UNIVERSIDAD DE INVESTIGACIÓN DE TECNOLOGÍA EXPERIMENTAL YACHAY

Escuela de Ciencias Matemáticas y Computacionales

Deterministically Crowded - NeuroEvolution of Augmenting Topologies

Trabajo de integración curricular presentado como requisito para la obtención del título de Ingeniero en Tecnologías de la Información

Autor:

Arellano Tamayo William Rodolfo

Tutor:

Ph.D - Martinez Campos Cédric

Co-Tutor:

Ph.D - Ortega-Zamorano Francisco

Urcuquí, Agosto 2019

Urcuquí, 22 de agosto de 2019

SECRETARÍA GENERAL
(Vicerrectorado Académico/Cancillería)
ESCUELA DE CIENCIAS MATEMÁTICAS Y COMPUTACIONALES
CARRERA DE TECNOLOGÍAS DE LA INFORMACIÓN
ACTA DE DEFENSA No. UITEY-ITE-2019-00004-AD

En la ciudad de San Miguel de Urcuquí, Provincia de Imbabura, a los 22 días del mes de agosto de 2019, a las 15:00 horas, en el Aula AI-102 de la Universidad de Investigación de Tecnología Experimental Yachay y ante el Tribunal Calificador, integrado por los docentes:

Presidente Tribunal de Defensa	Dr. CHANG TORTOLERO, OSCAR GUILLERMO , Ph.D.
Miembro No Tutor	Dr. CUENCA LUCERO, FREDY ENRIQUE , Ph.D.
Tutor	Dr. MARTINEZ CAMPOS, CEDRIC , Ph.D.

Se presenta el(la) señor(ita) estudiante **ARELLANO TAMAYO, WILLIAM RODOLFO**, con cédula de identidad No. 1717179673 de la **ESCUELA DE CIENCIAS MATEMÁTICAS Y COMPUTACIONALES**, de la Carrera de **TECNOLOGÍAS DE LA INFORMACIÓN**, aprobada por el Consejo de Educación Superior (CES), mediante Resolución RPC-SO-43-No.496-2014, con el objeto de rendir la sustentación de su trabajo de titulación denominado: **Deterministically Crowded-Neuro Evolution of Augmenting Topologies**, previa a la obtención del título de **INGENIERO/A EN TECNOLOGÍAS DE LA INFORMACIÓN**.

El citado trabajo de titulación, fue debidamente aprobado por el(los) docente(s):

Tutor	Dr. MARTINEZ CAMPOS, CEDRIC , Ph.D.

Y recibió las observaciones de los otros miembros del Tribunal Calificador, las mismas que han sido incorporadas por el(la) estudiante.

Previamente cumplidos los requisitos legales y reglamentarios, el trabajo de titulación fue sustentado por el(la) estudiante y examinado por los miembros del Tribunal Calificador. Escuchada la sustentación del trabajo de titulación, que integró la exposición de el(la) estudiante sobre el contenido de la misma y las preguntas formuladas por los miembros del Tribunal, se califica la sustentación del trabajo de titulación con las siguientes calificaciones:

Tipo	Docente	Calificación
Presidente Tribunal De Defensa	Dr. CHANG TORTOLERO, OSCAR GUILLERMO , Ph.D.	10,0
Tutor	Dr. MARTINEZ CAMPOS, CEDRIC , Ph.D.	10,0
Miembro Tribunal De Defensa	Dr. CUENCA LUCERO, FREDY ENRIQUE , Ph.D.	9,5

Lo que da un promedio de: 9.8 (Nueve punto Ocho), sobre 10 (diez), equivalente a: APROBADO

Para constancia de lo actuado, firman los miembros del Tribunal Calificador, el/la estudiante y el/la secretario ad-hoc.

ARELLANO TAMAYO, WILLIAM RODOLFO
Estudiante

Dr. CHANG TORTOLERO, OSCAR GUILLERMO , Ph.D.
Presidente Tribunal de Defensa

Dr. MARTINEZ CAMPOS, CEDRIC , Ph.D.
Tutor

Dr. CUENCA LUCERO, FREDY ENRIQUE , Ph.D.
Miembro No Tutor

TORRES MONTALVÁN, TATIANA BEATRIZ
Secretario Ad-hoc

AUTORÍA

Yo, **WILLIAM RODOLFO ARELLANO TAMAYO**, con cédula de identidad 1717179673, declaró que las ideas, juicios, valoraciones, interpretaciones, consultas bibliográficas, definiciones y conceptualizaciones expuestas en el presente trabajo; así cómo, los procedimientos y herramientas utilizadas en la investigación, son de absoluta responsabilidad de el/la autora (a) del trabajo de integración curricular. Así mismo, me acojo a los reglamentos internos de la Universidad de Investigación de Tecnología Experimental Yachay.

Urcuquí, Agosto 2019

WILLIAM RODOLFO ARELLANO TAMAYO
CI: 1717179673

AUTORIZACIÓN DE PUBLICACIÓN

Yo, **WILLIAM RODOLFO ARELLANO TAMAYO,** con cédula de identidad 1717179673, cedo a la Universidad de Tecnología Experimental Yachay, los derechos de publicación de la presente obra, sin que deba haber un reconocimiento económico por este concepto. Declaro además que el texto del presente trabajo de titulación no podrá ser cedido a ninguna empresa editorial para su publicación u otros fines, sin contar previamente con la autorización escrita de la Universidad.

Asimismo, autorizo a la Universidad que realice la digitalización y publicación de este trabajo de integración curricular en el repositorio virtual, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación Superior

Urcuquí, Agosto 2019.

WILLIAM RODOLFO ARELLANO TAMAYO
CI:1717179673

Acknowledgements

Many thanks to the following people:

- My advisor, Cédric Martinez Campos, Ph.D, for the many discussions we had on the subject, his invaluable edition and his willingness to direct this work.
- My co-advisor, Francisco Ortega Zamorano, Ph.D, for providing direction on the subject of this work, his knowledge of artificial neural nerworks and the help he has provided through all the process.
- My family, for their support and love that helped me to finish this work.

Resumen

Las redes artificiales neuronales (RNN) son modelos computacionales que aproximan la actividad cerebral para resolver un problema, sin embargo, el rendimiento de estas redes depende en la correcta configuración de su estructura y de una multitud de parámetros, que en muchos casos son fijados mediante un proceso de prueba y error. La presente tesis propone un algoritmo que automatiza el proceso de afinación de RNNs. El algoritmo propuesto es una variación del algoritmo "Neuro Evolution of Augmenting Topologies" (NEAT), el cual utiliza conceptos biológicos como evolución, mutación y especiación, para mejorar de manera consistente la adaptación de una población inicial de RNNs generadas de manera aleatoria. El algoritmo propuesto difiere al utilizar una estrategia diferente para la especiación de la población, se pasa de utilizar "fitness sharing" a utilizar "deterministic crowding", éste algoritmo propuesto se lo nombra como "Deterministically Crowded - Neuro Evolution of Augmenting Topologies" (DC-NEAT). A partir de la comparación de NEAT y DC-NEAT a la hora de resolver el problema de balanceo de una vara con configuración completa, observamos que DC-NEAT obtiene redes mas simples en estructura que NEAT, pero requiere un número mayor de generaciones.

Palabras Claves- redes neuronales artificiales, algoritmos genéticos, deterministic crowding, NEAT.

Abstract

Artificial Neural Networks (ANN) are computational models that approximates brain activity. However, their performance depend on the correct settings of their initial structures and large amount of parameters, which used to be manually tuned in a tiresome trial-and-error process. This thesis proposes an algorithm that automates the time-consuming task of fine-tuning ANNs. The proposed algorithm is a variation of Neuro Evolution of Augmenting Topologies (NEAT), which consists on improving the fitness of a population of initially random ANNs, using biological concepts, such as evolution, mutation and speciation. Our contribution to NEAT is the used of a different speciation strategy, deterministic crowding, to conserve and promote diversity in the population. The proposed algorithm is named as Deterministically Crowded - Neuro Evolution of Augmenting Topologies (DC-NEAT). After comparing DC-NEAT with NEAT for the full pole balancing problem, we observed that DC-NEAT produces simpler networks but requires more generations to obtain them.

Keywords- artificial neural networks, genetic algorithms, deterministic crowding, NEAT.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Artificial neural networks are computational models which are widely used in diverse applications. These networks are composed of units called neurons which transmit information between them utilizing connections. However, the design of these artificial neural networks is complex, the following questions must be answered: How many neurons? and How these neurons are connected?. Hornik in [1] determined that failure in the application of artificial neural networks can be attributed to the improper design of these networks. In general, design of artificial neural networks is done based on a trial-and-error process, where intuition are used to optimize the architecture of networks, the neurons and their connections. Some in their pursuit of automating the design of artificial neural networks have married the ideas of genetic algorithms with the design of networks. In particular, genetic algorithms are used to evolve the architecture of artificial neural networks. A population of networks is continuously changed to produce better performing networks, in the context of the problem that these networks are used on, by applying Darwinism and mutation. The population is pushed to better performance by the genetic algorithm.

Neuro Evolution of Augmenting Topologies (NEAT) is one method that uses genetic algorithms to evolve the architecture of artificial neural networks from minimal structures. This method uses *explicit fitness sharing* for the conservation and promotion of diversity in the population of networks (individuals). Explicit fitness sharing partitions the population in *niches* or *species* with the idea that highly likely networks should compete against each other and not with networks from other niches. Nonetheless, other niching methods exist like *deterministic crowding* which are simpler to implement and has less time complexity.

In this work, a new method for evolving artificial neural networks is introduced, Deterministically Crowded - Neuro Evolution of Augmenting Topologies (DC-NEAT). This new method is based on Neuro Evolution of Augmenting Topologies, indeed it borrows its genome encoding (how networks are encoded as genes) and the mutation operators of NEAT. Nevertheless, DC-NEAT differs from NEAT as it uses deterministic

crowding instead of explicit fitness sharing. The purpose is to create a version of NEAT that has a simpler design and is easier to implement.

The rest of this thesis is structured as follows. Chapter 2 offers an overview of genetic algorithms, here we introduce the canonical genetic algorithm (Section 2.1) which helps understand the mechanics involved in genetic algorithms. Section 2.2 shows the importance of *diversity* and niching methods, specially when applied to multi-modal function optimization. Finally, in Section 2.3 we examine the importance of niching methods to promote and conserve diversity in populations, additionally two niching methods are explored *fitness sharing* (2.3.1) and *deterministic crowding* (2.3.2).

In Chapter 3, a brief overview of artificial neural networks is given. Section 3.1 defines the concepts related to artificial neural networks. In particular, definitions for neuron, layer, connection and learning are given. In Section 3.2, the architecture of artificial neural network is defined. And, two network architectures of interest are exposed *feed-forward artificial neural networks* (3.2.1) and *recurrent artificial neural networks* (3.2.2). Lastly, Section 3.3 introduces the *backpropagation* learning method, which does weight adaptation of artificial neural networks.

Chapter 4 explains everything related to the method Neuro Evolution of Augmenting Topologies for evolving artificial neural networks. First, in Section 4.1 the algorithm of NEAT is described paying attention to the parameters that determine the behavior of NEAT. Section 4.2 describes the genetic encoding of networks in NEAT; how neurons and connections are encoded to genes. In Section 4.3, the crossover procedure of NEAT is explained. Crossover is the process where two parents share the genetic material to produce a new individual. Section 4.4 defines the similarity measurement used in NEAT, which plays an important role in the partitioning of the population. NEAT's explicit fitness sharing is detailed in Section 4.5 this method is based on fitness sharing. The chapter ends with Section 4.6 which introduces all the mutation operators available in NEAT.

Deterministically Crowded - Neuro Evolution of Augmenting Topologies is defined in Chapter 5, as noted before DC-NEAT is constructed based on NEAT. The chapter starts with Section 5.1 and Section 5.2 which describe the problems found in NEAT from a theoretical perspective, in specific with the NEAT's explicit fitness sharing and NEAT's similarity measurement. The algorithm of DC-NEAT is fully described in Section 5.3.

Chapter 6 presents the case of study problem used to test DC-NEAT, the one pole balancing problem. It describes the two possible configurations of the pole balancing problem. And, how DC-NEAT is used to solve these two configurations.

In Chapter 7, the results of using DC-NEAT to solve the pole balancing problem can be viewed. Here, we can see how the DC-NEAT performs in a dynamic problem. This chapter highlights the differences in performance of DC-NEAT when used to solve the two configurations of the pole balancing problem. Furthermore, a comparison with

NEAT is done.

Discussion of the results obtained is done in Chapter 8, here the behavior of DC-NEAT noted in the previous chapter is explained. The last chapter, Chapter 9, summarizes this thesis and shows the possibilities of future work with DC-NEAT.

Chapter 2

Genetic Algorithms

In his book *Genetics Algorithms in Search, Optimization and Machine Learning*, Goldberg defines *genetic algorithms* (GA) as "search algortihms based on the mechanics of natural selection and natural genetics"[2, p.1]. In general, a GA drives a *population* of artificial structures towards optimality through the use of fitness based reproduction and genetic operators (*crossover* and *mutation*). These artificial structures are known as *genomes*, and they represent the parameter space of the search problem. The genomes are composed of information units known as *genes*. These genes are moved, created, and eliminated in the population by genetic operators. In contrast to other search algorithms, like *hill climbing*, a GA works by: searching in a encoded parameter space, parallelizing the search over multiple points, using fitness information (pay-off information), and using probabilistic transition rules [2, p.7].

2.1 Canonical Genetic Algorithm

The *canonical genetic algorithm* [3] or *simple genetic algorithm* [2, 4] works with a constant population of N bit-strings. These bit-strings or *genomes* are of length l. Thus, an individual genome belongs to $G := \{0,1\}^l$. And, the population belongs to the search space $P := G^N$. The algorithm transforms at each generation (iteration) a population p_i to a new population p_{i+1}, with i representing the current generation. This transformation of the population is achieved through the application of *fitness proportionate selection*, *crossover*, and *mutation*. The algorithm runs iteratively until a stopping criterion is met, or a maximum generation M is reached. Alg. 1 shows the canonical genetic algorithm with the maximum number of generations as the stopping criterion.

The fitness function, $f : G \to \mathbb{R}$, assigns a real value to the individual's genome. This value represents the individual's goodness as a solution of the problem. The canonical genetic algorithm uses proportionate fitness selection to drive the population towards better solutions. When using the canonical genetic algorithm for function optimization (maximization), the objective function is used as the fitness function. However, when

Algorithm 1: Canonical genetic algorithm with maximum number of generations M as stopping criterion.

Input: Fitness function f, M, p_c, p_m
Output: P

```
Initialize population P
i = 0
repeat
    foreach g ∈ P do
        f(g)
    Select N individuals to reproduce
    Randomly pair the selected individuals
    Crossover the pairs taking into account the crossover probability p_c
    Mutate over the population in accordance to the mutation probability p_m
    i = i + 1
until i ≥ M
```

doing function minimization the additive inverse of the objective function is used as the fitness function.

Fitness proportiate selection adds Darwinism to the algorithm. Selecting the fitter individuals from the population which undergo reproduction and mutation ensures that the population overall fitness increases with the generations. In the case of the canonical genetic algorithm, N individuals are selected from the population. However, depending on the selection mechanism, replacement might be allowed. In other words, individuals can be selected more than once. *Roulette-wheel selection* (RWS) [5] and *Stochastic Universal Sampling* (SUS) [6] are examples of fitness proportionate selection mechanisms. RWS emulates a roulette wheel with a slot for each individual in the population. The size of each slot is proportionate to the corresponding individual's fitness. SUS is akin to RWS, where the only difference is that it uses N pointers instead of one to mark the "winner" on the roulette wheel. These pointers are equally spaced. Thus, only a single spin is necessary to select the N individuals from the population. In RWS, individuals with fitness greater than the population's average fitness tend to be selected more often. Therefore, SUS is used when it is desirable to avoid highly fit individuals to outclass the rest of the population.

