Research in Industrial Projects for Students

Sponsor

The Aerospace Corporation

Final Report

Combining Genetic Algorithms and Machine Learning (CgALM) for Modeling Complex Systems

Student Members

Rachel Duquette (Project Manager), *Boston College*
duquetra@bc.edu

Jacob Chang, *University of Notre Dame*

Katherine Thai, *Rutgers University*

Tongyu Zhou, *Williams College*

Academic Mentor

Minh Pham, minhrose@ucla.edu

Sponsoring Mentors

Dr. Victor Lin, victor.s.lin@aero.org

Dr. Leah Ruckle, leah.ruckle@aero.org

Dr. Karen Wood, karen.wood@aero.org

Mr. James E. Gidney, Jr., James.E.Gidney@aero.org

Date: August 21, 2019

This project was jointly supported by The Aerospace Corporation and NSF Grant DMS 1440415.

Abstract

Essential for services such as communication, navigation, and weather prediction, satellite constellations must be designed to minimize loss of coverage while subject to constraints on space traffic and the number of satellites available. Genetic algorithms (GAs) offer a versatile method of optimization with demonstrated success in applied problems. However, in the case of a computationally expensive problem, such as satellite coverage of the earth, the necessity of repeated fitness evaluations prevents convergence of a GA in a feasible time frame. Implementing a more efficient surrogate model to estimate the expensive objective function poses a potential solution to this dilemma. Recent advances in machine learning methods, in particular neural networks, make them a compelling candidate as a surrogate function. A genetic algorithm incorporating an ensemble of neural networks as a surrogate function is evaluated on a set of canonical test problems, including those with discrete inputs and multimodal objective functions, and finally applied to the problem of constellation design.

Acknowledgments

This RIPS 2019 report was created by Jacob Chang, Rachel Duquette, Katherine Thai, and Tongyu Zhou, with the assistance and mentorship of our academic mentor, Minh Pham.

Firstly, the team would like to thank The Aerospace Corporation for their dedication and collaboration with RIPS over the past several years. Without their sponsorship, this program and project would not have been possible. In particular, we would like to thank our industry sponsors, Victor, Leah, Karen, and Jim. Their support and guidance this summer was invaluable, and the project would not have been successful without them. We owe many thanks to Minh, our academic mentor, and Susana, RIPS Director, for their advice, assistance, and support in our endeavours throughout the program. Finally, we would also like to thank David Medina and the IT Department, the Finance Department, and all the IPAM staff who made this program possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Aerospace Corporation

This project is sponsored by The Aerospace Corporation, a federally funded research and development center committed to the space enterprise. They are headquartered in El Segundo, California, and provide services to both private companies and governmental agencies, such as the U.S. Air Force. These services include technical analyses and assessments, as well as day-of-launch support and risk evaluation.

Aerospace conducts research to solve a variety of optimization problems, ranging from the management of space traffic to the placement of satellites to minimize downtime and maximize global coverage. One challenge in solving optimization problems is the computational expense of explicitly evaluating the quality of potential solutions. Consequently, researchers are always interested in methods to mitigate the time and intensity of the required computations.

1.2 The Proposed Problem

The Aerospace Corporation has proposed a RIPS project in which machine learning techniques will be integrated into genetic algorithms with the goal of improving genetic algorithm performance on global optimization problems. The Aerospace team is developing an algorithm that leverages both genetic algorithm (GA) and machine learning (ML) techniques solve a complex modeling problem: the determination of Walker constellation parameters that optimize global dilution of precision performance, subject to constraints on the number and distribution of satellites. This application problem will be addressed in Chapter 5 of this report.

We will begin with a simple genetic algorithm framework. In an effort to improve the computational time required to evaluate the fitness of the individuals during each iteration of the algorithm, we will implement a machine learning method to approximate the objective function. Several machine learning techniques will be studied and compared before one is selected for the final software product.

We will also identify, compare, and select different methods and schemes for the selection, crossover, and mutation phases of the genetic algorithms.

The genetic algorithm with machine learning integration will then be used to solve for a set of parameters that define an ideal Walker constellation.

1.3 Our Team's Approach

We conducted an initial review of the existing literature, focusing specifically on schemes and techniques researchers used to optimize standalone genetic algorithm performance. Next, we investigated commonly used machine learning methods, comparatively evaluating them on their ability to approximate functions of varying degrees of complexity and their suitability for incorporation within a genetic algorithm.

After establishing aspects of the genetic algorithm we would like to test, we constructed a generic GA framework using the Python package DEAP (Distributed Evolutionary Algorithms in Python) [10]. This general algorithm was then modified for our operators of interest, including selection, crossover, and mutation, and tested on 18 common optimization benchmark functions. To effectively qualify the wide range of hyperparameter values affecting algorithm performance, we looked further into design experiments that would identify an ideal set of these hyperparameters to optimize our objective functions while simultaneously reducing computing time. A Taguchi orthogonal array was implemented alongside signal-to-noise ratios and ANOVA analysis to isolate ideal parameter levels and identify which factors were statistically significant in run time and identification of a function's global minimum.

Next, we implemented three machine learning techniques for function approximation, including support vector regression (SVR), multilayer perceptron (MLP) networks, and radial basis function (RBF) networks. We hand-tuned the hyperparameters for each model in order to optimize performance of the predictor. Because we desired greater generalizability in our models, we decided to not proceed with SVRs after finding that their parameters were too sensitive to minor adjustments. After identifying an ideal set of parameters for each category of benchmark function (see Tables 1.1, 1.2, 1.3), we embedded these methods within the genetic algorithms by substituting the expensive fitness evaluation step with the surrogate model.

We then constructed a hybrid machine learning-assisted genetic algorithm (MLGA) by introducing the machine learning models in ensemble after a set number of generations passed in the genetic algorithm. Because we wanted the predictions to maintain high accuracy, we weighted each model in the ensemble accordingly as a function of its generalized mean squared errors. After the genetic algorithm converged to a pool of optima, we then reintroduced the original objective function to evaluate fitnessses of the final population.

After confirming the effectiveness of the MLGA on our benchmark functions, we applied our model to the computationally expensive Walker constellation problem, trying to minimize 98% global position dilution of precision (PDOP).

1.4 Benchmark Functions

Most research in surrogate-assisted evolutionary optimization has focused either on a specific real-world application or on a set of general test problems, as there are no benchmarking functions designed specifically for testing surrogate function efficacy [17]. For the purposes of our problem, we tested genetic algorithms on eighteen canonical optimization problems spanning a wide variety of different features and levels of difficulty. The first fourteen test functions were drawn from Digalakis [7], with minor corrections from More [26]. Of these, the first eight are bounded problems, where the space of possible solutions is bounded by some cube or hypercube, and are listed in Table 1.1. The remaining six functions are nonlinear squares minimization problems, found in and are listed in Table 1.2.

