L 10</td>
<td>95</td>
<td>90</td>
<td>85</td>
<td>80</td>
<td>75</td>
<td>70</td>
<td>65</td>
<td>60</td>
<td>55</td>
<td>50</td>
</tr>
</tbody>
</table>

2.7 The role of the game AI

In this game, quite a few possibilities for tactics are open. This put a high demand on the AI. Some of the challenges include:

- Learn how to use the attacks most efficiently. Some units perform extra well against some other units, and very poorly against others. So optimizing the attacks is very important in a battle. Figure 10 illustrates an example of poor tactics and good tactics.
- Learn when it’s better to defend a unit than attacking with it.
- Learn when it’s favourable to move a unit forward or backward.
- Learn how to avoid “wasting” attacks by using too many stacks attacking the same enemy stack, which would be destroyed anyway using fewer attacks.
Figure 10. Smart and stupid tactics. Attacking an anti air unit with a bomber is not likely to succeed. Using a ground unit, like tanks, is much more effective. Also, a bomber is more effective against an artillery unit than a tank unit is.
3 Problem description

The aim of this dissertation is to investigate how AI for a turn-based computer game can coevolve into playing smarter by combining genetic algorithms with neural networks and using a reinforcement learning regime.

Genetic algorithms and neural networks are techniques that have shown great potential for learning, but they still aren’t any major players in modern computer game AI. The reasons for this seems to be that these techniques doesn’t deliver the kind of predictable, understandable, and tuneable control that the developers want over a games’ AI. But also, it’s possible that developers just don’t have enough experience with these techniques to use them successfully.

Most turnbased games use a rule-based approach. Since many unit types and tactics can be used, many rules would have to be scripted. This can be very time-consuming, and the AI will just do what the script-programmers have thought of. This might be acceptable, but also predictable.

In this dissertation, an alternative approach will be investigated. Instead of relying on scripts, neural networks and genetic algorithms will be used to evolve an AI that plays well and unpredictably, without having to write any rules.

One way to let the AI evolve would be to train it against a scripted AI. If the scripted AI is considered to perform well, then arguably, the evolving AI should be able to reach the same level. The downside is that since scripts are static, the AI would probably evolve so it specialized in beating the scripted AI. And when faced with dynamic human opposition it might fare less well, partly due to predictability.

Another way would be to train the AI by playing against humans. In theory, this might sound as the best approach, since how it fares against humans is the only real benchmark. But, since GA and NN are techniques that require many runs in order to let the AI evolve properly, this approach would be very time-consuming, and not very realistic.

This leaves the possibility of using a coevolutional approach, which means letting different populations of possible solutions compete against each other in order to evolve. This approach has several advantages. The AI populations can play each other many times, giving them appropriate time to evolve. New tactics and behaviours can be created in this process, without having someone program them manually.

Two populations will be used. They will start out with random values and will probably play poorly in the beginning compared to a human player. As the AI populations evolves this will probably change. The following techniques can be used to measure whether the evolved AI actually play smarter than before.

- It can compete against a random AI population that hasn’t been evolved. If the evolved AI consistently wins over the random AI, this suggests that the evolution process has actually made the AI play smarter.
- It can compete against a scripted AI population that is considered to play well. If the evolved AI consistently wins over the scripted AI, this suggests that the evolution process has actually made the AI play smarter.

Competing will result in a win or a loss for a specific solution. This makes genetic algorithms an appropriate choice of training regime, since they can be viewed as a kind of reinforcement learning.
4 Method

This dissertation is carried out as a case study. Genetic algorithms are used together with neural nets to control the artificial intelligence for the game. Three different strategies are tested and evaluated. The strategies have different architectures of their neural nets and use different ways of coding and decoding the input and the output for the nets.

Each strategy contains 3 different experiments. In the first experiment two populations will coevolve playing each other. In the second experiment one population will evolve by playing against a random AI. In the third experiment one population will evolve by playing against a scripted AI.

The success of each strategy and population is measured periodically by how it plays against a random AI and a scripted AI. More specifically, how many games the population wins and loses.

The expectations are that populations trained against a random AI or a scripted AI will specialize in beating that type, and fare less well against the other. And that the coevolved population will have better generalization skills and will fare well against both types.
5 Implementation

5.1 How the implementation works

Two populations are created with 50 AI individuals in each. Each individual contains a neural net initialized with random start weights.

