Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Bachelor Thesis

EVOLUTIONARY COMPUTATION IN CONTINUOUS OPTIMIZATION AND MACHINE LEARNING

Leslie Dahlberg
ldg14001@student.mdh.se

Examiner:

Mälardalen University
Västerås, Sweden

Supervisor: Ning Xiong

Mälardalen University
Västerås, Sweden

March 10, 2017

Abstract

Det här avsnittet ska helt enkelt vara just detta: en sammanfattning av hela rapporten. En lämplig omfattning är c:a 200 - 250 ord. En bra tumregel är att sammanfattningen ska hållas så kort det går, den ska vara kompakt men fortfarande tydlig, informativ och väcka intresse. Ge de viktigaste fakta och summera allt det som är väsentligt i rapporten. Följande bör ingå:

- *Presentation/introduktion av området för arbetet,*
- *Översiktlig presentation av uppgiften inklusive syfte och frågeställning,*
- *Motivation till varför området och uppgiften är viktiga och intressanta,*
- *Generell beskrivning av hur du angripit uppgiften, vad du har gjort,*
- *Sammanfattning av resultat och slutsatser och vad ditt arbete bidrar med.*

Inga detaljer ska vara med i sammanfattningen, inte heller beskrivning av hur rapporten är uppställd.

Sammanfattningen ska kunna läsas helt fristående från resten av rapporten, och av en ganska bred grupp av läsare. Den ska ge en bra grund för att en läsare ska kunna bedöma om hen är intresserad av att läsa hela rapporten.

Sammanfattningen är den del av en rapport som läses allra mest och av flest personer. Därför är det extra viktigt att du skriver en bra sammanfattning. Du behöver ha ett ordentligt grepp om innehållet i rapporten när du skriver sammanfattningen, och när hela rapporten är klar bör du granska och vid behov revidera sammanfattningen så att den överensstämmer med rapporten.

Contents

1 Introduction

Optimization is a problem-solving method which aims to find the most advantageous parameters for a given model. The model is known to the optimizer and accepts inputs while producing outputs. Usually the problem can be formulated in such a way that we seek to minimize the output value of the model or the output of some function which transforms the models output into a fitness score. Because of this the process is often referred to as minimization. It becomes obvious that this is useful when considering optimizing the layout of a circuit in order to minimize the power consumption. To achieve this the optimizer looks for combinations of parameters which let the model produce the best output to a given input [1].

When dealing with simple mathematical models, optimization can be achieved using analytical methods, often calculating the derivative of the functional model, but these methods prove difficult to adapt to complex models which exhibit noisy behavior. Additionally, the analytical model is not always known, which makes it impossible to use such methods. The field of evolutionary computation (EC), a subset of computational intelligence (CI), which is further a subset of artificial intelligence (AI), contains algorithms which are well suited to solving these kinds of optimization problems [2, 3].

Evolutionary computation focuses on problem solving algorithms which draw inspiration from natural processes. It is closely related to the neighboring field of swarm intelligence (SI), which often is, and in this thesis will be, included as a subset of EC. The basic rationale of the field is to adapt the mathematical models of biological Darwinian evolution to optimization problems. The usefulness of this can be illustrated by imagining that an organism acts as an “input” to the “model” of it’s natural environment and produces an “output” in the form of offspring. Through multiple iterations biological evolution culls the population of organisms, only keeping the fit specimen, to produce organisms which become continuously better adapted to their environments. Evolutionary computation is, however, not merely confined to Darwinian evolution, but also includes a multitude of methods which draw from other natural processes such as cultural evolution and animal behavior [4].

The purpose of this thesis is to explore the performance and usefulness of three emerging evolutionary algorithms: differential evolution, particle swarm optimization and estimation of distribution algorithms. The intention is to test and compare these against each other on a set of benchmark functions and practical problems in machine learning and then, if possible, develop a new or modified algorithm which improves upon the aforementioned ones in some aspect.

Research has been conducted on improving various evolutionary algorithms by hybridizing and extending them, which has resulten in a wide array of algorithms for both specific and general purposes. Since these algorithms accepts parameters which modify their efficiency, studies have been carried out which compare different combinations of parameters. My thesis will draw upon this work by comparing three algorithms both generally and on a very specifi problem.

2 Background

This section will aim to provide a general overview of the field of evolutionary compution. General terms and procedured which are often utizilied in EC will be explained and the most well known traditional approaches will be presented. Current research will also be introduced. The emerging algorithms relevant to this thesis can be found in the section “Algorithms”.

2.1 The basic structure of evolutionary algorithms

Evolutionary algorithms work on the concept of populations. A population is a set of individuals which in the case of optimization problems contain a vector of parameters which the model we wish to optimize can accept and transform into an output. The population is initialized by some procedure to contain a random set of parameter vectors, these should cover the whole parameter range of the model uniformely. The inital population is evaluated and an iterative process is started which continues as long as no suitable solution is found. During this iterative process the current population is selected, altered and evaluated. During selection a set of individuals which display promising characterisitcs are selected to live on to the next generation of the population. They are then altered randomly to create diversity in the population and evaluated. This process creates a new generation of the population on each iteration and continues until a solution is found or some other restriction is encoutered [5]. The concept is demonstrated in figure 1 with $P(t)$ representing the population at generation t.

```
$t \leftarrow 0$
initialize $P(t)$
evaluate $P(t)$
**while** termination-condition not fulfilled **do**
    $t \leftarrow t + 1$
    select $P(t)$ from $P(t-1)$
    alter $P(t)$
    evaluate $P(t)$
**end while**
```

Figure 1: Basic evolutionary algorithm

2.2 Components of evolutionary algorithms

Here the fundamental builing blocks of evolutionary algorithms will be presented and explained. Most of these term are universal to all approaches which will be covered in this thesis and neccesary to properly understand them.

2.2.1 Representation

The first step in using evolutionary algorithms is creating a representation which can encode all possible solutions to the problem at hand. Here two different terms are distinguished. The term phenotype denotes the representation that can be directly applied to the problem and the genotype denotes the specific encoding of the phenotype which is manipulated inside the evolutionary algorithm. In optimization tasks a valid phenotype could be a vector of integer numbers which act as parameters to a function while the genotype would be a binary representation of the numbers which can be altered by manipulating individuals bits. The terms phenotype, candidate solution and individual are used interchangably to denote the representation as used by the model while chromosome, genotype and individual are used to refer to the representation inside the evolutionary algorithm [6].

Binary representation Genetic algorithms (GA) traditionally use a binary representation to store individual genotypes. The representations is a string of fixed length over the alphabet $\{0, 1\}$. The problem thus becomes a boolean optimziation problem of the form $f : \{0, 1\}^l \rightarrow \mathbb{R}$, where the mappings $h : M \rightarrow \{0, 1\}^l$ and $h' : \{0, 1\}^l \rightarrow M$ are used to encode and decode the parameters and solutions [7].

Integer representation Integer representations have been proposed by some researches [8]. This approach is usefull when dealing with problems where we need to select certain elements in a particular order, e.g. graph-problems, path-finding problems, the knapsack problem, etc. [9].

Real-valued representation Real-valued or floating-point representations were originally used in evolutionary programming and evolution strategies and work well for problems located in continuous search-spaces. The problems take the form $f : M \subseteq \mathbb{R}^n \rightarrow \mathbb{R}$ [7].

Tree representation Tree representations are mainly used in genetic programming to capture the structure of programs. The econding varies but S-expressions are generally used. The tree structure is defined by a function-set and a terminal-set. The function-set defines the types of nodes in the tree, while the terminal-set contains the types of leaves the tree can contain [9].

2.2.2 Evaluation function

The evalutation function is responsible for improvement in the population. It is a function which assigns a fitness or cost value to every genotype and thus enables us to compare the quality of the genotypes in the population. It is also the only information about the problem that is available to the evolutionary algorithm and should therefore include all domain knowledge which is available about the problem [6]. The evaluation is also the process which takes up the most computational resources, 99% of the total computational cost in real-world problems [5].

2.2.3 Population

The population is a set of genotypes which contain the current best solutions to a problem. While genotypes remain stable and unchanging, the population continually changes through the application of selection mechanisms which decide which genotypes are allowed into the next generation of the population. The size of the population almost always remains constant during the lifetime of the algorithm. This in turn creates selection pressure which pushes the population to improve. A population's diversity is the measure of difference amoung the genotypes, phenotypes and fitness values [6].

