DTU Compute
Department of Applied Mathematics and Computer Science

Implementation and Evaluation of Recent Neuroevolution Algorithms

Master Thesis

Implementation and Evaluation of Recent Neuroevolution Algorithms

Master Thesis
June, 2023

By
Samy Haffoudhi

Cover photo: Vibeke Hempler, 2012
Published by: DTU, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Building 324, 2800 Kgs. Lyngby Denmark
`www.compute.dtu.dk`
ISSN: [0000-0000] (electronic version)
ISBN: [000-00-0000-000-0] (electronic version)

ISSN: [0000-0000] (printed version)
ISBN: [000-00-0000-000-0] (printed version)

Approval

This thesis has been prepared over five months at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark, DTU, in partial fulfilment for the degree Master of Computer Science & Engineering. The project has been supervised by Prof. Carsten Witt and corresponds to 30 ECTS points.

It is assumed that the reader has fundamental knowledge of computer science.

Samy Haffoudhi - s222887

.....

Signature

.....

Date

Abstract

Neuroevolution is a method for optimizing the topology, weights or other hyperparameters of neural networks by means of evolutionary algorithms. This technique is more general than traditional white-box gradient-based approaches, and can therefore be applied to a wider range of problems. It has been studied in research for decades and has been successfully applied to problems such as artificial life, evolutionary robotics and continuous domains of reinforcement learning. In this thesis, we are interested in the development of a framework that implements neuroevolution algorithms, and that is used to evaluate these algorithms on a selection of benchmark problems. Algorithms and benchmarks were collected from the state of the art in applied and theoretical research in the field of neuroevolution. The framework, implemented in Rust, is invoked through a command-line interface, and allows for a visualization of key problem characteristics and the evolution process, through a graphical user interface. The selected algorithms and benchmarks are presented in detail. Results collected from the conducted experiments are analyzed, discussed and used to provide a series of guidelines for the choice of algorithms and parameters with respect to problem classes.

Acknowledgements

TODO

Contents

Chapter 1

Introduction

Neuroevolution is a subfield of artificial intelligence which consists in the evolution of ANNs (artificial neural networks). ANNs are traditionally trained using gradient-based methods, such as stochastic gradient descent. Over the years, these methods have been successfully applied to a variety of problems, such as image classification, speech recognition and natural language processing. Such problems allow for supervised learning, where ANNs are trained on a dataset of input-output pairs. However, there is a class of problems for which supervised learning is not applicable, where instead of input-output pairs, only a measure of performance is available, and where such approaches are not applicable. In addition, the performance of ANNs is also heavily impacted by their architectures. However, the design of ANNs architecture is a complex and time-consuming task, which is topically done by hand, based on experience and trial-and-error.

Neuroevolution, on the other hand, as a more general approach, can in particular be applied to this other class of problems, as well as to the design of ANNs architectures. This method is based on the use of evolutionary algorithms, which are inspired by the process of natural selection. These algorithms maintain a population of individuals, which are mutated and recombined to evolve towards optimal solutions. They have shown success for black-box problems and have successfully been applied to a wide range of engineering problems over the years.

The field of neuroevolution has been researched for over 40 years, hence many different algorithms, benchmarks and applications have been proposed. As a matter of fact, neuroevolution encapsulates algorithms with different goals, such as the optimization of the weights of a fixed topology, the evolution of a network topology alongside the use of gradient-based methods for optimizing weights, the evolution of both the topology and weights, as well as the evolution of hyperparameters or reinforcement learning policies. However, in this thesis, we are interested in approaches relying entirely on neuroevolution, without the need for gradient-based methods, for evolving neural network parameters, using evolved or fixed topologies. In addition to these approaches, Various benchmark problems covering different problem classes sch as classification, continuous control or game planning, have also have been proposed in the applied and theoretical literature for evaluating and comparing the different algorithms.

The focus of this thesis is the development of a framework that implements a selection of neuroevolution algorithms. The framework is used to evaluate and compare the algorithms on a selection of benchmarks. The framework is implemented in Rust. It allows for the visualization of the problems and the solution process through a graphical user interface.

The algorithms can be run and tested through a command line interface, in order to allow for the execution of experiments and the collection of results, based on passed-in parameters and configurations.

Algorithms and benchmarks implemented in the framework were selected from the state-of-the art in the theoretical and applied research in the field of neuroevolution, with a particular focus on recent algorithms and becnhmarks proposed in 2023 and 2024 in the neuroevolution theory literature. In addition to these proposals, NEAT (NeuroEvolution of Augmenting Topologies), a classic algorithm in the field, and the use of evolution strategies with CMA-ES (Covariance Matrix Adaptation Evolution Strategy) for the evolution of weights, achieving state-of-the-art results, were also considered. Regarding the benchmarks, in addition to the simple two-dimensional binary classification benchmarks from the considered theory litterature and the XOR problem, the classic double pole balancing problem and the *cancer1* classification problem were also implemented.

1.0.1 Overview

The ordering of chapters in this report follows the chronological order of work performed for this project. The report is structured as follows:

Chapter 2 provides an overview of the background and context of the project. It introduces the notion of artificial neural networks, the main probme classes, evolutionary algorithms and neuroevolution.

Chapter 3 gives the results of a literature review conducted on neuroevolution algorithms and benchmarks. It presents the state-of-the-art in the field, with a particular focus on recent algorithms and benchmarks proposed in the neuroevolution theory literature from 2023 and 2024. Moreover, it also presents the selection of algorithms and benchmarks that were considered in this thesis and implemented in the framework, motivating the choices and discussing these selected algorithms and benchmarks in further detail.

Furthermore, Chapter 4 describes the process of designing and implementing the framework for running neuroevolution algorithms on benchmark problems, gathering statistics on these runs, and allowing for a visualization of the problems and solution process through a graphical user interface. This is done by first specifying the goals and requirements of the framework, followed by a walkthrough of the main design and implementation points of the framework.

Moreover, Chapter 5 lays out the results collected from running experiments using the implemented framework. Results are presented for the selection of algorithms and benchmarks, and are discussed and compared in detail. Based on these results, a collection of observations, hypothesises and guidelines for algorithms and parameters selection, given the problem at hand, are presented. These observations are backed up by statistical tests compiled from additional experiments, which are also summarized in the chapter.

Finally, **??** identifies the limitations of the project and discusses the potential lines of future work for refining and expanding on the results previously presented. The main work and contributions of this project are finally summed up in chapter **??** which concludes the report and reflects on its results.

Chapter 2

Background

2.1 Artificial Neural Networks

Artificial neural networks (ANNs) are a class of machine learning models, which are inspired by biological neural networks. ANNs are composed of interconnected neurons, which are organized in layers. The first layer of an ANN is referred to as the input layer, the last layer is the output layer, and the layers in between are hidden layers. In feed-forward ANNs, nodes in a layer are connected to nodes from the immediately preceding and succeeding layers. The connections between nodes are associated weights. Signal travels from the input layer to the output layer and the output of a node is computed by applying a non-linear activation function to the weighted sum of the inputs. The weights of the connections are typically learned using gradient-based optimization algorithms, such as backpropagation. Given a neuron with inputs $x_1, x_2, \ldots, x_n$ and weights $w_1, w_2, \ldots, w_n$, the output y of the neuron is computed as

$$y = f(\sum_{i=1}^{n} w_i x_i)$$

where f is the neuron's activation function.

An example of a feed-forward neural network is shown in Figure 2.1. This network consists of an input layer with four nodes, a hidden layer with five nodes, and an output layer with one node. The connections between nodes are represented by arrows, and the weights of the connections are not shown.

Activation functions are used to introduce non-linearity in the model, which is important to enable the model to learn complex patterns in the data. Without non-linear activation functions, the model would be limited to learning linear functions, which are not sufficient to model complex relationships in the data. Two commonly used activation functions are the **sigmoid** function and the **rectified linear unit** (ReLU) function. The sigmoid function is defined as $\sigma(x) = \frac{1}{1+e^{-x}}$, it maps the input to the range $[0, 1]$, making it useful for binary classification tasks. The ReLU function is defined as $f(x) = \max(0, x)$, which is a piecewise linear function that maps negative inputs to zero and positive inputs to the input itself. The ReLU function is commonly used in deep learning models, as it has been shown to improve the convergence of the optimization algorithm in deep architectures, by addressing the vanishing gradient problem. The graphs of the sigmoid and ReLU activation functions are shown in Figure 2.2.

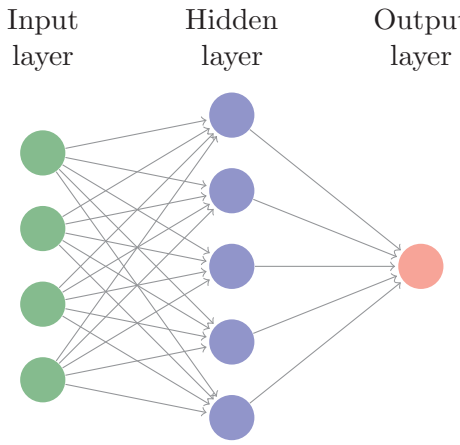


Figure 2.1: Graph representation of an ANN with one hidden layer.

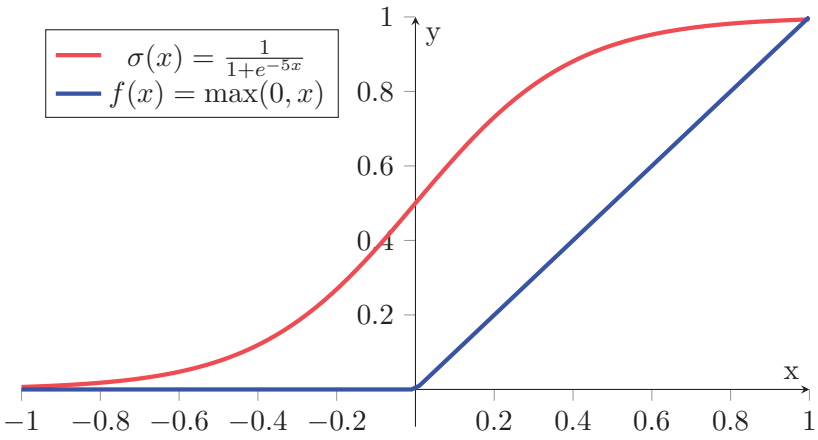


Figure 2.2: Graphs of the sigmoid and ReLU activation functions.

Any function can be approximated by ANNs. Over the years, these models have been applied to a wide range of problems, including classification and reinforcement learning tasks. The process of applying ANNs consists in the design of its architecture, the choice of parameters such as the loss function or parameters for the weigh optimization algorithm, and the optimization of its weight, in order to maximize its performance on a given task.

Many challenges are associated with the use of ANNs, such as the choice of the network architecture and hyperparameters, which can have an significant impact on the performace of the model. Furthermore, the optimization of the network weights using gradient=based algorithms can result in various issues, such as overfitting, which arises when the model performs well on the training data but poorly on unseen data, suboptimal local minima, and vanishing and exploding gradients, when using deep networks.

2.2 Problem classes and learning paradigms

ANNs can be used to solve a wide range of problems, making use of different learning paradigms. This section provides a brief overview of the main classes of problems and learning paradigms in machine learning.

2.2.1 Supervised Learning

Supervised learning is a class of machine learning tasks, where the goal is to learn a mapping from input data to output data, given a dataset of labeled examples. The dataset consists of pairs of input-output data, and the goal is to learn a function which maps the input to the output. The function is typically learned by minimizing a loss function, which depends on the problem at hand, and which measures the difference between the predicted output and the true output. The performance of the model is evaluated using a test set, which is separate from the training set. Supervised learning is used for tasks such as **classification** and **regression** problems. For classification tasks, the output is a discrete class label, while for regression tasks, the output is a continuous value. Famous examples of classification tasks include image classification, where the goal is to classify images into different categories or spam detection, where the goal is to classify emails as spam or not spam. When two classes are involved, the task is referred to as **binary classification**. Different metrics can be used to evaluate the performance of ANNs on classification tasks, such as the accuracy which measures the proportion of correctly classified examples. For regression tasks, metrics such as the mean squared error can be used.

2.2.2 Unsupervised Learning

Unsupervised learning is a machine learning task where the goal is to learn patterns in the data without the use of labeled examples. In other words, no ground-truth is provided from the (training) data and the goal is to learn the underlying structure of the data, such as clusters or a latent representation space. Unsupervised learning is used for tasks such as clustering, which consists in grouping similar examples together, or dimensionality reduction, which consists in reducing the number of features in the data while preserving as much information as possible.

2.2.3 Reinforcement Learning

Reinforcement learning problems can be compared to trial-and-error learning. These problems are composed of three main components: an agent, an environment, and a reward signal. The agent interacts with the environment by taking actions, and is provided with a reward signal which measures how well it is performing.. The agent interacts with the environment by observing the current environment state, and taking actions. It receives a reward signal from the environment, the state is updated and the process is repeated. As the agent interacts with the environment it learns a strategy mapping states to actions, which

is referred to as a policy, with the goal of maximizing its cumulative reward. Reinforcement learning is used for dynamic tasks such as game playing or **control tasks**. There is no labeled data in reinforcement learning, and the agent must learn from its own experience, by exploring the environment and learning from the feedback it receives.

2.3 Evolutionary Algorithms

Evolutionary algorithms (EAs) are a class of optimization algorithms, which are inspired by the process of natural selection. EAs are based on the idea of evolving a population of candidate solutions to a problem, in order to find the best solution. The process of evolution consists in the selection of the fittest individuals, which are then recombined to produce offspring, which are then mutated. The fitness of the offspring is evaluated and the process is repeated for a number of generations until a stopping criterion is met, such as a maximum number of generations or a desired level of performance. At each generation, the population is updated by replacing the worst-performing individuals with the offspring.

Individuals are represented by genotypes, which are encoded in a way that allows the application of genetic operators, such as crossover and mutation, and by phenotypes, which are the actual interpretation of the genotype used as solutions to the problem. The fitness of an individual is evaluated using a fitness function, which measures the quality of the solution.

Mutations are used to introduce diversity in the population, which is important to avoid premature convergence to suboptimal solutions. During mutations, parts of the genotype are modified while other parts remain unchanged, with the goal of performing a local search around the current genotype. The individuals reproduce using crossover, which combines the genotypes of two parents to produce offspring. The offspring inherit parts of the genotypes of the parents. The idea is that combining two good solutions can produce an even better solution.

The general structure of an evolutionary algorithm is shown in Algorithm 1.

Algorithm 1 General structure of an evolutionary algorithm.

```
Initialize population
Evaluate population
while termination criterion not met do
    Select parents
    Recombine parents
    Mutate offspring
    Evaluate offspring
    Select survivors
end while
return best individual
```

It should be noted that evolutionary algorithms often have a variety of parameters that need to be tuned, such as the population size, the mutation rate, the crossover rate, and the selection mechanism. The performance of the algorithm can be sensitive to the choice of these parameters, and finding the right combination of parameters can be a challenging task.

Evolutionary algorithms can be organized into different categories, including:

- **Genetic algorithms (GAs):** Genetic Algorithms are generally applied to optimization problems and follow the principles described above.
- **Genetic programming (GP):** Genetic programming is a variant of genetic algorithms where the individuals are computer programs, which are evolved to solve a problem. Programs are traditionally represented as trees.
- **Evolution strategies (ES):** Evolution strategies are black-box optimization algorithms, which are used to optimize continuous real-valued functions. Unlike other classes of EAs, evolution strategies do not maintain an explit population of individuals, but instead, a parametrized probability distribution used to sample the individuals.

2.4 Neuroevolution

Neuroevolution is a subfield of machine learning, which combines evolutionary algorithms and artificial neural networks. The idea is to evolve the weights, and potentially, the architecture or hyperparameters of ANNs using an EA instead of traditional gradient-based methods. Hence, this approach aims to offer an alternative to the manual trial-and-error process of designing suited architectures and choosing hyperparameters. Furthermore, this approach, being more general than traditional gradient-based methods, can be applied to a wider range of problems, such as reinforcement learning tasks, offering an alternative to the traditional reinforcement learning methods, which do not scale well for problems with large state space or partial observability, or other problems where no training data is available to perform supervised learning.

However, neuroevolution comes with a major limitation which its computational cost, making it not suitable for the evolution of networks with more than tens of thousands of parameters, while gradient-based methods have successfully been applied to networks with billions of parameters. It is still particularly interesting for problems where smaller networks can be used or where other methods are not applicable.

From the perspective of the EA, neural networks serve as the phenotype, one of the main challenges is to define an encoding strategy into genotypes, which allows for the conduction of genetic operations. For example, it is not trivial to define a crossover operation for neural networks when topologies are also being evolved. For this reason, neuroevolution has traditionally been used to evolve the weights of a fixed topology network. Genetic encoding strategies for neural networks can be divided into two main categories:

- **Direct encoding:** Genomes contain the information for all nodes and connections in the network, and are directly translated into a network. This is the strategy which is employed by most neuroevolution algorithms.
- **Indirect encoding:** Genomes contain a set of rules or instructions to generate the network, which is then constructed from these rules.

Neuroevolution algorithms evolving both the weights and the topology of the network are referred to as **TWEANNs** (Topology and Weight Evolving Artificial Neural Networks). Lastly, neuroevolution can also be used in combination with traditional training methods to optimize the hyperparameters of a network, such as the learning rate or the activation function, or the topology of the network, while the weights are learned using gradient-based methods.

Chapter 3

Literature Review

This chapter covers the first task of this thesis project, which is the review of the literature on neuroevolution algorithms and benchmarks. The first goal of this review is to identify the state of the art algorithms and benchmarks which were proposed in the neuroevolution literature, with a particular focus on the recent theory literature from 2023 and 2024. Following this, a selection of algorithms and benchmarks from the ones which were identified is made. This selection consists in the algorithms and benchmarks which are presented in detail in this section and which are implemented in the framework.

3.1 Methodology

Given the large amount of literature on this well-established field, the review was conducted in a systematic manner, following the guidelines of Wohlin 2014; Petersen, Vakkalanka, and Kuzniarz 2015. Indeed, using a rigid methodology is important in order to ensure that the review is impartial and precise.

3.1.1 Research Questions

The research goals are summarized in the following research questions:

- **RQ1:** What are the state of the art neuroevolution algorithms?
- **RQ2:** What are the different kinds of neuroevolution algorithms?
- **RQ3:** Which benchmarks are used to evaluate neuroevolution algorithms?
- **RQ4:** What are the key characteristics of these benchmarks?

RQ1 and **RQ3** are concerned with the identification of the state of the art algorithms and benchmarks, while **RQ2** and **RQ4** are concerned with the selection process.

3.1.2 Search Strategy

The search for papers was performed using Google Scholar and the DTU Findit database, which should provide an accurate representation fo the research that has been conducted on the topic. The keywords used for designing the search queries are:

"Neuroevolution", "Neural Networks", "Evolution Algorithm", "Evaluation"

3.1.3 Study Selection

Following the queries on the databases, the results are then filtered based on the title, abstract and full-text reading (in this order). An iteration of forward and backward snowballing were then conducted to include other studies which were missed in the initial search.

The following inclusion criteria were applied to the abstracts:

- **IC1:** The paper is published after 2002.
- **IC2:** It is clear that the work is proposing a new noeuroevolution algorithm or performing an evaluation of existing algorithms.
- **IC3:** The considered algorithm(s) rely solely on evolutionary algorithms

And the following exclusion criteria were used:

- **EC1:** The paper is not available in English.
- **EC2:** The full=text of the paper is not accessible
- **EC3:** The study is a duplicate of a previously included study.

The cut-off date of 2002 is motivated by the release year of the Stanley and Miikkulainen 2002 paper, proposing the NEAT algorithm, which is the most well-known neuroevolution algorithm is still the subject of many studies today.

3.2 Neuroevolution algorithms

A variety of neuroevolution algorithms have been proposed in the literature. These algorithms can be classified into different categories based on their main characteristics. The following three main distinctions have been identified during the review:

- **Conventional neuroevolution algorithms vs. TWEANNs** Conventional neuroevolution algorithms are those which only evolve the connection weights, considering a fixed topology, while TWEANNs (Topology and Weight Evolving Artificial Neural Networks) are those which also evolve the topology of the neural network.
- **The category of the evolutionary algorithm** such as genetic algorithms, evolutionary strategies, genetic programming, etc.
- **The encoding strategy** This refers to the way the neural network is encoded as a genotype, which is then evolved by the algorithm. The most common encoding strategies are direct encoding, indirect encoding and generative encoding.

3.2.1 Algorithms selection

The following algorithms were selected for the implementation in the framework:

- The (1 + 1) NA algorithm and its variants
- The Bias-Invariant (1+1) NA (BNA) algorithm
- The CMA-ES Evolutionary Strategy
- The NEAT algorithm

Given the duration of the project, the choice was made to limit the number of algorithms to four, in order to allow for a thorough implementation and evaluation of each of them, thus various criteria were used for the selection of the algorithm, to allow for a good coverage of the different categories of algorithms and interesting comparisons between them.

Therfore, the (1 + 1) NA and bias-invariant (1+1) NA algorithms were selected as the two theory paper proposals this thesis is particularly interested in. The CMA-ES algorithm, which is the representative of the evolutionary strategies category was selected because of its popularity in the modern neuroevolution literature and applications, and because of its status as a state-of-the-art algorithm for continuous optimization problems. Finally, the

NEAT algorithm, the representative of the TWEANN category, was selected because of its status as the most well-known neuroevolution algorithm, making it a subject of most comparison studies in the literature.