A scenario is then generated. This scenario specifies what unit types and how many of each will be used in the upcoming match. This makes sure all individuals always start on an equal playing field.

All units are deployed in the same way using a few simple rules. Basically all artilleries, transport ships and coast artilleries are placed in the support zone, and the rest of the units in the frontline zone.

Each individual in one population then confronts an individual from the other population. This means 50 different matches goes on at the same time. After the maximum turn rates have been reached another scenario is generated to be used for the next match.

Altogether there are six different scenario types:

- Ground
- Sea
- Air
- Ground and Sea
- Ground and Air
- Sea and Air

Ground means that only ground units are involved in the scenario. Ground and air means that both ground and air units are involved. These scenario types are used sequentially and looped. After a match has involved ground units, the next match will always be involving sea units.

The individuals in one of the populations are moved one step so that each individual gets a new opponent every match.

After all six different scenarios have been used, meaning a total of 300 matches have been played (6 scenarios * 50 matches), one generation is finished.

After a generation is finished, fitness points are given to each AI individual like this:

<table>
<thead>
<tr>
<th>Points</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+2</td>
<td>for each enemy unit destroyed</td>
</tr>
<tr>
<td>+1</td>
<td>for each own unit alive after a match</td>
</tr>
<tr>
<td>+4</td>
<td>for each match won</td>
</tr>
</tbody>
</table>

The fitness points are used to calculate a fitness value between 0 and 100 for each individual. To reach fitness 100, the individual have to win all six matches, destroying all enemy units and not lose any of its own. To get fitness 0 the individual have to lose all its units without destroying any enemy units.

The genetic algorithm is used to update the two populations. Thereafter the procedure is repeated with the new populations.
Every fiftieth generation the first population is evaluated. This is done by letting each individual in it battle against a random AI for 6 matches and then against a scripted AI for 6 matches. The results of these matches are presented in the results chapter.

To have something to compare the effectiveness of coevolution approach against, tests without coevolution will also be performed. For this only one population will be developed, but instead of evolving against another equal AI it will train against random AI and scripted AI. The evaluation every fiftieth generation will reveal if the neural nets in this approach overspecialize against the random AI or the scripted AI.

5.2 Description of the random AI
The random AI functions in the following way. First the action to perform is selected by random. Each action has an equal chance of being selected.

33 % chance of attacking
33 % chance of defending
33 % chance of moving

If attacking is selected, a random enemy unitstack within range is selected as target. If moving is selected a random empty spot is selected as move target. In either case, if no target is within range, no action is performed.

5.3 Description of the scripted AI
The scripted AI prioritizes attacking. Each unitstack is compared to each of the enemy’s unitstacks. Calculation determines where attacking is most probable to succeed and that unitstack is chosen as target. If the probability to succeed with an attack is zero, or all enemy units are out of range, an attack is not performed. If there are units out of range, the AI tries to get closer to them by moving to one of the center zones. If no units are out of range, and the probability to succeed with an attack on all units within range is zero, defending is performed.

5.4 Decisions made by the AI
The neural net is used in the Tactics phase each time an AI individual have to decide what action a unitstack should perform. The possible actions that can be chosen for them are the following:

- Attack somewhere
- Move somewhere
- Defend

If move or attack is decided, the AI also has to decide where.

5.5 Configuration
There are many different ways neural nets and genetic algorithms can be used in an application. Since different techniques can give vastly different results, three different ones will be tested in this investigation. The difference between the different techniques consists mostly of how input and output data to the net is constructed and interpreted.
The following configuration has been successful in other contexts and will be used for the genetic algorithm. For an explanation of crossover type, mutation type, scale type, selection method and elitism, read (Mitchell, 1996) and (Buckland, 2002).

