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: Peter Funk

Mälardalen University
Västerås, Sweden

Supervisor: Ning Xiong

Mälardalen University
Västerås, Sweden

April 6, 2017

Abstract

Evolutionary computation is a field which uses natural computational processes to optimize mathematical and industrial problems. Differential Evolution, Particle Swarm Optimization and Estimation of Distribution Algorithm are some of the newer emerging varieties which have attracted great interest among researchers. This work has compared these three algorithms on a set of mathematical and machine learning benchmarks and also synthesized a new algorithm from the three other ones and compared it with them. The results from the benchmark show which algorithm is best suited to handle various machine learning problems and presents the advantages of using the new algorithm. The new algorithms called DEDA (Differential Estimation of Distribution Algorithms) has shown promising results at both machine learning and mathematical optimization tasks.

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 resulted 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 specific problem.

2 Background

This section will aim to provide a general overview of the field of evolutionary computation. General terms and procedures which are often utilized in EC will be explained and the most well known traditional approaches will be presented. The necessary concepts from machine learning will be presented. The emerging algorithms relevant to this thesis can be found in the section "Algorithms".

2.1 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 uniformly. The initial 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 characteristics 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 encountered [5]. The concept is demonstrated in figure 1 with $P(t)$ representing the population at generation t.

Algorithm 1 Basic evolutionary algorithm

```
t ← 0
initialize P(t)
evaluate P(t)
while termination-condition not fulfilled do
    t ← t + 1
    select P(t) from P(t − 1)
    alter P(t)
    evaluate P(t)
end while
```

Here the fundamental building 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 necessary to properly understand them.

2.1.1 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 interchangeably 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 optimization problem of the form $f : \{0, 1\}^l \to \mathbb{R}$, where the mappings $h : M \to \{0, 1\}^l$ and $h' : \{0, 1\}^l \to M$ 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 useful 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].

2.1.2 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].

2.1.3 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.

2.1.4 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 receive a positive probability since the population might otherwise lose diversity and coalesce around a false local optimum [6].

Fitness proportional selection Fitness proportional selection (FPS) assigns a selection probability to an individual based on it's absolute fitness. This results in very good individuals overtaking the population quickly and premature convergence. If individuals have very similar fitness values the selection pressure becomes low. This mechanism is also very dependent on the exact form the of the fitness function [10].

Ranking selection In ranking selection the population if sorted according to the individuals fitness values and the selection pressure is kept constant. A constant number of the best individuals is the selected from the sorted list [10].

Tournament selection Tournament selection enables selection without global knowledge of the population's fitness. A number of individuals are chosen at random and the best one is selected. This process is repeated until the desired number of individuals is selected [10].

2.1.5 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].

Binary mutation The most commonly used mutation scheme for binary representations consists of randomly flipping bits (genes) in a chromosome with a certain probability. The number of alterations is not fixed with this approach, but the common procedures used can often be set to change a certain number of bits on average [9].

Binary recombination Three approached are normally used to recombine binary chromosomes. One-point crossover divides the chromosome into two sections, picking a random intersection point, and swaps the tails of the two chromosomes creating two offspring. N-point crossover generalizes this behavior by picking n random splitting points and assembling new chromosome by taking alternate sections of the parent chromosomes. Uniform crossover creates an array of uniform numbers from a probability distribution and chooses which parent to take a gene from by comparing the random value to a probability threshold [9].

Integer mutation Random resetting and creep mutation are used to mutate integer chromosomes. In random resetting integer values are changed with a certain probability. The new values are chosen at random from the pool of permissible values. Creep mutation samples small numbers from distributions and adds or subtracts them from genes at random [9].

Integer recombination For integer representations the same techniques that are used for binary representations apply [9].

Real-valued mutation For real-valued representations mutations is similar to integer mutation with the exception that new random values are drawn from continuous distributions and creep mutation used a gaussian distribution to sample values. A lower and upper bound is used to limit the span of the generated random numbers [9].

Real-valued recombination There are three common ways to recombine real-valued chromosomes. Discreet recombination work like n-point crossover and thus does not alter the values in the offspring chromosomes. Arithmetic recombination chooses values which fall in-between the values of the parent chromosomes for it's offspring. Blend recombination works like arithmetic recombination but allow for values which lie slightly outside of the interval defines by the parent genes [9].

Tree mutation Trees are usually mutated by selecting a node at random and re-generating it's subtree using the same approach which was used to generate the initial population [9].

Tree recombination Subtree crossover is commonly used to recombine trees. It randomly selects a node in each tree and then swaps the respective subtrees creating two new offspring [9].

2.1.6 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 referred 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].

2.1.7 Initatilization

Initialization is the process during which the initial 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].

2.1.8 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 population drops below a given threshold

2.2 Traditional Evolutionary Algorithms

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

2.2.1 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 multidimensional 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 [11]. See algorithm 2 and figure 1 for a simple GA.

