Coventry University
Faculty of Engineering, Environment and Computing
School of Computing, Electronics and Mathematics

Designing and Implementing a Differential Evolution Algorithm to Train Artificial Neural Networks

Author: Ryan Barnes-Batterbee
SID: 5818260
Supervisor: Mauro Innocente

Submitted in partial fulfilment of the requirements for the Degree of
Master of Data Science and Computational Intelligence
Academic Year: 2018

Declaration of Originality

This project is all my own work and has not been copied in part or in whole from any other source except where duly acknowledged. As such, all use of previously published work (from books, journals, magazines, the Internet, etc) has been acknowledged within the main report to an entry in the References list.

I agree that an electronic copy of this report may be stored and used for the purposes of plagiarism prevention and detection.

I understand that cheating and plagiarism constitute a breach of University Regulations and will be dealt with accordingly.

Signed: Ryan Barnes-Batterbee Date: 20/8/2018

Abstract

In this paper a new Differential Evolution (DE) algorithm is developed to train Artificial Neural Networks (ANNs). ANNs are traditionally trained using gradient-based algorithms which whilst efficient tend to converge towards local optima. DE algorithms are population-based algorithms which mimic the natural evolution of biological systems. They investigate a search space globally by iteratively trying to improve a candidate solution. The simulation of competing individuals in a population is achieved using three search operations: crossover, mutation and selection. Crossover is analogous to reproduction and is when multiple individuals are used to create new individuals. Mutation is analogous to biological mutation and is when individuals in the population are changed to maintain diversity. Selection is when the best individuals from the current population which should be used in a new offspring population are selected. The crossover and mutation strategies used by a DE algorithm can have a significant impact on its performance.

The DE algorithm developed in this paper, SADEGL, uses self-adaptive crossover and mutation strategies. The crossover and mutation strategies involve choosing between multiple crossover and mutation operators based on how successful they have been in recent iterations. The mutation operators are unique in that one is focused on exploiting the search space globally, another on exploiting the search space locally, meaning it is also exploring the search space globally, and a third operator focused purely on unbiased exploration of the search space.

The SADEGL is benchmarked using the CEC'2013 test suite. The performance of the algorithm is impressive and comparable to the 10^{th} best algorithm presented in the CEC'2013 competition. The SADEGL algorithm is then used to train ANNs on the PROBEN1 datasets. The network weights and biases obtained using SADEGL are fine tuned using BP. The performance on these datasets is compared to that of the same network trained using BP. SADEGL-BP consistently achieves better results than BP, at the expense of a much higher computational cost.

Additionally, a simple application is developed to allow users to train their own neural networks with SADEGL-BP.

Acknowledgements

I would like to thank my project supervisor Mauro Innocente who supported and advised me throughout my time working on this project. From the very start of this projects conception he was extremely enthusiastic about the topic, our scheduled meetings would often last much longer than intended as he was always willing to spare some extra time to discuss my projects progress. He encouraged me to focus on the aspects of the project I found most interesting, and his knowledge of the optimization field proved to be truly valuable.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Artificial Neural Networks (ANNs) are computing systems inspired by biological neural networks. They are used as surrogate and data-driven models, which are trained using known data, so that they can accurately predict new data. During training the coefficients of ANNs are formulated such that they minimise some defined error function. Data scientists often need to model data where finding the exact solution is far too computationally complex or is undesirable. They require models which provide accurate results, have a fast training time for known data and a fast testing time for new data. Therefore, they require well optimised models to represent data. ANNs are frequently used to successfully model data, however traditional training algorithms are gradient-based which whilst efficient, tend to converge towards local optima. Hence, in recent years alternative training algorithms have been the focus of much research.

Evolutionary Algorithms (EAs) are population-based optimisation algorithms which simulate the natural evolution of biological systems. The simulation of competing individuals in a population is achieved using three search operations: crossover, mutation and selection. Crossover is analogous to reproduction and is when multiple individuals are used to create new individuals. Mutation is analogous to biological mutation and is when individuals in the population are changed to maintain diversity. Selection is when the best individuals from the current population which should be used in a new offspring population are selected. The procedure of a

typical EA is to first generate an initial population, then evaluate the individuals of the population and finally apply the search operators appropriately to create a new offspring population. This process is repeated with the new population until some termination criterion is satisfied. Since EAs are stochastic global search methods they can find near optimum solutions, but not necessarily consistently.

The aim of this project was to design and implement an Evolutionary algorithm optimised to train Artificial Neural Networks.
Initially I investigated how EAs are currently used to train ANNs. The purpose of this was to compare and contrast the performance of various EAs when used to train ANNs.
Secondly, using the knowledge gained in the previous step I designed an EA optimized to train ANNs. The EA was then implemented, and its performance was evaluated using the CEC'2013 functions [9]. The algorithms performance was comparable to that of the 10^{th} best algorithm presented in the CEC 2013 competition results comparison paper [10].
Thirdly, the EA was used to train single layer ANNs using the PROBEN1 datasets [16]. Subsequently the network was trained using a gradient based method to fine tune the network weights obtained using the EA. Additionally, the networks were trained using only a gradient based algorithm and the results were compared to that of the designed EA-gradient hybrid algorithm.
Fourthly, a user interface was designed and implemented for the EA. The interface allows users to train ANNs using their own data and control parameters.
Finally, the performance of the EA is summarized and future adjustments which could improve the algorithm are detailed.

Chapter 2

Background Research and Literature Review

2.1 Backpropagation

Backpropagation (BP) is a gradient-based optimization method commonly used to train the weights of ANNs. When training a network using BP, the weights of the network are usually randomly initialized. Training data is then presented to the network and a a two-phase process is repeated: Forward pass and Backward pass. When data is presented to the network, it is propagated forward through the network, layer by layer, until it reaches the output layer. The output of the network is then compared to the target output, using a loss function. The resulting error value is calculated for each of the neurons in the output layer. The error values are then propagated from the output back through the network, layer by layer, until each neuron has a corresponding error value that reflects its contribution to the original output.

BP is a gradient based method, and therefore there is no guarantee that the global minimum will be reached. This is because if the intial weights of the network are chosen inappropriately, the BP algorithm can become stuck in a local minimum. Hence, despite how successful BP has been in training ANNs in the past, currently research is focused on finding alternative training algorithms which can produce well optmized networks without the need for such careful selection

of the intial weights [12].

2.1.1 Backpropagation Algorithm

Consider a neural network which corresponds to the function $y = f(w, x)$, which given a weight vector w, maps an input vector x to an output vector y.

1. Initialize the weights of the network
2. Training examples are presented to the network $(x_1, y_1), (x_2, y_2), ...(x_p, y_p)$.
3. The difference between the target outputs and the predicted output is used to calculate an overall error function.
4. The weights at the output layer are updated. Then the weights at each hidden layer are updated in backwards order.
5. Steps 2-4 are repeated until stopping criteria is met.

2.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are population-based metaheuristic optimization algorithms. EAs use mechanisms inspired by biological evolution, such as reproduction, mutation, and natural selection. Candidate solutions to the optimization problem play the role of individuals in a biological population. A fitness function determines the quality of these solutions and is used by the optimization algorithm to mimic "Survival of the Fittest". Evolution of the population takes place using reproduction, mutation, and selection operators. Unlike gradient based methods, EAs are non-deterministic, meaning the final network weights can be different on different runs, even when all the input variables are the same. Hence, whilst it is unlikely to get stuck in local optima due to its initialization, it is still not guaranteed to find the optimal solution. John Holland [6] popularized the first type of EAs Genetic Algorithms (GAs) in

the 1970's. Whilst it remains one of the most commonly used EAs, there also exists popular alternatives such as Differential Evolution. Swarm Intelligence (SI) algorithms such as Particle Swarm Optimisation (PSO) are often used for the same purposes as EAs are, however they mimic the collective behaviour of decentralized, self-organized biological systems, rather than evolution.

2.2.1 Exploration and Exploitation

Currently the design of EAs is largely focused on designing algorithms which achieve a suitable balance between exploration and exploitation. Consider exploration and exploitation as two different operations of an EA. Exploitation consists of investigating a limited but promising region of the search space with the hope of improving upon a promising solution S that an algorithm has already found. This operation therefore involves searching in the neighbourhood of S to improve upon it. However, exploration consists of investigating a much larger portion of the search space with the hope of finding other promising solutions that are yet to be refined. This operation therefore amounts to diversifying the search to avoid getting trapped in a local optimum.

Crepinsek [3] defined exploration and exploitation using the euclidean distance between individuals. The authors definition decides if a new individual at iteration T is exploring or exploitation based on how similar it is to another individual. They suggest various similarity measurements to determine if a new individual ind_{new} is exploring or exploiting, such as the distance to its parents, and the smallest distance to another individual in the previous population. However, the most appropriate similarity measurement for the closest neighbour SCN suggested is calculating the distance between the new individual and the nearest individual in the entire history of populations $\{P^t | t = 1, 2..., T\}$. Therefore when SCN is greater,equal or smaller than a threshold value X, a new individual can be categorized as either exploring and or exploiting. Hence where

$$SCN(ind_{new}, P^t) = min\ d(ind_{new}, ind)$$

a new individual is considered to be exploring when

$$SCN(ind_{new}, P^t) > X$$

and exploiting when

$$SCN(ind_{new}, P^t) \leq X.$$

This definition accurately describes an informal definition of exploitation and exploration, where individuals within the neighbourhood of previously visited points are considered to be exploiting , and all other individuals are exploring. However, this definition is not very useful in terms of piratical applications. The time required to calculate whether an individual is exploring or exploiting would increase exponentially with each iteration, and storing each point would require a very large amount of memory. Additionally, defining exploitation as exploiting promising regions of the search space rather than any region discovered is much more beneficial. Therefore, a useful formal mathematical definition of exploration and exploitation, should be focused on determining if an individual is exploiting an identified promising region of the search space or not.

Clerc [2] defined exploitation using mathematical formula as searching around good positions found in a search space. Consider a 1 dimensional search space $[x_{min}, x_{max}]$ and the following points $(p_0, p_1, ..., p_N, p_{N+1})$, where $(p1, ..., p_N)$ are the best positions found by an EA, $p_0 = x_{min}$ and $p_{N+1} = x_{max}$. Choose $\rho > 0$ which controls how large the region around a positions p_i should be considered promising. Note that if $p \geq 0.5$ the union of the regions is a single connected domain. Therefore, we can consider N local exploitation areas

$$e_i = [p_i - \rho(p_i - p_{i-1}), p_i + \rho(p_{i+1} - p_i)],\ i \in \{1, .., N\}.$$

This definition can easily be extended to a D dimensional search space with best positions represented by vectors $p_i = (p_{i,1}, p_{i,2}, ..., p_{i,D})$. Therefore, we have N local exploitation areas defined by

$$E_i = e_{i,1} \times ..... \times e_{i,D}, i \in \{1, ..., N\},$$

where on each dimension d we define intervals

$$e_{i,d} = [p_{i,d} - \rho(p_{i,d} - p_{i-1,d}), p_{i,d} + \rho(p_{i+1,d} - p_{i,d})],\ d \in \{1, .., D\}.$$

This definition of exploration and exploitation can easily be used practically to determine the exploitation and exploration rate of an algorithm at iteration t. Let us define $S_E(t)$ as the number of individuals inside a local exploitation area at iteration t. Therefore, using Clercs definition of exploitation we define the exploitation rate as

$$r_1(t) = \frac{S_E(t)}{S}.$$

Hence, we can also define the exploration rate as

$$r_2(t) = 1 - r_1(t).$$

Therefore, when aiming to adjust the balance of exploration and exploitation in an EA, these rates can be used as metrics to determine if adjustments to an algorithm are having the intended impact. For instance, if an EA is designed to focus on exploiting promising regions of the search space at iterations $T > t$ and exploration otherwise, then $r_1(T)$ should be significantly greater than $r_1(t)$.

2.2.2 Differential Evolution

Differential Evolution (DE) is currently one of the most popular EAs used to train ANNs [4]. A basic variant of the DE algorithm begins with a population of candidate solutions. These solutions are moved around in the search-space by using mathematical formulae to combine the positions of existing solutions. If the new position of a solution is an improvement then it is accepted and forms part of the population, otherwise the new position is discarded. The process is repeated and by doing so it is hoped, but not guaranteed, that a satisfactory solution will eventually be discovered. The mathematical formulae used to determine a solutions new

position is defined by the mutation and crossover strategies chosen.

The performance of DE primarily depends on the mutation strategy, the crossover strategy, and the control parameters: scale factor F, crossover rate CR, and population size NP. There are multiple distinct mutation strategies and crossover schemes proposed for use in DEAs. Each of these mutation and crossover operations may be effective for certain problems but perform poorly for others. Similarly, the performance of DE is dependant on the tuning of the control parameters F, CR and NP. Many researchers have reported results of DEAs which demonstrate the impact of parameter tuning [1], [8] . Therefore, researchers have naturally started to develop techniques to automatically find the optimal value for these control parameters. These techniques are known as adapative and self-adaptive strategies, and they often improve algorthims, however the programming requrired for them is usually complex and can increase the number of function evaluations required [5].