The crossover operator, $C : G^2 \to G^2$, creates two new individuals *offsprings* from each selected pair of *parent genomes*. An example of a crossover operator is *single point crossover*. In single point crossover, a random point in the interval $[0, l-1]$ is choosen and denoted as the *crossover point*. Then, using the crossover point and the parent genomes crossover is done as shown in Fig. 2.1. The crossover point divides the parents'

genomes in two fragments. The resulting fragments are then recombined to form two new individuals. Thus, crossover allows the exchange of genes between individuals inside the population. The exchange is important as it creates new combinations of genes. These new combinations help create fitter individuals. The crossover probability p_c determines if the pair goes through crossover or not. Pairs that do not experience crossover undergo mutation directly.

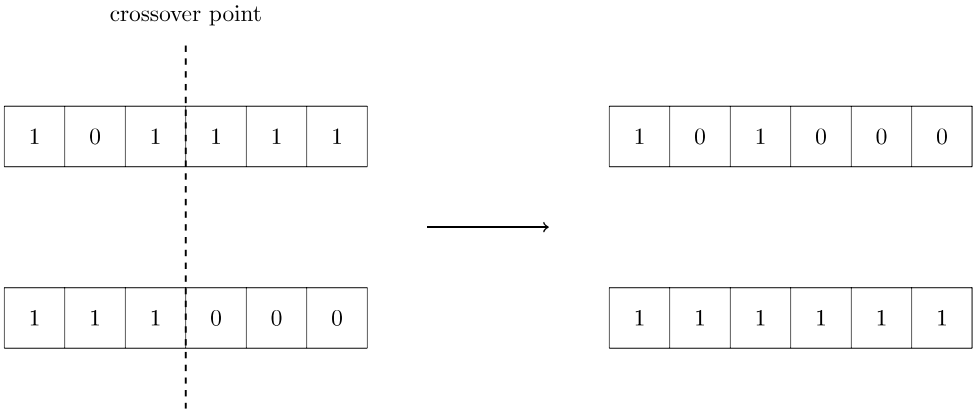


Figure 2.1: Example of single point crossover with $l = 6$ and crossover point at 3.

The mutation operator, $M : G \to G$, randomly mutates each bit of the individual's genome in accordance to the mutation probability. Mutation adds new genes to the population, increasing the search coverage. Nonetheless, increasing too much the mutation probability p_m adds noise to the search. If $p_m = 0.5$, the canonical genetic algorithm is transformed into a random search.

2.2 Diversity and Multi-modal function optimization

The selection mechanism of the canonical genetic algorithm favors the preservation of solutions, highly fit genomes. However, when working with multiple solutions of identical fitness, the population converges to a single solution. Solutions and subsolutions are lost in the process mainly due to: *selection pressure*, *selection noise*, and *operator disruption* [4]. Selection pressure is the consequence of the selection mechanism, as selection removes the lower fitness solutions from the population. Selection noise arises from the variance of the selection mechanism in a finite population, and the competition of similarly fit subsolutions. Selection noise ultimately causes good solutions to be removed from the population. Operator disruption is the direct consequence of crossover and mutation. These two genetic operators can inadvertently destroy good solutions from the population. Therefore, to preserve or increase diversity in a population it is common to reduce: selection pressure, selection noise, or operator disruption.

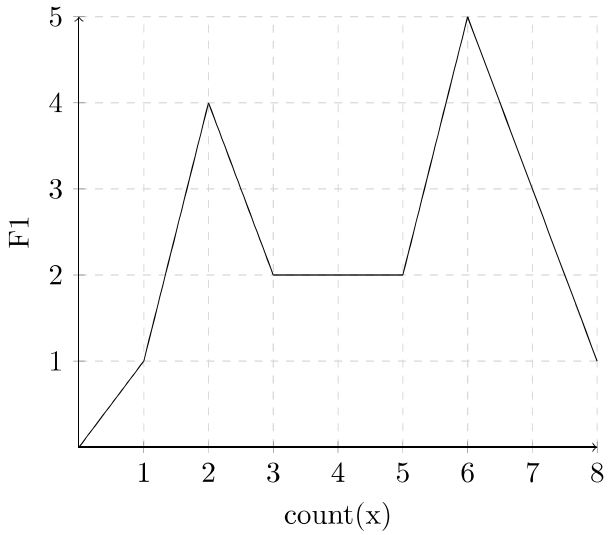


Figure 2.2: F1, unitation function, vs count(x), the number of ones in the bit string x, where x is an eight-bit string.

To understand the importance of diversity in the population, The canonical genetic algorithm was used to maximize the objective function $F1$ (shown in Fig. 2.2). $F1$ is a unitation[1] function over the eight-bit string x, and it has multiple optimal values. For example, the strings 11111100, 00111111, and 01111110 have the maximum fitness value of 5. The canonical genetic algorithm with RWS was run with population size $N = 16$, crossover probability $p_c = 0.9$, mutation probability $p_m = 0.01$, and maximum number of generations of 20. Fig. 2.3 shows the evolution of the population average fitness and best fitness through the generations. The average fitness increments until reaching a value of 4. The best fitness of the population reaches the maximum value of 5 at generation 9 and 10. But, the population loses these highly fit individuals and never recovers them. RWS alone fails to keep the highly fit individuals.

[1] An unitation function counts the number of 1s in a bit-string.

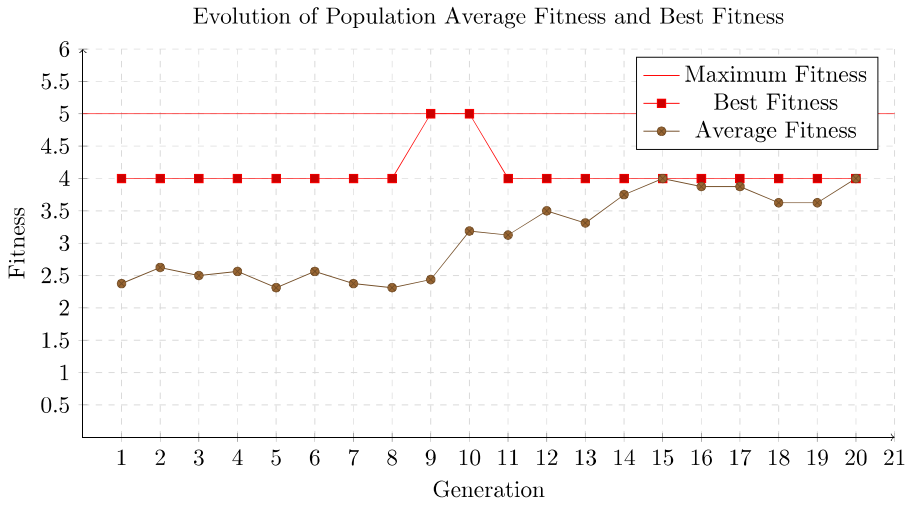


Figure 2.3: Population average fitness and best fitness evolution of the canonical genetic algorithm through the generations. The algorithm was run with: RWS as selection mechanism, $N = 16$, $p_c = 0.9$, $p_m = 0.01$, and $M = 20$. Maximum fitness value is shown in red.

The algorithm fails to search in both solution niches, see Fig. 2.2 at values $count(x) = 2$ and $count(x) = 6$. The algorithm does not preserve nor increase the diversity in the population. Thus, the search centers in only one niche; in this case the neighborhood centered at $count(x) = 2$. Diversity helps the canonical genetic algorithm to do parallel searches by maintaining multiple niches in the population. Therefore, genetic diversity is of great importance when using the canonical genetic algorithm to solve multi-modal function optimization [7].

2.3 Niching methods

Mahfoud defines in [4] *niching methods* as "techniques that promote the formation and maintenance of stable subpopulations in the genetic algorithm". These subpopulations or *niches* can be used to obtain either one final solution, either multiple final solutions. A successful niching method must be able to maintain fitness varying solutions and to do so for a long amount of time (generations) [4]. Niching methods allow genetic algorithms to do parallel search of multiple peaks by reducing the effect of early convergence caused by the selection mechanism of the canonical genetic algorithm [8]. In this section, we explore two niching methods: *fitness sharing* and *deterministic crowding*.

2.3.1 Fitness Sharing

Fitness sharing or simply *sharing* [9] reduces the fitness value of individuals in the population by sharing their fitness with the rest of the population. In specific, fitness

sharing divides the population in niches, and the individual shares its fitness with the individual's niche. This sharing reduces the payoff of highly dense populated regions. The niches are built using a *similarity measure* or "distance" that indicates how similar two individuals are. This measure can be either *genotypic* (based on the genetic search space), or *phenotypic* (based on the problem search space). Fitness sharing reduces the redundant search in the population, as it discourages large niches composed of similar individuals. And, it encourages the formation of new niches composed of highly dissimilar individuals.

Algorithm 2: Calculation of population shared fitnesses in fitness sharing.

Input: Fitness function (f), population array (P), population size N

Output: $\{f'_j : 1 \leq j \leq N\}$

1 Calculate the fitness f_i of each individual in P
2 $j = 0$
3 **repeat**
4 Compute the niche count m_j of each individual in P
5 Compute the shared fitness f'_j of each individual in P
6 **until** $j \geq N$

Alg. 2 shows the process of fitness sharing. The process starts by calculating the shared value of each individual with all individuals in the population, this value can be calculated using the *sharing function*,

$$sh(j,k) = \begin{cases} 1 - (\frac{d(j,k)}{\sigma_s})^{\alpha} & \text{if } d(j,k) < \sigma_s, \\ 0 & \text{otherwise.} \end{cases}, \tag{2.1}$$

where j and k represent individuals in the population, $d(j,k)$ is the similarity distance which determines the shape of the niche, the constant α determines the shape of the sharing function, and the *similarity threshold*, σ_s, is the size of the niche. The value of the similarity threshold, σ_s, must be positive, $\sigma_s > 0$, and depends on the shape of the search space landscape and the similarity distance, d. A typical value of α is 1, the sharing function then has a triangular shape (see Fig. 2.4). The sharing function, $sh(j,k)$, has values in $[0,1]$; the sharing function measures a sense of species' belonging of two individuals in the population. A value of one indicates that the individuals j and k belong to the same species, whereas a value of zero indicates that the individuals belong two different species.

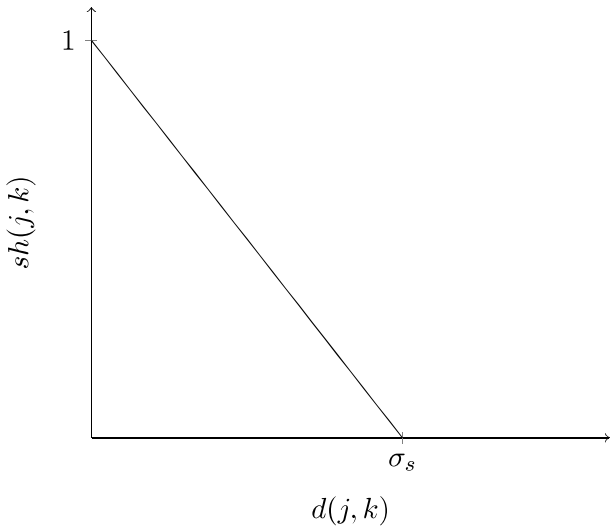


Figure 2.4: Sharing function with $\alpha = 1$.

The shared values of the individuals are used to calculate each individual niche count, for an individual j the *niche count* is

$$m_j = \sum_{k=1}^{N} sh(j,k), \tag{2.2}$$

where N is the total population size. The niche count has values in $[1, N]$; an individual can "belong" to more than one niche. Finally, the shared fitness of the individual is

$$f'_j = \frac{f_j}{m_j}. \tag{2.3}$$

After calculating the shared fitness f'_j for all individuals in the population, the selection mechanism uses these shared fitnesses instead of the previously calculated fitnesses. Any selection mechanism can be used with fitness sharing. However, the selection mechanism influences the stability of the genetic algorithm; stochastic universal selection is popular [4].

2.3.2 Deterministic Crowding

Mahfoud [4] proposed *deterministic crowding* as an improvement to De Jong's crowding method [10]; deterministic crowding can maintain several peaks of a multi-modal function. Alg. 3 shows how to apply deterministic crowding in the canonical genetic algorithm. Deterministic crowding compares parents with offsprings in two possible tournament configurations. On the one hand, parent 1 versus offspring 1, and parent 2 versus offspring 2. On the other hand, parent 1 versus offspring 2, and parent 2 versus offspring 1. The tournament configuration is decided by the minization of the parent-offspring distance. The victor of the tournament is chosen by better fitness, so that the parent is either kept in the population or replaced by an offspring.

Algorithm 3: Deterministic Crowding

Input: Population size N, Population array P, Fitness function f, similarity metric d, Maximum Generation M

Output: P

```
t = 0
repeat
    i = 0
    repeat
        Select randomly without replacement p1 and p2.
        Cross p1 and p2, obtaining c1 and c2.
        Mutate c1 and c2, yielding c1' and c2'.
        if (d(p1, c1') + d(p2, c2')) ≤ (d(p1, c2') + d(p2, c1')) then
            if f(c1') > f(p1) then
                Replace p1 with c1' in P
            if f(c2') > f(p2) then
                Replace p2 with c2' in P
        else
            if f(c2') > f(p1) then
                Replace p1 with c2' in P
            if f(c1') > f(p2) then
                Replace p2 with c1' in P
        i++
    until i ≥ N/2
    t++
until t ≥ M
```

Chapter 3

Artificial Neural Networks

Artificial Neural Networks (ANN) are computational approximations of mental activity in the brain, based on the hypothesis that mental activity consists primarily of electrochemical impulses of brain cells called neurons [11, p. 727]. ANNs can also be understood as computational frameworks that emulate the neurons networks inside the human brain which perform massively parallel computations [12]. This model consists of computational units known as *neurons* or *nodes*, interconnected by weighted links. These computational units can be divided as: *input* or *sensor* nodes, *hidden* nodes, and *output* nodes. The input nodes act as receptors of external of external signals. The hidden nodes do not receive direct information from the outside world. The output nodes are the ones that produce the output of the network. Nowadays, ANNs are used widely in many fields, among others, rainfall forecasting [13] and credit risk evaluation [14]

3.1 Concepts, terminology and definitions

Tsoi and Back [15] have done a compilation of concepts, definitions, and terminologies related to recurrent neural networks. However, the terms and definitions presented also encompass feedforward neural networks too. In this section, these terms and definitions are presented.

3.1.1 Neuron

A *neuron* or a *node* is a computing unit that functions as a static mapping of the inputs and the output. In principle, neurons themselves do not have dynamic behaviours. Whenever, an artificial neural network has a dynamic behaviour, it is the result of the connections between neurons.

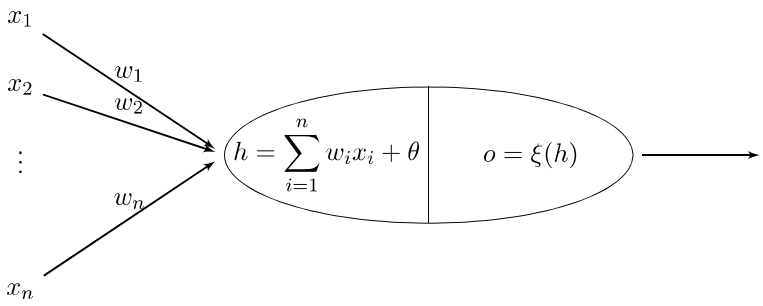


Figure 3.1: Diagram of a neuron.

Fig. 3.1 shows the representation of a neuron with n inputs and one output. The activation value h of the neuron is given by

$$h = \sum_{i=1}^{n} w_i x_i + \theta, \tag{3.1}$$

where x_i are the input values of the neuron with $x_i \in \mathbb{R}$, w_i are the *connection weights* of the inputs with $w_i \in \mathbb{R}$, and θ is the *bias value* of the neuron with $\theta \in \mathbb{R}$. The connection weights, w_i, are bounded by w_{min} and w_{max}, $w_{min} \leq w_i \leq w_{max}$. Typically, we have $w_{min} = -1$ and $w_{max} = 1$. The output of the neuron y is

$$o = \xi(h), \tag{3.2}$$

where ξ is the so called activation function of a neuron, a static monotic non-linear function, $\xi : \mathbb{R} \to \mathbb{R}$. Different activation functions results in different neuron models. Examples are shown in Table 3.1.

