Table of Contents

[1 Abstract 3](#_Toc419835959)

[2 Introduction 3](#_Toc419835960)

[3 Related Work 4](#_Toc419835961)

[4 Method 5](#_Toc419835962)

[4.1 Artificial Neural Network 5](#_Toc419835963)

[4.1.1 Feed-forward neural network 5](#_Toc419835964)

[4.2 Genetic Algorithm 6](#_Toc419835965)

[4.2.1 Evolution 6](#_Toc419835966)

[4.2.2 Mutate 7](#_Toc419835967)

[4.2.3 Crossover 7](#_Toc419835968)

[4.3 Training 8](#_Toc419835969)

[4.3.1 Fitness function 8](#_Toc419835970)

[4.3.2 Two-loop method 9](#_Toc419835971)

[5 Framework 11](#_Toc419835972)

[5.1 GVG-AI 11](#_Toc419835973)

[5.2 Available information 11](#_Toc419835974)

[5.3 Games 11](#_Toc419835975)

[5.3.1 Butterflies 12](#_Toc419835976)

[5.3.2 Chase 12](#_Toc419835977)

[5.3.3 Survive Zombies 12](#_Toc419835978)

[6 Experiments 13](#_Toc419835979)

[6.1 Inputs 13](#_Toc419835980)

[6.2 Experiment parameters 13](#_Toc419835981)

[6.3 Deep graphs 14](#_Toc419835982)

[6.4 Cross over experiments 14](#_Toc419835983)

[6.5 Two similar games experiments 17](#_Toc419835984)

[6.6 Two different games experiments 18](#_Toc419835985)

[7 Discussion 20](#_Toc419835986)

[7.1 Performance 20](#_Toc419835987)

[7.2 Deep vs non deep 21](#_Toc419835988)

[7.3 Playing two games 21](#_Toc419835989)

[7.4 Crossover 22](#_Toc419835990)

[7.5 Limitations 23](#_Toc419835991)

[8 Conclusion 23](#_Toc419835992)

[9 Future Work 24](#_Toc419835993)

[9.1 NEAT 24](#_Toc419835994)

[9.1.1 Topology 24](#_Toc419835995)

[9.1.2 Crossover 24](#_Toc419835996)

[9.1.3 Speciation 24](#_Toc419835997)

[9.2 Fitness Function 25](#_Toc419835998)

[10 Bibliography 26](#_Toc419835999)

[11 Appendix 27](#_Toc419836000)

[11.1 One game deep and non-deep game score 27](#_Toc419836001)

[11.2 One game compression training 28](#_Toc419836002)

[11.3 Two Similar games compression training 29](#_Toc419836003)

[11.4 Two different games compression training 29](#_Toc419836004)

# Abstract

The idea of having an AI capable of playing many different games with limited domain knowledge is an appealing one. For example, it could allow an AI to adopt on the fly without manual intervention, or it could allow game designers to avoid having to hard code behavior for a given state.

This project has focused on trying to see if it was possible to use neuro-evolution to evolve a controller capable of completing multiple games and to attempt to assess the effects of using deep neural network techniques on this particular problem.

A multilayer-perceptron was combined with a genetic algorithm and evolved by playing games from the General Video Game AI Competition (GVG-AI) framework. Furthermore, input was collected from playing the GVG-AI games and used to train a layer, the input representation layer, designed to compress the input down into a feature vector. Training was done in iterations, where first the input representation layer was trained, and then that layer was used to play the GVG-AI games where the rest of the neural network was trained. The input from that game was then used to train the next input representation layer, etc.

Training the input representation layer in loops turned out to be detrimental. The performance of a multilayer-perceptron where the entire network was evolved through playing the games turned out to perform better in our implementation. However, this was likely because of the way it was implemented, and not because the concept itself of training the input representation separately is flawed.

While the neural networks evolved performed better than a neural network with randomized values, they did not manage to win any games in the timeframe they were given, other than games that had a simple solution. It is possible that allowing the experiments to run longer will produce better results.

# Introduction

Ever since the invention of the computer, humans have been attempting to replicate human-like intelligence. In the 1950s, Alan Turing introduced a test that could determine whether a machine possessed artificial intelligence. In order to pass the Turing Test a computer must be able to convince a human that he or she is in fact talking to another human. In order to do this, a number of different capabilities are needed. These capabilities are:

* Natural language processing – to enable it to communicate successfully in English.
* Knowledge representation – to store what it knows or hears.
* Automated reasoning – to use the stored information to answer questions and draw conclusions
* Machine learning – to adapt to new circumstances and to detect and extrapolate patterns.