Mutation Strategies

DE creates a donor vector V_i corresponding to each population member X_i in the current generation through mutation. The most common mutation strategies create donor vectors using randomly chosen individuals from the current population $X_G = [X_1, X_2, ..., X_{NP}]$ or the best individuals from the current population. The standard naming convention used to represent the various mutation strategies is DE/x/y, where x describes which individual is to be perturbed, and y is the number of difference vectors considered for the perturbation of the individual described by x. For example, some of the commonly used mutation strategies are listed below [11]:

DE/rand/1:

$$V_i = X_{r_1} + F(X_{r_2} - X_{r_3}).$$

DE/rand/2:

$$V_i = X_{r_1} + F(X_{r_2} - X_{r_3}). + F(X_{r_4} - X_{r_5}).$$

DE/best/1:

$$V_i = X_{best} + F(X_{r_1} - X_{r_2}).$$

DE/best/2:

$$V_i = X_{best} + F(X_{r_1} - X_{r_2}) + F(X_{r_3} - X_{r_4}).$$

DE/current-to-best/1:

$$V_i = X_i + F(X_{best} - X_i) + F(X_{r_1} - X_{r_2}).$$

The indices r_1, r_2, r_3, r_4 and r_5 are all mutually exclusive integers from the range $[1, NP]$, which are randomly generated for each donor vector, and they are all different from the index i.

Crossover Schemes

Through crossover, the donor vector V_i mixes its components with the individual X_i to form a trial vector U_i. The DE family of algorithms uses mainly two different crossover schemes: binomial and exponential. Binomial crossover combines coordinates of X_i with coordinates of V_i according to the following formula:

$$U_j = \begin{cases} V_j \; if \; R_j \leq CR \; or \; j = I \\ X_j \; if \; R_j > \; CR \; and \; j \neq I \end{cases}$$

where I is a number randomly chosen from the set {1, 2, . . . , D}, D is the dimensionality of the problem, $R_1, R_2, ..., R_D$ are random variables uniformly distributed in $(0, 1)$ and $CR \in [0, 1]$ is the chosen crossover rate. In exponential crossover, an integer r is randomly chosen from $[1, D]$ which is the starting point for the exponential crossover. Coordinates of the trial vector after r depend on a series of Bernoulli experiments with probability CR. The coordinates of the donor vector will be transferred to the trial vector until the Bernoulli experiment is unsuccessful for the first time or the crossover length has reached $D - 1$. The remaining coordinates are transferred from the target vector X_i.

Selection

The selection phase involves the trial vector U_i and target vector X_i competing for a spot in the next generations population. If $f(U_i) < f(X_i)$ replace X_i with U_i in the next generation. Otherwise, add X_i to the next generations population.

Differential Evolution Algorithm

1. Select a crossover rate $CR \in [0, 1]$, the scale factor F and the population size NP.
2. Initialize a population of candidate solutions $X = (x_1, x_2, ..., x_D)$ with random positions in the search-space.
3. Repeat the following until the termination criteria is met:
   - 3.1. For each candidate solution X_i
     - 3.1.1. Create donor vector V_i using chosen mutation operator.
     - 3.1.2. Apply crossover operator to X_i and V_i to create trial vector U_i.
     - 3.1.3. If $f(U_i) < f(X_i)$ replace X_i with U_i in the next generation. Otherwise, add X_i to the next generations population.
   - 3.2. Replace current generation with the next generation.
4. Select the candidate solution from the population that has the best fitness value.

2.2.3 Adaptive and Self-adaptive Differential Evolution algorithm

Adaptive and Self-adaptive EAs are focused on adjusting the control parameters and learning strategies used by an algorithm during a populations evolution. Many diferrent adaptive and self-adaptive systems have been proposed for use in DEAs, since good adaptive control can enhance the robustness of algorthims and reduce the time requrired to find the correct control parameters. Additionally, its possible to improve an algorthims balance between exploration

and explotation, by changing its focus at diferrent stages of the evoltionary process. In adaptive systems, feedback from the evolutionary search is used to dynamically change the control parameters and learning strategies. For instance, Rechenberg 1/5-th rule [12]. Whilst self-adaptive systems adjust the parameters and learning strategies automatically during runtime. For instance, Qin and Suganthan [17] proposed a Self-adaptive DE (SADE) algorithm that attempts to automatically adapt the learning strategies used during evolution by comparing how successful the learning strategies used are relative to each other. The SA system proposed probabilistically selects one out of several candidate learning strategies for each. For instance, we could define two mutation strategies as candidates $rand/1$ and and $current-to-best/2$. The $rand/1$ strategy has proven a good strategy for maintaining diversity within a population, and the $current-to-best/2$ strategy shows good convergence properties. We define the probability of applying strategy $rand/1$ to each individual in the current population as p_1, and therefore the probability of applying $current-to-best/2$ is $p_2 = 1 - p_1$. Initially, these probabilities are set to be equal such that $p_1 = p_2 = 0.5$. A vector of size NP is randomly generated with uniform distribution in the range $[0, 1]$ for each element. If the value of the j^{th} element of the vector is smaller than or equal to p_1, then mutation strategy $rand/1$ will be applied to the j^{th} individual in the current population. Otherwise, the mutation strategy $best/2$ will be applied to the j^{th} individual in the current population. After creating all donor vectors, the crossover operator is used to create a trial vector and selection occurs as usual. The number of trial vectors which successfully enter the next generation due to the $rand/1$ strategy and the $current-to-best/2$ strategy are recorded as ns_1 and ns_2, respectively. Similarly, the numbers of trial vectors created by the mutation strategies $rand/1$ and $current-to-best/2$ which are not featured in the new population are recorded as nf_1 and nf_2, respectively. Those four numbers are accumulated within a specified number of generations, called the learning period. Then the probabilities p_1 and p_2 are automatically updated as:

$$p_1 = \frac{\frac{ns_1}{ns_1+nf_1}}{\frac{ns_1}{ns_1+nf_1} + \frac{ns_2}{ns_2+nf_2}}$$

and

$$p_2 = 1 - p_1.$$

The above expression represents the ratio for the success rate of the trial vectors generated by the $rand/1$ strategy to the success rate of the trial vectors generated by the $current - to - best/2$ strategy during the specified number of generations. The probability of applying the two strategies is updated, after the learning period, and the four counters are reset to zero. Therefore, the probability of a mutation strategy being selected becomes dependent on the success rates of the trial vectors generated by the various strategies. Where the more successful a mutation strategy is relative to other candidate strategies, the more likely it is to be selected during the next learning period. Hence, if trial vectors created through exploration ($rand/1$ generated) are much less successful than those created through exploitation ($current - to - best/2$ generated) throughout the learning period, then its much less likely that a given individual will explore in the next learning period.

2.3 Test Functions for Optimization

The task of any global optimization algorithm is to find the global optimum solution, this task can be defined mathematically as:

$$\underset{x}{Minimize}\ f(x)$$

where $f(x)$ is the problem to be solved. Piotrowski [14] found that the best performance of an MLP trained by various algorithms was obtained by a DE algorithm. However, this DE variant did not show very good performance for on test functions. However, it should be noted that the author only used 7 test functions: Ackley, Eggholder, Griewank, RANA, Rosenbrock, Schweffel, and Whitley. Testing the optimization algorithms on a larger set of test functions would potentially highlight which type of problems the algorithms performed best on. Test functions have diverse properties and features. In some functions such as Easom and Powell, the area that contains that global minima is very small, when compared to the entire search space. In functions such as Perm and Schaffer, the global minimum is very close to local minima [7]. Due to the diverse properties of different functions, to efficiently tackle optimizations problems it is important to understand;

- What aspects of the function search space makes the optimization process difficult?

- What algorithm is most effective for searching particular types of function search spaces.

Therefore, benchmark functions are classified in terms of features such as modality, basins, valleys, separability and dimensionality. So that when algorithms are tested on these benchmark functions, the type of problems where they performs better compared to other algorithms can be identified. This allows researchers to characterize the type of problems for which an algorithm is suitable. Hence, test functions are important to validate and compare the performance of optimization algorithms. The use of many different test functions has been reported in relevant literature; however, there is no standard set of benchmark functions. Ideally,a standard set of benchmark functions would include functions which have diverse properties so that they can fairly (without bias towards certain features/functions) test new algorithms. Some of the most significant features of benchmark functions are discussed in the following section.

Basins and Valleys

Basins are large areas surrounded by a relatively steep decline. Optimization algorithms can be easily attracted to such regions and become trapped in them since there is little information to direct the search process towards the minimum. Similarly, a valley is a narrow area of little change surrounded by regions of a steep descent. As with basins, optimization algorithms are attracted to these regions. And the search process of an algorithm may be slowed down in these narrow areas.

Dimensionality

The difficulty of a problem generally increases with its dimensionality. This is because as the number of parameters increases, the search space increases exponentially.

Modality

The number of peaks in the function landscape corresponds to the modality of a function. If algorithms encounters these peaks during a search process,they may become trapped in them. This prevents the algorithm from finding the global optimal solutions. A unimodal function has only one minimum. A multi-modal function has more than one local optimum. These functions are used to test the ability of an algorithm to escape from any local minimum. If the exploration process of an algorithm is poorly designed, or the algorithm is gradient-based, the algorithm will become trapped in local minima easily and be unable to search the functions search space effectively. Multi-modal functions with a large number of local minima are very challenging problems for many algorithms. Flat surfaces in functions also create a challenge for algorithms, because the flat surfaces do not give the algorithms any information to direct the search process towards the minima.

Separability

A function of p variables is called separable, if it can written as a sum of p functions of just one variable . On the other hand, a function is called non separable, if its variables show inter-relation among themselves or are not independent. If the objective function variables are independent of each other, then the objective functions can be decomposed into sub-objective functions. Hence, each sub-objective function can be treated as a 1 dimensional problem, with the other dimensions being treated as a constant. Therefore, the objective function can be expressed as

$$f(x_1, x_2, ..., x_p) = \sum_{i=1}^{p} f_i(x_i).$$

Separable functions are generally relatively easy to solve, compared to inseparable functions,because each of the function variables is independent of the others. If all the variables are independent, then a sequence of n independent optimization processes can be performed. Therefore, each design variable can be optimized independently. Hence, the separability of a function can be used as an indication of how difficult it is to solve.

2.4 Recent Literature - Training ANNs using EAs

This section contains a short but detailed review of literature published within the last 5 years which discusses the performance of various EAs when used to train the weights of ANNs. The purpose of this review is to understand the current state of research involving training ANNs using EAs, in order to design a EA which rivals or performs better than current algorithms (when used to train ANNs). Therefore, this review aims to identify promising evolutionary systems for ANN training.

Sarangi et al. [18] hybridized Differential Evolution (DE) and Backpropagation (BP) algorithms, referring to the result as the DE-BP algorithm. The authors discusses how the global search abilities of EAs can improve the performance of ANNs, at the expense of a very high computational cost for training the network. Therefore, they propose the DE-BP algorithm in which DE is utilized to locate some smaller and better search spaces in the overall solution space, then BP is utilized to locate the optimal solution in the chosen search spaces. Therefore, the hybrid algorithm should have a faster convergence speed than a DE algorithm and a smaller chance to get stuck in a poor local minima than a BP algorithm. The performance of the DE-BP algorithm is investigated using seven different datasets. The experimental results are compared with the results of implementing BP, DE and a GA-BP algorithm. Each of the training algorithms were executed ten times for each dataset, and the average accuracy of the classifiers was calculated. For most of the datasets, the classification rate of the DE-BP algorithm outperformed the other algorithms. The DE-BP algorithm achieved higher classifications rates whilst requiring significantly less training time than the DE and GA algorithms. The results of this experiment suggest that the hybridized DE-BP algorithm is more suitable to train ANNs than DE and GA-BP, however BP is still a viable alternative despite its poor solutions due to the short amount of time it requires to run.

Wang et al. [20] used a similar hybridized DE-BP algorithm to Sarangi et al. to train an ANN, however they used an improved DE algorithm. The authors choose DE to optimize the weights of the ANN since it was performing better than alternative evolutionary training algorithms such as GA in other research involving the training of ANN coefficients. DE algorithms are

amongst the most popular EAs due to them being easy to implement, the short time it takes for them to converge, and their robustness. Wang implemented an Adaptive DE (ADE) algorithm which uses an adaptive mutation factor where F is changed as the algorithm iterates. In DE algorithms if F is too large the global optimal solution acquired by the DE may provide low accuracy. However, if F is too small the diversity of the population is not guaranteed. The proposed adaptive mutation scheme chooses F at each iteration using the following formula:

$$F = F_{min} + (F_{max} - F_{min}) \times e^{1-\frac{GenM}{GenM-G+1}}$$

where F_{min} denotes the minimum value of the mutation factor, F_{max} denotes the maximum value, $GenM$ is the maximum iteration number and G is the present iteration number. Therefore, F is formulated such that it tends to F_{min} as the iteration count tends towards it maximum. Therefore, during the early iterations F is large and can guarantee the diversity of the population. And during the later iterations, F is smaller and can therefore retain good individuals. Note that whilst the authors refer to this as an adaptive strategy, it is actually deterministic as the control parameter is altered without taking into account any feedback from the evolutionary search. The authors investigated the performance of the ADE-BP algorithm using two datasets which have non-linear features such as fluctuation and cyclic tendency. The structure of the ANNs was designed using the conclusions of previous research, and the results are compared to that of alternative optimisation algorithms. The ADE-BP algorithm significantly outperformed other algorithms tested in terms of reducing the root-mean-square error and mean absolute error, including the DE-BP and GA-BP algorithms discussed by Sarangi et al. The results of this experiment suggest that introducing adaptive parameters to the search operators of EAs can further improve their ability to train ANNs.