Steady-state model In the steady-state model the entire population isn't replaces at once, rather only a fraction of the population is replaces a one time.

Generational model In the generational model the entire population is replaces at once.

2.2.4 Parent selection mechanism

Parent selection serves to improve the quality of a population by selecting which individuals will survive into the next generation. The selected individuals are called parents as they usually undergo some form of alteration or combination with other individuals before progressing to the next generation. The selection method is usually probabilistic and gives better solutions a higher probability and worse solutions a lower probability to survive. It's important that bad solutions still recieve a positive probabilty since the population might otherwise lose diversity and coalesce around a false local optimimum [6].

2.2.5 Variation operators

Variation operators introduce new features into the genotypes of a population by modifying or mixing existing genotypes. They can be divided into two types: unary operators which take one genotype and stochastically alter it to introduce random change and n-ary operators which mix the features of 2 or more genotypes. Unary operators are called mutation operators while n-ary operators are called cross-over or recombination operators. The biological equivalents to these are random mutation and sexual reproduction. Mutation operators allow evolutionary algorithms to theoretically span the whole continuum of the search space by giving a non-zero probability that a genotype mutates into any other other genotype. This has been used to formally prove that evolutionary algorithms will always reach the desired optimum given enough time. Recombination tries to create new superior individuals by combining the genes of two good parent genotypes [6, 5].

2.2.6 Survivor selection mechanism

Survivor selection takes place after new offspring have been generated and determines which individuals are allowed to live on into the next generation. It is often refered to as the replacement strategy and contrary to parent selection it is usually deterministic. Two popular mechanisms are fitness-based selection and age-based selection. Fitness-based selection determines the next generation by selecting the individuals with the highest fitness while age-based selection allows only the offspring to survive [6].

2.2.7 Initatilization

Initialization is the process during which the intial population is generated. The genotypes are usually generated randomly from a uniform distribution based on some range of acceptable input values. If a good solution is known before hand variations of it can be include in the initial population as a bias, but this can sometimes cause more problems than it solves [5].

2.2.8 Termination condition

The termination condidion determines for how long the algorithm is run. Four criteria are used to determine when to stop [6]:

1. If a maximum number CPU-cycles or iterations is reached
2. If a known optimum is reached
3. If the fitness value does not improve for a considerable amount of time
4. If the diversity of the polation drops below a given threshold

2.3 Main paradighms of evolutionary computation

Below the main paradighms of evolutionary computation will be discussed. They include genetic algorithms, evolution strategy, evolutionary programming and genetic programming.

2.3.1 Genetic algorithms

Genetic algorithms (GA) were introduced by John Holland in the 1960s as an attempt to apply biological adaptation to computational problems. GAs are multidemensional search algorithms which use populations of binary strings called chromosomes to evolve a solution to a problem. GAs use a selection operator, a mutation operator and a cross-over operator. The selection operator select individuals which are subjected to cross-over based on their fitness and cross-over combines their genetic material to form new individuals which are then randomly mutated [10]. See figure 2 for a simple GA.

```
Initialize a population of N binary chromosome with L bits
**while** termination-condition not fulfilled **do**
    Evaluate the fitness $F(x)$ of each chromosome $x$
    **repeat**
        Select two chromosomes probabilistally from the population
        based on their fitness
        With the cross-over probability $P_c$ create two new offspring
        from the two selected chromosomes using the crossover operator.
        Otherwise create two new offspring identical to their parent chromosomes.
        Mutate the two chromosomes using the mutation probability $P_m$
        and place the resulting chromosomes into the new population
    **until** N offspring have been created
    Replace the old population with the new population
**end while**
```

Figure 2: Basic genetic algorithm

2.3.2 Evolution strategy

Evolution strategies (ES) were first developed to solve parameter optimization tasks. They differ from GAs by representing individuals using a pair of vectors $\vec{v} = (\vec{x}, \vec{\sigma})$. The earliest versions of ES used a population of only one individual and only utilized the mutation operator. New individuals were only introduced into the population if they performed better than their parents. The vector

$\vec{x}$ represents the position in the search space and $\vec{\sigma}$ represents a vector of standard deviations used to generate new individuals. Mutation occurs according to equation 1 where $N(0, \vec{\sigma})$ is a vector containing random Gaussian numbers with the mean 0 and a standard deviation of $\vec{\sigma}$ [2].

$$\vec{x}^{t+1} = \vec{x}^{t} + N(0, \vec{\sigma}) \tag{1}$$

Newer versions of the algorithm include $(\mu + \lambda) - \text{ES}$ and $(\mu, \lambda) - \text{ES}$. The main point of these it that their parameters like mutation variance adapt automatically to the problem. In $(\mu + \lambda) - \text{ES}$ μ parents generate λ offspring and the new generation is selected from μ and λ while in $(\mu, \lambda) - \text{ES}$ μ parents generate λ offspring ($\lambda > \mu$) and the new generation is only selected from λ. These algorithms produce offspring by first applying cross-over to combine two parent chromosomes (including their deviation vectors $\vec{\sigma}$) and then mutating $\vec{x}$ and $\vec{\sigma}$ [2].

2.3.3 Evolutionary programming

Evolutionary programming (EP) was created as an alternative approach to artificial intelligence. The idea was to evolve finite state machines (FSM) which observe the environment and elicit appropriate responses [11]. The environmet is modeled as a sequence of input characters selected from an alphabet and the role of the FSM is to produce the next character in sequence. The fitness of an FSM is measured by a function which tests the FSM on a sequence of input characters, starting with the first character and then progressing to include one more addition character on each iteration. The function measures the correct prediction rate of the FSM and determines it's score [2].

Each FSM creates one offspring which is mutated by one or more of the following operators: change of input symbol, change of state transition, addition of state, deletion of state and change of initial state. The next generation is then selected from the pool of parents and offspring, selecting the best 50% of all available solutions. A general form of EP has recently been devised which can tackle continuous optimization tasks [2].

2.3.4 Genetic programming

Genetic programming (GP) differs from traditional genetic algorithms by evolving computer programs which solve problems instead of directly finding the solution to a problem. The individuals in the population are therefore data-structures which encode computer programs, usually rooted trees representing expressions [2].

At it's most basic the programs are defines as functions which take a set of input parameters and produce an output. The programs are constructed from building blocks such as variables, numbers and operators. The initial population contains a set of such programs which have been initialized as random trees [2].

The evolution process is similar to GAs in that the programs are evaluated using a function which runs a set of test cases and the programs undergo cross-over and other mutations. Cross-over is defined as the exchange of subtrees between programs and produces two offspring from two parents [2].

More advanced versions of GP include function calls which enable the programs to remember useful symmetries and regularities and facilitate code reuse [2].

3 Related Work

Ideas around evolutionary computation began emerging in 1950s. Several researchers, independently from each-other, created algorithms which were inspired by natural Darwinian principles, these include Holland's Genetic Algorithms, Schwefel's and Rechenberg's Evolution Strategies and Fogel's Evolutionary Programming. These pioneering algorithms shared the concepts of populations, individuals, offspring and fitness and ,compared to natural systems, they were quite simplistic, lacking gender, maturation processees, migration, etc [12].

Research has shown that no single algorithm can perform better than all other algorithms on average. This has been referred to as the 'no-free-lunch' and current solutions instead aim at finding better solutions to specific problems by exploiting inherent biases in the problem. This has led to the desire to classify different algorithms in order to decide which algorithms should be used in which situations, a problem which is not trivial [12].

Recent research has focused, among others, on parallelism, multi-population models, multi-objective optimization, dynamic environments and evolving executable code. Parallelism can easily be exploited in EC because of it's inherently parallel nature, e.g each individual in a population can be evaluated, mutated and crossed-over independently. Multi-core CPUs, massively parallel GPUs, clusters and networks can be used to achieve this. Multi-population models mimic the way species depend on each other in nature. Examples of this include host-parasite and predator-prey relationships where the the individual's fitness is connected to the fitness of another individual. Multi-objective optimization aims to solve problems where conflicting interests exist, a good example would be optimizing for power and fuel-consumption simultaneously. In such problems the optimization algorithm has to keep two or more interests in mind simultaneously and find intersections points which offer the best trade-offs between them. Dynamic environments include things like the stock markets and traffic systems. Traditional EAs perform badly in these situations but they can perform well when slightly modified to fit the task. Evolving executable code, as in Genetic Programming and Evolutionary Programming, is a hard problem with very interesting potential applications. Most often low-level code such as assembly, lisp or generic rules are evolved [12].