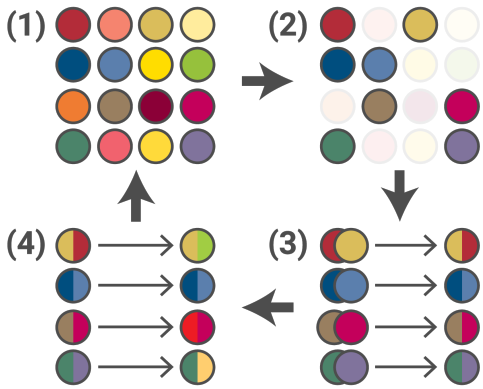

Figure 1: Stages in GA (1) Evaluation (2) Selection (3) Crossover (4) Mutation

Algorithm 2 Basic genetic algorithm

```
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 probabilistically 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
```

2.2.2 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].

2.2.3 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 [12]. The environment 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].

2.2.4 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].

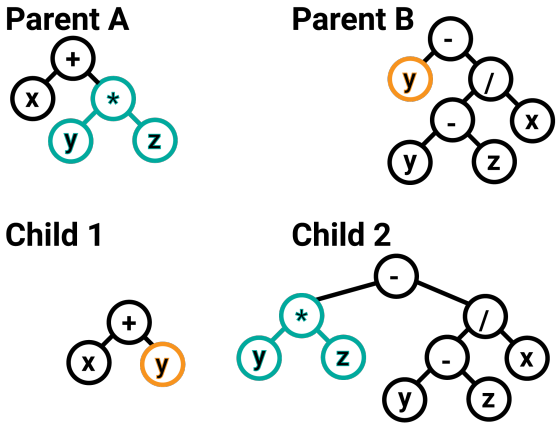

Figure 2: Crossover in GP with two parents producing two offspring

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], see figure 2.

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

2.3 Emerging Evolutionary Algorithms

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

2.3.1 Differential Evolution

Differential evolution [13] 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. See algorithm 3

Algorithm 3 DE algorithm

```
Init population within bounds
Evaluate fitness of populationCost
repeat
    for all Individuals x do
        Select three other random individuals a, b and c
        Create mutant vector v = a + F(b − c)
        Create trial vector u by blending x with v (taking at least one gene from u)
        Replace x with u if fitness(u) > fitness(x)
    end for
until End condition
```

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 $n = 1, 2, 3, ..., N$:

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

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. [14].

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 [15]. It's mutation equation is described below:

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

2.3.2 Particle Swarm Optimization

Particle Swarm Optimization (PSO) [16] was introduced in 1995 by Kenneth and Ebenhart [17]. 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. Neighbourhoods can be social or geographical. Social neighbourhoods 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.

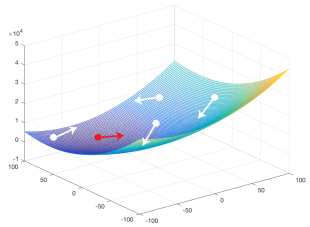

Figure 3: Illustration of particles in PSO (red represents the best particle)

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 algorithm 4. See figure 3 for an illustration of PSO.

Algorithm 4 PSO algorithm

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

In the benchmark the parameters for PSO have been set to $\omega = 0.8$ [18], $c_1 = c_2 = 1.494$ [19] and v_{max} = parameter range size [16].

2.3.3 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).

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 [20].

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 [21].

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 [22]. The algorithm is described in algorithm 5.

Algorithm 5 UMDA algorithm

```
Initialize population P
repeat
    Evaluate P
    Select the best t% of P into S
    Let m be the mean of 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
```

2.4 Machine learning concepts

The machine learning concepts needed for this thesis are artificial neural networks (ANN), specifically feed-forward neural networks (FFNN). Artificial neural networks are mathematical models which are based on the functioning of biological neuron in the brain. They are usefull for predicting future behaviour and events, trends, function approximation and data-classification. FFNNs are a popular type of ANN, often reffered to as mutli-layer perceptrons. ANNs have to be trained on training data in order to function properly and the most widely used and most popular method for this is backpropagation (BP) [23].

Backpropagation is a local minimization algorithm which works in n-dimensions. It can therefore be replaced by other optimiation algorithms such as genetic and other evolutionary algorithms. This is what will be considered in my thesis [23].

2.4.1 Feed-Forward Neural Networks

ANNs usually consist of five components

1. A graph containing nodes (neurons) and link between nodes
2. A variable which holds the state of each node
3. A real-valued weight for each link between nodes
4. A real-valued threshold for wach node

5. A transfer function which calculates the value of a nodes state variable based on the states of nodes which precede the node and are connected to it with links

FFNNs consists of one input layer of neurons which accept a vector of input signals, one or more "hidden" layers of intermediary neurons which process the signal and one output layer which produces the final result in the form of an output vector. In FFNNs all neurons are interconnected with all neurons in neigboring layers. These connections have weights which modulate the strenghs of the respective connections [24]. See figur 4 for a simple FFNN with two neurons (z_1 and z_2) in the input layer, which are linked to three hidden neurons (y_1, y_2 and y_3), which are in turn linked to two output neurons (o_1 and o_2). The weights are represented by v_{ij} and w_{ij}.