Piotrowski [14] discussed why DE is a popular tool for training ANN coefficients, despite having significant issues. Piotrowski suggests DEs popularity is due to its simplicity in terms of understanding, encoding and implementation, and its good performance. However, despite its popularity the standard DE algorithm has significant problems. For instance, the choice of control parameters is a challenging task which can significantly affect the performance of the

algorithm. Additionally, the standard DE algorithm can often fail to achieve a suitable balance between the exploration of a solution space and the exploitation of good solutions. Therefore, its possible that a DE algorithm will either be too slow to converge or converge prematurely. In this study the behaviour of eight DE variants is tested on benchmarks functions and applied to train ANNs. The results of this experiment signify that that the proper balance between the exploration and the exploitation capabilities of DE algorithms is crucial for them to successfully train ANNs. Algorithms with poorer exploitation capabilities showed very little or slow improvements during the later iterations of a run. Algorithms with poorer exploration capabilities found less optimal solutions. DE algorithm with Global and Local neighbourhood-based (DEGL) mutation operators were the best performing algorithms in terms of MLP training. DEGL algorithms are modified version of the DE/current-to-best/1 which favours exploitation over exploration, since all the donor vectors created are attracted to the same best position in the current population. The DEGL uses a different mutation strategy to effectively balance exploration and exploitation during the search process. The mutation strategy uses a combination of two different mutation models. A local neighbourhood model, where each individual is mutated using the best position found in a small neighbourhood of it and not in the entire population. And the global mutation model, where each individual is mutated using the best position found in the entire population. The global mutation model allows information to spread quickly amongst individuals, whilst the local mutation model only allows individuals to share information with their closest neighbourhood. By using a combination of these mutation operators its exploration and exploitation capabilities are well balanced. The current population $P = [X_1, X_2, ...X_{NP}]$ is organized on a ring topology, such that candidate solutions X_{NP} and X_2 are the two immediate neighbours of vector X_1. For every position X_i, we define a neighbourhood of radius k, where $0 < k \leq \frac{NP-1}{2}$ and $k \in \mathbb{Z}$, which consist of individuals $X_{i-k}, X_{i-k+1}, ...X_{i+k-1}, X_{i+k}$. A local donor vector is defined as

$$L_i = X_i + \alpha(X_{best_i} - X_i) + \beta(X_p - X_q)$$

and a global donor vector is defined as

$$G_i = X_i + \alpha(X_{best} - X_i) + \beta(X_{r_1} - X_{r_2}),$$

where X_{best_i} is the the best positions in the neighbourhood of X_i, $p, q \in [i-k, i+k]$ with $p \neq q \neq i$, and α, β are the scaling factors. These donor vectors are combined using a scalar weight $w \in (0, 1)$ to form the actual donor vector

$$V_i = w.G_i + (1-w)L_i.$$

The results also showed that the performance of DE algorithms used to train MLP is significantly affected by the population size chosen. The tested self-adapting and distributed DE algorithms required small population sizes. Whilst the DEGL variants performed better when the population sizes were larger. Hence, the results of this experiment highlight the significance of choosing a suitable population size for specific DEAs. Additionally, like the results of the previously discussed studies, these results also highlight the significant benefit of balancing a EAs local and global search capabilities.

The focus of research concerning the training of ANN coefficients is currently on designing algorithms which are well balanced in terms of exploration and exploitation capabilities. It is for this reason designing hybrid evolutionary and gradient-based optimisation algorithms is currently a popular research direction. EAs are well suited to global optimisation tasks, however gradient based algorithms are better suited to local search. Therefore, the combination of these two types of algorithms has the potential to create an algorithm which is well balanced in terms of its exploration and exploitation capabilities. Currently the most popular EA to train ANNs is DE. Research is currently focused on improving this algorithm using hybridization and adaptive parameters. Future research should focus on further improving the ability of the DE algorithm to train ANNs by further optimizing the balance between its exploration and exploitation capabilities, by introducing various adaptive and self-adaptive parameters, and hybridizing it with suitable local search methods.

Chapter 3

Designing and Implementing a Differential Evolution Algorithm

In this section the Self-adaptive Differential Evolution with Global and Local mutation (SADEGL) is designed and implemented. As the name suggest its mutation operators are inspired by the mutation strategy featured in [14]. However, instead of using the combination of the local and global donor vector to form a single donor vector, the local and global donor vectors are the result of two different mutation operators. The algorithm is designed to adapt according to the relative success of each operator in previous iterations using a self-adaptive scheme similar to that proposed in [17]. Additionally, a self-adaptive scheme is implemented to control the crossover rate. [14] found that higher population sizes performed better for DEGL algorithms, and it is common to scale the population size according to the dimensionality of the problem being solved. Hence, initially $NP = 10 \times D$ [15].

3.1 Self-adaptive Mutation Strategy

The DE algorithm features a mutation strategy which randomly selects one of three mutation operators for each individual in the population. Initially each operator has equal probability of being selected. Each time a trial vector created using a mutation strategy is successful its

success is noted, as are its failures. After a learning period of LP_M iterations, the success and failures of each mutation strategy are summed and used to calculate the probability of the strategy being selected during the next learning period. The success rate for each mutation strategy M_i during a single learning period is calculated as

$$MSR_i = \frac{MS_i}{MS_i + MF_i}$$

where MS_i is the amount of times trial vectors created using the mutation operator M_i were successful, and MF_i is the amount of times they were not selected. The probability of a mutation strategy being selected during the next learning period is

$$MP_i = \frac{MSR_i}{MSR_1 + MSR_2 + MSR_3}.$$

Therefore, the success of a mutation operator relative to the success of the other two operators determines the probability of it being selected in the next learning period.

If a mutation operator is mostly unsuccessful during a certain learning period, the probability of it being selected will be low. However, this mutation operator may be useful again in later iterations of the algorithm. But as $MP_i \to 0$, the success rate MSR_i becomes less statistically significant since the selection rate of the mutation operator should decrease and the sample size will be much smaller. Hence, if an operators selection probability is too low, it may fail to be recognized as a good operator for the current learning period. Additionally, if the probability of a mutation operator falls to zero, its impossible for that operator to be used again. Hence, this algorithm features a mechanism to prevent this issue. If any probability is less than the user defined control parameter $\delta < \frac{1}{3}$, each probability is reset to its initial probability.
As $\delta \to 1/3$ the adaptiveness of the mutation strategy decreases, since the strategy will more leniently reset the probabilities of the mutation operators. As $\delta \to 0$ the adaptiveness of the mutation strategy becomes less consistent, since there will be small sample size to calculate the success rates of operators with low chance of selection. Therefore, δ should be a relatively close to 0 compared to the initial probability of each mutation operator, hence in this scenario

$\delta = \frac{1}{3} \times \alpha$, where $0 < \alpha < 1$. For the remainder of this paper $\alpha = \frac{1}{10}$. Since this value for δ performed well during brief testing of possible values. If no trial vector is successful during the last learning period, the mutation strategy gains no information to direct the search further. The success of each mutation operator depends heavily on the current population, therefore the usefulness of each operator can change significantly during each learning period. Hence, it is inappropriate to use the previous operator probabilities if no information is gained, and therefore hence the probability of each operator being selected is set to equal.

3.2 Chosen Mutation Operators

The algorithm uses three different mutation operators current-to-best/1, current-to-local-best/1 and current-to-rand/1. Each of the choose mutation strategies have a distinct purpose. The current-to-best/1 strategy exploits the region surrounding the best found solution in the current population. The current-to-local-best/1 strategy explores the search space by exploiting the best solution found within a neighbourhood of K individuals, where the individuals in the population are ordered using ring topology. The purpose of the current-to-rand/1 strategy is purely to diversify the population and avoid stagnation.

3.3 Self-adaptive Crossover Rate Strategy

The algorithm features a CR strategy similar to its mutation strategy. One of two crossover operators is randomly selected for each individual solution. Initially each operator has a chosen probability of being selected. Each time a trial vector created using a crossover operator is successful it success is noted, as are its failures. Additionally, each time a trial vector U, created using C_i, replaces an individual X,the improvement of that solutions fitness $F(X) - F(U)$ is calculated and added to a list of improvements , CI_i. After a learning period of LP_C iterations, the success and failures of each crossover operator are summed and used to calculate the probability of the operator being selected during the next learning period. The success rate for

each crossover operator C_i for a single learning period is calculated as

$$CSR_i = \frac{CS_i}{CS_i + CF_i}$$

, where CS_i is the amount of times trial vectors created using the crossover operator C_i were successful and CF_i is the amount of times they were not selected. The success rates of each operator are then multiplied by the mean improvement of the crossover operator, such that

$$Cweight_i = CSR_i \times mean(CI_i).$$

The probability of a crossover operator being selected during the next learning period is

$$CP_i = \frac{Cweight_i}{Cweight_1 + Cweight_2}.$$

Therefore, the success of a crossover operator relative to the success of the other operator, determines the probability of it being selected during the next learning period. Like the mutation strategy, the crossover strategy features a mechanism to prevent operators from disappearing from the selection process. Unlike the mutation strategy this mechanism doesn't set the probabilities of each operator equal if any probability falls below a pre defined value, instead each operators probability is bounded in the range $[CPmin, 1 - CPmin]$. This is because unlike the successfulness of mutation operators, the successfulness of each crossover operator is expected to change less significantly in a single learning period [21]. Therefore, the minimum tolerable statistical significance of $Cweight_i$ should be relatively low, since the successfulness of an operator is not expected to change suddenly. This is also why if no successful trial vectors were created during the last learning period, the probabilities of each operator are not updated.

3.4 Chosen Crossover Operators

The algorithm uses two different crossover operators. When using either operator, Crossover rates values are generated for each individual in the population using the following formula

$$CR = \mathcal{N}(CRm, 0.1)$$

, hence crossover values are normally distributed with mean C_m and standard deviation 0.1. However, each crossover strategy C_i uses a different value for CR_m. For C_1, $CRm = 1$ and for C_2, $CRm = 0$. Additionally, the crossover rate for each individual is bounded between 0 and 1. The values are for CR_m are at the extreme ends of the possible range, because small crossover rate values will allow the algorithm to consistently make small improvements, and high crossover rate values will allow the algorithm to make fewer but more significant improvements. Note that this makes using just the success rate of crossover values a poor adaptation scheme, since there is a natural bias to low crossover rate values, since they are more consistently successful, but the improvements are usually insignificant. Hence, the mean improvement made by each operator is used to scale the success rate of each operator appropriately, and it is these scaled success rates used to determine the probabilities of each operator in the next learning period.

3.5 Learning Period Size

Both the mutation and crossover strategy adapt according to values generated in the previous learning period. The longer the learning period, the larger the sample size used to generate these values will be. Therefore, the information gained during the learning period will be more statically significant. However, since the successfulness of strategies can change during the learning period, the relevance and usefulness of information gained decays as the algorithm continues to iterate. And therefore the larger the learning period, the less relevant and useful information gained during the earlier iterations of the learning period will be. Hence, the size of the learning periods must be appropriately chosen so that the adaptive strategies are

determined based on a sample size large enough to provide reliable information, but small enough to provide relevant information. After briefly testing various learning period in range $[1, 30]$, a learning period of 5 iterations was found most suitable for this algorithm. Hence, for the rest of this paper $LP_C = LP_M = 5$

3.6 Scale factor

The scale factor f is randomly chosen for each individual. The random values chosen are normally distributed with mean 0.5 and standard deviation 0.3. These values are bounded between 0 and 2. The purpose of this randomization is to make the mutation process more diverse. Making the scale factor self-adaptive would further increase the complexity of the algorithm, and potentially make it worse by interfering with the mechanisms of the other self-adaptive strategies.

3.7 Differential Evolution Pseudocode

Pseudocode for the SADGELP algorithm is represented by Algorithm 1. Pseudocode for the mutation operators is represented by Algorithms 2,3 and 4. The algorithm was implemented in Matlab, and the code is provided in the appendices.

