Predator-prey simulation using neuroevolution for agent development

Vargić, Anamaria

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka, Faculty of Engineering / Sveučilište u Rijeci, Tehnički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:190:822305

Rights / Prava: Attribution 4.0 International/Imenovanje 4.0 međunarodna

Download date / Datum preuzimanja: **2024-05-19**

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Engineering

UNIVERSITY OF RIJEKA
FACULTY OF ENGINEERING
Graduate University Study of Computing

Graduate thesis

Predator-prey simulation using neuroevolution for agent development

Rijeka, Nov. 2023.

Anamaria Vargić
0069079864

UNIVERSITY OF RIJEKA
FACULTY OF ENGINEERING
Graduate University Study of Computing

Graduate thesis

Predator-prey simulation using neuroevolution for agent development

Mentor: doc.dr.sc. Goran Mauša

Rijeka, Nov. 2023.

Anamaria Vargić
0069079864

Umjesto ove stranice umetnuti zadatak za završni ili diplomski rad

Honesty statement

I declare that I created this thesis independently.

Rijeka, Nov. 2023.

_____

Name Surname

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Technology and real-world occurrences are often intertwined in some way. Technology can be inspired and modeled after these events, or it can be used to inspect them and predict the outcome. A domain that is heavily inspired by the real-world, is artificial intelligence, whose quality and popularity have grown significantly in the past few years. Deep learning is a field of artificial intelligence that aims to help machines make complex decisions and think for themselves. The goal is for machines to incrementally form layers of understanding that they can then use to solve problems. This is achieved by developing artificial neural networks; layered models that try to mimic the structure and learning process of the human brain. Like all models in artificial intelligence, neural networks need to be analyzed and improved in order to produce optimal results. This is usually an iterative process of training and testing the model, then introducing structural improvements or fine-tuning parameters [2].

Another important area of artificial intelligence is evolutionary computation. This domain aims to solve complex optimization problems by iteratively selecting the best solution. It is also heavily based on real-world processes, namely biological evolution. By combining evolutionary computation and deep learning, we get an interesting approach to the simultaneous training and architectural optimization of neural networks called neuroevolution. A sub-method of neuroevolution is the Neuroevolution of Augmenting Topologies (NEAT) algorithm, which uses evolutionary algorithms to evolve the connections and topology of a network [5] [6].

This thesis explains the basics of neural networks, evolutionary computation and demonstrates the application of the NEAT algorithm in developing a predator-prey simulation. Specifically, NEAT is used to evolve the predator and prey neural networks. By the end of the training process, NEAT should output the best performing predator network and the best performing prey network. Performance is measured by a fitness function, which differs for predators and prey. Predators are expected to hunt and eat as many prey as possible, while prey should flee from predators. After training, predators and prey are created from the previously obtained best performing networks. Their behavior is then observed in a test environment, which is visualised.

Chapter 2

Neuroevolution

2.1 Artificial Neural Networks

2.1.1 Structure

As previously mentioned, artificial neural networks (ANNs) are modeled after biological neural networks. ANNs consist of nodes, often referred to as neurons, which can be connected. Each node receives numerical data, processes it, and then outputs the result. The input and output data can be in different forms, depending on the task and domain. For example image processing requires an image input that is represented by a matrix, language processing requires text data, audio processing requires an input signal which can be a sequence of numbers, and so on. The output result can be a numerical value, text, a matrix which can represent a newly generated value, prediction or classification [2]. There are three types of layers:

- **Input layer** - the first layer which receives input data,
- **Output layer** - the last layer which provides the final output of a network,
- **Hidden layers** - a layer in between the input and output layers. [1] Networks often have multiple hidden layers, especially in the field of deep learning where networks have a vast amount of layers [2].

The direction of data flow through these layers depends on the type of neural

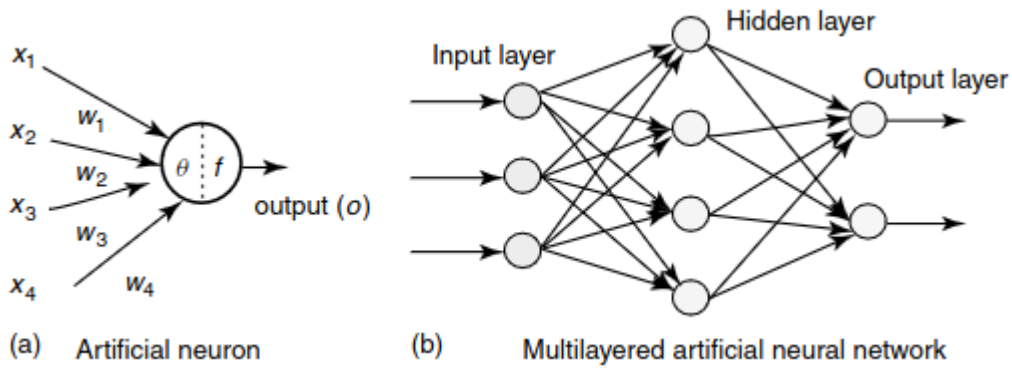

Figure 2.1 structure of a neural network, taken from [1]

network. In feed-forward networks data is passed in a strictly forward manner, from the input layer, through hidden layers and finally to the output layer. In addition to forward flow, recurrent neural networks allow the output of a node to be propagated back to the input of the same node or previous nodes [1].

There are many different types of neural networks, which differ based on their architecture and the types of layers they use.

Convolutional neural networks often use convolutional layers, which are based on the mathematical operation of convolution instead of matrix multiplication used with other types of networks, in order to improve performance. Convolutional layers will use a filter or kernel to perform convolution in combination with the input matrix. Each filter will produce a feature map. This process can be seen on figure 2.2. For this reason, convolutional neural networks are often used in image and pattern processing [2].

Pooling layers are often used in convolutional neural networks in order to reduce the output data of a node by summarizing data in a certain area of the input matrix. An example of this would be max pooling, which takes the local maximum of a section of input data [2].

Gradient descent is used in neural networks in order to minimize the error function, after the forward pass through the network is complete. This is the phase where learning occurs and networks adjust their weights. Gradient descent uses the derivative of a function in order to obtain information on how to minimize it, since

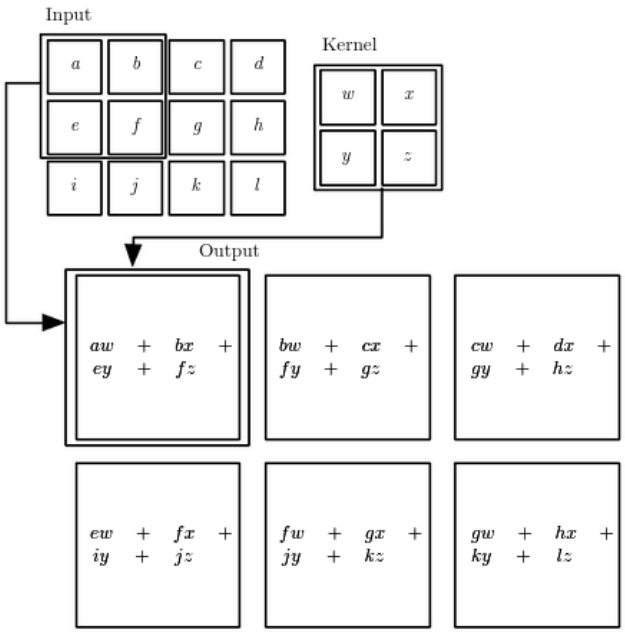

Figure 2.2 Convolution of an input matrix using a kernel, taken from [2]

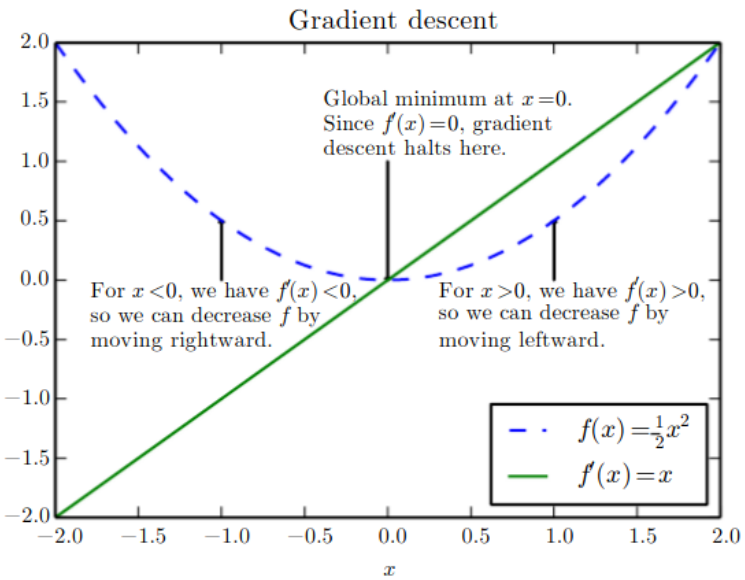

Figure 2.3 Gradient descent, taken from [2]

the derivative gives information about the slope of a function. Given figure 2.3, the network would keep adjusting its parameters until the global minimum at x = 0 is reached. The learning rate plays an important role here, as it defines the size of the step that is taken when finding the minimum. A smaller learning rate will be slower, and a larger learning rate will be faster but might skip the local or global minimum [2].