Evaluating the neural network The value of a neuron x_i in the network is determined according to the equations

$$x_i = f(\xi_i) \tag{11}$$

$$\xi_i = v_i + \sum_{j=0}^{n} \omega_{ij} x_j \tag{12}$$

where n is the number of neurons in the preceding layer, x_j are the neurons in the preceding layer and v_i are the weights between x_i and x_j.

The function f is the transfer function and is defines as

$$f(\xi) = \frac{1}{1 + exp(-\xi)} \tag{13}$$

The threshold is added as an extra neurons to each layer except the output layer to simplify the calculations. The threshold neuron always has the value of -1. The actuall threshold value is determined by the weights of the link going out of the threshold neuron. [25].

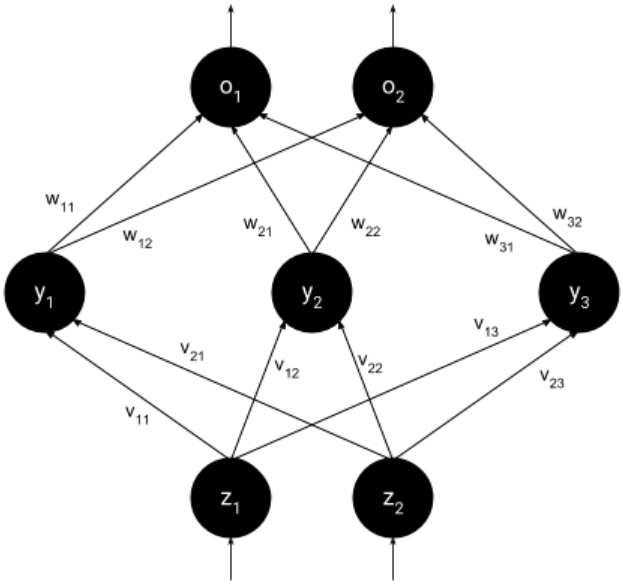

Figure 4: Feed-Forward Neural Network

Backpropagation Backpropagation (BP) is the most popular method for training neuron networks. Training data is send through the network and the output vector is subtracted from a result vector which contain the correct output values. An error value is then calculated by the means of the sum of squared errors. Subsequently all weights in the network are sligtly adjusted to accomodate the calculated error. After a number of iterations this process produces the correct combination of weights [4].

2.5 Method

A benchmark was constructed to perform the algorithm comparison. Both mathematical function optimization and machine learning test-sets were used. These are described in detail in the section 6. Matlab was used to implement the algorithms and benchmarks because it provides easy access to important mathematical and scientific constructs which are usually not available in conventional programming languages. Matlab is also widely used in the field of evolutionary computation.

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 processes, migration, etc [26].

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 [26].

3.1 Recent Research

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 [26].

3.2 Emerging Evolutionary Algorithms

Differential Evolution Differential evolution (DE) was conceived in 1995 by Storn and Price [27] and soon gained wide acceptance as one of the best algorithms in continuous optimization [28]. This spawned many new papers describing variations and hybrids of the algorithm [29], such as self-adaptive DE (SaDE) [15], opposition-based DE (ODE) [30] and DE with global and local neighborhoods (DEGL) [30].

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 [31, 32]. DE uses very few parameters and the effects of altering these parameters have been well studied [29]. DE comes in a total of 10 different varieties based on which mutation and cross-over operators are used [33]. 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 individuals) generally yields the best results [34]. [14] 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 include the use of fuzzy systems to control the parameters [35] and the SaDE algorithm [15].

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. [36] 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 [37]. Zhang et al. found a simple way to use control theory in order to find good parameters [38]. Yang proposed modified velocity PSO (MVPSO) in which particles learn the best parameters from the other particles [39].

Estimation of Distribution Algorithms Estimation of distribution algorithms (EDA) use probabilistic models to solve complex optimization problems. They have been successful at many engineering problems which at which other algorithms have failed, for instance: military antenna design, protein structure prediction, clustering of genes, chemotherapy optimization, portfolio management, etc [20].

Several techniques have been proposed to make EDA more efficient. Parallelization of fitness evaluation and model building has proven effective [40]. Local optimization techniques such as deterministic hill climbing (DHC) has been shown to make EDA faster [41].

3.3 Evolutionary Algorithms and 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 [42].

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 [43]. 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 back-propagation (BP) algorithms often get stuck in [44]. Larrañaga and Lozano [45] tested various EC algorithms (GA, EDA and ES) agains BP and concluded that EC is a competitive alternative to traditional approaches.

3.4 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 [32, 28], the application of EC to machine learning has not been studied with as much detail. My primary contribution to the field will be to find what algorithms work best for different ML problems and 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.

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 aforementioned benchmarks and problem sets?

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

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

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 Algorithm

My improved algorithm, Differential Estimation of Distribution (DEDA), draws upon DE and EDA, applying differential mutation to the selection population of the EDA algorithm. The algorithm is described in algorithm 6. The Matlab code can be viewed in appendix **??**.