3.1 Differential Evolution

Differential evolution (DE) was conceived in 1995 by Storn and Price [13] and soon gained wide acceptance as one of the best algorithms in continuous optimization [14]. This spawned many new papers descibing variations and hybrids of the algorithm [15], such as self-adaptive DE (SaDE) [16], opposition-based DE (ODE) [17] and DE with global and local neighborhoods (DEGL) [17].

DE is very easy to implement and has been shown to outperform most other algorithms consistently, it has also been shown that it in general performs better than PSO [18, 19]. DE uses very few patameters and the effects of altering these parameters have been well studied [15]. DE comes in a total of 10 different varieties based on which mutation and cross-over operators are used [20]. Eight of these schemes have been tested and compared, showing that the version called DE/best/1/bin (which utilizes the best current individual in the cross-over process instead of random invidivuals) generally yields the best results [21]. [22] measured the performance of different combinations of parameters, producing general recommendations for DE.

The desire to find optimal parameters have led to the use of self-adjusting DE algorithms. Examples inlude the use of fuzzy systems to control the parameters [23] and the SaDE algorithm [16].

3.2 Particle Swarm Optimization

Particle swarm optimization (PSO) is the most widely used swarm intelligence (SI) algorithm to date. Many modified versions of PSO have been proposed, among others quantum-behaved PSO (QPSO), bare-bones PSO (BBPSO), chaotic PSO, fuzzy PSO, PSOT-VAC and opposition-based PSO. PSO has also been hybridized with other evolutionary algorithm, for instance: genetic algorithms (GA), artificial immune systems (AIS), tabu search (TS), ant colony optimization (ACO), simulated annealing (SA), differential evolution (DE), bio-geography based optimization (BBO) and harmonic search (HS) [3].

Wang et al. [24] have compared the performance of different PSO-parameters and proposed a set of criteria for improving the performance of PSO. Fuzzy logic controllers (FLC) have been used to continuously optimize the PSO-parameters by Kumar and Chaturvedi [25]. Zhang et al. found a simple way to use control theory in order to find good parameters [26]. Yang proposed modified velocity PSO (MVPSO) in which particles learn the best parameters from the other particles [27].

3.3 Estimation of Distribution Algorithms

Estimation of distribution algorithms (EDA) use probabilitic models to solve complex optimization problems. They have been successfull at many engineering problems which at which other algorithms have failed, for instance: millitary antenna design, protein structure prediction, clustering of genes, chemotherapy optimization, portfolio management, etc [28].

Several techniques have been proposed to make EDA more efficient. Parallelization of fitness evaluation and model building has proven effective [29]. Local optimization techniques such as deterministic hill climbing (DHC) has been shown to make EDA faster [30].

3.4 Evolutionary Algorithms in Machine Learning

Evolutionary algorithms (EC) and machine learning (ML) are two growing and promising field in computer science and many attempts have therefore been made to combine the two. ML has been used to improve EC optimization algorithms with so called MLEC-algorithms where various techniques from AI and ML, such as artificial neural networks (ANN), cluster analysis (CA), support vector machines (SVM), etc. help EC algorithms to learn important features of the search space [31].

The opposite use case has also been proposed, using EC to improve ML techniques. An example of this is using DE to optimize feed-forward neural networks (FFNN). Here DE seems to perform similarly to traditional gradient based techniques [32]. Hajare and Bawane showed that using PSO to initialize weights and biases in a neural network before training yields better results than using random weights and help avoid local minima which backpropagation (BP) algorithms often get stuck in [33]. Larrañaga and Lozano [34] tested various EC algorithms (GA, EDA and ES) agains BP and concluded that EC is a competetive alternative to traditional approaches.

3.5 Contributions

EC is a an interesting alternative to traditional approaches in machine learning and continuous optimization and while algorithms such as DE, PSO, etc. have been compared on mathematical benchmarks before [19, 14], the application of EC to machine learning has not been studied with as much details. My primary contribution to the field will be to find what algorithms work best for different ML problems och based on this propose ML-specific improvements.

4 Problem Formulation

The purpose of this thesis is to explore the performance and usefulness of three emerging evolutionary algorithms: differential evolution (DE), particle swarm optimization (PSO) and estimation of distribution algorithms (EDA). The intention is to compare these against each other on a set of benchmark functions and practical problems in machine learning and then, if possible, develop a new or modified algorithm which improves upon then aforementioned ones in some aspect.

4.1 Research Questions

The questions asked in the thesis are:

- How do DE, PSO and EDA perform comparatively when applied to mathematical optimization problems?
- How do DE, PSO and EDA perform comparatively when applied to machine learning problems such as neural network optimization and artificial intelligence in games
- How are DE, PSO and EDA suited to these different problems?
- Can an improved algorithm which draws inspiration from the design of DE, PSO and/or EDA outperform any of them in some/all of the aformentioned benchmarks and problem sets?

4.2 Motivation

This research is interesting because it yield insight into the applicability of emerging evolutionary algorithms to currently popular machine learning methods such as artifical neural networks and also compares them more generally on generic mathematical optimization problems. The possibility of an improved novel algorithm which is better att handling machine learning problems also makes the work more interesting.

4.3 Outcomes

The goals in this works are:

- Benchmark DE, PSO, EDA on mathematical optimization problems
- Benchmark DE, PSO, EDA on machine learning problems
- Develop a new algorithm inspired by DE, PSO and/or EDA
- Benchmark the new algorithm
- Compare the new algorithm with DE, PSO and EDA and draw conclusions from the results

4.4 Limitations

The scope of this work limits the number of algorithms which can be included in the testing. The individual algorithms also have numerous variations and parameters which can dramatically affect their behavior and it will not be possible to take all these considerations into account. Furthermore, the benchmarking will be restricted to a standard set of testing functions which may or may not provide reliable information regarding the general usability the algorithms. Since evolutionary algorithms have a large number of potential and actual use cases the practical testing will only concern a small subset of the these and may therefore not provide accurate data for all possible use cases.

5 Method

A benchmark was constructed to compare the algorithms. Two distict problem sets were used: 1. Mathematical functions and 2. Machine learning problems. The algorithms, the problem sets and the benchmark program were developed in Matlab. Matlab was chosen because it works well for scientific computation and provides convenient access to features which are needed to solve the problems at hand (manipulating matrices, measuring performance, parallel computing, dynamic data-structures, probability distributions, etc.).

To ensure that the results are accurate and usefull the mathematical function benchmark was run in the following way:

- Benchmark at dimension 10, 30 and 50
- Benchmark at population size $100 + dimension * 5$
- Set maximum number of generations to $10000 * dimension/populationsize$ to ensure the number of evaluations remains constant throughout the benchmark
- Run each algorithm 30 times on each problem and record averages and standard deviations

The machine learning benchmark is less uniform and the parameters used are described in "Results"

6 Ethical and Societal Considerations

This work does not have any ethical or societal considerations.

7 Algorithms

This section describes the three algorithms which were benchmarked together with my own algorithm contribution. The Matlab code can be viewed in appendix A.

7.1 Differential Evolution

Differential evolution [35] is a stochastic optimization algorithm which works on populations of parameter vectors. The problem to minimize will be denoted by $f(x)$ where $X = [x_1, x_2, x_3, ...x_D]$ and D is equal to the number of variables taken as input parameters by $f(x)$. The algorithm consists of multiple steps which will be described in detail below.

The first step in DE is to create an initial population, the size of the population is N and it will be represented by a matrix x where g is the generation and $n = 1, 2, 3, ..., N$:

$$x^g_{n,i} = [x^g_{n,1}, x^g_{n,2}, x^g_{n,3}, ..., x^g_{n,D}] \tag{2}$$

The population is randomly generated to uniformly fill the entire parameter space ($x^U_{n,i}$ is the upper bound for parameter x_i and $x^L_{n,i}$ is the lower bound for parameter x_i):

Mutation is the first step when creating a new generation from the population. Mutation is performed individually for every vector x in the population. The mutation procedure is as follows: select three random vectors for each parameter vector (this requires that the population has a size of $N > 3$) and create a set of new vectors v called mutant vectors according to the formula below where $n = 1, 2, 3, ..., N$:

$$v^{g+1}_n = [x^g_{r1n} + F(x^g_{r2n} - x^g_{r3n}) \tag{3}$$

The value of F can be chosen from the interval $[0, 2]$ and determines the influence of the differential weight $(x^g_{r2n} - x^g_{r3n})$.