Long short-term memory (LTSM) networks are a variation of recurrent neural networks. The main goal of LTSM networks is to address the vanishing gradient problem, where the error signal used to update the weights of a neural network vanishes over time. This problem is solved by implementing constant error flow through memory cells and gates, which decide which data is important and should be stored, and which data is irrelevant and should be forgotten [7].

Dropout is often used to prevent overfitting, where a network might learn a certain pattern from the input data which cannot be applied to new data. Dropout fixes this by randomly removing nodes and their connections from a network [8], in other words *thinning* the network which helps it generalize.

2.1.2 Learning

Learning is the process where artificial neural networks learn to make decisions or predictions based on the information and patterns they extract from the input data, where the input data represents information about the problem being solved. The three main types of learning are:

- **Supervised learning** - training data that is fed to the network is labeled, meaning there is a predefined set of possible outputs. The network must then identify the relationship between the input and output data. Classification and regression models use supervised learning. In a classification problem, each sample has an associated class. The model must use the classified training data to learn how to predict which class a test sample belong to. In a regression problem, the goal is to estimate the relationship between variables. Contrary to classification, the output labels are continuous [9],

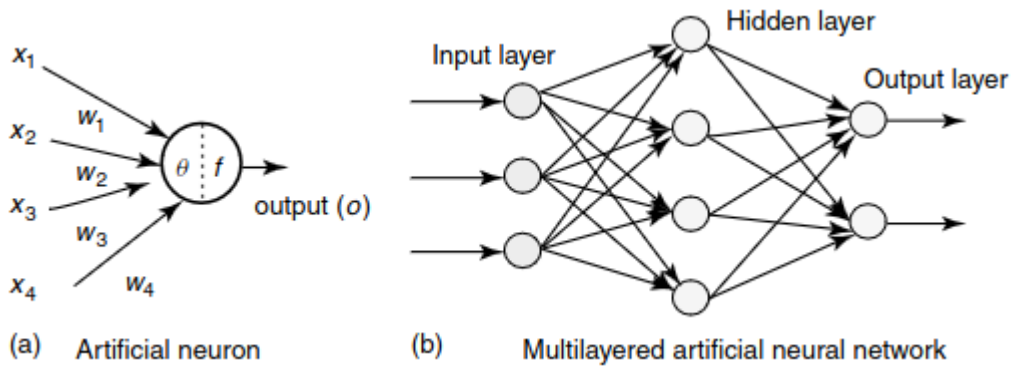

Figure 2.4 structure of a neural network, taken from [1]

- **Unsupervised learning** - training data is unlabeled, meaning there is no previously defined output set. This way the network must identify patterns within the data and form a set pf possible outputs by itself. This type of learning is called clustering, as the model must form clusters of similar data. Types of clustering include hierarchical; where clusters are formed iteratively, from previously defined clusters and partitional; where all the clusters are formed at the same time [10],
- **Reinforcement learning** - with reinforcement learning, models are trained to make the right decision based on a reward system. They must learn to perform the action that will maximize their reward [11]. Reinforcement learning is frequently used for sequential decision making problems. This type of learning does not require the use of differentiable functions, because it does not use techniques like gradient descent, but instead uses methods such as Q-learning [12],
- **Zero-shot learning** - a relatively new approach, this learning method aims to solve the problem where new classes appear in data after training is completed. This is done by giving descriptions of classes, in order for the model to be able to learn the characteristics of a new class by comparing its description to ones it has been trained on [13].

All learning algorithms are based on a similar approach; networks learn by making and modifying connections between nodes. This approach is based on Hebb's rule

which refers to the way that synaptic connections are formed between neurons in biological neural networks during the learning process. Hebb's Rule states that if a certain neuron repeatedly participates in activating another neuron, then their connection should become stronger [14]. Neural networks implement this in the following manner; each neuron receives data through an incoming connection. Each connection has an assigned weight, whose value depends on the importance of that connection. Input values are then transformed into a single value by an aggregation function. This single value is then passed through an activation function. If the value output by the activation function is higher than a certain threshold, the neuron will pass data to the next neuron. Neurons often have a bias; a value that is added to the weighted sum in order to offset the activation function. [1]

There are many different activation functions. Selecting a proper activation function is very important, and depends on the type of problem, data, and desired outputs. This process often implies testing different functions. A few of the most popular activation functions are the rectified linear, sigmoid, hyperbolic tangent (tanh), and softmax activation functions.

The Rectified linear activation function is defined by equation 2.1 and is often used in deep-layered neural networks.

$$f = max(0, x) \tag{2.1}$$

The sigmoid function is defined by equation 2.2 and outputs values in a range of 0 to 1. In other words, if a value is less than 0 it will get mapped to 0, and if it is larger than 1 it will get mapped to 1. For this reason it is often used for binary classification problems, as the output can be considered as the probability that an input belongs the class 1.

$$f = \frac{1}{1 + e^{-x}} \tag{2.2}$$

The softmax activation function will turn a list of output values into a list of probabilities that sum up to the value of 1. Because of this, softmax activation is used for multiclass activation problems. This function is used in the last layer of a neural network, to transform the raw outputs of a neural network to probabilities.

The hyperbolic tangent function is similar to the sigmoid function, but it maps values to a range of -1 to 1. It is defined by equation 2.3 [15] [1].

$$f = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.3}$$

Figure 2.5 shows the rectified linear, sigmoid and hyperbolic tangent activation functions

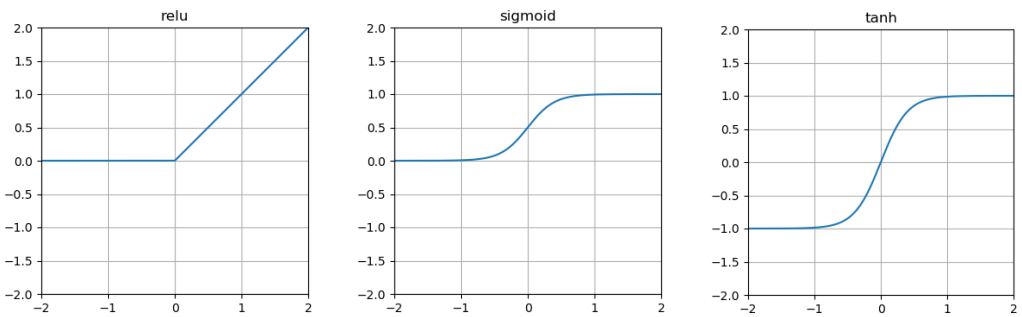

Figure 2.5 rectified linear, sigmoid and hyperbolic tangent activation functions, taken from [3]

An aggregation function in a neural network defines how inputs to a node are processed and turned into a single value. For example, the sum of products aggregation function creates a single value by summing up the products of connection weight and value, for all connections. Other aggregation functions would include sum of all inputs, product of all inputs, taking the min or max value, taking the mean value [1] [16].

2.2 Evolutionary computation

Evolutionary computation consists of many different algorithms inspired by biological processes. These algorithms can be separated into genetic algorithms, genetic programming, evolution strategies and evolutionary programming. Evolutionary algorithms are essentially search algorithms, which find the optimal solution by mimicking biological evolution. In particular, they reenact the term survival of the fittest;

only the best individuals will reproduce and successfully adapt to the environment. In the context of evolutionary algorithms, we consider the environment to be a problem and an individual to be a possible solution. The search starts with a population of possible solutions, and then continuously selects the best candidate until the globally best solution is found or some other stopping condition is met. Candidates are evaluated using a predefined fitness function which describes the quality of that solution for the problem. Optimal candidates can be those that maximize their fitness, but can also be those which minimize it. For example, if we are trying to find a solution with the lowest cost or expenditure [6].

Each individual has a genotype and a phenotype. A genotype is the internal representation and carries information about the characteristics of the individual, while the phenotype is the external representation of the individual.

Evolutionary algorithms often use genetic operators such as selection, mutation and crossover. Selection is used to choose which candidates will be parents for offspring, and also chooses the individuals for the next generation. Popular selection techniques include roulette wheel selection, tournament selection, Boltzmann selection, elitist selection. Mutation is used to modify the parameters of a candidate in order to introduce diversity. Mutation techniques include Bit flip mutation, Scramble mutation, Inversion mutation. Crossover is used in order to produce a new, child candidate from two parent candidates which have been selected [17] [6]. The operator technique being used depends on how the individual is represented; binary string, real-valued vector, permutation, finite-state representations, parse trees, and so on. For example bit strings can use bit flipping for mutation and one-point, two-point and uniform crossover. Real-valued vectors use blend and arithmetic recombination, and uniform and non-uniform mutations. Permutations can use swap, insert, scramble and inversion mutation [17].

Figure 2.6 shows a bit flip mutation on a bit string representation. Here, the value of the fifth bit is changed from 0 to 1 which represents the mutation of a gene. Figure 2.7 shows one point crossover on a bit string representation. Here, the point of crossover is after the first four bits which separates the parent genes into two segments. Then the offspring gets a combination of segments from the parents, for example the first four bits from the first parent and the last two bits from the second

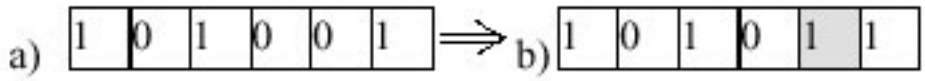

Figure 2.6 Bit flip mutation, taken from [4]

parent. Figure 2.8 shows a grow mutation on the parse tree representation. Here, a child is created by appending a randomly generated subtree onto a randomly selected node of the parent tree.

