UNIVERSITY OF ZAGREB

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS No. 6886

Evolution of Artificial Neural Network Architecture

Matija Bačić

Zagreb, May 2020.

Contents

Introduction

One of the main goals of computer science is to efficiently process data. In the early days of artificial intelligence, symbolic paradigm provided best algorithms for data inference and was the dominant paradigm until late 1980s. It uses logic and search to draw conclusions about the given data. This branch of artificial intelligence includes state space search, minimax, automated reasoning, and other algorithms.

In the late 1980s a new paradigm gained momentum. Connectionism attempts to construct a system whose architecture is similar to that of a human brain. Fundamental building block of this paradigm is a neuron. Name of this paradigm implies that the main data structure is an interconnected network of neurons called artificial neural network (ANN). Function and design of a neuron in an artificial neural network is inspired by neurons in a human brain.

Main distinction of symbolic paradigm and connectionism is the method of teaching a computer how to react to new data. Symbolic paradigm attempts to encode the knowledge and methods of inference in advance. That is to say, computer is only able to infer using predefined methods of drawing a conclusion. User specifies the rules of the modeled domain. On the other hand, connectionism creates data structures and algorithms whose main purpose is to train those data structures. It could be said that connectionism transfers the responsibility of obtaining methods of inference on a specific domain to the computer. This avoids the need for symbolic representation of everyday tasks.
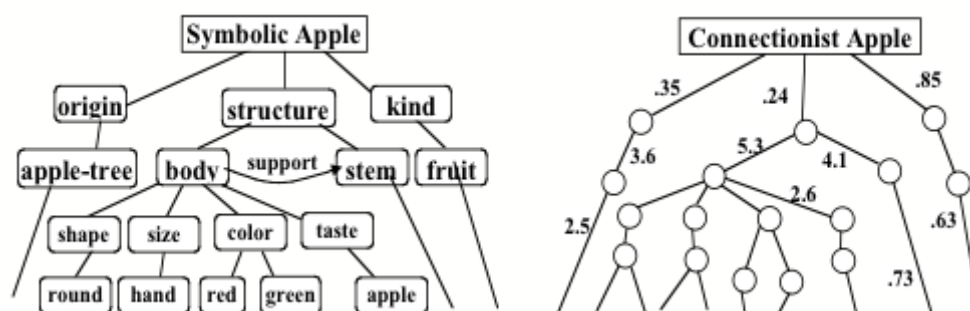

Figure 0.1 Symbolic vs connectionist apple [1]

On the Figure 0.1 Symbolic vs connectionist apple [1] a description of an apple can be seen from both paradigm perspectives. While symbolic paradigm's description of an apple is a tree of its features, connectionist apple is a weighted tree whose interpretation is beyond our comprehension.

Each of these approaches have advantages as well as disadvantages. Symbolic approach has explainable and verifiable behaviors. Connectionist solutions are robust to noise and able to learn non-symbolic data. While one might sound inferior or superior to the other, good choice of a paradigm heavily depends on type of a problem that needs to be solved. Furthermore, recent studies have shown limitations of using only one of these approaches. Hybrid system that uses both paradigms may possibly counteract the weaknesses of symbolic as well as connectionist approach.

Further study in this thesis will only be based on connectionist approach and methods of training artificial neural networks.

1. Introduction to artificial neurons

As stated before, main component of any artificial neural network is an artificial neuron. Artificial neuron is a processing unit which roughly resembles neurons in a human brain. Looking at it from a mathematical point of view, artificial neural network is a directed graph. Thus, artificial neuron is nothing but a node in that directed graph. Hereinafter, term neuron will be used instead of artificial neuron and term neural network instead of artificial neural network.
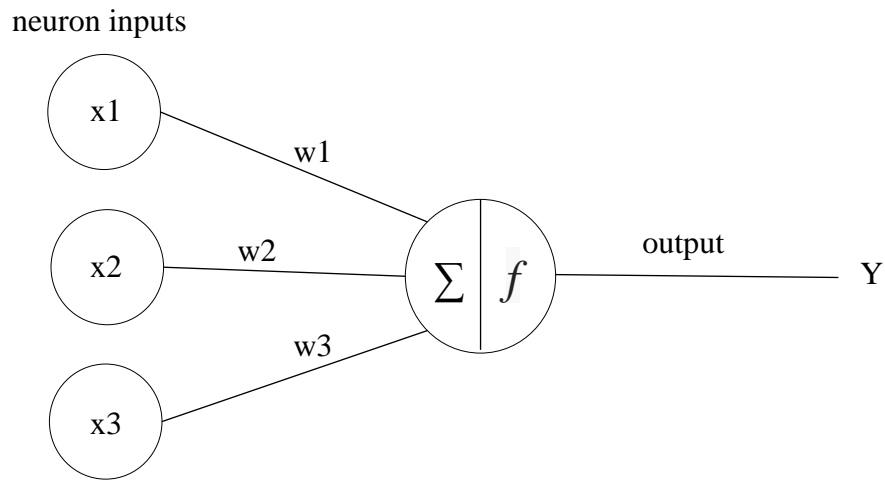
neuron inputs

Figure 1.1 Artificial neuron

As illustrated on Figure 1.1, neuron is a node with inputs and an output. It takes a weighted sum of its inputs, passes it through an activation function and sends that output to the next neuron in a network. Output of a neuron Y can be expressed as follows:

$$Y = f\left(\sum_{i=1}^{n} x_i \cdot w_i\right) \tag{1}$$

Where n represents number of inputs, x_i is the i-th input of a neuron, w_i is a weight associated with the i-th connection and f is an activation function. Usually, each neuron in a neural network has the same activation function. If that is not the case, f_j can be written instead of f as the activation function of the j-th neuron in the network.

Activation function is used for constraining the output of a function. For instance, if the model needs to answer a question with a yes or no answer, an activation function with
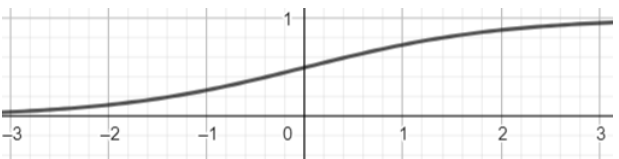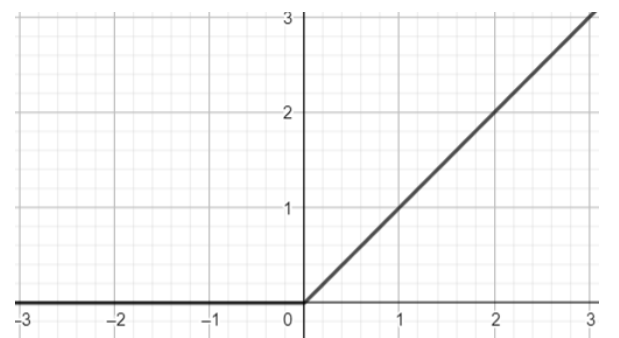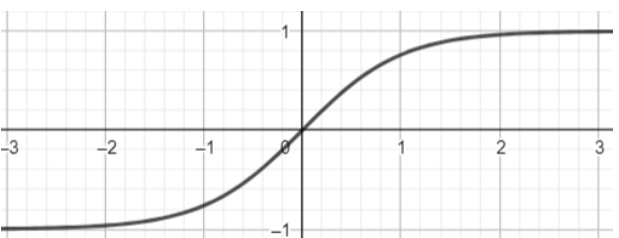
rage [0, 1] could be used. Value closer to zero would indicate a negative answer, whereas values closer to one a positive answer.

Limitations of this model can be seen if the output for value 0 at the input should be a positive answer (value closer to 1). Assuming that every input is 0, neuron would not be able to produce output other than 0. Therefore, a bias node, whose value is usually set to 1, is added as input to a network. Effect of a bias node can be interpreted as left or right shift of an initial activation function. Taking bias value b into consideration, output of a neuron can be calculated as follows:

$$Y = f\left(\sum_{i=1}^{n} x_i \cdot w_i + b\right) \tag{2}$$

Some of the most popular activation function used can be seen in Table 1.1 along with range and their graphs.