Crossover occurs after mutation and is applied individually to every vector x. A new vector u called the trial vector is constructed from the mutant vector v and the original vector x. The trial vector is produced according to the formula below with $i = 1, 2, 3, ..., D$ and $n = 1, 2, 3, ..., N$:

$$u^{g+1}_{n,i} = \begin{cases} v^{g+1}_{n,i}, & \text{if } rand() \le CR \wedge i = I_{\text{rand}} \\ x^g_{n,i}, & \text{otherwise} \end{cases} \tag{4}$$

I_{rand} is a randomly selected index from the interval $[1, D]$ and CR is the crossover constant which determines the probability that an element is selected from the mutant vector.

Selection is the last step in creating a new generation. The trial vector u is compared with the original vector x for fitness and the vector with the lost cost is selected for the generation according to the formula below where $n = 1, 2, 3, ..., N$:

$$x^{g+1}_n = \begin{cases} u^{g+1}_n, & \text{if} f(u^{g+1}_n) < f(x^g_n) \\ x^g_{n,i}, & \text{otherwise} \end{cases} \tag{5}$$

After selection is performed for every vector in the population the population is evaluated to determine if an acceptable solution has been generated. If a solution has been found the algorithm terminates, otherwise the mutation, crossover and selection is performed again until a solution is found or a maximum number of iterations has been reached.

In the benchmark the parameters for DE have been set to $F = 0.6$ and $CR = 0.9$ as recommended by Gamperle et al. [22].

Variants Different DE schemes are classified as DE/x/y, where x symbolizes the vector which is mutated and y is the number of differential vectors used. The value of x can be "rand" for random vector or "best"' for the best vector in the population. The algorithm abose is therefore classified as DE/rand/1. The variant DE/best/2 is considered to be a good alternative to DE/rand/1 [16]. It's mutation equation is described below:

$$v^{g+1}_n = [x^g_{best} + F(x^g_{r1n} - x^g_{r2n}) + F(x^g_{r3n} - x^g_{r4n}) \tag{6}$$

7.2 Particle Swarm Optimization

Particle Swarm Optimization (PSO) [36] was introduced in 1995 by Kenneth and Ebenhart [37]. The optimization problem is represented by an n-dimensional function

$$f(x_1, x_2, x_3, ...x_n) = f(\vec{X}) \tag{7}$$

where $\vec{X}$ is a vector which represents the real parameters given to the function. The intent is to find a point in the n-dimensional parameter hyperspace that minimizes the function.

PSO is a parallel search technique where a set of particles fly through the n-dimensional search space and probe solutions along the way. Each particle P has a current position $\vec{x}(t)$, a current velocity $\vec{v}(t)$, a personal best position $\vec{p}(t)$ and the neighborhoods best position $\vec{g}(t)$. A neighborhood N is a collection of particles which act as independent swarms. Neigborhoods can be social or geographical. Social neigborhoods do not change and contain the same particles during the entire optimization process, while geographical neighborhoods are dynamic and consist only of particles which are near to one another. The neighborhood is often set to be identical to the whole swarm of particles, denoted S.

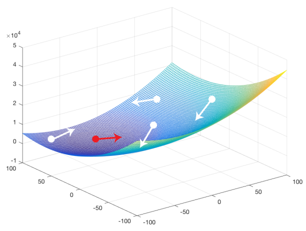

Figure 3: Illustration of particles in PSO (red represents the best particle)

The algorithm has a set of general properties: v_{max} restricts the velocity of each particle to the interval $[-v_{max}, v_{max}]$, an inertial factor ω, two random numbers ϕ_1 and ϕ_2 which affect the velocity update formula by modulating the influence of $\vec{p}(t)$ and $\vec{g}(t)$, and two constants C^2 and C^1 which are termed particle "self-confidence" and "swarm confidence".

The initial values of $\vec{p}(t)$ and $\vec{g}(t)$ are equal to $\vec{x}(0)$ for each particle. After the particle have been initialized an iterative update process is started which modifies the positions and velocities of the particles. The formula below describes the process (d is the dimension of the position and velocity and i is the index of the particle):

$$v_{id}(t+1) = \omega v_{id}(t) + C_1\phi_1(p_{id}(t) - x_{id}(t)) + C_2\phi_2(g_{id}(t) - x_{id}(t)) \tag{8}$$

$$x_{id}(t+1) = x_{id}(t) + v_{id}(t+1) \tag{9}$$

The "self-confidence" constant affects how much self-exploration a particle is allowed to do while "swarm-confidence" affects how much a particle follows the swarm. ϕ_1 and ϕ_2 are random numbers which push the particle in a new direction while ω keeps a particle on the path it's currently on. The PSO algorithm is described in figure 4. See figure 3 for an illustration of PSO.

In the benchmark the parameters for PSO have been set to $\omega = 0.8$ [38], $c_1 = c_2 = 1.494$ [39] and v_{max} = parameter range size [36].

7.3 Estimation of Distribution Algorithm

Estimation of distribution algorithms are stochastic search algorithms which try to find the optimal value of a function by creating and sampling probability distributions repeatedly. The first step is creating population $P(0)$ and filling it with solution parameter vectors created from a probability distribution which covers the whole search space uniformly. Then all solutions in $P(g)$ are evaluated and the best solutions $S(g)$ are selected (a threshold variable t is used to determine how many solutions are selected, setting $t = 50\%$ means that the best 50% of the solutions are selected).

```
Init particals with random postitions $\vec{x}(0)$ and velocities $\vec{v}(0)$
repeat
    for all Particles $i$ do
        Evaluate fitness $f(\vec{x_i})$
        Update $\vec{p}(t)$ and $\vec{g}(t)$
        Adapt the velocity of the particle using the above-mentioned equation
        Update the position of the particle
    end for
until $\vec{g}(t)$ is a suitable solution
```

Figure 4: PSO algorithm

After selection a probabilistic model $M(g)$ is constructed from $S(g)$ and new solutions $O(g)$ are sampled from $M(g)$. Finally $O(g)$ is incorporated into $P(g)$ The generation counter is incremented $g = g + 1$ and the selection, model and sampling stages are repeated until a suitable solution is found [28].

The most difficult part is constructing the probabilistic model, this will differ for continuous and discreet optimization and a model of appropriate complexity has to be chosen depending on the nature of the problem. The simplest method for continuous EDAs is using a continuous Univariate Marginal Density Algorithm (UMDA). However depending on the complexity of the problem other methods such as Estimation of Baysian Networks (EBNA) can be used [40].

UMDA The UMDA algorithm is an EDA algorithm which uses a set of independent probability distributions to sample new solution vectors. The probability model can be expressed as a product of the individual probabilities

$$p(x) = \prod_{d=1}^{D} p_d(x_d) \tag{10}$$

where $p(x)$ is the global multivariate density, D is the vector length and $p_d(x_d)$ are the individual univariate marginal densities [41]. The algorithm is described in figure 5.

```
Initialize population P
repeat
    Evaluate P
    Select the best t% of P into S
    Let m be the mean S
    Let s be the standard deviation of S
    Sample S' from normal distribution using m and s
    Create new generation of P from S' and S
until Termination condition
```

Figure 5: UMDA algorithm

7.4 Improved Algorithm: Differential EDA

My improved algorithm, differential EDA (DEDA), draws upon DE and EDA, applying differential mutation to the selection population of the EDA algorithm. The algorithm is described in figure 6.

```
Initialize random population $P$
**repeat**
    Sort population $P$
    Select $S$ from $P$
    Build probabilistic model $M$ from $S$
    Sample $P'$ from $M$
    Let $S_{DE}$ be DE applied to $S$
    Create new generation $P$ from $P'$ and $S_{DE}$
**until** Termination condition
```

Figure 6: Improved algorithm

8 Mathematical Problem Set

The functions for the benchmark are taken from the 2005 CEC conference on continuous evolutionary optimization [42]. The functions are losely based on the popular optimization benchmark suite created by DeJong [43].

For all functions $x = [x_1, x_2, x_3, ..., x_D]$ are the input parameters, $o = [o_1, o_2, o_3, ..., o_D]$ is the global optimum, D is the dimension and M is an orthogonal matrix with parameters unique to each function. The matrices for o and M can be obtained from [42]. Illustrations of the functions can be found in figures 8, 9, 10, 11, 12, 13, 14, 15, 16 and 17.

The function properties are described in figure 7.