One possible way of tackling these problems is to mimic the way the biological brain works. This concept, called artificial neural networks, goes back to the mid-1900s and has been shown to be highly useful in tackling problems that contains noisy data, such as handwriting recognition.

Another method rooted in biology is genetic algorithms. The basic idea mimics the evolutionary changes species go through as they reproduce in order to adapt to their environment. This method essentially allows programs to adapt to their environment as long as they have an indication of how they are doing.

Neuro-evolution takes these two concepts and combines them, so you get an artificial neural network that has been trained using genetic algorithms. The strength of this method is that you only need to provide it with the environment and a fitness functions for determining how well it is doing, and it will gradually train itself to behave appropriately in the environment.

One area where this is of interest is the area of AI for computer games. Traditionally these AIs have handcrafted behaviors. This is ideal in the sense that it allows for designers to control how AIs behave in specific situations, however, it is unfeasible for some types of games for designers to design every possible behavior in a state. One possible solution to this problem would be to evolve an AI through neuro-evolution, and going one-step further, it might be possible to evolve an AI that could play many different types of games.

This project aims to explore the possibilities regarding using neuro-evolution to create a general video game AI that can be used for different types of games. Furthermore, concepts from deep neural networks, such as compressing large input down into a small feature vector, will be explored to see if they provide benefits to the process.

# Related Work

The topic of creating a general AI capable of playing a game without domain specific knowledge has shown increasingly greater interest in the AI community. (Levine, et al.) This interest has ended up in a competition called the General Video Game AI (GVG-AI) Competition based around the concept of having agents play many different types of games created in the VGDL language. (Perez, et al., 2014) The framework provided by the GVG-AI competition is therefore ideal for attempting to create a general AI and was chosen for this project.

Several solutions to the problem of general video game AI has been proposed already. A popular choice is to use an approach based Monte-Carlo Tree Search (MCTS); however, MCTS have issues with being limited by the depth of calculations. It is possible to end up in situations where it would be obvious for a human what to do, but the high branching factor means it is impossible for MCTS to determine without significant computational time. (Ross, 2014)

Another approach is to use neuro evolution, which is a combination of an artificial neural network and a genetic algorithm. Using neuro evolution to attempt to create an agent capable of playing games is not a new concept. It has been used before to learn how to play Atari games. (Hausknecht, Lehman, Miikkulainen, & Stone, 2014) HyperNEAT, a technique that uses indirect encoding, was used to tackle this problem and it was possible for it to learn to play using nothing but the raw visual input of the game. With an object representation as input however, NEAT, which uses direct encoding, outperformed HyperNEAT. These results were from evolving a distinct network for each individual game, however.

In previous work, in an attempt to drive a car in a racing game by visual input alone, a Max-Pooling Convolutional Neural Network (MPCNN) has been used to reduce the high dimensional input space, the visual input in the form of images, into a low dimensional feature vector that is fed into a recurrent neural network. (Koutník, Schmidhuber, & Gomez, 2014) This allows an evolutionary algorithm to work in a low dimensional feature space. In this project, the same tactic is used of compressing the high dimensional input down. This is done in the same way by having a fitness based on maximizing the distance between the output vectors of the compression layer. However, instead of an MPCNN and a recurrent neural network, the implementation uses a multi-layer perceptron for both the compression and the behavior part.

# Method

## Feed-forward neural network

Figure 4‑1 - Feed Forward Artificial Neural Network

The implementation uses a feed-forward artificial neural network (FFANN). A FFANN is separated into three types of layers: Input layer, hidden layers and output layer. An example of a FFANN with one hidden layer can be seen in Figure 4‑1.

Each layer has a number of nodes and each of these nodes has weights leading to each of the nodes in the next layer. The only exception is the output layer that only holds the values they receive from the last hidden layer. The output node with the largest value determines the result.

When the neural network is activated, it receives an input vector with the size of the input layer, providing each input node with a value. These values are passed on to the nodes of the first hidden layer following the simple formula: with *vi* being the value of node i and *wij* being the weight from node i to node j. The nodes of the hidden layers works similarly, but instead of just passing on the input value they have an activation function, with ai being the threshold of the node, a randomly assigned value that each node in the hidden layers has. The activation function can be any function, but it is common to use a sigmoid function (for example ) to normalize the output values of the neuron as it outputs between 0 and 1. The different hidden layers may very well have different activation functions.