Genetic algorithms are a subset of evolutionary algorithms which use crossover as the key factor for finding the best solution, therefore the probability of mutation is often quite small with genetic algorithms and is usually set to a percentage between 1 and 5. Genetic algorithms will repeat the process of evaluation for a certain number of generations, selecting parents, crossover and mutation, and then finally creating a new population. The initial population is either a random set of solutions, or a set of possible solutions which are based on domain-specific knowledge. Every individual has the possibility to be selected for crossover, but those with a better fitness value have a higher chance of being selected [6]. Individuals for the next generation can be selected in a number of ways, for example:

- children can completely replace parents, if enough are produced,
- a certain number of children can replace a certain number of parents,
- a number of most fit individuals are passed to the next generation, this is often

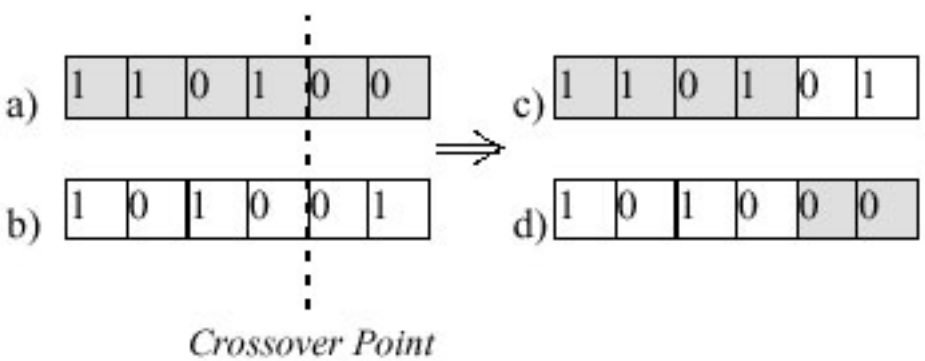

Figure 2.7 One point crossover, taken from [4]

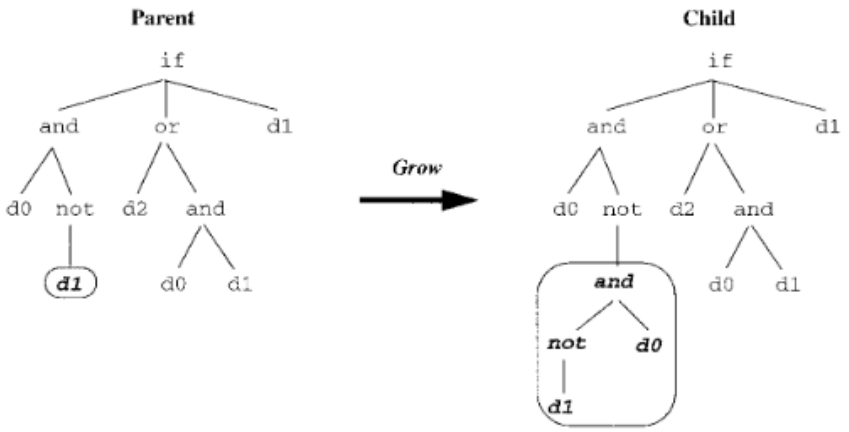

Figure 2.8 Parse tree mutation, take from [2]

referred to as elitism [6].

2.3 NEAT algorithm

Neuroevolution of augmenting topologies aims to optimize a neural network by modifying not only its connections, but also its structure, which is achieved by using genetic algorithms. The main principles NEAT adheres to are:

- allow meaningful crossover to occur between networks that are structured differently,
- prevent newly formed structures from disappearing too early on,
- prevent structures from becoming too complex, without separately tracking their complexity.

Each network has a genome, which consists of two lists; a list of nodes called node genes and a list of connections called connection genes. This genome structure can be seen in figure 2.9. These lists track information such as which nodes are connected, the weights of the connections, which layer the nodes belong to and whether a certain connection is enabled [5], [18].

Figure 2.9 also demonstrates how a given genome translates into a phenotype in

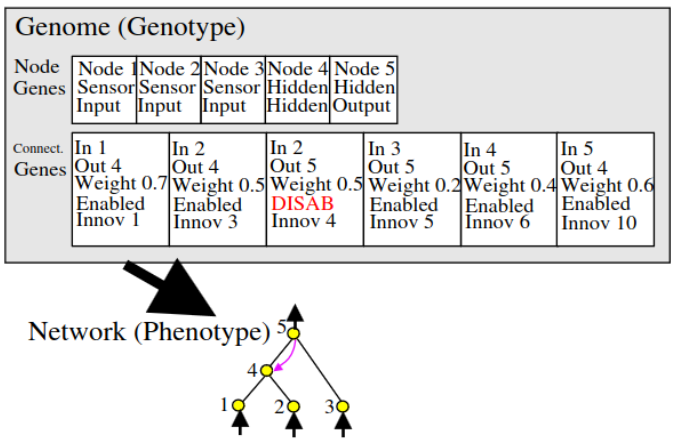

Figure 2.9 genotype of a single network, taken from [5]

genetic algorithms. In this case the phenotype is a neural network with a certain structure built from the genotype, which has information about the connections and weights [6].

Since information about both nodes and connections is stored in the genotype, this means that genetic operations will affect both of these factors. This means that NEAT can both add and remove nodes and connections throughout the evolution process. New nodes are added by replacing a connection, for example if node A and node B were previously connected, a new node C will be added so that A connects to C, which connects to B.

New nodes are numbered in an increasing manner. This number is referred to as the global innovation number and is used in order to track the history of all genes in the system. This is important in order to be able to perform crossover, because NEAT needs to be able to match the genes of parents. Figure 2.10 shows the crossover process: the innovation number is displayed at the top of the gene, and genes that have the same innovation number are considered matching genes. There are also disjoint genes; those that are within the innovation numbers of the other parent but do not have an equivalent, and excess genes; those which are outside the innovation numbers of the other parent. Usually matching genes are randomly inherited and disjoint or excess genes are inherited from the fittest parent. In figure 2.10, the first

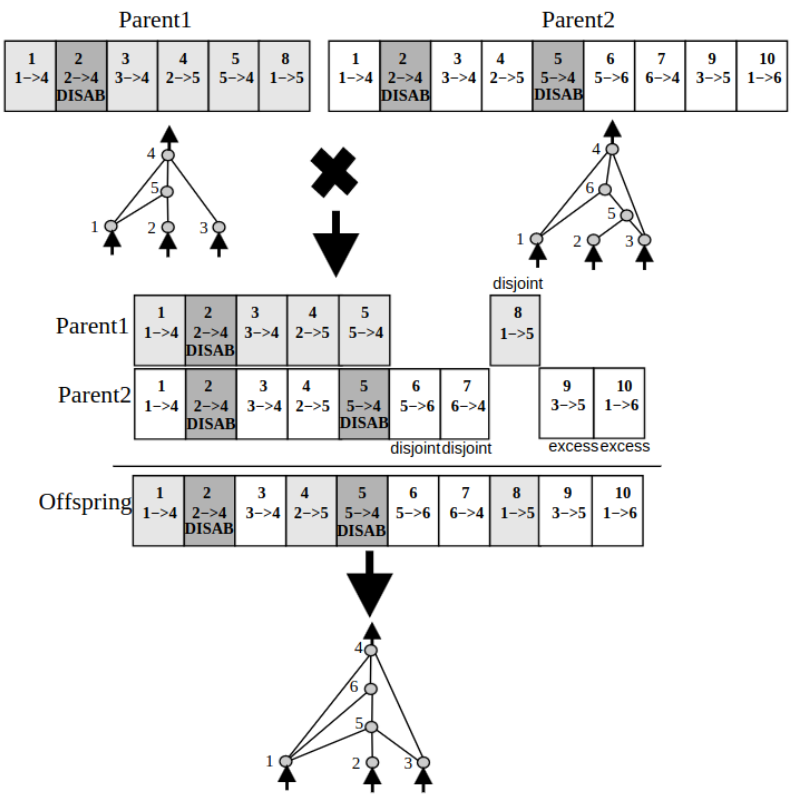

Figure 2.10 Crossover process used by the NEAT algorithm, taken from [5]

five genes are matching genes where genes 1, 2 and 4 are inherited from parent 1 and genes 3 and 5 are inherited from parent 2. In this case the offspring inherits all disjoint genes from parents 1 and 2.

An important part of NEAT is speciation, which protects newly created topologies and promotes diversity. This is done by separating similar topologies into species, so that all members compete with other members of that species instead of the whole population. Similarity of individuals is measured by genomic distance, which can be calculated efficiently by using their number of disjoint and excess genes. Equation 2.4 shows the calculation of genomic distance, where E is the number of excess genes, D is the number of disjoint genes, $\overline{W}$ is the average weight differences of matching genes, N is the size of the larger genome and c_1, c_2, c_3 allow us to modify the importance of E, D and $\overline{W}$.

$$\delta = \frac{(c_1 E)}{N} + \frac{(c_2 D)}{N} + c_3 \overline{W} \tag{2.4}$$

Speciation by putting a genome into the first species where its genomic distance is less than a certain threshold δ_t. If a genome does not belong to any species, then a new one is created.