Table 1.1 Well known activation functions

Activation function	Range	Graph
Sigmoid (Logistic curve) $\sigma(z) = \dfrac{1}{1 + e^{-z}}$	<0, 1>	
ReLU (Rectified linear unit) $R(z) = \max(0, z)$	[0, +∞>	
Hyperbolic tangent $T(z) = \tanh(z)$	<-1, 1>	

2. Neural networks

2.1. Introduction to artificial neural networks

In a neural network, neurons can be divided into input, output, and hidden layer. Input layer neurons are ones which gather information for processing while those in the output layer give information back after processing. Hidden layer neurons are responsible for calculation or mapping the input to the output. Example of a neural network can be seen on Figure 2.1. This neural network consists of 3 input layer nodes, 2 hidden layers each with 4 nodes and 2 output layer nodes. Each connection has a weight associated with it which, for the sake of not cluttering, is not shown in the figure. Each node is equivalent to an earlier defined neuron. Furthermore, each connection is a directed connection from left to right on the figure below.

INPUT LAYER   HIDDEN LAYER   HIDDEN LAYER   OUTPUT LAYER
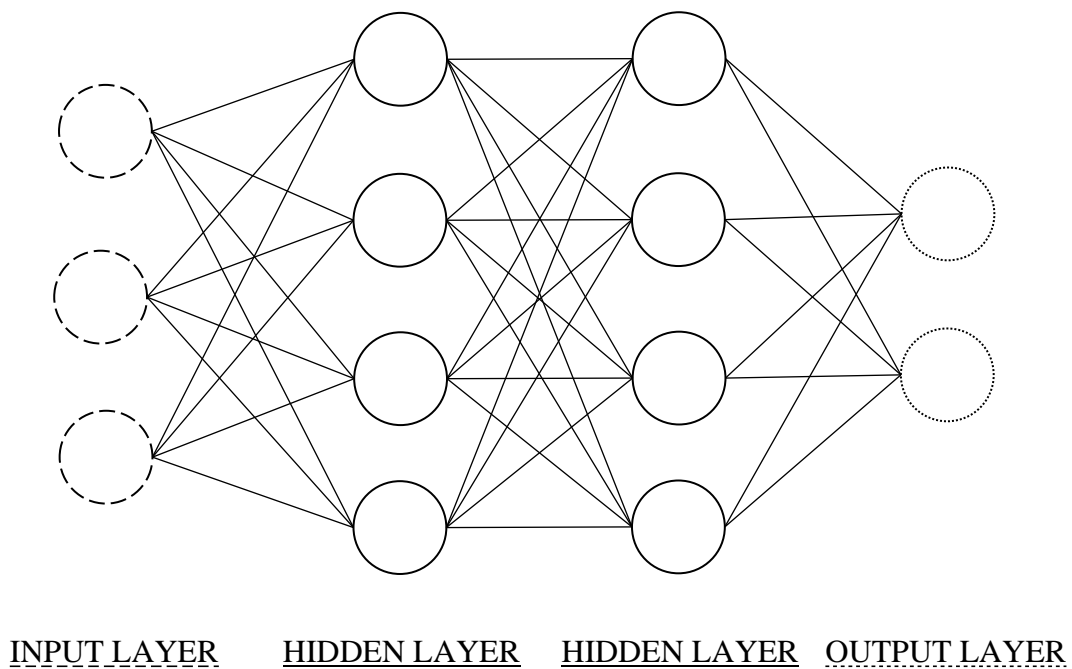
Figure 2.1 Neural network example

Evaluation of a neural network is a process in which network calculates values of the output nodes based on the input values. Values of input layer nodes are set manually. For each node, activation function of a weighted sum is calculated and passed to the next neuron in a chain. Pseudocode of such process is demonstrated in Code 2.3, while Code 2.1 and Code 2.2 show basic data structures of neural networks.

Code 2.1 Neuron data structure

```
struct Neuron {
        Neuron[] inputs;
        float value;
        int visited;
}
```

Code 2.2 Neural network data structure

```
struct NeuralNetwork {
        Neuron[] neurons;        // All neurons in a network
        Neuron[] input_neurons;  // Input neurons of a network
        Neuron[] output_neurons; // Output neurons of a network
}
```

Code 2.3 Function for evaluating neural network

```
void evaluate_network (NeuralNetwork network) {
        foreach node in network.neurons {
                node.visited = 0;
        }
        foreach node in network.output_neurons {
                evaluate_node_recursively (node);
        }
}


void evaluate_node_recursively (Neuron node) {
        if (visited)
                return;

        node.visited = 1;
        float value = 0;
        foreach node_in in node.inputs {
                evaluate_node_rec (node_in);
                value += node_in.value * weight(node_in, node);
        }
        node.value = activation_function(value);
}
```

Problem with this approach for evaluating neural network is speed of execution. A way around that would be to utilize the power of a graphics processing unit (GPU). GPU has efficient algorithms for matrix multiplication. To make use of its processing power, expression (2) will have to be rewritten in a matrix form.

6

2.2. Matrix representation

Suppose that layer i has n nodes, and a layer i + 1 has m nodes. Let a_i be the n-dimensional vector row of layer i values and a_{i+1} m-dimensional vector row of layer i + 1 values. It can be written:

$$a_{i+1} = f(a_i * W_i) \qquad (3)$$

where f is an activation function of the neural network and W_i is an n×m matrix of connection weights between layers i and i + 1 (as can be seen in Figure 2.2).

$$a_i = \begin{bmatrix} a_{11} & a_{12} \end{bmatrix}$$

$$a_{i+1} = \begin{bmatrix} a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix}$$

$$W_i = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix}$$

Layer i   Layer i + 1
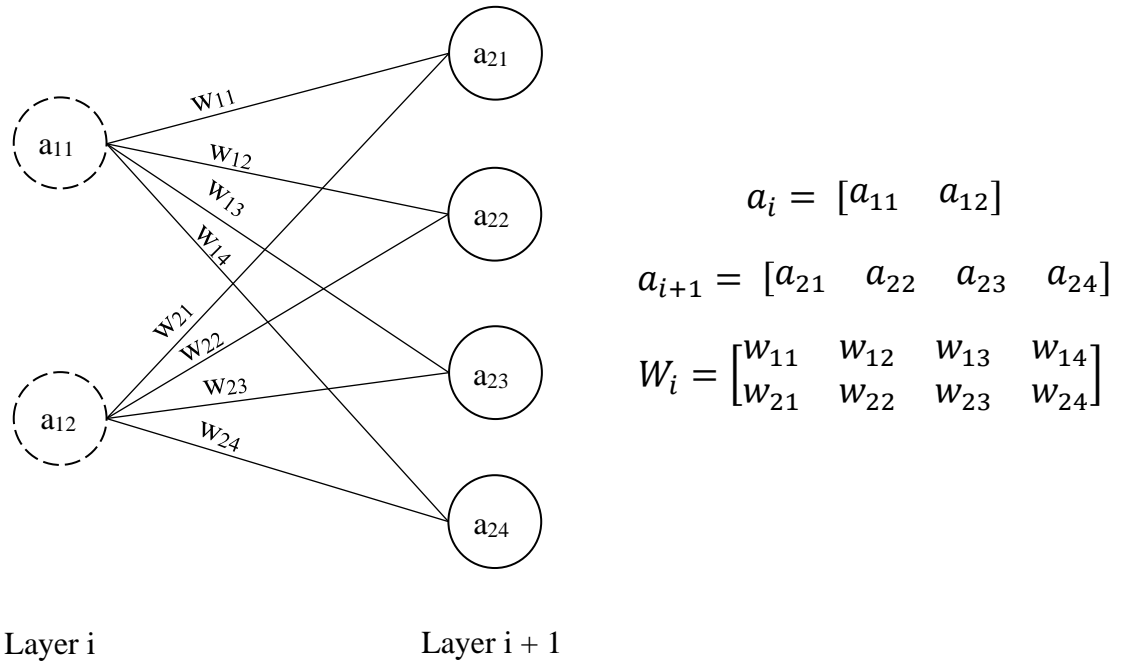
Figure 2.2 Matrix representation of a neural network

From equation (3) and Figure 2.2:

$$\begin{bmatrix} a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} = f \left(\begin{bmatrix} a_{11} & a_{12} \end{bmatrix} \cdot \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} \right) \qquad (4)$$

2.3. Training neural networks

Training is a process in which neural networks adapt to the given domain and learns input-output mapping. In other words, training is a process of minimizing output error (absolute difference between expected and calculated output). Two methods proved as most popular for such training: backward propagation and neuroevolution.