## Genetic Algorithm

A genetic algorithm uses the principle of natural selection to generate a solution for a given problem. It holds a population of solutions that can all be described as for instance a sequence of numbers referred to as the solution’s chromosome. After having evaluated all solutions and assigned them a fitness value based on how good they performed, the evolution of new solution begins. A new solution is made by pairing two existing solutions. This can be done by using a crossover where a random point in the chromosome is selected; the one half of the one solution is then combined with the other half of the other solution (this is the way we do it). Afterwards the new solution is mutated by selecting one or more genes in the chromosome and change it to a new random value. This process is repeated until a new population of solutions is generated. Some algorithms save some of the best solutions for the new population as well as you are not guaranteed that the new population contains any improvements compare to the old one.

## Neuro-evolution

In our implementation of neuro-evolution, the new population is generated the following way.

1. Sort the neural networks in descending order according to their fitness
2. Keep the first third of the top scorers and put them into the new population
3. Each neural network is given a probability of being selected for pairing based on its fitness. The top scorer has 50% chance of being paired with, the second highest has 25% chance, etc. following the formula with i being the neural network’s rank. The last neural network’s chance of being selected is with n being the size of the population.
4. The remaining individuals of the new population are generated by iterating through the population of old neural networks in descending order according to their fitness. For each neural network:
   1. Crossover with another neural network selected based on the probabilities created in step 3. If crossover is disabled, then copy the neural network.
   2. Mutate the newly created neural network from a.
   3. Put into new population
   4. Repeat until population size has been reached

### Mutate

In our implementation, mutation works by going through every weight and threshold in the neural network, calculating a random value, and if that value is lower than the mutation chance then apply the following function:

Where *v* is the value of a weight or threshold, *r* is a random number between -0.5 and 0.5, and *m* is the mutation rate

### Crossover

The final implementation of the crossover function works by picking a layer at random. The chosen layer is the splitting point, all weights and neurons before this layer will be chosen from the first neural network, and all weights and neurons on the layer and after will be chosen from the second neural network.

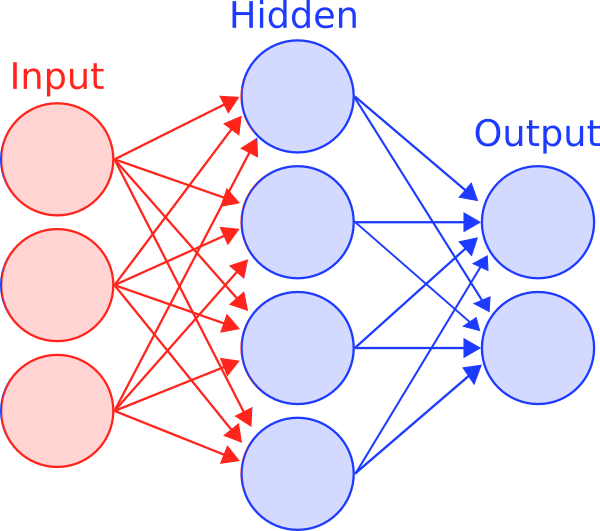


Figure 4‑4 - New neural network after crossover

Figure 4‑2 – Neural Network #1

Figure 4‑3 – Neural Network #2

An example of this can be seen in Figure 4‑2 to Figure 4‑4 where the   
splitting point is the middle layer.

Note that if the splitting point chosen is the input or output layer, nothing will happen.

### Fitness function

When evaluating how well a neural network did when completing a game, there must be a way to determine the fitness of a neural network in an environment. Furthermore, the more gradient the function can be, the better, as this allows for any little improvement in a neural networks performance to be selected during the selection process.

First, winning a game is the end goal; therefore winning the game should be rewarded. In the implementation, it is rewarded 1000 fitness points for winning. However, since it is binary result whether you either win or lose, it is insufficient.

Each game also contains a game score, and is, per definition, related to how well one does in the game. As such, for every game score point, 10 fitness points are given. The details of what you receive a score for in each game can be found in section 5.3.

While it is possible to create very specific fitness functions related to the games, for example distance to enemies, it would also mean that a handcrafted fitness function would need to be made for every game. The goal in this project has been to create one implementation that could learn to play several games at once; therefore, the fitness function must be generic enough to work with every game.

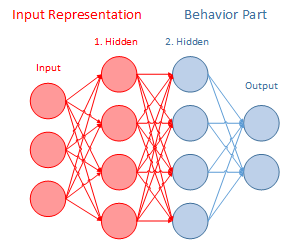
#### Encouraging movement

When trying to determine which behavior will lead to the best performance, it is advantageous to explore the environment. The way this is done is by giving 1 fitness point for every new tile explored. This encourages the agent to explore the environment, which makes it more likely to traverse more parts of the map and give it more possibilities to move into advantageous areas.

#### Penalizing death

When the game ends, every remaining game tick left of the maximum amount of allowed game ticks is subtracted from fitness. So if the agent dies at game tick 400, and the maximum number of game ticks allowed was 500, 100 would be subtracted from the fitness score.

