# NaiN

Machine learning about how to play games

Hannes Ingelhag
Fredrik Johnson

**Abstract**— This report will describe how to implement an AI with neural networks using the method neuroevolution of augmenting topolgies, NEAT, for NES games. The goal is for the AI to complete level one of Super Mario Bros.

**Index Terms**—AI, Neural networks, NEAT, Simulation, Bizhawk, LUA, Super Mario Bros

## 1 Background and Related Work

Super Mario Bros. is a revolutionary platformer for Nintendo Entertainment System(NES) and every other platformer. This simple 2D game where the goal is to save the princess by completing levels, has been the inspiration for almost all games coming after. The purpose is to implement an AI to evaluate the effiency of neural networks regarding machine learning algorithms. There are annually competitions held for developers writing AI for Mario games, however these are not learning about the environment. They are instead pre-programmed to handle every challenge that the AI might encounter. There has been a few cases where the implementation has used neural networks for Super Mario Bros, these have inspired this project.

## 2 Theory

Neural networks are models inspired by the biological brain, which uses interconnected neurons to evaluate a given input. Given a large amount of unknown input data a neural network can estimate an approximate function to solve the given problem. The network is trained with a known input and desired output and builds up the connection between neurons with weighted links. By using mutation the brains envolve into new generations, where the goal for the network is to solve the given problem more efficiently than its previous generation. This is achieved by mutating previous generations. When the network has evolved and can solve given tasks, no one has any insight about the meaning of the different neurons and links within the network, shown in *figure 1*

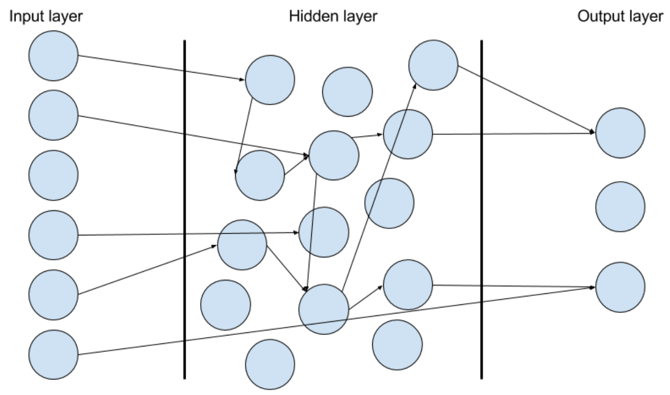

Fig. 1: General representation of a neural network

In general all neural networks works similarly even though the structure differs. The biggest difference is the method used for mutating and evolving a network. In this implementation Neuro-Evulotion of Augmenting Topologies (NEAT) has been chosen as the method.

- *Hannes Ingelhag, Linköping University, Sweden e-mail: hanin602@student.liu.se*
- *Fredrik Johnson, Linköping University, Sweden e-mail: frejo989@student.liu.se*

### 2.1 NEAT

The NEAT algorithm has been developed from other well recognized algorithms, e.g;Topology and Weight Evolving Artificial Neural Networks (TWEANNs) [1], used for neural networks. A brief explanation of what NEAT is and how it has been used in the implementation will be given, however complete explanation of the algorithm can be seen here [2]. Several neural networks are used in NEAT to be able to mutate and evolve to a better network. These networks are represented as brains and stored in a population, which contains all the different brains in a generation. The population is simply a number that limits the amount of brains in the simulation. When all the brains in a population has been simulated and evaluated they are mutated to create a new generation of brains.

What characterizes NEAT from previous algorithms is the introduction of unique innovation numbers for the links between neurons and how the network is mutated between generations. The innovation number is a unique index for each link preserved over different generations. The innovation numbers characterizes the different brains in a population. These traits makes it possible to categorize the brains into different species which makes the mutations easier to compute. The brains for a new generation is created by either combining two brains or copying a brain. The new brain is then mutated utilizing four different mutations: *Weight-*, *Link-*, *AddNode-*, *Enable-or-disable-mutation*, following each of these mutations are described below.

#### 2.1.1 Mutations

Probability of mutation is different depending on the mutation, there is a different probability to add a new neuron in contrary to change the weight of a link, see *table 1* in appendix. When a brain is about to be mutated the different probabilities are modified with +/- 5% from the current values, this is done to assure that a brain will change over time.