NEAT also uses explicit fitness sharing which states that members of a species must have a similar fitness to other members so that a species cannot dominate the population. This is achieved by dividing individual fitness by the number of individuals in the species. Adjusted fitness f'_i for each individual i is calculated by equation 2.5, by taking into consideration the genomic distance δ to every other organism j. The sharing function $sh(\delta(i,j))$ is set to 0 when the genomic distance is above the threshold δ_t and set to 1 if it is below it.

Every species can have a certain number of offspring, which is assigned based on the sum of individual adjusted fitnesses. Reproduction is performed by first replacing the lowest performing members. After this, the children of the remaining members replace the rest of the species population.

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{2.5}$$

Unlike other neuroevolution algorithms which start with randomly generated populations, such as the GeNeralized Acquisition of Recurrent Links (GNARL) algorithm, NEAT starts with a population of networks that have no hidden layers. By doing this, it adheres to the principal of minimizing dimensionality by growing the networks only when necessary and removing redundant connections and nodes. This results in better performance compared to other neuroevolution algorithms, which start with an array of topologies with randomly generated structure. [5] [18]

Chapter 3

Methodology

3.1 Task description

The given task is to simulate predator and prey behavior. Predators should be able to hunt prey, while prey should be able to flee predators. Both types of agents must be able to move independently, without human control. In order to achieve this behavior, predators and prey should be controlled by artificial intelligence, namely neural networks. The best possible neural networks for each agent type should be obtained by using the NEAT algorithm. This will also require defining the type of input and output data, selecting the proper type of neural network as well as activation functions and other parameters. The behavior of agents should be visualized once training is done, and occasionally during training to check performance.

3.2 Tools

The following tools were used in solving the given task:

- Python 3.11,
- Pygame 2.1.2 - used for visualizing the simulation,
- NEAT-Python 0.92 - Python implementation of the NEAT algorithm,

- PyGui 1.10.1 - creating the user interface and drawing parts of the simulation,
- NumPy, math - mathematical operations,
- Pyplot - graphing entity performance,
- Graphviz - visualizing neural networks,
- Time - timing utilities for entities,
- Random - generating random numbers.

The training and simulation are run on Ubuntu 22.04. with an Intel i3 processor that has x86 64 architecture and a speed of 2.00GHz.

3.3 NEAT-Python

This library is a Python implementation of the NEAT algorithm. This library implements the NEAT algorithm as described in the previous chapter, with the exception that disjoint and excess genes are not differentiated. NEAT-Python calculates the output of a node using equation 3.1, where O is the output value of a node, *inputs* are the input values, and *activation*, *bias*, *response*, *aggregation* are the activation function, bias, response and aggregation function of a node.

$$O = activation(bias + (response \times aggregation(inputs))) \tag{3.1}$$

The most important steps in using this module are the configuration file and defining a proper fitness function. The following list explains each parameter in the neat configuration file, and the values set for entity networks.

- **fitness criterion** - this parameter determines whether max, min or mean fitness is desired,
- **fitness threshold** - once this threshold is met, evolution will stop if fitness is set as the stop condition,
- **no fitness termination** - if set to true then evolution should stop when a certain number of generations is reached, or when the fitness threshold is

reached. For entities this is set to true

- **pop size** - the number of individuals each generation will consist of,
- **reset on extinction** - if set to true a new random population will be created if all individuals are removed because of stagnation, or if an exception will be thrown,
- **activation default** - the activation function assigned to new nodes,
- **activation mutate rate** - the probability that a random activation function from the activation options list will be assigned to a node,
- **activation options** - the possible activation functions that can be assigned through mutation,
- **aggregation default** - the aggregation function assigned to new nodes,
- **aggregation mutate rate** - the probability that a random aggregation function from the aggregation options list will be assigned to a node,
- **aggregation options** - the possible aggregation functions that can be assigned through mutation,
- **bias init mean** - the mean of the distribution that can be used to assign bias to new nodes,
- **bias init stdev** - the standard deviation of the distribution that can be used to assign bias to new nodes,
- **bias max value** - the maximum possible bias value,
- **bias min value** - the minimum possible bias value,
- **bias mutate power** - the standard deviation of the zero-centered distribution which can be used to select a bias mutation value,
- **bias mutate rate** - the probability that the bias of a node will be changed by mutation,
- **bias replace rate** - the probability of the bias being completely replaced by a new value,
- **compatibility disjoint coefficient** - defines the importance of disjoint and

excess genes in calculating genomic distance by the equation 2.4,

- **compatibility weight coefficient** - defines the importance of weight, bias, response multiplier differences in calculating the genomic distance. It is also the value added to activation function, aggregation function and enabled status differences,
- **conn add prob** - the probability that a new connection between existing nodes will be added by mutation,
- **conn delete prob** - the probability that an existing connection between two nodes will be removed by mutation,
- **enabled default** - defines whether newly created connections are enabled,
- **enabled mutate rate** - the possibility that mutation will change the enabled status of a connection,
- **feed forward** - defines whether a network is strictly feed forward, or if it is allowed to have recurrent connections. This is set to true
- **initial connection** - defines how the initial nodes of a network will be connected,
- **node add prob** - the probability that a new node will be added by mutation,
- **node delete prob** - the probability that a new node will be removed by mutation,
- **num hidden** - the number of hidden nodes in the initial genome,
- **num inputs** - the number of input nodes in the initial genome,
- **num outputs** - the number of output nodes in the initial genome,
- **response init mean** - the mean of the distribution that can be used to select response multiplier values for new nodes,
- **response init stdev** - the standard deviation of the distribution that can be used to select multiplier values for new nodes,
- **response max value** - the maximum possible response value,
- **response min value** - the minimum possible response value,

- **response mutate power** - the standard deviation of the zero-centered distribution from which a response mutation is chosen,
- **response mutate rate** - the probability that the response value of a node will be changed by mutation,
- **response replace rate** - the probability that mutation will completely replace the response of a node by a random value,
- **weight init mean** - the mean of the distribution used to select weight values for new nodes,
- **weight init stdev** - the standard deviation of the distribution used to select weight values for new nodes,
- **weight max value** - the maximum possible value of a weight,
- **weight min value** - the minimum possible value of a weight,
- **weight mutate power** - the standard deviation of the zero-centered distribution used to select a weight mutation,
- **weight mutate rate** - the probability that the weight of a node will be changed by mutation,
- **weight replace rate** - the probability that mutation will completely replace the weight of a node by a random value,
- **compatibility threshold** - individuals that have genomic distance less that this value are considered to be part of the same species,
- **species fitness func** - the function used to determine species fitness,
- **max stagnation** - the maximum number of generations that a species can be stagnant, after which it is removed. This value is set to 15
- **species elitism** - the number of species that will not be removed by stagnation,
- **elitism** - the number of most fit individuals that will be passed on to the next generation,
- **survival threshold** - the percent of each species allowed to reproduce each generation.

This library comes with many different examples and offers implementations of recurrent, feed-forward, continuous-time recurrent and spiking neural networks. It offers many different activation functions that include tanh, ReLu, sigmoid, hat, identity, clamped and more [3].

In order to use NEAT-Python, a fitness function must be defined which takes in two parameters; genomes and the path to the configuration file. Inside the fitness function the entities and networks are created from the genomes and then evaluated. One run through this fitness function is one generation. NEAT-Python can be configured to run for a certain number of generations, or until a fitness threshold is reached.

The fitness function is run on a population by calling population.run() and passing a reference to the fitness function and the number of generations. If termination by fitness threshold is desired, then the number of generations can be omitted. Upon termination, the most fit individual is returned. NEAT-python should return the most fit individual ever seen, in all generations. During development it was discovered that this is not the case and it instead returned the most fit individual of the last generation, so tracking of the globally optimal individual was implemented manually.