### Two-loop method

We now introduce the two-loop method for splitting up the neural network into two parts, one trained for responsiveness, the input representation part, and another for winning the game, the behavior part.

The purpose of the input representation part is the reduce a large input with similar values into a smaller feature vector, this means that when evolving the behavior part during the game, the amount of neurons that evolution have to work on is reduced, increasing the likelihood of evolution inducing behavior.

Figure 4-5 – Neural network divided into Input Representation and Behavior Part

The input representation consists of the input layer and the first hidden layer, and the behavior part consists of the rest of the hidden layers and the output layer. Figure 4-5 shows how a neural network with two hidden layers is divided into Input Representation and Behavior Part.

The behavior part is trained the same way as described in section 4.3. After fitness evaluation, the neural network is split up in the two parts, and based on the fitness it gained from the evaluation it is exposed to mutation and crossover as per usual. The input representation is left untouched during this process, and afterwards the resulting behavior part is reunited with its input representation to form a complete neural network again. All behavior parts always share the same input representation part.

The input representation is trained in a different way. It is trained by having it process input vectors stored from recent executions. The fitness is then evaluated by taking calculating the distances of all the vectors to each other, taking the minimum distance and adding the average distance.

This way is it possible to assess how sensitive the input representation is to small changes in the input vectors. Moreover, because the input vectors are from actual executions, you know that they are likely to occur and therefore important the neural network are able to distinguish between them. Besides the fitness function, the algorithm used to train the input representation is the identical to that described in Section 4.3. The behavior part is not used for this part and therefore not evolved during training.

After having trained the input representation the behavior part is trained again. This time it is supposed to be more sensitive to the input it meets, making it improve faster. Therefore, it will ideally reach better results than first iteration, which will in return change the received input. The new input vectors are then stored for the next training of the input representation.

# Framework

The General Video Game AI Competition (GVG-AI) is a contest exploring the problem of how to create a controller that is able to play different types of games successfully. The GVG-AI competition takes place every year, and players from around the world creates controllers to compete against each other.

A java framework is available to the participants. This framework contains functionality to play games with humans as well as programmed controllers. It contains a number of example controllers and example games for you to test your own controller on.

Participants are supposed to build a non-human controller that is able to choose what actions to execute when playing the games. The games run in time ticks. For each game tick the controller chooses and action to be executed.

## Controllers

When you create your own controller, it extends a class that has a method that is called once every game step. This method has to returns an action that is executed in the game. The method takes the game state as a parameter that provides the controller with information about the state of the game.

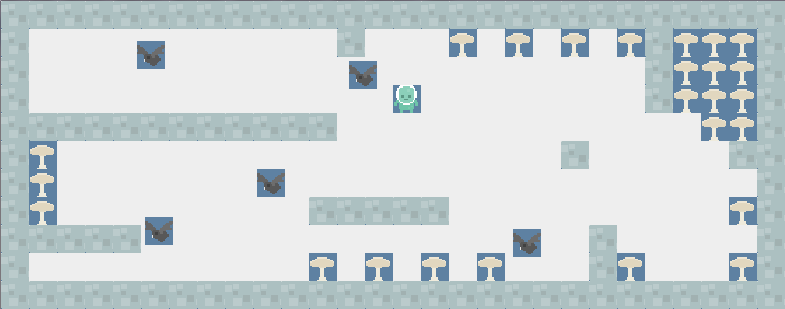
This information includes simple data about the game such as number of executed game ticks, whether the game is over, who won, score, the size of the game and the legal actions for this particular game. It has information about the avatar (the character controlled by the player), its position, speed and which direction it is facing. Finally, it has information about the game board that is accessible as a 2-dimensional array with each entry being a list of the entities on that position in the grid. All the entities are sorted into categories, such as immovable, NPCs, resources etc. For each of these categories the game state can provide a list of all entities in the game. The details about how the information is used is described in section 6.1.

The controllers has to output one of the defined actions. These include movement in four directions (up, down, right and left), special action (that depends on the game) and do nothing. Not all games allow all of these actions, that will depend on the rules of the particular game. The last action, the framework supports, is ‘Escape’, which exits the game.

## Games

The GVG framework contains 20 different games that vary in objectives, play rules and way of rewarding. We selected three games for our experiments that has a somewhat similar ruleset, but different objectives: Butterflies, Chase and Survive Zombies. In these games, the player can move in all four directions, but does not have a special action.