Type	h	$\xi(h)$
Sigmoid and related functions.	$\sum_{i=1}^{n} w_i x_i$	$\frac{1}{1+e^{-h}}$ $\tanh(h)$ $\frac{h}{1+\lvert h\rvert}$
Radial Basis Functions (RBF)	$\sum_{i=1}^{n} \frac{(x_i - c_i)^2}{\sigma^2}$, where c_i, $i = 1, 2, ..., n$, composes $\mathbf{c}$ the center of the RBF and σ its width.	e^{-h}

Table 3.1: Examples of activation functions and their respective activation value.

The activation function is crucial to the behaviour and computational power of an ANN [16]. Thus, some authors [17, 18, 19] use spline interpolation functions as activation functions, with different ranges of success and performance.

3.1.2 Layer

A layer is a one dimensional array of neurons. Neurons in a layer perform computations in a lockstep manner: The outputs of the layer are computed at the same time in "parallel". Thus, a layer receives the inputs and computes the outputs in one computational step. In this work, we focus in three types of layers: *input layer*, *output layer*, and *hidden layer*. The input layer is composed by the neurons in the network that receive the inputs to the network. The inputs to the network and the neurons in the input layer form a one-to-one relationship. The output of each of these neurons is given by the neuron's activation function evaluated at the value of the respective input. The output layer is the last layer in the network, and it is formed by the neurons that produce the output of the network. A hidden layer is one that is neither an input layer nor an output layer. An ANN must have an input and output layer, however it can have zero or more hidden layers.

3.1.3 Connection

A connection defines the link between two neurons. Connections can be classified by representation and the number of outputs. The connection can represent either a constant weight, or a linear dynamical system.

Linear dynamical connections have internal state

$$\boldsymbol{z}(t) = \boldsymbol{F}\boldsymbol{z}(t-1) + u(t)\boldsymbol{g}, \tag{3.3}$$

where $\boldsymbol{z}(t)$ is a vector of dimension n_s, $\boldsymbol{F}$ is an $n_s \times n_s$ matrix, $\boldsymbol{g}$ is a vector of dimension n_s, and $\boldsymbol{u}(t)$ is the scalar input to the connection. The values of $\boldsymbol{F}$ and $\boldsymbol{g}$ define the linear dynamic system. The current internal state of the connection depends on the previous internal state of the connection.

A connection has only one input but can have one or more outputs. Thus, connections can be either Single Input Single Output (SISO) or Single Input Multiple Outputs (SIMO). Table 3.2 shows the output of different classes of connections. From this point onwards, whenever we use the term connection, we refer to a static (i.e. with constant weight) and SISO connection.

	Constant weight	Linear Dynamical System
SISO	$y(t) = wu(t)$, where w is the constant weight.	$y(t) = \boldsymbol{h}^T \boldsymbol{z}(t)$, where $\boldsymbol{h}$ is a vector of dimension n_s.
SIMO	$\boldsymbol{y}(t) = \boldsymbol{w}u(t)$, where both $\boldsymbol{w}$ and $\boldsymbol{y}(t)$ are vectors of dimension p.	$\boldsymbol{y}(t) = \boldsymbol{H}\boldsymbol{z}(t)$, where $\boldsymbol{H}$ is a $p \times n_s$ matrix, and $\boldsymbol{y}(t)$ is vector of dimension p.

Table 3.2: Output $y(t)$ of the connection depending on the classification, the input $u(t)$, and the internal state $\boldsymbol{z}(t)$.

A connection can be further classified as *feedforward connections* and *feedback connections*, see Fig. 3.2. In a feedforward connection, the signal travels from the input towards the output. And, in a feedback connection, the signal travels from the output towards the input. The terms input and output refer here to the network's input and output layers. In this sense, feedforward connections involve two neurons: the flow of information, from the output of one neuron to the input of the other neuron, coincides with the flow of information in the network, from the input layer to the output layer. Whereas, feedback connections can involve one or two neurons: the flow of information between the neurons, in this case, goes against the flow of information in the network. A feedback connection that involves only one neuron is known as a *local recurrent* (LR) connection, as it connects the neuron's output to the same neuron's input.

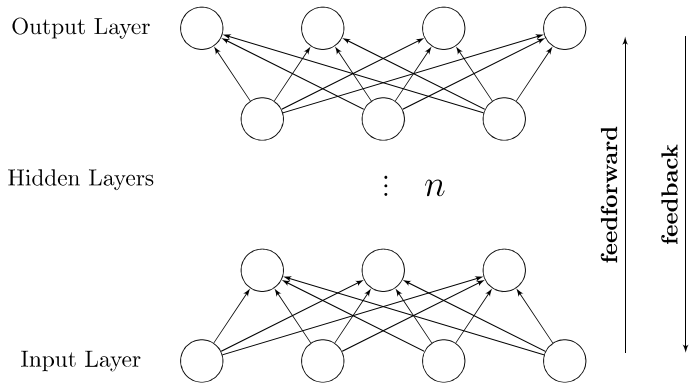


Figure 3.2: Artificial neural network with n hidden layers, the input layer indicates the network's inputs and the output layer the network's outputs. Here, the arrows show the direction of information flow for feedforward and feedback connections.

We have two types of non-local connections *global connections* and *non-local recurrent connections*. In a global connection, a neuron from one layer is connected to a neuron

of another layer. In a non-local recurrent connection, a neuron is connected to a neuron in the same layer. Global connections can be either feedforward or feedback. However, non-local recurrent connections are only feedback connections.

3.1.4 Learning

Learning is the process of altering the internal representation of an ANN (i.a. connection, connections weights, and activation functions) to adapt the network to the problem. Basheer et al [12] define the goal of the process of learning in an ANN as: "An ANN-based system is said to have learnt if it can (i) handle imprecise, fuzzy, noisy, and probabilistic information without noticeable adverse effect on response quality, and (ii) generalize from the tasks it has learned to unknown ones." There are many methods for approaching learning in ANNs. Among others, we can use gradient descent to adapt the weights in the process known as *backpropagation* [20], or use genetic algorithms to do the same [21].

3.2 Artificial Neural Network Architectures

In an artificial neural network, the arrangement of neurons and connections defines the network's *architecture*. As a consequence of all possible connection classes, many ANN architectures are possible. In this work, we focus in two particular ANN architectures: *feedforward neural networks* (FNN) and *recurrent neural networks* (RNN).

3.2.1 Feedforward Neural Networks

A feedforward neural network is an ANN where all connections are feedforward and of constant weight [1]. Consequently, this kind of networks do not have feedback connections. In this regard, if this type of ANN is viewed as a weighted directed graph, no cycles are present.

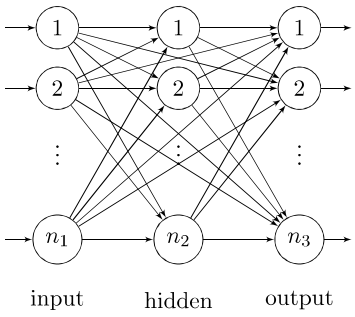


Figure 3.3: An example of a feedforward neural network.

For understanding how a feedforward neural network computes its output. Let us consider a three layered ANN, an input layer with n_1 neurons, a hidden layer with n_2

neurons, and an output layer of n_3 neurons, see Fig. 3.3. Given the input vector $\boldsymbol{u}$ of size n_1, the output of the input layer y_1 given by

$$\boldsymbol{y_1} = \boldsymbol{\xi_1}(\boldsymbol{u}), \tag{3.4}$$

where $\boldsymbol{\xi_1}$ is a vector of dimension n_1 with $\boldsymbol{\xi_1}^T = [\xi_1(\cdot), \xi_2(\cdot), ..., \xi_{n_1}(\cdot)]^T$, and each $\xi_i(\cdot)$ is the i-th neuron's activation function. The output of the hidden layer $\boldsymbol{y_2}$ is given by

$$\boldsymbol{y_2} = \boldsymbol{\xi_2}(\boldsymbol{W_1^2 y_1} + \boldsymbol{\theta_2}), \tag{3.5}$$

where $\boldsymbol{W_1^2}$ is $n_2 \times n_1$ matrix that represents the weights of the connections between the input layer neurons and the hidden layer neurons, $\boldsymbol{\theta_2}$ is a vector of dimension n_2 that represents the bias values of the neurons, and $\boldsymbol{\xi_2}$ is a vector of dimension n_2 of the neurons' activation functions. Finally, the output of the network $\boldsymbol{y_3}$ is given by

$$\boldsymbol{y_3} = \boldsymbol{\xi_3}(\boldsymbol{W_1^3 y_1} + \boldsymbol{W_2^3 y_2} + \boldsymbol{\theta_3}), \tag{3.6}$$

where $\boldsymbol{W_1^3}$ and $\boldsymbol{W_2^3}$ are $n_2 \times n_1$ and $n_3 \times n_2$ matrices representing the weights of the connections of the input layer neurons to the output layer neurons, and the connections of the hidden layer neurons to the output layer neurons, respectively. The n_3 vector $\boldsymbol{\theta_3}$ is the neurons' bias values. And, $\boldsymbol{\xi_3}$ is the vector of dimension n_3 of the neurons' activation functions.

3.2.2 Recurrent Neural Networks

A recurrent neural network is an ANN that includes feedback connections. Some examples of RNNs are Hopfield networks, Bidirectional Associative Memory (BAM), Boltzmann machines, and recurrent backpropagation networks [22]. Fig. 3.4 shows an example of a recurrent neural network, we can observe local, non-local and global recurrent connections. In this figure, we have three layers: the input layer, a hidden layer, and output layer. The hidden layer is formed by neurons 4 and 5. The connection from neuron 5 to neuron 4 is a non-local recurrent connection, a feedback connection in the same layer. The self looping connection of neuron 5 is a local recurrent connection. The connection from the output neuron 7 to neuron 4 is global feedback connection. These feedback connections introduce a sense of memory to the network. This sense of memory allows recurrent neural networks to work on sequential information, where a sense of order and time is needed.

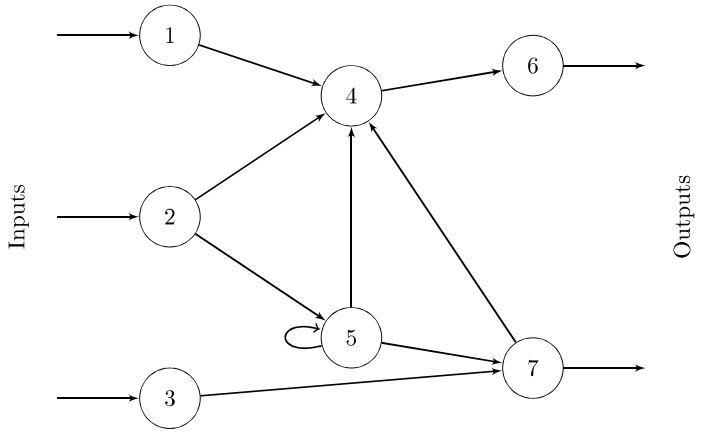


Figure 3.4: Example of a recurrent neural network.

The computation model of a recurrent neural network is similar to that of a feed-forward neural network. However, all neurons are "activated" at the same time, the neurons have an initial activation (output) value, and the output of each neuron is saved for the next computational tic. A recurrent neural network is used to process a signal (an input that depends on time), $\boldsymbol{u}(t)$. As computers are discrete machines (work in discrete steps), recurrent neural networks also work in discrete steps. Thus, the time t is defined in a discrete manner. And, so does the output of each neuron $o_i(t)$, for $i = 1, 2, ..., n$ with n being the number of neurons in the network. Let us define $\boldsymbol{o}(t) = [o_1(t), o_2(t), ..., o_n(t)]$, the output (state) vector of all neurons. For a neuron i that is not an input neuron, its output o_i is given by

$$o_i(t) = \xi(\sum_{j \in l} w_{ji} o_j(t-1) + \theta_i), \tag{3.7}$$

where ξ is the neuron's activation function, l is the set of neurons which are connected to the neuron i (the output of the connection is the neuron i), w_{ji} is the weight of the connection from neuron j to neuron i, $o_j(t-1)$ is the previous output of the neuron j, and θ_i is the bias value. For an input neuron i, the output is

$$o_i = \xi(\sum_{j \in l} w_{ji} o_j(t-1) + u_i(t) + \theta_i), \tag{3.8}$$

where $u_i(t)$ is the input value that match to the i input neuron. The other values have the same meaning as in 3.7.

The output of the network is given by the output values of the neurons in the output layer. The output of the network is defined in a discrete manner and can be fed back to the network.

3.3 Backpropagation

Backpropagation is a learning procedure that minimizes the difference between the network's output and the desired output (target) by adjusting the connection weights, this

procedure can be used for both, feedforward and recurrent neural networks [20, 23, 24]. When using backpropagation with recurrent neural network, the network is "unrolled" in multiple feedforward neural networks, one for each time step of the input signal. These networks are connected in a train like fashion, the output of the first network is connected to the input of the second network, and so on. This method works with signals of finite time. However, it duplicates the network multiple times, thus increasing the computation time.

The backpropagation procedure can be divided in four stages: the *feedforward stage*, where the outputs of the network are calculated using the input training pattern; the *associated errors stage*, where the output of the network is compared to the its expected value; the *backpropagation stage*, where the associated errors are propagated backwards; and the *weight updating stage*, where using the backpropagated values weights are updated. The training pattern is formed by the input vector $\boldsymbol{x} = [x_1, x_2, ..., x_h]$ and the target outputs $\boldsymbol{t} = [t_1, t_2, ..., t_m]$, where h and m are the number of inputs and outputs respectively.

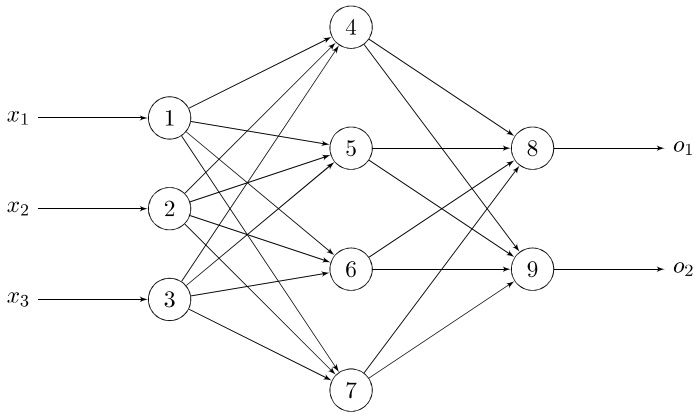


Figure 3.5: An example of a feedforward neural network for backpropagation, this network is a multilayer perceptron.

For understanding backpropagation, let us assume a FNN with three layers, like the one in Fig. 3.5. In this network, each layer is only connected to the preceding and proceeding layers. For this network we define the *weight matrix* or *adjacency matrix*

$$\boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1n} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & w_{n3} & \dots & w_{nn} \end{bmatrix}, \tag{3.9}$$

where each w_{ij} is the weight of the connection from the neuron i to the neuron j and n is the number of neurons in the network. When two neurons are not connected the connection weight between is defined as zero, $w_{ij} = 0$. For the input neurons, $i \in \{1, 2, 3\}$,

the output is defined as

$$y_i = \xi(x_i). \tag{3.10}$$

The activation value for a non-input neuron i is given by

$$h_i = \sum_{i}^{n} w_{ij} y_j + \theta_i, \tag{3.11}$$

where o_j is the output value of the neuron j and θ_i is the bias value of neuron i. Thus, the output of a neuron i is

$$y_i = \xi(h_i). \tag{3.12}$$

When working with backpropagation the activation function must be differentiable. Here, we assume that all neurons in the network have the same activation function, ξ.