Function	Modal	Separable
F_1	Uni	Yes
F_2	Uni	No
F_3	Uni	No
F_4	Uni	No
F_5	Uni	No
F_6	Multi	No
F_7	Multi	No
F_8	Multi	No
F_9	Multi	Yes
F_{10}	Multi	No

Figure 7: Function properties

8.1 F_1: Shifted Sphere Function

$$F_1(x) = \sum_{i=1}^{D} z_i^2 \tag{11}$$
$$z = x - o$$
$$x \in [-100, 100]^D$$

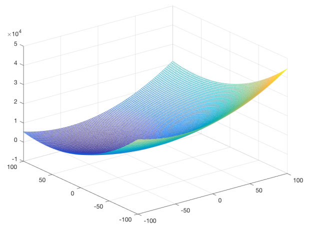

Figure 8: 3-D map for 2-D function

8.2 F_2: Shifted Schwefel's Problem

$$F_2(x) = \sum_{i=1}^{D} (\sum_{j=1}^{i} z_j)^2 \tag{12}$$
$$z = x - o$$
$$x \in [-100, 100]^D$$

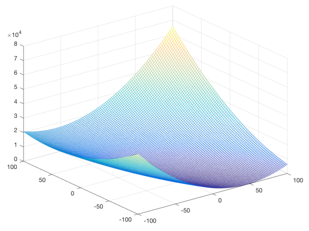

Figure 9: 3-D map for 2-D function

8.3 F_3: Shifted Rotated High Conditioned Elliptic Function

$$F_3(x) = \sum_{i=1}^{D} (10^6)^{\frac{i-1}{D-1}} z_i^2 \tag{13}$$
$$z = (x - o) * M$$
$$x \in [-100, 100]^D$$

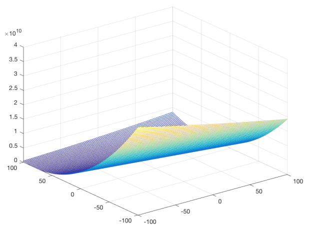

Figure 10: 3-D map for 2-D function

8.4 F_4: Shifted Schwefel's Problem with Noise in Fitness

$$F_4(x) = (\sum_{i=1}^{D} (\sum_{j=1}^{i} z_j)^2) * (1 + 0.4|N(0,1)|) \tag{14}$$
$$z = x - o$$
$$x \in [-100, 100]^D$$

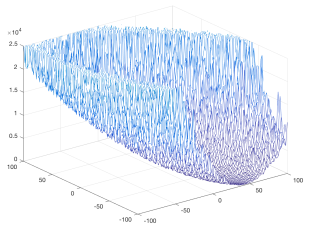

Figure 11: 3-D map for 2-D function

8.5 F_5: Schwefel's Problem with Global Optimum on Bounds

$$F_5(x) = max\{|A_i x - B_i|\} \tag{15}$$
$$i = 1, ..., D, x \in [-100, 100]^D$$
$$A \text{ is a } D * D \text{ matrix}, a_{ij} = \text{ random numbers in } [-500, 500], det(A) \neq 0$$
$$B_i = A_i * o, o_i = \text{ random numbers in } [-100, 100]$$
$$o_i = -100, \text{ for } i = 1, 2, ..., [D/4], o_i = 100, \text{ for } i = [3D/4], ..., D$$

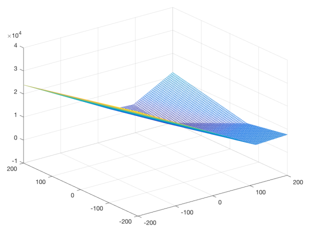

Figure 12: 3-D map for 2-D function

8.6 F_6: Shifted Rosenbrock's Function

$$F_6(x) = \sum_{i=1}^{D-1} (100(z_i^2 - z_{i+1})^2 + (z_i - 1)^2) \tag{16}$$
$$z = x - o + 1$$
$$x \in [-100, 100]^D$$

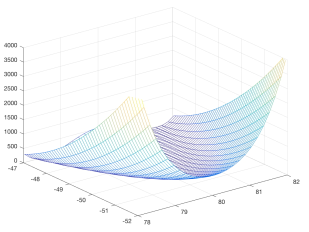

Figure 13: 3-D map for 2-D function

8.7 F_7: Shifted Rotated Griewank's Function without Bounds

$$F_7(x) = \sum_{i=1}^{D} \frac{z_i^2}{4000} - \prod_{i=1}^{D} \cos\frac{z_i}{\sqrt{i}} + 1 \tag{17}$$
$$z = (x - o) * M$$
$$x \in [0, 600]^D$$

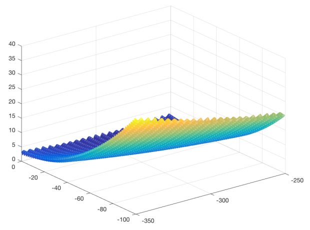

Figure 14: 3-D map for 2-D function

8.8 F_8: Shifted Rotated Ackley's Function with Global Optimum on Bounds

$$F_8(x) = -20 \exp(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} z_i^2}) - \exp(\frac{1}{D}\sum_{i=1}^{D}\cos(2\pi z_i)) + 20 + e \tag{18}$$

$$z = (x - o) * M$$

$$x \in [-32, 32]^D$$

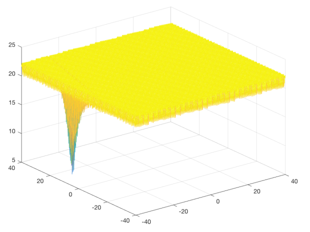

Figure 15: 3-D map for 2-D function

8.9 F_9: Shifted Rastrigin's Function

$$F_9(x) = \sum_{i=1}^{D} z_i^2 - 10\cos(2\pi z_i) + 10 \tag{19}$$

$$z = x - o$$

$$x \in [-5, 5]^D$$

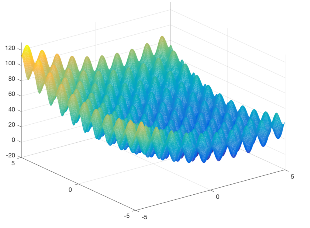

Figure 16: 3-D map for 2-D function

8.10 F_{10}: Shifted Rotated Rastrigin's Function

$$F_{10}(x) = \sum_{i=1}^{D} z_i^2 - 10\cos(2\pi z_i) + 10 \tag{20}$$
$$z = (x - o) * M$$
$$x \in [-5, 5]^D$$

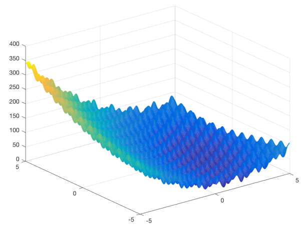

Figure 17: 3-D map for 2-D function

9 Machine Learning Problem Set

To evaluate the optimization algorithms further I set have set out to apply them on a variety of machine learning problems. These will predominantly be applications of feed-forward neural networks (FFNN). The design of the neural network is one input layer with a size based on the problem dimension, two hidden layers and one output layer. I have chosen to use two hidden layers to make sure a good solution can be found quickly [44]. The size of the hidden layers was set to 2/3 and 1/3 of the mean of the size of the input and output layers.

9.1 ML_1: Training FFNNs to do function approximation

The optimization algorithm were used to train FFNNs to perform function approximation and curve-fitting on seven sample data-sets which are available in Matlab's Neural Network Toolbox. The data-sets used are the following

1. simplefit_dataset (Simple fitting dataset)
2. abalone_dataset (Abalone shell rings dataset)
3. bodyfat_dataset (Body fat percentage dataset)
4. building_dataset (Building energy dataset)
5. chemical_dataset (Chemical sensor dataset)
6. cho_dataset (Cholesterol dataset)
7. engine_dataset (Engine behavior dataset)
8. house_dataset (House value dataset)

The data sets contain two matrices. One with sample input vectors to the neural network and one with the expected correct output vectors. The function which the optimization algorithm directly optimize is the sum of squared errors as defined by

$$sse(x,t) = \frac{1}{2n}\sum_{i=1}^{n}(y(x) - t)^2 \tag{21}$$

where x is the input vector to neural network y, t is the correct expected output which $y(x)$ should produce and n is the length of vector x.

9.2 ML_2: Training FFNNs to do classification

The optimization algorithm were used to train FFNNs to perform classification tasks on eight sample data-sets which are available in Matlab's Neural Network Toolbox. The data-sets used are the following

1. simpleclass_dataset(Simple pattern recognition dataset)
2. cancer_dataset (Breast cancer dataset)
3. crab_dataset (Crab gender dataset)
4. glass_dataset (Glass chemical dataset)
5. iris_dataset (Iris flower dataset)
6. thyroid_dataset (Thyroid function dataset)
7. wine_dataset (Italian wines dataset)

The evaluation procedure is the same as for ML_1.