**Link Weight Mutate**
Changes all weights of the links between neurons within a brain. The new weight is either modified or gets a new random value depending on a probability of perturbation which is a constant value, see *table 2*, set by the developer.

**Add Link Mutate**
Adds a new link between two random neurons, one of them cannot be an input node. The newly created link is assigned a unique innovation number, *figure 2*. This mutation has a chance to be performed with or without bias, which means that one of the two nodes is guaranteed to be an input node.

**Add Node Mutate**
Adds a new node to the network. This is done by choosing a random existing link in the network. Create a new neuron with two new links, one to the input and one to the output given by the

already existing link. Lastly disable the old link. *figure 3* shows how a new node is added to the network.

**Enable or Disable Link Mutate**
Takes a random link in the network and activate/deactivates it depending on current state.

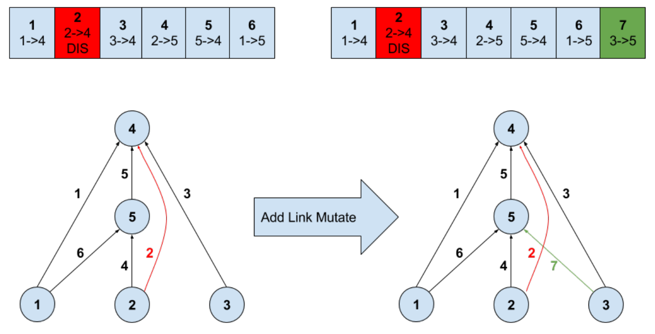

Fig. 2: How a new link could be added during an add link mutation. The second link exists but is currently disabled

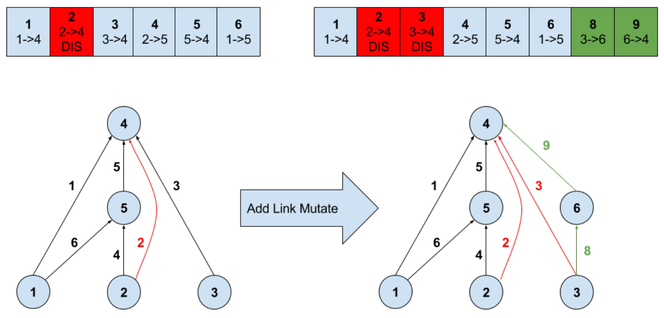

Fig. 3: How a new node could be added during an add node mutation

## 3 Method

During simulations several constants were used, see *table 2*. The implementation is using an emulator Bizhawk [3] which allows to extract data and manipulate the game using Lua. The extracted data is used as input, see *figure 4*, for the network and the output is which buttons on the emulated control that has been pressed, see *figure 5*. When the computer percieves the world, there are three different representations:

- 1 - something that mario can stand on
- 0 - empty space
- -1 - moving sprites(generaly enemies)

### 3.0.2 Structure

As previously mentioned a population is used to keep track of the brains, however this population is not enough to keep track of everything efficiently. Therefore a structure according to *figure 6* was created to simplyfy the task of managing the population.

### 3.0.3 Simulation

The implementation iterates through each brain in the population and evaluates the fitness achieved by the brain. Fitness is a measure of a brains performance, this is calculated using *equation 1*.

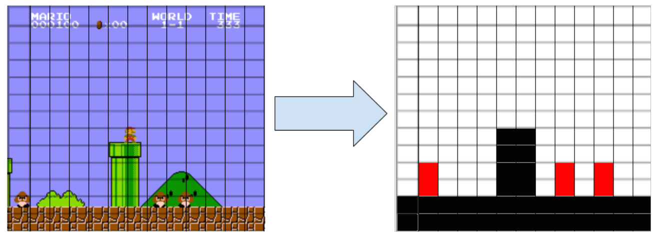

Fig. 4: This in the input for the neural network. Value 1 is black, 0 is white and -1 red.