Figure 3.1 shows an example of statistics reporting for each generation, which includes the best and mean fitness with its standard deviation and the mean genomic distance and standard deviation. Information about species and stagnation, and generation run time are shown.

```
 ****** Running generation 40 ******

Population's average fitness: -9.00000 stdev: 14.96663
Best fitness: 10.00000 - size: (4, 6) - species 1 - id 112
Average adjusted fitness: 0.525
Mean genetic distance 1.537, standard deviation 0.221
Population of 5 members in 1 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1   40     5     10.0    0.525    13
Total extinctions: 0
Generation time: 22.097 sec (20.702 average)
```

Figure 3.1 NEAT-Python statistics output for a generation

Table 3.1 *First Part of the NEAT-Python configuration parameters*

fitness criterion	max
fitness threshold	For entities this value is set to 1000, even though it is not used
no fitness termination	True
pop size	For predators this parameter is set to 5 and for prey this number is set to 20. These numbers were chosen by trial and error, and also to keep training time from being too slow
reset on extinction	False
activation default	For both entity types the sigmoid activation function is used. This function is used because networks have 2 outputs, so the sigmoid function will give the probability that the entity should turn left or right because it maps values to a range of 0 to 1.
activation mutate rate	0.0
activation options	sigmoid
aggregation default	sum
aggregation mutate rate	0.0
aggregation options	sum
bias init mean	0.0
bias init stdev	1.0
bias max value	30.0
bias min value	-30.0
bias mutate power	0.5
bias mutate rate	0.7
bias replace rate	0.1
compatibility disjoint coefficient	1.0
compatibility weight coefficient	0.5
conn add prob	0.5
conn delete prob	0.5
enabled default	True
enabled mutate rate	0.01
feed forward	True

Table 3.2 *Second Part of the NEAT-Python configuration parameters*

initial connection	full - all nodes will be connected with each other, but there will be no recurring connections because the network is feed forward
node add prob	0.2
node delete prob	0.2
num hidden	0
num inputs	2
num outputs	2
response init mean	1.0
response init stdev	0.0
response max value	30.0
response min value	-30.0
response mutate power	0.0
response mutate rate	0.0
response replace rate	0.0
weight init mean	0.0
weight init stdev	1.0
weight max value	30
weight min value	-30
weight mutate power	0.5
weight mutate rate	0.8
weight replace rate	0.1
compatibility threshold	3.0
species fitness func	max
max stagnation	15
species elitism	2
elitism	2
survival threshold	0.2

3.4 Entities

There are two types of entities; predators and prey. Predators and prey have the same initial neural network structure; two input nodes and two output nodes. The neural network type of choice is a feed-forward network. This network type was chosen since feed-forward networks are often used for classification problems, and it was the most suitable network type that NEAT-Python offers. The inputs are the angle and distance to the closest entity of the opposite class in their field of view, and the outputs are rotate left or right by 5 degrees. This initial neural network can be seen in figure 3.2. The entity field of view can be seen in sections 3.4.1 and 3.4.2.

After the entity has rotated, it will automatically move forward. Entities move

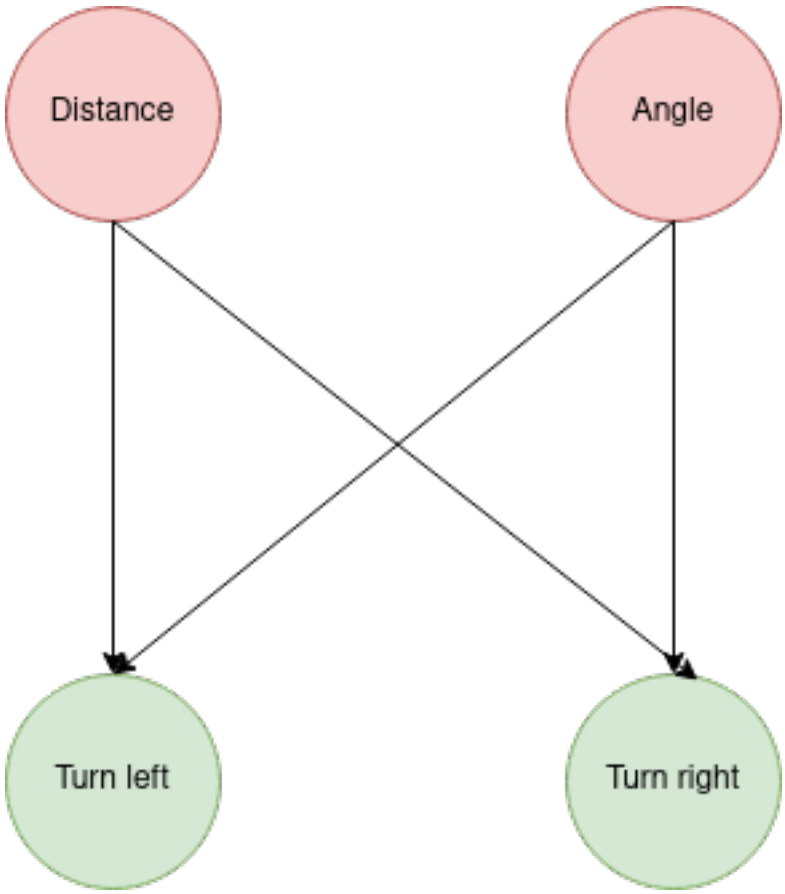

Figure 3.2 Initial neural network for both entity types

forward by using equations 3.2 and 3.3 to calculate their new coordinates x_n and y_n based on their current coordinates x_o, y_o, angle α and velocity v. The velocity is constant and is equal to 1 pixel per frame.

$$x_n = x_o + v * \cos(\alpha \times \pi/180) \tag{3.2}$$

$$y_n = y_o + v * \sin(\alpha \times \pi/180) \tag{3.3}$$

Both entity types have a vision angle and a vision radius. These elements are used to calculate a field of view which is used to detect the proximity of entities of the opposite class. This is done in the following manner: in each frame for each entity it is first checked if there is an entity inside the vision radius by using equation 3.4, where x_1 and y_1 are the coordinates of the current entity, x_2 and y_2 are the coordinates of the target entity potentially in the current entity's vision radius, V.

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 < V^2 \tag{3.4}$$

If this condition is true, the angle to the target entity is calculated with the equation 3.5

$$\arctan(\frac{y_2 - y_1}{x_2 - x_1}) \tag{3.5}$$

The vision bounds are then calculated by equations 3.6 and 3.7, where A is the angle the entity is currently facing, V_a is the vision angle, B_l and B_r are the left and right vision bounds. If the target is within the vision bounds, the distance to the target is calculated using dist function of the Math module. Since the closest entity should be detected, if the distance to the target is the minimum of all targets, the distance angle and difference are stored. During angle calculations, *mod* 360 is used to maintain the angles in a range from 0 to 360.

$$B_l = A - V_a \mod 360 \tag{3.6}$$

$$B_r = A + V_a \mod 360 \tag{3.7}$$

3.4.1 Predators

Predators are represented by red boids. They have a hunger value that ranges from 0 to 4, and increases approximately every 2 seconds. When their hunger reaches level 4, predators die and are removed from the simulation or training. They can reduce their hunger by eating prey. Predators have a vision radius of 250 and a vision angle of 60, so they have a long but focused field of view. They must eat prey to survive, and for each consumed prey, their fitness grows by 5. Predators are punished by 30 fitness if they die. Figure 3.3 shows a predator and its field of vision.