9.3 ML_3: Training FFNNs to do clustering

The optimization algorithm were used to train FFNNs to perform clustering tasks on one sample data-sets which is available in Matlab's Neural Network Toolbox. The data-sets used is the following

1. simplecluster_dataset (Simple clustering dataset)

The evaluation procedure is the same as for ML_1 and ML_2.

9.4 ML_4: Neuroevolution of Game Controller for "Snake"

Machine learning test set ML_1 uses evolutionary algorithms to evolve weights for feed-forward neural networks, which control the behavior of game-agent in the classic game of "Snake".

9.4.1 Game Description and Rules

The game is made up of a $n * m$ square grid through which a snake can freely move. The snake consists of a sequence of interconneted blocks and is divided into a head which moves in a certain direction and a tail which follows the head. Food blocks randomly appear on block at a time on the grid and the snake's objective is eat these blocks. When the snake's head touches the food block the block vanishes and the snake grows one square in length. If the head of the snake tries to move outside the grid it dies and the game ends. If the head of the snake collides with the tail it also and dies and the game ends. The snake's head can move in four direction: up, down, left and right. If the snake tries to move in the opposite of it's currect direction it also dies. The snake has a starvation counter which forces it to pursue food and eat. The starvation counter is initialized to the starvation threshold when the game begins and decreases by one every time the snake moves. When the snake eats, the starvation counter is increased once by the starvation threshold. Figure 18 illustrates the game grid. Figure 19 described the algorithm for the game.

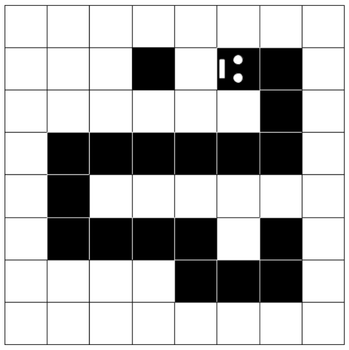
Figure 18: Snake $8 * 8$ game-field with snake hunting food

The game has the following parameters and settings:

1. Dimensions - $n * m$ blocks in the grid
2. Starting point (x, y) for snake head
3. Starting direction for snake head $\{left, right, up, down\}$
4. Starvation threshold t

9.4.2 Representation of Game State

In order to control the snake with a neural network a representation of the state of the game has to be generated before each move. The chosen representation is a 12-dimensinal vector in the range $[0, 1]$. The initial default value of all dimensions is zero. Dimensions 1-4 indicates with a value of one whether any dangerous object (the tail or the edge of the grid) is immediatly to the left, to the right, above or below the snake's head. Dimensions 5-8 indicate the distribution of the tail relative to the head of the snake. Floating-point numbers indicate how much of the tail is above, below, to the right and to the left of the head. Dimensions 9-12 indicate with a value of one when food is to the left, to the right, above or below the snake's head.

```
$initializeSnake()$
$foodEaten \leftarrow 0$
$movesMade \leftarrow 0$
while $alive$ do
    $state \leftarrow getGameState()$
    $move \leftarrow getNextMove(state)$
    $moveSnake(move)$
    if $collides(head, tail)$ then $die()$
    end if
    if $outOfBounds(head)$ then $die$
    end if
    if $collides(head, food)$ then
        $eatFood(food)$
        $growSnake()$
        $foodEaten \leftarrow foodEaten + 1$
    end if
    $movesMade \leftarrow movesMade + 1$
end while
return $foodEaten + sigmoid(movesMade)$
```

Figure 19: Snake game

9.4.3 Neural Network Controller

The neural network which controls is a feed-forward neural network with one hidden layer. The input layer has 12 neurons which correspond to the game-state representation. The output layer has four neurons, which correspond to the decision to move left, right, up or down. The hidden layer has 8 neurons. Figure 20 illustrates the network.

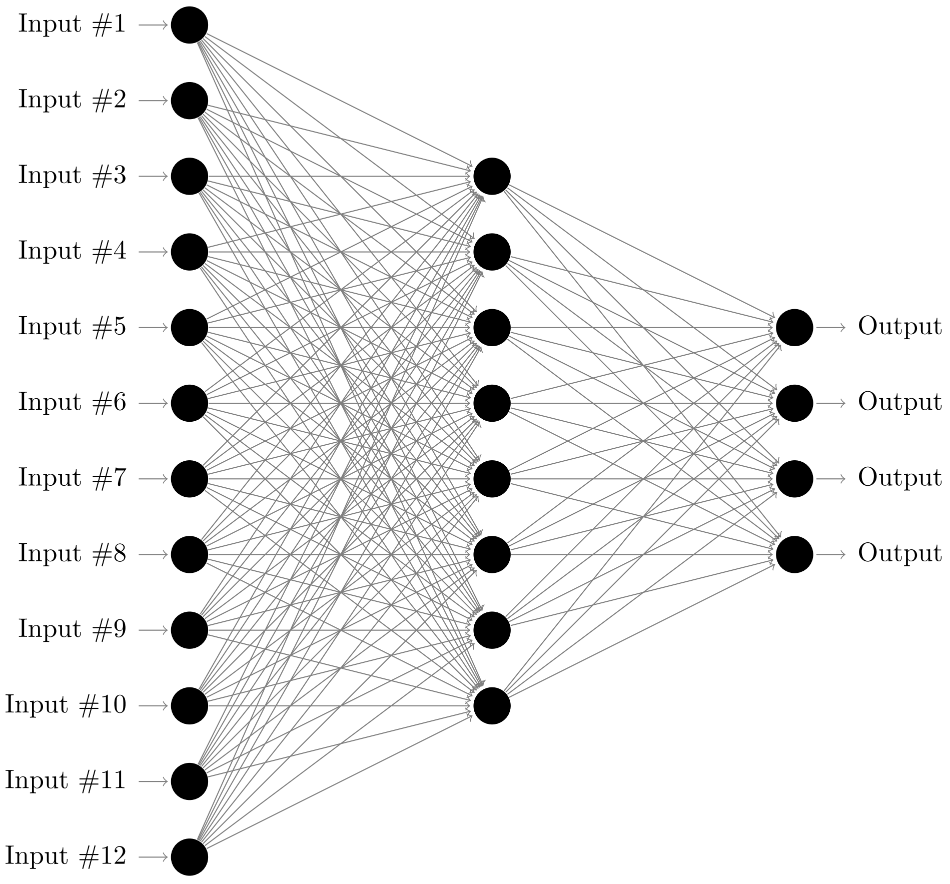

Figure 20: FFNN snake controller

9.4.4 Fitness Function and Evaluation

The objective function, which is run by the optimization algorithms, takes the evolved weights of the neural network and attempts to play the game using them. The fitness function of gameplay if determined by the equation below:

$$fitness = \text{foodEaten} + \frac{1}{1 - e^{-\text{movesMade}}} \tag{22}$$

The fitness function ensured that the snake has an “easy start” when it will be revarded for not dying, but then has to pursue food in order to maximize it’s fitness.

10 Results

This section presents benchmarks and measurements of the four main algorithms presented in this thesis. It is divided into two parts: the function optimization benchmark and the machine learning benchmark.

10.1 Mathematical Problem Set

The function test-bed was tested with dimension size $D = 10$, $D = 30$ and $D = 50$. The parameters for different dimension sizes are presented in figure 21.

Dimension	Max. Generations	Population size
10	667	150
30	1200	250
50	1429	350

Figure 21: Parameters for different dimension sizes in the benchmark

The benchmark ran each test-case 30 times and recorded the average minimum value of the function and the standard deviation. The results are presented in figure 22, 23 and 24. The lowest value is marked in bold font.