In the feedforward stage, the output of the network is computed from the input vector, $\boldsymbol{x}$. For the example network, the input vector is $\boldsymbol{x} = [x_1, x_2, x_3]$. And, the output vector is $\boldsymbol{o} = [o_1, o_2]$. The outputs are given by

$$o_1 = \xi(\sum_{j=1}^{n} w_{j8}(\xi(\sum_{k=1}^{n} w_{kj} y_k + \theta_j)) + \theta_8), \tag{3.13}$$

and

$$o_2 = \xi(\sum_{j=1}^{n} w_{j9}(\xi(\sum_{k=1}^{n} w_{kj} y_k + \theta_j)) + \theta_9). \tag{3.14}$$

From the target vector $\boldsymbol{t} = [t_1, t_2]$ and the output vector $\boldsymbol{o}$, we calculate the network's error for the training pattern as

$$E = \sum_{i=1}^{2} \frac{(t_i - o_i)^2}{2}. \tag{3.15}$$

After calculating the associated error, we start the backpropagation stage. We propagate backwards the errors by calculating the partial derivative of the error with respect to each weight w_{ij}, using the chain rule twice we obtain

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial h_j} \frac{\partial h_j}{\partial w_{ij}}. \tag{3.16}$$

From (3.11), we know that

$$\frac{\partial h_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} (\sum_{k=1}^{n} w_{kj} y_k + \theta_j) = \sum_{k=1}^{n} \frac{\partial w_{kj}}{\partial w_{ij}} y_k = \frac{\partial w_{ij}}{\partial w_{ij}} y_i = y_i. \tag{3.17}$$

The terms of the RHS are zero for any value of k except for $k = i$. The term $\frac{\partial y_j}{\partial h_j}$ depends on the activation function of the neuron, thus

$$\frac{\partial y_j}{\partial h_j} = \frac{\partial \xi(h_j)}{\partial h_j}. \tag{3.18}$$

The partial derivative of the quadratic error with respect to the output of the neuron, $\frac{\partial E}{\partial y_j}$, depends on whether or not the neuron is an output neuron or not. If the neuron is in the output layer, we have that y_j is o_j. Thus, we obtain

$$\frac{\partial E}{\partial y_j} = \frac{\partial}{\partial y_j}(t_j - y_j)^2 = t_j - y_j. \tag{3.19}$$

However, If the neuron j is an inner neuron, we have

$$\frac{\partial E}{\partial y_j} = \sum_{l \in L} w_{jl} \frac{\partial E}{\partial y_l} \frac{\partial y_l}{\partial h_l}, \tag{3.20}$$

where L is the set of neurons that receive as input the output of the neuron j. This definition is recursive. Therefore, we have

$$\frac{\partial E}{\partial w_{ij}} = y_i \frac{\partial E}{\partial y_j}. \tag{3.21}$$

Finally, the weight updating stage uses

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}, \tag{3.22}$$

where η is a constant known as the learning rule, as the updating value. The new weight is the old weight plus Δw_{ij}. The definition of Δw_{ij} determines that the value of E decreases.

Backpropagation can be used to reduce the error E and train the network. However, it fails when looking a global minimum of the error function. Also, backpropagation does not change the architecture of the network, it only alters the connection weights.

Chapter 4

Neuro Evolution of Augmenting Topologies

Besides learning procedures like Backpropagation, Evolutionary Algorithms (EA) can be used to adapt ANN-based systems, in particular, Topology and Weight Evolving Artificial Neural Networks (TWEANN) uses EAs to perform automatic architecture alteration and connection weight adaptation of ANNs [25]. Miller *et al.* in [26] indicated that the search space generated by the architectures performance values, and all the possible architectures, has characteristics which make EAs good candidates for searching optimal network architectures. Neuro Evolution of Augmenting Topologies [27] is a TWEANN which uses a genetic algorithm with explicit fitness sharing to evolve artificial neural networks. The evolution objective is to obtain as minimally complex artificial neural networks as possible. Thus, the evolution process starts with minimally connected artificial neural networks; inputs neurons fully connected to output neurons without hidden neurons.

4.1 Algorithm

Parameter	Description
N	**Population size**, the number of individuals in the population.
σ_s	**Similarity threshold**, the maximum distance between two individuals in order to belong to the same species (niche).
p_k	**Kill percentage**, indicates the fraction of the population to be replaced at each generation.
p_c	**Crossover percentage**, determines the fraction of the selected population that will go through crossover.
p_I	**Interspecies probability**, the probability for which two individuals from different species can reproduce.
p_{an}	**Add-node mutation probability**, the probability of an add-node mutation to take place.
p_{ac}	**Add-connection mutation probability**, the probability of an add-connection mutation to take place.
p_w	**Weight mutation probability**, the probability of weight mutation to occur at a connection gene.
p_{gr}	**Gene re-enabled probability**, the probability of a connection gene being re-enabled in an offspring if it was inherited as disabled.

Table 4.1: Parameters of Neuro Evolution of Augmenting Topologies.

The algorithm of NEAT requires many input parameters. Thus, before talking of the steps in NEAT, it is important to enumerate and describe each control parameter. Table 4.1 shows the parameters used to govern the behaviour of a NEAT run[1]. The first parameter of importance is the population size, N. The size of the population determines how many parallel searches can be done. But, with a larger N the computational cost of the algorithm is increased.

The similarity threshold is a value that specifies the niche size in the population. Nonetheless, a difficulty arises when selecting an adequate threshold value. The selection requires an *a priori* knowledge of the landscape of the search space of all possible structural configurations of ANNs. The dimensionality of this search space increases as the search done by NEAT progresses.

The kill percentage p_k and the crossover percentage p_c indicate the portion of the population that survives, and the portion of the population that participates in crossover and produce new individuals. The add-node mutation, add-connection mutation and

[1]The parameters table is based on the Matlab implementation of NEAT found on the official project web page.

weight mutation probabilities are necessary to induce the addition of new structures in the population. However, these probabilities should not be too high, as they introduce noise into the search.

Alg. 4 shows the general process of NEAT. The process of NEAT starts with the creation of an *initial population*. The initial population is composed of fully connected and structurally minimal artificial neural networks. In other words, networks composed only of input and output neurons, which are fully connected; each output neuron has a connection coming from all input neurons. After creating this initial population, NEAT starts the iterative process of each generation. This process stops either when the maximum number of generations is reached, or the fitness objective is achieved. Finally, NEAT outputs the final population.

At the start of each generation, the fitness of each individual is calculated using the fitness function. This fitness function depends on the problem where NEAT is applied, whether the problem is solved using supervised or unsupervised learning. For example, when using NEAT for solving the XOR gate problem, the fitness function depends on the training set. This training set is built using the truth table of the logical XOR function. However, when using NEAT to solve the double pole balancing problem, a training set can not be used as the system is dynamical (changes over time). Thus, the fitness function in this case depends on the simulation of the physical system.

The second and third step in each generation relates to the niching method used in NEAT, explicit fitness sharing. Niching is used to conserve and promote diversity in the population, with the goal of avoiding that the population be overun by a highly fit individual. And, the search can look in multiple subpopulations or niches. The details of the niching method are discussed later. The output of the speciation process L is maintained through the generations. However, individuals from the old population in L are removed from the species sets, only keeping the species' representative.

The later steps at each generation comprises the reproduction and mutation stages of NEAT. Not all individuals in the current population are selected for reproduction, the percentage of the population that is not going to be selected is given by the kill percentage, p_k. The selection of individuals to be reproduced is done using stochastic universal sampling, SUS. After this selection, the selected individuals are paired taking into account the interspecies probability, p_I. In principle, individuals from different species should not be able to reproduce. However, in NEAT to allow the exchange of genes between species, interspecies reproduction is allowed. The parameter p_I is used to regulate the probability of interspecie reproduction. Not all pairs undergo crossover, the portion of the pairs that do so is given by the crossover percentage, p_c. After the crossover process, the offsprings are produced and these offsprings replace their parents in the population. Finally, all individuals in the population are subject to mutation process. The mutation process is parametrized by the different mutation probabilities:

add-node mutation probability p_{an}, add-connection mutation probability p_{ac}, the weight mutation probability p_w, and the gene re-enabled probability p_{gr}.

Algorithm 4: Algorithm of NEAT.

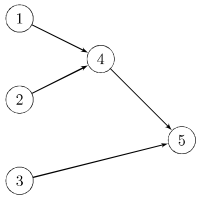
Input: Population size N, similarity threshold σ_s, kill percentage p_k, crossover percentage p_c, interspecies probability p_I, add-node mutation probability p_{an}, add-connection mutation probability p_{ac}, weight mutation probability p_w, gene re-enabled probability p_{gr}, fitness function f, max. number of generations G_{max}, fitness objective γ

Output: Population P

```
Create initial population of fully connected networks of inputs and outputs P of size N;
Set generation count g = 0;
repeat
    Calculate fitness of P in f;
    Speciate P using σ_s creating the species list L;
    Do the fitness sharing process;
    Select group of individuals R to be reproduced using p_k;
    Pair individuals in R taking into account the p_I;
    Crossover pairs with probability p_c;
    Mutate individuals with p_an, p_ac, p_gr, and p_w;
    Increase generation count;
until best_fitness(P) ≥ γ or g ≥ G_max
```

4.2 Genome Encoding

In NEAT, an ANN is encoded by a genome (see Fig. 4.1) that is composed of two gene lists: the *node genes* (Fig. 4.1b), and the *connection genes* (Fig. 4.1c). Node genes represent the neurons or nodes of the artificial neural network. Each node gene has a unique label, and the neuron's type. A neuron can be of type: sensor or input, hidden, or output. The connection genes represent the connectivity mapping of the encoded ANN. A connection gene is composed of: *in node* field, *out node* field, *weight*, an *enabled flag*, and an *innovation number*. The in node field and out node field, indicates the neurons to be connected. The weight indicates the strength of this connection. The enabled flag indicates whether or not this connection gene is expressed in the genome's resulting network. The innovation number is used to track corresponding genes (i.e. that represent the same structure in the current generation).



(a) Expressed artificial neural network.

1	2	3	4	5
Input	Input	Input	Hidden	Output

(b) Node genes.

In: 1	In: 2	In: 3	In: 1	In: 2	In: 4
Out: 5	Out: 5	Out: 5	Out: 4	Out: 4	Out: 5
Weight: 1	Weight: 1	Weight: 1	Weight: 1	Weight: 1	Weight: 1
Disabled	Disabled	Enabled	Enabled	Enabled	Enabled
Innov: 1	Innov: 2	Innov: 3	Innov: 4	Innov: 5	Innov: 6

(c) Connection genes.

Figure 4.1: NEAT genome encoding example.

Formally, a node gene is a tuple (η, θ) such that $\eta \in \mathbb{N}$ and $\theta \in \{0, 1, 2\}$, the value η is the neuron's label and the value θ represents the neuron's type. The representation of the type of neurons given by the value of θ follows: If $\theta = 0$, η represents an input neuron; If $\theta = 1$, η represents a hidden neuron; and If $\theta = 2$, η represents an output neuron. Then, we can define the *node genes space* as $\Xi := \{(\eta, \theta) \,|\, \eta \in \mathbb{N} \wedge \theta \in \{0, 1, 2\}\}$. A connection gene is formally represented as a tuple $(\alpha, \beta, \omega, \kappa, \rho)$ such that: $\alpha, \beta, \rho \in \mathbb{N}$; $\omega \in \mathbb{R}$ with $\omega \in [-1, 1]$; and $\kappa \in \{0, 1\}$. The values α and β represent the in-node and out-node of the connection, respectively. The value ω is the connection weight. The value κ is the enabled flag with $\kappa = 0$ indicates that the gene is not expressed and $\kappa = 1$ indicates that the gene is expressed. And, the value ρ is the connection gene's innovation number. Then, we can define the *connection genes space* as $\Gamma := \{(\alpha, \beta, \omega, \kappa, \rho) \,|\, \alpha, \beta, \omega \in \mathbb{N} \wedge \omega \in \mathbb{R} \wedge \kappa \in \{0, 1\}, \omega \in [-1, 1]\}$. We should note that both the node genes space and the connection gene space are both infinite. With both these definitions we can define the genome space as $G := \{(\Phi, \Psi) \,|\, \Phi \subseteq \Xi \wedge \Psi \subseteq \Gamma\}$.

4.3 Crossover

In NEAT, individuals in the same generation might have different genome sizes. Thus, the crossover process must take into account the varying genome size. The crossover process produces one offspring from two parent genomes. This process is divided in two stages: the *matching* stage, and the *inheritance* stage.

In the matching stage, the innovation numbers are used to align the genomes' connection genes (See Fig. 4.2). The alignment produces three sets of connection genes: *matching genes* M, the *disjoint genes* D, and the *excess genes* E. The matching genes are the connection genes that belong to both parent, they have the same innovation numbers.

Disjoint genes are connection genes that belong to only one parent, and their innovation numbers are less or equal to the minimum of the parents genomes' maximum innovation numbers. The excess genes are the rest of connection genes, they belong to

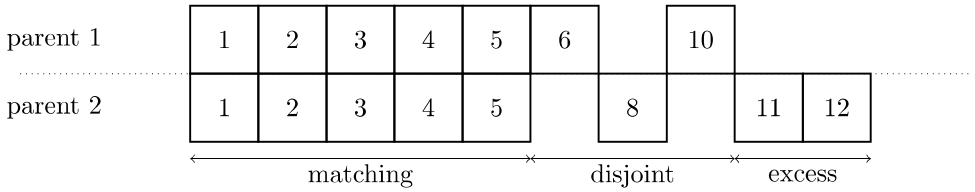


Figure 4.2: Example of the matching stage of crossover. The connection genes are represented by their innovation numbers.

the same parent. The disjoint and excess genes represent the non-common structure between the parents.

In the inheritance stage, an offspring is produced by selecting genes from the sets obtained from the matching stage. This selection is done in accordance to the following selection rules: matching genes are inherited randomly, and non-matching genes (disjoint and excess genes) are inherited depending on the parents fitnesses. If the parents' fitnesses are not equal, the genes from the fitter parent are selected. Otherwise, non-matching genes are inherited randomly.

4.4 Similarity Measurement

In NEAT, the sets of genes, obtained from the matching stage in crossover (i.e. matching, disjoint, and excess), are used to measure similarity between two genomes. This measure δ is given by

$$\delta = c_1\frac{|E|}{N} + c_2\frac{|D|}{N} + c_3\overline{W}, \tag{4.1}$$

where $|E|$ is the number of excess genes, $|D|$ is the number of disjoint genes, N is the number of genes of the larger genome, $\overline{W}$ is the average weight differences of matching genes, and the coefficients c_1, c_2, and c_3 are used to adjust the factors. The distance δ is a "genotypic distance", it measures similarity in the genetic space. In conjunction with the similarity threshold δ_s, this distance is used to speciate the population.

4.5 Niching Method

The niching method used in NEAT is explicit fitness sharing. A derivation of fitness sharing that uses a list of species, L. Each species $l \in L$ is a tuple, (r, T), where r is the genome of the species' representative, and T is the set of individuals in the species. The collections of Ts is a partition of the population. An individual q is said to belong to a species $l = (r, T)$, if and only if, the distance $d(q, r)$ is less than the dissimilarity threshold, σ_s. Niching in NEAT can be divided in two stages, the *speciation*, and the

fitness sharing.

Algorithm 5: Speciation in NEAT.

Input: An individual q, the list of species L, similarity threshold σ_s.

Output: L

```
added? = false
foreach (r, T) ∈ L do
    if d(q, r) < σ_s then
        add(q, T)
        added?=true
        break;
if added? == false then
    create_new_species(q, L)
```

The process of speciation is shown in Alg. 5. In this process, either a non-speciated individual is added to an existing species, or a new species is created. In this new species, the individual is chosen as the representative. Speciation in NEAT is a sequential process and an individual can only belong to one species. In particular, an individual is added to the first species that complies with $d(q, r) < \sigma_s$.

In the fitness sharing process, the fitness of each individual in the population is shared with the individual's species. Therefore, a meassure of the amount of individuals that are in the same species of a individual q is needed, the niche count m_q. Which is given by

$$m_q = |T_q|, \tag{4.2}$$

where T_q is the set of individuals that belong to the same species as q. In contrast to fitness sharing, in explicit fitness sharing the niche count is a natural number. Finally, for the fitness of an individual f_i, the shared fitness f_i' is given by

$$f_i' = \frac{f_i}{m_i}, \tag{4.3}$$

where m_i is the niche count.

Explicit fitness sharing differs from fitness sharing as it only allows a one-to-many relationship between species and individuals. A species can have multiple individuals however an individual can only belong to one species. In fitness sharing, a many-to-many relationship is allowed, a species can have multiple individuals and an individual can belong to multiple species.