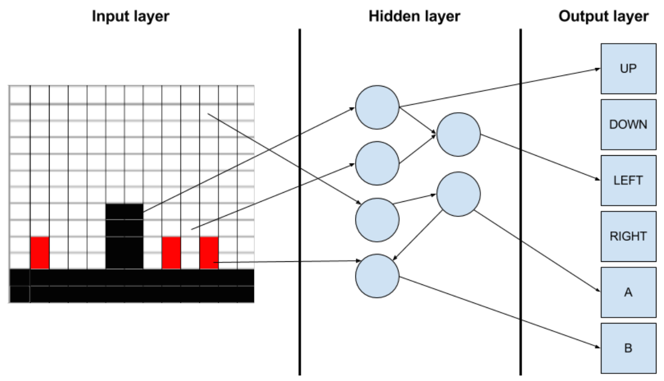

Fig. 5: Illustrate how the neural network is structured in this project

$$Fitness = xPosition/Time \tag{1}$$

Fitness depends on the position of Mario and the time it took to get there, because if the goal is to complete the level as fast as possible and the finish line is always to the right. When all the brains has been tested, the population is then evaluated and a new generation is generated according to *figure 7*.

**Remove weak brains**
All the brains are sorted according to the fitness value, the bottom half is then culled.

**Remove stale species**
Species that has not improved their best fitness value over the set value of stale species,see *table 2* generations are culled. All the brains gets a global rank according to their fitness

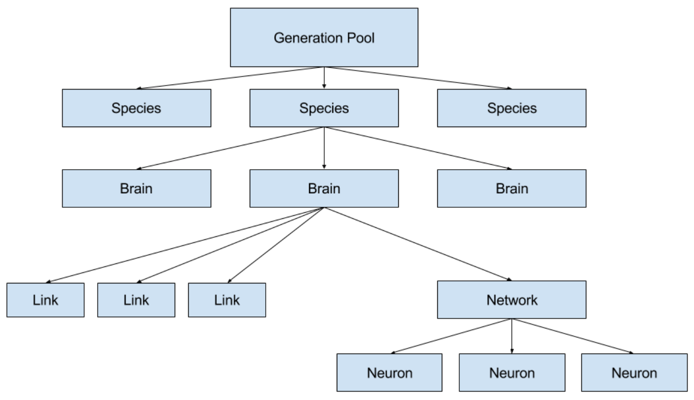

Fig. 6: The structure of the generation pool, including species, brains, links and neurons. A more specific model can be found in the appendix, see *figure 9*

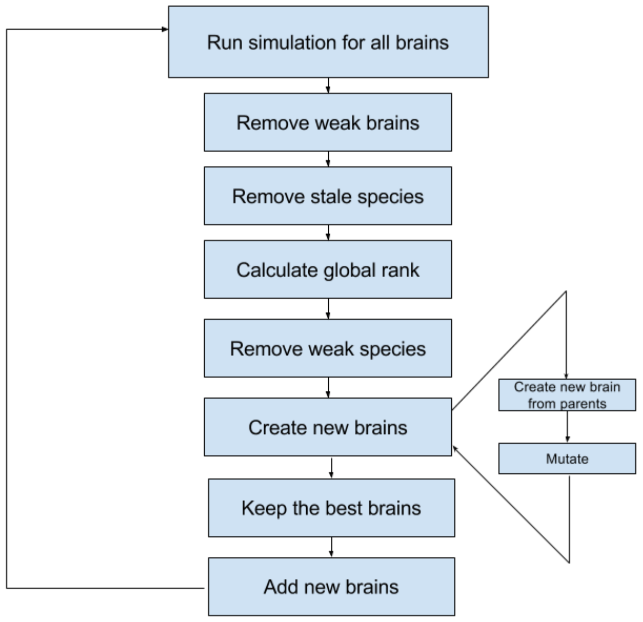

Fig. 7: The algorithm used to create a new generation

**Calculate global rank**
Calculates the global rank for every brain in the population based on the fitness.

**Remove Weak species**
Using the global rank of the brains in the species an average rank value for the species is calculated. If the average rank is below the set threshold the species is removed.

**Create new brains**
When a new brain is created there are two major steps. First decide if the brain is going to be a combination of two brains or just a copy of one. When the new brain has been created it is mutated according to the four mutations explained earlier.

**Cull species to best brain**
Only the best brain in each species are kept, this is done to guarantee that the new generation does not deteriorate over time.