Algorithm 6 Improved algorithm

Initialize random population $P(0)$
repeat
  Sort population $P(g)$
  Select $S(g)$ from $P(g)$
  Build probabilistic model $M(g)$ from $S(g)$
  Sample $P'(g)$ from $M(g)$
  Create $S_{DE}(g)$ by applying differential mutation, recombination and selection to $S(g)$
  Create new generation $P(g)$ from $P'(g)$ and $S_{DE}(g)$
  $g = g + 1$
until Termination condition

The algorithm belongs to the class of estimation of distribution algorithms, which use probability distributions to sample populations. Initially a population $P(0)$ is created by uniformly sampling values from a problem-specific parameter interval. Then the individuals in $P(g)$ (g stands of generation) are evaluated and the best solutions are selected into $S(g)$ (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). A probabilistic model $M(g)$ is then constructed from the selected population $S(g)$. New individuals $P'(g)$ are sampled from $M(g)$. The unselected individuals $P_u(g)$ in $P(g)$ are subjected differential mutation, cross-over and selection and produce a new set of individuals S_{DE}. Algorithm 7 explains how $S_{DE}(g)$ is created. Finally $P(g + 1)$ is created by combining $S_{DE}(g)$ with $P'(g)$. This procedure is repeated until the best solution in $P(g)$ is good enough or until a pre-determined number of evaluations/iterations is reached.

Algorithm 7 Differential mutation, recombination and selection used in DEDA

for all Individuals x **do**
  Select three other random individuals a, b and c
  Create mutant vector $v = a + F(b - c)$
  Create trial vector u by blending x with v (taking at least one gene from u)
  Replace x with u if $fitness(u) > fitness(x)$
end for

The motivation behind the new algorithm is that EDA produces useful new genetic material when sampling it's probability model M but fails to create anything new from the selected population which is permitted to proceed into the next generation unaltered. By applying the principles of differential evolution, which only alter a chromosome if the alteration improves the individual, the selected population can be further improved before entering the next generation without risking any detrimental effects. Furthermore, the next algorithm will be able to approach problems from two perspectives at once, possibly enlarging it's area of applicability.

6 Experimental Results and Evaluation

This section describes the design of five benchmarks (F_{1-10}, FA_{1-6}, CLS_{1-7}, CLU_1, IGC_1), presents the results of the measurements and discusses their significance.

6.1 Mathematical Function Optimization (F_{1-10})

The functions F_{1-10} for the mathematical function optimization benchmark have been taken from the 2005 CEC conference on continuous evolutionary optimization algorithms [46]. Many functions from the well known DeJong test-bed are included [47].

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 [46]. The functions are illustrated in two dimensions in figures 5, 6, 7, 8, 9, 10, 11, 12, 13 and 14. The benchmark parameters and settings are listed in table 1.

Parameter	Value (D=10)	Value (D=30)	Value (D=50)
Repeat measurements	30	30	30
Generations	667	1200	1429
Population Size	150	250	350