3.2.2 (1 + 1) NA

The (1 + 1) NA algorithm and its variants were introduced in Fischer, Larsen, and Witt 2023. In this work, the authors consider a simple neuroevolution setting where these algorithms are used to optimize the weights and activation function of a simple artificial neural network.

The artificial neural network topology

Artificial neurons with D inputs and a binary threshold activation function are considered. These neurons have D parameters, the input weights $w_1, \ldots, w_D$ and the threshold t. Let $x = (x_1, \ldots, x_D) \in \mathbb{R}^D$ be the input of the neuron. The neuron outputs 1 if $\sum_{i=1}^{D} w_i x_i \geq t$ and 0 otherwise. This can be interpreted geometrically as the neuron outputting 1 if the input vector x is above or on the hyperplane with normal vector $w = (w_1, \ldots, w_D)$ and bias t. j Furthermore, an alternative representation of the decision hyperplane can be used by considering spherical coordinates. The normal vector to the decision hyperplane is described by by $D - 1$ angles and the bias, where the bias corresponds to the distance from the origin measured in the opposite direction of that of the normal vector. As a matter of fact, for $D = 2$, the normal vector can be represented by its cartesian coordinates (x_1, x_2) or by its polar coordinates (r, θ), where r is the distance from the origin and θ is the angle with the x_1 axis. Similarly, for $D = 3$, the normal vector can be represented by its cartesian coordinates (x_1, x_2, x_3) or by its spherical coordinates (r, θ, ϕ), where r is the distance from the origin, θ is the angle with the x_1 axis and ϕ is the angle with the x_3 axis. It is easy to convert between these two representations. In addition, the spherical representation uses one less parameter than the cartesian representation, and hence, allows for the reduction of the number of inputs to the neurons to $D - 1$.

The ANNs which are considered in the study contain two layers, a hidden layer with $N > 1$ neurons and an output layer with a single neuron. Each of the hidden neurons are connected to the D inputs and output a binary value. The output neuron is connected to the N hidden neurons and computes the Boolean OR function of their outputs. This architecture is motivated by the problems which are considered in the study, described in **??**. Geometrically, these ANNs output the union of a number of N-dimensional hyperplanes.

The (1 + 1) NA algorithm

Let's consider an ANN with N neurons in the hidden layer and D inputs, with parameters $(\phi_{1,1}, \ldots, \phi_{1,D-1}, b_1, \ldots, \phi_{N,1}, \ldots, \phi_{N,D-1}, b_N$. In the paper Fischer, Larsen, and Witt 2023, the search space $[0, \ldots, n]^{ND}$ is considered, where r is the resolution of the continuous $[0, 1]$ domain. This discretiisation allows for the values $\{0, 1/r, 2/r, \ldots, 1\}$. Setting the parameters of ANNs is tipically a continous optimization problem, but rigurous runtime analysis is much less developed for continous optimization than for discrete optimization, which motivates this choice. Let $f : \{0, \ldots, r\}^{ND} \to [0, 1]$ be the fitness function which measures the performance of the ANN and is to be maximized.

The (1 + 1) NA algorithm is given in Algorithm 2. It maintains a single individual and mutates all angles and biases independently, based on a global search operator using the harmonic distribution Harm(r) on $\{1, \ldots, r\}$: For $l \sim \text{Harm}(r)$,

$$Prob(l = i) = \frac{1}{H_r} \text{ for } i = 1, \ldots, r, \text{ where } H_r = \sum_{i=1}^{r} \frac{1}{i}.$$

Algorithm 2 (1 + 1) NA

$t \leftarrow 0$
Select x_0 uniformly at random from $\{0, \ldots, r\}^{DN}$.
while termination criterion not met **do**
  Let $y = (\varphi_{1,1}, \ldots, \varphi_{1,D-1}, b_1, \ldots, \varphi_{N,1}, \ldots, \varphi_{N,N-1}, b_N) \leftarrow x_t$;
  for all $i \in \{1, \ldots, N\}$ **do**
    Mutate φ_i and b_i with probability $\frac{1}{DN}$, independently of each other and other indices;
    Mutation chooses $\sigma \in \{-1, 1\}$ uniformly at random and $l\text{Harm}(r)$ and adds σl to the selected component, the result is then taken modulo r for angles and modulo $r + 1$ for biases.
    for $i \in \{1, \ldots, N\}$ **do**
      Set bias $2b_i/r - 1$ for neuron i.
      for $j \in \{1, \ldots, D\}$ **do**
        Set the j-th polar angle to $2\pi\varphi_{i,j}/r$ for neuron i.
      end for
    end for
    Evaluate $f(y)$
    if $f(y) \geq f(x_t)$ **then**
      $x_{t+1} \leftarrow y$
    else
      $x_{t+1} \leftarrow x_t$
    end if
  end for
  $t \leftarrow t + 1$
end while

The evaluation
The ANNs which are evolved using the (1 + 1) NA algorithm output the union of N-dimensional hyperplanes. To check whether or not the input is above the hyperplane described by one of the neurons, the normal vector $\overrightarrow{n}$ to the hyperplane is computed using the angles and a norm of 1. For an input vector $\overrightarrow{x}$, the dot product $(\overrightarrow{x} - |b|\overrightarrow{n}) \cdot \overrightarrow{n}$ is computed, where b is the bias of the neuron. If $b \geq 0$, the output of the neuron is 1 if the dot product is positive and 0 otherwise. If $b < 0$, the output of the neuron is 1 if the dot product is negative and 0 otherwise.

3.2.3 Bias-Invariant (1+1) NA (BNA)
In **bna**, the authors extend upon the analysis in Fischer, Larsen, and Witt 2023 by considering more realistic ANN settings, presenting the Bias-Invariant (1+1) NA (BNA) algorithm. The considered ANNs uses Rectified-Linear-Unit (ReLU) activation functions, commonly used in real-world ANNs. This allows for the construction of bended hyperplanes, resulting in solutions to the problems described in **??** which are invariant to the bias.

The artificial neural network topology
The considered ANNs contain three layers, in which each of the neurons uses a ReLU activation function i.e they output $\max(0, \sum_{i=1}^{k} w_i x_i)$ for k inputs from the previous layer. The weights between the first and second layer and between the second and third layer are fixed. The topology for $D = 2$ in shown in **??**. The use of ReLU activation functions results in piecewise linear output. Hence, as described in **??** for the case $D = 2$, these networks compute a V-shaped area of positive classification. Such topologies are considered as a single neuron, refered to as a **V-neuron**, and which can be part of a standard ANN topology.

Therefore, these V-neurons can be described by $D + 1$ parameters:

- The bias b
- The $D - 1$ angles $\varphi_1, \ldots, \varphi_{D-1}$.
- The bend angle θ.

The area of positive classification is a (multi-dimensional) cone, all points positively classified correspond to points forming an angle smaller than the bend angle θ with the normal vector to the hyperplane given by the bias b and the $D - 1$ angles $\varphi_1, \ldots, \varphi_{D-1}$.

The Bias-Invariant (1+1) NA algorithm
The BNA algorithm is given in Algorithm 3. It is mostly the same as the (1+1) NA algorithm, with the difference that the bend angles are also mutated.

The evaluation
V-neurons output whether or not the input is in the cone described by the neuron. The vector $\overrightarrow{\varphi}$ is computed using the angles $\varphi_1, \ldots, \varphi_{D-1}$ and a norm of 1. Given an input vector $\overrightarrow{x}$, the dot product $(\overrightarrow{x} - |b|\overrightarrow{\varphi}) \cdot \overrightarrow{\varphi}$ is computed. The angle α between the input vector and the normal vector can then be computed as $\alpha = \cos^{-1}(\frac{(\overrightarrow{x} - |b|\overrightarrow{\varphi}) \cdot \overrightarrow{\varphi})}{\|\overrightarrow{x} - |b|\overrightarrow{\varphi}\| \|\overrightarrow{\varphi}\|})$. If $b \geq 0$, the output of the neuron is 1 if $\alpha \leq \theta$ and 0 otherwise. If $b < 0$, the output of the neuron is 1 if $\pi - \alpha \leq \theta$ and 0 otherwise.

3.2.4 The CMA-ES Evolutionary Strategy
CMA-ES, short for *Covariance Matrix Adaptation Evolution Strategy*, is a kind of evolution strategy (ES). An ES is an optimization technique based on evolution, and belonging to the class of evolutionary algorithms (EA). This kind of blax-box optimization algorithms aim at optimizing a function $f : \mathbb{R}^n \to \mathbb{R}$, for which the analytic form is not known, but for which evaluations of the function are possible. As it ts the case for CMA-ES,

Algorithm 3 Bias-Invariant (1 + 1) NA (BNA)

$t \leftarrow 0$
Select x_0 uniformly at random from $\{0, \ldots, r\}^{DN}$.
while termination criterion not met **do**
  Let $y = (\theta_1, \varphi_{1,1}, \ldots, \varphi_{1,D-1}, b_1, \ldots, \theta_N, \varphi_{N,1}, \ldots, \varphi_{N,N-1}, b_N) \leftarrow x_t$;
  for all $i \in \{1, \ldots, N\}$ **do**
    Mutate φ_i and b_i with probability $\frac{1}{(D+1)N}$, independently of each other and other indices;
    Mutation chooses $\sigma \in \{-1, 1\}$ uniformly at random and $l\mathrm{Harm}(r)$ and adds σl to the selected component, the result is then taken modulo r for angles and modulo $r + 1$ for biases.
    for $i \in \{1, \ldots, N\}$ **do**
      Set bias $2b_i/r - 1$ for neuron i.
      Set bend angle $\pi\theta_i/r$ for neuron i.
      for $j \in \{1, \ldots, D\}$ **do**
        Set the j-th polar angle to $2\pi\varphi_{i,j}/r$ for neuron i.
      end for
    end for
    Evaluate $f(y)$
    if $f(y) \geq f(x_t)$ **then**
      $x_{t+1} \leftarrow y$
    else
      $x_{t+1} \leftarrow x_t$
    end if
  end for
  $t \leftarrow t + 1$
end while

These algorithms are typically stochastic and are used for the optimization of non-linear or non-convex continuous optimization problems.

ES algorithms maintain a population of candidate solutions. These candidate solutions are sampled from a multivariate normal distribution. Parameters of the distribution are updated at each generation based on the performance of the candidate solutions. As a matter of fact, a simple greedy ES algorithm could consist in updating the mean of the distribution, and using a fixed standard deviation. The mean is updated to the best solution after the evaluation of the fitness of each of the candidate solutions. The next generation is then sampled around this mean. However, this kind of simple greedy algorithms is particularly prone to getting stuck at local optima because of the lack of exploration.

In order to allow for more exploration, rather than exclusively relying on the single best solution, genetic algorithm maintain a proportion of the best solutions from the current generation, and generate the next one trough recombinations and mutations. However, this approach is also prone to getting stuck at local optima, as in practice, candidates solutions end up converging to a local optimum.

CMA-ES addresses these issues and allows for the adaption of the search space when needed, reducing it when the confidence in current solutions is high, for fine-tuning, or increasing it when the confidence is low, in order to allow for more exploration. This is done by adapting the covariance matrix of the multivariate normal distribution, which stores pairwise dependencies between the parameters for the sample distribution. This makes CMA-ES a powerful and widely used optimization algorithm. The main drawback of this algorithm is its computational cost, induced by the use of the covariance matrix, which makes it less suitable for high-dimensional problems.

...

CMA-ES can be used for the evolution of fixed-topology neural networks, by considering the weights of the connections as the parameters of the optimization problem, and converting between a vector representation of the network, for the optimization input, and the standard graph representation, for the evaluation.

3.2.5 Neuroevolution of Augmenting Topologies (NEAT)

The NEAT algorithm was introduced in Stanley and Miikkulainen 2002. It is a TWANN (Topology and Weight Evolving Artificial Neural Network) algorithm, which evolves, simultaneously, both the topology and weights of neural networks. The main idea behind this algorithm is to start from a minimal topology, incrementally adding new neurons and connections to, the networks, which allows for the evolution of complex neural networks while keeping the computational cost low and justifying each new addition to the network topology. The following sections describe the main components of the algorithm.

Genetic Encoding

NEAT uses a direct encoding of the neural networks. The goal of the encoding strategy is to allow crossover among different network topologies. Each genome contains two sets of genes, which specify nodes and connections in the network:

- **Node genes** Each node gene contains an identifier and layer (input, hidden, output or bias, which is an input that is always set to 1.0.
- **Connection genes** Each connection gene specifies an input node identifier, an output node identifier, a weight, whether the connection is enabled or disabled, and an innovation number.

Where node identifiers are shared between the individuals in the population, the enabled flag specifies whether or not the connection is expressed in the phenotype (i.e the network) and the innovation number is used to track the historical origin of the gene.

Mutations
NEAT uses two types of mutations: weight mutations and structural mutations. Weight mutations are used to perturb the weights of the connections in the network,while structural mutations are used to modify the topology of the network. There are two types of structural mutations:

- **Add connection** This mutation adds a new connection between two unconnected nodes in the network. The connection is assigned a random weight.
- **Add node** This mutation adds a new node in the network, splitting an existing connection into two. The old connection is disabled and two new connections are added to the new node. The connection leading into the new node is assigned a weight of 1, while the connection leading out from the new node is assigned the weight of the old connection. This allows for the minimization of the initial effort of the mutation, as the activation of the output layer node remains the same and the weights of the new connections can be optimized in future generations.

Because of these two types of mutations, inserting new nodes and connection genes, genomes can only grow larger over time, resulting in the evolution of increasingly complex networks.

Crossover
One of the challenges of evolving neural networks is the crossover operator, because of the different topologies of the networks. NEAT addresses this issue by making use of the innovation numbers in connection genes, which allows for the tracking of the historical origin of each gene. This allows for the matching of genes between individuals. Genes are assigned increasing innovation numbers as they appear in the population and innovation numbers are inherited. Hence, the matching of genes is done by comparing the innovation numbers. Thus, the crossover operator consists in inheriting matching genes from one parent at random, and adding the remaining disjoint and excess genes from the fittest parent. This strategy is particularly cost-effective as it requires no topological analysis of the networks.

Speciation
An issue with the current strategy is that the population is unable to protect topological innovation, because of smaller topologies optimizing faster and the addition of new connections usually resulting in an initial drop in fitness. To address this issue, NEAT uses speciation, which groups individuals into species based on their genetic similarity. This strategy allows for the protection of topological innovation by having individuals compete within their specie, rather than the entire population. In addition, as it is the case with the crossover operator, historical matching allows for an efficient solution.

The similarity between two individuals is computed as a weighted sum of the number of excess E genes, the number of disjoint D genes and the average weight difference $\bar{W}$ of matching genes:

$$\delta = c_1 \frac{E}{N} + c_2 \frac{D}{N} + c_3 \bar{W}$$

Where c_1, c_2 and c_3 are coefficients which control the importance of each term, and N is the number of genes in the larger genome.

At each generation, individuals are sequentially assigned to a specie based on the similarity with the representative of the specie, which is a randomly selected individual from the previous generation which was part of the specie, and a similarity threshold. If no specie is found, a new specie is created.

Each species is given a number of offspring in proportion to the sum of the adjusted fitness of its members. Where the adjusted fitness f'_i of an individual i is given by $f'_i = f_i/n_i$, where f_i is the fitness of the individual and n_i is the number of individuals in the specie. This is done to prevent large species from dominating the population. The offspring are generated using the crossover and mutation operators, on members of the species, after the selection of the fittest individuals. At each generation, the population is replaced by the offspring.

3.3 Neuroevolution benchmarks

3.3.1 Unit hypersphere sphere classification problems

These problems, which can be thought of as a kind of ONEMAX for the (1 + 1) NA algorithm, were introduced in Fischer, Larsen, and Witt 2023. These problems consist in the binary classification of points in the D-dimensional unit hypersphere.

Half The HALF problem consists of all points with non-negative x_D coordinate on the unit hypersphere:

$$\text{HALF} = \{x \in \mathbb{R}^D, \|x\|_2 = 1 \text{ and } \varphi_{D-1} \in [0, \pi]\}.$$

Quarter The QUARTER problem consists of all points with non-negative x_{D-1} and x_D coordinate on the unit hypersphere:

$$\text{QUARTER} = \{x \in \mathbb{R}^D, \|x\|_2 = 1 \text{ and } \varphi_{D-1} \in [0, \pi/2]\}.$$

TwoQuarters The TWOQUARTERS problem consists of all points with either both negative or non-negative x_{D-1} and x_D coordinate on the unit hypersphere:

$$\text{TWOQUARTERS} = \{x \in \mathbb{R}^D, \|x\|_2 = 1 \text{ and } \varphi_{D-1} \in [0, \pi/2] \cup [\pi, 3\pi/2]\}.$$

LocalOpt The LOCALOPT problem consists of all points with polar angle φ_{D-1} between 0 and 60, 120 and 180, 240 and 300 degrees:

$$\text{LOCALOPT} = \{x \in \mathbb{R}^D, \|x\|_2 = 1 \text{ and } \varphi_{D-1} \in [0, \pi/3] \cup [2\pi/3, \pi] \cup [4\pi/3, 5\pi/3]\}.$$

3.3.2 XOR

This classic benchmark problem is a binary classification problem, which consists in the classification of the four points $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$, according to the XOR function. The points $(0, 0)$ and $(1, 1)$ are of class 0, while the points $(0, 1)$ and $(1, 0)$ are of class 1. The popularity of this simple problem comes from its non-linear nature, which makes it impossible to solve with a single-layer perceptro.

3.3.3 Dataset Classification Problems

Classification using datasets is a classical use-case for neural networks. It consists in training a network on labeled data and using it to predict the label of unseen data. Hence, the hypersphere classification problems, presented in Section 3.3.1, from the theory studies, differ from this kind of task, by having algorithms trained and tested on the same data.

Although these problems are not common use-cases for neuroevolution because of the use of labeled data, they are particularly interesting for this study, by allowing to show how neuroevolution can be applied to these common tasks, testing the algorithms on larger state spaces, and potentially observing whether common behaviors which occur when training ANNs using gradient-based methods, such as over-fitting or under-fitting, also apply to neuroevolution.

The proben1 benchmark, presented in Prechelt 1994, introduces various standard benchmark datasets, including the *cancer* dataset, which contains 699 entries, consisting in cell descriptors gathered by microscopic for tumors being benign or malignant. Each dataset entry contains 9 input features, and a binary output,

The algorithms are evolved using the first 90% of the data, and are tested on the remaining 10%.

3.3.4 Pole Balancing

The pole balancing problem, as described in Wieland 1991, is a classical benchmark in control theory, reinforcement learning and neuroevolution literature. It consists in controlling a cart with one degree of freedom, which moves along a one-dimensional track, by applying a horizontal force to it, in order to balance a pole attached to it using a hinge. Some of the reasons for the popularity of this benchmark problem are its simplicity, its relevance to real-world control problems and its unstable and non-linear dynamics.

The difficulty of the problem can be adjusted by changing the number of poles. Indeed, if the poles have different lengths, they will react differently to the forces applied to the cart. As the single pole variant has become too easy for current techniques, we consider the widely used case of double pole balancing, where two poles are attached to the cart.

The state of the system is described by the cart position x, the cart velocity $\dot{x}$, the pole angles θ_1 and θ_2 and the angular velocities $\dot{\theta}_1$ and $\dot{\theta}_2$. This task is Markovian, as the state contains all the information needed to determine the future evolution of the system. A more challenging variant of the problem consists in removing the velocity informations from the state, which requires the use of recurrent connections, which were not considered in this project.

The dynamics of the system are described by the following equations:

$$\ddot{x} = \frac{F - \mu_c \text{sgn}(\dot{x}) + \sum_{i=1}^{N} \tilde{F}_i}{M + \sum_{i=1}^{N} \tilde{m}_i}$$

$$\forall i \in \{1, N\}, \ddot{\theta}_i = -\frac{3}{4l_i}(\ddot{x}\cos\theta_i + g\sin\theta_i + \frac{\mu_{pi}\dot{\theta}_i}{m_i l_i})$$

$$\forall i \in \{1, N\}, \tilde{F}_i = m_i l_i \dot{\theta_i}^2 \sin\theta_i + \frac{3}{4} m_i \cos\theta_i (\frac{\mu_{pi}\dot{\theta}_i}{m_i l_i} + g\sin\theta_i)$$

$$\tilde{m_i} =_i (1 - \frac{3}{4}\cos^2\theta_i).$$

Where:

- F is the magnitude of the force applied to the cart
- μ_c is the cart friction coefficient
- M is the cart mass
- N is the number of poles
- l_i is the length of the i-th pole
- μ_{pi} is the pole friction coefficient
- m_i is the mass of the i-th pole
- g is the gravity constant

The equations for motion are integrated using the Euler method. For a time step Δt:

$$\begin{aligned} x_{t+1} &= x_t + \dot{x}_t \Delta t \\ \dot{x}_{t+1} &= \dot{x}_t + \ddot{x}_t \Delta t \\ \theta_{i,t+1} &= \theta_{i,t} + \dot{\theta}_{i,t} \Delta t \\ \dot{\theta}_{i,t+1} &= \dot{\theta}_{i,t} + \ddot{\theta}_{i,t} \Delta t. \end{aligned}$$

The fitness function is defined as the sum of the time steps during which the poles are balanced and the cart is within the allowed bounds, over 1000 time steps. A pole is considered balanced if its angle is within 30 degrees of the vertical position. The evaluated algorithms outputs are mapped to the magnitude of the force $F \in [-10, 10]$N to be applied to the car at each time step.