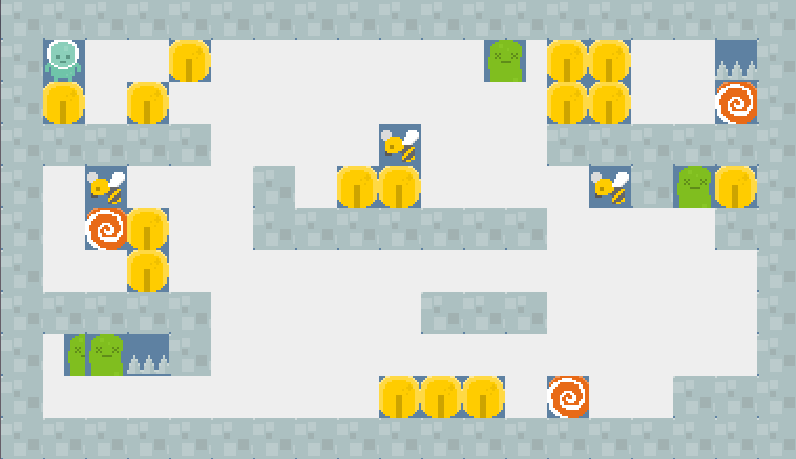
### Butterflies

In Butterflies, the player has to catch all butterflies before they eat all the mushrooms. When a butterfly reaches a mushroom, it eats it and multiplies. The game ends when all the butterflies are caught or all the mushrooms are eaten. If the player catches the butterflies it wins, else it loses. The butterflies move randomly. The player scores points for each butterfly it catches.

### Chase

Chase is very similar to Butterflies. Instead of chasing butterflies, the player chases spiders. The spiders do not move randomly but rather away from the player. The only way for the player to lose is if it runs out of time and still has not caught them all.

### Survive Zombies

Survive Zombies is different from the other two games. There are two different forms of NPCs in this game, zombies and bees. The zombies move towards the player and if they hit it, they will take a small amount of health. The bees move around randomly and do not cause the player any harm, but if they hit a zombie both die. On the map there are spawn points for the bees and the zombies, who spawn at a random rate. There are coins around the map, they give the player extra health if it picks them up. When a bee hits a zombie, they both die and leave a coin behind. The goal of the game is to survive for 1000 game ticks.

# Experiments

In order to test the implementation, a variety of experiments have been set up and run. Since the goal is to create an agent capable of playing several games at once, there will be experiments with an agent trained on just one game (chase), on two similar games (chase and butterflies), and on two games with different goals (chase and survive zombies). These games will all be tested with neural networks trained with the two-loop method, and neural networks who do not use the compression loop. Furthermore, a test will be done to see which effect the crossover has on the development.

## Inputs

The input for the neural networks is a conversion of the game grid into a sequence of numbers. The input consists of five copies of the game grid each providing information about one particular type of item. For each position on the grid, if the item in question is present then the associated input node is activated. The first grid contains all the positions that the controller previously has been on. The second grid only holds the current position of the controller. The third grid holds the position of all “resources”, which is the group of non-moving items that interacts with the controller if it walks into them. Ideally, the input would split the different types of resources across as many grids as there were types. This would require us to know on beforehand how many types there were in the game with the most types. The fourth grid contains the positions of all NPCs, which is non-projectile, moving entities. This holds the same problem as the resources as some games have multiple types of NPCs, and they may have different roles. The fifth grid contains the positions of all static objects. Lastly, we give it the elapsed game ticks divided with 10 and the game ID so it has a potential way of distinguishing between the different games.

## Experiment parameters

The parameters for the experiments are:

* **Population size** is the amount of neural networks that the genetic algorithm evaluates for each generation.
* **Neural network size** shows how many nodes there are in each of the layers in the neural networks.
* **Mutation chance** is the chance during evolution for each threshold and each weight in the neural network to have added the mutation rate times a random number between -0.5 and 0.5.
* **Generations** are the amount of evolutions the genetic algorithm performs before switching between input representation and behavior part.
* **Rounds** are the amount of cycles of evolving both the input representation and the behavior part.
* **Game replays** is the number of times that each game is played per neural network per generation. Running the games several times evens out the impacts of random behavior and provides a more evened out impression of the neural networks ability to play that particularly game.

## Deep graphs

The deep graphs shown in section 6.4 to 6.6 have flat sections, these are the generations spent on training the input representation and can be seen in Appendix 11.1-11.4.