Algorithm 1 Minimize function F

Input: D dimensional function F
Output: The best found candidate solution.
Initialize control parameters

Randomly generate and evaluate an initial population

for np=1 to NP **do**
  $X(np) \leftarrow$ Randomly generate a D dimensional indivdual
  $Y(np) \leftarrow F(X(np))$
end for
$g_{best} \leftarrow \mathbb{Z} \in [1, NP]$ s.t $Y(g_{best}) \equiv min(Y)$

Intiliaze storage variables and mutation probablities

for each mutation operator M_i **do**
  $MS_i, MF_i, MSR_i \leftarrow 0$
  $MP_i \leftarrow \frac{1}{3}$
end for
for each crossover operator CR_j **do**
  $CI_j \leftarrow []$
  $CS_j, CF_j, Cweight_j \leftarrow 0$
end for
$CP_1, CRm1 \leftarrow 0.9, 1$
$CP_2, CRm2 \leftarrow 0.1, 0$
$iter \leftarrow 0$

Main Loop of Algorithm

while termination criteria is not met **do**
  $iter \leftarrow iter + 1$
  for np=1 to NP **do**
    $F \leftarrow \mathcal{N}(0.5, 0.3)$ truncated within range $[0, 2]$
    Randomly select a mutation operator M_i using the probablities MP_1, MP_2 and MP_3
    Randomly select a crossover operator C_j using the probabilities CP_1 and CP_2
    Create donor vector V using M_i
    Apply binomial crossover using C_j and store the result as trial vector U
    if $F(U) < F(X(np))$ **then**
      $X(np) \leftarrow U$
      $Y(np) \leftarrow F(U)$
      Add $F(X(np)) - F(U)$ to the list CI_j
      $MS_i \leftarrow MS_i + 1$
      $CS_j \leftarrow CS_j + 1$
    else
      $MF_i \leftarrow MF_i + 1$
      $CF_j \leftarrow CF_j + 1$
    end if
  end for

39: $g_{best} \leftarrow \mathbb{Z} \in [1, NP]$ s.t $Y(g_{best}) \equiv min(Y)$

Determine new probabilities for crossover operators

40: **if** mod(iter,LP_C)=0 **then**
41: **for** each crossover operator C_j **do**
42: **if** mutation operator has been selected atleast once **then**
43: **if** mutation operator has been successful atleast once **then**
44: $Cweight_j \leftarrow \frac{CS_j}{CS_j+CF_j} \times \bar{X}(CI_j)$
45: **else**
46: $Cweight_j \leftarrow 0$
47: **end if**
48: $CS_j, CF_j \leftarrow 0$
49: **end if**
50: **end for**
51: **if** the sum(CSR) is not equal to 0 **then**
52: **for** each crossover operator C_j **do**
53: $CP_j \leftarrow \frac{Cweight_j}{sum(Cweight)}$ truncated within range $[CRmin, 1 - CRmin]$
54: **end for**
55: **end if**
56: $CI_1, CI_2 \leftarrow []$
57: **end if**

Determine new probabilities for mutation operators

58: **if** mod(iter,LP_M)=0 **then**
59: **for** each mutation operator M_i **do**
60: **if** mutation operator has been selected atleast once **then**
61: $MSR_i \leftarrow \frac{MS_i}{MS_i+MF_i}$
62: $MS_i, MF_i \leftarrow 0$
63: **end if**
64: **end for**
65: **if** the sum(MSR) is not equal to 0 **then**
66: **for** each mutation operator M_i **do**
67: $MP_i \leftarrow \frac{MSRi}{sum(MSR)}$
68: **end for**
69: **else**
70: $MP_1, MP_2, MP_3 \leftarrow \frac{1}{3}$
71: **end if**
72: **if** either MP_1, MP_2 or $MP_3 < \delta$ **then**
73: $MP_1, MP_2, MP_3 \leftarrow \frac{1}{3}$
74: **end if**
75: **end if**
76: **end while**

Return the best found canditate solution

77: **return** $X(g_{best})$

Algorithm 2 M_1 - Create current-to-local-best/1 donor vector

Input: X, NP, F, np, K, Y
Output: Donor vector V
$neighbourhoodList \leftarrow []$
for k=0 to 2K **do**
    $neighbourhoodList(k) \leftarrow mod(np - K - 1 + k, NP) + 1$
end for
$l_{best} \leftarrow \mathbb{Z} \in neighbourhoodList$ s.t $Y(l_{best}) \equiv min(Y)$ in the neighbourhood of np
$j, k \leftarrow$ Unique randomly selected integers $\in$ neighbourhood List where $j, k \neq np$
$V \leftarrow X(np) + F[X(l_{best}) - X(np) + X(j) - X(k)]$
return V

Algorithm 3 M_2 - Create current-to-best/1 donor vector

Input: X, NP, F, np, g_{best}
Output: Donor vector V
$j, k \leftarrow$ Unique randomly selected integers in range $[1, NP]$ where $j, k \neq np$
$V \leftarrow X(np) + F[X(g_{best}) - X(np) + X(j) - X(k)]$
return V

Algorithm 4 M_3 - Create current-to-rand/1 donor vector

Input: X, NP, F, np
Output: Donor vector V
$j, k, q \leftarrow$ Unique randomly selected integers in range $[1, NP]$ where $j, k, q \neq np$
$V \leftarrow X(np) + F[X(q) - X(np) + X(j) - X(k)]$
return V

Chapter 4

Benchmarking the Differential Evolution Algorithm

In this section the DE algorithm designed in the previous chapter is benchmarked using the CEC 2013 Competition Rules. The CEC'13 test suite includes 28 benchmark functions with diverse properties. The main features of each function are briefly described in [9]. For each function the search range is limited to $[-100, 100]^D$ where D is the dimensionality of the problem. Hence, the DE algorithm uniformly randomly generates an initial population within this range and truncates all trial vectors within the range. The algorithm stops when either the maximum number of functions evaluations has been reached or the error value of an individual is less than 10^{-8}, where the error value is the difference between the evaluation of the best found candidate solution and the global optimum. The maximum number of function evaluations is $maxFE = 10^4 \times D$. The algorithm is run 51 times for each function and for each run the smallest error value found after $(0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0) \times maxFE$ is recorded. The best, worst, mean, median and standard deviation values of the error values of the 51 runs for each function are calculated and presented in this section. The algorithm is benchmarked for three different dimensions $D = 10, 30, 50$.

Additionally, using the CEC'13 guidelines the algorithm complexity is calculated for each dimension to demonstrate the relationship between the algorithms computational complexity and

dimensionality. The computational complexity is evaluated using the [9] measurement. The algorithm was implemented in Matlab R2017a and run on a desktop computer with an Intel Core i5-4590 @ 3.30GHz and 8.00 GBs of RAM, using Windows 10 operating system.

4.1 Chosen control parameters

Participants of the CEC'13 competition are requested not to search for the best distinct set of parameters for each function/dimension. Hence, the control parameters are chosen to be the same for all dimensions and functions. The control parameters chosen are shown in Table 4.1.

Table 4.1: The control parameters chosen for the CEC'13 runs

Control Parameter	***Value***
NP	10D
K	$\frac{NP}{10}$
δ	$\frac{1}{30}$
$CPmin$	$\frac{1}{20}$
LP_C	5
LP_M	5

4.2 Results

The values of the function errors are shown in Table 4.2 for $D = 10$, Table 4.3 for $D = 30$, and Table 4.4 for $D = 50$. The computational complexity values obtained are shown in Table 4.5. The tables are presented in the format requested in the CEC'2013 competition guidelines. Hence, errors smaller than 10^{-8} are set to zero. The results in these tables provide little information about the performance of the algorithm by themselves. To understand the performance of the algorithm, we must understand how difficulty each function is to optimize. Therefore, the performance of the algorithm is analysed by comparing the results to the results of other algorithms on these problems.

The algorithms presented in the CEC'2013 results comparison paper [10] were ranked by comparing the best solutions for each function of each algorithm. The paper features 21 high performance algorithms, hence to determine how well the proposed SADEGL algorithm performs, comparisons are initially made to the results of the middle ranking algorithm (10^{th} ranked) 'b6e6rl' [19]. Note that the intended use of SADEGL is to train ANNs, and the results found by the algorithm will be fine tuned using BP, hence finding the most optimal solutions is not necessary at this stage, the algorithm should however consistently find near optimal solutions.

For each dimension the best, worst, median, mean and standard deviations for SADEGEL and b6e6rl are compared and ranked relatively. The algorithm with the minimum value for each statistic is noted. The standard deviation alone provides no useful information to compare the algorithms, an algorithm could have a small standard deviation for a function but a much worse mean than the other algorithm, hence the algorithm could just be consistently performing poorly on that function. Hence, the standard deviations values are not compared and discussed. The comparisons for each function and dimension are summarized in Table 4.6.

For the 10 dimensional functions the best values were most commonly found by both functions, however 7 were found by SADEGL and 11 by b6e6rl. The amount of worst values found was similar for both algorithms, with b6e6rl providing 13 of the worst values and SADEGL providing 12. The amount of times each algorithm had the smallest median and mean is similar.

For the 30 dimensional functions the best values were most commonly found by SADEGL, with it finding 14 of the best values and b6e6rl finding 9. The worst values were most commonly found by b6e6rl with it finding 17 of the worst values and b6e6rl finding 8. The amount of times each algorithm had the smallest median and mean is very similar.

For the 50 dimensional functions the best values were most commonly found by b6e6rl, with it finding 14 of the best values and SADEGL finding 10. The amount of times each algorithm had the worst value for a function was equal. And the amount of times each algorithm had the smallest median and mean is very similar.

From the results of the best value comparisons we can infer that SADEGL is atleast competi-

tive with b6e6rl finding better or equal best values approximately 60% of the time for the $10D$ problems, 67% for the $30D$ problems, and 50% for the $50D$ problems. From the results of the worst, mean, and median value comparisons we can also infer that SADEGL is able to consistently find good results. Since regardless of dimension the amount of times either algorithm has the worst value, or smallest median/mean across the 51 runs is close to equal. However, in the 30 dimensional case b6e6rl finds worse values than SADEGL 68% of the time. Suggesting either that the control parameters used for SADEGL are most appropriate when $D = 30$, or control parameters for *b6e6rl* are not appropriate for when $D = 30$.

The consistency of stochastic optimization algorithms can be analysed effectively using the Interquartile Range (IQR) **??** , hence the IQR is calculated for each each function and for each dimension. The IQRs are shown in Table 4.7. Apart from a few exceptions, the interquartile range of the runs increases as the dimensionality increases. The majority of functions have relatively small interquartile range for all dimensions, the two noticeable exceptions are functions 2 and 3. These functions are mostly flat which makes its difficult for a function to navigate the surface space. The relatively small interquartile ranges for the majority of functions suggest that the algorithm can consistently find good solutions to problems. This demonstrates that SADEGL algorithm has suitable balance between exploration and exploration.

From the computational complexity values we can clearly see that the increase in complexity is linear with the increase in dimensionality. However, by testing the algorithm with $NP = 150$ its clear that majority of the computation complexity is a result of $NP = 10D$ increasing linearly as the dimensionality increases. Therefore, if in future tests it is shown that large NP values do not produce significantly better results than smaller NP values, smaller values of NP should be used to reduce the computational complexity of the algorithm. The reason for this increase in computation complexity is likely to be the use of the local operator, since as $NP \rightarrow \inf$, $K \rightarrow \inf$, meaning the algorithm must search a wider range of values for the local minimum. Additionally note that the algorithm is computationally less expensive than *b6e6rl* for each dimension when $NP = 150$, however when $NP = 10D$ it is only less expensive when $D = 10$, it is slightly more expensive when $D = 30$ and significantly more expensive when $D = 50$.