Chapter 4

The framework

This chapter presents the design and implementation of the framewor which was developed as part of this thesis. It list the requirements of the framework, goes through its development lifecycle and presents the architecture of the framework.

This framework, which is the main contribution of this thesis, allows for a fair comparison of the considered algorithms by having them implemented in a common environment and language, in order to avoid performacne differences induced by the programming language, and running them on the exact same benchmark problem implementations and settings.

The code for the framework is available at `https://github.com/MSc-Thesis-Samy/code` and includes a README file with instructions on how to use it.

4.1 Requirements

4.1.1 Goals and functional requirements

The overreaching goal of the framework is to provide a tool for the evaluation of neuroevolution algorithms on benchmark problems, based on the selection presented in Chapter 3. Tests are specified through a command line interface, they consist in an algorithm and problem pair, along with a set of additional parameters. The framework collects the results of the tests as well as the list of passed-in parameters, algorithm and problem. These tests can either be run individually or in batch mode, where the framework runs a set of tests in parallel, and collects statistics on these runs.

Furthermore, the framework also allows for the visualization of the problem, solution process and network structure through a graphical user interface. In addition, it generates graphs for visualizing the test results, plotting the performance of the algorithm on the benchmark problem over the generation count.

4.1.2 Non-functional requirements

Non-functional requirements are the requirements that specify the quality of the system, rather than the features it should have. Apart from the functional requirements that specify the features expected of the framework, a number of non-functional requirements have also been identified.

- Usability and user experience: the framework should be easy to use and provide a good user experience.
- Documentation: The framework should be well documented, providing a clear and concise guide on how to use it.

- Error handling: All errors should be handled gracefully as to not result in runtime errors.
- Performance: The framework should allow for the execution of tests in parallel, making use of multiple CPU cores.
- :Extensibility: The framework should be easily extensible, allowing for the addition of algorithms and benchmarks without any major changes to the existing codebase.
- Support: The framework should be able to run on the three major operating systems: Windows, Linux and MacOS.

4.2 Architecture

The framework is implemented in Rust. This general-purpose programming language, originally intended to serve as an alternative for system languages such as C and C++, offers a good balance between performance provided by such low-level languages and the safety and ease of use of higher-level languages such as Python. Furthermore, various libraries (referred to as Rust crates) which could be used when implementing aspects of the framework, such as designing the command-line interfaces and graphical-user interfaces, or running CMA_ES are available in Rust. Lastly, the language can target a range of platforms, including Windows, Linux and MacOS.

The framework is divided into three main components:

- The core: This component is responsible for the execution of the tests and the collection of results. It contains the algorithms and benchmark implementations.
- The command-line interface: This component allows the user to specify the tests to be run, as well as the parameters for these tests.
- The graphical user interface: This component allows the user to visualize the problem, solution process and network structure, as well as the test results.

The core is used by both the command line interface and the graphical user interface. And the graphical user interface is used by the command line interface. Indeed, the UI allows for the visualization of the solution process, but interaction with the framework is done when invoking it through the command line interface.

The dependency graph of the framework is shown in **??**.

4.2.1 Background on Rust features

This section provides a brief overview of the features of the Rust programming language before going into the details of the framework's implementation.

Object-oriented capabilities

Nowadays, object-oriented languages are considered the norm for the development of large-scale software systems in the industry. Rust is inspired by various programming paradigms, such as functional programming and object-oriented programming. Although there is no consensus on the list of features which define an object-oriented programming language, Rust can be considered object-oriented. Indeed, it allows for the definition of structs and enums which can store data and methods using implementation blocks.

It also allows for encapsulation through the use of the `pub` keyword, which specifies the visibility of objects, thus defining the public API for interacting with them. When not using the `pub` keyword, the object is private and can only be accessed by the module it is defined in. Modules are used to organize code and define the visibility of objects, and

can be nested to form a hierarchy. However, a major difference with other object-oriented languages such as Java or C# is that Rust does not have a class-based inheritance system. Instead, it takes a different approach which consists in polymorphism, which is a more general system referring to code that can work with multiple types. In practice, this is achieved through the use of generics in method and object definitions, and the use of traits, which are similar to interfaces in other languages. In fact, traits allow for the definition of a default implementation for a set of methods, which can be overridden by the implementing type.

Enums
...

Library management
Cargo is Rust's build system and package manager. Most projects are managed using this tool which handles the download and building of the dependencies of projects. In Rust, packages are referred to as crates.

Attributes
Attributes are metadata applied to some module, crate or object. They are, for example, used to enable compiler features or mark functions as unit tests. or to define the behavior of the code. They are defined using the `#` symbol and are placed before the object they are associated with.

Testing
In Rust, unit-testing is usually done by defining a test module (with a test attribute) at the end of the file containing the functionalities to be tested. Test functions correspond to functions defined in such modules, which are marked with a specific test attribute. Tests can be run with `cargo test`. They fail when a *panics* occur, and utility macros such as `asserteq` or `assert` can be used to panic when conditions are not met.

Concurrency
Concurent programming refers to different parts of a program executing independently of each other, while parallel programming refers to different parts of a program executing in parallel, at the same time. For simplicity, in this section, "concurrency" should be understood as "concurrency and/or parallelism". These concepts have become particularly important in the context of modern computing, where multi-core processors are the norm. However, writing concurrent programs can be difficult, as it can be error-prone, hard to debug and reason about. Rust aims at addressing these issues by making it use of its main feature: the borrowing and ownership system, which allows many concurrency issues to be caught at compile time, rather than at runtime, as it is the case in other languages. In Rust, concurrency is achieved through the use of threads, which are lightweight processes that can run concurrently. The standard library provides various methods for creating and managing threads. However, in this project, the `rayon` crate is used instead. It is a data parallelism library which allows for a particularly easy way to parallelize code, by providing parallel iterators and parallel maps, and abstracting away the details of thread creation and management, guaranteeing data-race free execution and benefiting from parallelism when possible.

For example, Listing 4.1 shows how a function `sum_of_squares`, which computes the sum of the squares of the elements of an array. By simply using the `par_iter` method from the `rayon` crate, instead of the `iter` method, the function can be parallelized, as demonstrated by the `sum_of_squares_parallel` function.

4.2.2 Core
This component contains the core functionality of the framework Various constants and utility functions, used across the project, are defined in the `utils.rs` and `constants.rs`

```rust
use rayon::prelude::*;

fn sum_of_squares(input: &[i32]) -> i32 {
    input.iter().map(|x| x * x).sum()
}

fn sum_of_squares_parallel(input: &[i32]) -> i32 {
    input.par_iter().map(|x| x * x).sum()
}
```

Listing 4.1: Sum of squares

files.

Algorithms
The algorithms are implemented as structs and are accessed as variants of an `Algorithm` enum, which lists all the implemented algorithms. The algorithm structs and the `Algorithm` enum implement the `NeuroevolutionAlgorithm` trait, which defines the methods that an algorithm must implement. These methods are `optimization_step`, `optimize_cmaes`, `evaluate` and `optimize`, which is implemented as a default method in the trait. The `Algorithm` enum and the `NeuroevolutionAlgorithm` trait are defined in the `neuroevolution_algorithm.rs` file. Each of the different algorithm structs are defined in their own file, such as `vneuron.rs` or `neat.rs`.

Benchmarks
The problems are implemented as variants of a `Benchmark` enum, and are defined in the `becnhmarks.rs` file. This enum contains three variant, for each of the implemented benchmark types: `PoleBalancing`, `Classification` and `SphereClassification`. The two classification enums hold an instance of the `LabeledPoints` type, storing the labeled data. The file contains functions to generate the data for each of the classification problems, i.e, parsing it from a text file for the *cancer* problem and generating and iterating other angle values for the sphere classification problems. In particular, the `Benchmark` enum defines a `evaluate` method, taking as input an algorithm (an instance of the `Algorithm` enum) and which returns the fitness of the algorithm on the task.

Classification problems For the sphere classification and dataset classification tasks, the fitness is computed using the `classification` function from the `benchmarks.rs` file, which computes the MAE (Mean Absolute Error). For the bna and $(1 + 1)$ NA algorithm, which output boolean values, this is equivalent to computing the accuracy (i.e, the number of correct predictions divided by the number of total predictions), while also allowing for the evaluation of the CMA-ES and NEAT method, outputting probables when using the sigmoid activation on the output neuron.

Pole Balancing The logic for the pole balancing simulation is implemented in the `pole_balancing.rs` file, where the state is defined, along with methods responsible for updating it based on the equations and the Euler method. The `pole_balancing` function in the `benchmarks.rs` file simply updates the state, using the algorithm output as the applied force, and checking whether or not the success conditions for the task are still met.

Testing
Unit-tests were implemented across the core component to test the behavior of the algorithms and benchmarks. These tests were implemented in parallel with the functionalities to avoid and identify potential bugs early in the development process before building up with more functionalities. In addition, they were ran at each new push to github using a

workflow responsible for compiling the project, targeting linux, and running all the tests. This is to ensure that new changes do not break any past working functionality.

In fact, the bna and $(1 + 1)$ NA problems were tested using the sphere classification problems, where optimal solutions are known. These tests consist in checking whether one of these algorithms, with parameters corresponding to an optimal solution, does indeed lead to a maximum fitness value of 1.0. These tests are defined in the `benchmarks.rs` file. For example, the `test_half_network` function checks that a decision line corresponding to the x-axis for the continuous $(1 + 1)$ NA algorithm gives a fitness of 1.0. For the BNA algorithm, where there are infinitely many solutions to the sphere classification problems, different solutions were checked.

Furthermore, two additional tests in the `benchmarks.rs` file are responsible for checking that the data is loaded properly from the *cancer* text file. Tests in the `pole_balancing.rs` file test the physics of the simulation implementation, for example verifying that a pole at the lowest position, with no external force, does not lead to any movement of the cart or the pole.

Lastly, regarding the CMA-ES and NEAT algorithms, the behavior of their core functions was tested. Test functions in the `neat.rs` test the behavior of functions such as the crossover or initialization, using examples from the original paper Stanley and Miikkulainen 2002 when available. The functions in `neural_network.rs` test the output of the feed-forward method and activation functions.

4.2.3 Command-line interface

The command lien interface is implemented using the `clap` crate, which is a widely used command line argument parser in Rust. It allows for the execution of tests, by providing the algorithm, the problem and additional parameters. These additional parameters are optional and have default values, they are used to specify parameters for the algorithms, such as the number of neurons, parameters for the optimization, such as the number of iterations, and toggling the visualization of the solution process and network structure.

Arguments can be of three different types:

- **Positional**: These are required arguments which are specified in the order they are defined in the command line interface.
- **Named**: These are optional arguments which are specified by their name and a value.
- **Flags**: These are optional arguments which are specified by their name and are either present or not.

The `Cli` struct is defined in the `cli.rs` file, it contains members for each of the command line arguments and derives the `Parser` trait. The argument name is set to the member name, a short name,help message, default value and the argument type are specifies in an attribute on the member.

The arguments are parsed in the `bin/main.rs` file, which is the entry point of the program.

In order to ensure that the passed-in arguments are valid, the rust type system is leveraged, specifying appropriate data types for each of the argument. For example, the iteration number is set to an unsigned integer `u32`, while the algoithm and benchmarks are set to two enums, `AlgorithmType` and `Problem`, with variants for each possible option.

The command line interface is shown in 4.4. Parameters for the BNA and $(1 + 1)$ NA

```
input_ids = [1, 2]
output_ids = [5]
bias_id = 3

[[neurons]]
id = 4
inputs = [1, 2, 3]
activation = "sigmoid"

[[neurons]]
id = 5
inputs = [1, 2, 3, 4]
activation = "sigmoid"
```

Listing 4.2: Example of configuration File for CMAE-ES

```
population_size = 150
n_inputs = 2
n_outputs = 1
weights_mean = 0.0
weights_stddev = 0.8
perturbation_stddev = 0.2
new_weight_probability = 0.1
enable_probability = 0.25
survival_threshold = 0.25
connection_mutation_rate = 0.3
node_mutation_rate = 0.03
weight_mutation_rate = 0.8
similarity_threshold = 15.0
excess_weight = 1.0
disjoint_weight = 1.0
matching_weight = 0.3
champion_copy_threshold = 5
stagnation_threshold = 1500
```

Listing 4.3: Example of configuration File for NEAT

algorithms are passed in as named arguments. For NEAT and CMA-ES, where more parameters can be specified, the path to `.toml` configuration files are passed-in instead. Examples of such configuration files are shown in 4.2 and 4.3. The CMA-ES configuration file is used to specify the fixed network topology and activation functions, while the NEAT configuration file is used to specify the various parameters of the algorithm.

4.2.4 Graphical user interface

The UI is implemented using the `ggez` crate, which is intended to be a simple 2D game framework. In particular, it provides a simple interface for creating windows, drawing geometrical shapes and handling user input. This crate relies on the definition of a `State` struct, which holds the parameters of the game state, and which implements the `EventHandler` trait. This trait defines two methods: `update`, which is used for updating the state, and `draw` which is used for rendering the state. These two methods are called by the game loop, which is triggered in the `bin/main.rs` file if the `gui` flag is set when invoking the program.

This game abstraction is particularly suited for the implementation of visualization in the framework. The `State` struct defined in `gui.rs` holds an `algorithm`, a `problem`, and two additioal members keeping track of the number of iterations. An instance of this struct is

```
Neuroevolution framework for testing algorithms on benchmark problems.

Usage: main [OPTIONS] <ALGORITHM> <PROBLEM>

Arguments:
  <ALGORITHM>  The algorithm to test [possible values: oneplusonena, bna,
      neat, neural-network]
  <PROBLEM>    the benchmark problem [possible values: half, quarter, two-
      quarters, square, cube, xor, pole-balancing, proben1-train, proben1-
      test]

Options:
  -r, --resolution <RESOLUTION>  Resolution, when applicable [default:
      1000]
  -i, --iterations <ITERATIONS>  Number of iterations [default: 500]
  -c, --continuous               Use the continuous version of the
      algorithm, when applicable
  -e, --es                       Optimize using cma-es
  -n, --neurons <NEURONS>        Number of neurons, when applicable [
      default: 1]
  -g, --gui                      Display visualization
  -f, --file <FILE>              Configuration file
  -o, --output <OUTPUT>          Results output file
  -t, --test-runs <TEST_RUNS>    Number of runs
  -h, --help                     Print help
  -V, --version                  Print version
```

Listing 4.4: Command line interface

created in the `bin/main.rs` file using the arguments passed in the command line interface. The `update` method updates the `algorithm` by running an optimization step, and the `draw` renders the different visualizations, based on the `problem` and `algorithm` members.

The number of iterations and the best fitness value are shown in the top left corner of the window.

Classification problems
For the sphere classification problem, the unit-sphere is shown, along with its points which are labeled as `true`, which are shown in green, and its points labeled as `false` which are shown in red. When running the $(1 + 1)$ NA algorithm, the decision line and normal vector are shown. In the case of the BNA algorithm, the normal vector and decision cones are shown. For the CMA-ES and NEAT algorithms, the output of the network is shown.

Pole balancing
The visualization of the pole balancing problem consists in drawing the cart and pole, and updating their position based on the state of the simulation. It is implemented in the `pole_balancing_gui.rs` file. Compared to the classification problems, where the visualization is updated as the algorithm is evolved in the `update` method, the pole balancing visualization does not update the passed-in algorithm, but rather updates the simulation state based on the output of the algorithm.

Chapter 5

Experiments

5.1 Empirical performance testing

This section presents the methodology and results of the empirical performance testing of the selected algorithms on the selected benchmarks, using the framework which was designed and implemented for this thesis. The goal of this phase is to design and run experiments to evaluate the performance of the algorithms on the benchmarks, and based on the empirical results, which are backup-up by statistical tests, formulate hypotheses on the relative performance of the algorithms given the problem class, and suggest guidelines for the choice of algorithms, as well as for the choice of parameters for these algorithms.

In the following sections, results are presented for each algorithm on the selected benchmark problems they can be applied to, in chronological order of the experiments. This allows for a detailed analysis of the performance of each of the algorithms and experimentation with various configurations, before comparing them on the same problems, as well as motivating the different experiments which were conducted. Then, following the comparison, guidelines for the choice of algorithms and parameters are formulated..

5.1.1 Methodology

As it is the case with the literature review, it is important to define a clear and systematic methodology for the empirical performance testing phase. This will ensure the reproducibility of the experiments, and the validity of the results.

First of all, it is worth noting that the implementation of the different algorithms and benchmarks as parts of a ingle framework, is a key factor in ensuring the fairness of the comparison between the different algorithms and the validity of the results. Indeed, the different algorithms are implemented in the same language, are queried using a same API, and are run on the same hardware. However, this also means that errors or small changes in the implementation of the algorithms or benchmarks are possible, and could thus affect the results compared to the original descriptions or implementations.

As one of the goal of the experiments is to evaluate different configurations of the algorithms, the ideal experiment would consist in testing all possible parameters of the algorithms on all possible benchmarks. However, this is obviously not feasible, as there are infinitely many possible configurations of the algorithms. Therefore, the experiments will be designed to test a subset of the possible configurations, which will be chosen based on intuition and the literature review. Furthermore, since guidelines for the choice of parameters are sought, experiments will be guided towards better performance of the algorithms on the problems. This is done by iteratively identifying the parameter values which lead to better

performance, and testing more configurations around these values, in order to refine the choice of parameters.

Performance measures
The goal of the experiments is to evaluate the performance of algorithms on benchmark problems, but how is performance defined in this case? The performance of an algorithm can be measured in many way, depending on the problem and goals. For example, in the case of a sorting algorithm, performance could be measured as the number of comparisons, but in a context where memory usage is important, it is also worth considering the space complexity of the algorithm.

In the case of this thesis, the following performance measures are considered:

- **Execution time**: the time taken by the algorithm to solve the problem.
- **Quality of the solution**: the quality of the solution found by the algorithm, i.e the fitness of the best performing individual in the final population. For the implemented benchmarks, all fitness values are in the range $[0, 1]$, and the higher the fitness, the better the solution.
- **Generations**: the number of generations taken by the algorithm to solve the problem. This metrics only makes sens because of the "early stopping" criterion used in the implementation of the algorithms, which stops the algorithm when a fitness threshold is reached.

Testing workflows in practice
Given the high computational cost induced by the high number of experiments to run, the experiments were run on the high-performance computing cluster (HPC) of DTU. This will make it possible to benefit from the high number of cores available in the cluster, because of the parallel nature of the implemented testing logic. More precisely, the central DTU HPC cluster (LSF 10) was used [1]. It contains nodes with 10, 12, 16 or 24 cores. Applications are run on the cluster by mean of job scripts, with the resource manager parsing the scripts and handling the usage of the available resources. Job scripts contain speciation for the resources requirements, job constraints, a specific queue to use and commands to setup the environment and run the application. Queues are used to order jobs which are not run immediately.

In addition to running the unit-tests, by having the target set to linux, it also allows to verify that the project should be able to compile on the HPC. An example of a job script is shown in Listing 5.1, it runs the $(1 + 1)$ NA algorithm on the *Quarter* benchmark with a resolution of 50 to 1000, and 1000 runs for each resolution, and saves the results in CSV files. The configuration of the job script is done using the LSF directives, which are comments starting with `#BSUB`.

```
#!/bin/sh

#### General options
### -- specify queue --
#BSUB -q hpc
### -- set the job Name --
#BSUB -J Bench
### -- ask for number of cores (default: 1) --
#BSUB -n 10
### -- specify that the cores must be on the same host --
#BSUB -R "span[hosts=1]"
### -- specify that we need 4GB of memory per core/slot --
```

[1] `https://www.hpc.dtu.dk/?page_id=2520`

```
#BSUB -R "rusage[mem=4GB]"
### -- specify that we want the job to get killed if it exceeds 5 GB per
    core/slot --
#BSUB -M 5GB
### -- set walltime limit: hh:mm --
#BSUB -W 24:00
### -- set the email address --
# please uncomment the following line and put in your e-mail address,
# if you want to receive e-mail notifications on a non-default address
#BSUB -u s222887@dtu.dk
### -- send notification at start --
#BSUB -B
### -- send notification at completion --
#BSUB -N
### -- Specify the output and error file. %J is the job-id --
### -- -o and -e mean append, -oo and -eo mean overwrite --
#BSUB -o ~/Output_%J.out
#BSUB -e ~/Output_%J.err

n_runs=1000
problem=quarter
algorithm=oneplusonena

cd ~/code-master

for resolution in $(seq 50 50 1500)
do
    ./target/release/main $algorithm $problem -i 200 -n 1 \\
    -r $resolution -t $n_runs \\
    -o ~/output/$algorithm/$problem/$algorithm_$problem_$resolution.csv
done
```

Listing 5.1: Example of a job script

A python script was written to merge the results of the different runs in a single CPU file, extract the relevant metrics and statistics, and output data which directly be used in the pgfplots plots of this report.