Table 1: Benchmark parameters for F_{1-10}

Shifted Sphere Function (F_1) Unimodal, Separable

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

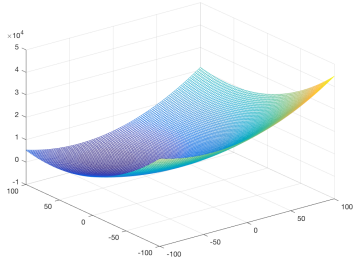

Figure 5: 3-D map for 2-D function

Shifted Schwefel's Problem (F_2) Unimodal, Non-Separable

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

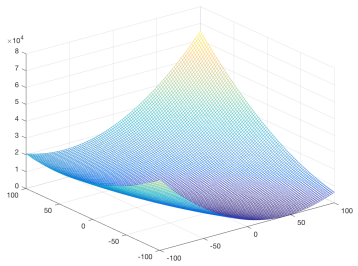

Figure 6: 3-D map for 2-D function

Shifted Rotated High Conditioned Elliptic Function (F_3) Unimodal, Non-Separable

$$F_3(x) = \sum_{i=1}^{D} (10^6)^{\frac{i-1}{D-1}} z_i^2$$

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

$$x \in [-100, 100]^D$$

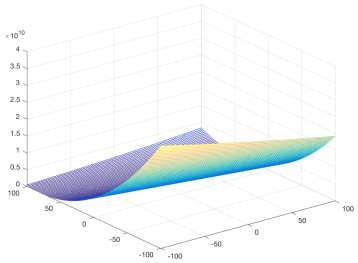

Figure 7: 3-D map for 2-D function

Shifted Schwefel's Problem with Noise in Fitness (F_4) Unimodal, Non-Separable

$$F_4(x) = (\sum_{i=1}^{D} (\sum_{j=1}^{i} z_j)^2) * (1 + 0.4|N(0,1)|)$$

$$z = x - o$$

$$x \in [-100, 100]^D$$

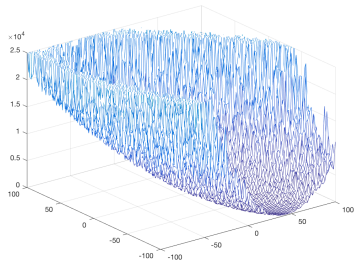

Figure 8: 3-D map for 2-D function

Schwefel's Problem with Global Optimum on Bounds (F_5) Unimodal, Non-Separable

$$F_5(x) = max\{|A_i x - B_i|\}$$

$$i = 1, ..., D, x \in [-100, 100]^D$$

A is a $D*D$ matrix, $a_{ij} =$ random numbers in $[-500, 50$

$B_i = A_i * o, o_i =$ random numbers in $[-100, 100]$

$o_i = -100$, for $i = 1, 2, ..., [D/4], o_i = 100$, for $i = [3D/4]$

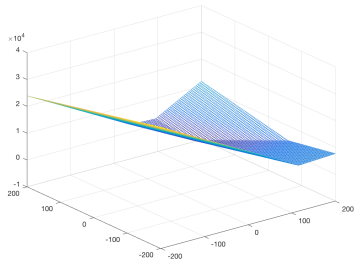

Figure 9: 3-D map for 2-D function

Shifted Rosenbrock's Function (F_6) Multimodal, Non-Separable

$$F_6(x) = \sum_{i=1}^{D-1} (100(z_i^2 - z_{i+1})^2 + (z_i - 1)^2)$$

$$z = x - o + 1$$

$$x \in [-100, 100]^D$$

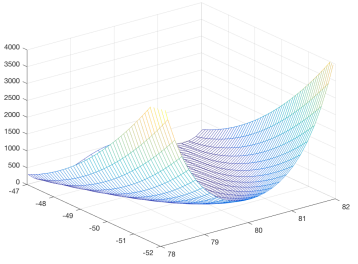

Figure 10: 3-D map for 2-D function

Shifted Rotated Griewank's Function without Bounds (F_7) Multimodal, Non-Separable

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

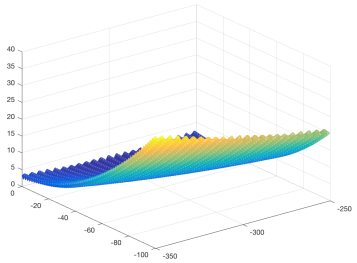

Figure 11: 3-D map for 2-D function

Shifted Rotated Ackley's Function with Global Optimum on Bounds (F_8) Multimodal, Non-Separable

$$F_8(x) = -20 \exp\left(-0.2 \sqrt{\frac{1}{D} \sum_{i=1}^{D} z_i^2}\right) - \exp\left(\frac{1}{D} \sum_{i=1}^{D} \cos\left(2\pi z_i\right)\right)$$
$$z = (x - o) * M$$
$$x \in [-32, 32]^D$$

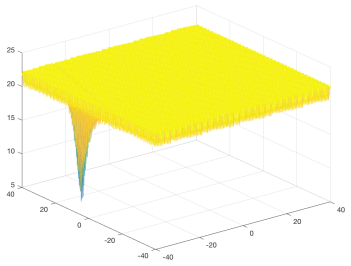

Figure 12: 3-D map for 2-D function

Shifted Rastrigin's Function (F_9) Multimodal, Separable

$$F_9(x) = \sum_{i=1}^{D} z_i^2 - 10 \cos\left(2\pi z_i\right) + 10$$
$$z = x - o$$
$$x \in [-5, 5]^D$$

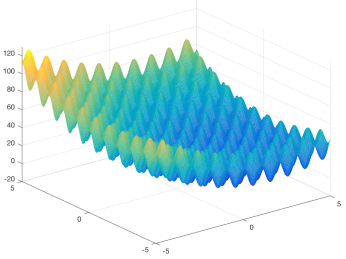

Figure 13: 3-D map for 2-D function

Shifted Rotated Rastrigin's Function (F_{10}) Multimodal, Non-Separable

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

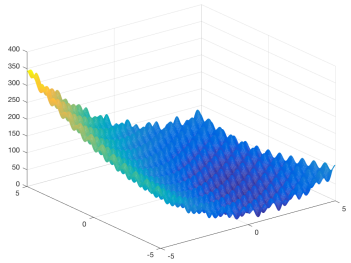

Figure 14: 3-D map for 2-D function

6.2 Function Approximation (FA_{1-6})

The optimization algorithms were used find the optimal weights for feed-forward neural networks. The neural networks were evaluated on six function approximation data-sets which are available in Matlab's Neural Network Toolbox. The benchmark parameters and settings are listed in table 8. The data-sets are listed in table 3.

Parameter	Value
Repeat measurements	5
Generations	250
Population Size	50

Table 2: Benchmark parameters for FA_{1-6}

Data-set	Description	Input-Dimension	Output-Dimension	Weight-Dimension
simplefit	Simple fitting	1	1	4
bodyfat	Body fat percentage	13	1	196
chemical	Chemical sensor	8	1	81
cho	Cholesterol	21	3	528
engine	Engine behavior	2	2	12
house	House value	13	1	196

Table 3: Data-sets for FA_{1-6}

Evalutation Procedure 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 optimizes is the sum of squared errors as defined by equation 14

$$sse(x, t) = \frac{1}{2n} \sum_{i=1}^{n} (y(x) - t)^2 \tag{14}$$

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.