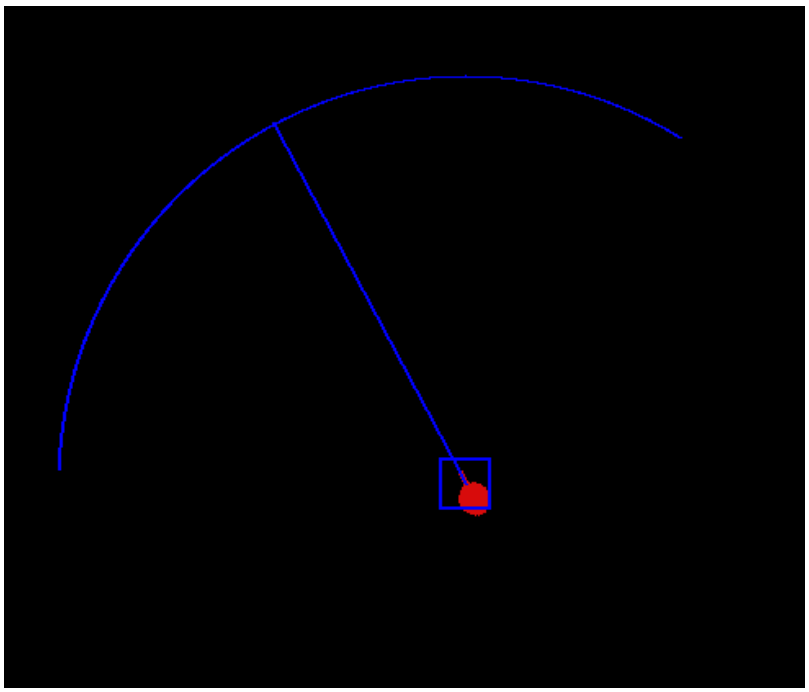

Figure 3.3 predator field of vision

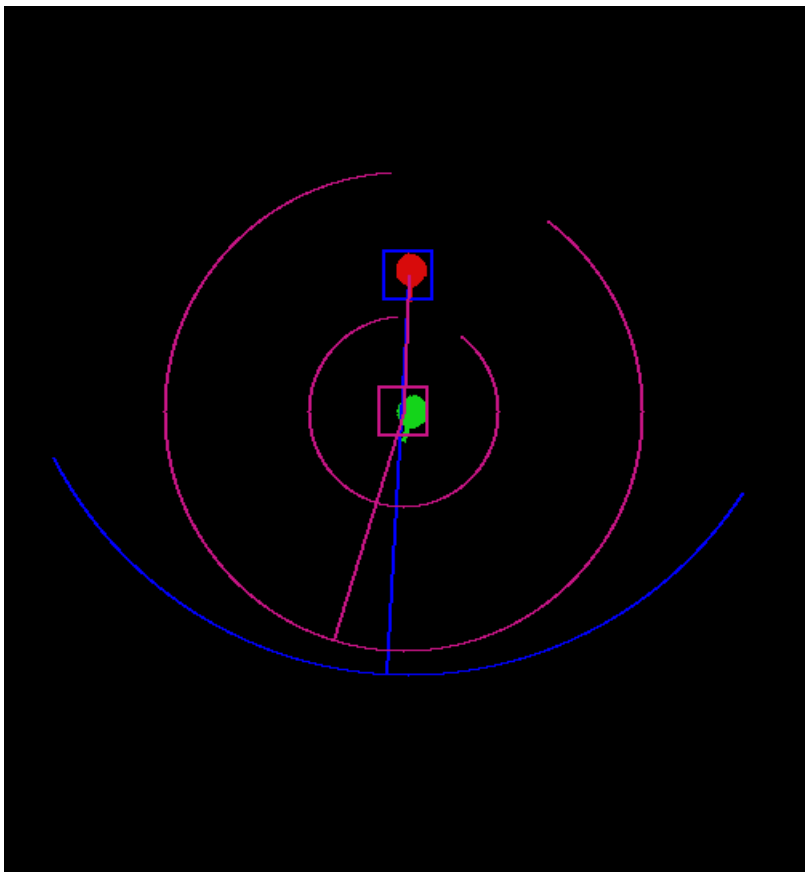

Figure 3.4 predator and prey pursuit visualisation

3.4.2 prey

prey are represented by green boids. They do not have hunger, their only goal is to survive. In addition to having a field of view, prey also have a danger zone. This zone lets the prey know that a predator has come too close. prey have a vision angle of 160, a vision radius of 150 and a danger radius of 60, so prey can see in a wider radius around themselves but cannot see very far. Figure 3.4 shows a predator in pursuit of a prey. The predator is in the prey field of view (the outer arc in the figure), but outside the prey danger zone (the inner arc).

prey must learn to flee from predators. They are rewarded 0.01 fitness for each frame in which they survive. They are also rewarded 1 fitness if the predators are in their field of vision but not in their danger zone. If a predator is in their danger zone,

they get punished by a certain amount of fitness that is calculated by equation 3.8, where d is the current distance to the predator. By doing this, the prey recognize that they will get rewarded by running away from predators, but will be punished if they allow the predators to get too close. If prey were not punished when a predator was in the danger zone, they would learn to seek out predators and be eaten. prey are punished by 50 fitness if they die. This fitness mechanism was chosen so that it would not be more profitable for prey to simply idle.

$$y = -0.25d + 25 \tag{3.8}$$

3.4.3 User interface

The interface is quite simple and consists of a button with the label Toggle lines. Clicking on this button will enable drawing of the vision lines. These lines will draw the entities field of view, a line pointing in the current direction they are moving, and a line to the closest entity if one exists in the field of view. This will also draw the prey danger zone, and highlight the entities Pygame *rect* object.

Clicking on an entity will show an information box just under the toggle button, which can be seen in figures 3.6 and 3.5. For predators the displayed information will include current hunger and the number of prey eaten up until now, marked as 'Food eaten'. For prey the displayed information will be the time that they have survived, displayed in seconds.

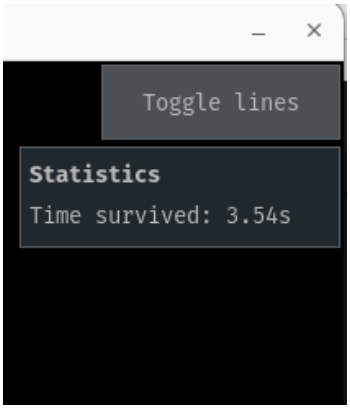

Figure 3.5 user interface prey information

Figure 3.6 user interface predator information

3.5 Training and testing

Inside the evaluation function entities perform the following actions in each frame:

1. update class specific parameters - predators increase their hunger if needed, prey increase their survival time and receive a reward if they are still alive.
2. calculate inputs to the neural network - if there is one or multiple targets inside the entities field of view, the distance and angle to the closest target are saved as inputs. If there are no targets in the entities field of view, the input distance is the entity types vision range + 1 and, for predators the input angle is their vision angle + 5, while the input angle for prey is 180.
3. have their neural network process the inputs - a neural network is activated by calling the activate() function on a neural network and passing in the inputs. This function will return the outputs of the network which are then saved to a variable
4. process outputs - entities then move base on the outputs. Outputs are processed in the following manner; if the first output is greater than 0.5 the entity will rotate left and then move forward. If the second output is greater than 0.5 the entity will rotate right and then move forward
5. correct position - if entities reach the edge of the window, their coordinates are modified so that they 'pass over' to the opposite edge. This is done so entities do not disappear off of the screen
6. update other class parameters - predators will then check for eaten food, which is done by checking collision between predators and prey. Since each entity has an assigned Pygame rect object which defines the area and location of a certain entity, the rect.colliderect() function is used. If there is collision, the predator will decrease its hunger by 1, increase its food count and fitness. The eaten prey will then be flagged for death so it can later be removed from the population
7. remove flagged entities - remove predators or prey, that have been marked dead, from the population

Predators are first trained on prey that does not move. During predator training, the generation is terminated if all predators die of hunger. When the best predator network is found, it is stored and then used to create a trained population of predators that will be used to train prey. During prey training, a generation is terminated once all predators or prey have died.

Once prey have been trained by running from predators, the best prey network is stored. Then, two populations are created in a test environment from the best predator and prey in order to check if they behave as they should. The test environment includes the visualisation of the simulation, so after all entities are updated the user interface and simulation are drawn. In the test environment entities have a chance to reproduce every 10 seconds. For prey this probability is equal to 30% and for predators it is equal to 50% if their hunger is above 2. This feature is implemented simply to keep the population from going extinct, in order to keep the simulation running. Upon inspection, it can be observed that predators successfully hunt prey, while prey flee.

Another approach to training that was attempted, included the following steps:

1. train predators to hunt static prey
2. train prey to flee from moving predators
3. train predators to hunt moving prey
4. repeat steps 2 and 3, until 500 generations are run

This was implemented by using NEAT-Python's checkpointing system that allows users to save a population's state, and then later restore it and continue training. Two separate *checkpointer* objects were created, one saved and restored predator population checkpoints and the other saved and restored prey checkpoints.

This approach was proven to be unsuccessful. The reason for this is that during predator training with moving predators, predators cannot successfully catch prey because their velocity is constant.

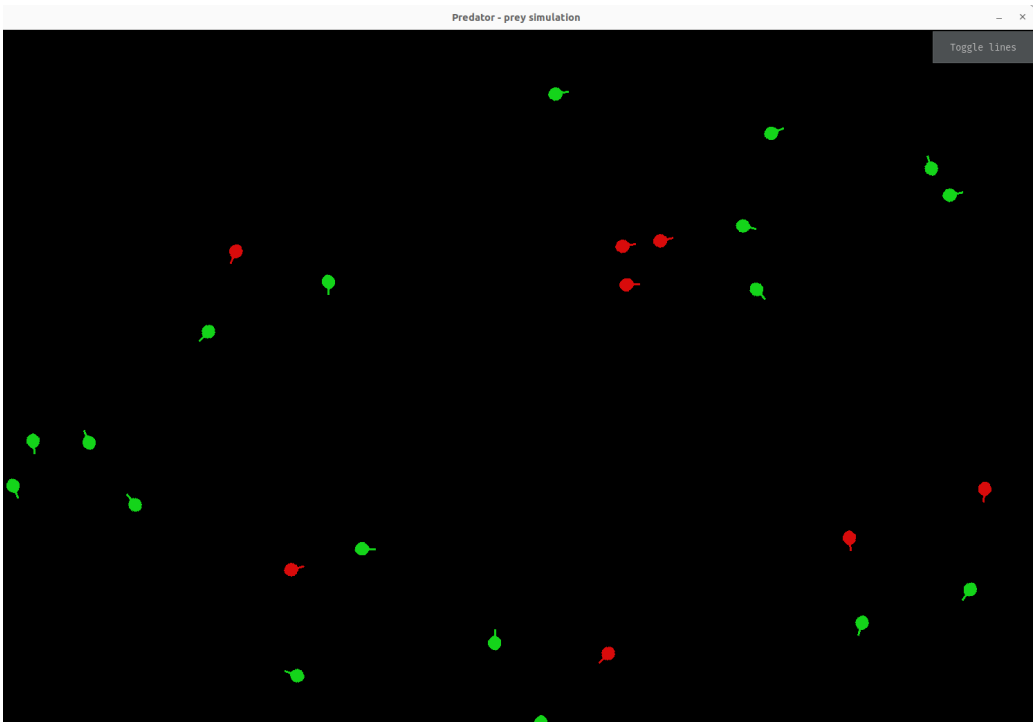

Figure 3.7 simulation visualisation

Chapter 4

Results

Figures 4.1 and 4.2 shows the fitness of the best performing prey and predators in each generation during training, for 200 generations. Predator fitness appears to oscillate rather than show a globally upward trend, This is due to the fact that predators learn to hunt prey, but if there is no nearby prey they do not explore, so they do not eat and gain fitness. This means that the fitness of the highest performing predator