Backward propagation is a mathematically oriented method for training artificial neural networks. It uses, what is called, forward propagation to calculate the output of the network for a given input combination. Then, using output error and a technique called gradient descent, this method adjusts connection weights through a step known as backward propagation, or backpropagation. This training method will not be discussed further in this thesis.

Evolutionary algorithm for training neural networks relies on Darwin's theory of evolution and natural selection. More general method for evolving any data structures is known as genetic algorithm. This method can be explained by a six-step process:

1. Generate an initial population of size N
2. Calculate fitness of every individual in the population (evaluation)
3. If stop condition is satisfied, terminate the process
4. Create a new population
   a. Select two individuals from a population
   b. Create an offspring through a crossover operation
   c. Mutate the offspring and add it to the new population
5. Repeat step 4 while size of the new population is less than N
6. Go to step 2 with population = new population (advance into next generation)

At the start of an algorithm, initial population is generated with a fixed size N. Individuals of this population, in the case of neuroevolution, are artificial neural networks. Those individuals can be randomly generated or taken from the output of some other algorithm.

Fitness function plays a big role in this process as it determines how much an individual is adapted to its environment. Bigger the fitness function is for an individual, grater the probability of passing its genetic material to the next generation.

Stop condition of evolutionary algorithm can be expected fitness value or number of generations an algorithm is supposed to evolve before terminating. Often, a combination of both conditions is used to avoid infinite loop problem.

Evolving a population into the next generation requires creating a new population using selection, crossover, and mutation operators on an existing population. Selection operator chooses an individual from a population based on some criteria. Typically, an individual has higher chance of being picked if said individual has high fitness score relative to the others in the population. Once two individuals have been selected, crossover operator creates an offspring of those two genomes. Offspring is then mutated using a mutation operator before it is stored in the new population.

2.4. Neural network topology problem

Under the premise that all networks in a population have the same topology, mutation and crossover operators modify only connection weights rather than the whole structures. Consequently, one must choose network topology in advance and that can create a problem. There are some well-established topologies fit for various tasks.

One of such topologies, known as deep feedforward network (DFFN), can be seen on Figure 2.1. A simpler form of that topology can be achieved using only one hidden layer (feedforward network - FFN). Any directed acyclic graph can represent feedforward neural network. That is to say, a feed forward network is such topology in which connections do not form a cycle.

Another widely used topology, fit for sequence processors, is recurrent neural network (RNN). Cycles in a graph of said topology allows it to possess an internal state (a memory). This makes them appropriate for modeling tasks like handwriting recognition and speech recognition. Subtype of a recurrent neural network, called Elman network, can be seen on Figure 2.3.
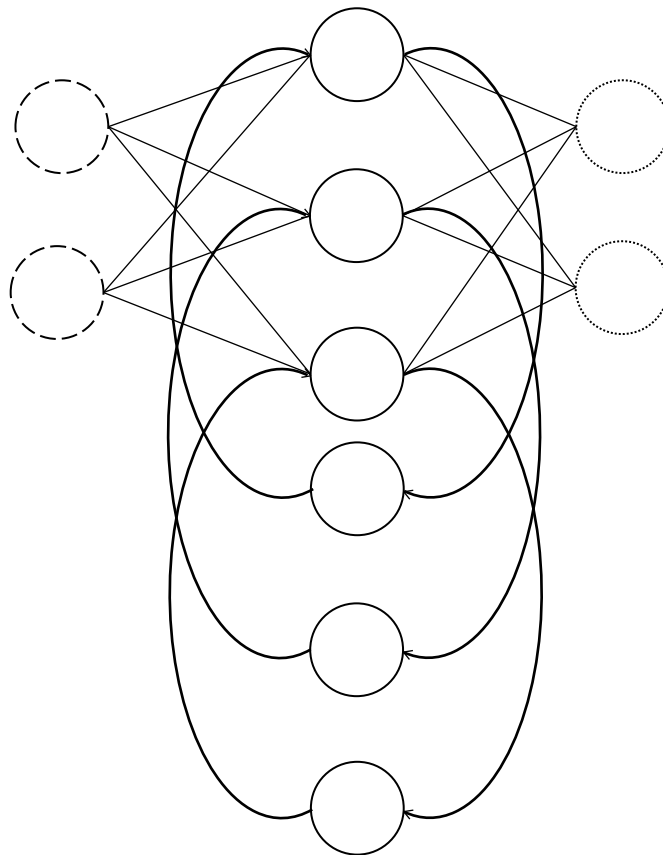

Figure 2.3 Recurrent neural network – Elman network

3. NeuroEvolution of Augmenting Topologies (NEAT)

Nevertheless, question arises – would it be more efficient to evolve network topology along with connection weights? NeuroEvolution of Augmenting Topologies (hereinafter NEAT) is a method of evolving neural networks which proposes a solution to the given question. Idea is to evolve whole structure of a neural network along with connection weights rather than having a fixed, in advance selected, topology.

The original paper on NEAT [2] raises three important questions:

1. How to encode neural networks to allow crossover of different topologies?
2. How to protect topological innovation that needs few generations to be optimized so that it does not disappear from population too early?
3. How to minimize network topology though evolution?

As will be explained, NEAT contains solutions to each of these questions. In the subchapters that follow, main concepts and ideas of NEAT will be described. For further study, original paper on NEAT is available online from the MIT press [2].

3.1. Encoding genes

Genome is a sequence of network connections and nodes. In other words, neural network is stored as an array of node connections and an array of nodes which those connections connect.

Connection is an ordered pair of nodes that are linked together. Each connection has a unique innovation number which tracks historical markings and is beneficial in crossing over networks with different topologies. Innovation number helps in finding and lining up corresponding genes in a crossover operation (as will be explained later). Connections with the same innovation number connects two nodes with identical ids where id of a node determines node type (sensor, hidden, or output node). To prevent explosion of innovation numbers and node ids, tracking of existing innovations and ids is done globally.

Main components of each connection, aside from innovation number, are connection weight, in node id, out node id, and an enable bit. Variable parts of each connection are its weight and an enable bit. Enable bit indicates whether connection is active or not. In other words, enable bit specifies whether a connection takes part in a network evaluation process.

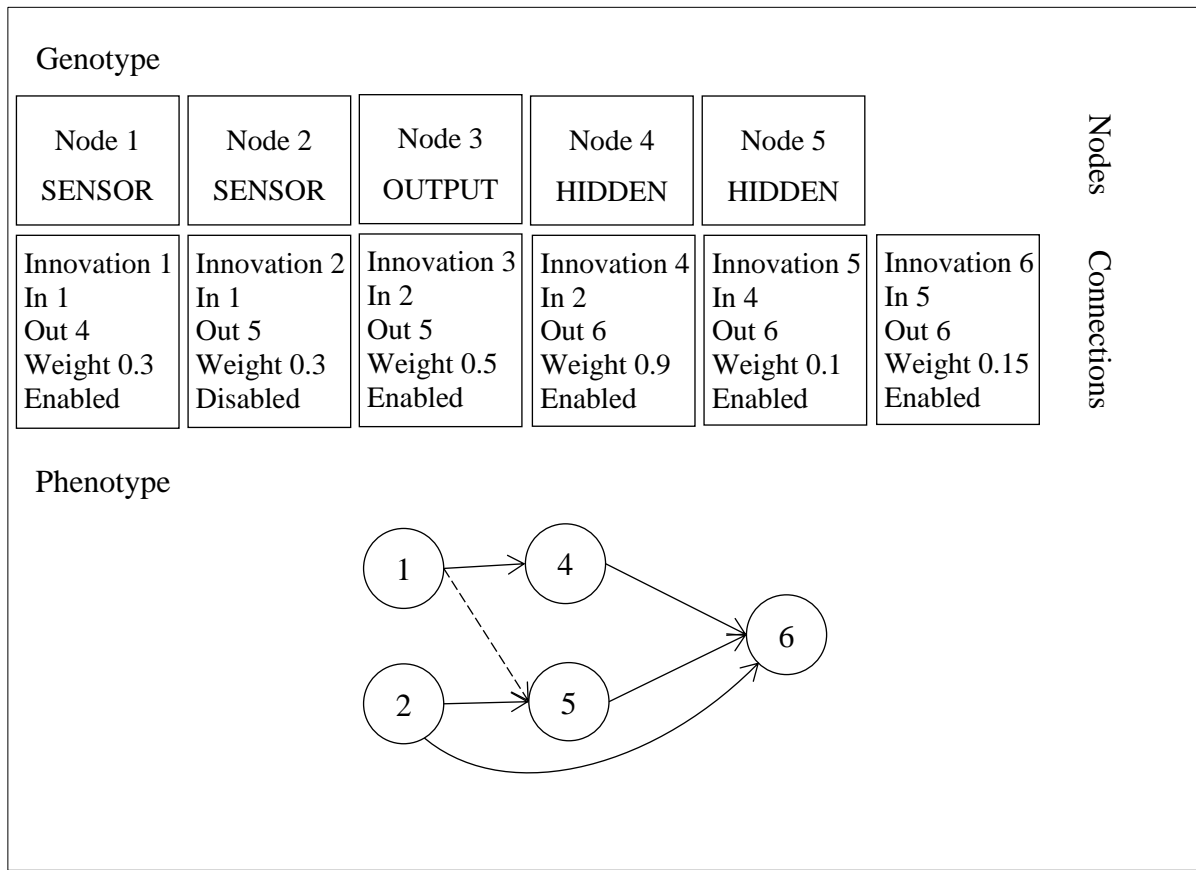An example of genotype to phenotype mapping is shown on Figure 3.1.