F	DE		PSO		EDA		DEDA	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
1	3,88E-17	3,42E-17	1,65E-16	3,64E-16	**1,67E-27**	4,07E-28	0,00E+00	0,00E+00
2	1,83E-08	1,24E-08	2,07E-06	3,36E-06	2,51E+02	1,42E+02	**4,23E-27**	3,10E-27
3	5,19E-01	2,29E-01	4,88E+05	7,21E+05	1,09E+05	8,32E+04	**4,96E-02**	3,01E-02
4	3,36E-07	2,59E-07	6,31E-05	9,10E-05	3,21E+02	2,16E+02	**1,73E-26**	2,16E-26
5	**4,24E-13**	7,82E-13	4,94E+02	4,49E+02	1,67E+02	1,37E+02	6,57E+00	1,26E+01
6	**5,43E-05**	1,24E-04	2,11E+02	7,15E+02	1,34E+04	4,30E+04	6,71E+00	5,83E-01
7	4,87E-01	7,61E-02	2,74E-01	1,79E-01	**1,87E-01**	2,23E-01	3,41E-01	1,24E-01
8	2,04E+01	7,20E-02	**2,03E+01**	8,82E-02	2,04E+01	8,15E-02	2,04E+01	6,88E-02
9	2,50E+01	4,44E+00	**1,76E+00**	1,49E+00	2,35E+01	3,08E+00	1,80E+01	3,52E+00
10	3,26E+01	4,25E+00	**1,73E+01**	7,50E+00	2,11E+01	3,43E+00	2,05E+01	3,62E+00

Figure 22: Benchmark results for $D = 10$

F	DE		PSO		EDA		DEDA	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
1	1,03E+00	1,53E-01	4,54E-10	7,66E-10	1,81E-25	7,66E-10	**8,58E-28**	1,40E-28
2	3,07E+03	6,91E+02	1,42E+03	7,01E+02	1,71E+03	7,01E+02	**4,84E+02**	1,80E+02
3	3,28E+07	5,84E+06	1,92E+07	1,12E+07	4,93E+06	1,12E+07	1,88E+07	4,86E+06
4	6,06E+03	1,23E+03	1,50E+04	6,87E+03	**3,63E+03**	6,87E+03	**6,74E+02**	1,96E+02
5	1,38E+03	3,54E+02	6,92E+03	2,55E+03	2,70E+03	2,55E+03	**1,31E+02**	9,21E+01
6	2,99E+03	1,24E+03	**5,30E+02**	8,60E+02	8,57E+04	8,60E+02	7,32E+03	2,21E+04
7	1,22E+00	5,36E-02	**5,77E-02**	5,18E-02	5,86E+00	5,18E-02	2,54E-01	4,25E-01
8	2,09E+01	4,24E-02	**2,09E+01**	5,74E-02	2,10E+01	5,74E-02	2,10E+01	5,41E-02
9	1,91E+02	9,58E+00	**2,53E+01**	5,97E+00	1,53E+02	5,97E+00	1,60E+02	8,27E+00
10	2,19E+02	1,09E+01	**1,14E+02**	3,78E+01	1,59E+02	3,78E+01	1,60E+02	8,53E+00

Figure 23: Benchmark results for $D = 30$

F	DE		PSO		EDA		DEDA	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
1	5,64E+02	8,54E+01	1,20E-04	1,49E-04	1,04E-24	9,53E-26	**2,14E-26**	4,42E-27
2	5,00E+04	4,88E+03	3,69E+04	1,02E+04	**3,55E+03**	6,45E+02	3,96E+04	5,00E+03
3	2,52E+08	3,33E+07	8,68E+07	4,64E+07	**1,31E+07**	2,94E+06	1,96E+08	3,16E+07
4	7,02E+04	9,89E+03	1,12E+05	3,09E+04	**8,17E+03**	1,71E+03	5,08E+04	7,70E+03
5	1,33E+04	1,03E+03	1,48E+04	3,64E+03	4,31E+03	2,95E+02	**2,71E+03**	3,84E+02
6	7,19E+06	2,18E+06	1,62E+04	3,11E+04	**1,18E+04**	2,68E+04	2,23E+04	1,09E+05
7	3,32E+01	6,43E+00	1,05E+00	1,24E-01	1,25E+01	3,91E+00	**6,04E-01**	6,19E-01
8	2,12E+01	2,28E-02	**2,11E+01**	5,09E-02	2,11E+01	4,97E-02	2,11E+01	3,28E-02
9	3,94E+02	1,24E+01	**7,93E+01**	1,46E+01	3,14E+02	1,41E+01	3,27E+02	1,13E+01
10	4,48E+02	1,87E+01	**2,79E+02**	6,36E+01	3,20E+02	1,17E+01	3,27E+02	9,26E+00

Figure 24: Benchmark results for $D = 50$

10.2 Machine Learning Problem Set

10.2.1 ML_4: Neuroevolution of Game Controller for "Snake"

The ML_4 results are shown in figure 26. The parameters are shown in figure 25.

Parameter	Value
Individuals	140
Generations	1000
Grid dimension	10*10
Weight dimension	140
Repeat measurements	10
Repeat fitness measurements	5

Figure 25: Benchmark parameters for ML_4

F	DE		PSO		EDA		DEDA	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
snake	2,32E+01	2,19E+00	2,92E+01	4,16E+00	2,78E+01	1,34E+01	2,69E+01	3,59E+00

Figure 26: Benchmark results for ML_4

10.2.2 ML_{1-3}: Sample Data Sets

The algorithms were tested on four machine learning benchmarks: ML_1, ML_2, ML_3 and ML_4 as described in the section "Method". The dimensionality of the problem depends on the size of the input and output vectors of the data-sets and the size of the hidden layer of the neural networks. The hidden layer size was defines as the mean of the input- and output-layer sizes. The population size was set to 50, the max. number of generations was set to 100 and every measurement was made 5 times. The figures 27, 28, 29 and 26 show the results obtained for problems (average minimum value and standard deviation).

F	DE		PSO		EDA		DEDA	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
simplefit_dataset	0	0	0	0	0	0	0	0
abalone_dataset	0	0	0	0	0	0	0	0
bodyfat_dataset	0	0	0	0	0	0	0	0
building_dataset	0	0	0	0	0	0	0	0
chemical_dataset	0	0	0	0	0	0	0	0
cho_dataset	0	0	0	0	0	0	0	0
engine_dataset	0	0	0	0	0	0	0	0
house_dataset	0	0	0	0	0	0	0	0

Figure 27: Benchmark results for ML_1

F	DE		PSO		EDA		DEDA	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
simpleclass_dataset	0	0	0	0	0	0	0	0
cancer_dataset	0	0	0	0	0	0	0	0
crab_dataset	0	0	0	0	0	0	0	0
glass_dataset	0	0	0	0	0	0	0	0
iris_dataset	0	0	0	0	0	0	0	0
thyroid_dataset	0	0	0	0	0	0	0	0
wine_dataset	0	0	0	0	0	0	0	0

Figure 28: Benchmark results for ML_2

F	DE		PSO		EDA		DEDA	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
simplecluster_dataset	0	0	0	0	0	0	0	0

Figure 29: Benchmark results for ML_3

11 Discussion

Här presenterar du tolkning av resultaten och bedömer deras signifikans. Diskutera möjliga konsekvenser av resultaten, och presentera eventuella rekommendationer. Det är viktigt att du redogör för om du uppnått de mål du satte upp och därmed besvarat din frågeställning och uppnått syftet med arbetet. Avsnittet ska också innehålla reflektioner kring arbetet, som till exempel dess begränsningar. Du kan också diskutera lösningar på problem som du identifierat och diskuterat tidigare, eller ta upp andra problem som arbetet inte behandlat, frågor som ej besvarats. Koppla också dina resultat till tidigare arbeten. På så sätt kan diskussionen bli ett samtal med det du skrev i tidigare avsnitt. Slutligen ska du sätta ditt eget arbete i ett större sammanhang, bredda ditt perspektiv. Kan dina resultatet generaliseras? Kan det du gjort användas i något annat sammanhang?

12 Conclusions

I detta avsnitt ska du summera rapporten samt presentera slutsatser och slutanalys. Ge en kort översikt av syftet och frågeställningen. Du ska sedan tydligt tala om de viktigaste resultaten, förklara deras signifikans och sätta in dem i sitt sammanhang. Alla slutsatser ska ha stöd i tidigare delar av rapporten. Du ska däremot inte presentera nya detaljer.

En expert ska kunna läsa detta avsnitt oberoende av resten av rapporten.

13 Future Work

Här ska du diskutera vad som återstår att göra. Du kan ta upp förbättringar som kan göras, och utvidgningar av ditt arbete som kan resultera i nya frågeställningar för framtiden.

A Algorithms

This appendix will present the matlab code for the four algorithms descibed under “Algorithms”.