4.6 Mutation

NEAT uses mutation to add structure to the population and to do weight adaptation. The possible mutations are: add-node mutation, add-connection mutation, weight muta-

tion, and reenabling mutation. These mutations occour in the population with different probabilities.

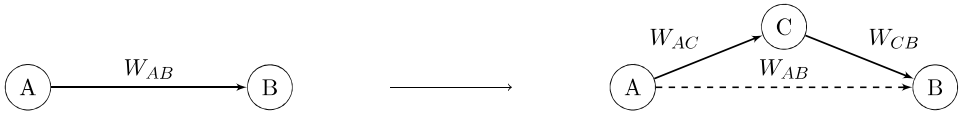


Figure 4.3: Add node mutation, W_{AB}, W_{AC}, and W_{CB} represent the weight of the connections. The dashed line represents a disabled connection.

The add-node mutation adds a node (neuron) to the network by splitting the connection of two connected nodes. Fig. 4.3 shows an example of the add-node mutation. The connection between the nodes A and B has a weight of W_{AB}, the new node C is added by splitting this connection. Two new connections are formed, one between A and C with a weight $W_{AC} = 1$, and the other one between B and C with a weight $W_{CB} = W_{AB}$. The old connection from A to B is disabled. The new connections weights W_{AC} and W_{CB} are set to these values to reduce the effect on the network's fitness that the new node introduces. It should be noted that the new node's activation function introduces an element of non-linearity to the network. At the genome level, the mutation introduces two new connection genes and a node gene, and disables a connection gene.

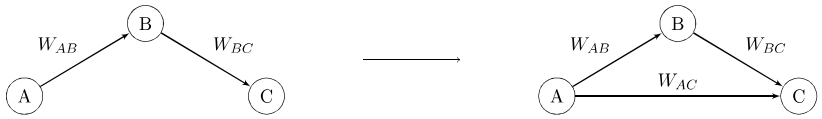


Figure 4.4: Example of the add connection mutation.

The add-connection mutation creates a new connection between two previously unconnected nodes, see Fig. 4.4, here a new connection from node A to node C is introduced with a random weight W_{AC}. In the case of NEAT working with recurrent neural networks, the possible new connections for the example network (left network on Fig. 4.4) are: from B to A, from C to B, from A to C, and from C to A. Meaning that the process of adding a connection takes into account the directivity of the connection. However, when NEAT works with feedforward neural networks this directivity is ignored. In practice, whenever a new connection is going to be added to the network, the algorithm checks whether or not the addition of the connection introduces a cycle in the network. If recurrent networks are allowed and a cycle is going to be introduced, the connection is added. If recurrent networks are not allowed, say when working with feedforward networks, the introduction of cycles is not allowed.

NEAT uses weight mutation to adapt the network's weights. This mutation does not introduce any structural alteration (new connections or nodes), like the add-node and add-connection mutations. In weight mutation, a connection gene is randomly chosen,

and its weight is changed to a new random value. In some implementations of NEAT, the connection weights are bounded by w_{min} and w_{max} parameters, such that the new weight $w \in [w_{min}, w_{max}]$. For this work $w_{min} = -1$, and $w_{max} = 1$. The re-enabling connection mutation takes a previously disabled connection and enables it. Disabled connections result from the add node mutation.

Chapter 5

New Method

In this chapter, we propose a new algorithm for Topology and Weight Evolving Neural Networks. This new algorithm is based on NEAT, it uses the same genome encoding and mutation operators as NEAT. However, it introduces a new method for calculating the similarity between two genomes, and uses a different niching method, that is, deterministic crowding.

The chapter is divided in three sections. Section 5.2 exposes the problems of NEAT's similarity measure, δ. Section 5.2 shows the weaknesses of NEAT's niching method, explicit fitness sharing. Finally, in section 5.3, we show the new algorithm.

5.1 The problem of NEAT's similarity measure

For convenience, we shall remember the definition of δ given in 4.4:

$$\delta = c_1\frac{|E|}{M} + c_2\frac{|D|}{M} + c_3\overline{W}, \tag{5.1}$$

where $|E|$ and $|D|$ are the number of excess and disjoint genes, $\overline{W}$ is the average weight difference of matching genes, M is the number of connection genes of the larger genome, and c_1,c_2 and c_3 are real value coefficients. Formally, we define the similarity measure as a map $\delta : G \times G \to \mathbb{R}$, where G is the set of genomes in NEAT. In the same manner, we define $E : G \times G \to \mathbb{N}$, $D : G \times G \to \mathbb{N}$, and $\overline{W} : G \times G \to \mathbb{R}$. A genome in NEAT is a tuple (N, C), where N is the set of node genes and C is the set of connection genes.

In the process of NEAT, we want the population to converge to a global optima. To ensure convergence we need a metric, in the purely mathematical sense. Although δ is non-negative, nullifying if two copies of the same genome are provided, it does not satisfy the triangular inequality

$$\delta(x, z) \leq \delta(x, y) + \delta(y, z), \tag{5.2}$$

for $x, y, z \in G$.

Proof. Consider the particular case satisfying the following assumptions:

(i) $x := (N_x, C_x)$, $y := (N_y, C_y)$ and $z := (N_z, C_z)$ are genomes with $N_x \subset N_y$ and $N_z \subset N_y$;

(ii) the coefficients c_1, c_2 and c_3 have the same value c, $c_1 = c_2 = c_3 = c$;

(iii) all connection weights in C_x, C_y and C_z have the same value; and

(iv) the sets of connection genes complies with $C_x \subset C_y$, $C_z \subset C_y$, $C_x \cap C_z = \emptyset$, $|C_x| = |C_z|$, and $|C_y| = |C_x| + |C_z|$,

From (5.1) and (5.2), we have

$$\begin{aligned}
\delta(x,z) &= c_1\frac{E(x,z)}{\max(|N_x|,|N_z|)} + c_2\frac{D(x,z)}{\max(|N_x|,|N_z|)} + c_3\overline{W}(x,z)\\
\delta(x,y) &= c_1\frac{E(x,y)}{\max(|N_x|,|N_y|)} + c_2\frac{D(x,y)}{\max(|N_x|,|N_y|)} + c_3\overline{W}(x,y)\\
\delta(y,z) &= c_1\frac{E(y,z)}{\max(|N_y|,|N_z|)} + c_2\frac{D(y,z)}{\max(|N_y|,|N_z|)} + c_3\overline{W}(y,z).
\end{aligned} \tag{5.3}$$

The terms N and N' are defined as the number of connection genes for the largest genome, in number of connection genes, from the two involved genomes in δ. Therefore, from assumption (iv), we know that $\max(|N_x|,|N_y|) = \max(|N_y|,|N_z|) = |C_y|$, $\max(|N_x|,|N_z|) = |C_x|$ and $|C_x| < |C_y|$. Assumptions (ii) and (iii) simplifies (5.3) to

$$\begin{aligned}
\delta(x,z) &= c\frac{E(x,z)}{|C_x|} + c\frac{D(x,z)}{|C_x|}\\
\delta(x,y) &= c\frac{E(x,y)}{|C_y|} + c\frac{D(x,y)}{|C_y|}\\
\delta(y,z) &= c\frac{E(y,z)}{|C_y|} + c\frac{D(y,z)}{|C_y|}.
\end{aligned} \tag{5.4}$$

Using (iv) we obtain

$$\begin{aligned}
E(x,z) + D(x,z) &= |C_x \cup C_z| = |C_x| + |C_z|\\
E(x,y) + D(x,y) &= |C_x \cup C_y| - |C_x \cap C_y| = |C_y|\\
E(y,z) + D(y,z) &= |C_y \cup C_z| - |C_y \cap C_z| = |C_z|\\
\delta(x,z) &= c\frac{|C_x| + |C_z|}{|C_x|}\\
\delta(x,y) &= c\frac{|C_x|}{|C_y|}\\
\delta(y,z) &= c\frac{|C_z|}{|C_y|}.
\end{aligned} \tag{5.5}$$

Should δ abide the triangular inequality (5.2), we should obtain in the above case

$$\frac{|C_x| + |C_z|}{|C_x|} \leq \frac{|C_x| + |C_z|}{|C_y|}, \tag{5.6}$$

from where, we deduce that $|C_x| \geq |C_y|$, which contradicts $|C_x| < |C_y|$.

The reader might find assumption (ii) restrictive. However, it should be noted that for any configuration of the coefficients c_1, c_2 and c_3, three genomes $x, y, z \in G$ can be constructed, such that

$$c_1 E(x,z) + c_2 D(x,z) = c_1(E(x,y) + E(y,z)) + c_2(D(x,y) + D(y,z)). \tag{5.7}$$

With these x, y and z, we can prove, in an analogous manner, that δ violates the triangular inequality. We emphasize with this, that the triangular inequality can easily be violated and such cases should not be neglected

Another caveat with δ is that it works in the genotypic space of NEAT, and Mahfoud noted in [4] the importance of phenotypic distances. These distances reduce the number of replacement errors, where a replacement error is defined as, given three individuals, a genetic algorithm wrongly measures the individuals' similarities to each other. And, thus wrongly determines the closeness between them.

5.2 The problem of NEAT's niching method

NEAT's niching method, explicit fitness sharing, requires the maintenance of a species list, a structure that increases the time and memory complexity of the algorithm. Also, speciation is strongly dependent on the species' representatives, when deciding the species of an individual, the method (as described in NEAT) selects the first species for which the similarity between the individual and the species' representative is less or equal to the similarity threshold, σ_s. Thus, the method does not always choose the most similar species to the individual, the appropriate one.

Explicit fitness sharing requires an *a priori* knowledge of the search space landscape, when deciding the value of the similarity threshold, δ_s [28]. Moreover, the fixing of δ_s assumes that the niches in the search space have the same size. Additionally, the computation of each individual niche count, m_i, has complexity, in the worst case scenario, $O(N^2)$, with N being the population size [8]. The complexity is based on the number of computations of the sharing function, $sh(i,j)$.

5.3 Deterministically Crowded - NEAT

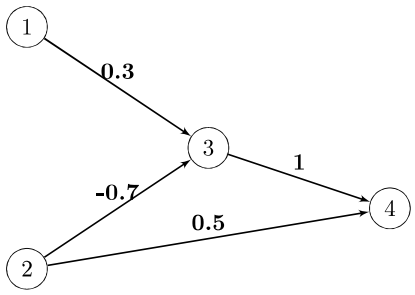
In this work, we propose Deterministically Crowded - Neuro Evolution of Augmenting Topologies (DC-NEAT), as a new algorithm for TWEANN, based on NEAT. The algorithm uses the same encoding as NEAT, the list of node genes and connection genes. In addition, DC-NEAT have the same mutation operators as NEAT: add-node mutation, add-connection mutation, weight mutation, and the connection re-enable muta-

tion. Nonetheless, DC-NEAT differs from NEAT in the crossover procedure, the genome distance, and the niching method.

DC-NEAT employs deterministic crowding as the niching method. However, deterministic crowding needs that the crossover procedure produces two offspring from the same parents; NEAT's crossover produces only one offsprings from the two parents. Thus, a new crossover must be employed in DC-NEAT. Furthermore, since NEAT's similarity measure is not a metric, a new phenotypic distance is used.

5.3.1 New phenotypic distance

All networks encoded by NEAT's genome encoding can be represented as an adjacency matrix. For example, the network in Fig. 5.1a is represented by the adjacency matrix $\boldsymbol{A}$ in 5.1b. The elements of $\boldsymbol{A}$, a_{ij}, are defined as the weight of the connection from neuron i to neuron j, for $i, j = 1, 2, ..., n$ where n is the number of neurons in the network. A weight of value zero, $a_{ij} = 0$, indicates that there is no connection between neurons i and j.



(a) Example of an artificial neural network that is possible as the output of NEAT.

$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 0.3 & 0 \\ 0 & 0 & -0.7 & 0.5 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(b) The adjacency matrix of the network in Fig. 5.1a

Figure 5.1: Example of an artificial neural network and its adjacency matrix.

We leverage this possible representation of networks to define the phenotypic distance in DC-NEAT. We define the distance between adjacency matrices of size $n \times n$,

$$\begin{aligned} d : \mathbb{R}^{n^2} \times \mathbb{R}^{n^2} &\to \mathbb{R} \\ (\boldsymbol{X}, \boldsymbol{Y}) &\mapsto d(\boldsymbol{X}, \boldsymbol{Y}) = \|\boldsymbol{X} - \boldsymbol{Y}\|_\infty . \end{aligned} \tag{5.8}$$

Then, we define the helper function

$$\begin{aligned} h : G &\to \bigcup_{n=1}^{\infty} \mathbb{R}^{n^2} \\ x &\mapsto h(y). \end{aligned} \tag{5.9}$$

This helper function transforms the genome x into the adjacency matrix $h(x)$. The value, n, is the number of node genes present in the genome. We define the genome distance

$$\begin{aligned} \delta : G \times G &\to \mathbb{R} \\ (x, y) &\mapsto \delta(x, y) = d(h(x), h(y)). \end{aligned} \tag{5.10}$$

The distance δ transforms the genomes x and y into their adjacency matrix representation $h(x)$ and $h(y)$, then utilizes d to calculate the distance value. This distance is a metric, $(\mathbb{R}^n, d)$. In the case, that the dimensions of the adjacency matrices from the genomes x and y do not match. The matrix with the smaller size is padded with columns and rows of zeroes, until the dimensions of the ajacency matrices match. It should be noted that hidden nodes' genes are added after the input nodes' genes and output nodes' genes, thus hidden nodes would be represented by number larger than the input and output nodes.

5.3.2 New Crossover procedure

DC-NEAT's crossover procedure is a map

$$\begin{aligned} \zeta : G \times G &\to G \times G \\ (x, y) &\mapsto \zeta(x, y). \end{aligned} \tag{5.11}$$

The function ζ takes two genomes, parent 1 and parent 2, and produces two offspring, offspring 1 and offspring 2. Hereafter, we denote parent 1 and parent 2 as p_1 and p_2, and offspring 1 and offspring 2 as s_1 and s_2. Then, we define $p_1 = (N_{p_1}, C_{p_1})$, $p_2 = (N_{p_2}, C_{p_2})$, $s_1 = (N_{s_1}, C_{s_1})$, and $s_2 = (N_{s_2}, C_{s_2})$. In ζ, the connection genes of both parents are completely inherited by both offspring. In other words, we have $C_{p_1} \cup C_{p_2} = C_{s_1} \cup C_{s_2}$ and $C_{p_1} \cup C_{p_2} = C_{s_1} \cup C_{s_2}$.

The crossover procedure starts by aligning the connection genes of both parents, C_{p_1} and C_{p_2}, in a similar fashion as in NEAT's crossover matching stage, this alignment is done using the connection genes' innovation numbers. The alignment produces a set of tuples, the *aligned pairs*, denoted Q, where for $(t_1, t_2) \in Q$, $t_1 \in C_{p_1} \cup \{\varnothing\}$ and $t_2 \in C_{p_2} \cup \{\varnothing\}$. Here, the symbol $\varnothing$ denotes that for a connection gene, t_i with $i \in \{1, 2\}$, a matching connection gene, a connection gene with the same innovation number, can not be found in the other genome. Formally, we have that $Q \subseteq (C_{p_1} \cup \{\varnothing\}) \times (C_{p_2} \cup \{\varnothing\})$ and $(\varnothing, \varnothing) \notin Q$. For example, for the parent genomes p_1 and p_2, with connection genes $C_{p_1} := \{1, 2, 3, 7\}$ and $C_{p_2} := \{1, 2, 3, 4, 6\}$, here the connection genes

are represented by their innovation numbers, we have that $T = \{1, 1, 2, 2, 3, 3, 4, 6, 7\}$ and $Q = \{(1, 1), (2, 2), (3, 3), (\varnothing, 4), (\varnothing, 6), (7, \varnothing)\}$.