Table 1.1: Benchmarking Functions

	Name	Formula	Bounds	Dim
F1	Sphere	$f_1 = \sum_{i=1}^{2}(x_i)^2$	$\lvert x_i\rvert \leq 5.12$	2
F2	Rosenbrock	$f_2 = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$	$\lvert x_i\rvert \leq 2.048$	2
F3	Step	$f_3 = \sum_{i=1}^{5}[x_i]$	$\lvert x_i\rvert \leq 5.12$	5
F4	Quartic	$f_4 = \sum_{i=1}^{30}(ix_i^4 + \text{Gauss}(0,1))$	$\lvert x_i\rvert \leq 1.28$	30
F5	Shekel's Foxholes	$f_5 = \frac{1}{\frac{1}{500} + \sum_{j=1}^{25}\frac{1}{f_j(X)}}$ $f_j = 0.002 + \sum_{j=1}^{25}\left(\frac{1}{j} + \sum_{i=1}^{2}(x_i - a_{ij})^2\right)$	$\lvert x_i\rvert \leq 65.534$	2
F6	Schwefel	$f_6 = 10V + \sum_{i=1}^{10}(-x_i \sin(\sqrt{\lvert x_i\rvert}))$, $V = 418.9829101$	$\lvert x_i\rvert \leq 500$	10
F7	Rastrigin	$f_7 = 20A + \sum_{i=1}^{20}(x_i^2 - 10\cos(2\pi x_i))$, $A = 10$	$\lvert x_i\rvert \leq 5.12$	20
F8	Griewangk	$f_8 = 1 + \sum_{i=1}^{10}\left(\frac{x_i^2}{4000}\right) - \prod_{i=1}^{10}\left(\cos\left(\frac{x_i}{\sqrt{i}}\right)\right)$	$\lvert x_i\rvert \leq 600$	10

Table 1.2: Squares Problems

	Name	Formula	Dim
F9	Watson Function	$f_i(x) = \sum_{j=2}^{n}(j-1)x_j t_i^{j-2} - \big(\sum_{j=1}^{n} x_j t_i^{j-1}\big)^2 - 1,$ $1 \le i \le 29,\ t_i = \frac{i}{29},$ $f_{30}(x) = x_1,$ $f_{31}(x) = x_2 - x_1^2 - 1$	$2 \le n \le 31$ $m = 31$
F10	Extended Rosenbrock Function	$f_{2i-1}(x) = 10(x_{2i} - x_{2i-1}^2)$ $f_{2i}(x) = 1 - x_{2i-1}$	$n \in \mathbb{Z}_2,$ $m = n$
F11	Penalty Function II	$f_1(x) = x_1 - 0.2$ $f_i(x) = a^{\frac{1}{2}}\big(\exp\big(\frac{x_i}{10}\big) + \exp\big(\frac{x_{i-1}}{10}\big) - y_i\big),$ $2 \le i \le n$ $f_i(x) = a^{\frac{1}{2}}\big(\exp\big(\frac{x_{i-n+1}}{10}\big) + \exp\big(\frac{-1}{10}\big) - y_i\big),$ $n < i < 2n$ $f_{2n}(x) = \big(\sum_{j=1}^{n}(n-j+1)x_j^2\big) - 1$ where $a = 10^{-5}$ and $y_i = \exp(\frac{i}{10}) + \exp(\frac{i-1}{10})$	n variable, $m = 2n$
F12	Powell Badly Scaled Function	$f_1(x) = 10^4 x_1 x_2 - 1$ $f_2(x) = \exp(-x_1) + \exp(-x_2) - 1.0001$	$n = 2,$ $m = 2$
F13	Gulf Research and Development Function	$f_i(x) = \exp\big(\frac{-\lvert y_i - x_2\rvert^{x_3}}{x_1}\big) - t_i$ $t_i = \frac{i}{100}$ $y_i = 25 + (-50\ln(t_i))^{\frac{2}{3}}$	$n = 3,\ n \le$ $m \le 100$
F14	Extended Powell Singular Function	$f_{4i_3}(x) = x_{4i-3} + 10x_{4i-2}$ $f_{4i_2}(x) = 5^{\frac{1}{2}}(x_{4i-1} - x_{4i})$ $f_{4i_1}(x) = (x_{4i-2} - 2x_{4i-1})^2$ $f_{4i}(x) = 10^{\frac{1}{2}}(x_{4i-3} - x_{4i})^2$	$n \in \mathbb{Z}_4,$ $m = n$

An area of open research within the usage of surrogates in evolutionary computation is their implementation on non-continuous problems, in particular on combinatorial problems. The optimization problem suggested by Aerospace on Walker constellation design features a discrete input space, adding additional importance to testing in this domain of functions. To that end, we also investigated four optimization problems with discrete inputs, suggested by Aerospace: Ackley, Levy, Schaffer No. 2, and Bukin No. 6 (Table 1.3. These functions, while normally continuous, were chosen to be discretized on the criteria that both the global minima and local minima were located on integer points and that the search space remained a reasonable size with integer points only.

Table 1.3: Discretized Benchmarking Functions

	Name	Bounds	Dimensions
F15	Ackley	$\lvert x_i\rvert \leq 33$	d = 10
	$f_{15} = -a\exp\big(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\big) - \exp\big(\sqrt{\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)}\big) + a + \exp(1)$		
F16	Levy	$\lvert x_i\rvert \leq 10$	d = 10
	$f_{16} = \sin^2(\pi w_1) + \sum_{i=1}^{d-1}(w_i-1)^2[1+10\sin^2(\pi w_i+1)] + (w_d-1)^2[1+\sin^2(2\pi w_d)]$		
	$w_i = 1 + \frac{x_1-1}{4}, \forall i \in 1,\ldots,d$		
F17	Schaffer No. 2	$\lvert x_i\rvert \leq 100$	$d = 2$
	$f_{17} = 0.5 + \frac{\sin^2(x_1^2-x_2^2)-0.5}{[1+0.001(x_1^2+x_2^2)]^2}$		
F18	Bukin No. 6	$\lvert x_1\rvert \leq 15,\ \lvert x_2\rvert \leq 3$	$d = 2$
	$f_{18} = 100\sqrt{\lvert x_2 - 0.01x_1^2\rvert} + 0.01\lvert x_1 + 10\rvert$		

Chapter 2

Genetic Algorithms

Genetic algorithms are inspired by Darwinian natural selection. The algorithm maintains a population of individuals, where each individual represents a candidate solution to the objective function, which evolves over the run of the algorithm. Selection, crossover, and mutation schemes aid the algorithm in traversing the search space and locating the global optimum. The "fitness" of each individual is determined by direct computation of the objective function. In the case of minimization, individuals with lower fitness values are demonstrated to be better candidate solutions and are favored in selection processes moving forward.