## Cross over experiments

The cross over experiments were done in one game: Chase.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Results** | Random (control) | Deep cross over | Deep non cross over | Non deep cross over | Non deep non cross over |
| Fitness (after 300 generations) | 5.05 | 81.0 | 69.0 | 94.0 | 86.0 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Population size | Neural network size | Mutation rate | Mutation chance | Number of generations per round | Number of rounds | Game replays | Cross-over |
| 40 | {2242,10,10,4} | 0.4 | 5 % | 400 | N/A | 3 | No |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Population size | Neural network size | Mutation rate | Mutation chance | Number of generations per round | Number of rounds | Game replays | Cross-over |
| 40 | {2242,10,10,4} | 0.4 | 5 % | 400 | N/A | 3 | Yes |

Both with and without cross over the non-deep controller is steadily improving. However it is climbing a little faster with cross over. The minimum fitness remains just above 0, which shows that getting score in Chase is hard, as just a game score of 1 would be awarded with 10 fitness points. The points are most likely exclusively movement rewards.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Population size | Neural network size | Mutation rate | Mutation chance | Number of generations per round | Number of rounds | Game replays | Cross-over |
| 40 | {2242,10,10,4} | 0.4 | 5 % | 50 | 6 | 3 | No |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Population size | Neural network size | Mutation rate | Mutation chance | Number of generations per round | Number of rounds | Game replays | Cross-over |
| 40 | {2242,10,10,4} | 0.4 | 5 % | 50 | 6 | 3 | Yes |

The deep controllers are resetting between each round having to start all over more or less. Those without cross over struggles to improve in general, where as those with display a large growth compared to all other executions. These are the only neural networks that manage to get a fitness value above 110. Their minimum values are higher than the others’ as well, indicating that some of them may even have managed to get a game score of 1 or 2.

## Two similar games experiments

These experiments were done on two games that share similar beneficial behavior and goals: Chase and Butterflies.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Random (control) | Deep | Non deep |
| Fitness (after 300 generations) | 50,93333 | 207 | 374,166667 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Population size | Neural network size | Mutation rate | Mutation chance | Number of generations per round | Number of rounds | Game replays | Cross-over |
| 40 | {2242,10,10,4} | 0.4 | 5 % | 400 | N/A | 3 | Yes |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Population size | Neural network size | Mutation rate | Mutation chance | Number of generations per round | Number of rounds | Game replays | Cross-over |
| 40 | {2242,10,10,4} | 0.4 | 5 % | 50 | 6 | 3 | Yes |

The large fluctuation is a result of the random movements of the butterflies making the resulting score vary greatly from game to game. The deep controller does not manage to improve between rounds, but achieves both rising and dropping results. The non-deep controller on the other hand seems to be slowly climbing even though the fluctuations make it hard make a definitive call.

## Two different games experiments

These experiments were designed to test the two different approaches’ ability to learn to play games with significantly different goals. The two selected games were Survive Zombie and Chase. The execution ran 1000 time steps.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Random (control) | Deep | Non deep |
| Fitness (after 200 generations) | 721.075 | 1246 | 1584.166667 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Population size | Neural network size | Mutation rate | Mutation chance | Number of generations per round | Number of rounds | Game replays | Cross-over |
| 40 | {1541,10,10,4} | 0.25 | 10 % | 50 | 4 | 3 | Yes |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Population size | Neural network size | Mutation rate | Mutation chance | Number of generations per round | Number of rounds | Game replays | Cross-over |
| 40 | {1541,10,10,4} | 0.25 | 10 % | 50 | 4 | 3 | Yes |

The resulting fitness values span from 530 to 1621. The fitness values from the Chase would always be 1000 + what it earned from the execution as it cannot die and thus loose points for ending the game prematurely. The fluctuations therefore come from Survive Zombies as it may die almost immediately if it runs towards the zombies or actually win the game. If the controller survives 1000 time steps, it is rewarded with another 1000 fitness points for winning. This makes the resulting values vary from 1000 to 3000 + points awarded for behavior and score divided by 2 (the number of games).

The non-deep controller does find a stable way of winning Survive Zombies after about 40 generations at which point it plateaus. It continues to experiment creating solutions that die early in Survive Zombies, but the continuously rising average score indicates that it produces increasingly many winning neural networks. It plateaus as well after about 130 generations.

The deep controller never manages to create a stable host of winning neural networks. One or a few of them does manage to win, but somehow they fail at subsequent attempts. It does not appear to improve at all during the four rounds of evolution.

# Discussion

## Performance

At a very early stage, it became apparent that the neural network solution was significantly less suited for certain games than for others. A game like Chase requires that the controller move very specifically according to the position of nearby spiders and obstacles. For instance, the controller needs to be able to recognize if the avatar is next to a spider and then move onto that spider regardless of the state of the rest of the game map. An alternative option in terms of winning Chase would be to learn a path that would hit all of the spiders. This is only possible as the spiders’ movements are somewhat predictable.