After the matching is done, we create the two offspring by randomly assigning each aligned pair component to each offspring, see Alg. 6. When, either component (t_1 or t_2 in Alg. 6) of the pair has the value $\varnothing$, it is ignored and nothing is added to the offspring connection genes. Therefore, we can obtain offspring with different number of connection genes in their genomes. To build the complete genome for the offsprings, the node genes from both parents are joined, and then each offspring's node genes are created taking into account the connection genes. If a node does not participate in any of the connections in the offspring, it is removed from the offspring's node genes list. Thus, the node genes sanity is maintained.

Algorithm 6: Algorithm for the creation of two offspring from the aligned pairs, Q.

Input: The aligned pairs Q

Output: The offspring 1 and 2 connection genes, C_{s_1} and C_{s_2}

```
C_{s_1}=[];
C_{s_2}=[];
foreach (t_1, t_2) ∈ Q do
    p = random(0,1);
    if p ≤ 0.5 then
        if t_1 ≠ ∅ then
            Append t_1 to C_{s_1};
        if t_2 ≠ ∅ then
            Append t_2 to C_{s_2};
    else
        if t_1 ≠ ∅ then
            Append t_2 to C_{s_1};
        if t_2 ≠ ∅ then
            Append t_1 to C_{s_2};
return C_{s_1} and C_{s_2}
```

5.3.3 Deterministic Crowding in DC-NEAT

DC-NEAT uses deterministic crowding for niching, instead of explicit fitness sharing. Deterministic crowding was chosen for three main reasons: (i) deterministic crowding does not require any *a priori* knowledge of the search space landscape shape, (ii) deterministic crowding has a simpler implementation compared to explicit fitness sharing, and (iii) deterministic crowding has lower time complexity.

Explicit fitness sharing in NEAT calls for the fine tuning of the similarity threshold, σ_s, this parameter determines the niche size of each species. Thus, knowledge of the search space landscape formed by the fitness values of all possible ANNs configurations is needed, which in practice is not available. In explicit fitness sharing, species can be understood as hyperspheres of radius σ_s and centered at the species' representative. And, since the election of the species' representative is arbitrary. NEAT might wrongly partition a cluster of individuals into two species. Specially, if the representative was chosen from the boundary of the cluster. Deterministic crowding does not require the similarity threshold parameter, in fact, no knowledge of the search space landscape is presumed.

In contrast with explicit fitness sharing, deterministic crowding does not explicitly manage the species in the population. Indeed, deterministic crowding works without NEAT's species list, L, reducing the implementation complexity of DC-NEAT. Furthermore, in deterministic crowding, all the population participates in the breeding and niching process. Therefore, NEAT's kill percentage, p_k, is not used. Additionally, since in deterministic crowding species are implicit, the idea of inter-species reproduction is mute; the probability, p_I, is not adopted in DC-NEAT. Consequently, by introducing deterministic crowding in DC-NEAT, we reduce the number of parameters and the implementation complexity.

In deterministic crowding, the niching procedure has a fixed time complexity proportionate to the number of similarity calculations of genome pairs of order $O(N)$, where N is the population size. Whereas, explicit fitness sharing has a worst case time complexity of $O(N^2)$; this is the case when all individuals in the population do not belong to any of the available species.

5.3.4 Algorithm

Alg. 7 shows the whole process of DC-NEAT, compared to NEAT's algorithm (Alg. 4), we can observe that it is simpler. In fact, it has less input parameters. The order of the mutation and crossover steps is different when compared with NEAT's algorithm. From Alg. 7, we note that the whole population is involved in the procedure of deterministic crowding. As a consequence, all genes in the population have the possibility of surviving, there is no kill-off population fraction. Mutation in DC-NEAT is parameterized by the

same mutation probabilities as NEAT, i.e. p_{an}, p_{ac}, p_w, and p_{gr}. In the same manner as NEAT, DC-NEAT has the same stopping criteria, either the maximum number of generations G_{max} or the fitness objective value γ.

Algorithm 7: Algorithm for Deterministic Crowded NEAT.

Input: Population size N, add-node mutation probability p_{an}, add-connection mutation probability p_{ac}, weight mutation probability p_w, gene re-enable probability p_{gr}, fitness function f, max. number of generations G_{max}, fitness objective γ

Output: Population P

```
Create initial population P of size N of fully connected networks;
Set generation count g = 0;
repeat
    Randomly pair the population P creating R;
    foreach (p1, p2) ∈ R do
        Cross p1 and p2 creating c1 and c2;
        Mutate c1 and c2 producing c1' and c2', using p_an, p_ac, p_w, and p_gr;
        if (d(p1, c1') + d(p2, c2')) ≤ (d(p1, c2') + d(p2, c1')) then
            if f(c1') > f(p1) then
                Replace p1 with c1' in P
            if f(c2') > f(p2) then
                Replace p2 with c2' in P
        else
            if f(c2') > f(p1) then
                Replace p1 with c2' in P
            if f(c1') > f(p2) then
                Replace p2 with c1' in P
    Increment generation counter g;
until best_fitness(P) ≥ γ or g ≥ G_max
```

Chapter 6

Pole Balancing Problem

In this work, we use the pole balancing problem for testing and comparing the performance of NEAT and DC-NEAT. Brownlee in [29] stated: "The pole balancing problem is a pseudo-standard benchmark problem, from the field of control theory and artificial neural networks, for designing and testing controllers on complex and unstable non-linear systems". Additionally, some work has been done to solve the pole balancing problem and its variations by combining ANNs with GAs [30, 31, 32]. In particular, NEAT has been used to solve the pole balancing problem and its variations [27, 30, 33]. Therefore, we set the one pole balancing problem as the case of study to test the proposed DC-NEAT and compare its performance with NEAT.

6.1 System Description

Brownlee in [29] offers a canonical description of the one pole balancing problem. The one pole balancing problem is a feedback control system, part of the output is used as an input to the system. The system can be divided in two parts: the *controller* and the *pole system*. The controller excites the pole system and receives the state variables of the system as input. The state variables are used to decide the excitation of the pole system. Thus, the pole system and the controller form a closed loop. This system has as objective to balance the pole, preventing it to fall.

The pole system is constituted by a cart on a track with a pole connected to it by a hinge, see Fig. 6.1. The hinge allows the pole to move, changing the value of the angle θ measured from the vertical axis clockwise. The pole is balanced, if the value of θ is on certain range. In the figure, we can observe that the track is delimited by the values of $-x_{min}$ and x_{max}. The cart can only move horizontally and between the delimited area. It should be noted that here we use an ideal model, thus sources of noise like friction and wind are not taken into account.

The state of the pole system is given by the variables:

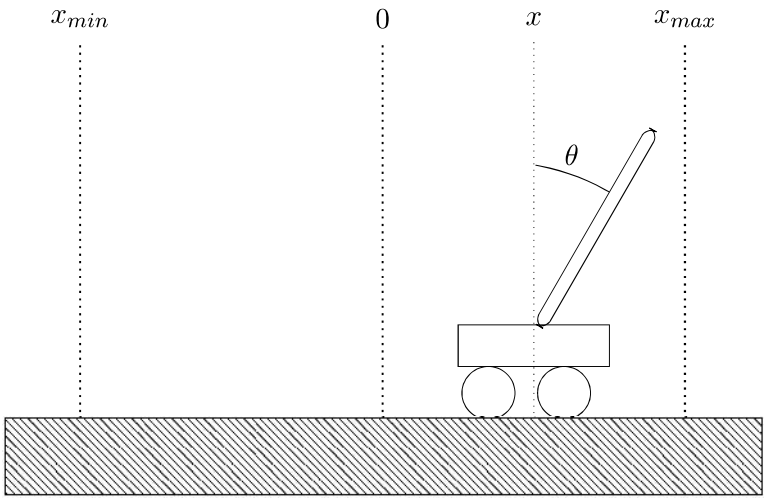


Figure 6.1: One pole balancing system.

- θ, the pole angle;
- $\dot{\theta}$, the angular velocity of the pole;
- x, the position of the cart with respect to the center of the track; and
- $\dot{x}$, the velocity of the cart.

The state of the system can be calculated by solving

$$\ddot{\theta}_t = \frac{g \sin\theta_t - \cos\theta_t \frac{F_t - m_p l \dot{\theta}_t^2 \sin\theta_t}{m_c + m_p}}{l[\frac{4}{3} - \frac{m_p \cos^2\theta_t}{m_c + m_p}]} \tag{6.1}$$

and

$$\ddot{x}_t = \frac{F_t + m_p l[\dot{\theta}_t^2 \sin\theta_t - \ddot{\theta}_t \cos\theta_t]}{m_c + m_p}, \tag{6.2}$$

where $\ddot{\theta}_t$ and $\ddot{x}_t$ are the acceleration of both, the pole's angle and the cart's position, the F_t is signed magnitude of the applied force to the cart parallel to the track, m_p and m_c are the masses of the pole and the cart, l is the distance from the hinge and the center of mass of the pole, and g is the gravitational acceleration $-9.81 m/s^2$. For all quantities in the system, we use the international system of units.

The state of the system is solved using Euler's method and equations (6.1) and (6.2). Time is discretized by $k \in \mathbb{N}$, with a time step, Δt, equals to 0.002s, thus we have $t_k = k\Delta t$ and $x_k \approx x_{t_k}$. Therefore, the position and velocity of the cart and the pole's angle and its angular velocity are approximated by the rules

$$\begin{aligned} \dot{x}_t &= \dot{x}_{t-1} + \Delta t \ddot{x}_{t-1} \\ x_t &= x_{t-1} + \Delta t \dot{x}_{t-1}, \end{aligned} \tag{6.3}$$

$$\begin{aligned}\dot{\theta}_t &= \dot{\theta}_{t-1} + \Delta t \ddot{\theta}_{t-1} \\ \theta_t &= \theta_{t-1} + \Delta t \dot{\theta}_{t-1}.\end{aligned} \tag{6.4}$$

Thus, the current state of the system depends explicitly on the previous state of the system. For example, the pole's angle, θ_t, depends on the previous pole's angle, θ_{t-1}, and the previous pole's angular velocity, $\dot{\theta}_{t-1}$. Consequently, to simulate the pole balancing system, we need to set the values of the initial state, i.e. x_0, $\dot{x}_0$, θ_0, and $\dot{\theta}_0$, the initial values of the cart's acceleration and the pole's angular acceleration , $\ddot{x}_0$ and $\ddot{\theta}_0$, are calculated from these initial values, the system parameters and the initial signed magnitude of the applied force, F_0.

The angle is bounded by the values of θ_{min} and θ_{max}. If the angle, θ_t, is outside this bounded region, the simulation is ended. If the cart's position, x_t, is not inside the region $[-x_{min}, x_{max}]$, the simulation also ends. The force exerted to the cart must be non-zero and of constant magnitude, thus only the force direction is changed.

The controller observes the state of the pole system and exerts the force to the pole system. The controller uses the state variables of the system (i.e. θ_t, $\dot{\theta}_t$, x_t, and $\dot{\theta}_t$) to decide the sign of F_t. In particular, F_t can be either be $F_t = -|F_0|$ or $F_t = |F_0|$, where F_0 is the initial signed magnitude of the applied force.

6.2 Experimental Setup

The experimental setup is composed of the *pole balancing simulation system* and the *controller recurrent neural network*. The pole balancing simulation system uses equations (6.1), (6.2), (6.3) and (6.4) to simulate the state of the pole balancing system from an initial state. Whereas, the controller recurrent neural network receives the pole balancing system's state, from the simulation system, to decide the direction of F_t. The simulation system receives this value and simulates the new state of the pole balancing system.

Two experimental configurations will be used, one called the *full-state pole balancing problem*, and the other one called the *half-state pole balancing problem*. In the full-state pole balancing, the observable state is given by all the state variables for the pole's angle and the cart position (θ_t and x_t) and their respective velocities ($\dot{\theta}_t$ and $\dot{x}_t$). In the half-state pole balancing, only the angle (θ_t) and position (x_t) are observable by the controller.

The controller of the system is obtained using either DC-NEAT or NEAT, each individual in the population of either NEAT or DC-NEAT acts as the controller of the pole balancing system. The fitness of each individual is calculated as the number of time steps, k, for which the pole is balanced and the cart's position is between bounds, in other words, the pole's angle, θ_t, is between $[\theta_{min}, \theta_{max}]$ and the cart's position, x_t, is between $[-x_{min}, x_{max}]$.

An individual of DC-NEAT, in this case, represents a recurrent neural network which

has as input the state of the pole balancing system and outputs a real value, o_t. And, since all neurons on the network will use the sigmoid as their activation function,

$$\xi(h) = \frac{1}{1 + e^{-h}}. \tag{6.5}$$

The value o is then transformed by the function $\Phi : \mathbb{R} \to \mathbb{R}$ to the signed magnitude, F_t. This function is defined as

$$\Phi(o_t) = \begin{cases} F_0 & o_t \geq 0.5 \\ -F_0 & o_t < 0.5 \end{cases}, \tag{6.6}$$

where F is the force magnitude and it is set as a parameter. Here, the value 0.5 was chosen to discriminate between F_0 and $-F_0$ as $\xi(0)$ is 0.5. Thus, we say $F_t = \Phi(o_t)$.

Chapter 7

Results

In total two configurations of the pole balancing system were used, the full state and the half state, to check DC-NEAT. The pole balancing systems were parameterized by the values shown in Table 7.1. In both configurations, the pole balancing system started with the cart centered in $x = 0$ and the pole's angle $\theta_0 = 0$, all other quantities had initial value zero: $\dot{\theta}_0 = 0$, $\ddot{\theta}_0 = 0$, $\dot{x}_0 = 0$, and $\ddot{x}_0 = 0$, except for the initial applied force which had an initial value of 10 N, $F_0 = 10\text{N}$. Since the pole balancing system can be balanced by a network for an infinitely amount of time, a cut-off value of 200000 time steps was chosen to stop the simulation. Thus, the maximum fitness value that a network can have is 200000.

Parameter	Value
Cart's mass, m_c	1.0 kg
Pole's mass, m_p	0.1 kg
Pole's length, l_p	0.5 m
Gravity, g	9.82 m/s^2
Force magnitude, F	10 N
Minimum angle, θ_{min}	$-15.0°$
Maximum angle, θ_{max}	$15.0°$
Minimum position, x_{min}	-2.0 m
Maximum position, x_{max}	2.0 m
Time delta, Δt	0.002 s

Table 7.1: Parameter values used for all pole balancing system configurations.

7.1 Full state pole balancing problem

As noted in Chapter 6, the full state configuration of the pole balancing problem is the one where the controller observes the full state of the pole balancing system. Therefore,

the initial population was conformed by networks with five neurons, four input neurons and one output neuron. For this configuration DC-NEAT was run for 100 generations and with a population size of 100 networks. Also, the mutation probabilities were fixed as shown in Table 7.2. Fig. 7.1 shows the evolution of the population's best fitness

Probability	Value
p_{an}	0.1
p_{ac}	0.4
p_{w}	0.6
p_{gr}	0.3

Table 7.2: Mutation probabilities used in DC-NEAT for the full state configuration of the pole balancing problem.

value through the generations for all runs done. The best fitness value of the population at each generation either increases or remains constant. Thus, as observed in all runs the best fitness value of the population has an upward trend. Of all the runs done, five runs achieved the cut-off value of 200000 time steps. In other words, at least the best individual in the population can balance the system for 200000 time steps (approximately 6-7 minutes) or more. Additionally, we can observe for all the runs that achieved this cut-off value, the change was abrupt.