<table>
<thead>
<tr>
<th>Crossover type</th>
<th>Position based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation type</td>
<td>Insertion</td>
</tr>
<tr>
<td>Scale type</td>
<td>None</td>
</tr>
<tr>
<td>Selection method</td>
<td>Tournament</td>
</tr>
<tr>
<td>Elitism</td>
<td>On</td>
</tr>
</tbody>
</table>

The following parameters will be identical within the three configurations:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crossover rate</td>
<td>0.7</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>0.1</td>
</tr>
<tr>
<td>Population size</td>
<td>50</td>
</tr>
<tr>
<td>Units per team</td>
<td>8</td>
</tr>
<tr>
<td>Turns per match</td>
<td>20</td>
</tr>
<tr>
<td>Matches per generation</td>
<td>6</td>
</tr>
<tr>
<td>Elite individuals</td>
<td>4</td>
</tr>
</tbody>
</table>

The following architectures will be used in the different strategies:

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Input nodes</th>
<th>Hidden nodes</th>
<th>Output nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategy 1</td>
<td>16</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>Strategy 2</td>
<td>1</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Strategy 3</td>
<td>11</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

5.6 Strategy 1

Architecture of the neural net

The neural net used in this strategy consists of 16 input neurons, 10 hidden neurons in one hidden layer and 16 output neurons.

How the input values are determined

Each of the 16 input values is deduced from the 16 location spots on the playing board. The default value is 0. The spot where the active unitstack is located gets a value of -1. For the spots that are occupied by an enemy unit the following formula is used to get an input value:

The attack value of the active unitstack is compared to the defence value of the enemystack using Table 4. This gives a value between 0 and 1, where a higher value represents a higher probability to succeed in a possible attack.

How an action is determined from the output values

16 output values are returned from the neural net. Just one value between 1 and 16 is needed to determine the action of the unitstack. So the number of the spot with the highest output value is used. Then, to identify which action should be used, the following conditions are tested for:

If an enemystack inhabits the spot and it is within attacking range of the active unitstack, then an attack is decided.
If the spot is empty and it is within moving range for the active unitstack, then a move action is decided.
Otherwise a defence action is decided.

5.7 Strategy 2

Architecture of the neural net

The neural net used in this strategy consists of 1 input neuron, 8 hidden neurons in one hidden layer and 3 output neurons.

How the input values are determined

The active unitstack is compared to each of the enemystacks within range to determine where most damage can be inflicted. This is calculated the following way:

First, the attack value of the active unitstack is compared to the defence value of the enemystack, using Table 4. The resulting value is multiplied with 1.2 to weigh in the possibility of a perfect hit (meaning double damage). Then it’s multiplied with the number of units in the active unitstack. This is repeated for each enemystack.

When the highest expected damage to inflict is extracted, it is used as input.

For example, a scenario with 2 tank units against a stack of infantry units gives the following:

Attack value 7 against defence value 4 results in a hit probability of 0.65 using Table 4. This value is multiplied with 1.2, which gives 0.78. This value is then multiplied with the number of attackers (2) which results in an expected damage to inflict of 1.56. If 1.56 is the highest expected damage to inflict this value is used as input.

How an action is determined from the output values

3 output values are returned from the neural net. The one with the highest value determines which action is to be performed. 1 = ATTACK. 2 = MOVE. 3 = DEFEND.

If an attack is decided, the unitstack where the highest damage is expected to be inflicted is attacked. This unitstack was identified during the input process. If a move is decided, the following rules are used. If the unitstack is in a support zone, it moves to the nearest frontline zone. If it’s already in the frontline zone it moves to the other frontline zone.

This is a high-level strategy very similar to script, where little has to be figured out by the neural net.
5.8 Strategy 3

Architecture of the neural net
The neural net used in this strategy consists of 11 input neurons, 10 hidden neurons in one hidden layer and 11 output neurons.

How the input values are determined
11 input nodes are used. Each node symbolizes one unittype. The node representing the active unittype receives a value of 2. The rest of the nodes get a value of 0.

How an action is determined from the output values
Each of the 11 output nodes also symbolizes one unittype. In this case which unittype to attack. The one chosen is the one of the existing unit types with the highest value. If this target is out of range, the active unitstack instead tries to move one step closer to it. If this is not possible due to all spots in the zone being occupied, defence is decided.

The point with this strategy is to let the neural net figure out which unit types are effective against which others.
6 Results

6.1 Strategy 1

6.1.1 Experiment 1: Population evolved through coevolution

In this experiment two populations have coevolved. Every fiftieth generation one of the populations have been measured against a random AI and a scripted AI.