GAs are versatile and effective tools in applied problems as they are able to traverse search spaces relatively well. However, one disadvantage of these algorithms is the necessity to constantly reevaluate the objective function, which can be computationally expensive, given a complex objective function. An outline of a generic genetic algorithm can be seen in Figure 2.1

2.1 Background

2.1.1 Selection

After generating the starting population and evaluating fitnesses for the original population members, selection schemes must be implemented to determine which individuals are chosen as parents. Several different selection strategies are outlined below.

- **Tournament**: A random number of individuals is taken from the population and placed in direct competition with one another, selecting the strongest individual as a parent. The tournament is repeated until there are sufficient parents to move onto the crossover stage. Small tournament sizes are helpful for maintaining diversity, since weaker population members have a smaller pool of individuals to "defeat" and have greater opportunity to move forward as parents. On the other hand, larger tournament sizes allow the strongest members of a population to pass their traits onward more quickly, but risks premature convergence upon a local minimum if there is insufficient diversity in the initial population. Another downside of this technique is that certain individuals might not be chosen to compete in tournaments at all and hence, will be lost, even if they are considered strong population members.

  Tournament-style selection schemes are generally considered to be the most efficient [28], due to the binary competition selections rather than utilizing sorting and rank-

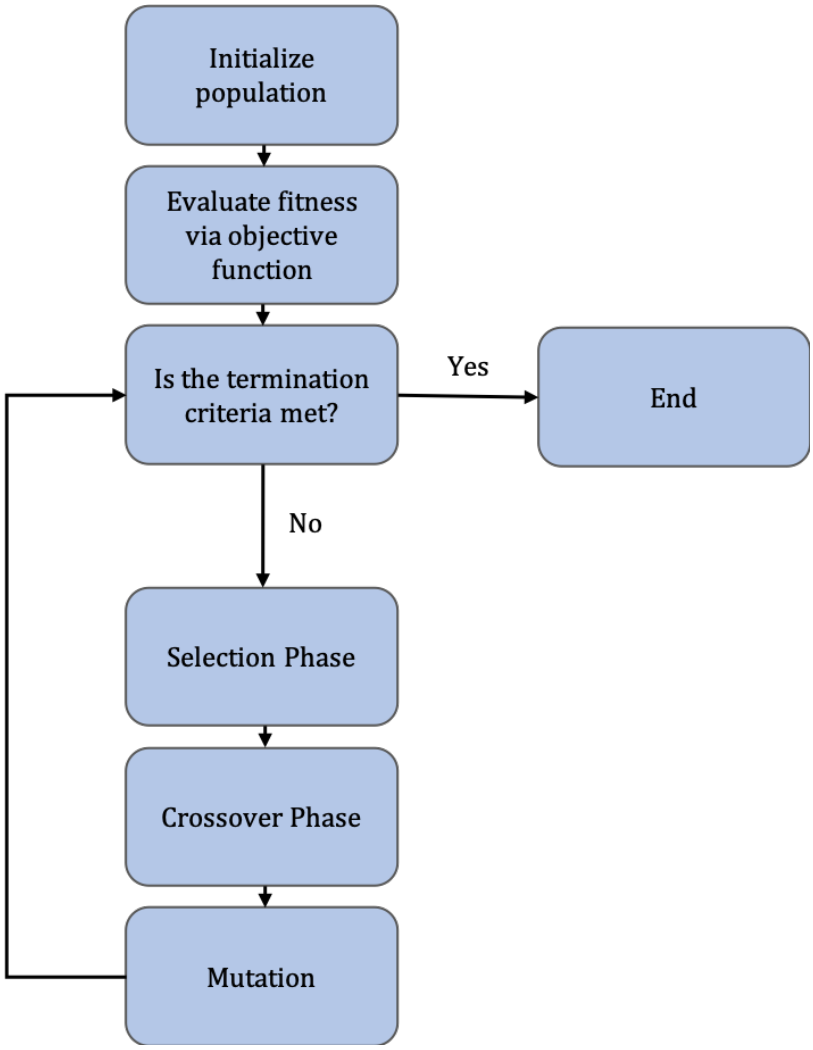


Figure 2.1: Generic outline of a genetic algorithm

ing algorithms. Overall, this technique is quite efficient while also preserving low susceptibility to takeover by dominant individuals.

- **Elitism**: With selection schemes being largely reliant on random processes, even the strongest population members face the possibility of being eliminated. Elitism is a general strategy for giving the strongest performers preferential treatment in selection. We explored two main forms of elitism: archival and elitist recombination [21].

    - **Archive**: Archival systems keep record of the top performing population members and allow them to be incorporated into the population at each selection stage of the algorithm. This ensures that highly fit individuals constantly have the opportunity to lend their attributes to the formation of a new generation, even if they have been eliminated in other stages due to random processes. In our algorithms, the archive is set as a given percentage of the population size, and before the selection phase, individuals from the existing population are randomly selected for replacement by the archive members. One downside of this technique is the potential for rapid loss of diversity, and hence, convergence

upon a local minimum.

- **Elitist recombination**: This is a secondary tournament where offspring must compete with their parents in order to be included in the next generation. After parents are selected and crossover occurs to form offspring, the family unit competes against one another. The top two performers of this tournament are passed forward. This prevents backwards movement and ensures that good solutions are not lost during the search process. One disadvantage of this technique is the necessity for additional objective function evaluations in order to compare performers within a family.

This list is not exhaustive, and information on alternative methods may be found in Appendix A.

2.1.2 Crossover

Also known as recombination, crossover is a critical step in ensuring the overall improvement of the genetic algorithm throughout each iteration. It provides a means of exchanging information, or "traits," between two parents to pass onto their offspring. It is a stochastic process that generates more diversity while maintaining an overall high fitness in the existing population. While we explored various crossover schemes, such as partially matched, blend, simulated binary, and voting recombination, operating on a wide variety of different data structures, we focused on the following two as they they were most suitable for a genetic algorithm where candidate solutions are represented as arrays of points. More information on the other methods investigated may be found in Appendix A.

- **One-point**: A single, common intersection point is randomly selected on both parents. Values to the right of this intersection point are directly swapped to form two offspring.
- **Uniform**: Each consecutive bit in the offspring is chosen from either parent with a probability p for one parent and a probability $1 - p$ for the other. While $p = 0.5$ for both parents is the most standard setup, other mixing ratios can also be used, resulting in certain offspring that will inherit more genetic information from one parent than the other.

To perform an adequate search that is simultaneously comprehensive and efficient, the crossover scheme utilized must ultimately be global and unbiased. In a study conducted by Poli and Langdon [30], one-point was found to only exchange very small amounts of genetic material in a converged population. The search performed with this operator also grew increasingly local and biased as time went on. In contrast, although uniform crossover also became more local, it remained largely unbiased as any information in either parent has the same chance of being inherited by the offspring. For our implementation of the genetic algorithm, we implemented and tested both the one-point and uniform crossover schemes.