**Add new brains**
The newly created brains are added to the population, which species the brain belongs to is chosen according to *equation 2*, where the current brain gets compared with the best brain in each species.

$$C1*E/N+C2*D/N+c3*W < DELTA\_THRESHOLD \quad (2)$$

- C1 = DELTA_DISJOINTS
- C2 = DELTA_EXCESS
- C3 = DELTA_WEIGHTS

where the $E$ and $D$ are excess and disjoints, differences in links between brains. $W$ represents the difference in weights between the brains *equation 3*

$$W = \sum |weightA - weightB| \quad (3)$$

**Run Simulation again**
When the new brains has been added the algorithm starts over.

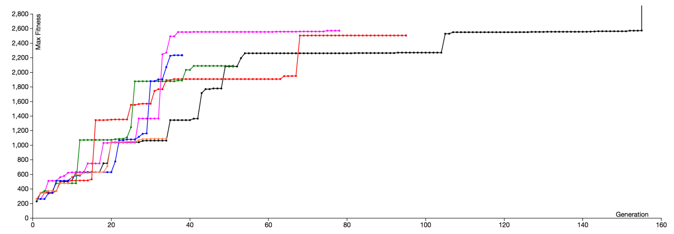

Fig. 8: Show six different simulations with six different populations. Black line represents a population of 100 brains, red 200, megenta 300, green 400, blue 500 and coral 600. All simulations ran for 24 hours.

## 4 Results

Simulations was done with 100, 200, 300, 400, 500 and 600 sized populations which ran over a 24 hour training session, the graphs shown in *figure 8* shows how the different populations evolved over time.

The simulation which used a size of 100 brains managed to complete level within 24 hours on first level. This population was also used to try level two. Since the brains had been evolved over such a long time the simulation got a head start trying out level two.

## 5 Discussion

Since Bizhawk was used for this implementation we were stuck with the limitations of it e.g. memory leaks each time a new brain was tested which inevitably led to crashes of the emulator and time consuming since everything had be rendered so our brains could play the game.

We had limited time and resources so we figured that a small test would help us out finding the best size for a population. This test was ran over a 24 hour period with six different sizes spanning from 100-600, where the simulations with lower population were best. Therefore we used a population of 100 for the large scale simulation.

To visualize all data we got from the simulations a web-based application was built. This helped us a lot when we decided which population should been used in the simulations. It also gave us the last input before Mario died and how each species looked like for every generation.

Using our chosen population another test was carried out, the goal of this was to see if several simulations using the same values would converge against the same result or scatter.With the result of this test we conclude that only one simulation for each population was needed.

Compared to our source of inspiration that managed to complete level one in 24 hours using a population of 300, we completed level one in 24 hours with a population of 100. That is a decrease of one third of the population but otherwise the same constants.

## 6 Conclusions and Future Work

During this project we conclude that it is possible to use neural network with the specific method NEAT to complete a Super Mario Bros level. This implementation would also solve other games provided the correct input and given enough time. Time is the greatest constraint using this implementation since we only simulate one brain at a time and the fact that it needs to be rendered.

Improvements that can be done to the implementation is trying other the constant values, which is a tiresome trial and error process. Acquire an emulator that does not need rendering to withdraw the world inputs. Write a script that manages to handle several emulators and distribute the simulations to run several simulations at the same time.

## References

[1] Zhang, B.-T. and Muhlenbein, H. (1993). Evolving optimal neural networks using genetic algo- rithms with Occams razor. Complex Systems, 7:199220.

[2] Kenneth O. Stanley and Risto Miikkulainen, 2002, Evolving Neural Networks through Augmenting Topologies, Evolutionary Computation, Volume 10, Number 2.

[3] Bizhawk, 2015-09-28, *http://tasvideos.org/BizHawk.html*

## 7 APPENDIX

Table 1: Probability rates used by the algorithm when it mutates a brain

| Probability Rate Name | Explanation |
|---|---|
| Link weight mutate | Probability of changing the weights of the links withing the brain |
| Add link | Probability of adding a link between two neurons |
| Bias add link | Probability of adding a link between an input node and a neuron |
| Node mutation | Probability of adding a new node in the network |
| Enable Mutation | Probabilty of activating a deactivated link |
| Disable Mutation | Probability of deactivating an activated link |

Table 2: Constant used in this implementation

| Constant | Value |
|---|---|
| DELTA_DISJOINT | 2.0 |
| DELTA_EXCESS | 2.0 |
| DELTA_WEIGHTS | 0.4 |
| DELTA_THRESHOLD | 1.0 |
| STALE_SPECIES | 15 |
| MUTATE_CONNECTIONS_CHANCE | 25% |
| LINK_MUTATION_CHANCE | 200% (it will always make a link mutate two times) |
| BIAS_MUTATION_CHANCE | 40% |
| NODE_MUTATION_CHANCE | 40% |
| DISBALE_MUTATION_CHANCE | 40% |
| ENABLE_MUTATION_CHANCE | 20% |
| PETURB_CHANCE | 90% |
| CROSSOVER_CHANCE | 75% |
| STEP_SIZE | 0.1 |
| TIMEOUT_CONSTANT | 70 |
| MAX_NODES | 2000 |
| POPULATION | 200 |