5.2 Results of the experiments

This section presents the results of the different experiments which were conducted to highlight properties of the algorithms, observe the impact of different parameters on the performance, and evaluate the algorithms on the selected benchmarks.

5.2.1 Results of the $(1+1)$ NA algorithm

The $(1+1)$ NA algorithm can be applied to binary classification problems. Results are thus presented for the unit-sphere classification problems, the *XOR* problem, and the Cancer1 classification problem. The parameters which can be tuned for this algorithm are the resolution and the number of neurons.

Unit-sphere classification problems

The unit-sphere classification problems Fischer, Larsen, and Witt 2023 were introduced to evaluate the $(1+1)$ NA algorithm. For these simple problems, optimal solutions are known. In particular, these problems can be used to observe some of the properties of the algorithm.

The first experiment consisted in running the algorithm with different configurations on the simple `Half` and `Quarter` benchmarks. Results for these benchmarks are shown in Figure 5.4. The maximum number of iteration was set to 200, and the evolution stopped

when the fitness was 2% away from the maximal fitness of 1.0. Since these tasks can be solved with a single neuron, the number of neurons was set to 1 and different resolutions ranging from 2 to 1500 were tested. The results show that, for small resolutions, the algorithm is unable to solve the problems, and thus stops after 200 iterations. This is because such low resolutions do not allow to set the parameters to values which are close enough to the optimal ones. On the other hand, for resolutions allowing for a finer discretization of the space, the algorithm is always able to find the optimal solutions, and the number of iteration it takes to do so does not vary with the resolution over the tested range. Lastly, these results also show, that as expected, the CPU time is proportional to the number of iterations, as the same steps are performed by the algorithm at each generation.

Because of the lack of a stagnation criteria in the previous experiment, the algorithm would only stop when it reached a fitness close enough to the optimal on, or when the maximum number of iterations was reached. As it can be seen with the results for the lower resolutions, this is far from ideal, as the algorithm continues to run even though no progress can be made.

Figure 5.5 and Figure 5.6 show the results of the $(1 + 1)$ NA algorithm on the *Half* and *Quarter* benchmarks, with a single neuron, over resolutions ranging from 2 to 1500, but with different values of maximum stagnation iterations. The maximum number of iterations was kept at 200, and the evolution stopped when the fitness was 2% away from the maximal fitness of 1.0. Five different values of maximum stagnation iterations were tested, ranging from only 5 generations to 80 generations, which is close to the number of iterations it takes solve the problems, as it can be seen in Figure 5.4.

This time, the algorithm is not always solving the problem, even for high-enough resolutions. Higher number of stagnant iterations allow the algorithm to reach a higher fitness, but at the expanse of more iterations (and thus, more CPU time). As a matter of fact, for the *Half* problem and a resolution of 800, the algorithm reaches, on average, a fitness of 0.84 and stops after 15 iterations with 5 stagnant iterations, while it reaches a fitness of 0.98 after 69 iterations with 80 stagnant iterations. In addition, this stagnation criteria allows to stop the algorithm for the small resolutions of 2 and 5, while still letting the algorithm reach the maximal possible fitnesses in these two cases. Lastly, an interesting phenomenon can be observed on both problems, for stagnant iterations values of 60 and 80 arond resolutions of a 100, where the number of iterations is at its highest. This can also be observed in the results of the previous experiment, when ignoring the lower resolutions, and could be explained by these resolutions being hgh enough to allow for setting the parameters to set the parameters to values which are close enough to the optimal ones, but not high enough to make drawing these values as liely as for higher resolutions.

The more complex *TwoQuarters* problem requires two neurons to be solved. With a single neuron, the best fitness that can be achieved is 0.75, because at least a quarter of the unit sphere cannot be classified correctly with a single decision line. Such a solution is shown in Figure 5.1. Figure 5.7 shows the results of the $(1 + 1)$ NA algorithm on the *TwoQuarters* benchmark, with a single or two neurons, a fixed resolution of 400, and values of maximum iterations ranging from 50 to 2500. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of iterations was reached. The resolution of 400 was chosen based on the results of the previous experiments, where it could be seen that this value was in the range of resolutions where the problems could be solved and the number of iterations needed to do so was the lowest and not varying.

As expected, in the case of one neuron, the algorithm is unable to solve the problem, and

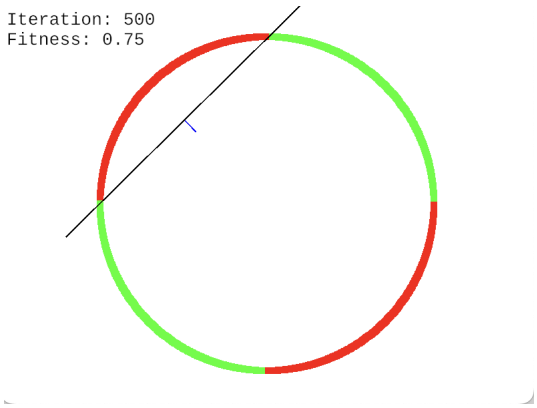
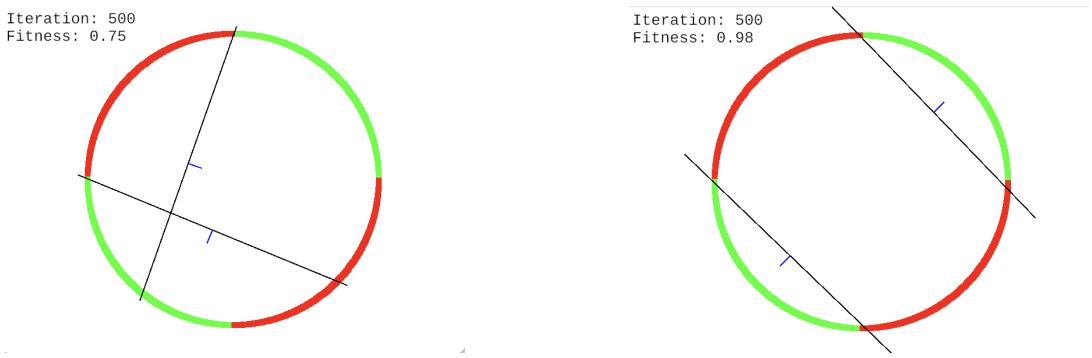


Figure 5.1: Visualization of a solution with a fitness of 0.75 after evolution of the $(1 + 1)$ NA algorithm with one neuron on the *TwoQuarters* problem. The top-left quarter of the unit sphere is misclassified.



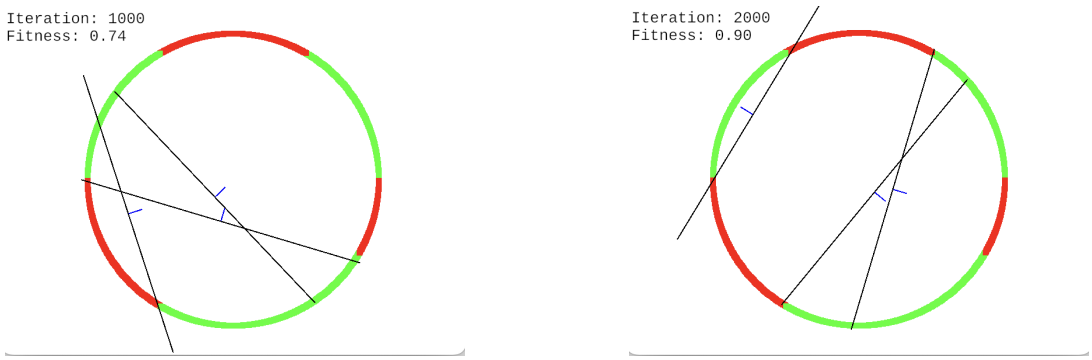
(a) Local optima

(b) Optimal solution

Figure 5.2: Visualization of two configurations of the $(1 + 1)$ NA algorithm, with two neurons, after evolution on the *TwoQuarters* problem.

the maximum fitness of 0.75 is almost reached at the lowest tested maximum number of iterations of 50. As a consequence, it can also be seen that the number of iterations taken by the algorithm is equal to the maximum number of iterations, since the fitness is never within 2% of the optimal one. In the case of two neurons, the algorithm, which can theoretically solve the problem, achieves a higher average fitness, but is unable to find the optimal solutions on all runs and gets stuck at a local optima with a fitness of 0.75. The optimal solution and an example of a local optima are shown in Figure 5.2. It can be seen that the number of iterations taken by the algorithm in the case of two neurons is lower than for one neuron. However, the CPU time is higher, because of iterations being more computationally expensive, as the algorithm has to evolve two neurons instead of one. Furthermore, the average fitness, iterations and CPU time all seem to grow logarithmically with the maximum number of iterations.

In Figure 5.8, the results of the algorithm, with two neurons, on the *TwoQuarters* problem are shown with different values of maximum stagnation iterations. The other parameters were kept the same as in the previous experiment, and the number of stagnant iterations was varied from 5 to 1000 with a step size of 20. On average, the algorithm stops before 1200 iterations, without always solving the task, even when it is allowed to stagnate for 1000 iterations. This shows how hard it is for the algorithm to leave the local optima it



(a) Local optima with fitness of 0.75

(b) Local optima with fitness of 0.9

Figure 5.3: Visualization of two configurations of the $(1 + 1)$ NA algorithm, with three neurons, after evolution on the *LocalOpt* problem.

Table 5.1: Results of the $1 + 1$ NA algorithm on the *XOR* problem. The algorithm was tested with a maximum number of stagnant iterations of 200.

Number of neurons	Average fitness	Average iterations	Average CPU time (s)
1	0.75	200	0.01
2	0.83	168	0.01

gets stuck in.

Finally, the *LocalOpt* problem requires three neurons to be solved. For this problem, there are local optima with fitnesses of 0.75 and 0.9, as shown in Figure 5.3. Figure 5.9 shows the results of the algorithm on this problem, with a fixed resolution of 400, one, two or three neurons, and maximum number of iterations ranging from 50 to 2000. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of iterations was reached. For all configurations, the algorithm was unable to solve the problem. In the case of one neuron, it reaches the maximum possible fitness of 0.75 after approximately 200 iterations. The fitness curve for two and three neurons are almost identical, and converge to a fitness of 0.85. Since the algorithm was unable to find the optimal solution, the maximum number of iterations was always reached, while the CPU time results reflect the impact of the number of neurons on the computational cost of the algorithm.

XOR problem

The *XOR* problem requires two neurons to be solved. The results of the $(1 + 1)$ NA algorithm on this problem are shown in Table 5.1. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of 200 stagnant iterations was reached, and the resolution was set to 400. For one neuron, the maximum reachable fitness is 0.75, while this value corresponds to a local optima in the case of two neurons. A solution and a local optima are shown in Figure 5.10. As it was the case with the *LocalOpt* and *TwoQuarters* benchmarks, even though two neurons are enough to solve the problem, the algorithm often gets stuck in a local optima. The average fitness is 0.79 for two neurons, and the number of iterations is 174, which shows that the algorithm was able to converge to the optimal solution in some cases, but ended up in a local optimla in most cases.

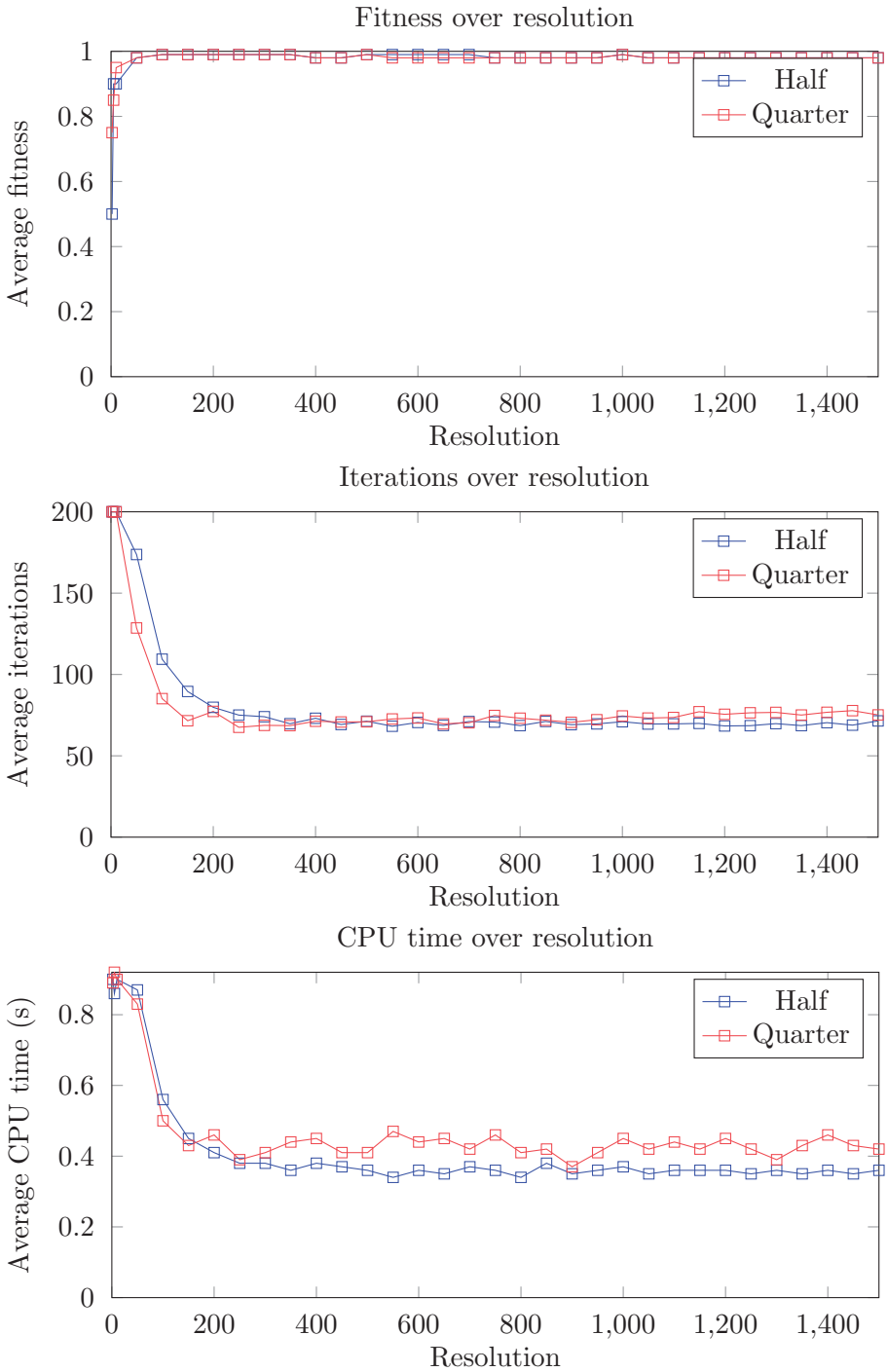


Figure 5.4: Performance metrics of the $(1 + 1)$ NA algorithm on the *Half* and Quarter benchmarks, with a single neuron, over different resolutions. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0. The maximum number of iterations was set to 200.

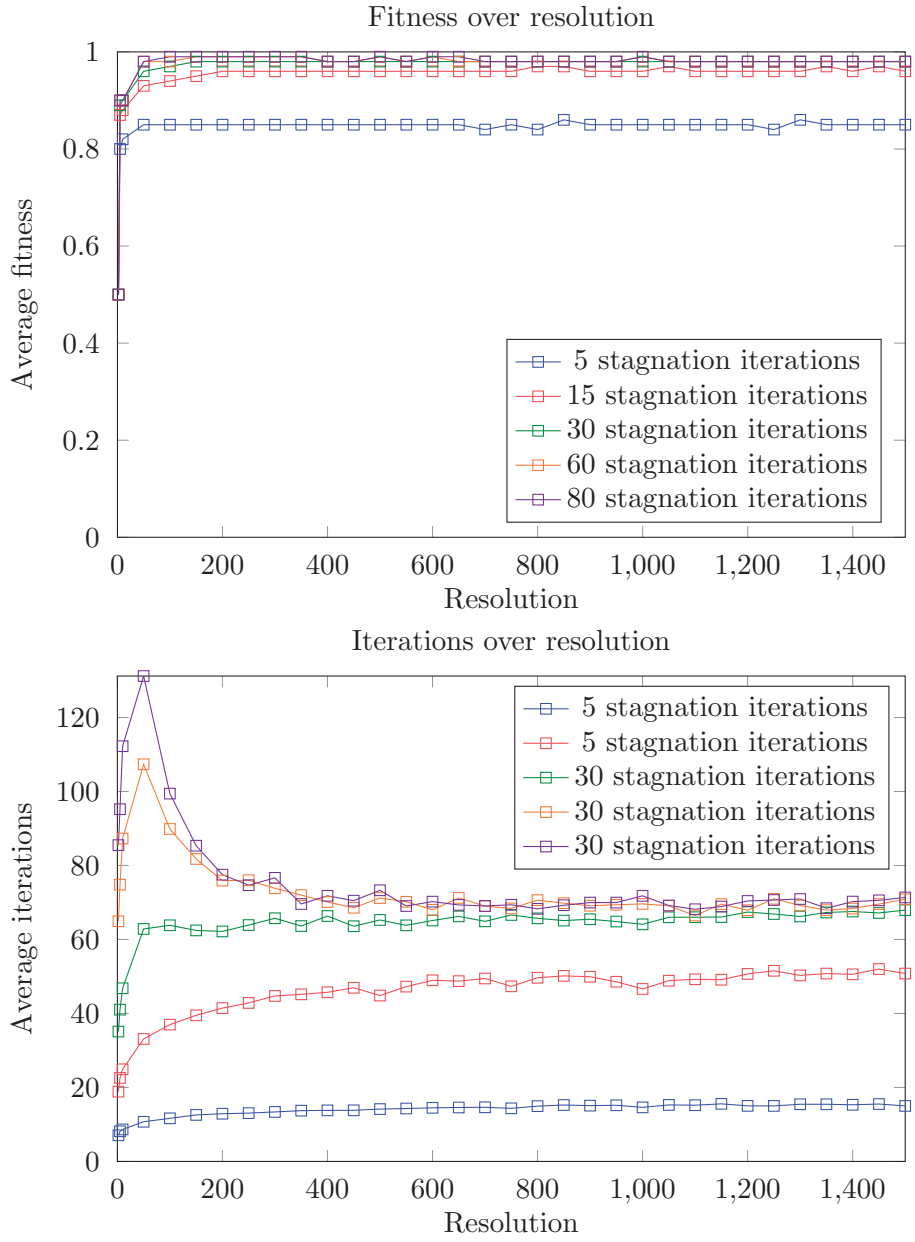


Figure 5.5: Performance metrics of the $(1+1)$ NA algorithm on the *Half* benchmark, with a single neuron, over different resolutions. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of stagnation iteration was reached.

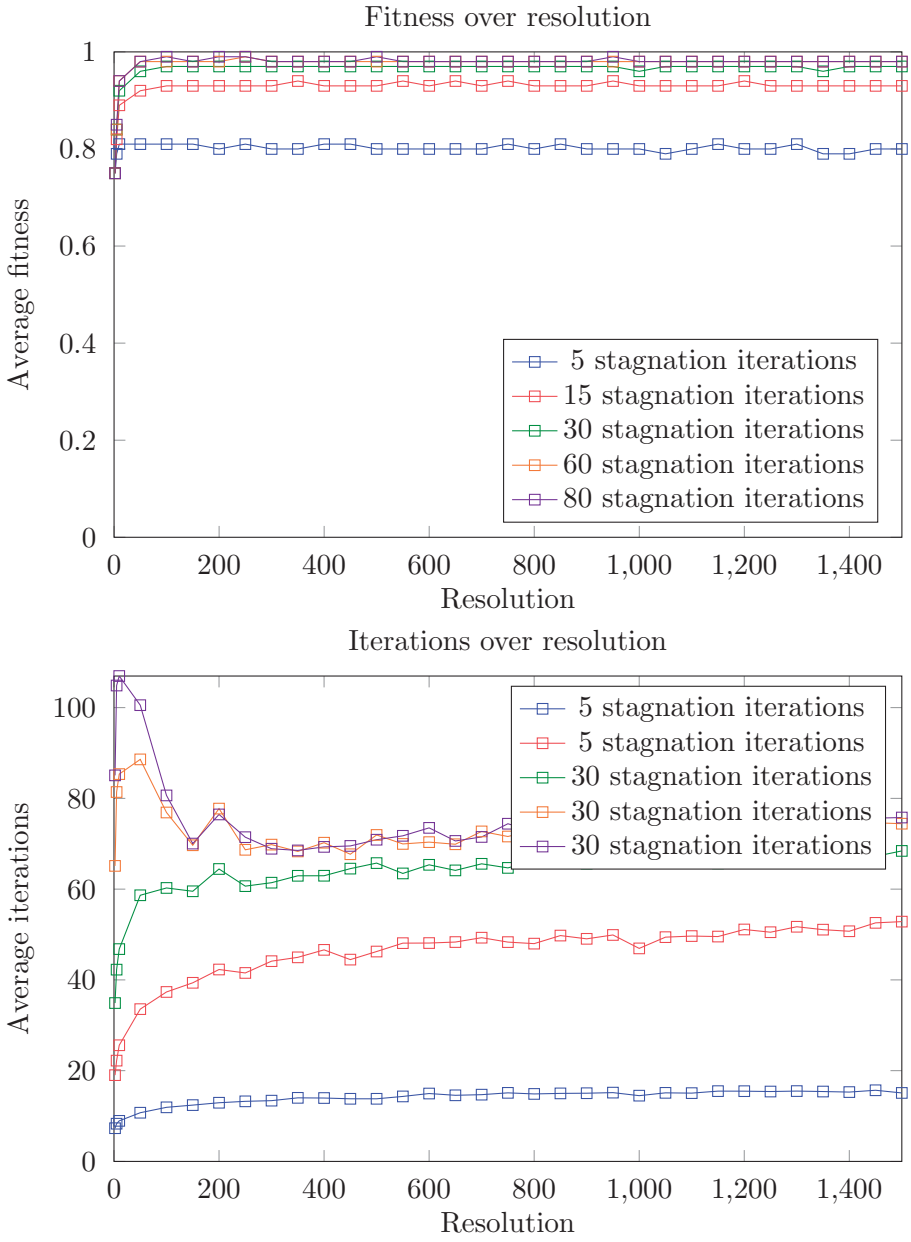


Figure 5.6: Performance metrics of the $(1 + 1)$ NA algorithm on the *Quarter* benchmark, with a single neuron, over different resolutions. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of stagnation iteration was reached.