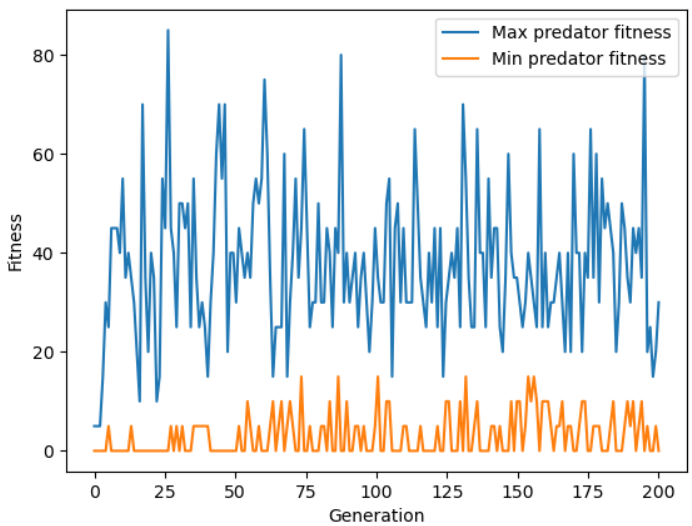

Figure 4.1 minimum and maximum predator fitness throughout 200 generations

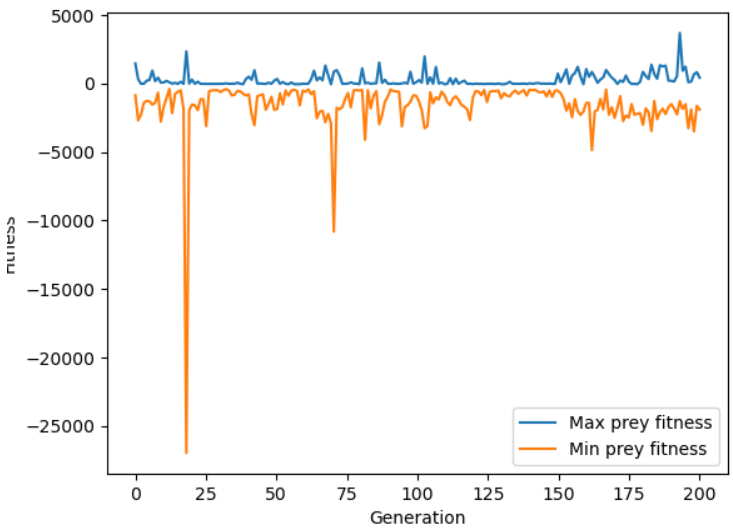

Figure 4.2 minimum and maximum prey fitness throughout 200 generations

depends on where the predator is spawned and how much prey there is around them when they spawn.

This predator behavior affects prey fitness as well. During prey training, they get a higher reward for fleeing than for idling and simply staying alive. This means that the fitness of the highest performing prey depends on whether they are being chased, which of course depends on the predators ability to discover and chase prey. If a prey is spawned far from any predators, it will have a lesser fitness even though it managed to survive. The resulting behavior is that prey do not move if there are no nearby predators. This behavior makes sense, because if prey were to move around randomly they would increase their chances of running into a predator and being eaten or being too close to a predator.

This observation would mean that NEAT-Python performs quite fast which can be seen by enabling visualisation for the training period. It can be observed that the entities do not know how to move, hunt, or flee in the first few generations. Figure 4.1 shows the maximum and minimum individual fitness for each predator generation. The most important segment of the graph are the first 15 generations, where the predators are first learning to move and eat prey.

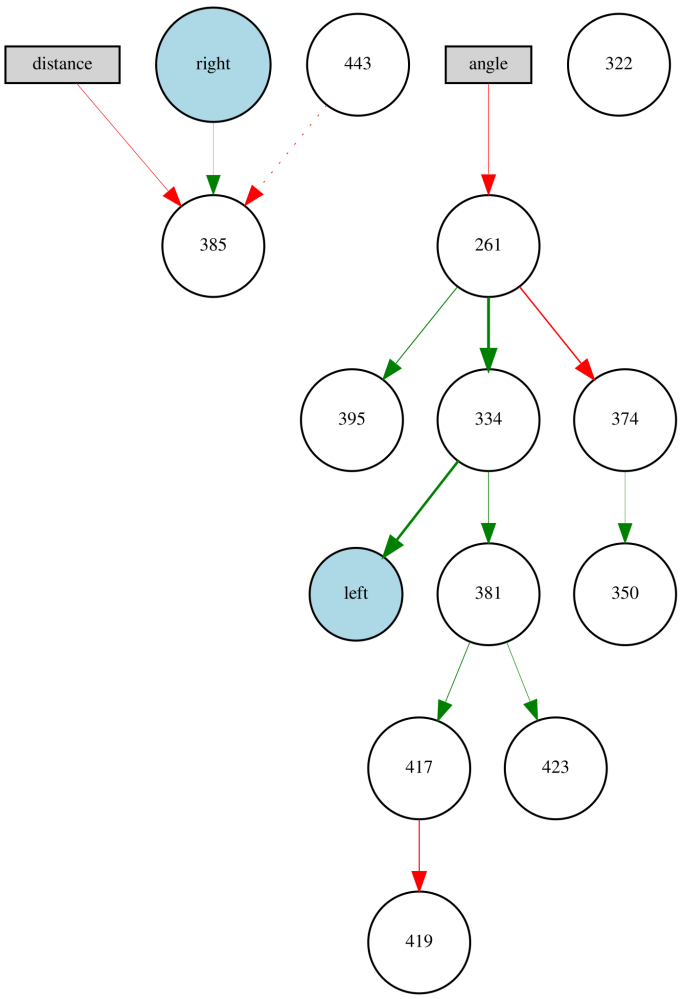

Figure 4.3 Predator neural network after 2000 generations

Figure 4.3 and 4.4 show the neural network for the top-performing predator and prey after 2000 generations. The innovation numbers on the nodes show that, during the development process, many nodes and connections have been added and removed. This can be confirmed by comparing this figure to figures 4.5 and 4.6 which show predator and prey networks after only 1 generation. The red lines represent negative weights, and the green lines represent positive weights. Solid lines represent enabled connections, dotted lines represent disabled connections. The figures show a number of disconnected or *dangling* nodes, including the *distance* input node, *right* output node, nodes 443 and 322. A possible cause for this could be that NEAT simply

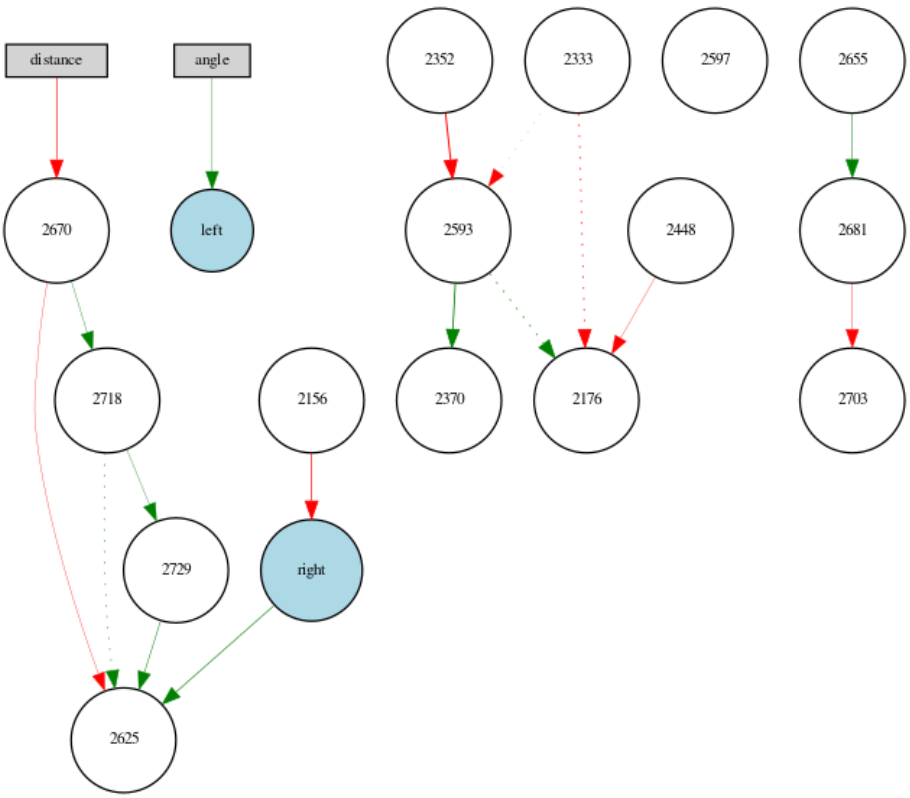

Figure 4.4 Prey neural network after 2000 generations

identified these nodes as unnecessary, since the entities display correct behavior with these networks. Another possible cause could be the NEAT-Python module itself, because unlike the original NEAT algorithm, mutations can remove connections which may lead to unusual structures. This problem was also found on the NEAT-Python github issue page, where other users found that feed-forward networks had an increasing number of dangling nodes and unusual structure with a higher number of generations. Another unusual observation is the connections from the output nodes to other nodes, which could simply be a by-product of NEAT trying to evolve the topology. The most important nodes in this structure would be *angle*, 261, 334, and *left*. Other nodes such as 381, 374, 395 which are enabled but are not connected to any output are useless, but could also be a result of the network simply trying to identify patterns in data.