6.3 Classification (CLS_{1-7})

The optimization algorithms were used find the optimal weights for feed-forward neural networks. The neural networks were evaluated on seven classification data-sets which are available in Matlab's Neural Network Toolbox. The benchmark parameters and settings are listed in table 4. The data-sets are listed in table 5.

Parameter	Value
Repeat measurements	5
Generations	250
Population Size	50

Table 4: Benchmark parameters for CLS_{1-7}

Data-set	Description	Input-Dimension	Output-Dimension	Weight-Dimension
simpleclass	imple pattern recognition	2	4	18
cancer	Breast cancer	9	2	110
crab	Crab gender	6	2	56
glass	Glass chemical	9	2	110
iris	Iris flower	4	3	35
thyroid	Thyroid function	21	3	528
wine	Italian wines	13	3	224

Table 5: Data-sets for CLS_{1-7}

Evalutation Procedure Same as FA_{1-6}.

6.4 Clustering (CLU_1)

The optimization algorithms were used find the optimal weights for feed-forward neural networks. The neural networks were evaluated on one clustering data-set which is available in Matlab's Neural Network Toolbox. The benchmark parameters and settings are listed in table 6. The data-sets are listed in table 7.

Parameter	Value
Repeat measurements	5
Generations	250
Population Size	50

Table 6: Benchmark parameters for CLU_1

Data-set	Description	Input-Dimension	Output-Dimension	Weight-Dimension
simplecluster	Simple clustering	2	4	18

Table 7: Data-sets for CLU_1

Evalutation Procedure Same as FA_{1-6}.

6.5 Intelligent Game Controllers (IGC_1)

Machine learning test set IGC_1 uses evolutionary algorithms to evolve weights for feed-forward neural networks, which control the behavior of a game-agent in the classic game of "Snake". The benchmark parameters and settings are listed in table 8.

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

Table 8: Benchmark parameters for FA_{1-6}

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 interconnected 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 current 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 15 illustrates the game grid. Figure 8 described the algorithm for the game. The game's parameters are explained in table 9.

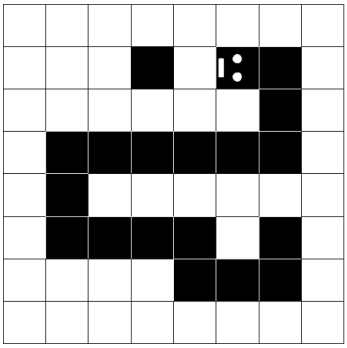

Figure 15: Snake $8 * 8$ game-field with snake hunting food

Parameter	Description
D^2	Dimension of grid ($m * n$ blocks)
(x, y)	Starting point for snake head
d_s	Starting direction for snake head ($= \{left, right, up, down\}$)
t	Starvation threshold

Table 9: Parameters snake game

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-dimensional 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 immediately 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.

Algorithm 8 Snake game

```
initializeSnake()
foodEaten ← 0
movesMade ← 0
while alive do
    state ← getGameState()
    move ← 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 ← foodEaten + 1
    end if
    movesMade ← movesMade + 1
end while
return foodEaten + sigmoid(movesMade)
```

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 16 illustrates the network.