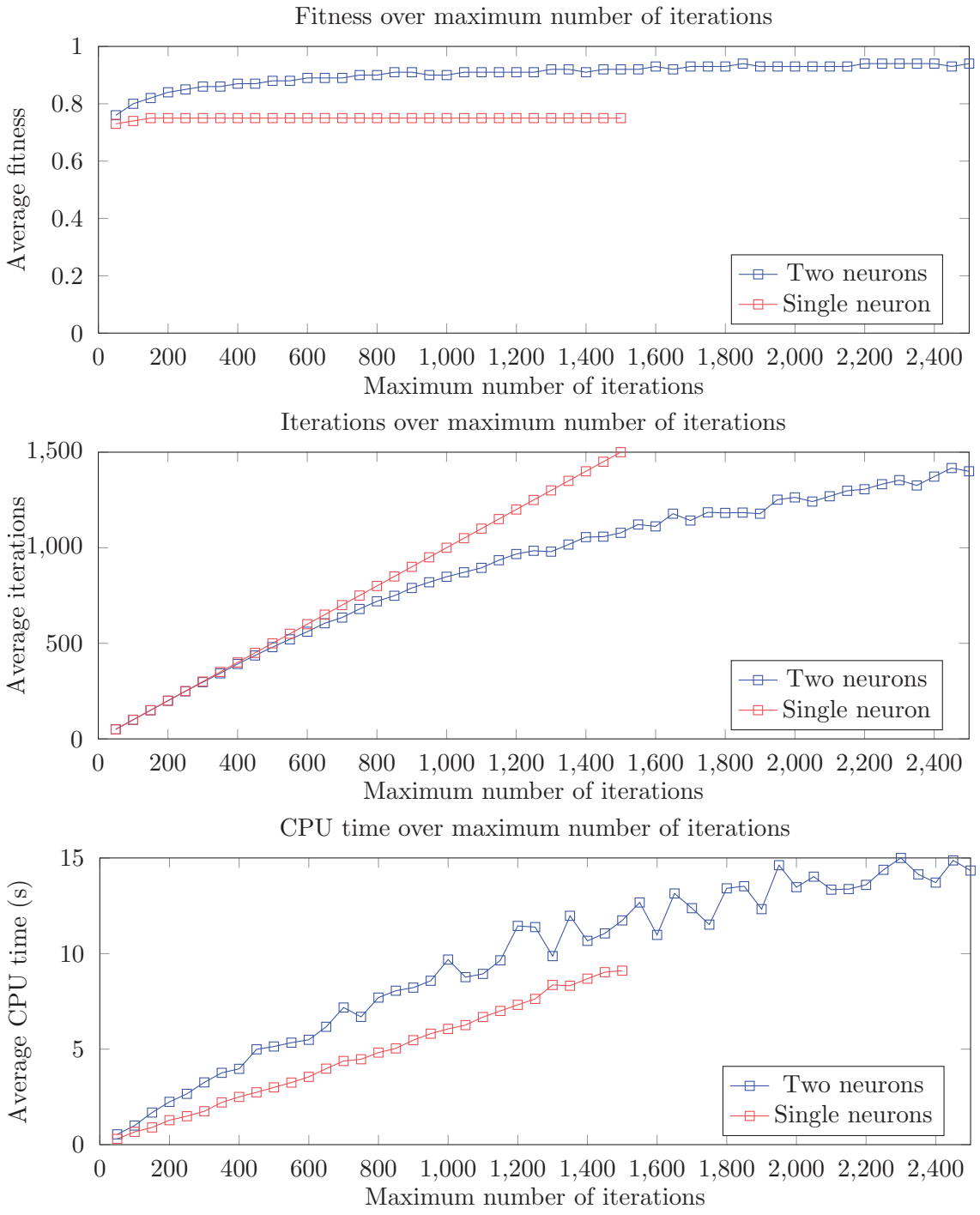


Figure 5.7: Performance metrics of the (1+1) NA algorithm on the *TwoQuarters* benchmark, over different maximum iterations. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of iterations was reached.

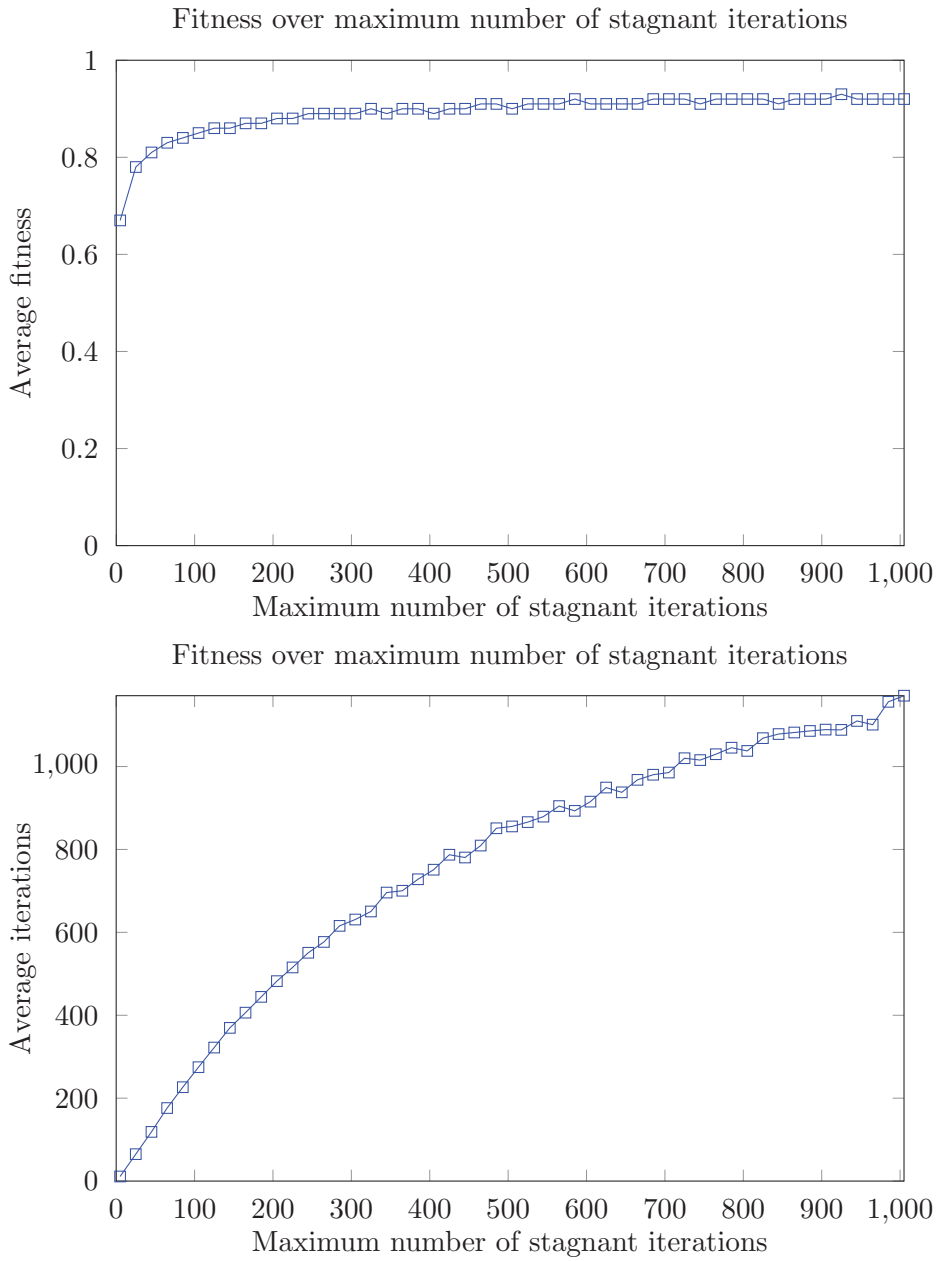


Figure 5.8: Performance metrics of the (1+1) NA algorithm on the *TwoQuarters* benchmark with two neurons, over different numbers of maximum stagnant iterations. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0, when 2500 iterations was reached, or the maximum number of stagnant iterations was reached.

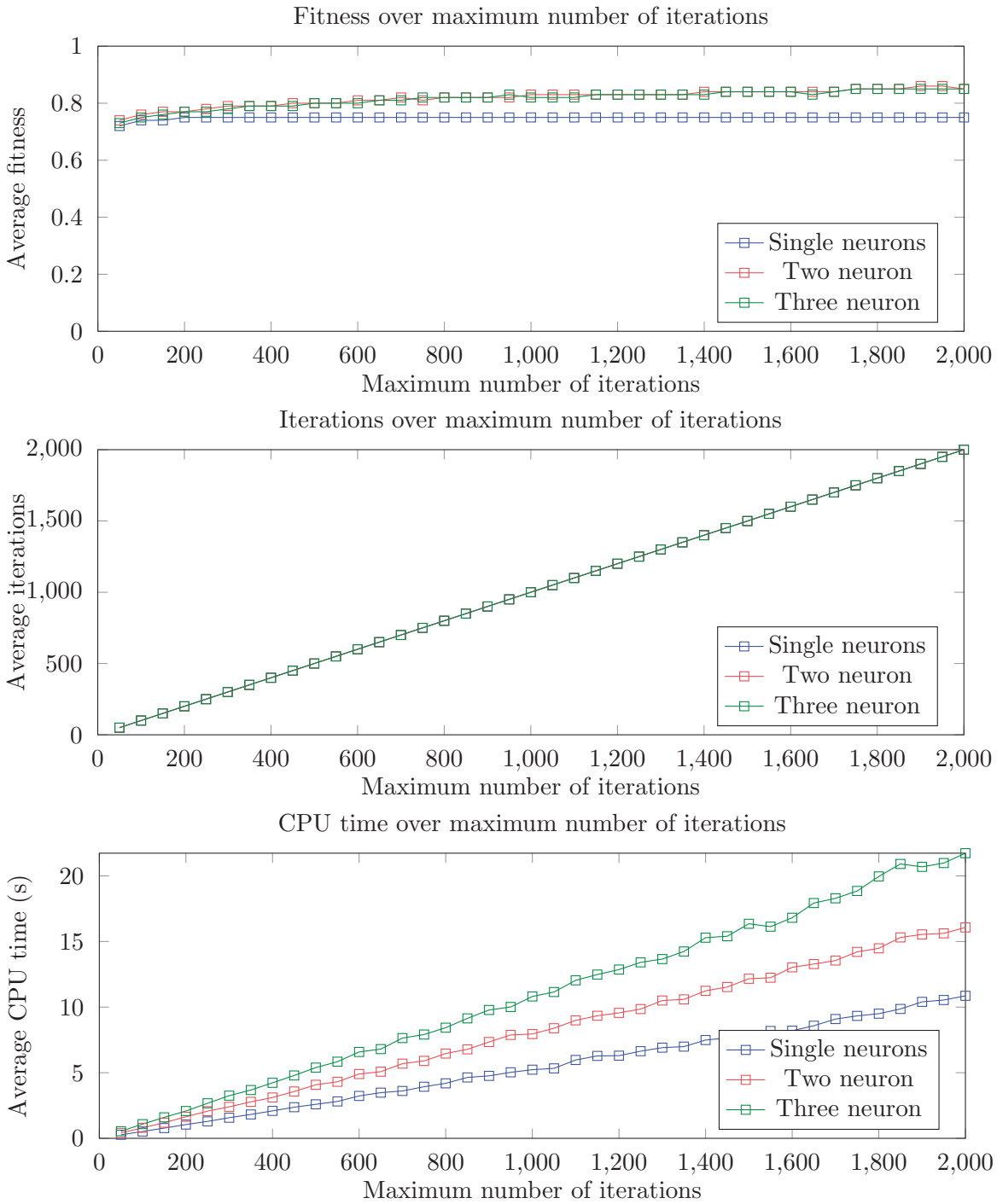
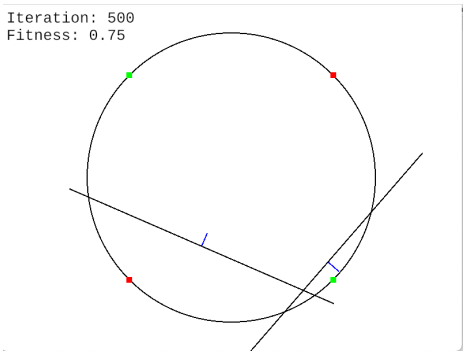
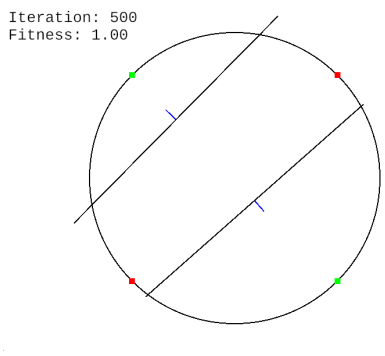


Figure 5.9: Performance metrics of the $(1 + 1)$ NA algorithm on the *LocalOpt* benchmark, over different maximum iterations. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of iterations was reached.



(a) Local optima



(b) Optimal solution

Figure 5.10: Visualization of two configurations of the $(1 + 1)$ NA algorithm, with two neurons, after evolution on the *XOR* problem.

Table 5.2: Results of the $1 + 1$ NA algorithm on the *Proben1 Cancer1* problem. The algorithm was tested with 10000 iterations.

Number of neurons	Average test fitness	Average CPU time (s)
1	0.96	72
2	0.96	137
3	0.92	205
4	0.84	258

Proben1 classification problem
The last problem on which the $(1 + 1)$ NA algorithm was tested is the *Proben1 Cancer1* benchmark. This problem differrs from the previous ones in the sense that the data which was used to evolve the algorithm is different from the data which was used to evaluate it. In addition, no optimal solution, which would allow us to set the number of neurons, is known in this case, and the input space is higher-dimensional, as it contains 9 features. Hence, an interesting experiment consists in testing the algorithm with different number of neurons. The results are shown in Figure 5.11. Fitness for both the train and test sets is shown.

First of all, it can be seen that the fitness on the train set and test set are very close, which shows that the algorithm did not ovrfit the train set.. In addition, the algorithm performs best with one neuron, where it reaches an average fitness of 094 on both sets. However, this is not coherent with the results of the previous experiments, a higher number of neurons should not result in a decrease in average fitness. This would suggest that the stagnation termination criteria, or at least setting it to a value of 200, is not well-suited for getting high fitness values on this problem. One could then try to use a maximum number iteration criteria or set the stagnation criteria to a higher value. Results for the maximum number of iterations set to 10000 are shown in Table 5.2 for one to three neurons. With this high number of maximum iterations, the algorithm was able to reach higher fitness values, which confirms that the stagnation criteria was too low. However, this value is still too low starting from three neurons. The one neuron case still gets the best result, but more importantly, the CPU time is already way higher than for the other problems.

5.2.2 Results of the BNA algorithm

The BNA algorithm can be applied to binary classification problems. As it is the case for the $(1 + 1)$ NA algorithm, the BNA algorithm was tested on the spere classification

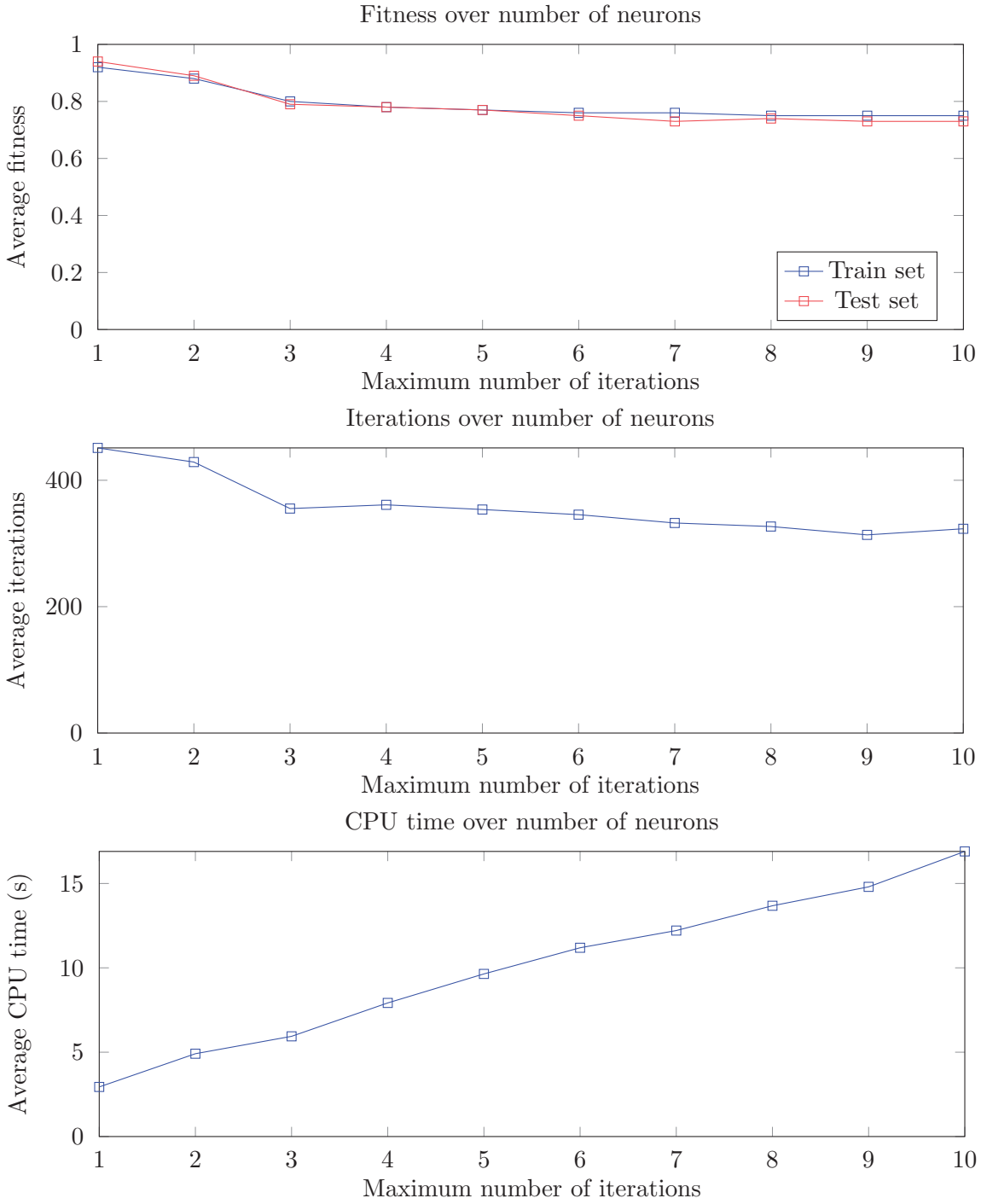


Figure 5.11: Performance metrics of the $(1 + 1)$ NA algorithm on the *Proben1 Cancer1* benchmark, over different number of neurons. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of 200 stagnant iterations was reached.

Table 5.3: Results of the BNA algorithm on the *XOR* problem. The algorithm was tested with a maximum number of stagnant iterations of 200.

Number of neurons	Average fitness	Average iterations	Average CPU time (s)
1	0.75	207	0.01
2	0.79	174	0.01

problems, the *XOR* problem and the *Cancer1* classification problem. The parameters which can be tuned for this algorithm are the resolution and the number of V-neurons, Varying the resolution would simply highlight the same properties as the ones observed for the $(1 + 1)$ NA algorithm. Thus the focus will be on the number of V-neurons and the the termination criteria for the algorithm. For the following experiments, the resolution was set to 400.

Unit-sphere classification problems
The first experiment consisted in testing the algorithm on the *Half* and *Quarter* benchmarks. The results are shown in Figure 5.12. The algorithm was tested with a maximum number of iterations ranging from 5 to 300. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of iterations was reached. For both problems, it can be seen that the fitness converges to 1.0 and the number of iterations taken by the algorithm plateaus around 110 iterations. Furthermore, as it was the case for the $(1 + 1)$ NA algorithm, the results on the two problems were almost identical, and the CPU time graphs reflect how the CPU time is proportional to the number of iterations.