![Figure 11. Neural net measured against a random AI](image)

![Figure 12. Neural net measured against a scripted AI](image)

The results indicate that the coevolved population after 2000 generations play equal to or slightly better than a random AI. But against the scripted AI no performance improvement is shown. The scripted AI consistently wins.
6.1.2 Experiment 2: One population evolved against a random AI

In this experiment there is no coevolution. Only one population is used which have been trained against a random AI. Every fiftieth generation the results are measured as before.

![Figure 13. Neural net measured against a random AI](image1)

The results show that the neural net quickly learns how to beat the random AI, and then continues to improve. The scripted AI consistently wins, but some improvement is noted here.

![Figure 14. Neural net measured against a scripted AI](image2)
6.1.3 Experiment 3: One population evolved against a scripted AI

In this experiment there is no coevolution. Only one population is used which have been trained against a scripted AI. Every fiftieth generation the results are measured as before.

![Figure 15. Neural net measured against a random AI](image1)

![Figure 16. Neural net measured against a scripted AI](image2)

The results show that the neural net takes a while to reach the same performance as the random AI. But it continues to improve until it becomes slightly better. It also takes a while to play effectively against the scripted AI, but after about 1000 generations it has reached close to an equal performance.
6.2 Strategy 2

6.2.1 Experiment 4: Population evolved through coevolution

In this experiment two populations have coevolved. Every fiftieth generation one of the populations have been measured against a random AI and a scripted AI.

![Figure 17. Neural net measured against a random AI](image1)

![Figure 18. Neural net measured against a scripted AI](image2)

The results show that the neural net very quickly learns how to beat the random AI, and soon reaches a performance top. The neural net also quickly learns how to reach an equal, and eventually slightly better level, compared to the scripted AI.
6.2.2 Experiment 5: One population evolved against a random AI

In this experiment there is no coevolution. Only one population is used which have been trained against a random AI. Every fiftieth generation the results are measured as before.

Figure 19. Neural net measured against a random AI

The results show that the neural net very quickly learns how to beat the random AI, and soon reaches a performance top. The neural net also quickly learns how to reach an equal level compared to the scripted AI.

Figure 20. Neural net measured against a scripted AI
6.2.3 Experiment 6: One population evolved against a scripted AI

In this experiment there is no coevolution. Only one population is used which have been trained against a scripted AI. Every fiftieth generation the results are measured as before.

Figure 21. Neural net measured against a random AI

The results show that the neural net very quickly learns how to beat the random AI, and soon reaches a performance top. The neural net also quickly learns how to reach an equal level compared to the scripted AI.
6.3 Strategy 3
6.3.1 Experiment 7: Population evolved through coevolution

In this experiment two populations have coevolved. Every fiftieth generation one of the populations have been measured against a random AI and a scripted AI.

Figure 23. Neural net measured against a random AI

Figure 24. Neural net measured against a scripted AI

The results show that the neural net immediately dominates the random AI. Against the scripted AI it improves for about 800 generations, where it’s consistently much better than the scripted AI.
6.3.2 Experiment 8: One population evolved against a random AI

In this experiment there is no coevolution. Only one population is used which have been trained against a random AI. Every fiftieth generation the results are measured as before.

![Figure 25. Neural net measured against a random AI](image)

The results show that the neural net immediately dominates the random AI. Against the scripted it takes longer to win, and after that the scripted AI several times manage to win more matches. This indicates that training against the random AI doesn’t give as high performance as training with coevolution like the previous experiment.

![Figure 26. Neural net measured against a scripted AI](image)
6.3.3 Experiment 9: One population evolved against a scripted AI

In this experiment there is no coevolution. Only one population is used which have been trained against a scripted AI. Every fiftieth generation the results are measured as before.

![Figure 27. Neural net measured against a random AI](image)

The results show that the neural net immediately dominates the random AI. Also, it very quickly becomes better than the scripted AI, and manage to beat it consistently with at big marginal.

![Figure 28. Neural net measured against a scripted AI](image)
6.4 Interpreting the results

6.4.1 Strategy 1

When this strategy is used the performance continues to evolve for a long time. This is clearly illustrated in figure 13 where the AI continues to improve during 2000 generations.

In experiment 1 where this strategy use a coevolutional approach it takes about 1600 generations before it plays at the same level as the random AI. Against the scripted AI it starts out by improving the performance. But after about 600 generations it quickly starts to degenerate and then performs consistently very poorly. This could be an indication that this strategy gives a fairly weak AI. Or it could possibly be a case of the population being overfitted at playing the other population. As described by Björn Olsson (2001), when two opposing populations are coevolving, we expect that each will become increasingly efficient at exploiting the weaknesses of the other. While this generally is a good thing, it shouldn’t be taken too far so a given AI just plays well against another given AI. In this experiment that might have been the case.