Table 4.2: Value of Function Errors for $D = 10$

Problem	**Best**	**Worst**	**Median**	**Mean**	**Std**
1	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
2	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
3	0.00E+00	6.32E+00	0.00E+00	2.51E-01	1.24E+00
4	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
5	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
6	0.00E+00	9.81E+00	9.81E+00	5.58E+00	4.91E+00
7	2.97E-06	1.11E+00	1.25E-03	6.52E-02	2.08E-01
8	2.02E+01	2.05E+01	2.04E+01	2.04E+01	6.93E-02
9	0.00E+00	5.43E+00	2.22E+00	2.38E+00	1.55E+00
10	0.00E+00	4.43E-02	9.86E-03	1.23E-02	1.17E-02
11	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
12	9.95E-01	1.39E+01	4.93E+00	5.14E+00	1.95E+00
13	0.00E+00	1.56E+01	5.30E+00	5.76E+00	3.21E+00
14	1.19E-02	8.02E-01	3.45E-01	3.56E-01	1.27E-01
15	2.61E+02	9.72E+02	5.81E+02	5.88E+02	1.53E+02
16	4.27E-01	1.30E+00	8.59E-01	8.44E-01	1.77E-01
17	1.01E+01	1.02E+01	1.02E+01	1.02E+01	1.72E-02
18	1.68E+01	2.98E+01	2.23E+01	2.25E+01	2.62E+00
19	1.97E-01	3.94E-01	3.39E-01	3.27E-01	4.61E-02
20	1.92E+00	3.53E+00	2.48E+00	2.57E+00	4.11E-01
21	4.00E+02	4.00E+02	4.00E+02	4.00E+02	0.00E+00
22	6.59E+00	3.74E+01	1.93E+01	2.03E+01	7.54E+00
23	3.10E+02	1.17E+03	7.49E+02	7.42E+02	1.91E+02
24	1.12E+02	2.14E+02	2.07E+02	2.04E+02	1.39E+01
25	1.61E+02	2.15E+02	2.00E+02	2.02E+02	7.64E+00
26	1.02E+02	2.00E+02	1.26E+02	1.53E+02	4.71E+01
27	3.00E+02	5.91E+02	3.00E+02	3.82E+02	1.01E+02
28	3.00E+02	5.42E+02	3.00E+02	3.07E+02	3.65E+01

Table 4.3: Value of Function Errors for $D = 30$

Problem	**Best**	**Worst**	**Median**	**Mean**	**Std**
1	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
2	1.57E+03	6.41E+04	1.51E+04	1.80E+04	1.46E+04
3	1.76E-02	4.98E+06	1.09E+04	3.52E+05	9.55E+05
4	1.31E-05	9.84E-03	4.77E-04	1.11E-03	1.76E-03
5	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
6	0.00E+00	2.64E+01	1.12E+00	6.05E+00	1.02E+01
7	6.86E-01	5.91E+01	1.19E+01	1.46E+01	1.14E+01
8	2.08E+01	2.10E+01	2.10E+01	2.10E+01	4.55E-02
9	2.47E+01	3.17E+01	2.89E+01	2.88E+01	1.48E+00
10	0.00E+00	1.11E-01	3.20E-02	3.46E-02	2.37E-02
11	2.75E+00	6.48E+00	4.49E+00	4.48E+00	7.89E-01
12	4.49E+01	1.02E+02	6.37E+01	6.54E+01	1.18E+01
13	5.89E+01	1.25E+02	9.94E+01	9.67E+01	1.62E+01
14	3.70E+02	7.03E+02	5.85E+02	5.85E+02	6.98E+01
15	3.83E+03	5.48E+03	4.84E+03	4.81E+03	3.33E+02
16	1.48E+00	2.38E+00	1.92E+00	1.90E+00	2.30E-01
17	3.64E+01	4.03E+01	3.82E+01	3.82E+01	7.38E-01
18	1.13E+02	1.69E+02	1.43E+02	1.42E+02	1.21E+01
19	2.03E+00	3.44E+00	2.63E+00	2.64E+00	3.13E-01
20	1.11E+01	1.25E+01	1.17E+01	1.17E+01	3.59E-01
21	2.00E+02	4.44E+02	3.00E+02	3.41E+02	9.33E+01
22	4.13E+02	9.26E+02	6.07E+02	6.17E+02	1.14E+02
23	4.44E+03	5.75E+03	5.29E+03	5.19E+03	3.21E+02
24	2.12E+02	2.76E+02	2.36E+02	2.35E+02	1.70E+01
25	2.33E+02	2.90E+02	2.58E+02	2.60E+02	1.17E+01
26	2.00E+02	3.75E+02	2.00E+02	2.16E+02	4.54E+01
27	3.58E+02	1.10E+03	9.14E+02	7.79E+02	2.53E+02
28	3.00E+02	3.00E+02	3.00E+02	3.00E+02	2.47E-13

Table 4.4: Value of function errors for $D = 50$

Problem	**Best**	**Worst**	**Median**	**Mean**	**Std**
1	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
2	4.42E+04	2.37E+05	8.94E+04	1.06E+05	4.88E+04
3	1.45E+03	3.73E+07	1.51E+06	4.93E+06	7.85E+06
4	4.84E-03	1.32E-01	2.72E-02	4.00E-02	3.12E-02
5	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
6	1.49E+01	4.91E+01	4.34E+01	4.25E+01	6.84E+00
7	6.51E+00	7.12E+01	2.88E+01	3.14E+01	1.46E+01
8	2.10E+01	2.12E+01	2.11E+01	2.11E+01	3.72E-02
9	5.39E+01	6.03E+01	5.73E+01	5.75E+01	1.59E+00
10	0.00E+00	1.08E-01	2.96E-02	3.10E-02	2.25E-02
11	3.31E+01	4.56E+01	3.69E+01	3.75E+01	2.94E+00
12	1.53E+02	2.25E+02	1.84E+02	1.86E+02	1.79E+01
13	1.84E+02	3.03E+02	2.52E+02	2.52E+02	2.75E+01
14	1.65E+03	2.30E+03	1.99E+03	1.99E+03	1.34E+02
15	9.72E+03	1.13E+04	1.05E+04	1.05E+04	3.48E+02
16	2.00E+00	3.06E+00	2.60E+00	2.58E+00	2.29E-01
17	8.38E+01	9.67E+01	9.18E+01	9.16E+01	2.80E+00
18	2.74E+02	3.60E+02	3.10E+02	3.11E+02	1.72E+01
19	5.65E+00	1.03E+01	7.70E+00	7.75E+00	9.83E-01
20	1.99E+01	2.19E+01	2.11E+01	2.11E+01	4.91E-01
21	2.00E+02	1.12E+03	8.36E+02	8.47E+02	3.30E+02
22	1.66E+03	2.93E+03	2.19E+03	2.21E+03	2.59E+02
23	9.63E+03	1.18E+04	1.11E+04	1.10E+04	5.85E+02
24	2.46E+02	3.60E+02	2.77E+02	2.86E+02	2.89E+01
25	2.97E+02	3.77E+02	3.25E+02	3.30E+02	1.97E+01
26	2.00E+02	4.49E+02	4.32E+02	3.67E+02	1.03E+02
27	7.24E+02	1.90E+03	1.66E+03	1.45E+03	3.88E+02
28	4.00E+02	3.46E+03	4.00E+02	4.60E+02	4.28E+02

Table 4.5: Computational Complexity Values for $D = 10, 30, 50$

Dimension	**SADEGL: NP=10D**	**SADEGL: NP=150**	**b6e6rl**
10	55	55	71.69
30	79	68	74.32
50	103	81	83.15

Table 4.6: The Amount of Times Each Algorithm has the Smallest Value for Each Statistic

	10D					30D					50D				
Algorthim	**Best**	**Worst**	**Median**	**Mean**	**Std**	**Best**	**Worst**	**Median**	**Mean**	**Std**	**Best**	**Worst**	**Median**	**Mean**	**Std**
SADEGL	7	12	10	13	16	14	8	12	11	9	10	13	11	13	14
b6e6rl	11	13	12	12	9	9	17	12	14	17	14	13	14	13	12
Neither	10	3	6	3	3	5	3	4	3	2	4	2	3	2	2

Table 4.7: Interquartile Range for Each Dimension

Function	**Interquartile range D10**	**Interquartile range D30**	**Interquartile range D50**
1	0	0	0
2	0	16921.9007	81170.90799
3	8.11868E-06	143947.4791	5599550.437
4	0	0.001075387	0.038227375
5	0	0	0
6	9.812422739	3.986623879	1.13687E-13
7	0.008621581	16.14992314	20.57514953
8	0.08484927	0.048921656	0.047887856
9	2.702298948	2.037383	2.369488669
10	0.01734752	0.02644738	0.027080525
11	0	0.875832101	3.973660245
12	1.789251854	15.90927073	24.98065322
13	4.34960701	23.63636022	39.91411286
14	0.133232013	108.8192656	165.9433995
15	191.1865983	482.4869741	558.3730348
16	0.262447835	0.293128814	0.249663544
17	0.023258432	0.93222181	3.693163058
18	2.4686356	14.4721881	20.23203058
19	0.05640083	0.466037865	0.998384382
20	0.523114715	0.59604981	0.755450047
21	0	143.5441417	285.743276
22	11.37878795	151.1289799	281.5400353
23	245.2006469	494.7870744	850.1116086
24	9.223922921	25.42793888	26.10996356
25	6.507018383	10.57566826	26.33630262
26	94.61846141	0.001317367	209.2966631
27	191.0625553	510.2145808	762.5400855
28	0	4.54747E-13	4.54747E-13

Chapter 5

Training ANNs using SADEGL

5.1 PROBEN1

In this chapter the SADEGL algorithm is hybridized with BP in order to train ANNs. The performance of the SADEGL-BP hybrid is benchmarked using the PROBEN1 datasets [16].

PROBEN1 contains 15 data sets from 12 different domains, all but one dataset uses real world data. The collection of datasets features both pattern classification and function approximation. Each datasets is presented in the same simple format, using an attribute representation so that it can be directly used for neural network training. Additional, there exists three different variations of each dataset. Each variation of a dataset uses a different permutation of the data for training, validation and testing data. Using the different permutations is beneficial since the datasets have a relatively small number of examples, and therefore benchmark results are less reliable, but using different splits increases the significance of the benchmarking results. In addition to the small size of the datasets, many of the datasets have missing values which have been replaced with fixed values, or handled alternatively during the preparation of the dataset.

The PROBEN1 rules are not followed in their entirety, as the purpose of this research is not to compete with the algorithms of other researchers. But to compare the performance of SADEGL-BP and BP when used to train ANNs. The PROBEN1 datasets were chosen to

evaluate the performance of the algorithms, since these are real world datasets with problems such as missing values, and insufficient amounts of data instances. And therefore they should be appropriately challenging for the sake of comparison between the two algorithms of interest.

5.2 Training Multilayer Perceptions

Multilayer Perceptions (MLP) are the class of neural networks chosen to train the PROBEN1 datasets. The universal approximation theorem for neural networks states that every continuous function that maps intervals of real numbers to some output interval of real numbers can be approximated with error less than some chosen value δ by an MLP with just one hidden layer [13]. Hence, the choice of neuron count in the hidden layer will determines how accurate the model will be. In this paper the neuron count is arbitrarily chosen to be $n = 5$, since a large quantity of datasets are being used, and the purpose of the results is to be observe the performance difference between training a network with SADEGL-BP and BP.

When training an MLP using DE algorithms the individuals in the population each represent a different combination of the weights and biases of the network. Therefore, the dimensionality used for each problem is $D = n \times (inputCount + outputCount + 1) + outputCount$, where $inputCount$ is the number of input features in the dataset, and $outputCount$ is the number of output features in the dataset.

Each dataset is divided into training, validation and testing data using the PROBEN1 specified splits. The algorithm uses the training data to train the network, the validation data to determine when to stop training the network and testing data to evaluate the performance of the network. The stopping criteria chosen for training the network using SADEGL-BP, involves stopping the algorithms evolutionary process when the algorithm is either improving too slowly or over fitting to the training data. Hence, the max amount of generations the algorithm can continue without significant reduction in the error of the validation set , $maxGenerationStagnation$, is introduced as a new control parameter. An additional control parameter $stopTol$ must also be defined as the minimum improvement required in

maxGenerationStagnation iterations of the algorithm.

5.3 Hybridizing SADEGL with BP

Once the SADEGL algorithm has trained the weights/biases of the network, they are further fine-tuned using BP. If the result of training the networking using BP, reduces the error of the testing data further, the networks weights/biases are selected. If the result of the BP training doesn't reduce the error, the weights/biases found using SADEGL are instead selected.

5.4 BP Control Parameters and Termination Criteria

BP is performed using Matlabs ANN toolbox. The criteria for BP training being terminated is as follows;

- The number of epochs has reached 1000
- The MSE is 0
- Validation has failed 6 times
- The gradient is less then $1e-5$

The learning rate used is 0.1.

5.5 SADEGL-BP Pseudocode

Pseudocode representing SADEGL-BP is shown by Algorthim . The algorithm was implemented in Matlab, and the code is provided in the appendices.