As previously mentioned, the fitness function, activation function and output

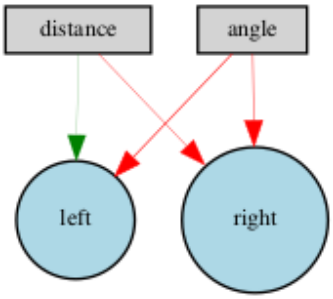

Figure 4.5 initial predator neural network

interpretation are the most important factors for successfully applying NEAT. This was learned through trial and error; trying different activation and fitness functions, and processing outputs differently.

Fitness functions are important because they encourage the network to perform the correct action. Fitness functions should be as simple as possible and cover all possible scenarios, otherwise neural networks can 'exploit' the rewarding system in an unplanned way. An example of this was during the first approach towards creating the predator-prey simulation; networks had 4 outputs; left, right, forward and backwards. They were rewarded for changing their direction of movement and as a result, they would rapidly move forward then backward, essentially staying in the same place and get a very large fitness, even though this was not the desired behavior.

Outputs and inputs should be logically connected in some way, in order for the neural network to be able to interpret the inputs and correlate them to the correct

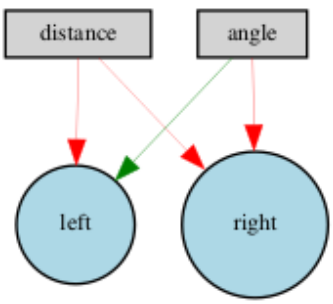

Figure 4.6 initial prey neural network

```
****** Running generation 802 ******

Population's average fitness: -6.00000 stdev: 21.30728
Best fitness: 35.00000 - size: (4, 3) - species 1 - id 1089
Average adjusted fitness: 0.354
Mean genetic distance 1.662, standard deviation 0.984
Population of 5 members in 2 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1  802     3     35.0    0.542   743
     2  680     2    -10.0    0.167   333
Total extinctions: 0
Generation time: 10.008 sec (10.008 average)
```

Figure 4.7 Speciation output for predators

output. In the first approach, networks had 16 inputs and the 4 previously mentioned outputs. The inputs represented sensor rays of the entity, which essentially tracked the distance to an entity if it was touching the ray. This approach did not work because the neural network could not comprehend what these 16 values had to do with the 4 output directions. A better option would have been to pass 4 inputs that would represent the distance to an entity in each of those directions.

The average running time for predator training is 10.06 seconds, while for prey it is 11.21 seconds. It is important to note that disabling visualisation significantly improves training speed. Predator training with visualisation lasts on average 15.12 seconds, while prey training lasts on average 13 seconds.

Figure 4.7 4.8 shows that predators and preys were separated into 2 species during training. Even thought they showed stagnation for more than 15 generations, they were not removed because of the *species elitism* parameter in the configuration file.

By running the test simulation, it can be observed that predators successfully chase prey, while prey flee. This means that well performing neural networks were obtained. Predators and prey exhibit so called pursuit and evasion behaviors, types of steering behaviors. This terminology is often used in video game development and animation. As discussed in Steering behaviors for autonomous characters [19], Pursuit behavior is a steering behavior that aims to move the entity towards another moving target, for example the player or another entity. Evasion is a steering behavior

```
****** Running generation 122 ******

Population's average fitness: -675.29156 stdev: 1382.90079
Best fitness: 426.20250 - size: (9, 5) - species 1 - id 2167
Average adjusted fitness: 0.833
Mean genetic distance 1.947, standard deviation 0.837
Population of 20 members in 2 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1  122    11    426.2    0.883    27
     2   16     9    230.6    0.783    11
Total extinctions: 0
Generation time: 12.014 sec (12.012 average)
```

Figure 4.8 Speciation output for prey

opposite of pursuit. It aims to move the entity away from a moving target such as the player or another entity [19].

Further steps to improve current behavior would include rewarding predators for exploring. This could be achieved by splitting the map into segments and punishing predators if they lingered in a segment for an extended period of time. If they were to be rewarded for changing segments, they might try to exploit this mechanism by repeatedly traversing a segment border in order to maximize their reward. Another step would be to introduce movement energy, so entities would have to pay attention to the energy cost of movement. This would most likely require adding inputs to the neural network such as current energy and cost of each movement. Entities could also be improved by enabling them to see other members of their class, which would perhaps result in more group-like behavior.

Chapter 5

Conclusion

This thesis explains the basics of neural networks, evolutionary computation and the NEAT algorithm. These concepts are then applied by creating a predator-prey simulation in which agents move independently and mimic the behavior of real-life predators and prey. The simulation was created in Python, using the NEAT-Python module for evolving the agent neural networks, and Pygame in order to visualize the simulation.

It is observed that the activation function, fitness functions and output interpretation are the key factors in successfully obtaining a well performing neural network using NEAT-Python. The results show that the networks start to behave as expected after just 15 generations. Entity behavior and networks show satisfactory behavior after 1000 generations. Network structures have been observed to be more complex the more generations are run. This case study showed that NEAT-Python causes dangling nodes with no apparent function to be left in the network. Despite this, the entities still learned the desired behavior.

Fitness growth should be observed in the context of the problem. During predator training the fitness of predators depends on where they have been spawned on the map, so their fitness will only grow if they recognize potential rewards in their proximity. Despite this behavior during training, predators still learn fairly quickly that their main objective is to hunt prey.

Outputs and inputs must be reasonably correlated, otherwise neural networks

will not be able to learn how to map the inputs to the proper output. It was also concluded that training without visualisation was 5 seconds faster for predators, and 1.8 seconds faster for prey.

Simulations like the one shown in this thesis can be used for educational purposes such as demonstrating animal behavior, simulating the spreading of viruses, observing ecosystem behavior, video game development, autonomous robots and so on. They contribute to the popularization of artificial intelligence and science as a whole, by providing an interactive and efficient way to test ideas and observe results in any domain.

Bibliography

[1] A. Abraham, "Artificial neural networks," *Handbook of measuring system design*, 2005.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[3] Neat-python documentation. , Electronic , https://neat-python.readthedocs.io/en/latest/index.html

[4] Genetic algorithm examples. , Electronic , https://neo.lcc.uma.es/cEA-web/GA.htm

[5] K. O. Stanley and R. Miikkulainen, "Evolving neural networks throughaugmenting topologies," *The MIT Press Journals*, 2002.

[6] J. H. Holland, "Genetic algorithms," *SCIENTIFIC AMERICAN*, 1992.

[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, 1997.

[8] N. Srivastavan, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdino, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, 2014.

[9] R. Sathya and A. Abraham, "Comparison of supervised and unsupervised learning algorithms for pattern classification," *International Journal of Advanced Research in Artificial Intelligence*, 2013.

[10] T. S. Madhulatha, "An overview on clustering methods," *IOSR Journal of Engineering*, 2012.

[11] A. Talwar and Y. Kumar, "Machine learning:an artificial intelligencemethodology," *International Journal Of Engineering And Computer Science*, 2013.

[12] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2018.

[13] B. Romera-Paredes and P. Torr, "An embarrassingly simple approach to zero-shot learning," *International conference on machine learning*, 2015.

[14] D. O. Hebb, *The organization of behavior; a neuropsychological theory*, 1949.

[15] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neuralnetworks," *International Journal of Engineering Applied Sciences and Technology*, 2020.

[16] K. Teknomo. , Electronic , https://people.revoledu.com/kardi/tutorial/NeuralNetwork/AggregationFunctions.html

[17] T. Back, D. B. Fogel, and Z. Michalewicz, *Evolutionary Computation 1 Basic Algorithms and Operators*. INSTITUTE OF PHYSICS PUBLISHING.

[18] K. O. Stanley and R. Miikkulainen, "Efficient evolution of neural network topologies," *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, 2002.

[19] C. W. Reynolds, "Steering behaviors for autonomous characters," *Game developers conference*, 1999.

Abstract

This thesis shows the creation of a predator-prey simulation, where entities move autonomously using neural networks. The NEAT algorithm is used in order to obtain the best performing neural networks for each entity type. Predators are first trained using static prey, after which these predators are used to train prey to flee. The best neural networks obtained by training are then put into a visualised test environment to observe their behavior. Pursuit and evasion behavior is successfully achieved, and it is observed that the most important role in successfully applying NEAT are the activation function, fitness function and output interpretation in regards to the given inputs.

Keywords **— Neuroevolution of augmenting topologies, neural network, genetic algorithm**

Appendix A

Source code

A.1 Predator configuration File

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 10000000
no_fitness_termination = True
pop_size              = 5
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = sigmoid
activation_mutate_rate  = 0.0
activation_options      = sigmoid

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum
```

```
# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full

# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_hidden              = 0
num_inputs              = 2
num_outputs             = 2
```

```
# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 15
species_elitism      = 2

[DefaultReproduction]
elitism            = 2
survival_threshold = 0.2
```

A.2 Prey configuration File

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 10000000
no_fitness_termination = True
pop_size              = 20
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = sigmoid
activation_mutate_rate  = 0.0
activation_options      = sigmoid

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5
```

```
# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full

# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_hidden              = 0
num_inputs              = 2
num_outputs             = 2

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
```

```
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 15
species_elitism      = 2

[DefaultReproduction]
elitism            = 2
survival_threshold = 0.2
```