Fig. 7.2 shows the evolution of the average fitness value of the population. The

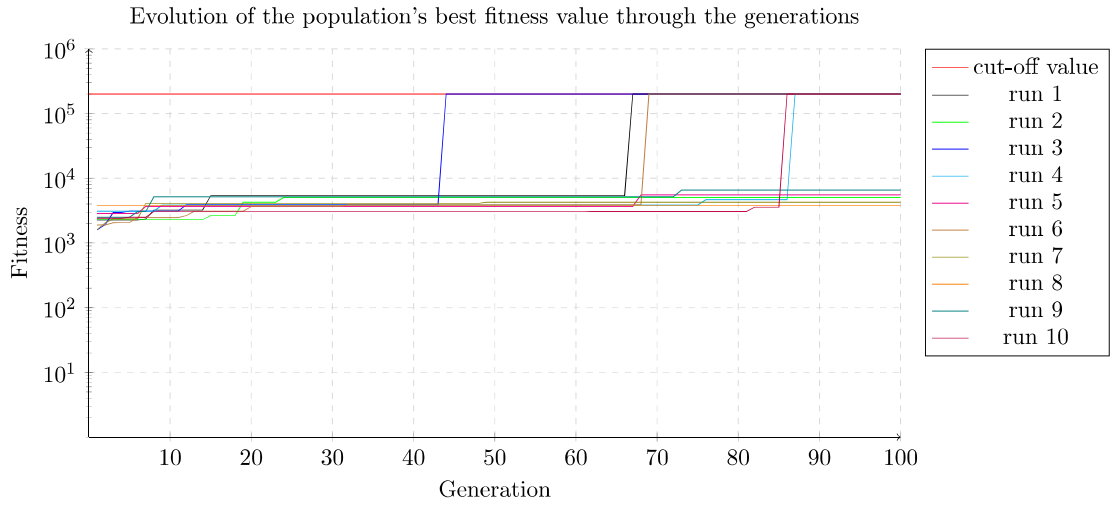


Figure 7.1: Evolution of the population's best fitness value through the generations for ten runs of DC-NEAT for the full state configuration of the pole balancing problem.

average fitness has an upward trend, in the same manner as the population's fitness value. Besides, we can observe that for all runs the population's average fitness behaves

in the same manner.

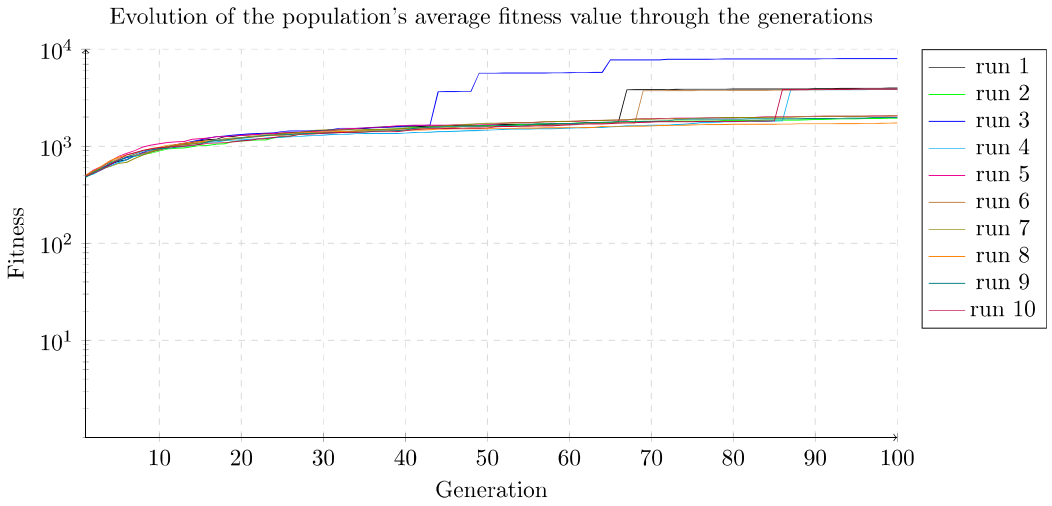


Figure 7.2: Evolution of the population's average fitness value through the generations for ten runs of DC-NEAT for the full state configuration of the pole balancing problem.

Fig. 7.3 shows an example of the architecture of a cut-off network, which is a network that balances the system for at least 200000 time steps. In this network, the input nodes are nodes 1, 2, 3, and 4; and the output node is node 5. As we can observe, the network has two recurrent connections, from the input node 2 to input node 3 and a loop connection from input node 3 to itself. Besides, the network also has a new node, hidden node 6.

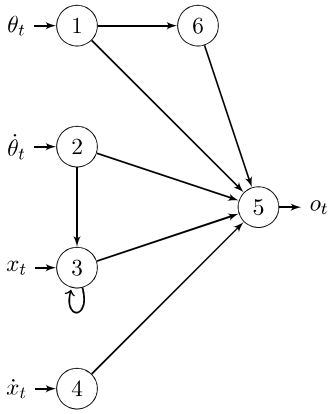


Figure 7.3: Example of the architecture of a cut-off network.

7.2 Half state pole balancing problem

The half state configuration of the pole balancing problem is the one where the controller observes only two variables, the pole's angle and the cart's position. Therefore, the initial population was conformed by networks with three neurons, two input neurons

and one output neuron. This configuration is more difficult to solve, when compared to the full state configuration, as the controller must internally "calculate" the two missing state values, the pole's angular velocity and cart's velocity. Therefore, DC-NEAT was run for 500 generations and with a population of size 200 networks. Moreover, the mutation probabilities were increased to promote the addition of new structure (nodes and connections) to the population, these new mutation probabilities can be found in Table 7.3. In the same manner as the full state configuration, 10 runs of DC-NEAT were

Probability	Value
p_{an}	0.6
p_{ac}	0.7
p_{w}	0.6
p_{gr}	0.3

Table 7.3: Mutation probabilities of DC-NEAT for the half state configuration of the pole balancing problem.

done. In Fig. 7.4, we observe the evolution of the population's best fitness value for all runs. Only two runs achieve the cut-off value, namely run 5 and run 10. However, for all runs the best fitness value of the population has an upward trend. Despite that sometimes the best fitness value appears to have stagnated, given enough generations the value increases. For some runs in Fig. 7.4, we observe large period of time where the population's best fitness value, for example for run 1, from generation 16 to generation 146, the fitness value is fixed to the value 6301.

The evolution of the population's average fitness is shown in Fig. 7.5. We can observe that for all runs the population's average fitness has an ascending trend. Also, the average fitness increases with less jumps when compared to the best fitness value.

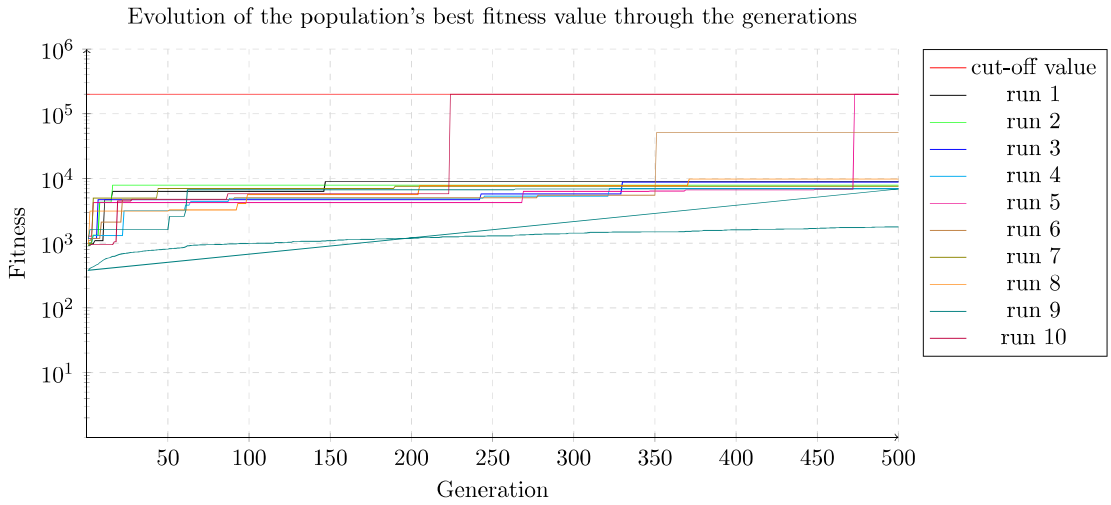


Figure 7.4: Evolution of the population's best fitness value through the generations for ten runs of DC-NEAT for the half state configuration of the pole balancing problem.

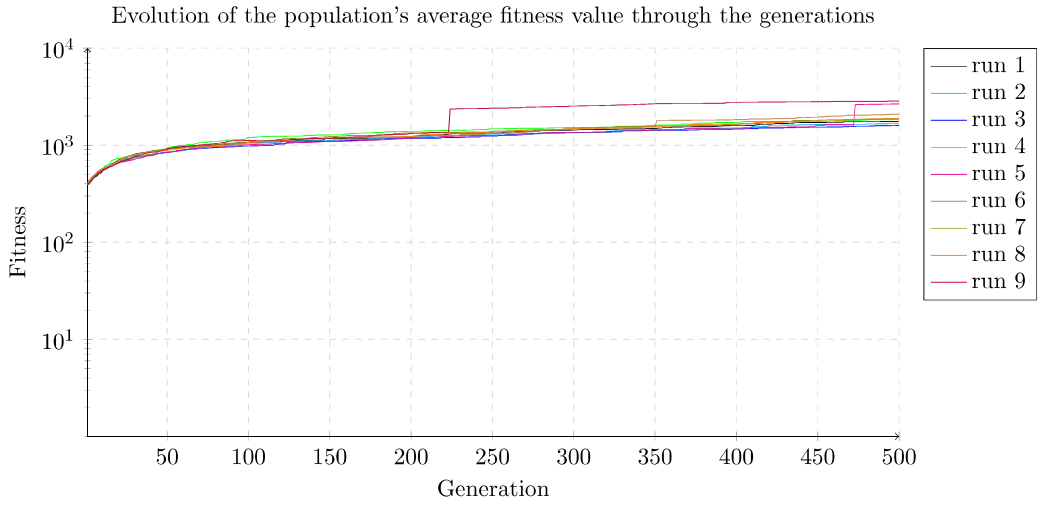


Figure 7.5: Evolution of the population's average fitness value through the generations for ten runs of DC-NEAT for the half state configuration of the pole balancing problem.

An example of the architecture of a cut-off network for the half state configuration is shown in Fig. 7.6. This network's architecture is more complex than the one for the full state configuration, in particular, this architecture has more hidden nodes and connections. This architecture has some notable features. Node 6, 9 and 10 act as bias for nodes 5, 7, 1 and 4, this due to the activation scheme used for the recurrent neural networks. Since, all neurons, in recurrent neural networks, start with state 0 and neurons 6, 9, and 10 do not receive information from other neurons, the output of this neurons will always be 0.5, $\xi(0) = 0$. Node 8 does not have any outgoing connection, in other

words there is no other node in the network that receives information from node 8. Thus, we can safely remove this node from the network, and the network's behavior would not change.

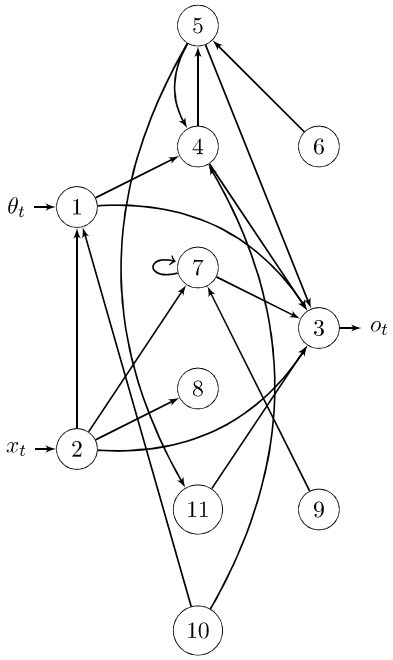


Figure 7.6: Example of the architecture of a cut-off network for the half configuration of the pole balancing problem.

7.3 NEAT vs DC-NEAT

The full state configuration of the pole balancing problem was used to compare NEAT and DC-NEAT. In this experiment both NEAT and DC-NEAT start with different initial populations. NEAT was run with the same population size and mutation probabilities as DC-NEAT (see Table 7.2); both were run for 100 generations. Additionally, to reduce the number of meta parameters, NEAT's kill percentage, p_k, crossover percentage, p_c, and interspecies probability, p_I, were set to 1. Therefore, the whole population participates in the selection and breeding process, likewise, individuals from different species are always allowed to reproduce. Fig. 7.7 shows the evolution of the population's best fitness value of 5 runs of NEAT and 5 runs of DC-NEAT. For all runs of NEAT the population's best fitness value increases and decreases in an erratic manner. In contrast with DC-NEAT all runs of NEAT achieve the cut-off value of 200000, however, all runs lose it afterwards. In comparison, when DC-NEAT achieves an individual with fitness equal to the cut-off value, it remains in the population. Nonetheless, NEAT tends to find an individual with cut-off fitness faster than DC-NEAT, for example run 3 of NEAT achieves it at generation 10.

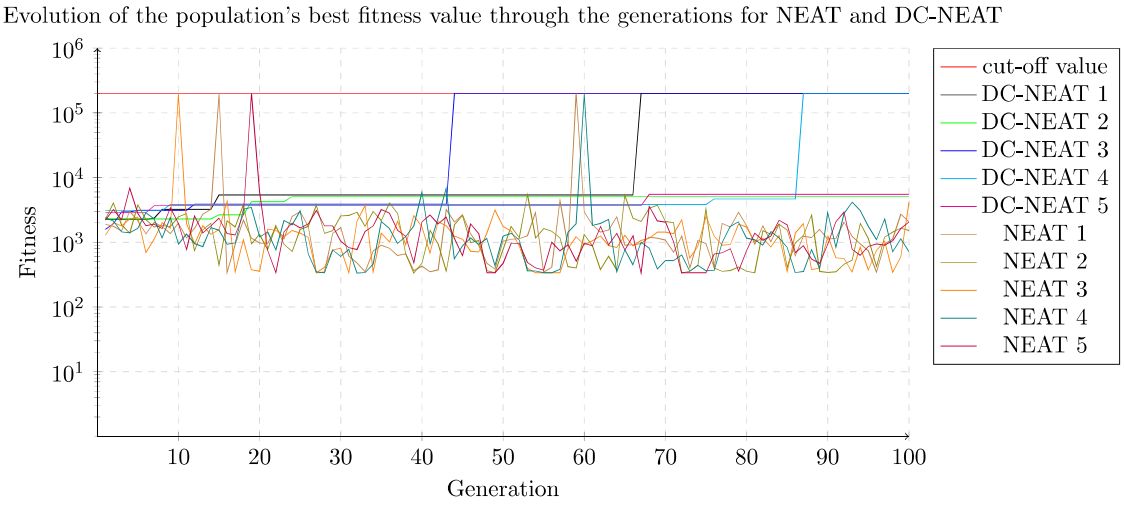


Figure 7.7: Comparison of the evolution of the population's best fitness value of NEAT and DC-NEAT for the full state configuration of the pole balancing problem.

In figure 7.8, we observe the different behavior of the evolution of the population's average fitness value for both NEAT and DC-NEAT. As with the population's best fitness value, NEAT's population average fitness increases and decreases in a erratic manner. In contrast, DC-NEAT's population average fitness value has a more deterministic behavior, in fact the population's average fitness only increases or remains constant.

A comparison of the population's best individual complexity and the population's

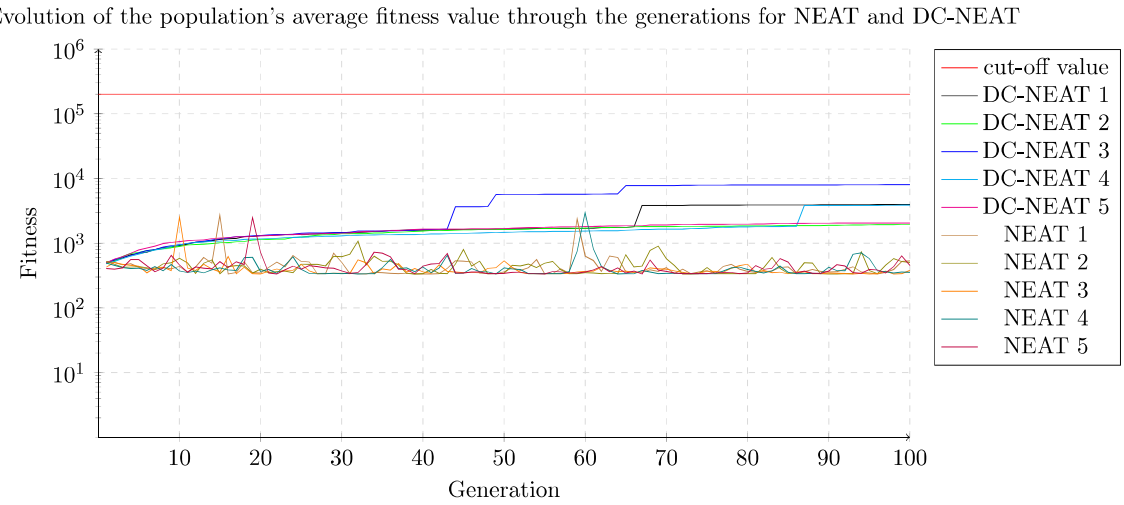


Figure 7.8: Comparison of the evolution of the population's average fitness value of NEAT and DC-NEAT for the full state configuration of the pole balancing problem.

average complexity is given in Fig. 7.9 and Fig. 7.10, the complexity of an individual (network) is defined as the number of enabled connection genes of the individual's genome. Furthermore, the generations at which NEAT and DC-NEAT obtains an individual with fitness equal to the cut-off value are marked in both figures. In Fig. 7.9, we can note that for NEAT, the complexity of the population's best individual grows linearly with the generations. In contrast, the complexity of the population's best individual remains fairly constant for DC-NEAT. The population's average network complexity for NEAT and DC-NEAT behaves in a similar manner as can be seen 7.10.