Figure 3.1 Genotype to phenotype mapping example

3.2. Mutation

Standard mutation (connection weight mutation) is present in the NEAT method of neural network evolution. Connection weights are mutated in two ways: weight perturbation and random weight value assignment. Perturbation is achieved by adding a random, small, uniformly, or normally distributed value to the existing weight.

To allow topology augmentation, NEAT proposes two additional mutation types: add node mutation and add connection mutation. Add node mutation effectively splits an existing connection in two and adds a new hidden node where the old connection used to be. What happens is an original connection is disabled and then a node and two new connections are created. First connection in node id is copied from the original connection in node id, while second connection out node id is copied from the original connection out node id. Left unfilled node ids are populated with the id of a new node. Weight of the first connection is

set to value one, and weight of the second connection is copied from original connection so not to disturb phenotype and let it adjust the topological innovation.

Add connection mutation connects two unconnected nodes with a new connection. New connection receives a random weight. Both mutations can be seen on Figure 3.2 and Figure 3.3.

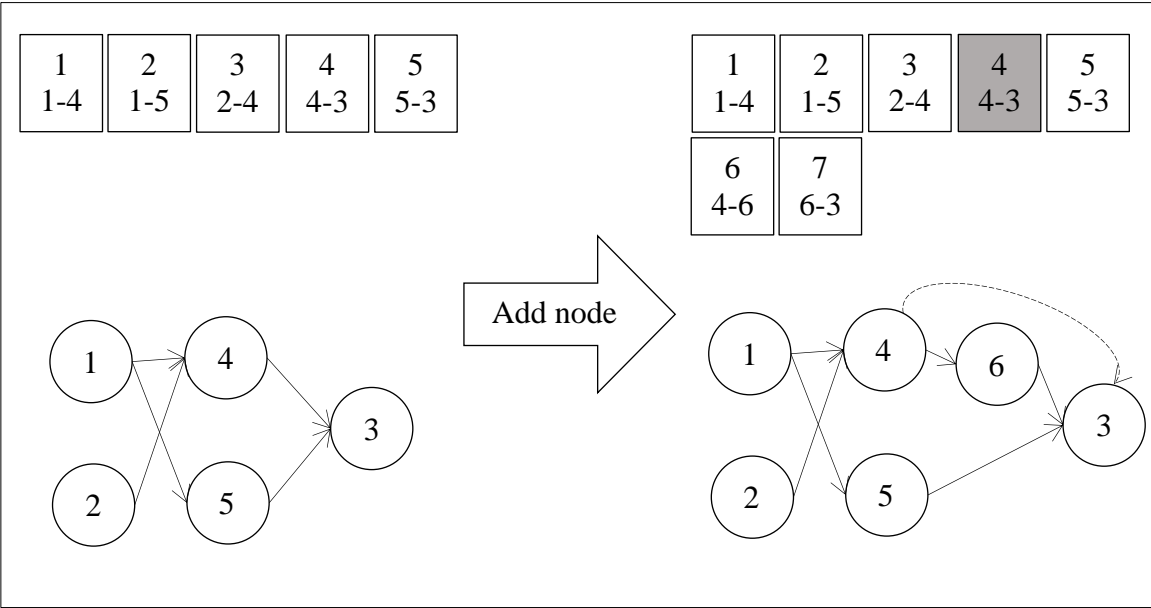Next subchapter explains how crossover of two, seemingly, incompatible topologies is accomplished.
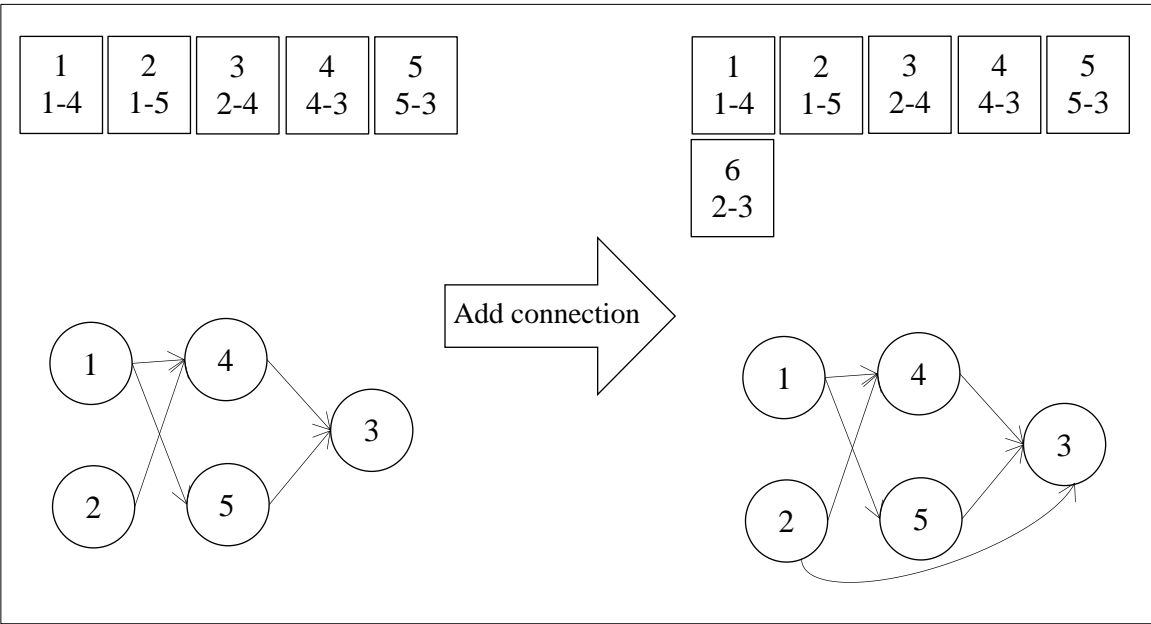