2.1.3 Mutation

Mutation introduces or reintroduces diversity into the population by perturbing individuals. This additionally diversity can help the GA to explore the search space and can also help to prevent premature convergence at a local minima.

During each iteration of the genetic algorithm, certain offspring are chosen for mutation. Within this pool of offspring, certain genes are then mutated. Because we must choose the individuals to mutate, select the specific genes on which the mutations occur, and actually perform the mutation, there are many different mutation strategies and schemes to consider when developing GAs.

Mutation Operators

- **Gaussian**: A random value from a Gaussian distribution with user-defined standard deviation and mean is added to a "gene." This is used in classical evolutionary programming.
- **Cauchy**: A random value from a Cauchy distribution is added to a "gene." The thicker tails of the Cauchy distribution encourage larger mutations and are best suited for a search space that contains local optima that are far away from each other [39].

Mutation Schemes

Mutation schemes generally describe the way in which individuals and genes are selected for mutation. The simplest mutation schemes have a predetermined frequency of mutation, such as mutating one gene per individual or mutating a gene with probability $1/n$, so that, on average, one gene per individual is mutated. The probability that an individual is mutated and the probability that a gene is mutated are both parameters that must be chosen.

There are many other mutation schemes and algorithms, which were investigated but not utilized in this project. Some of these options are listed in Appendix A.

2.1.4 Termination Criterion

In general, a genetic algorithm should terminate when the population of solutions contains a solution attaining the global minimum or when the population of solutions has ceased to improve and further generations would not improve the result. The termination criterion is then exceedingly important: it must allow the genetic algorithm enough time to successfully converge while maintaining a suitable exploration phase, yet avoid unnecessary computation. Furthermore, an effective termination criterion must assure that the algorithm will eventually terminate. In practice, it is difficult to achieve reliability (assurance of termination) while simultaneously avoiding premature or late convergence.

- **Limit-Based Termination**: Convergence criteria that take the form of a limit on either the run time or number of generations for which the genetic algorithm may run. There are a number of variations of this termination criterion, such as a bound on the number of objective function evaluations. Due to the nature of a genetic algorithm, these variations are all linearly dependent, meaning that given a bound in one form, it is possible to convert to an equivalent bound in any of the alternate forms [16]. Such a criterion assures reliability, but at the expense of performance. When the genetic algorithm reaches this limit, it halts, without regard to the fitness of the individuals in the population. If the criterion is set too high, the genetic algorithm may run for several generations without improvement, delaying receiving a result and wasting computational power. Conversely, a too low criterion will prevent the algorithm reaching a good solution, rendering the result minimally useful.

- **Bound-Based Termination**: Criterion that causes algorithm to terminate when a solution hits a specified bound. In this case, the genetic algorithm terminates when the fitness of the best individual in the population reaches or exceeds a certain threshold. While this termination strategy ensures the success of the algorithm's result, it is not guaranteed to cause the algorithm to halt in finite time. In particular, if the population contains outliers or the algorithm converges to a local minimum, it may never terminate [16]. In practice, this termination criterion also requires some knowledge of the fitness value of the optimal point in order to set an appropriate bound, limiting its utility. Other bound-based termination criteria terminate when the standard deviation of the population or the difference between the fitness values of the best and worst individuals in the population drop below a certain threshold. Both attempt to capture the internal diversity of the population and terminate when it has reached a level at which further improvement is unlikely or will be infeasibly slow. Like a bound on the fitness, these termination criteria are not reliable, as outliers in the population or local minima of the fitness function may cause the algorithm never to satisfy the necessary criterion. Furthermore, they are unsuited to discrete spaces where the differences between values are necessarily not arbitrarily small [16].

- **Running Mean Termination**: The fitness of the best individual in the current population is compared to the average of the fitness values of the best individuals in the last n generations. Based on testing of continuous and discrete benchmark functions, Jain et al [16] recommend $n = 15$ as the comparison distance. When the difference is less than the chosen precision level, the algorithm halts. Unlike the previous criteria, running mean is guaranteed to converge within finite time and is suitable for discrete spaces. Running mean does not guarantee the fitness of the final solution, but where n is chosen in proportion to the complexity of the function, it avoids needless computation if the algorithm has ceased to improve due to being trapped in a local minimum. Thus running mean provides a reasonable compromise between reliability and performance.

2.2 Experiments & Methodology

2.2.1 Design of Genetic Algorithms

Our genetic algorithms were implemented using DEAP [10]. However, the selection schemes, mutators, crossover schemes, and stopping criterion were written as functions and appended onto the algorithm's DEAP framework. A generic form of the algorithm used in this report uses tournament-based selection, one-point crossover, Gaussian mutation, and a generation-based stopping criterion. This generic algorithm was tested on the 18 canonical benchmark functions described in Chapter 1.

As the functions were tested, modifications such as the running mean termination criterion, elitism, uniform crossover, and Cauchy mutation were developed and implemented. While each of these techniques had their own strengths and benefits, they also increased the variety of hyperparameters needed to be tested and tuned to improve GA performance.

2.2.2 Taguchi Design of Experiments

Orthogonal arrays were implemented to create a list of experiments and efficiently test different parameter combinations while reducing the experimental numbers of a full factorial

approach. In each orthogonal array, the columns "consist of a number of conditions depending on the levels assigned to each factor," while the rows represent a unique experiment [31].

The formal definition of an orthogonal array, as written in [14] is presented below:

Definition 1 (`Orthogonal Array`) *An $N \times k$ array A with entries from S is said to be an orthogonal array with s levels, strength t and index λ (for some t in the range $0 \leq t \leq k$ if every $N \times t$ subarray of A contains each t-tuple based on S exactly λ times as a row.*

In order to find or construct an appropriate orthogonal array, one must first determine which factors and levels to test. It is important to note that orthogonal arrays are non-trivial to construct, especially for arrays with different levels of factors (i.e. one factor with three levels and another factor with two levels), or when a factor has more than three levels. If a pre-existing appropriate array cannot be found, it is often best to reduce the number of factors being tested and/or consolidate the levels of the factors. This restriction in variable freedom is one of the major downsides of Taguchi's method, since researchers must often conform their experiments to the design rather than the design to the experiments.

2.2.3 Analysis of Results

After our initial experiments were conducted, signal-to-noise ratios (SNR) were calculated and compared to determine optimal parameter settings. While this ratio was originally and most typically used in science and engineering problems, it also has great utility in the realms of mathematics, statistics, and data analysis. SNR is a useful tool for determining the strength of a certain signal with respect to background noise. In GAs, this technique is highly useful for filtering through the noise coming from the various random processes inherent to the algorithm. With repeated trials of GAs often yielding drastically different results, SNRs can isolate the variation in performance and fitness convergence coming from specific hyperparameters rather than the stochasticity of the algorithms themselves.