The controller has even more trouble winning Butterflies because it cannot even just conjure a path as it could for Chase. In Butterflies, the controller is rewarded for hitting the butterflies who moves randomly so their positions cannot be predicted. Furthermore, the random behavior means that the controller is rewarded when the butterflies accidentally hits it which may be independent of actual improvements or deterioration of its behavior. In theory, it will continue to improve slowly and given enough time, it may find a way to beat the game systematically.

On the other end of the scale is a Survive Zombies. Unlike Chase, this game does not require the controller to develop a specific path to win. Quite the contrary, it needs to avoid the zombies, and as long as it stays safe, it will win. It does not even have to learn to react properly to the input, as it will be rewarded with a victory if it finds a safe spot that the zombies cannot reach.

## Deep vs non deep

Looking at the experiments with the two-loop method (section 6); it is clear a pattern forms. After the input representation has been trained with the newest input, the behavior layer must be retrained, which causes the new neural networks to start out with a lower fitness than they ended with last loop. With the current setup of running 50 generations of training input representation and 50 generations of training behavior, it results in the fitness overall not increasing with each loop.

One possible reason for this is that when collecting the input for training the input representation, only the input from the first runs are collected, which means that they are collected from the runs where the behavior layer has not yet adapted to the new input representation layer. This means that the input from the later generations in a loop, where the neural networks are likely to reach further in the game are not represented.

Furthermore, the input representation is only trained on the games behavior in the previous loop. Which means there is a higher likelihood that the neural networks become sensitive to input based around the current behavior, and less sensitive towards behavior that deviates from it.

## Playing two games

Playing different games in the GVG framework proved to have certain issues. First off, the different games rewarded the players in very different ways. Chase for instance only rewards the controller when it catches a spider, and only with one point. Survive Zombies on the other hand rewards the controller with one point for each honey token it collects. This may seem fair, but there are far more honey tokens in Survive Zombies than there are spiders in chase, and more honey tokens spawn when zombies die. Furthermore, the spiders in chase move away from the avatar. Therefore, changes to the neural network that would lead to picking up more honey tokens can easily take precedence even though catching one more spider is more important for winning. Two other impactful details became apparent from the experiments. The first is that we punish dying early very hard, so it has a significantly greater impact than increasing the game score. This makes games where you can die (like Survive Zombies) have a much greater impact on the fitness function than games where you cannot. The other thing is that games that are easier to win for the controller will have greater impact on the score. All of this makes games like Survive Zombies influence the fitness significantly more than games like Chase.

As can be seen in section 6.6 what happens is that the evolution experiments somewhat randomly until it finds a solution that wins Survive Zombies and at most gets one point in Chase. At this point, evolution stalls completely making hardly any progress. Evolution appears to solve the easier game first and then attempt to solve the other game. Given enough generations, it is possible that the other game would be solved as well.

When playing a game like Butterflies it is likely that the random behavior of the butterflies can have detrimental effects on the learning process. The game rewards the controller if the butterflies hits the avatar by accident regardless of whether it improves its behavior or not. This makes it hard to learn this game, but learning it while learning a different game is even harder. Even if the controller improves in the other game these improvements may be overwritten by the random rewards gained from butterflies hitting the avatar.

Ideally, the controller would develop behavior that reacted properly to the input it was given, always able to identify threats and goals in the current situation. Achieving this state proved to hold some very huge challenges. The input from the different games varies greatly and in addition to that, the game rules vary too. For instance, the controller is supposed to catch the NPCs in catch and butterflies, where it has to avoid some of them in Survive Zombies.

## Crossover

As described in section 4.2.3, the crossover currently works by joining two neural networks between layers. Experiments in section 6.4 show that evolving with crossover reaches a higher fitness faster than without crossover. This is true for both deep and non-deep neural networks. In fact, when training the input representation separately, the crossover almost doubles the fitness value increase for each generation.

However, whether this is because of the effect of two neural networks being merged or because of the side effect of having a higher chance of the top scorers in the population to spread out more is difficult to determine.

When crossover is enabled, there is a higher likelihood that elements of the top scorers are selected (see section 4.2.1), thus more parts of the top scorers are propagated out into the next population. When crossover is disabled, this does not happen. In that case, only two versions of the top scorers are kept: The original top scorer and a mutated version of it. It is possible that having the roulette selection with mutation only on the selected individual will produce similar effects.

Furthermore, the crossover works by splitting on layers. If the outermost layers are selected during the crossover process, then nothing will happen. This means that for the non-deep neural networks consisting of four layers, there is only a 50% chance a crossover will happen, and for the deep neural networks, only three layers are part of the crossover process, which means there is a 33.33% chance a crossover will occur. Alternatives to this crossover technique will be discussed in (REF FUTURE WORK).