Figure 3.2 Add node mutation

Figure 3.3 Add connection mutation

3.3. Crossover operator

Next question that needs to be answered is how networks with different topologies can be crossed over in a meaningful way. Innovation number plays a big role in a proposed solution. The idea is to line up genomes of two networks by innovation number. Those genes that match will be randomly inherited from any of the parents. Ones which do not have matching gene in the other parent we call disjoint if they are in the middle or excess if they are at the end of a genome. Both disjoint and excess genes are inherited from more fit parent or from both parents randomly in case of equal fitness. This process is illustrated on Figure 3.4.
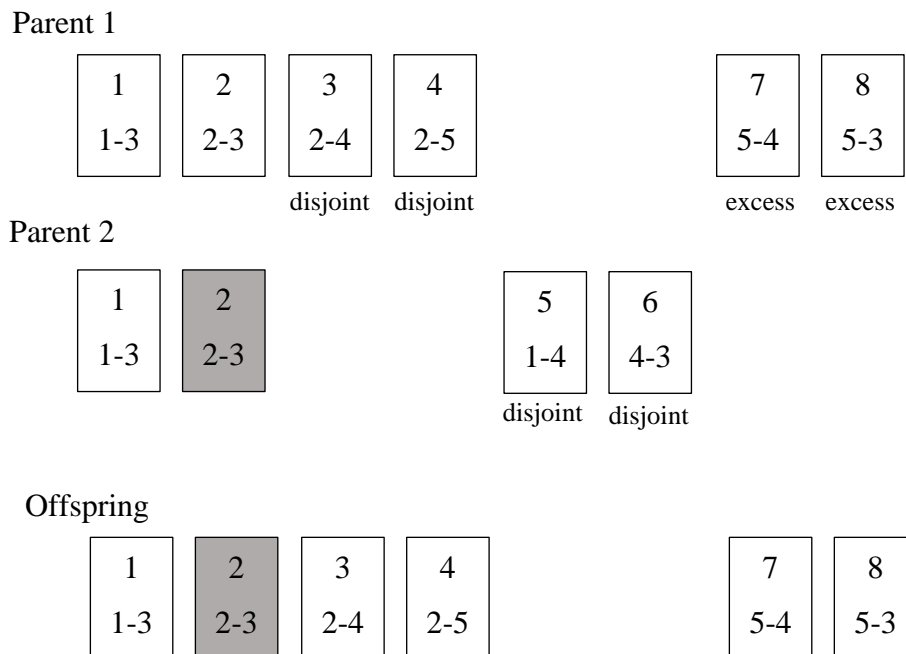
Figure 3.4 NEAT Crossover operator assuming parent 1 to be more fit (grayed out genomes represent disabled connections)

In this way, algorithm slowly enhances topology of a population and has broader state space to search. However, it is shown that new topological advancements take longer to optimize and thus risk dying out quicker than less complex topologies. As it is explained in the next subchapter, speciation is a mechanism to counteract this trend of initial fitness reduction as a result of topological innovation.

3.4. Speciation

Speciation is a mechanism of separating genomes into groups based on topological similarities to reduce initial negative effect of topological innovation and protect more complex structures. It allows individuals in a population to compete only within their own species. Historical markings play a big role here as well. By calculating number of excess, disjoint genes and average weight difference between matching genes, similarity among two neural networks can be quantified. Compatibility distance δ measures said similarity. More distant two networks are, less they are compatible for crossover. Compatibility distance can be expressed as:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W} \tag{5}$$

where c_1, c_2 and c_3 are fixed real numbers, E represents the number of excess genes, D number of disjoint genes, W average weight difference between matched genes, and value N number of genes in a larger genome.

The idea is to decide on a threshold δ_t above which two genomes are no longer considered similar enough to belong to the same species. It can be shown that for every new genome, compatibility distance needs to be calculated only against species representatives (one random genotype) rather than each individual in a group. Once value δ falls below the threshold for the first time, genome is added to that species without further calculation against remaining species.

To prevent domination of one species on a whole population, fitness scores of each genome is adjusted. By penalizing similar topologies, NEAT stops the species from becoming too big regardless of how high the fitness score is. Mechanism for achieving this effect is called explicit fitness sharing developed by Goldberg and Richardson in 1987. Adjusted fitness is calculated as follows:

$$f_i^{adj} = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{6}$$

where sh function is of value 0 when the argument is above δ_t, otherwise it has a value of 1. Effectively, sum in the denominator represents the number of genomes in a species in which genome i resides.

3.5. Minimizing topology

In previous sections first two questions have been answered. Crossover using innovation number has been introduced as well as speciation for protecting topological innovation that needs few generations to be optimized. Minimizing topology is one of many goals of NEAT algorithm. Original authors propose starting with initial population in which all individuals are the same, have zero hidden nodes and in which every input node is connected to every output node. Thus, algorithm starts searching state space with simple topologies first and evolve towards more complex ones using add node or add connection mutations.

3.6. Implementation

Implementing NEAT has shown to be not an easy task. One of many implementation ideas is discussed in this section.

Class ConnectionMarkings and NodeMarkings represent unchangeable connection and node attributes. Also, they are responsible for tracking global node id and connection innovation numbers. These classes also contain node id to node type and connection innovation to in node and out node id mappings. To prevent explosion in the number of ConnectionMarkings and NodeMarkings, before add node or add connection mutation, all existing markings are checked for availability.

Classes Connection and Node represent concrete instances in a neural network (genome). Attributes of these classes can be change during weight mutation or crossover: weight, enabled for a connection and value for a node.

A Genome is a sorted dictionary of node id to node mappings and connection innovation to connection mappings. Usage of sorted dictionary instead of a list is justified because of quicker fetching and enumerating though nodes as well as connections.

Class NEATPopulation represents a collection of individuals (population of neural networks). Its job is to iterate though generations and separate individuals into species.

Finally, class Algorithms is a static class which contains methods for a crossover operation and evaluation of a network (feedforward or reccurent).

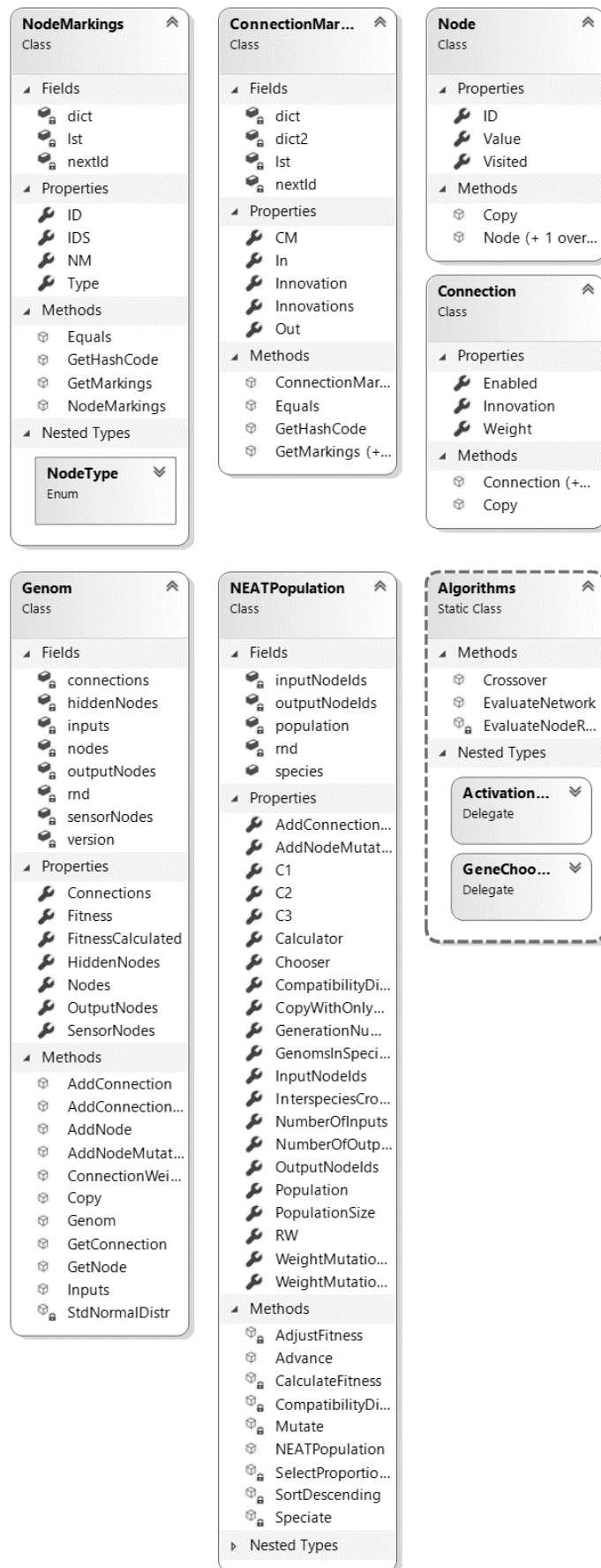A class diagram of this implementation can be seen on Figure 3.5.