There are several different forms of the SNR equation, depending on the type of optimization problem at hand (i.e. minimization versus maximization). These forms are represented below [13]:

$$SNR = -10 \log \big(\frac{\hat{y}}{s^2 y}\big) \tag{2.1}$$

$$SNR = -10 \log \big((1/n)\big(\sum_{i=1}^{n} y_i^2\big)\big) \tag{2.2}$$

$$SNR = -10 \log \big((1/n)\big(\sum_{i=1}^{n} \frac{1}{y_i^2}\big)\big) \tag{2.3}$$

Since our optimization problems dealt with minimization, we were mainly concerned with Equation (2.2). Here, note that n represents the population sizey_i represents the final fitness value of each individual in the last generation, and $\hat{y}$ represents the predicted fitness value of an individual. We ran each experiment 100 times and calculated SNRs for each trial. An average SNR for each experiment was then computed, to be used in later statistical analysis.

The average SNR values from the experiments were compared on a factor by factor level, comparing the average SNR for a factor across all levels. For example, when looking

at the effect of population size, we would average the SNR scores for all experiments that used a population size of 250, then the average SNR scores for all experiments of population size 500, and then 1000. These average scores can be compared, and the highest SNR is determined to be the "preferred" setting of the three options. This process can be performed on all the different factors until we can isolate an "ideal" set of parameters.

Next, analysis of variance (ANOVA) was used to determine which factors were statistically significant in affecting optimization. Essentially, an ANOVA test compares the variation of the SNR values against the various factors to determine whether or not they have a statistically significant overall effect. In ANOVA, the test statistic is the F-statistic from the Fisher distribution, with a typical p-value < 0.05).

After performing ANOVA on all factors to test their effect on SNR, one can determine which variables have a statistically significant effect on optimization. Using this information, we are able to determine which variables are most sensitive to future tuning, and the levels at which these variables should be set. Additionally, this also provides information on which variables have relatively little effect on optimization, and signals that certain factors may be left constant to reduce the quantity of hyperparameters in question.

2.3 Results

The following section contains some results for the sphere, Rastrigin, and Bukin No. 6 functions. While the sphere function is quite simple to optimize, the Rastrigin function has a large number of local minima, making minimization difficult. The Bukin No. 6 function is interesting in this analysis because it has been discretized, adding an additional layer of complexity.

2.3.1 Determination of Parameters

Initial tests of the generic GA were performed with a 1000 generation-limit. This is shown in Figure 2.3.

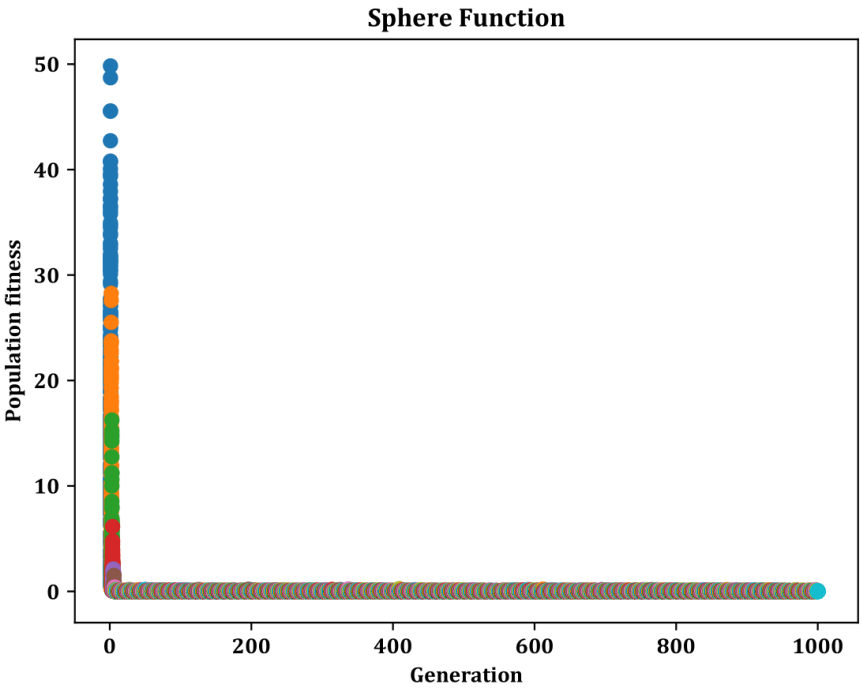


Figure 2.2: Sphere function: 1000 generation limit convergence plot

However, when looking at this plot, it is clear that after the first few generations, improvements in fitness convergence are quite negligible. Due to this, the running mean termination criterion (with the 15 generation comparison recommended by Jain et. al [16]) was implemented to prevent the GA from unnecessarily evaluating generations. Examples of plots displaying this improved convergence efficiency are shown by Figures 2.3, 2.4, and 2.5.

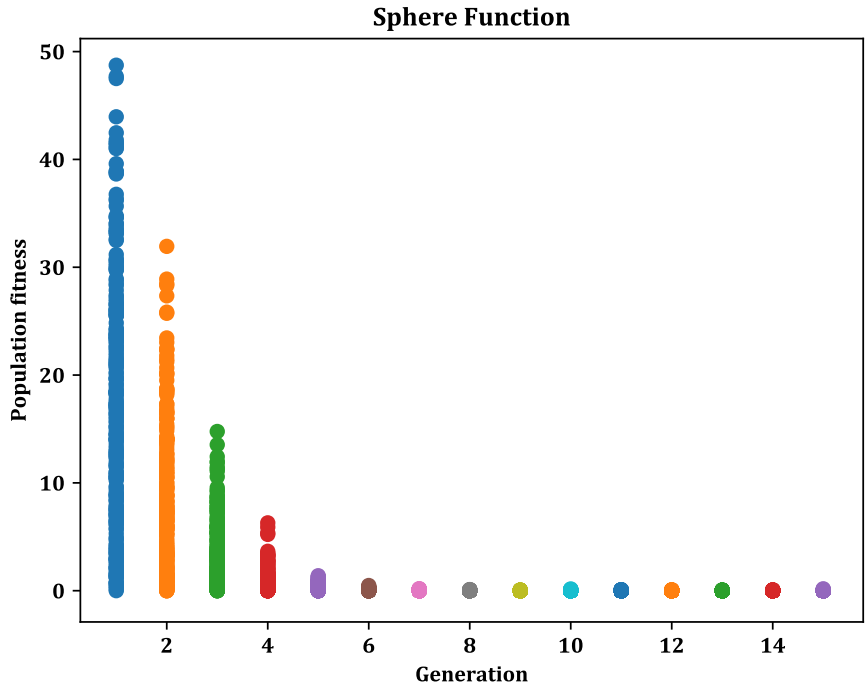


Figure 2.3: Sphere function: running mean termination criterion plot: convergence plot of the sphere function after the running mean stopping criterion was implemented (with 15 generation comparison). It now terminates after 15 generations, rather than 1000.