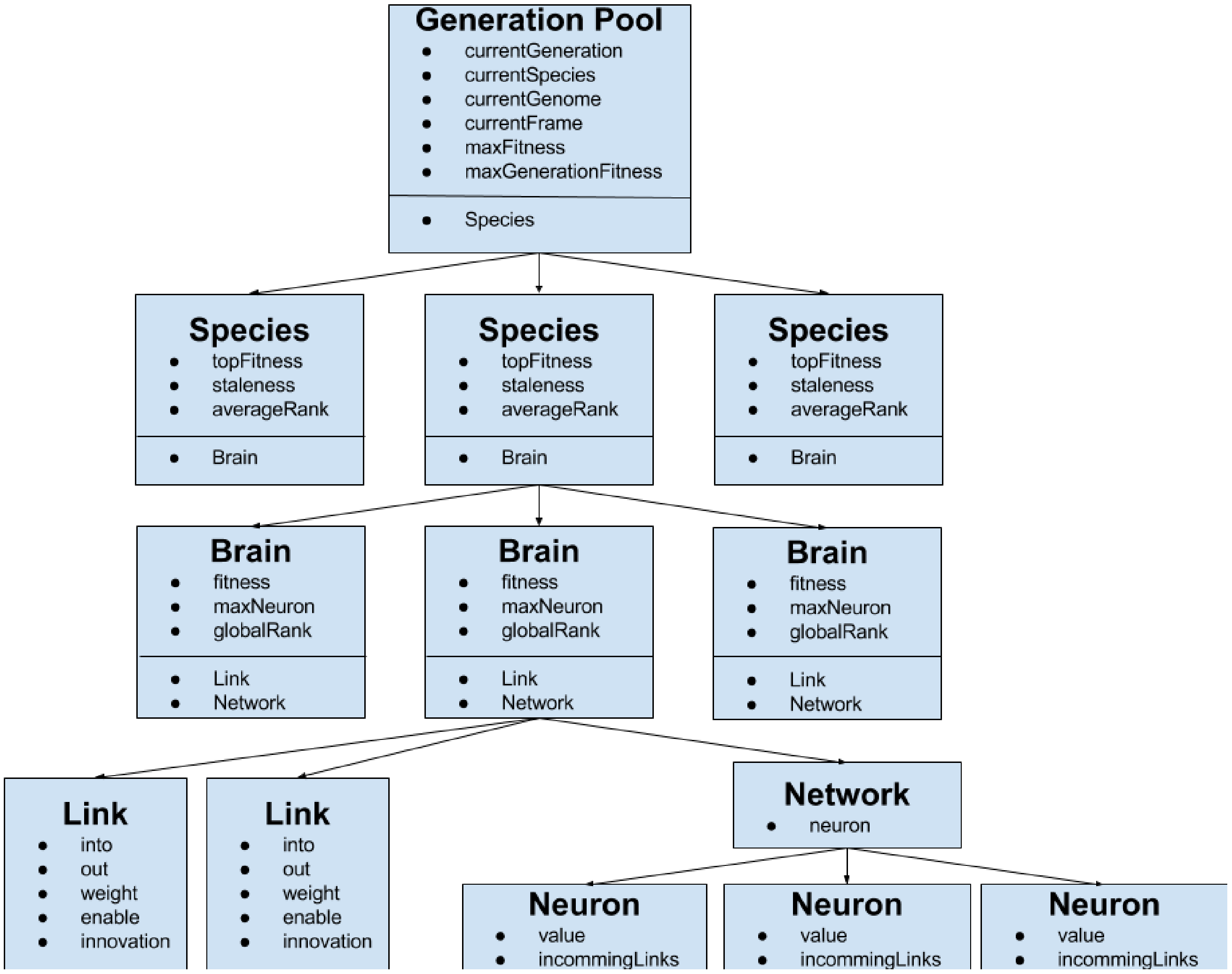

Fig. 9: A detailed structure over the generation pool