The results for the *TwoQuarters* benchmark are shown in Figure 5.15 and the results for the *LocalOpt* benchmark are shown in Figure 5.16. For both problems, the same observations can be made as for the $(1 + 1)$ NA algorithm. For the *TwoQuarters* benchmark, with two V-neurons, the fitness converges to approximately 0.85, while the algorithm quickly reaches the maximal fitness of 0.75 in the case of one V-neuron. Regarding the LocalOpt benchmark, it was able to find the optimal solution once and also got stcuk in local optima, with an average fitness converging to 0.75 for all three cases. Examples of local optima and optimal solutions are shown in Figure 5.13 and Figure 5.14.

XOR problem
The *XOR* problem requires two V-neurons to be solved, an example of a solution is shown in Figure 5.18. The results of the algorithm, with one and two neurons, on the task are presented in Table 5.3. The algorithm was tested with a maximum number of stagnant iterations of 200. Using two neurons allows for a slight increase in fitness, from 0.75 to 0.79, while also resulting in a slight decrease in the number of iterations, but the CPU time remains the same. This shows that the algorithm gets stuck in a 0.75 fitness local optima, in most runs.

Proben1 classification problems
The results of the BNA algorithm on the *Proben1* classification problems, with different number of V-neurons and a maximum number of stagnant iterations of 200, are presented in Figure 5.19. Once again, the same observations as the experiment with the $(1 + 1)$ NA algorithm on the task can be made. For this configuration, the fitness drops with the number of V-neurons, while the number of iterations and CPU time increase. The fitness on the train and test sets are very close. Results in the case of a maximum iteration number of 10000 are presented in Table 5.4. This criterion allowed for higher fitness values. It was however not enough to allow the configurations with more than three V-neurons to reach a fitness as high as the one with a single V-neuron. But the CPU time is already particularly high and would make a further increase in the number of iterations impractical.

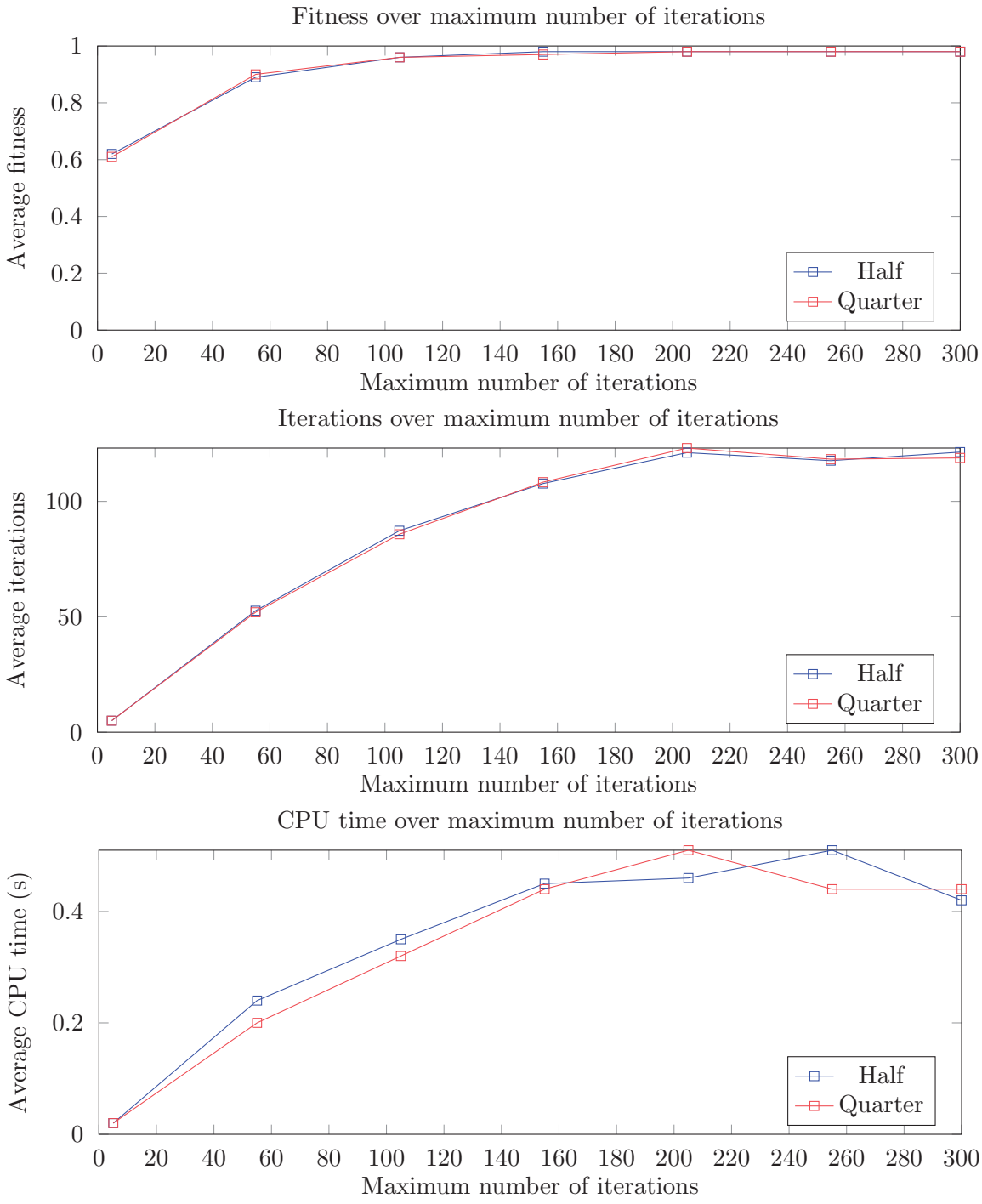
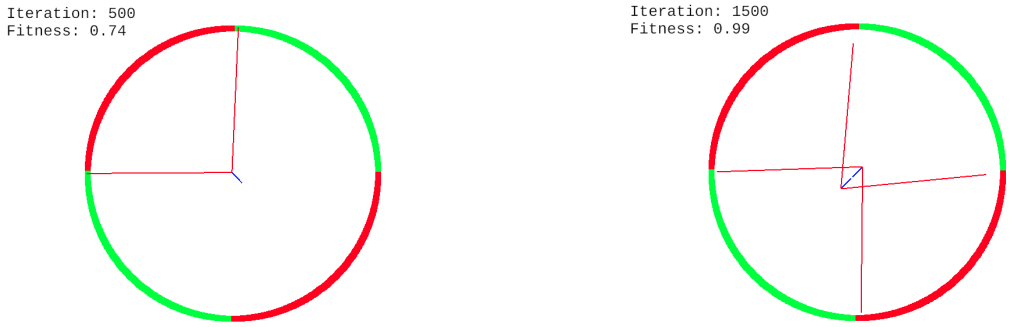
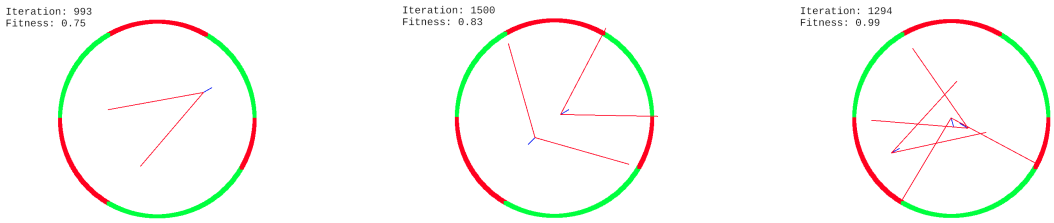


Figure 5.12: Performance metrics of the BNA algorithm on the *Half* and *Quarter* benchmarks, over different maximum iterations. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of iterations was reached.



(a) Local optima (b) Optimal solution

Figure 5.13: Visualization of two configurations of the BNA algorithm, with two V-neurons, after evolution on the *TwoQuarters* problem.



(a) Local optima. (b) Local optima. (c) Optimal solution.

Figure 5.14: Visualization of two configurations of the BNA algorithm, with three V-neurons, after evolution on the *LocalOpt* problem.

Table 5.4: Results of the BNA algorithm on the *Proben1 Cancer1* problem. The algorithm was tested with 10000 iterations.

Number of neurons	Average test fitness	Average CPU time (s)
1	0.96	199
2	0.94	415
3	0.85	660
4	0.83	862

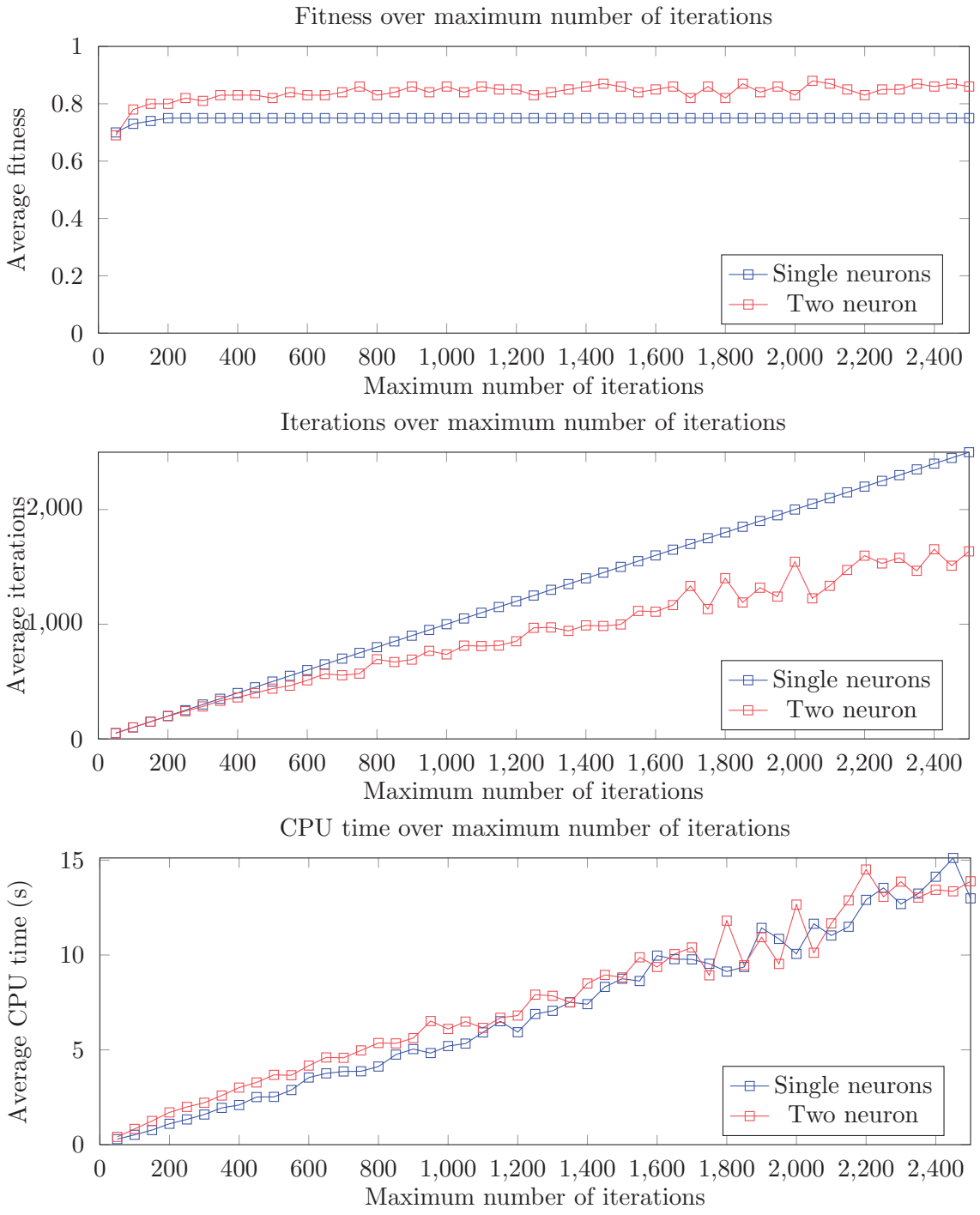


Figure 5.15: Performance metrics of the BNA algorithm on the *TwoQuarters* benchmark, over different maximum iterations. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of iterations was reached.

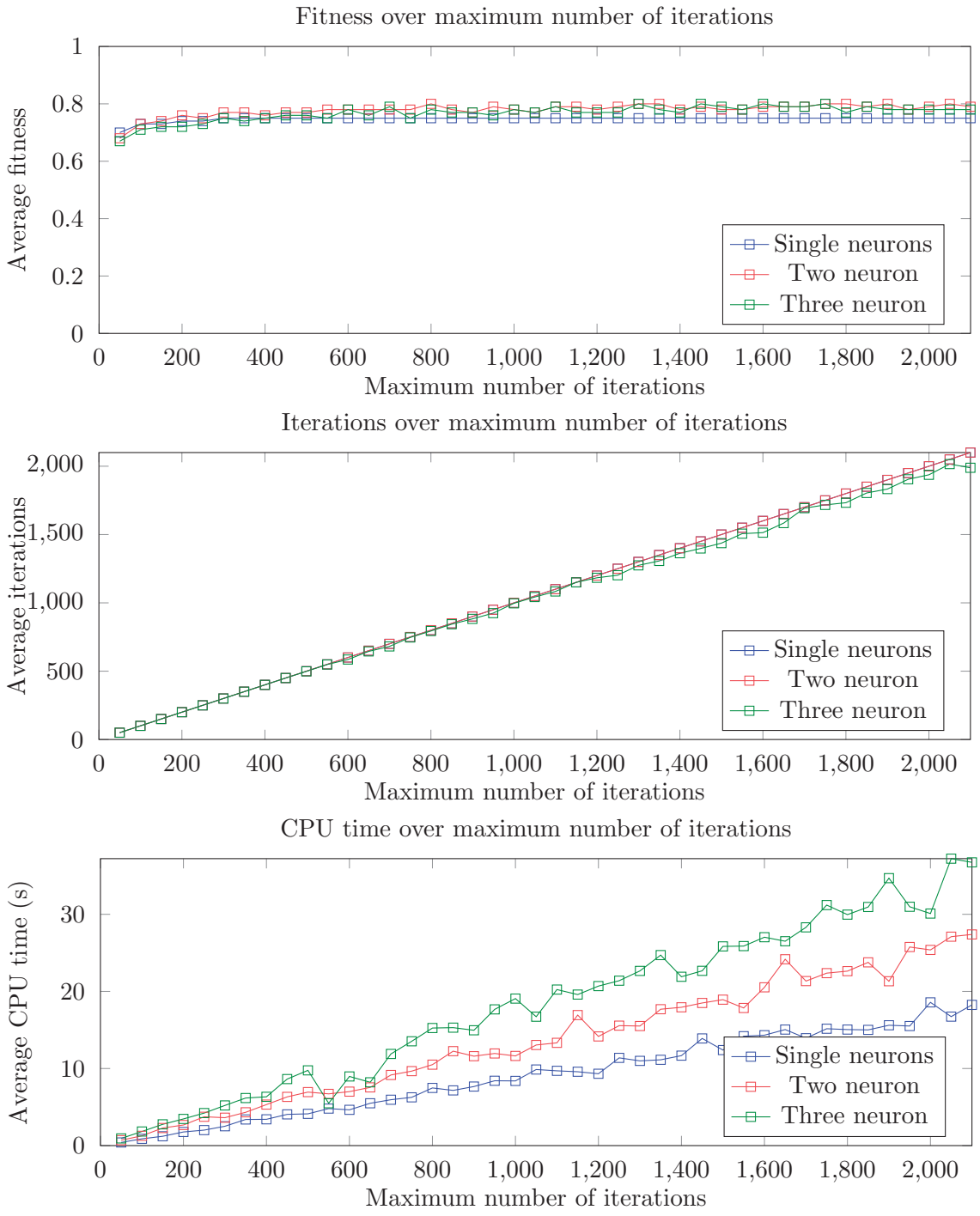


Figure 5.16: Performance metrics of the BNA algorithm on the *LocalOpt* benchmark, over different maximum iterations. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of iterations was reached.

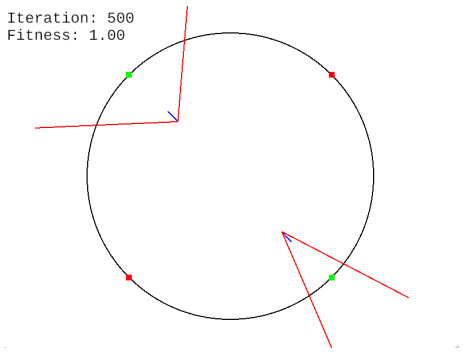


Figure 5.17: Local optima with fitness of 0.75

Figure 5.18: Visualization of a solution to the *XOR* problem, using the BNA algorithm with two V-neurons.

5.2.3 Results of the NEAT algorithm

The NEAT algorithm can be applied to all of the selected benchmark problems. Experiments are thus conducted on the unit-sphere classification problems, the *XOR* problem, the *Proben1 Cancer1* problem and the double pole balancing problem. Many parameters can be tuned for this algorithm, including:

- The population size
- The various mutation probabilities
- The weights for computing the similarity between genomes
- The similarity threshold
- The parameters of the connection weights distribution
- The parameters of the weights perturbation distribution
- The survival threshold

XOR problem

The NEAT algorithm was first tested on the *XOR* problem, as done in Stanley and Miikkulainen 2002. In addition to being used for evaluating the performance of the algorithm, different experiments aimed at understanding the impact of some of the parameters were conducted on this benchmark. The results are presented and discussed in this section.

Compared to the (1+1) NA and BNA algorithms discussed previously, the NEAT algorithm is population-based. Thus, the first experiment consisted in testing different population sizes.The algorithm was evolved with a maximum number of 500 generations. For the other parameters, the same values as in Stanley and Miikkulainen 2002 were used. The results are presented in Figure 5.20. The average fitness increases with the population size, while the number of iterations decreases as a consequence because of the maximum fitness tolerance termination criterion. However, the CPU time increases with the population size, because of the increased duration of an iteration. For example, a population of 50 individuals resulted in a fitness of 0.82 in 2.3 seconds on average, while a population of 1500 individuals resulted in a fitness of 0.96 in 66.6 seconds on average.

One of the main characteristics of the NEAT algorithm is the speciation mechanism, where individuals are separated into species based on their similarity. The second experiment consisted in testing different similarity thresholds, which would impact the number of

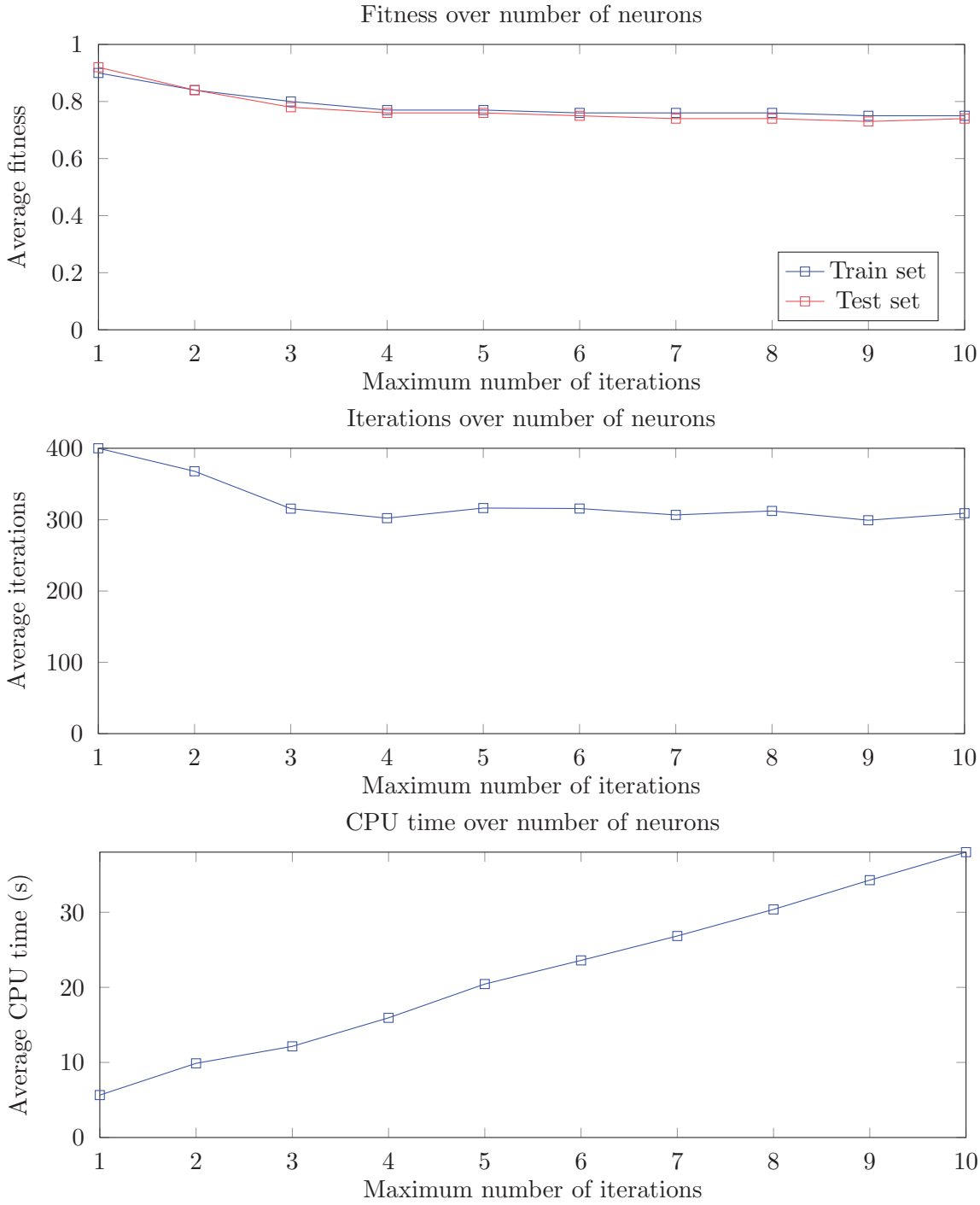


Figure 5.19: Performance metrics of the BNA algorithm on the *Proben1 Cancer1* benchmark, over different number of neurons. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of 200 stagnant iterations was reached.