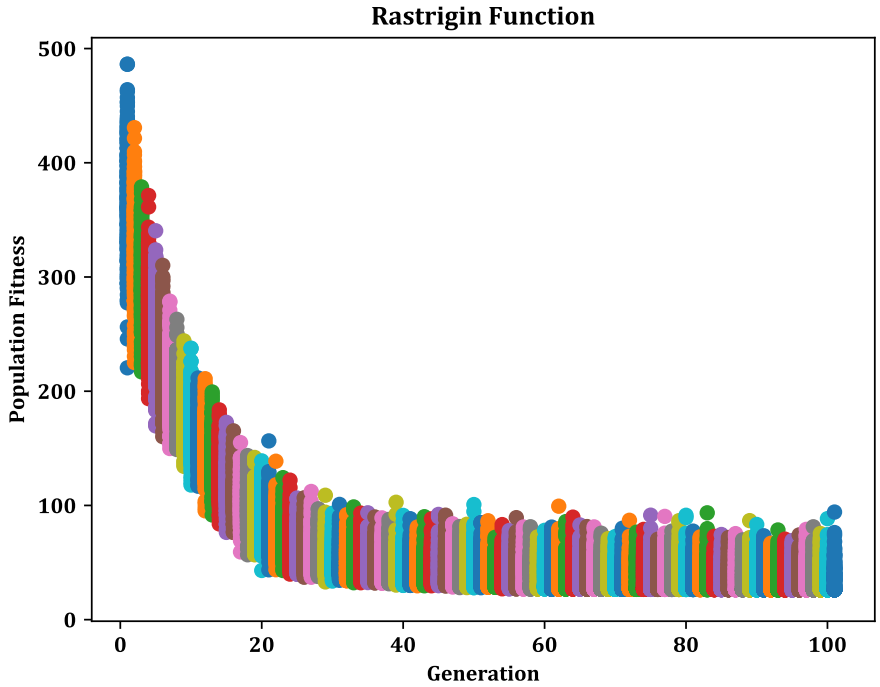


Figure 2.4: Rastrigin function: running mean termination criterion plot - convergence plots of the Rastrigin function after the running mean stopping criterion was implemented (with 15 generation comparison). It now terminates after approximately 100 generations, rather than 1000.

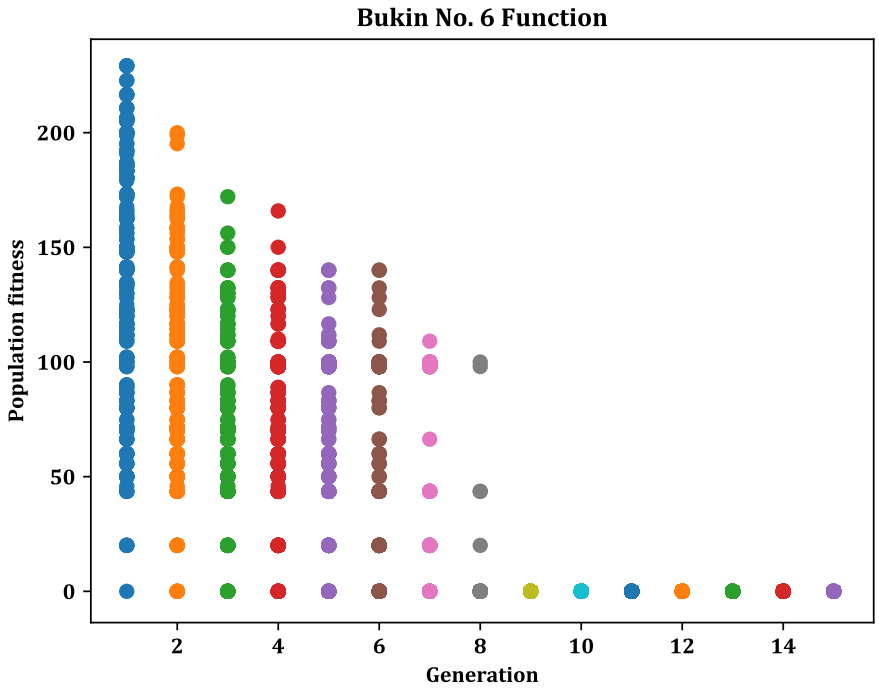


Figure 2.5: Bukin No. 6 function: running mean termination criterion plot - convergence plots of the Bukin No.6 function after the running mean stopping criterion was implemented (with 15 generation comparison). It now terminates after approximately 15 generations, rather than 1000.

As seen from Figures 2.3, 2.4, and 2.5, the running mean criterion has a drastic effect on the amount of generations run, and thus, the amount of objective function evaluations necessary. Although these four functions are relatively trivial to run to 1000 generations, as the objective function becomes more complex, such as the satellite constellation problem explored in Chapter 5, evaluations become extremely computationally draining and non-trivial.

Due to the stochasticity of GAs, another problem we encountered in our initial tests was that the population member with the best fitness in the final generation of the algorithm was not always the best member throughout the entire algorithm. Often, the best population members were accidentally eliminated through either selection, crossover, and mutation. To control this, some forms of elitism were implemented and tested.

While elitism did not seem to cause significant changes in convergence for most functions, many of the highly multimodal functions converged more quickly with the assistance of elitist recombination. This could indicate that elitist recombination could help oscillatory functions converge more easily and accurately. However, it is important to note that while convergence is faster in a generational sense, elitist recombination increases the number of objective function evaluations within a single generation. However, although elitist recombination the oscillatory functions in converging more accurately, it is important to note that for more complex functions, the quick convergence may hinder the genetic algorithm from fully exploring the search space and locating the global minimum. In general, within GAs, a careful balance must be struck between increasing algorithm speed and avoiding premature convergence.

Aside from the running mean termination criterion and elitism, different crossover and mutation schemes were also explored or implemented, leaving a variety of parameters to test within the GAs. In order to formulate a more efficient and accurate algorithm, we wanted to explore these combinations and tune parameters to determine the ideal schemes and settings moving forward. This led us to implement the Taguchi method to provide a systematic yet efficient way to test and tune parameters.

2.3.2 Taguchi Results

Given the quantity and range of the various hyperparameters involved in our algorithm, we found that a combinatorial approach to testing parameter sets would be unfeasible. This was especially evident when considering the stochasticity of GAs and the need to test a multitude of trials for each experimental parameter set.

Taguchi's method of robust experimental design, as outlined previously, was employed to design experiments and reduce the amount of experimental trials necessary to draw meaningful results. The orthogonal array that was implemented consisted of 36 different experiments and can be found in Appendix A.