NodeMarkings
Class

⊿ Fields
- 🔒 dict
- 🔒 lst
- 🔒 nextId

⊿ Properties
- 🔧 ID
- 🔧 IDS
- 🔧 NM
- 🔧 Type

⊿ Methods
- Equals
- GetHashCode
- GetMarkings
- NodeMarkings

⊿ Nested Types

NodeType
Enum

ConnectionMar...
Class

⊿ Fields
- 🔒 dict
- 🔒 dict2
- 🔒 lst
- 🔒 nextId

⊿ Properties
- 🔧 CM
- 🔧 In
- 🔧 Innovation
- 🔧 Innovations
- 🔧 Out

⊿ Methods
- ConnectionMar...
- Equals
- GetHashCode
- GetMarkings (+...

Node
Class

⊿ Properties
- 🔧 ID
- 🔧 Value
- 🔧 Visited

⊿ Methods
- Copy
- Node (+ 1 over...

Connection
Class

⊿ Properties
- 🔧 Enabled
- 🔧 Innovation
- 🔧 Weight

⊿ Methods
- Connection (+...
- Copy

Genom
Class

⊿ Fields
- 🔒 connections
- 🔒 hiddenNodes
- 🔒 inputs
- 🔒 nodes
- 🔒 outputNodes
- 🔒 rnd
- 🔒 sensorNodes
- 🔒 version

⊿ Properties
- 🔧 Connections
- 🔧 Fitness
- 🔧 FitnessCalculated
- 🔧 HiddenNodes
- 🔧 Nodes
- 🔧 OutputNodes
- 🔧 SensorNodes

⊿ Methods
- AddConnection
- AddConnection...
- AddNode
- AddNodeMutat...
- ConnectionWei...
- Copy
- Genom
- GetConnection
- GetNode
- Inputs
- 🔒 StdNormalDistr

NEATPopulation
Class

⊿ Fields
- 🔒 inputNodeIds
- 🔒 outputNodeIds
- 🔒 population
- 🔒 rnd
- species

⊿ Properties
- 🔧 AddConnection...
- 🔧 AddNodeMutat...
- 🔧 C1
- 🔧 C2
- 🔧 C3
- 🔧 Calculator
- 🔧 Chooser
- 🔧 CompatibilityDi...
- 🔧 CopyWithOnly...
- 🔧 GenerationNu...
- 🔧 GenomsInSpeci...
- 🔧 InputNodeIds
- 🔧 InterspeciesCro...
- 🔧 NumberOfInputs
- 🔧 NumberOfOutp...
- 🔧 OutputNodeIds
- 🔧 Population
- 🔧 PopulationSize
- 🔧 RW
- 🔧 WeightMutatio...
- 🔧 WeightMutatio...

⊿ Methods
- 🔒 AdjustFitness
- Advance
- 🔒 CalculateFitness
- 🔒 CompatibilityDi...
- 🔒 Mutate
- NEATPopulation
- 🔒 SelectProportio...
- 🔒 SortDescending
- 🔒 Speciate

▷ Nested Types

Algorithms
Static Class

⊿ Methods
- Crossover
- EvaluateNetwork
- 🔒 EvaluateNodeR...

⊿ Nested Types

Activation...
Delegate

GeneChoo...
Delegate

Figure 3.5 NEAT implementation class diagram

17

4. NEAT performance evaluations

4.1. XOR problem

First test of NEAT performance is the XOR problem. It is a test suggested in the official NEAT paper. Idea is to evolve a neural network which is capable of preforming XOR (exclusive or) function. Truth table for said function is shown in Table 4.1.

Table 4.1 XOR truth table

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Notice, xor function for input (0,0) gives output of 0 which means no bias node is needed. Thus, our initial population has 2 input nodes and 1 output node. All inputs are connected to all outputs, consequently total of 2 connections are present in genomes of initial population. NEAT parameters used for this problem can be seen in the table below.

Table 4.2 NEAT parameters

Parameter	Value
Number of input nodes	2
Number of output nodes	1
Population size	100
Weight mutation probability	60%
Weight perturbation probability	90%
Add node mutation probability	2.5%
Add connection mutation probability	2.5%
c_1	1
c_2	1
c_3	0.4
δ_t (compatibility distance threshold)	3
Minimum number of genomes in species survival	5
Percentage of population copied with only mutation	50%

With 100 iterations of the algorithm and limiting number of generations in each iteration to 100, test results are quite promising. Out of 100 iterations, solution was found in every iteration (fail rate was 0%), while optimal solution was found in 18% of iterations. Optimal solution is defined as a solution that correctly solves a given problem and has least number of nodes needed. In the case of the xor problem, topology of the optimal solution can be seen on Figure 4.1. Minimum number of nodes in a solution was 5, while maximum was 14. The average number of nodes was 7.5567 with a standard deviation of 2.4. Solution was found in average of 43.2 generations with minimum being 15 and maximum 95. Overall, it can be concluded that NEAT yields quite consistent results. According to the official paper on NEAT, it performs at least 90% better than conventional neuroevolution algorithm and at least 10% better than other well-known topology evolving algorithms (DPNV, TWANN and others).
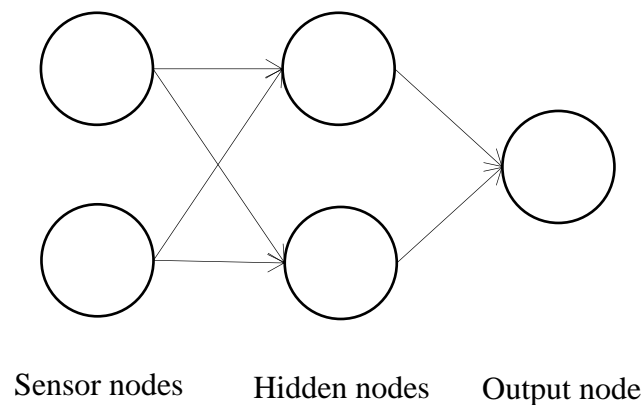

Sensor nodes   Hidden nodes   Output node

Figure 4.1 XOR optimal topology

4.2. Single-agent system in Unity

Next test scenario is built using Unity engine. Unity is a popular 3D game engine. It allows for easier and faster game and simulation development. The idea is to create a simulation of a popular game "Flappy bird" and train a neural network to play the game.

Flappy bird game has a few simple rules. Only character in a game is a bird which moves at a constant speed from left to the right of the screen. Camera follows the bird. Along the way, pipes appear – top and bottom pipe, leaving a small gap between them. Goal is to pass thought as many pipes as possible without touching them or touching the ground. Only control a player has is a flap. A flap is performed by applying a small force on a bird in the upwards direction.

Application is divided into three game modes (three states): player, ai, load ai. Player mode allows the user to play the game by pushing a space bar on a keyboard. AI mode is a state in which NEAT algorithm runs and trains a neural network to play the game. Lastly, best fit individual is saved after each generation of AI mode and user can load it in the load AI mode. Game can be seen on Figure 4.2.
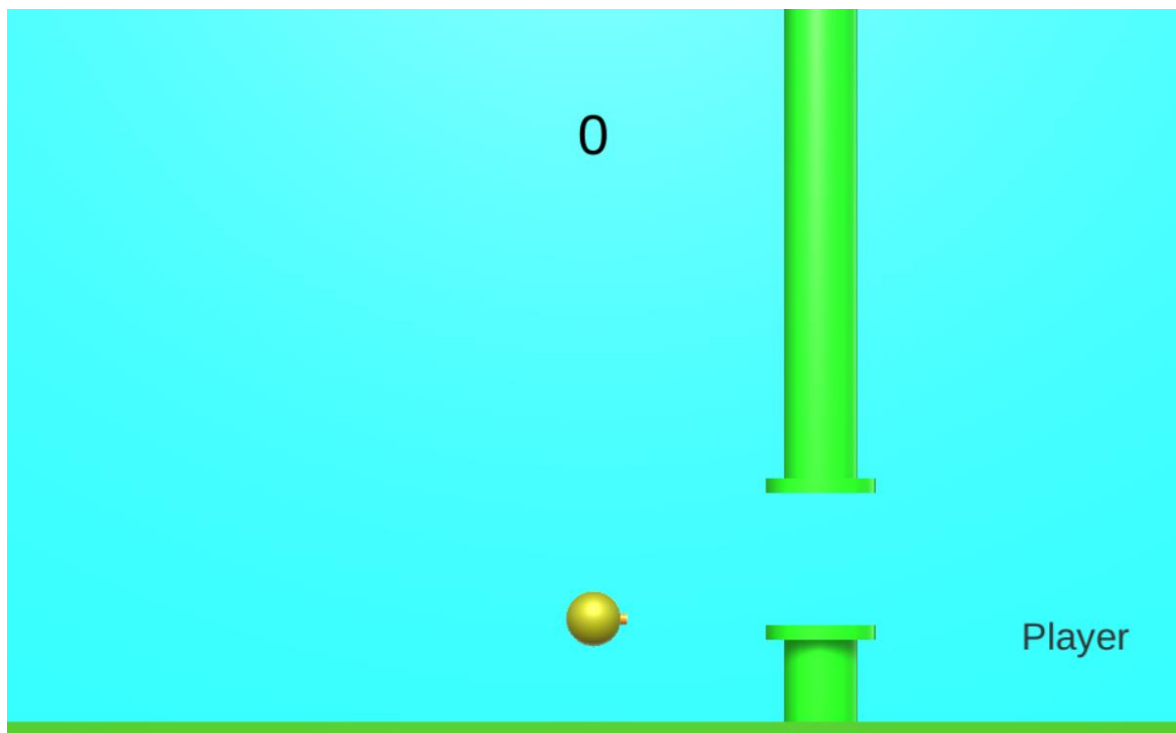