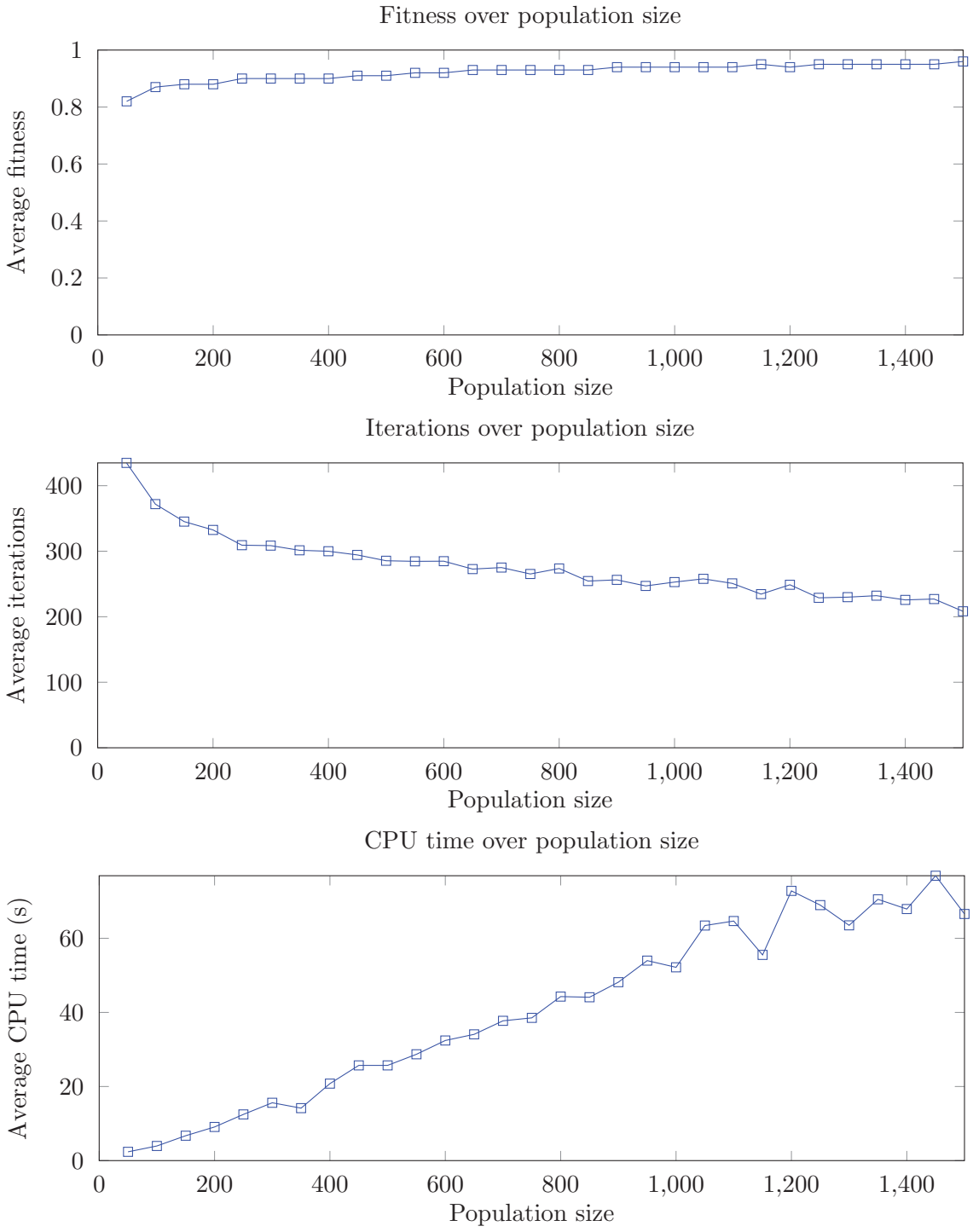


Figure 5.20: Performance metrics of the NEAT algorithm on the *XOR* benchmark, over different population sizes. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of 500 generations was reached. The other parameters were set to the same values as in Stanley and Miikkulainen 2002.

Table 5.5: Results of the NEAT algorithm on the *Pole balancing* problem. The algorithm was tested with the same configuration as in Stanley and Miikkulainen 2002, with a maximum of 100 generations.

Average fitness	Average iterations	Average CPU time (s)
1.00	11	6.19

species and the number of individuals in each species. Indeed, a higher similarity threshold would result in fewer species, while a lower similarity threshold would result in more species. The results are presented in Figure 5.21. The population size was set to 500 individuals and the maximum number of generations to 500. It can be observed that the highest fitness values are reached in the 2.5 to 8 range, where the iterations and CPU time are also the lowest. The fitness was lower for the smaller similarity thresholds, where there were more species, but with a lower number of individuals in each species. Starting from thresholds greater than 8, the performances remain stable, which is because of these values resulting in a single species.

Another parameter is the maximum number of generations (or iterations).Figure 5.22 presents the results of the algorithm on the problem, with different maximum numbers of generations, a population size of 300 and a similarity threshold of 8.0. As expected, all three metrics increase with the number of generations, particularly in the 50 to 200 range, before stabilizing. The algorithm was able to solve the problem in less than 150 generations on average.

Finally, the most important characteristic of the algorithm is its ability to evolve topologies, by adding connections or nodes through mutations. This final experiment consisted in testing different values for the probability of adding a new node (and its connections) to the genome. The results are presented in Figure 5.23. The population size was set to 300 individuals, the similarity threshold to 8.0 and the maximum number of generations to 300. As it was the case for the other parameters, there is a range of values for the new node mutation probability that result in the best performance, in this case around the value of 0.05, which is consistent with the choice made in Stanley and Miikkulainen 2002 of 0.03. Furthermore, for a value of 0.0, no hidden nodes are added to the network, and because of the starting topology only consists in a input and output layer, the algorithm cannot solve the problem, which requires hidden neurons. For higher values, too many nodes are added, which results in an increase in the CPU time, and a decrease in the fitness, because of the increased complexity of the networks.

Double pole balancing problem

The NEAT algorithm was also tested on the double pole balancing control problem. The configuration described in Stanley and Miikkulainen 2002 was found to be particularly efficient on this task, the results using it are presented in Table 5.5. The algorithm was always able to solve the problem, and in 11 iterations on average. The CPU time was 6.19 seconds on average, which is high when compared to the previous benchmarks, when considering the same number of iterations. This is because of the fitness evaluation being more computationally expensive for this problem, as it requires simulating the pole balancing system.

Unit sphere classification problems

The algorithm was tested on the *Half*, *Quarter* and *TwoQuarters* and *LocalOpt* unit sphere classification problems. The performance metrics are presented in Table 5.6.

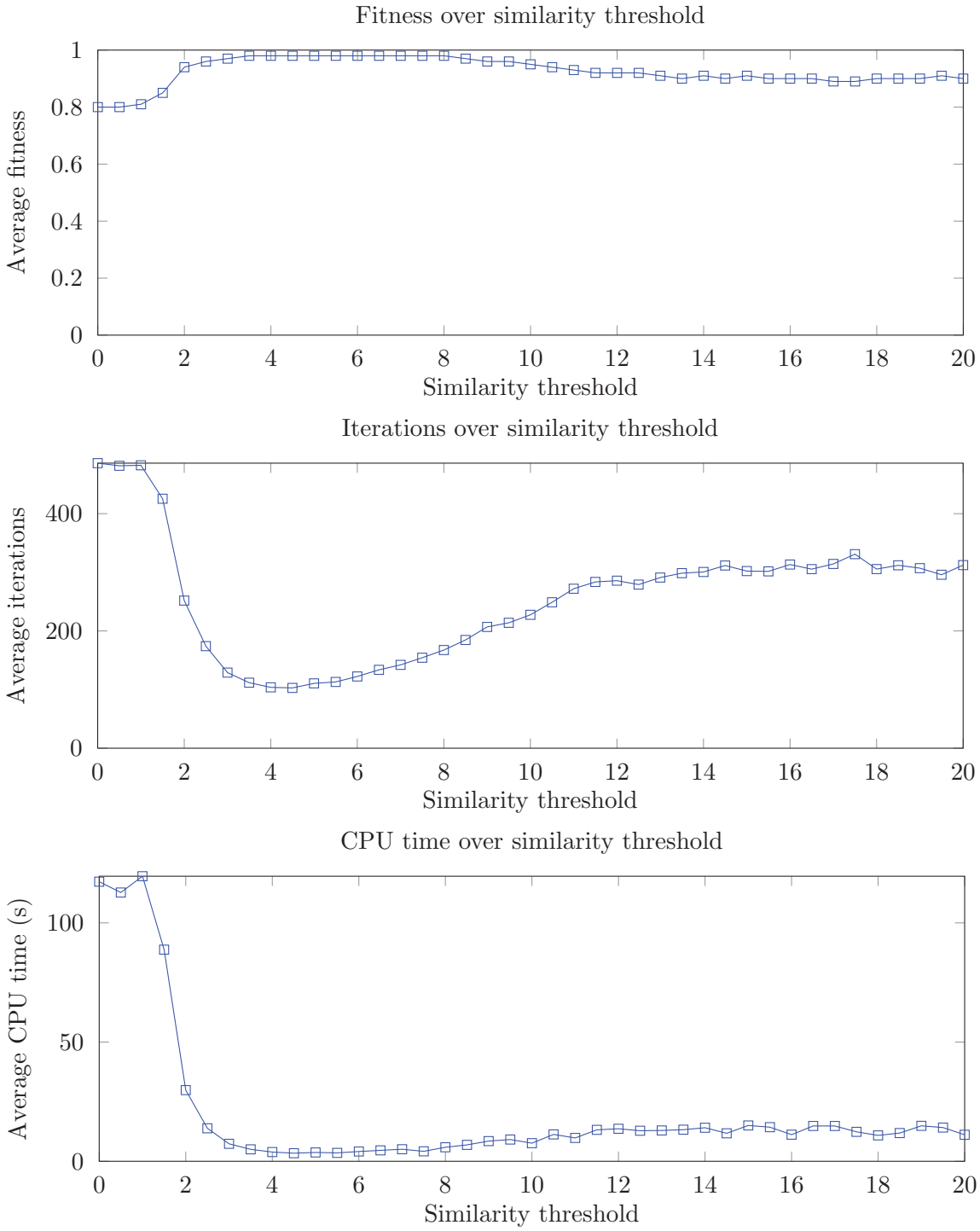


Figure 5.21: Performance metrics of the NEAT algorithm on the *XOR* benchmark, over different similarity thresholds. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of 500 generations was reached. The other parameters were set to the same values as in Stanley and Miikkulainen 2002.

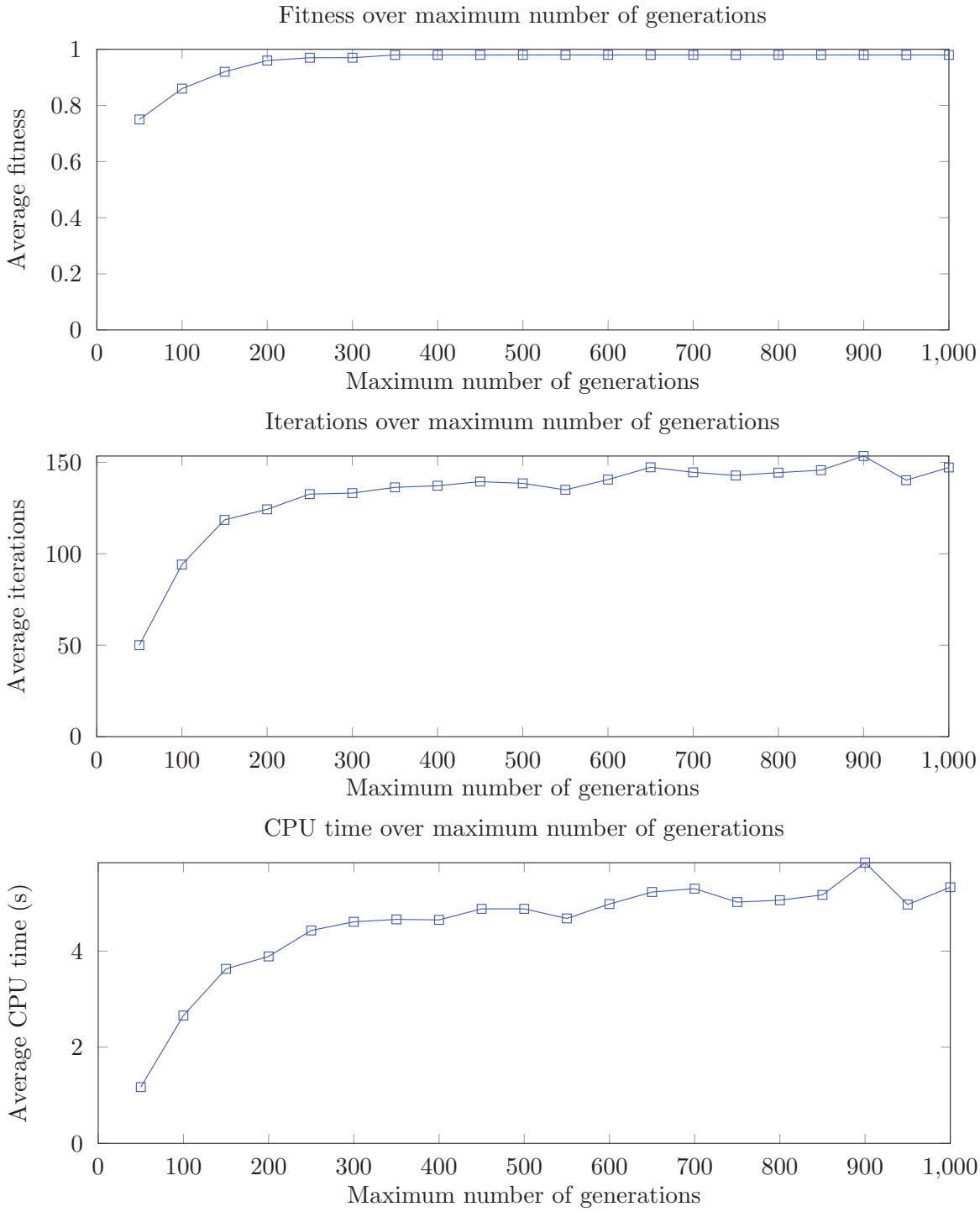


Figure 5.22: Performance metrics of the NEAT algorithm on the *XOR* benchmark, over different maximum generations. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of generations was reached. The other parameters were set to the same values as in Stanley and Miikkulainen 2002.

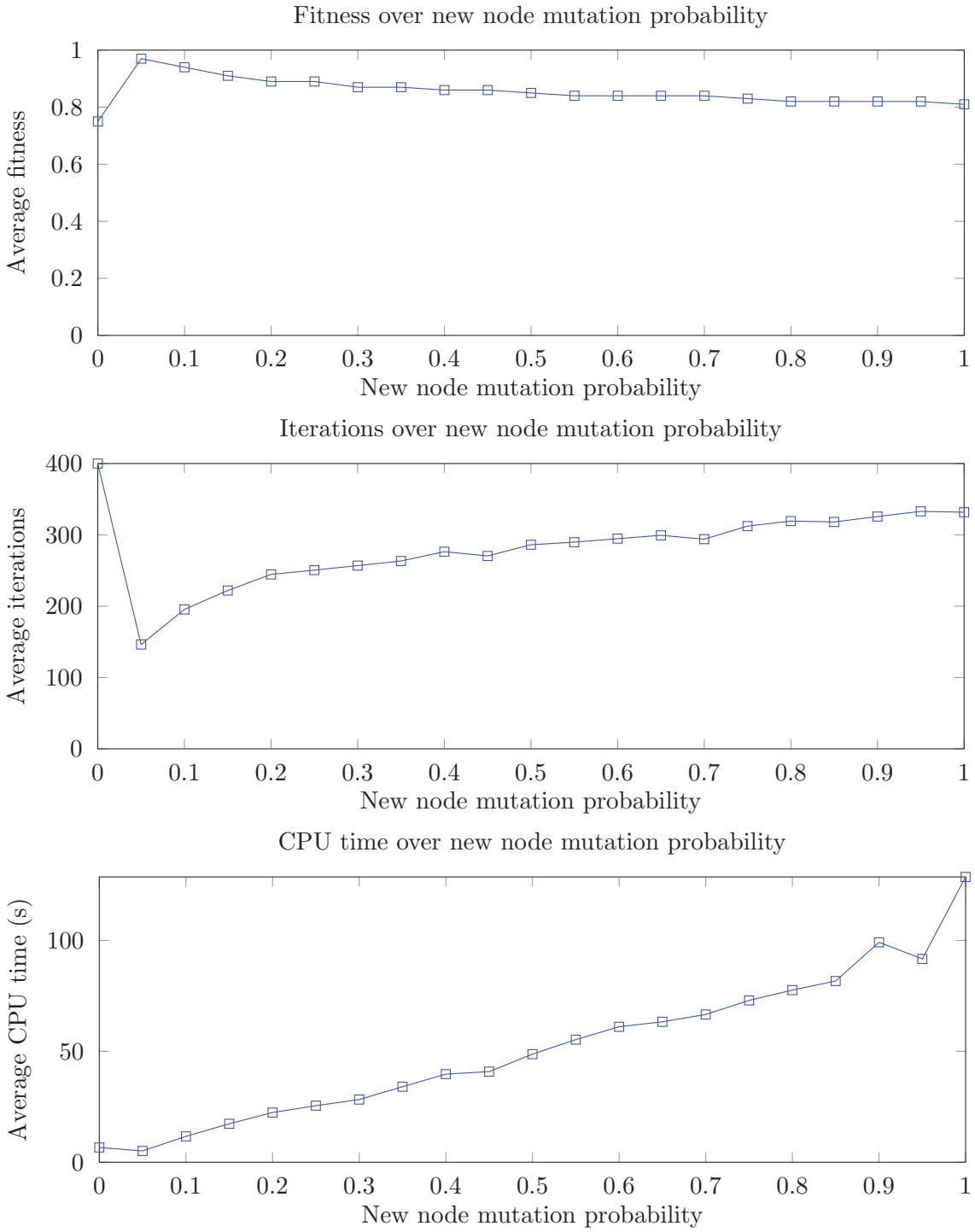


Figure 5.23: Performance metrics of the NEAT algorithm on the *XOR* benchmark, over different new node mutation probabilities. The evolution stopped when the fitness was 2% away from the maximal fitness of 1.0 or the maximum number of generations was reached. The other parameters were set to the same values as in Stanley and Miikkulainen 2002.

Table 5.6: Results of the NEAT algorithm on the unit-sphere classification problems.

Problem	Average fitness	Average iterations	Average CPU time (s)
Half	0.98	24	127
Quarter	0.98	9	39
TwoQuarters	0.74	100	935
LocalOpt	0.65	100	1134

Table 5.7: Results of the NEAT algorithm on the *Proben1 Cancer1* problem.

Average test set fitness	Average iterations	Average CPU time (s)
0.98	7	35.8

Proben1 classification problems
Finally, the NEAT algorithm was tested on the *Proben1 Cancer1* problem. The results are presented in Table 5.7. The algorithm was able to solve the problem in all of the runs, in the particularly low number of 7 Iterations on average. However, the CPU time was high. As a matter of fact, it was higher than for the pole balancing problem, with an average of 35.8 seconds, despite the lower number of iterations.

5.2.4 Results of the CMA-ES algorithm

The CMA-ES algorithm can be used to evolve the connection weights of networks, which can be used to solve all the selected benchmark problems. Experiments are thus conducted on the unit-sphere classification problems, the *XOR* problem, the *Proben1 Cancer1* problem and the double pole balancing problem. Many parameters can be tuned for this algorithm. However, the focus was put on the fixed network topologies, while using the default parameters for the algorithm provided by the `cmaes` crate [2]. For all of the following experiments, the sigmoid activation function was used for hidden and output neurons.