## Limitations

One of the primary problems with regard to the experiments is that each variation have only been run once. Due to the nature of the current implementation, where each run might produce different results depending on which behavior is initially selected, each run might end up with varying results. This means that the difference between variations might have less to do with the parameters of the experiment, and more to do with the difference in evolved behavior.

# Conclusion

This project has focused on trying to see if it was possible to use neuro-evolution to evolve a controller capable of completing multiple games and to attempt to assess the effects of using deep neural network concepts on this particular problem.

A deep neural network concept was employed based on the idea of compressing the input down to a feature vector. The implementation turned out to have a negative effect and ended up degrading the growth of fitness through generations. Likely because of the way it was implemented, and not because the concept itself is flawed.

While the neural networks evolved performed better than a neural network with randomized values, they did not manage to win any games, other than games that had a simple solution, in the timeframe they were given. It is possible that allowing the experiments to run longer will produce better results.

Overall, this project has not been enough to conclude whether it is possible to create an AI capable of playing several games using neuro-evolution.

# Future Work

## NEAT

Neuro-evolution of Augmented Topologies (NEAT) is a method proposed for evolving neural networks incrementally from minimal structures. NEAT has characteristics that could possibly solve some of the issues discussed in the current implementation.

### Topology

The current implementation uses a fixed topology, meaning that no nodes or weights are ever added or removed, and all neural networks have the same structure. This simplifies cross over functions as it allows for the more crude methods of simply splitting on layers, or replacing sections of nodes, without the need to worry about lining up different topologies. However, it also means that the size of the neural network need to be chosen by hand beforehand. The disadvantage of this is that it is hard, if not impossible, to tell what the optimal number of hidden layers are, and how many neurons there should be in them.

NEAT starts out with a minimal structure that grows over time if it is advantageous in the environment. This allows structure to become part of the evolutionary variables, and means that neural networks who has unnecessary structure will not survive.

### Crossover

In NEAT, the crossover function is tightly coupled with an innovation index. This is to solve the problem of how to compare neural networks of different structures when doing cross over, and it allows crossover to blend the differences of each neural network into one with a completely new structure. This allows the crossover to have more of an effect since it blends the differences together, which means unlike the current implementation it is not possible to have a crossover that does nothing.

### Speciation

In the current implementation, there is a strong bias towards the behavior that is doing best in a given moment, and very quickly the other population becomes variations of the agent with the best fitness. One way to prevent this is to give alternatives more time to try to survive in the environment. This is also known as speciation, where one way of implementing it is that the more different neural networks are, the less likely they are to be combined and merged with each other, in essence creating different species. In the current implementation, this could be implemented by treating the neural networks thresholds and weights as a vector and have speciation be determined by the distance between these vectors.

However, this only works if the structures are the same. In NEAT, the speciation is determined by the previously mentioned innovation index. The more differences there are between innovation indexes, the further apart the two species are from each other. This allows structurally different neural networks to be comparable to each other.

For all the reasons mentioned, a natural next step would be to integrate NEAT into the solution.

## Fitness Function

As discussed section 7.3, the fitness function has a number of flaws, especially when it comes to comparing different games. The current implementation is very sensitive to early death and winning the game as well as different scoring mechanisms. This can be normalized by running a number of generations initially storing the greatest and lowest resulting fitness values as a frame of reference. Future runs would then have their fitness value evaluated against the frame of reference for that particular game, resulting in relative fitness values that are more comparable between games. It will be preferable to keep winning separate from the normalization, as this is the ultimate goal of the games. The impact of winning the game should therefore not be normalized.

## Enhanced input representation training

As described in section 7.2 the training of the input representation is lacking in some regards. The ideal way of training it is to have a random controller play the games in question a large amount of times and then store all the input vectors and use them to train the input representation. This way the input representation is trained in distinguishing the important features of the foreseeable game states.

# Bibliography

Hausknecht, M., Lehman, J., Miikkulainen, R., & Stone, P. (2014). A Neuroevolution Approach to General Atari Game Playing.

Koutník, J., Schmidhuber, J., & Gomez, F. (2014). Online Evolution of Deep Convolutional Network.

Levine, J., Congdon, C. B., Ebner, M., Kendall, G., Lucas, S. M., Miikkulainen, R., . . . Thompson, T. (u.d.). General Video Game Playing.

Perez, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S. M., Couetoux, A., . . . Thompson, T. (2014). The 2014 General Video Game Playing Competition.

Ross, B. (2014). General Video Game Playing with Goal Orientation.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *The MIT Press Journals*.

# Appendix

## One game deep and non-deep game score

## One game compression training

## Two Similar games compression training

## Two different games compression training