A.1 Differential Evolution

```
function [success, iterations, minimum, value] = DE(CostFunction, dimension,
    lowerBound, upperBound, maxIterations, populationSize, objectiveValue)

    %% Parameters
    F = 0.6;
    pCR=0.9;

    %% Setup
    success = false;
    dimensionSize = [1 dimension];
    emptyIndividualPosition = [];
    bestIndividualCost = inf;
    populationPosition = repmat(emptyIndividualPosition, populationSize, 1);
    populationCost = zeros(1,populationSize);
    bestCost = zeros(maxIterations,1);

    %% Pre-Evaluation & Initialization
    for i=1:populationSize
        populationPosition(i,:) = unifrnd(lowerBound, upperBound, dimensionSize);
        populationCost(i) = CostFunction(populationPosition(i,:));
        if populationCost(i) < bestIndividualCost
            bestIndividualCost = populationCost(i);
            bestIndividualPosition = populationPosition(i,:);
        end
    end

    lastBestValue = bestIndividualCost;

    %% Iteration
    for iterations=1:maxIterations

        %% Success
        if bestIndividualCost <= objectiveValue
            success = true;
            break;
        end

        %% Mutation
        for i=1:populationSize
            x = populationPosition(i,:);
            A = randperm(populationSize);
            A(A==i)=[];
            a=A(1);
            b=A(2);
            c=A(3);
            y = populationPosition(a,:) + F*(populationPosition(b,:) -
                populationPosition(c,:));
            y = max(y, lowerBound);
            y = min(y, upperBound);
            z = zeros(size(x));
            j0 = randi([1 numel(x)]);
            for j=1:numel(x)
                if j==j0 || rand <= pCR
                    z(j) = y(j);
                else
                    z(j) = x(j);
                end
            end
            newSolutionPosition = z;
            newSolutionCost = CostFunction(newSolutionPosition);
            if newSolutionCost < populationCost(i)
                populationPosition(i,:) = newSolutionPosition;
                populationCost(i) = newSolutionCost;
                if populationCost(i) < bestIndividualCost
                   bestIndividualCost = populationCost(i);
                   bestIndividualPosition = populationPosition(i,:);
                end
            end
        end



        %% Return values
        bestCost(iterations) = bestIndividualCost;
        minimum = bestIndividualPosition;
        value = bestIndividualCost;

%          if lastBestValue ~= bestIndividualCost
%              disp(bestIndividualCost);
%              lastBestValue = bestIndividualCost;
%
%          end

    end
end
```

A.2 Particle Swarm Optimization

```
function [ success, iterations, minimum, value ] = PSO( CostFunction, dimension,
    lowerBound, upperBound, maxIterations, populationSize, objectiveValue )


    %% Parameters
    vmax = (abs(upperBound)+abs(lowerBound));
    omega = 0.8;
    c1 = 1.494;
    c2 = c1;

    %% Initialization
    position = lowerBound + (upperBound - lowerBound) * rand(populationSize,
        dimension);
    velocity = -vmax + (vmax + vmax) * rand(populationSize, dimension);
    best = position;
    bestValue = zeros(1,populationSize);
    globalBest = best(1,:);
    globalBestValue = CostFunction(globalBest);
    lastBestValue = globalBestValue;

    %% Pre-Evaluation
    for i=1:populationSize
        bestValue(i) = CostFunction(best(i,:));
    end

    %% Iteration
    iterations = 0;
    while (globalBestValue > objectiveValue) && (iterations < maxIterations)

        %% Evaluation
        for i = 1:populationSize
            cost = CostFunction(position(i,:));
            if cost <= bestValue(i)
                best(i,:) = position(i,:);
                bestValue(i) = cost;
                if cost <= globalBestValue
                    globalBest = best(i,:);
                    globalBestValue = bestValue(i);
                end
            end
        end

        %% Update
        for i = 1:populationSize
            f1 = rand(1,dimension);
            f2 = rand(1,dimension);
            velocity(i,:) = omega * velocity(i,:) + c1 * f1 .* (best(i,:) -
                position(i,:)) + c2 * f2 .* (globalBest - position(i,:));
            position(i,:) = position(i,:) + velocity(i,:);
        end

        iterations = iterations + 1;
%         if lastBestValue ~= globalBestValue
%             disp(globalBestValue);
%             lastBestValue = globalBestValue;
%         end

    end

    %% Success
    if globalBestValue <= objectiveValue
        success = true;
    else
        success = false;
    end

    %% Return
    minimum = globalBest(1,:);
    value = globalBestValue;
end
```

A.3 Estimation of Distribution

```
function [ success, iterations, minimum, value ] = EDA_UMDA( CostFunction,
    dimension, lowerBound, upperBound, maxIterations, populationSize,
    objectiveValue )

    %% Parameters
    selectionThreshold = 0.5;

    %% Initialization
    selectionCount = floor(selectionThreshold * populationSize);
    samplingSize = populationSize-selectionCount;
    population = lowerBound + (upperBound - lowerBound) * rand(populationSize,
        dimension);



    value = inf;
    bestIndividualIndex = -1;
    sort_list = zeros(populationSize);
    population2 = zeros(selectionCount, dimension);

    lastBestValue = value;

    %% Iteration
    iterations = 0;
    while value > objectiveValue && iterations <= maxIterations

        %% Sorting & Evaluation
        for i = 1:populationSize
            sort_list(i) = CostFunction(population(i,:));
        end
        [~, ids] = sort(sort_list);

        %% Selection
        for i = 1:selectionCount
            ii = ids(i);
            population2(i,:) = population(ii,:);
        end

        %% Model building
        m = mean(population2);
        s = std(population2);
        %% Sampling
        NG = normrnd(repmat(m, samplingSize,1),repmat(s,samplingSize,1));

        %% New generation
        population = [population2; NG];

        %% Evaluation
        for i = 1:populationSize
            cost = CostFunction(population(i,:));
            if cost < value
                bestIndividualIndex = i;
                value = cost;
            end
        end

%         if lastBestValue ~= value
%             disp(value);
%             lastBestValue = value;
%         end

        iterations = iterations + 1;
    end

    %% Success
    success = false;
    if value <= objectiveValue
        success = true;
    end

    %% Return
    minimum = population(bestIndividualIndex,:);
end
```

A.4 Differential Estimation of Distribution

```
function [ success, iterations, minimum, value ] = DEDA( CostFunction, dimension,
    lowerBound, upperBound, maxIterations, populationSize, objectiveValue )

    %% Parameters
    selectionThreshold = 0.5;
    F = 0.6;
    pCR=0.9;

    %% Initialization
    selectionCount = floor(selectionThreshold * populationSize);
    population = lowerBound + (upperBound - lowerBound) * rand(populationSize,
        dimension);
    value = inf;
    bestIndividualIndex = -1;
    sort_list = zeros(populationSize);
    population2 = zeros(selectionCount, dimension);
    samplingSize = populationSize-selectionCount;

    lastBestValue = value;

    %% Iteration
    iterations = 0;
    while value > objectiveValue && iterations <= maxIterations

        %% Sorting & Evaluation
        for i = 1:populationSize
            sort_list(i) = CostFunction(population(i,:));
        end
        [~, ids] = sort(sort_list);

        %% Selection
        for i = 1:selectionCount
            ii = ids(i);
            population2(i,:) = population(ii,:);
        end
        %% Model building
        m = mean(population2);
        s = std(population2);


        %% Sampling
        NG = normrnd(repmat(m,samplingSize,1),repmat(s,samplingSize,1));

        %% Mutation
        for i=1:selectionCount
            x = population2(i,:);
            A = randperm(selectionCount);
            A(A==i)=[];
            a=A(1);
            b=A(2);
            c=A(3);
            y = population2(a,:) + F*(population2(b,:) - population2(c,:));
            y = max(y, lowerBound);
            y = min(y, upperBound);
            z = zeros(1, dimension);
            j0 = randi([1 numel(x)]);
            for j=1:numel(x)
                if j==j0 || rand <= pCR
                    z(j) = y(j);
                else
                    z(j) = x(j);
                end
            end
            if CostFunction(z) < CostFunction(population2(i,:))
                population2(i,:) = z;
            end
        end

        %% New generation
        population = [population2; NG];

        %% Evaluation
        for i = 1:populationSize
            cost = CostFunction(population(i,:));
            if cost < value
                bestIndividualIndex = i;
                value = cost;
            end
        end

%         if lastBestValue ~= value
%             disp(value);
%             lastBestValue = value;
%         end

        iterations = iterations + 1;
    end

    %% Success
    success = false;
    if value <= objectiveValue
        success = true;
    end

    %% Return
    minimum = population(bestIndividualIndex,:);
end
```