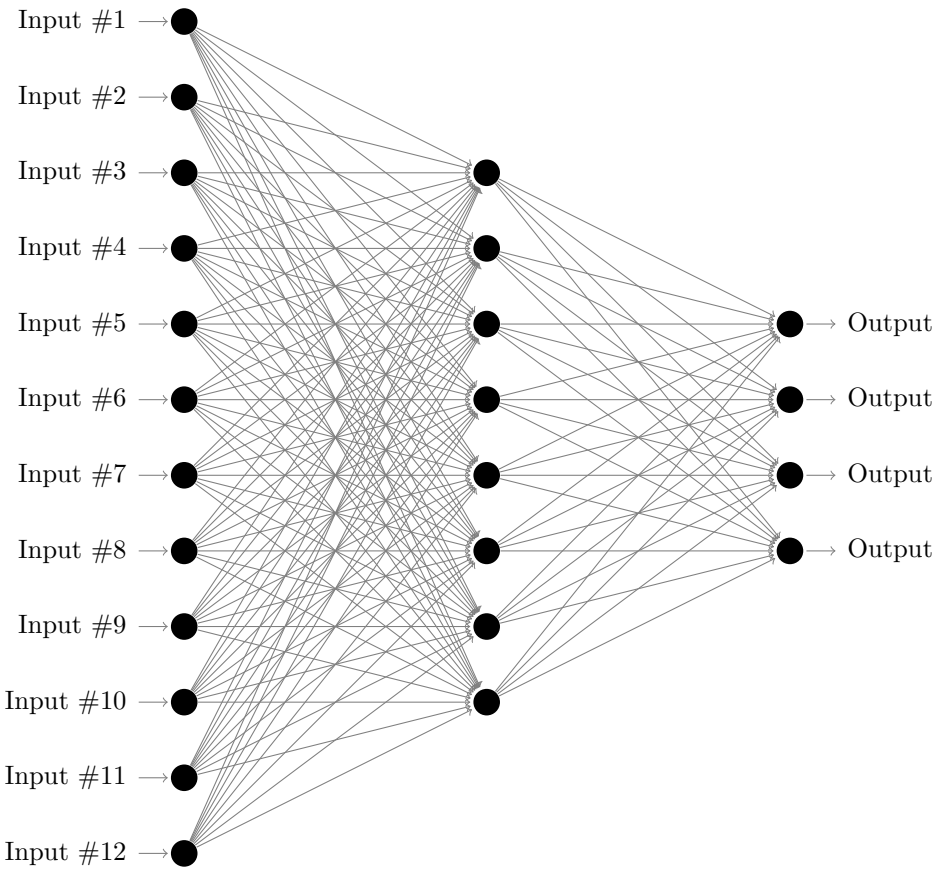

Figure 16: FFNN snake controller

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 15.

$$fitness = \text{foodEaten} + \frac{1}{1 - e^{-\text{movesMade}}} \tag{15}$$

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

6.6 Results

This section presents results for the benchmarks of the four algorithms presented in this thesis.

Function Optimization at D=10 DEDA significantly outperforms the other algorithms at most unimodal functions. DE significantly outperforms the others at one unimodal and one multimodal function. PSO outperforms the others by a slight margin on most multimodal function. Separability does not seem to influence the results. Table 10 presents the data.

For F_1 EDA significantly outperforms DE and PSO, while DEDA lags far behind. For F_2 DEDA finds significantly better results than the rests, while EDA performs the worst. For F_3 DEDA performs best followed by DE, but PSO and EDA do not find good solutions at all. For F_4 DEDA significantly outperforms the rest and EDA does not find any good results at all. For F_5 DE significantly outperforms the rest, DEDA finds an acceptable results and PSO and EDA do not find any good results. For F_6 DE significantly outperforms the rest, DEDA finds an acceptable results and PSO and EDA do not find any good results. For F_7 EDA find the best solution but all algorithms achieve similar results. For F_8 PSO find the best solution but all algorithms achieve

similar results. For F_9 PSO find the best solution but all algorithms achieve similar results. For F_{10} PSO find the best solution but all algorithms achieve similar results.

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

Table 10: Benchmark results for F_{1-10} $D = 10$

Function Optimization at D=30 DEDA moderately outperform the other algorithms at most unimodal functions. PSO outperform all other algorithms on the multimodal functions, but the results are fairly even. DEDA significantly outperforms the others on separable unimodal algorithms. Table 11 presents the data.

For F_1 DEDA performs best closely followed by EDA. PSO also finds a good result but DE finds a significantly worse result. For F_2 all algorithms find similar solutions and DEDA finds the best one. For F_3 all algorithms find similar solutions and EDA finds the best one. For F_4 all algorithms find somewhat similar solutions and DEDA finds the best one. For F_5 all algorithms find somewhat similar solutions and DEDA finds the best one. For F_6 all algorithms find somewhat similar solutions and DEDA finds the best one. For F_7 PSO finds the best solution followed EDA and DE and DEDA find worse solutions. For F_8 all algorithms find similar solutions and PSO finds the best one. For F_9 DE, EDA and DEDA find similar solutions, but PSO finds a somewhat better solution. For F_{10} all algorithms find similar solutions and PSO finds the best one.

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

Table 11: Benchmark results for F_{1-10} $D = 30$

Function Optimization at D=50 The results are fairly even. DEDA significantly outperforms the others on separable unimodal algorithms. The multimodal functions favor PSO, the unimodal function favor UMDA while DEDA's results are more scattered. Table 12 presents the data.

For F_1 DEDA find the best results closely followed by EDA. DE finds a bad solution and PSO is in-between DE and EDA. For F_2 DE, PSO and DEDA find similar solutions. EDA outperforms the all by a moderate margin. For F_3 all algorithms perform similarly. EDA performs best, followed by PSO. For F_4 EDA performs best, followed by DE and DEDA. PSO finds the worst solution. For F_5 DEDA finds the best solution, closely followed by EDA, while DE and PSO find somewhat worse solutions. For F_6 DE performs signigicantly worse than the rest, which perform very similarly. EDA finds the best solution. For F_7 DEDA finds a good solution while PSO performs somewhat poorer and DE and EDA find the worst solutions. For F_8 all algorithms find similar solutions.

PSO finds the best solution. For F_9 all algorithms find somewhat similar solutions. PSO finds the best solution. For F_{10} all algorithms find similar solutions. PSO finds the best solution.

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

Table 12: Benchmark results for F_{1-10} $D = 50$

Function Approximation The results are fairly even but DEDA performs best overall. PSO is second best. Table 13 presents the data.