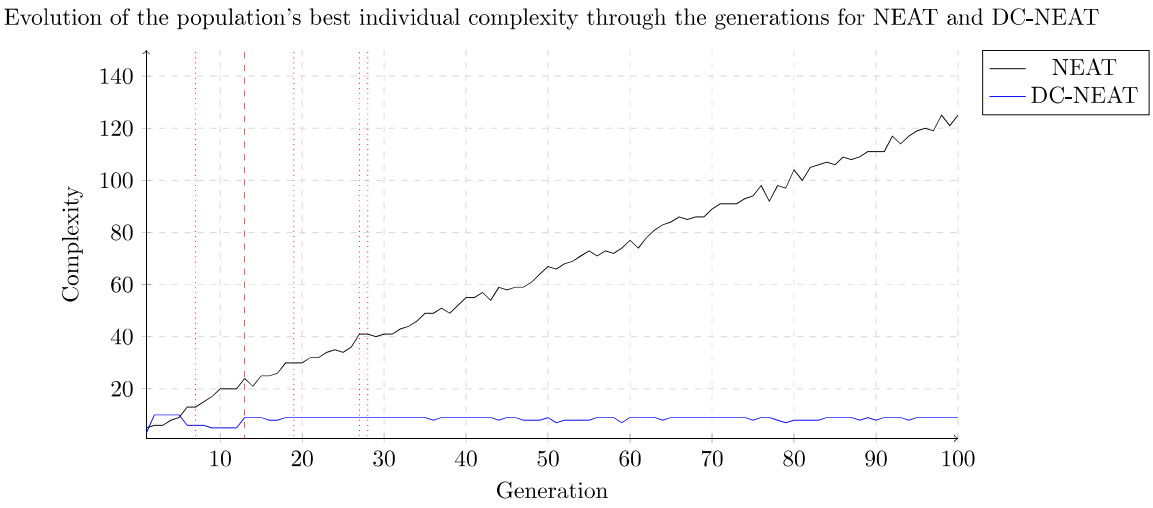


Figure 7.9: Comparison of the evolution of the population's best individual complexity of NEAT and DC-NEAT. Generations at which NEAT and DC-NEAT has an individual with fitness 200000 are marked with red dotted lines (NEAT) and red dashed line (DC-NEAT).

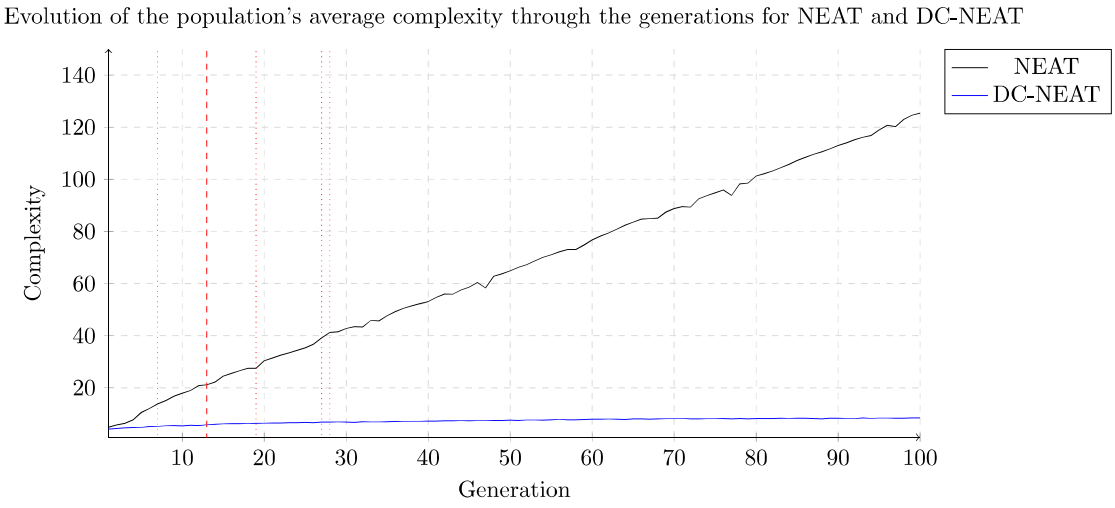


Figure 7.10: Comparison of the evolution of the population's average complexity of NEAT and DC-NEAT. Generations at which NEAT and DC-NEAT has an individual with fitness 200000 are marked with red dotted lines (NEAT) and red dashed line (DC-NEAT).

Chapter 8

Discussion

In chapter 7, we showed the behavior of DC-NEAT on the pole balancing for both configurations, full state and half state. For the full state configuration DC-NEAT achieved the cut-off value in less than 100 generations. Whereas, for the half state configuration DC-NEAT requires at least 500 generations to achieve the cut-off value. This difference in the number of generations is due to the difficulty present in the half state configuration which then requires a more extensive search.

In DC-NEAT, both the population's best fitness value and average fitness can only remain constant or increase, these values will never decrease in a run of DC-NEAT. DC-NEAT uses deterministic crowding as its niching method, in this method, parents are only replaced in the population by their corresponding offspring if the corresponding offspring has a better fitness value than the parent. Therefore, whenever a replacement occurs the average fitness of the population increases. Likewise, if no replacement occurs, the population's average fitness remains constant. Additionally, in DC-NEAT parents can survive in the population for multiple generations. Finally, individuals with the best fitness value of the population are kept until an individual with better fitness is found.

DC-NEAT runs are parameterized by the population size and the mutation probabilities, the maximum number of generations is only used to control the run time. The population size determines the amount of parallel searches that DC-NEAT does over the genome space, G. The decision of the population size should balance the computational cost of larger populations with the increased search coverage of G that is provided by the larger population. Mutation probabilities determine the rate at which new structure genes (node and connection genes) enter the population. Large mutation probabilities introduce noise to the population and might even reduce DC-NEAT to a random search over G. Whereas, small mutation probabilities reduce the rate at which the gene pool of the population changes thus increasing the number of generations necessary to achieve target fitness values. Therefore, a balance is required when deciding the mutation probabilities.

Regarding the stagnation of the population in DC-NEAT, from Fig. 7.1 and Fig. 7.5

that for long periods of time (100-200 generations) the population's best fitness value does not change, intuition might suggest that the population have stagnated. However, if we examine the same periods of time, where the population have "stagnated", in Fig. 7.2 and Fig. 7.5 we discover that in these periods of time the population's average fitness has increased. In these periods of time, new individuals have been produced that are fitter and have replaced their respective parents in the population, we can say that the population has continue to evolve. Therefore, whenever we are examining population stagnation in DC-NEAT, we should pay attention to the population's average fitness.

When comparing NEAT and DC-NEAT, using the pole balancing problem, in its full state configuration, we noticed that the complexity of individuals in the population of NEAT grows linearly. Whereas, in DC-NEAT the complexity lingers around a value less than 10. This indicates that new structure in DC-NEAT is added more slowly than in NEAT. Because, individuals in DC-NEAT are only replaced if more fit individuals are found, and new individuals are only produced from the current population. In other words, the genetic pool (structures) tends to remain fairly constant. This behavior of DC-NEAT can be viewed as a double edge sword, the networks produced would be less complex than those produced by NEAT. However, DC-NEAT might need more generations to produce these networks.

In contrast to NEAT, DC-NEAT does not have a fitness proportionate mechanism like stochastic universal sampling or roulette-wheel selection. All individuals in the population participate once in the breeding process, unlike NEAT, where a highly fit individual could be selected multiple times to participate in the breeding process denying other individuals the opportunity of passing on their genes. Thus, in NEAT in order to reduce the possibility of individuals overcoming the population, the population is partitioned into explicit species. However, in DC-NEAT this explicit partition is not required, as individuals are not pitched against each other, by the fitness proportionate selection method, but against their offspring, in the tournaments.

To measure the computational load of NEAT and DC-NEAT, the run time (measured in seconds) was first considered. However, it was determined, that it is not a good measure for computational load in this case. As, the run time depends mainly on the simulation of the pole balancing system, fitter individuals have a longer simulation time and thus increases the total run time. Therefore, in this case to analyze the computational load the number of generations was used instead of the run time.

Chapter 9

Conclusion

This thesis aimed to create a new algorithm for evolving a population of artificial neural networks based on the algorithm Neuro Evolution of Augmenting Topologies. Based on a theoretical analysis of NEAT, we identified two problems with NEAT, the similarity measurement is not a distance and explicit fitness sharing requires the fixation of the similarity threshold. We demonstrated that NEAT's similarity measurement violates the triangular inequality, thus concluding that it is not a distance. Also, we pointed out that explicit fitness sharing has a worst case time complexity of $O(N^2)$, where N is the population size. Therefore, we proposed the method, Deterministically Crowded - Neuro Evolution of Augmenting Topologies, which keeps NEAT's genome encoding and mutation operators, and replaces explicit fitness sharing with deterministic crowding. In addition, NEAT's similarity measurement was replaced by a new measurement that leverages the matrix representation of artificial neural networks when viewed as weighted directed graphs and the maximum norm. Deterministic crowding was chosen as DC-NEAT's niching method because it has simple implementation and does not require the similarity threshold. Also, it has a better time complexity than explicit fitness sharing, deterministic crowding has a fixed time complexity of $O(N)$.

Using the pole balancing problem to examine the behavior of DC-NEAT when solving problems with dynamical systems. We found that networks produced could balance the system for at least 6-7 minutes where obtained by DC-NEAT in 100 generations (for the full state configuration) and 500 (for the half state configuration) achieving the pole balancing problem objective. Taking into account the simplicity of the implementation and the reduction of the number of parameters with respect to NEAT, we can view DC-NEAT as a viable alternative to NEAT.

From the comparison of the behavior of NEAT and DC-NEAT when solving the full state configuration of the pole balancing problem, we understand that because DC-NEAT uses deterministic crowding as its niching method, DC-NEAT takes more generations to solve the problem than NEAT. However, whenever a solution (individual) is found in DC-NEAT the solution stays in the population through the generations, in contrast with

NEAT where the solution is lost through the generations. Additionally, we discovered that as result of DC-NEAT favoring small changes in the population that increases the population's total fitness, in terms of the complexity (number of connections) of the obtained networks, DC-NEAT is an improvement over NEAT.

As future work, we should compare the performance of DC-NEAT and NEAT for multiple problems (dynamic and static). Additionally, the niching capabilities of both NEAT and DC-NEAT should be analyzed, however for this a metric for measuring diversity in the case of artificial neural networks must be proposed. Finally, we recommend extending either NEAT or DC-NEAT to work with convolutional artificial neural networks. These type of networks tend to have complex architectures and the design is done by trial-and-error, which presents a problem since the training of these networks tends to be computationally costly.

Bibliography

[1] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, Jul. 1989.

[2] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[3] Y. Leung, Y. Gao, and Z.-B. Xu, "Degree of population diversity - a perspective on premature convergence in genetic algorithms and its markov chain analysis," *IEEE Transactions on Neural Networks*, vol. 8, no. 5, pp. 1165–1176, Sep. 1997.

[4] S. W. Mahfoud, "Niching methods for genetic algorithms," Ph.D. dissertation, Champaign, IL, USA, 1995, uMI Order No. GAX95-43663.

[5] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *CoRR*, vol. abs/1109.3627, 2011.

[6] J. E. Baker, "Reducing bias and inefficiency in the selection algorithm," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1987, pp. 14–21.

[7] A. Toffolo and E. Benini, "Genetic diversity as an objective in multi-objective evolutionary algorithms," *Evolutionary Computation*, vol. 11, no. 2, pp. 151–167, 2003.

[8] B. Sareni and L. Krahenbuhl, "Fitness sharing and niching methods revisited," *IEEE Transactions on Evolutionary Computation*, vol. 2, no. 3, pp. 97–106, Sep. 1998.

[9] D. E. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1987, pp. 41–49.

[10] K. A. De Jong, "An analysis of the behavior of a class of genetic adaptive systems." Ph.D. dissertation, Ann Arbor, MI, USA, 1975, aAI7609381.

[11] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[12] I. Basheer and M. Hajmeer, "Artificial neural networks: fundamentals, computing, design, and application," *Journal of Microbiological Methods*, vol. 43, no. 1, pp. 3 – 31, 2000, neural Computting in Micrbiology.

[13] K. C. Luk, J. E. Ball, and A. Sharma, "An application of artificial neural networks for rainfall forecasting," *Math. Comput. Model.*, vol. 33, no. 6-7, pp. 683–693, Mar. 2001.

[14] A. Khashman, "Neural networks for credit risk evaluation: Investigation of different neural models and learning schemes," *Expert Syst. Appl.*, vol. 37, no. 9, pp. 6233–6239, Sep. 2010.

[15] A. C. Tsoi and A. Back, "Discrete time recurrent neural network architectures: A unifying review," *Neurocomputing*, vol. 15, no. 3, pp. 183 – 223, 1997, recurrent Neural Networks.

[16] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85 – 117, 2015.

[17] S. Scardapane, M. Scarpiniti, D. Comminiello, and A. Uncini, *Learning Activation Functions from Data Using Cubic Spline Interpolation.* Cham: Springer International Publishing, 2017, pp. 73–83.

[18] L. Vecci, F. Piazza, and A. Uncini, "Learning and approximation capabilities of adaptive spline activation function neural networks," *Neural Networks*, vol. 11, pp. 259–270, 04 1998.

[19] S. Guarnieri, F. Piazza, and A. Uncini, "Multilayer feedforward networks with adaptive spline activation function," *IEEE Transactions on Neural Networks*, vol. 10, no. 3, pp. 672–683, May 1999.

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–, Oct. 1986.

[21] D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'89. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 762–767.

[22] L. C. Jain and L. R. Medsker, *Recurrent Neural Networks: Design and Applications*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1999.

[23] F. J. Pineda, "Generalization of back-propagation to recurrent neural networks," *Phys. Rev. Lett.*, vol. 59, pp. 2229–2232, Nov 1987.

[24] L. Fausett, Ed., *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[25] X. Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423–1447, Sept 1999.

[26] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing neural networks using genetic algorithms," in *Proceedings of the Third International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 379–384.

[27] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evol. Comput.*, vol. 10, no. 2, pp. 99–127, Jun. 2002.

[28] K. Deb and D. E. Goldberg, "An investigation of niche and species formation in genetic function optimization," in *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 42–50.

[29] J. Brownlee, "The pole balancing problem: a benchmark control theory problem," 2005.

[30] F. J. Gomez and R. Miikkulainen, "Solving non-markovian control tasks with neuroevolution," in *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 1356–1361.

[31] B. Maričić, "Genetically programmed neural network for solving pole-balancing problem," in *Artificial Neural Networks*, T. KOHONEN, K. MÄKISARA, O. SIMULA, and J. KANGAS, Eds. Amsterdam: North-Holland, 1991, pp. 1273 – 1276.

[32] C. Igel, "Neuroevolution for reinforcement learning using evolution strategies," in *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, vol. 4, Dec 2003, pp. 2588–2595 Vol.4.

[33] D. Pardoe, M. Ryoo, and R. Miikkulainen, "Evolving neural network ensembles for control problems," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 1379–1384.