Algorithm 5 Training MLP using SADEGL-BP

Initialize control parameters for SADEGL
Initialize stop tolerance $stopTol$
Initialize max amount of stagnating generations $maxGenerationStagnation$
Initialize error function $F1$ for training data
Initialize error function $F2$ for validation data
Initialize error function $F3$ for testing data
Generate an initial population uniformly distributed $\in [-1, 1]^D$
Select the best individual in the population g_{best} and store it as $bestWB$
Evaluate $F2(g_{best})$, then store the result as $bestVal$
while termination criteria is not met **do**
    Evaluate each individual in the current population using $F1$ as the fitness function
    Select the best individual in the population g_{best} and evaluate $F2(g_{best})$
    if $F2(g_{best}) < bestVal$ **then**
        if $(F2(g_{best}) - bestVal) > stopTol$ **then**
            $GenerationStagnation \leftarrow 0$
            $bestVal \leftarrow F2(g_{best})$
        else
            $GenerationStagnation \leftarrow GenerationStagnation + 1$
        end if
        $bestWB \leftarrow g_{best}$
    else
        $GenerationStagnation \leftarrow GenerationStagnation + 1$
    end if
    if $GenerationStagnation == maxGenerationStagnation$ **then**
        Stop the evolutionary loop
    else
        Adapt the mutation and crossover rate strategies
    end if
end while
Initialize an MLP with n neurons
Set the weights and biases to $bestWB$
Train the network using BP
Test the network using the testing data, save the error as $bpError$
if $bpError \leq F3(bestWB)$ **then**
    return The weights and biases of the network trained using SADEGL and then BP
else
    return The weights and biases of the network trained using SADEGL
end if

5.6 Results: Comparisons with BP

Statistics for the MSE values over 10 runs for each dataset are shown in Table 5.1 for both BP and $SADEGL - BP$. THe average run time for both algorithms is shown in Table 5.2. SADEGL-BP finds the best solution for every single dataset, and BP finds the worst solutions for every single dataset.The median MSE for each dataset is always smaller when using SADEGL-BP than BP. Additionally, note that the IQR for SADEGLBP is smaller than the IQR for BP, for every single dataset.The worst value for SADEGL-BP is better than the best value found by BP in 39/45 datasets. The datasets where this not true are the three variations of the thyroid dataset, diabetes1, cancer3 and flare3. Each of these datasets has a common feature; the amount of examples for one of the outputs is substantially more than the amount for other outputs. This could suggest the SADEGL-BP is less consistent when there is significant class imbalance. Note that there is no noticeable relationship between the dimensionality of a problem by SADEGL-BP. Suggesting that the algorithm exploration capabilities are enabling it to effectively search high dimensional search spaces. However, in general both the average CPU time required to run SADEGL-BP and BP increases as the dimensionality of the problem increases. Naturally, the average CPU run time for SADEGL-BP is far greater than average run time for BP.

5.7 Results: SADEGL-BP

Table 5.3 shows the significance of the BP hybridization in SADEGL-BP. The average, average improvement of the datasets is 0.085289596 with standard deviation 0.076509426, suggesting the improvement made varies relatively significantly with each dataset. The average, best improvement of the datasets is 0.109263422 with standard deviation 0.092729408 which suggest the best improvement varies very relatively significantly between datasets. The average, worst improvement of the datasets is 0.001479909 with standard deviation 0.005260822 suggesting that regardless of dataset, some runs of the SADEGL-BP will not benefit from the BP hybridization. However, the hybridization is cheap to implement in terms of computational

Table 5.1: MSE values for SADEGLBP and BP

Dataset	best SADEGL-BP	best BP	Worst SADEGL-BP	Worst BP	Median SADEGL-BP	Median BP	IQR SADEGL-BP	IQR BP
building1.dt	0.013039	0.046722	0.023688	0.12478	0.021025	0.0670895	0.006963	0.031413
building2.dt	0.011885	0.034687	0.019181	0.097531	0.016589	0.0526535	0.003771	0.049797
building3.dt	0.013318	0.037368	0.019396	0.12272	0.016235	0.0562505	0.00412	0.028197
cancer1.dt	0.037295	0.21967	0.090763	0.76966	0.051431	0.60223	0.026136	0.33733
cancer2.dt	0.043452	0.36842	0.063127	1.0319	0.0492725	0.63902	0.006847	0.21785
cancer3.dt	0.042835	0.04548	0.065999	0.76206	0.053832	0.25623	0.016717	0.20915
card1.dt	0.13799	0.34336	0.22075	0.54951	0.15961	0.45349	0.05365	0.10186
card2.dt	0.13218	0.30633	0.1907	0.72263	0.15798	0.507585	0.0266	0.24026
card3.dt	0.125	0.29169	0.25695	0.6903	0.2112	0.431915	0.05966	0.23316
diabetes1.dt	0.18893	0.21992	0.22411	0.46176	0.19499	0.314685	0.01215	0.07983
diabetes2.dt	0.16527	0.30469	0.21737	0.67425	0.19828	0.3895	0.01367	0.11502
diabetes3.dt	0.18448	0.41819	0.20633	0.83932	0.198245	0.66251	0.01087	0.15556
flare1.dt	0.005703	0.062674	0.016179	0.54143	0.0108045	0.133515	0.0036891	0.229216
flare2.dt	0.0056462	0.16394	0.020638	0.98454	0.011805	0.37028	0.005415	0.21594
flare3.dt	0.0053369	0.026865	0.02753	0.46098	0.0084095	0.109955	0.0020447	0.155731
gene1.dt	0.18045	0.41456	0.20947	0.88508	0.19545	0.53253	0.01401	0.1474
gene2.dt	0.18979	0.3747	0.2036	0.6935	0.195835	0.524	0.00521	0.19061
gene3.dt	0.19058	0.34268	0.20595	0.75987	0.19645	0.497715	0.00252	0.17077
glass1.dt	0.11972	0.19157	0.14874	0.46311	0.132515	0.238385	0.00752	0.12212
glass2.dt	0.10809	0.23801	0.12733	0.84386	0.12511	0.301195	0.01133	0.07357
glass3.dt	0.11795	0.2034	0.17965	0.87819	0.14706	0.258865	0.02862	0.06484
heart1.dt	0.15302	0.41203	0.22526	0.74825	0.181845	0.53748	0.03134	0.23655
heart2.dt	0.1523	0.38485	0.21261	0.65591	0.185235	0.49747	0.02996	0.15341
heart3.dt	0.15598	0.37108	0.23905	0.90487	0.168535	0.50877	0.02506	0.20577
hearta1.dt	0.056208	0.13339	0.070184	0.24692	0.0646245	0.16761	0.006988	0.04108
hearta2.dt	0.051309	0.098796	0.076079	0.39911	0.0693205	0.200595	0.007769	0.16152
hearta3.dt	0.055084	0.114	0.069958	0.46969	0.0634365	0.197465	0.005679	0.10284
heartac1.dt	0.031445	0.10223	0.063649	0.51566	0.0389205	0.188415	0.016048	0.19475
heartac2.dt	0.045563	0.082019	0.06511	0.58866	0.0495645	0.1745	0.009406	0.18235
heartac3.dt	0.04667	0.15808	0.067027	0.6123	0.059298	0.24164	0.006651	0.1806
heartc1.dt	0.12304	0.20614	0.17604	0.77626	0.146935	0.42904	0.01534	0.12475
heartc2.dt	0.071984	0.35677	0.1112	0.92351	0.089802	0.456765	0.021572	0.13859
heartc3.dt	0.11362	0.33779	0.17805	0.65972	0.13948	0.548595	0.02097	0.14376
horse1.dt	0.14101	0.26264	0.18416	0.83255	0.16911	0.518115	0.0157	0.32652
horse2.dt	0.16824	0.30989	0.22329	0.87997	0.19056	0.49199	0.00997	0.0975
horse3.dt	0.15502	0.31396	0.18005	1.0124	0.175855	0.518115	0.0087	0.37541
mushroom1.dt	0.061638	0.33491	0.15802	0.78199	0.10114	0.53672	0.027291	0.26189
mushroom2.dt	0.056216	0.30372	0.16077	0.94163	0.119145	0.424235	0.06512	0.33558
mushroom3.dt	0.054472	0.28636	0.16533	0.67162	0.0939695	0.37584	0.040659	0.22021
soybean1.dt	0.05722	0.11677	0.11323	0.69782	0.0717435	0.2241	0.016915	0.11761
soybean2.dt	0.053438	0.17224	0.1076	0.43332	0.075084	0.260965	0.018939	0.20382
soybean3.dt	0.056795	0.15397	0.10243	0.39233	0.077126	0.21368	0.027104	0.09003
thyroid1.dt	0.049363	0.050557	0.062949	0.23542	0.055273	0.11584	0.005913	0.056643
thyroid2.dt	0.052773	0.057781	0.073528	0.093758	0.055694	0.069546	0.012782	0.01445
thyroid3.dt	0.050842	0.057937	0.068085	0.16795	0.0580325	0.0833615	0.008325	0.033761

Table 5.2: Average CPU Run Time for SADEGEL-BP and BP

Dataset	**Average runtime SADEGL-BP**	**Average runtime BP**	**Dimensionality**
building1.dt	100.59221	0.145316	93
building2.dt	97.9219	0.1265635	93
building3.dt	91.66719	0.1140645	93
cancer1.dt	105.96251	0.1187515	62
cancer2.dt	103.35	0.0875015	62
cancer3.dt	102.35626	0.093751	62
card1.dt	135.26407	0.114064	272
card2.dt	146.9922	0.0984385	272
card3.dt	118.28438	0.1015635	272
diabetes1.dt	90.90783	0.100001	57
diabetes2.dt	80.52503	0.098439	57
diabetes3.dt	93.12971	0.085939	57
flare1.dt	119.69689	0.096876	143
flare2.dt	110.48126	0.095314	143
flare3.dt	120.19846	0.0968755	143
gene1.dt	179.89533	0.151566	623
gene2.dt	183.14532	0.182816	623
gene3.dt	177.50782	0.157815	623
glass1.dt	107.0469	0.0890625	86
glass2.dt	121.06876	0.084376	86
glass3.dt	97.05626	0.075	86
heart1.dt	134.38438	0.0843755	192
heart2.dt	119.15315	0.095314	192
heart3.dt	126.19846	0.0875005	192
hearta1.dt	90.8469	0.090626	186
hearta2.dt	99.24532	0.0828135	186
hearta3.dt	104.17033	0.0812505	186
heartac1.dt	92.43907	0.076563	186
heartac2.dt	95.86095	0.089063	186
heartac3.dt	91.28125	0.085938	186
heartc1.dt	131.10157	0.0703125	192
heartc2.dt	129.44062	0.071875	192
heartc3.dt	136.68752	0.095313	192
horse1.dt	125.84688	0.092189	313
horse2.dt	126.3047	0.0875	313
horse3.dt	139.76721	0.090625	313
mushroom1.dt	243.9047	0.265627	642
mushroom2.dt	248.37188	0.210939	642
mushroom3.dt	273.78128	0.246877	642
soybean1.dt	200.94845	0.1640635	529
soybean2.dt	203.18907	0.137503	529
soybean3.dt	206.22971	0.170314	529
thyroid1.dt	144.94063	0.131253	128
thyroid2.dt	131.19845	0.14844	128
thyroid3.dt	130.64689	0.193752	128

complexity.

Table 5.3: SADEGL and SADEGL-BP comparison

Dataset	Max improvement	Least Improvement	Average improvement
building1.dt	0.025615	0	0.0137795
building2.dt	0.028219	0	0.022082
building3.dt	0.025587	0	0.017681
cancer1.dt	0.021973	6.2E-05	0.0187755
cancer2.dt	0.017101	0	0.014988
cancer3.dt	0.016406	0	0.013564
card1.dt	0.018952	0	0.014075
card2.dt	0.019106	0	0.015304
card3.dt	0.020052	0	0.016846
diabetes1.dt	0.090763	0.001727	0.0656595
diabetes2.dt	0.090763	0	0.043549
diabetes3.dt	0.059282	0	0.037295
flare1.dt	0.063127	0	0.0520675
flare2.dt	0.079066	0.005375	0.050366
flare3.dt	0.050884	0.000636	0.045364
gene1.dt	0.071974	0	0.056869
gene2.dt	0.080109	0.006721	0.050516
gene3.dt	0.056672	0	0.04497
glass1.dt	0.22075	0	0.15961
glass2.dt	0.22075	0	0.149505
glass3.dt	0.22779	0	0.13799
heart1.dt	0.20785	0	0.14298
heart2.dt	0.20193	0.03206	0.159055
heart3.dt	0.1907	0	0.15798
hearta1.dt	0.22588	0	0.211885
hearta2.dt	0.25695	0	0.19683
hearta3.dt	0.23853	0	0.13818
heartac1.dt	0.20189	0	0.193275
heartac2.dt	0.2019	0	0.18974
heartac3.dt	0.22411	0	0.18902
heartc1.dt	0.27013	0.00026	0.186125
heartc2.dt	0.23882	0.01528	0.204045
heartc3.dt	0.2029	0	0.171255
horse1.dt	0.20074	0	0.18553
horse2.dt	0.20071	0	0.198245
horse3.dt	0.20633	0	0.18984
mushroom1.dt	0.017506	0.001102	0.00873225
mushroom2.dt	0.013416	0.001427	0.00893285
mushroom3.dt	0.014892	0.000974	0.00940155
soybean1.dt	0.020638	0	0.01586
soybean2.dt	0.014997	0	0.0082026
soybean3.dt	0.010549	0	0.0097877
thyroid1.dt	0.010129	0	0.00682465
thyroid2.dt	0.029505	0.0009719	0.00927395
thyroid3.dt	0.010911	0	0.00617525

Chapter 6

Designing a User Interface for the SADEGL-BP algorithm

In this chapter an application is designed to act a user interface for training single layer MLPs using the SADEGL-BP algorithm.