Although experiment 2 where the AI has been trained against the random AI, it also becomes very good at beating it. That it has been overspecialized against it is indicated by the poor results against the scripted AI. It’s notable though that it plays better against the scripted AI than what the coevolved population does.

In the third experiment the AI has been trained against a scripted AI. After 1000 generations it plays about as good as it. It also beats the random AI, but not consistenly or with a big margin.

Experiment 2 and 3 together indicates that an AI trained against a random AI or scripted AI does overspecialize against them. Experiment 1 show that the coevolutional approach was not particularly successful in combination with this strategy.

6.4.2 Strategy 2

With this strategy a much higher performance level was reached during all three experiments. The AI very quickly evolved into totally dominating the random AI in all three cases. This makes it a bit difficult to compare the generalization capabilities between them.

In experiment 4 the coevolved AI reaches a performance level slightly higher than the scripted AI. Experiment 5 shows that the AI trained against the random AI just reaches the level of the scripted AI. This means that the coevolved AI fares better against the scripted AI than this population does. In experiment 6, a bit unexpected, the AI trained against the scripted AI doesn’t perform much better against the scripted AI than what the population trained against the random AI does.

6.4.3 Strategy 3

In this strategy, all 3 populations beats the random AI totally and consistenly. The difference from strategy 2 is that these do it right from the start.

The coevolved AI in experiment 7 quickly learns how to beat the scripted AI. The population trained against a random AI in experiment 8 takes longer before being able to beat the scripted AI, and it does it inconsistently. The population trained against a scripted AI in experiment 9 quickly learns how to defeat the scripted AI and does it slightly better than the coevolved population.
7 Conclusion

The results show that coevolution can be useful for developing a smart AI. But the results are dependent on how the neural nets are coded and decoded. With strategy 1 the coevolved population fared less well than the ones specialized against random AI or script AI. But this might be a weakness of the strategy instead than of the coevolution.

Strategy 2 was the simplest strategy to be evolved. It basically just had to decide whether to attack, move or defend depending on a single value of how much damage it might be able to inflict. The neural net very quickly learns that attacking is best most of the time and from there on performs very well. Although with this strategy the coevolved population performed slightly better than the other ones.

In strategy 3 attacking is the primary action used. The effectiveness of this is shown by the immediately good results against the random AI. Here the gradual improvement is done when the AI learns to match which unit types are effective against each other.

But the immediately good performance just by having an attack as the primary action indicates that the domain used might not have been a balanced one. Just by attacking all the time a high performance is reached. This is not optimal. The domain would have been better suited for the task if moving and defending had been more useful than they are compared to attacking. The are specific situations where moving or defending are more useful than attacking, but they are too few compared to attacking.

The population trained against the random AI in experiment 8 with strategy 3 shows clearly how it’s been overspecialized playing a random AI, by getting quite a poor result against the scripted AI, when it’s compared to the results of the coevolved population.

This experiments show that a coevolved population can reach a high performance in this kind of turnbased strategy games. It also shows that how you code and decode the neural net and which strategy you use plays a very big role in the final results. Although there is no crystal clear result illustrating that the coevolved populations are better at generalisation, some of the results do indicate that is the case.

7.1 Discussion

The domain might have been better suited for the task by being more balanced between the different actions. Attacking is more useful much more frequently than moving or defending. The neural net probably learns this very quickly and then plays very well just by having identified that simple rule. Although, improvements are still made after this, as is illustrated by experiments 7 to 9.

The neural nets were not used to deploy the units on the playing board. This could probably have been achieved successfully, but it might have required a separate neural net.
References

Black & White (2001) [Computer game], Lionhead Studios, EA Games.
Colin McRae Rally 2 (2001) [Computer game], Codemasters.
Creatures (1996) [Computer game], Cyberlife Technology, Mindscape.
Frontline: Europe (2005) [Computer game].
Half-Life (1998) [Computer game], Valve, Sierra On-Line.
The Sims (2000) [Computer game], Maxis, Electronic Arts.