With the 11 different factors that we decided to test, ranging from 2 to 3 levels each, a factorial approach would have necessitated 11,664 different experiments. With the Taguchi experimental approach, we were able to run 100 trials of each experiment, a drastic difference from the 1,166,400 experiments that would have been necessary for 100 trials of each factorial experiment.

To analyze the results of the Taguchi experiments, SNRs were computed with respect to fitness values. Average run time was also computed to give some measure of the overall algorithm speed. Although robustness in algorithm convergence is most important, an overly slow algorithm is nearly as impractical as an inaccurate one. By comparing these values and performing ANOVA tests, a set of parameters ideal for minimizing fitness was

Sphere Function		
	Fitness-Geared GA	Speed-Geared GA
Fitness	0.002	0.003
Run Time (in seconds)	0.228	0.036

Table 2.1: Sphere function: fitness vs. speed - the two forms of the genetic algorithm were run ten times each and then average fitness convergence and run times were computed.

found, as well as a set that was determined to be ideal for minimizing run time. The results of these analyses on the benchmark functions are shown in Appendix A.

Using the two different ideal parameter sets, we ran two versions of our genetic algorithm. The results, shown in Tables 2.1 and 2.2 provide information on the minimum determined by the algorithm as well as the amount of time, in seconds, that it took to converge. It can be observed that for the sphere function, both algorithms converge closely to the global minimum, yet the speed-geared algorithm performed six-times faster. Meanwhile, for the Rastrigin function, the fitness-geared algorithm converges much more accurately, and although the speed-geared algorithm is nearly 9 times faster, its convergence is unreliable.

For the sphere function's fitness-geared algorithm, ideal parameters were population size 1000, tournament-style selection set at 5% of population size, uniform crossover with crossover probability 0.3, Gaussian mutation with mutation probability 0.9, elitist recombination, and the running mean termination criterion with a 5 generation comparison average with a 0.1 precision. For the speed-geared algorithm, ideal parameters were population size 250, tournament-style selection set at 3% of population size, one-point crossover with crossover probability 0.3, Cauchy mutation with mutation probability 0.9, archival elitism, and the running mean termination criterion with a 5 generation comparison average with a 0.1 precision.

For fitness, these parameters were found to be statistically significant with $p < 0.05$:

- Population size
- Crossover probability
- Mutation scheme
- Mutation probability
- Number of generations for termination criterion comparison
- Precision level

For speed, these parameters were found to be statistically significant with $p < 0.05$:

- Population size
- Tournament size
- Crossover scheme
- Crossover probability
- Mutation scheme

- Mutation probability
- Number of generations for termination criterion comparison
- Precision level
- Elitist recombination
- Archival elitism

Rastrigin Function		
	Fitness-Geared GA	Speed-Geared GA
Fitness	7.616	98.350
Run Time (in seconds)	3.158	0.362

Table 2.2: Rastrigin function: fitness vs. speed - the two forms of the genetic algorithm were run ten times each and then average fitness convergence and run times were computed.

For the Rastrigin function's fitness-geared algorithm, ideal parameters were population size 1000, tournament-style selection set at 5% of population size, uniform crossover with crossover probability 0.3, Gaussian mutation with mutation probability 0.3, elitist recombination, and the running mean termination criterion with a 5 generation comparison average with a 0.1 precision. For the speed-geared algorithm, ideal parameters were population size 250, tournament-style selection set at 3% of population size, one-point crossover with crossover probability 0.3, Cauchy mutation with mutation probability 0.3, archival elitism, and the running mean termination criterion with a 5 generation comparison average with a 0.1 precision.

For fitness, all parameters besides archival elitism were found to be statistically significant with $p < 0.05$. For speed, all parameters were found to be statistically significant with $p < 0.05$.

When tuning parameters for genetic algorithms, it is important to consider the trade-off between speed and accuracy. While it is obvious that a fast algorithm that converges poorly has little practical utility, an extremely accurate algorithm that takes days or weeks to converge is equally impractical. While the benchmark functions all had trivial run times, the objective function for the Walker constellation problem that will be discussed in Chapter 5 is non-trivial to compute.

The crucial balance between speed and robust convergence inspired the idea to implement surrogate functions to minimize GA run time without sacrificing accuracy. The next chapter will discuss our work in using different machine learning techniques to accomplish this goal.

Chapter 3

Machine Learning

Machine learning (ML) is "a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty" [27]. It is based on the idea that machines can effectively derive information from data and identity patterns with minimal human intervention. For this project, we explored three different ML techniques: support vector regression (SVR), multilayer perceptron networks (MLP), and radial basis function (RBF) networks. After implementing SVRs on our benchmark functions, we decided not to pursue them further because of their sensitivity to pre-selected, user-defined settings. The rest of this report will focus on the two neural networks, although a review of SVRs, methodology, and results can be found in Appendix B.

3.1 Background

3.1.1 Multiple Layer Perceptron (MLP) Neural Networks

A multilayer perceptron, also known as a feed-forward neural network, is one of the simplest forms of a neural network. The original perceptron was composed of three layers: a layer to receive the input, a layer to process it, and a layer to output the result. These early perceptrons were used exclusively for classification. A multilayer perceptron is composed of layers of perceptrons, and may be used for both classification and regression. For the purposes of this problem, a multilayer perceptron was used for regression.

Architecture

The base unit of a multilayer perceptron is a node, modeled on a neuron in the human brain. In general, a node takes the following form: The node receives an input $\mathbf{x} = (x_1, \ldots, x_N)$, and the dot product of the inputs and the weights $\mathbf{w} = (w_1, \ldots, w_N)$ is computed. This sum is then passed into an activation function σ. The activation of the node is then

$$a = \sigma\left(\sum_{i=1}^{n} w_i x_i\right) \tag{3.1}$$

In a feed-forward neural network, nodes are arranged into layers. The input is passed into the first layer, and the outputs from the one layer of nodes form the input for the next layer, allowing information to propagate forward through the network. The final layer contains a

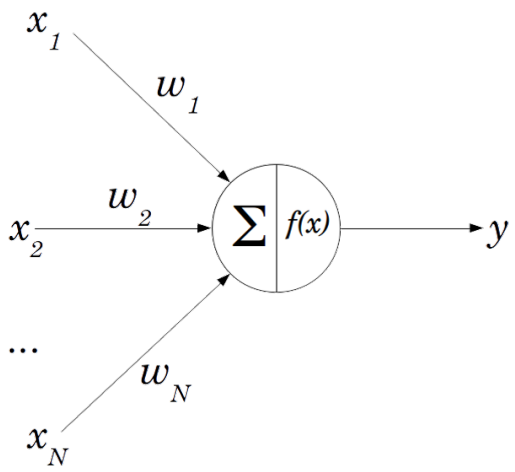


Figure 3.1: General form of a neural network node

single neuron whose activation is the output of the network. This is modeled by

$$\hat{y} = f\left(\sum_{i=1}^{L} w_i a_i\right) \tag{3.2}$$

where $\hat{y}$ is the network output, L is the number of nodes in the final hidden layer, and a_i is the activation of the ith node in the final hidden layer.