6.1 Design Requirements

The user interface needs to at minimum to allow a user to change the control parameters of the SADEGL algorithm, and input their own data into the algorithm without directly interacting with the script. The user interface also needs to return the best found weights/biases, and the MSE of the network on the testing dataset, in a way which allows the user to easily save and interact with the data. The user interface needs to allow a user to choose their own training, validation and testing examples, however it would be also beneficial that they could choose random splits. Additionally, the user interface needs to allow a user to choose their own stopping tolerance and the duration of the stagnation period, aswell as the amount of neurons in the hidden layer. The user should also be able to decide how many times they want to run the algorithm.

The user interface would benefit from the automation/suggestion of various control parameters. For instance, in this paper it is recommend to use $K = \frac{NP}{10}$, hence when the population is updated K could be updated to this value. In general the control parameters should default to the values recommend in this paper. Additionally, after selecting data, data could be automatically split into a sensible ratio such as 50/25/25 for training, validation and testing respectively.

Whilst not necessary for the sake of functionality, the user experience will greatly benefit from the ability to pause/stop the algorithm from the user interface. Additionally, information being returned in runtime will decrease the "black box" feeling of running the script. Hence, a log should be updated in real time, informing the user of the current state of the algorithm and errors which occur during set-up. A symbol should be used to demonstrate the current state of the algorithm - paused, error or running. A graph should plot the MSE of each run to inform the user of how the algorithm is performing. Additionally, the best weights/biases and the MSE should be returned to both the workspace and the user interface after each run.

6.2 Design Implementation

The user interface was implemented using Matlab's App designer. The design view can be seen in Figure 6.1. Panels are used to organize the user interface into 7 distinct sections; Dataset, Control Parameters, Meta Settings, Run Algorithm , Best Positions Found, Log and Function Errors.

The Dataset panel features numeric edit fields, which allow users to input integers greater than 1. The user interacts with these fields to specify the amount of input and output variables. Users can select their dataset using the a button labelled "Choose Dataset", the name of the selected file will show in the non-editable text field next to the button. A tab group is used to toggle between using predetermined splits and random splits. If the user is using predetermined splits they enter the how many instances should be used for training, validation and testing. If the user is using random splits, they enter the percentage of data a split should use, hence this numeric field accepts real numbers in range $(0, 1]$.

The Control Parameters panel features numeric edit fields. The population, Neighbourhood and learning period fields only accept integers greater than 0. Additionally, the neighbourhood radius size changes to $\frac{NP}{10}$ after the population size is updated. The probability fields accept real numbers in range $(0, 1]$. Likewise, the Meta Setting panel features numeric fields, all of which only accept integers except for the stop tolerance field which accepts all real numbers greater or equal to 0.

The run algorithm panels features three buttons which allow the user to run, pause and stop the algorithm. As well as an uneditable field which displays the current run count. The buttons in this panel disable and re-enable themselves according to the current state of the user interface. If a dataset has been selected the run button becomes enabled, and when pressed the user interface checks for errors such as incorrect number of input and outputs. If there are no errors, the algorithm begins running and the button is disabled. When the run button becomes disabled, the pause button becomes enabled. The pause button stops the algorithm running at the end of the current iteration. When pressed the button becomes highlighted to inform the user their action has been recognized, and that the algorithm will pause momentarily. After the algorithm has come to a stop, this button changes to a resume button and the stop button becomes enabled. The resume button will resume the iteration progress, disable the stop button and change back to the pause button. After the stop button is pressed, the stop button becomes disabled, the resume button is switched back to the pause button, and the algorithm quickly fine tunes the current best solution of the current run, and stops the algorithm. After the algorithm has stopped, the run button becomes re-enabled.

The best positions found panel features a non editable table, in which the best weights and biases are stored after each run is completed. This table is also stored to the workspace, so users can easily manipulate the data. Similarly, the Function Errors panel features a non editable table in which the functions errors are stored after each run. Additionally, the panel features a graph which plots the the MSE for SADEGL and SADEGL-BP after each run.

The Log panel features a non editable text box. This textbox is cleared at the start of each run. During a run any update made to the textbox starts on a new line. The textbox is used

to inform the user when a run has started, ended, been paused or is stopping. The textbox also informs users of 3 detectable errors when a run is attempted with incorrect details. If the number of input and output variables, is not the same, an error is returned informing the user of this. However, it is not able to detect if the input/output split is actually correct. If the user has selected predetermined splits and the total value of each numeric field is not equal to the total number of instances, the user is informed. Similarly, if the user has selected random splits and the sum of the numeric fields is not 1, the user is informed. Due to the restriction of values in numeric fields, no other detectable errors can occur.

Additionally, the Log panel features a 'Lamp' which is used to inform the user of the current status of the application. If the algorithm is not running it is coloured grey, if an error occurs after pressing run it flashes red, if is running it is coloured green, and if it is paused/stopping it is coloured amber.

An image of the algorithm paused after multiple runs is shown in Figure 6.2. There are currently no known bugs, and it is impossible for the application to begin running the algorithm without detecting incorrect user input that would cause errors. This is due to the three mechanisms which produce error messages, and the strict limitations on numeric fields.

Users should save their data as comma separated variables, with the rows being data instances and columns being the inputs and output variables. Inputs variables must be stored in the left most columns. If the user wishes to use predetermined splits training instances must be in the top most rows, followed by the validation data and finally the testing data. The data should be preprocessed in the same fashion as the PROBEN1 datasets.

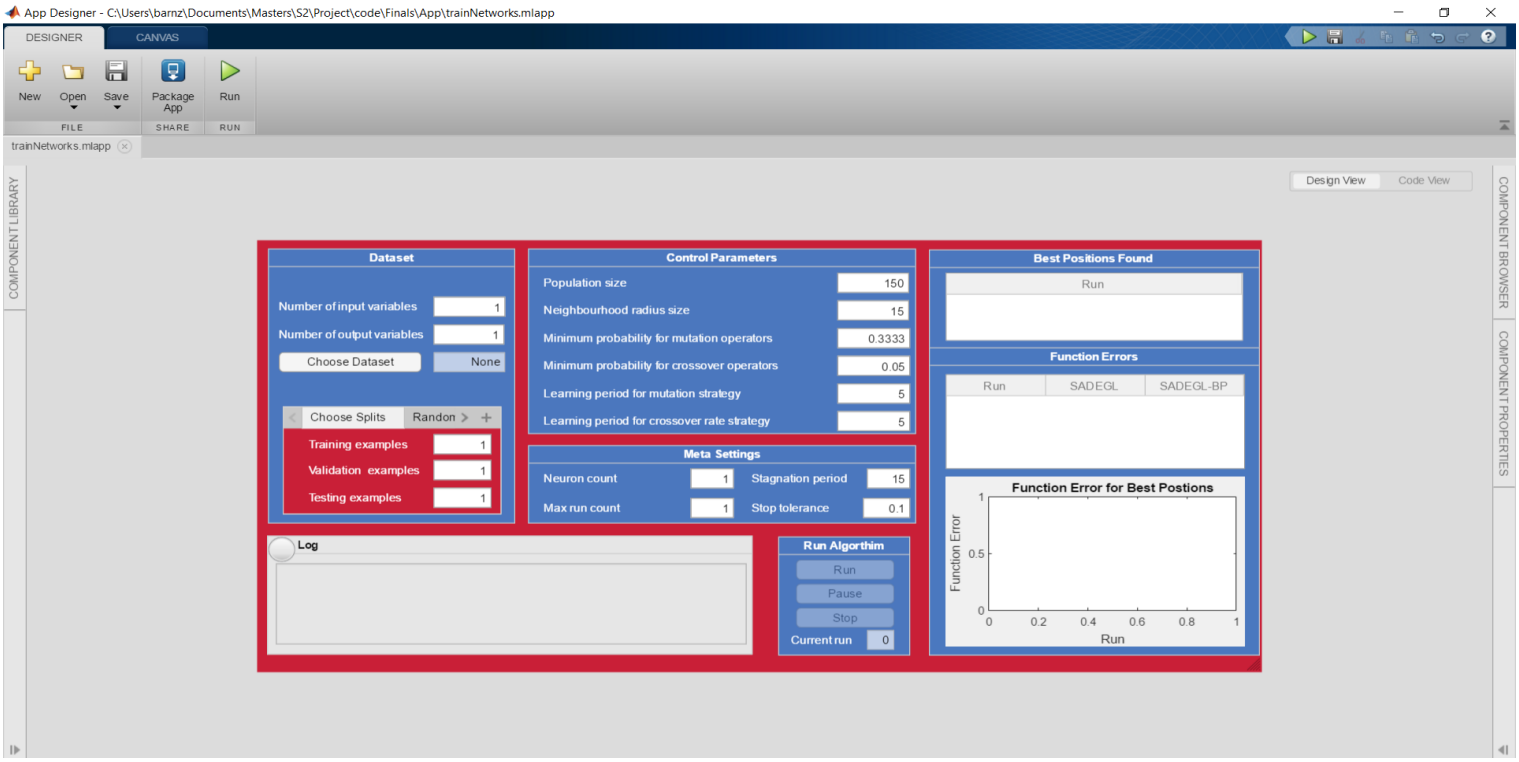

Figure 6.1: Design View of the User Interface for the SADEGL-BP algorithm

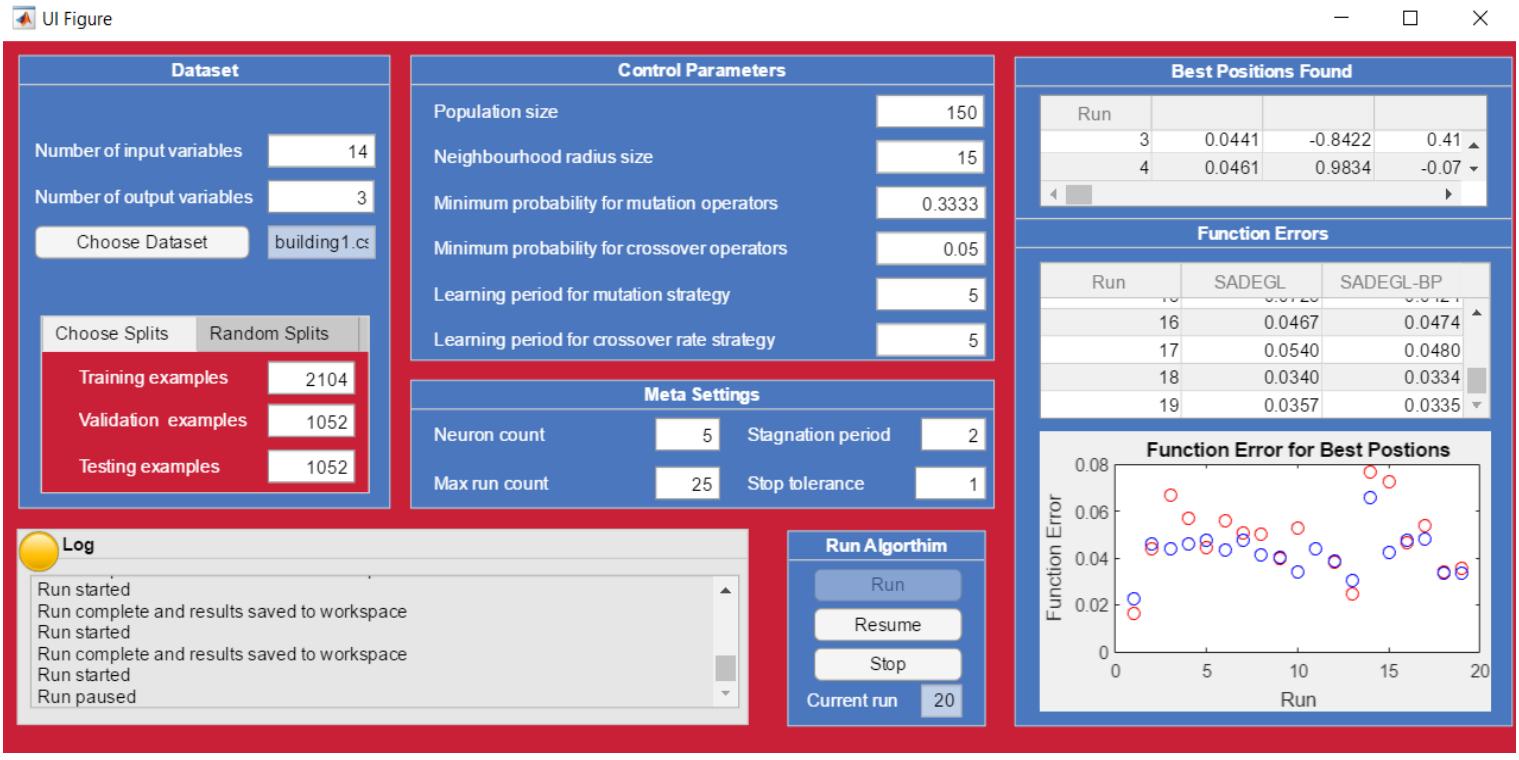

Figure 6.2: View of the Application in Use

Chapter 7

Project Management

7.1 Project Schedule

This project had 5 major tasks

1. Conduct background and literature review on DE, BP, and ANN training
2. Design and implement a DE algorithm, and benchmark it using a suitable test suite
3. Use the DE algorithm to train a ANN
4. Summarize the performance of the DE algorithm when used to train ANNs and optimize objective functions
5. Design and implement a user interface for the application

The time required for each task would be significant, hence a work plan was formulated to allow me to complete this project within the give time frame. The initial aim was to complete task 1 by may, so that I could use the knowledge gained to complete task 2 and 3 during June and July. During august I planned to complete tasks 4 and 5.