Figure 4.2 Flappy bird simulation

While Unity is a 3D game engine, this simulation is built as to appear 2D. That is, camera moves only along the x-axis along with the bird and is always pointed to the same direction. Near the top of the screen, current score is shown. Score reflects through how many pipes a player (human or AI) has successfully passed.

Parameters of the NEAT algorithm can be seen in Table 4.3. Each neural network has 3 inputs and 1 output. Inputs of a network are as follows: horizontal distance to the next pipe (d), vertical distance to the top pipe cover (dt), vertical distance to the bottom pipe cover (db) (as can be seen on Figure 4.3). A single output is expected: whether to jump or not. Each network in a population is evaluated simultaneously. This is achieved by moving bird game objects into a separate physics layer and turning off collision detection between objects in that layer. This process in shown on Figure 4.4.

No bias nodes were needed. Activation function used is a sigmoid function. Value between 0 and 0.5 on the output is mapped to not jumping in the given frame, while for value between 0.5 and 1 a space bar press is simulated. To obtain finer separation between networks, fitness of a network was calculated based on distance traveled rather than achieved score.

Table 4.3 Flappy bird NEAT parameters

Parameter	Value
Number of input nodes	3
Number of output nodes	1
Population size	100
Weight mutation probability	80%
Weight perturbation probability	90%
Add node mutation probability	3%
Add connection mutation probability	5%
c_1	1
c_2	1
c_3	0.4
δ_t (compatibility distance threshold)	3
Minimum number of genomes in species survival	5
Percentage of population copied with only mutation	25%

From 100 runs of the algorithm following statistics was calculated. NEAT algorithm with given parameters takes on average 7.4 generations with standard deviation of 2.3 to find

the solution to the given problem. Here, a solution is defined as one which can achieve a score of at least 50.
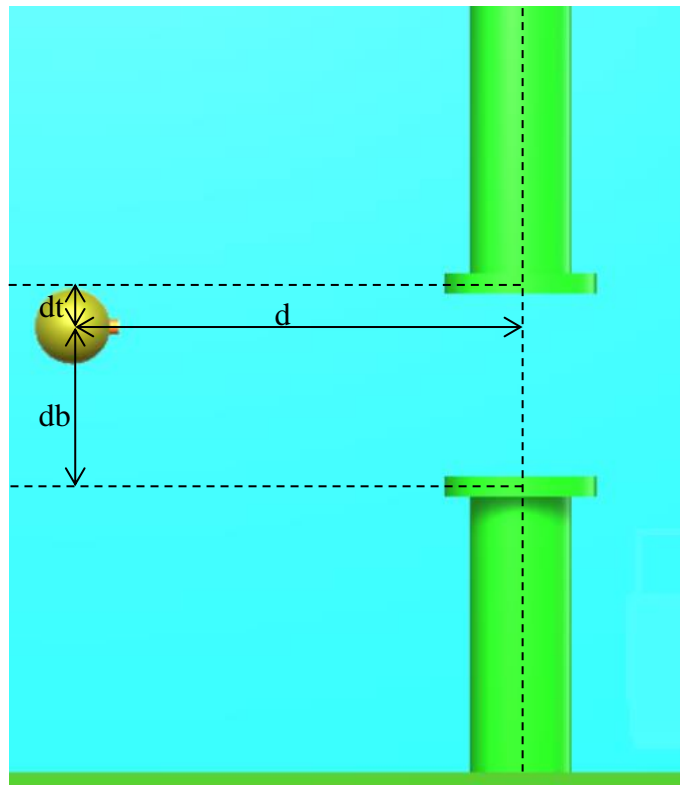