XOR problem
The CMA-ES algorithm was first tested on the evolution of connection weights for the *XOR* problem. Fully connected networks with 0 to 5 hidden neurons were tested. The results are presented in Figure 5.24. For 0 to 2 hidden neurons, the algorithm was only able to find the 0.75 fitness local optima. The number of iterations and the CPU time increased with the number of hidden neurons, but both remained particularly low. For the 4 and 5 hidden neurons case, the algorithm was able to find an optimal solution in the majority of the runs.

Double pole balancing problem
Fully connected networks with 0 to 6 hidden neurons were considered for the double pole balancing problem. The results are presented in Figure 5.25. The main observation which can be made is that when initially switching from a two-layers to a three-layers network, the fitness drops significantly from 0.67 to 0.1 but increases as more hidden neurons are added and eventually reaches 0.89 for 6 hidden neurons. Furthermore, in addition to the lower fitness values compared to the simpler *XOR* problem, the number of iterations and CPU time were significantly higher, with the CPU time reaching 300 seconds for 6 hidden neurons.

Unit sphere classification problems
The algorithm was tested with networks ranging from 0 to 6 hidden neurons on the *Half*, *Quarter*, *TwoQuarters* and *LocalOpt* unit sphere classification problems. The results are presented in Figure 5.26. Firstly, it can be noted how similar the tendencies for these

[2] `https://docs.rs/cmaes/latest/cmaes/options/struct.CMAESOptions.html`

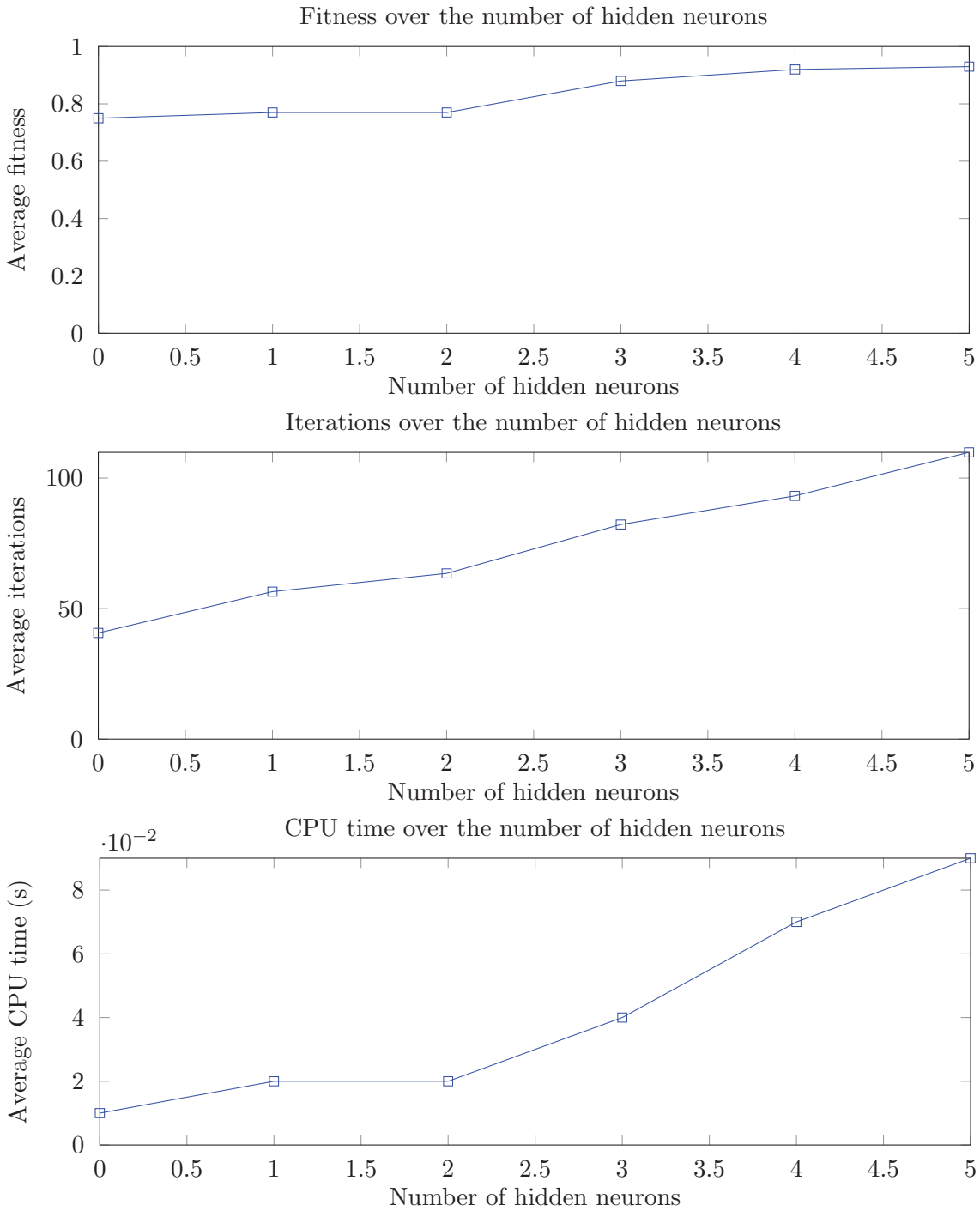


Figure 5.24: Performance metrics of the CMA-ES algorithm on the *XOR* benchmark, over different number of hidden neurons.

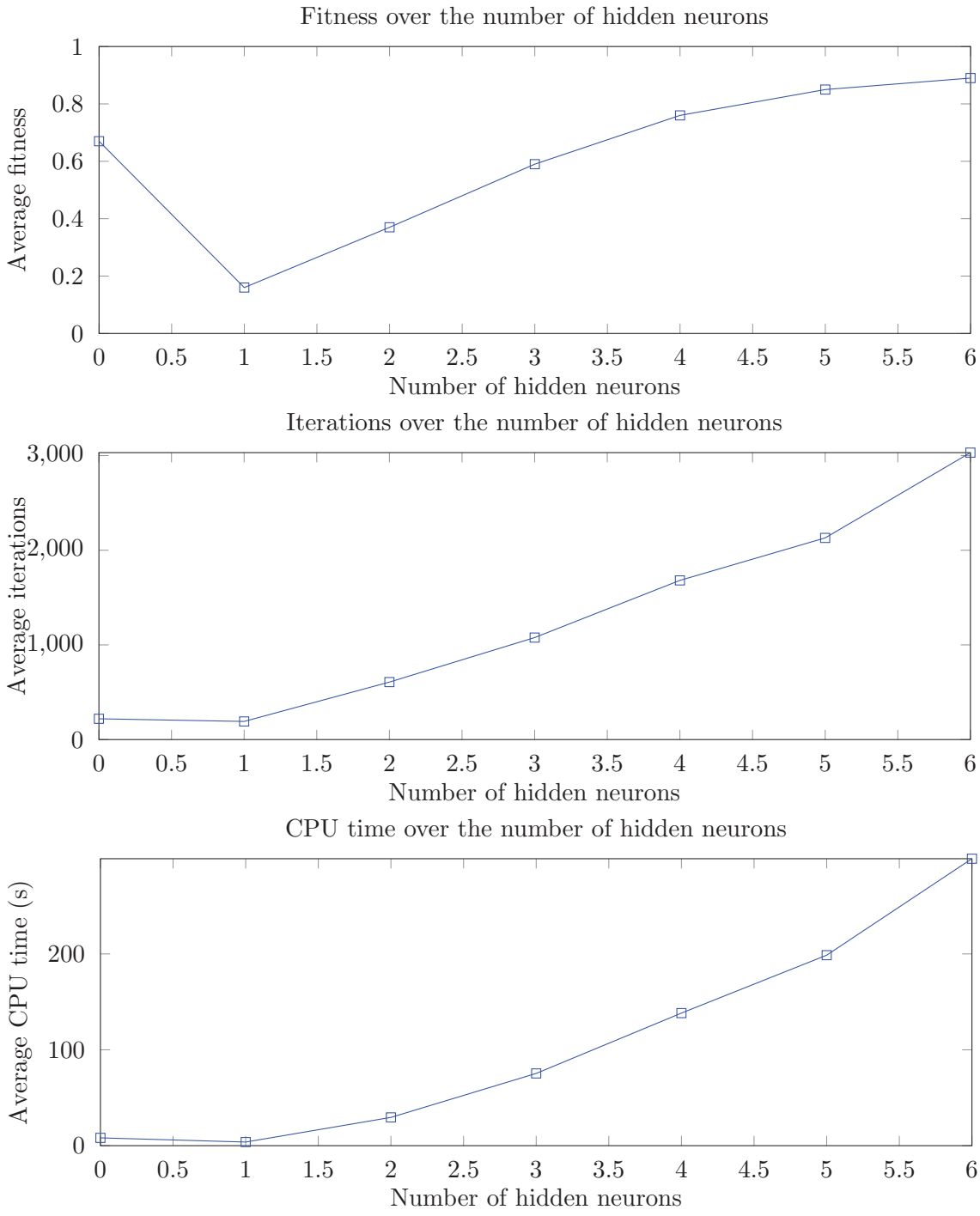


Figure 5.25: Performance metrics of the CMA-ES algorithm on the *Double Pole Balancing* benchmark, over different number of hidden neurons.

Table 5.8: Comparison of the performance of the algorithms on the unit-sphere classification problems.

	Problem							
	Half		***Quarter***		***TwoQuarters***		***LocalOpt***	
Algorithm	Fitness	CPU	Fitness	CPU	Fitness	CPU	Fitness	CPU
(1 +1) NA	1.0	0.4	1.0	0.4	0.94	14	0.85	15
BNA	1.0	0.4	1.0	0.4	0.85	14	0.80	28
NEAT	1.0	127	1.0	39	0.74	935	0.65	1134
CMA-ES	1.0	6	1.0	6	0.77	6	0.75	68

metrics are for the four problems, and how they reflect the problem complexities. For the *Half* and *Quarter* problems, the algorithm was able to always find the optimal solution, except for the 1 hidden neuron case, in less than 400 iterations and 45 seconds, on average. However, for the *TwoQuarters* and *LocalOpt* problems, the algorithm was never able to find an optimal solution, for the LocalOpt problem the fitness was even lower than 0.75 for all but the 6 hidden neurons case. As a consequence, the number of iterations and CPU time were higher than for the two other problems.

Proben1 classification problems
Finally, the CMA-ES algorithm was tested on the *Proben1 Cancer1* problem, with fully connected networks with 0 to 3 hidden neurons. The results are presented in Figure 5.27. Here, it was able to average a fitness of 0.98, on the test set, for the $0, 2$ and 3 hidden neurons cases, and 0.78 for the 1 hidden neuron case. The number of iterations and CPU time increased with the number of hidden neurons, with the CPU time reaching 20.79 seconds for 3 hidden neurons, with 319 iterations.

5.3 Discussion

This section presents a comparison of the performance of the algorithms on the different benchmarks. Then, based on this comparison, and the insights from the various experiments conducted and analyzed in Section 5.2, guidelines are proposed for the selection of the algorithm based on the problem at hand, and for the selection of the parameters for the algorithms.

The comparison is based on the fitness and CPU time metrics, as the number of iterations/-generations is not directly comparable between the different algorithms. Furthermore, for each of the algorithm, the most efficient configuration(s) were used to represent it in the comparison, where the efficiency is defined as a trade-off between the fitness and the CPU time.

5.3.1 Unit-sphere classification problems

...

5.3.2 XOR problem

...

5.3.3 Proben1 classification problems

...

5.3.4 Double pole balancing problem

...

5.3.5 Guidelines

...

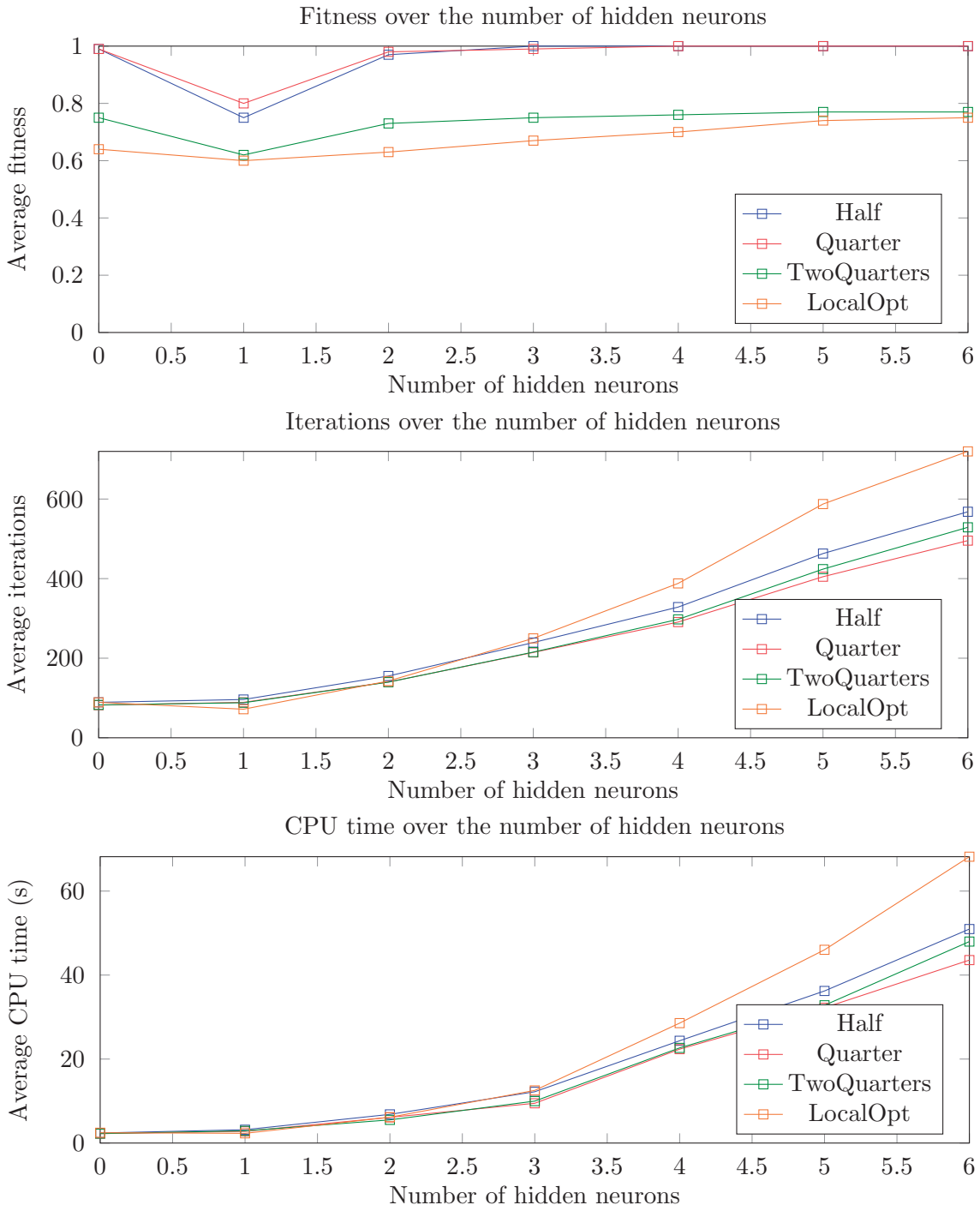


Figure 5.26: Performance metrics of the CMA-ES algorithm on the unit-sphere classification benchmarks, over different number of hidden neurons.

Table 5.9: Comparison of the performance of the algorithms on the *XOR* benchmark.

Algoeithm	Fitness	CPU time (s)
(1 + 1) NA	0.83	0.01
BNA	0.79	0.01
NEAT	0.96	67
CMA-ES	0.93	0.09

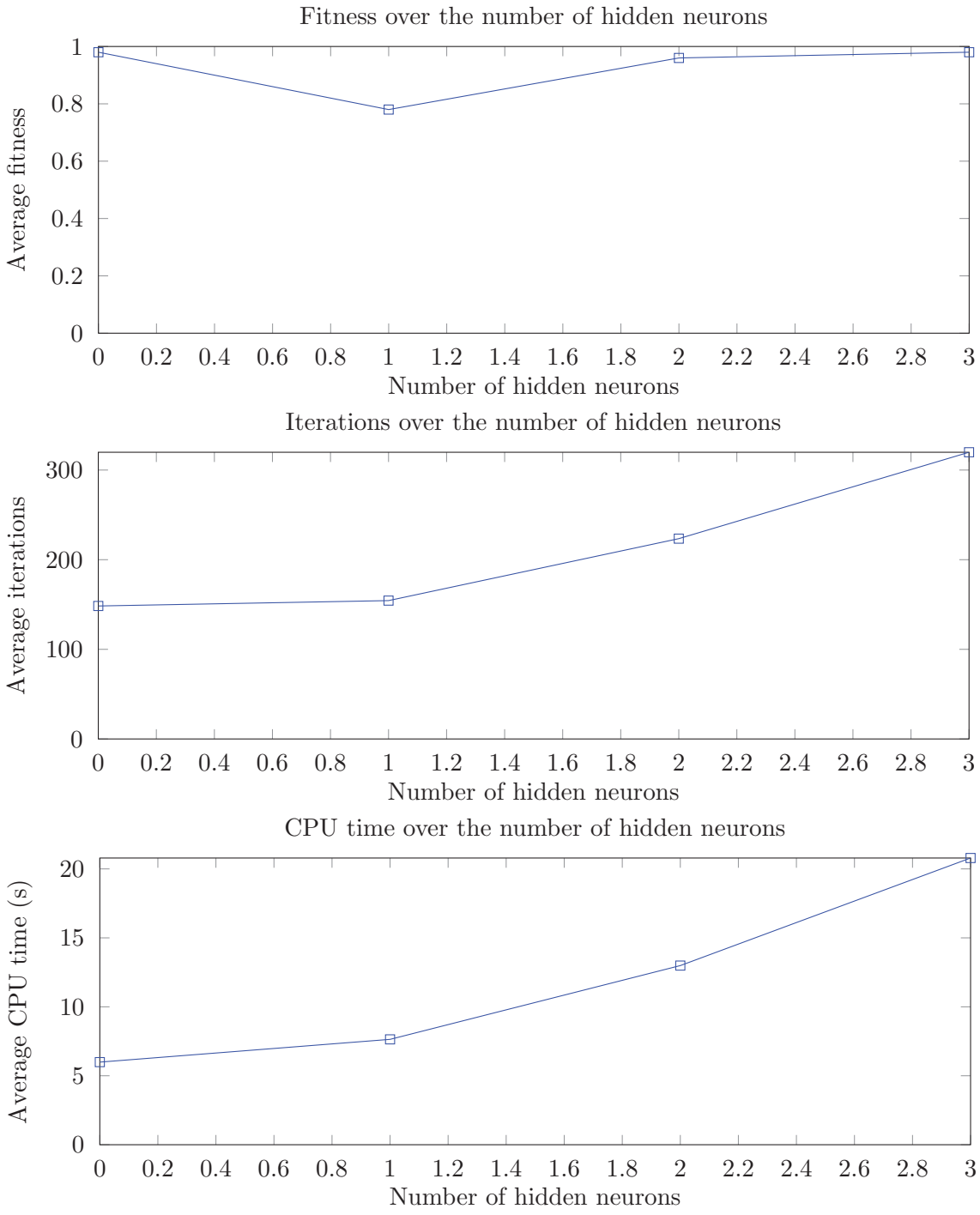


Figure 5.27: Performance metrics of the CMA-ES algorithm on the *Proben1* benchmark, over different number of hidden neurons.

Table 5.10: Comparison of the performance of the algorithms on the *Proben1 Cancer1* benchmark.

Algoeithm	Fitness	CPU time (s)
(1 + 1) NA	0.96	72
BNA	0.96	199
NEAT	0.98	37
CMA-ES	0.98	21

Table 5.11: Comparison of the performance of the algorithms on the *Double Pole Balancing* benchmark.

Algoeithm	Fitness	CPU time (s)
NEAT	1.0	6.4
CMA-ES	0.89	300

Bibliography

Wohlin, Claes (2014). "Guidelines for snowballing in systematic literature studies and a replication in software engineering". In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE '14. Association for Computing Machinery.

Petersen, Kai, Sairam Vakkalanka, and Ludwik Kuzniarz (2015). "Guidelines for conducting systematic mapping studies in software engineering: An update". In: *Information and Software Technology* 64, pp. 1–18.

Stanley, Kenneth O. and Risto Miikkulainen (2002). "Evolving Neural Networks through Augmenting Topologies". In: *Evolutionary Computation* 10.2, pp. 99–127.

Fischer, Paul, Emil Lundt Larsen, and Carsten Witt (2023). "First Steps Towards a Runtime Analysis of Neuroevolution". In: *Proceedings of the 17th ACM/SIGEVO Conference on Foundations of Genetic Algorithms"*. Association for Computing Machinery", "61–72".

Prechelt, Lutz (1994). *PROBEN 1 - a set of benchmarks and benchmarking rules for neural network training algorithms*. Karlsruhe 1994. (Technical report. Fakultät für Informatik, Universität Karlsruhe. 1994,21.)

Wieland, A.P. (1991). "Evolving neural network controllers for unstable systems". In: *IJCNN-91-Seattle International Joint Conference on Neural Networks*. Vol. ii, 667–673 vol.2.

Appendix A

Title

Technical
University of
Denmark

Richard Petersens Plads, Building 324
2800 Kgs. Lyngby
Tlf. 4525 1700

www.compute.dtu.dk