For *simplefit* DE and PSO find the best solutions. For *bodyfat* DEDA finds the best solution but all algorithms deliver similar results. For *chemical* DEDA, PSO and DE find similar solutions. DEDA performs best and EDA finds a much worse solution. For *cho* PSO finds the best solution but all algorithms deliver similar results. For *engine* DEDA performs best, closelly followed by PSO. DE and EDA perform somewhat worse. For *house* PSO finds the best solution but all algorithms deliver similar results.

F	DE		PSO		EDA		DEDA	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
simplefit	**6,91E-01**	1,67E-16	6,91E-01	3,90E-04	6,64E+00	7,12E-01	1,09E+00	1,13E-01
bodyfat	1,92E+01	2,59E+00	1,90E+01	2,14E+00	1,96E+01	2,11E+00	**1,69E+01**	1,26E+00
chemical	2,58E+01	5,53E-04	2,46E+01	1,97E+00	1,15E+05	2,38E+03	**2,43E+01**	2,11E+00
cho	2,07E+03	2,16E+02	**1,86E+03**	4,84E+01	5,32E+03	6,24E+02	1,97E+03	1,10E+02
engine	1,29E+05	7,09E+04	7,86E+04	6,86E+03	9,91E+05	2,59E+03	**5,61E+04**	1,27E+04
house	3,19E+01	4,81E-01	**2,00E+01**	1,93E+00	2,45E+01	2,68E+00	2,03E+01	5,90E+00

Table 13: Benchmark results for F_{1-10}

Classification DEDA and UMDA perform best. The results are even overall. Table 14 presents the data.

For *simpleclass* DEDA finds the best solution but all algorithms deliver similar results. All algorithms find good solution for *cancer* but EDA has a slight edge. For *crab* EDA finds the best solutions, closely followed by PSO and DE, while DEDA finds a somewhat worse solution. For *glass* PSO finds the best solutions, closely followed by EDA and DEDA, while DE finds a somewhat worse solution. For *iris* DEDA finds the best solution and DE, PSO and EDA find somewhat worse solutions. For *thyroid* EDA finds the best solution but all algorithms deliver similar results. For *wine* DEDA finds the best solution but all algorithms deliver similar results.

F	DE		PSO		EDA		DEDA	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
simpleclass	1,76E-01	1,29E-02	1,83E-01	7,07E-03	1,97E-01	2,56E-02	**1,43E-01**	7,76E-03
cancer	7,74E-02	9,95E-03	6,48E-02	1,40E-02	**3,23E-02**	9,32E-04	5,26E-02	6,65E-03
crab	1,83E-01	3,25E-02	1,27E-01	2,54E-02	**1,20E-01**	3,70E-02	6,35E-02	5,26E-02
glass	1,25E-01	3,54E-02	**7,45E-02**	1,95E-02	9,71E-02	2,11E-02	9,91E-02	4,33E-02
iris	1,63E-01	7,31E-03	1,26E-01	3,22E-02	1,31E-01	3,18E-02	**6,67E-02**	5,23E-02
thyroid	2,00E-01	3,66E-02	2,25E-01	4,84E-02	**1,17E-01**	1,24E-02	1,80E-01	3,57E-02
wine	3,45E-01	2,33E-02	3,06E-01	2,25E-02	2,79E-01	2,52E-02	**2,59E-01**	7,80E-02

Table 14: Benchmark results for CLS_{1-6}

Clustering The results are fairly even but DEDA performs best overall. Table 15 presents the data.

F	DE		PSO		EDA		DEDA	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
simplecluster	1,65E-01	1,39E-02	1,74E-01	1,57E-02	2,17E-01	2,05E-02	**1,51E-01**	8,12E-03

Table 15: Benchmark results for CLU_1

Intelligent Game Controllers DE performs significantly worse than the other algorithms. PSO, UMDA and DEDA performs similarly, with PSO showing the best results. Table 16 presents the data.

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

Table 16: Benchmark results for IGC_1

6.7 Discussion

PSO is the best algorithm for multimodal functions and also performs best at controlling neural networks in game controllers. DEDA performs well at unimodal function at all dimensions, while UMDA works well at higher dimensions. DEDA produces the best results when optimizing neural networks. UMDA also works well when optimizing neural networks for clustering.

7 Conclusions

This thesis has compared four evolutionary algorithms on mathematical and machine learning benchmarks. The four algorithms are Differential Evolution (DE), Particle Swarm Optimization (PSO), Estimation of Distribution Algorithm (EDA) and my own algorithm Differential Estimation of Distribution Algorithm (DEDA).

The algorithms perform similarly on many of the benchmark problems, while some problems showed wide ranging differences. For the mathematical benchmarks PSO and DEDA performed best, while DEDA and EDA worked best for machine learning problems on standard data sets. PSO produces the best results for neural network game controllers. The results show that combining features from DE and EDA into a new algorithm (DEDA) provides a clear benefit in machine learning and good performance overall.

8 Future Work

Since there exit many variations of the three popular algorithms presented in this thesis, it would be interesting to compare different varieties of algorithms which performed best on the machine learning problem sets in order to find specific optimization which work well with machine learning.