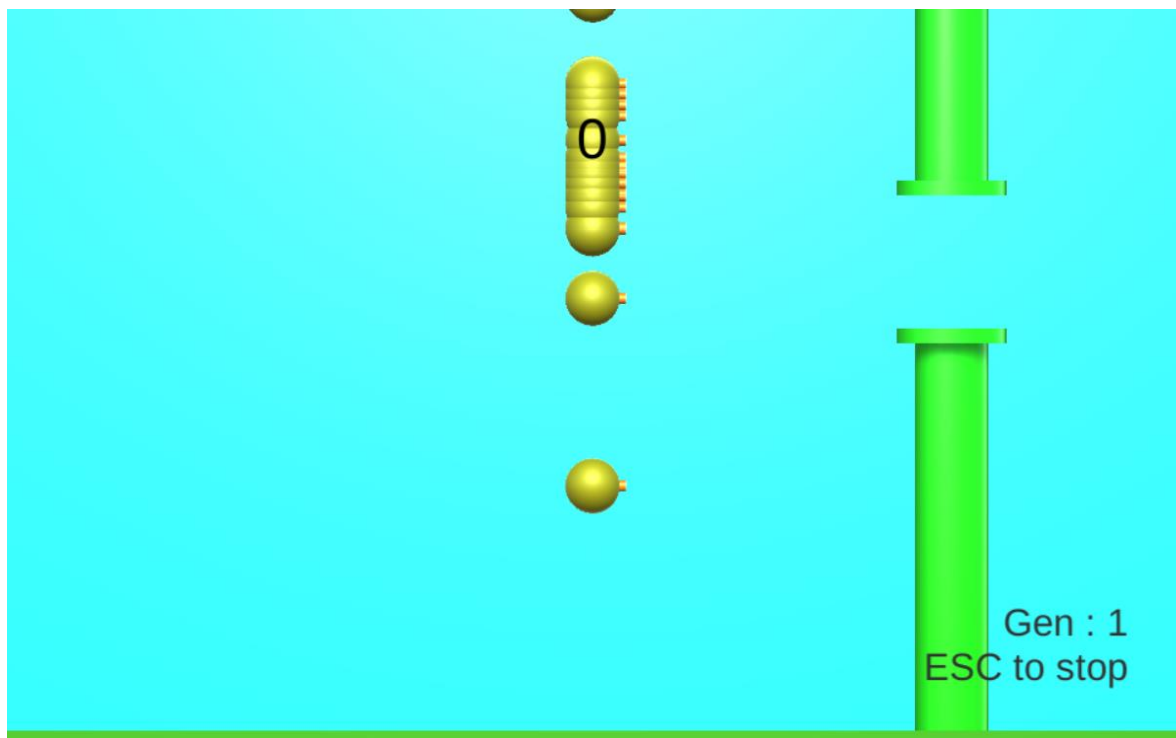Figure 4.3 Neural network input

Figure 4.4 Training neural networks using NEAT

Additionally, one representative run was taken (meaning a run whose minimum number of generations before reaching the goal falls within one standard deviation from the previously calculated mean) on which some statistical parameters were calculated. Figure 4.5 shows the progression of maximum and average fitness per generation, while Figure 4.6 shows a network topology of the trained network. Total of 9 generations passed in this run before a network satisfied trained conditions.
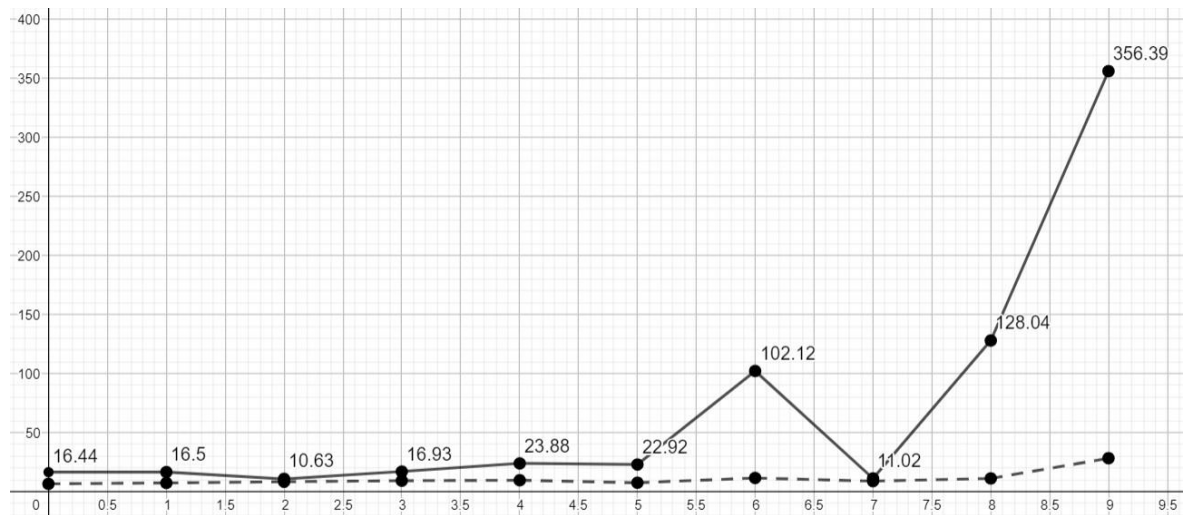

Figure 4.5 Maximum (filled line) and average (dashed line) fitness per generation
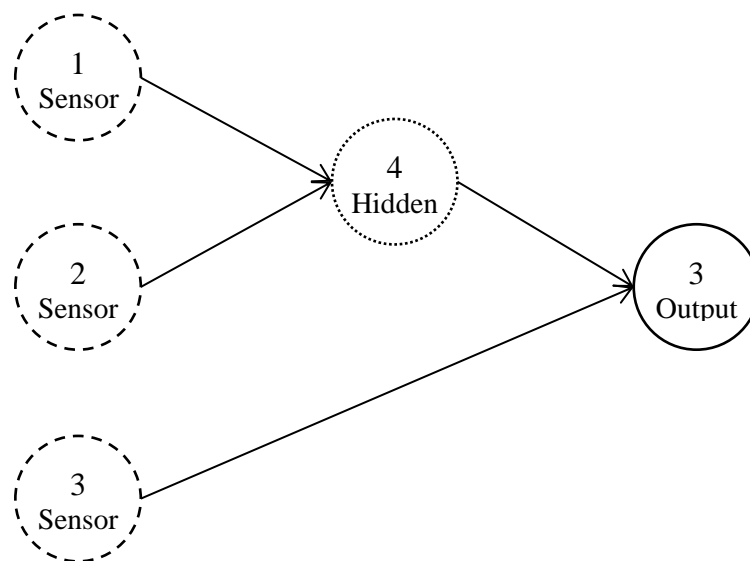

Figure 4.6 Topology of a trained network

Conclusion

Neuroevolution is a powerful tool for training artificial neural networks. It does not involve complex mathematical concepts so one does not need to have an advanced mathematical background to understand theory behind it. It is a powerful, yet simple to understand idea. Neural networks have shown in recent years a great potential for solving all kinds of problems and modeling complicated ideas. Of course, every system has its limitations and so do artificial neural networks. There are some problems which are just too farfetched for neural networks. An example of such would be deciding whether an integer is prime or not. It would be very hard, if not impossible, for a neural network to learn such behavior without memorizing the whole training set. For these kinds of problems, it would be better to utilize the power of other machine learning tools which reside in symbolic paradigm.

Nevertheless, neural networks have shown to be an excellent tool for most everyday tasks. This is why neuroevolution with augmenting topologies (NEAT) is so important in this field as it solves problem of manually choosing network topology and opens doors for further research in this area. Concepts like tracking historical markings and disabling genes in a genome without fully removing them show great potentials for further development.

Bibliography

[1]   Marvin Minsky, Logical vs. Analogical or Symbolic vs. Connectionist or Neat vs. Scruffy, Summer 1991., *Artificial Intelligence at MIT Expanding Frontiers*, https://web.media.mit.edu/~minsky/papers/SymbolicVs.Connectionist.html, May 2020.

[2]   O. Stanley, K.; Miikkulainen, R. Evolving Neural Networks through Augmenting Topologies, The MIT Press Journals, 2002.

[3]   Dertat, A., Applied Deep Learning - Part 1: Artificial Neural Networks, 8.8.2017., https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6, May 2020.

[4]   Sharma, S, Activation Functions in Neural Networks, 6.9.2017., https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6, May 2020.

[5]   Heidenreich, H., NEAT: An Awesome Approach to NeuroEvolution, 4.1.2019., https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f, May 2020.

[6]   Ayinal, R., A Digestible Overview of Neural Networks, 26.3.2020., https://medium.com/the-open-manuel/a-digestible-overview-of-neural-networks-2cd717b1d0f7, May 2020.

[7]   Unknown author, Effect of Bias in Neural Network, unknown date, https://www.geeksforgeeks.org/effect-of-bias-in-neural-network/, May 2020.

[8]   Unknown author, Backpropagation for dummies, 12.6.2015., https://codesachin.wordpress.com/2015/12/06/backpropagation-for-dummies/, May 2020.

[9]   Bajada J., Symbolic vs Connectionist A.I., 8.4.2019., https://towardsdatascience.com/symbolic-vs-connectionist-a-i-8cf6b656927, May 2020.

[10]   Hulstaert L., Gradient descent vs. neuroevolution, 20.12.2017., https://towardsdatascience.com/gradient-descent-vs-neuroevolution-f907dace010f, May 2020.

Sažetak

Konektivistička paradigma u umjetnoj inteligenciji danas je najrašireniji pristup. Pojam neuronske mreže uveo je revoluciju u računarsku znanost jer sada možemo riješiti puno kompleksnije probleme na jednostavniji način - treniranjem. Ideja više nije unijeti sve poznato znanje u računalo, već pustiti neuronsku mrežu da sama zaključuje kako se ponašati u još neviđenim situacijama. Dva su načina za to postići: matematički orijentirani način propagacije unazad te neuroevolucija. Zbog svoje jednostavnosti u matematičkom pogledu, nauroevolucija ima značajnu popularnost u svijetu umjetne inteligencije. Međutim, veliki problem predstavlja odabir topologije neuronske mreže koju paniramo trenirati. Iako postoje mnoge dobro poznate topologije za modeliranje određenih problema, postavlja se pitanje bismo li dobili napredak evoluiranjem cijele topologije, a ne samo težine spojeva mreže. Algoritam neuroevolucije s povećavanjem topologija (NEAT) upravo daje odgovor na to pitanje.

Ključne riječi: neuroevolucija, topologija, neuronska mreža, NEAT, evolucijski algoritmi

Summary

The connectivism in artificial intelligence is the most widespread paradigm today. The concept of neural network has revolutionized computer science since it enables solving complex problems in a simpler way – by training. The idea is not to enter all the available knowledge into the computer, rather let the artificial neural network decide how to behave in unprecedented situations. There are two ways to achieve that: a mathematically oriented way of backwards propagation and neuroevolution. Because of its mathematical simplicity, nauroevolution has gained significant popularity in the world of artificial intelligence. However, a big problem is choosing the neural network topology beforehand. Although there are many good already known topologies for modeling specific problems, question arises would we gain significant improvement by evolving network topologies along with connection weights. The neuroevolution with augmenting topologies (NEAT) provides the answer to the posed question.

Keywords: neuroevolution, topology, neural network, NEAT, evolutionary algorithms