All tasks were completed on time, but more time was given to some tasks than expected. After completing task 1 the project became more focused on designing a high performing DE

algorithm which would be useful for training ANNs, rather than designing a high performing ANN model trained using DE. Hence, significantly more time was spent on designing the SADEGL algorithm than the neural network itself. However, this was not a case of time mismanagement, the extra time investment was required to improve the designed DE algorithm. Additionally, the application designed required relatively little time to develop due to its simple design. However, given more time I would like to have implemented extra functionality such as the ability to use alternative stopping conditions. Similarly, I would have designed an application for general use of the SADEGL algorithm.

7.2 Risk Management

The main risk of the project was failing to complete the objectives by the deadline. Failure to complete one task successfully, would prevent the next task from being completed successfully.

If my knowledge gained during task 1 was insufficient the performance of the developed DE algorithm would be unsatisfactory, therefore preventing me from completing my other tasks. To avoid this issue I dedicated a significant amount of time to researching DE and other EAs, and how they have been used to train ANNs currently. I identified both the importance of exploration and exploitation in EAs, and the research gap that currently existed - The lack of self-adaptive DE algorithms which use global and local neighbourhood mutation operators. Designing this algorithm based on current DE techniques ensured my algorithm would have atleast competitive performance with other modern DE algorithms.

Running evolutionary algorithms takes a significant amount of time, hence identifying any mistakes before the algorithm was benchmarked was crucial. If a mistake did occur after benchmarking the algorithm and using it to train ANNs, both tasks would have to be repeated. Rigorous and methodical checking of the Matlab implementation was used to prevent this.

Managing multiple similar scripts and results simultaneously can cause data to be mixed up. Hence, good storage structure was essential. The ANN,DE algorithm, and user interface were all stored separately. Additionally, results and scripts were frequently backed up onto a hard

drive and cloud storage systems since recreating them would be time consumed. Recreation would be essential if an unknown bug was to be introduced to the scripts/user interface, or the results of the algorithms became mixed.

7.3 Quality Management

Strict standards were adopted when benchmarking the SADEGL and SADEGL-BP algorithms. The CEC'2013 standards were adopted to evaluate SADEGL, all rules were followed accurately and results produced as described. Similarly, the PROBEN1 datasets were used to evaluate the SADEGL-BP algorithms ability to train ANNs. Most importantly the random number generation state for all runs of the algorithms are stored, and hence all results are reproducible since only reproducible results are should be considered scientific.

7.4 Social, Legal, Ethical and Professional Considerations

For this project there were no relevant Social, Legal, Ethical or Professional Considerations. Note that the paper uses only datasets which are publicly available for use.

Chapter 8

Critical Appraisal

In this paper I have identified a promising research gap in the field of evolutionary optimization, and designed an algorithm to fill it. The algorithm performed exceptionally well with results comparing favourably to the 10^{th} best algorithm presented in the CEC 2013 competition results paper [10].

The algorithms mutation strategy was was inspired by a mutation operator which created donor vectors for an individuals using both the best solution in the entire population, and the best solution in a mathematical ring surrounding the individual. The algorithm I developed used the concept of global and local searching for its mutation operators. It is unique compared to its inspiration in that global and local searches are separate operators which are chosen adaptively such that the algorithm chooses the most successful operator in recent iterations. This mutation strategy was designed based on the knowledge I had gained regarding the importance of both exploration and exploitation in evolutionary algorithms. Additionally, I implemented a 'current-to-rand' mutation operator to allow the algorithm to purely focus on exploration in a limited search space, rather than exploiting the best solutions.

The algorithm designed also features an adaptive crossover scheme, which unlike most adaptive crossover schemes in current research, is biased to extreme crossover rates that are close to 0 or 1. This decision was based on research which identified that less extreme values close to 0.5 neither progress consistently, or make large improvements when they do progress. Additionally,

unlike most adaptive crossover rate schemes, the algorithm uses the amount of improvement made by trial vectors to determine how to adapt the system, rather than just the number of times a crossover rate is successful.

I successfully trained ANNs using the developed DE algorithm and used BP to fine tune the results. Hybridizing EAs with local search techniques has become standard in recent years. The scope of this project was already ambitious, so other local search techniques were not experimented with. Experimenting with other techniques could have lead to interesting results.

The hybrid evolutionary-gradient algorithm developed consistently outperformed BP when used to train ANNs. However, the scope of the ANN training is limited to a single layer MLP. Its performance and usefulness on more advanced neural network structure (such as deep neural networks) is unknown.

The designed user interface for the algorithm is functional, however a more advanced application would be beneficial. More control parameters could be made available for changing such as the scale factor F, as well as a wider range of stopping conditions. Additionally, better pause functionality should be incorporated by adding more regularly pause checks when running the algorithm. Currently the algorithm has to finish the current iteration, before the application pauses it. Other potential improvements to the application include methods to generate relevant statistics in the application and the ability to save results without accessing the workplace.

Chapter 9

Student Reflections

During this project my time management was the only significant issue. I spent too much time fine tuning the DE algorithm, instead of moving onto the next task. If I had spent less time fine tuning the algorithm, the results of the ANN training would likely not have been significantly worse, and I would have had additional time to either implement a user interface with more functionality or test the algorithm on more complex ANN structures. Its also possible that the project was too ambitious and its scope to large, a simple user manual could have replaced the user interface.

Chapter 10

Conclusion

10.1 Summary of Thesis Achievements

The original aim of this project was to design and implement an optimised Differential Evolution algorithm to train Artificial Neural Networks. This was to be achieved by completing the following objectives

- Develop and implement a DE algorithm.
- Design an ANN as a surrogate model.
- Train the ANN using the DE algorithm.
- Produce a technical report of the DE algorithm and ANN which discusses the accuracy of its predictions when compared to the original model.

However, after completing a background and literature review on evolutionary algorithms and their applications to neural network. I discussed with my supervisor the new direction I wanted to take the research project in, whilst the aim of the project remained the same, the objectives changed to the following

- Develop and implement a DE algorithm.

- Design and implement a method to train ANNs using the DE algorithm
- Benchmark the performance of the ANNs trained by the DE algorithm using various datasets
- Evaluate the performance of the ANNs, comparing their performance to the same network trained using BP

These objectives have been successfully achieved.

- A Self-adaptive DE algorithm which adjusts both its mutation and crossover rate strategy according to the performance of its mutation and crossover operators was designed and successfully implemented. The algorithm is competitive with state of the art algorithms, with results favourably comparable to the 10^{th} best algorithm in the CEC'2013 competition.
- A script was designed and implemented to map the weights and biases of a single layer MLP of n neurons to the dimensions of the DE algorithm. The script uses provided training data to improve the DE algorithms population, and provided validation data as stopping criteria to avoid over fitting the algorithm to the training data.
- The ANN was benchmarked using the 45 PROBEN1 datasets.
- The performance of the networks was evaluated using both the designed DE algorithm and BP. The designed algorithm consistently outperformed BP.

Additionally, an application was designed and implemented which allows users to train ANNs using SADEGL-BP with their own data, with some flexibility in terms of control parameters, stopping conditions and training/validation/testing splits.

To summarize this project has successfully produced the following deliverables

- A script for the SADEGL algorithm

- A script that maps the structure of neural network to the SADEGL algorithm, and hybridizes it with BP
- An application which functions as a user interface for the SADEGL-BP algorithm

The designed SADEGL algorthim can be used minimize any real objective function. And the SADEGL-BP script or application can be used to train single layer neural networks with any amount of neurons.

10.2 Future Work

Population Size NP is not normally used as an adaptive control parameter in evolutionary algorithms. In SADEGL-BP the control parameter which determines the size of a local neighbour, K, is dependent on population size. An adaptive system which either adjust the size of the population or the size of local neighbourhoods K could further improve the exploration and exploitation capabilities of the algorithm. Additionally, in this paper local neighbourhoods are defined using mathematical rings. Other mathematical structures could be used to define alternative neighbourhoods for mutation operators. Alternatively, an adaptive algorithm which shuffles the population in order to change the local neighbourhoods could improve the algorithm. Additionally, Scale factor F is not adaptive in the SADEGL-BP algorithm. A self-adaptive strategy similar to the one used for crossover rates could be implemented, so that the algorithm decreases/increases the scale factor according to the success of scale factor values in recent iterations.

In this paper SADEGL is only hybridized with BP due to time constraints. Further experimentation should focus on hybridizing SADEGL with alternative local search methods. The algorthim should also be used to train more advanced forms of ANNs with suitably challenging datasets. Additionally, a method to optimize the amount of neurons a single layer MLP uses whilst simultaneously training the weights and biases should be developed for SADEGL. A simple method to do this would to be extend the dimensionality of a problem by 1, and this

dimension would be a boolean value representing the amount of neurons the individual should use. Whenever an individual is evaluated it should then be evaluating using only the specified amount of neurons. A suitable mutation and crossover scheme for this dimension would need to be designed.

The application which functions as a user interface should be developed further, so that it allows for both more control of parameters and stopping criteria. Additional functionality should be added to improve the applications of it, such as the ability to generate visual and statistical representations of the results.

Bibliography

[1] Janez Brest, Sao Greiner, Borko Boskovic, Marjan Mernik, and Viljem Zumer. Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. *IEEE transactions on evolutionary computation*, 10(6):646–657, 2006.

[2] Maurice Clerc. From theory to practice in particle swarm optimization. In *Handbook of Swarm Intelligence*, pages 3–36. Springer, 2011.

[3] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)*, 45(3):35, 2013.

[4] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. Differential evolution: a survey of the state-of-the-art. *IEEE transactions on evolutionary computation*, 15(1):4–31, 2011.

[5] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[6] John H Holland. Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computing*, 2(2):88–105, 1973.

[7] Momin Jamil and Xin-She Yang. A literature survey of benchmark functions for global optimisation problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, 4(2):150–194, 2013.

[8] Niklas Lavesson and Paul Davidsson. Quantifying the impact of learning algorithm parameter tuning. In *AAAI*, volume 6, pages 395–400, 2006.

[9] JJ Liang, BY Qu, PN Suganthan, and Alfredo G Hernández-Díaz. Problem definitions and evaluation criteria for the cec 2013 special session on real-parameter optimization. *Computational Intelligence Laboratory, Zhengzhou University, Zhengzhou, China and Nanyang Technological University, Singapore, Technical Report*, 201212:3–18, 2013.

[10] I Loshchilov, T Stuetzle, and T Liao. Ranking results of cec13 special session & competition on real-parameter single objective optimization. In *2013 IEEE congress on evolutionary computation, CEC, Cancun, Mexico*, pages 20–23, 2013.

[11] Rammohan Mallipeddi, Ponnuthurai N Suganthan, Quan-Ke Pan, and Mehmet Fatih Tasgetiren. Differential evolution algorithm with ensemble of parameters and mutation strategies. *Applied soft computing*, 11(2):1679–1696, 2011.

[12] Silja Meyer-Nieberg and Hans-Georg Beyer. Self-adaptation in evolutionary algorithms. In *Parameter setting in evolutionary algorithms*, pages 47–75. Springer, 2007.

[13] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta numerica*, 8:143–195, 1999.

[14] Adam P Piotrowski. Differential evolution algorithms applied to neural network training suffer from stagnation. *Applied Soft Computing*, 21:382–406, 2014.

[15] Adam P Piotrowski. Review of differential evolution population size. *Swarm and Evolutionary Computation*, 32:1–24, 2017.

[16] Lutz Prechelt et al. Proben1: A set of neural network benchmark problems and benchmarking rules. 1994.

[17] A Kai Qin and Ponnuthurai N Suganthan. Self-adaptive differential evolution algorithm for numerical optimization. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 2, pages 1785–1791. IEEE, 2005.

[18] Partha Pratim Sarangi, Abhimanyu Sahu, and Madhumita Panda. A hybrid differential evolution and back-propagation algorithm for feedforward neural network training. *International Journal of Computer Applications*, 84(14), 2013.

[19] Josef Tvrdík and Radka Poláková. Competitive differential evolution applied to cec 2013 problems. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1651–1657. IEEE, 2013.

[20] Lin Wang, Yi Zeng, and Tao Chen. Back propagation neural network with adaptive differential evolution algorithm for time series forecasting. *Expert Systems with Applications*, 42(2):855–863, 2015.

[21] Daniela Zaharie. Influence of crossover on the behavior of differential evolution algorithms. *Applied soft computing*, 9(3):1126–1138, 2009.