Learning

Learning refers the process of adjusting the weights of the network in order to minimize the error of the network's output. This is accomplished via backpropagation. The predictions of the neural network are computed for a set of points for which the real values are known, typically a subsection of the training set. The predicted values are compared to the real values by means of a loss function, such as mean squared error (MSE).

$$MSE = \frac{1}{p}\sum_{i=1}^{p}(\hat{y}_i - y_i)^2 \tag{3.3}$$

where i is each training point, p is the number of training points, $\hat{y}_i$ is the predicted value at each training point, and y_i is the true value of each training point. This information about the accuracy of the network's prediction is used to adjust the weights of the network according to an optimization algorithm. Over many iterations of this process, the network's loss should gradually decrease, indicating a more accurate estimation of the true value for each input.

Multilayer Perceptron Parameters

The design, construction, and tuning of a multilayer perceptron involves choices on a number of functions and hyperparamaters, which are parameters whose values are determined before applying the network with any data.

- **Number of hidden layers**: A multilayer perceptron may have one or more hidden layers. More than three layers (input, output, and one hidden layer) is referred to as deep learning [24]. There is no upper limit on the number of layers, although computational expense and speed limit the practical utility of highly deep networks in practice.

- **Number of nodes in each hidden layer**: The input layer of a neural network has the same number of nodes as elements or features of the input, and the number of nodes in the output layer likewise corresponds to the number of features in the output, or, in the case of a non-binary classification problem, the number of possible values. However, each hidden layer may be comprised of any number of nodes. Larochelle et al. [20] found that the same number of nodes in each hidden layer performed on average better than either increasing (more nodes in each subsequent hidden layer) or decreasing (fewer nodes in each subsequent layer) schemes. A larger number of nodes than required has less negative impact than too few: capturing extraneous features does not prevent predicting accurately, while a network structure which lacks enough depth to encode the complexity of the problem will struggle to provide meaninful outputs [1].

- **Optimization algorithm**: The most basic optimization algorithm for adjusting the network's weights is stochastic gradient descent and is still widely used in research [33]. Based on stochastic gradient descent, many extension algorithms have been developed over the last several decades to improve the speed of convergence. Three of the most recent are RMSProp, Adadelta, and Adaptive Moment Estimation (Adam). Of these, Adam is recommended for general cases as it has been found to slightly outperform RMSProp and Adadelta [33].

- **Learning rate**: The initial learning rate is a hyperparameter of the optimization algorithm. If the learning rate at any point is too large, it will cause the network's loss to increase rather than decrease over iterations and epochs. In general, the accepted best practice is that the ideal learning rate is within a factor of two of the smallest learning rate which causes the network to diverge (i.e. its loss to increase). Typical values range from 1 to 10^{-6} [1].

- **Mini-batch size**: The mini-batch size is the size subset of the training upon which the neural net updates each iteration. If the mini-batch size equals the size of the training set, the number of iterations and the number of epochs are the same. Theoretically, the batch size should have a direct impact only on the training time of a network; as the batch size increases, fewer updates are performed per epoch, requiring more epochs to reach a desired level of error. A large batch size allows for more intuitive parallelization. Conversely, when combined with a large learning rate, a small batch size allows for extended initial exploration of the space and introduces noise into the network's perception of the system, which can aid in avoiding overfitting and, in practice, tends to produce better results. Typical choices of values range between one and a few hundred, with 32 traditionally recommended as a starting point [1]. More recently, Masters and Luschi found that batch sizes under 32 outperformed higher values and that even for large data sets, values higher than 64 rarely improved performance [25].

- **Number of epochs**: The number of epochs is the number of full passes of the training set. It is either set initially or determined by the choices of acceptable error

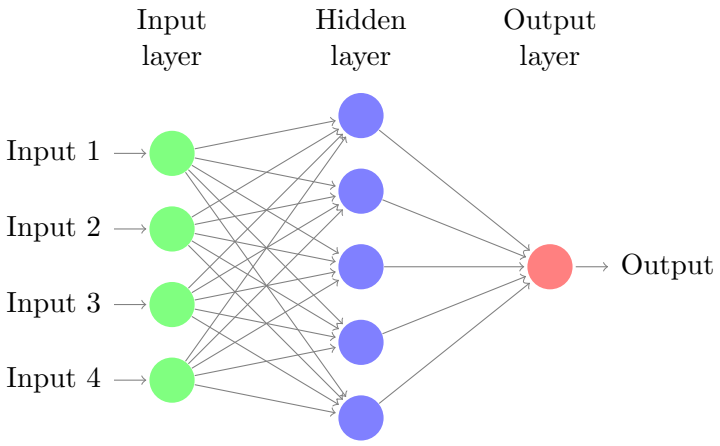
levels, batch size, and learning rate.

There are naturally many more features and parameters than those listed above which may be implemented and tuned in a neural network. For the sake of initial exploration of the suitability of neural networks for incorporation within a genetic algorithm, we limited our initial exploration to the features listed above.

3.1.2 Radial Basis Function (RBF) Neural Networks

A radial basis function (RBF) network, similar to a MLP, is also a feed-forward neural network that trains through backpropagation and can be tuned on many of the same parameters: number of nodes, optimization algorithm, learning rate, batch size, and number of epochs. However, a RBF network differs from MLP as is only has a single hidden layer. Its additional advantage over other machine learning techniques is its support for incremental training, which allows the incorporation of newly generated populations from each iteration of the genetic algorithm into training, potentially increasing overall model accuracy.

Architecture



An RBF network computes a mapping $\mathbb{R}^n \to \mathbb{R}$. It consists of one input layer, one hidden layer, and one output layer with only a single neuron. Each neuron in the input layer is a predictor variable, while each neuron in the hidden layer is the radial basis activation function. To perform regression with this network, the last neuron on the output layer computes the following equation:

$$f(x) = \sum_{i=1}^{N} w_i \varphi(x, c_i) \tag{3.4}$$

where $x \in \mathbb{R}^n$, $N \in \mathbb{N}$ is the number of components, $w_i \in \mathbb{R}$ is the weight of the activation functions, φ denotes the specific activation function used, and $c_i \in \mathbb{R}$ is the center of the activation functions.

Activation Functions

Many different functions can be used as the activation function for an RBF network activation function:

$$\varphi(x, c_i) = e^{-(||x-c_i||^2)/2\sigma^2}, \qquad \text{Gaussian}, \tag{3.5}$$

$$\varphi(x, c_i) = \frac{1}{(\sigma^2 + ||x - c_i||^2)^\alpha}, \qquad \alpha > 0, \tag{3.6}$$

$$\varphi(x, c_i) = \frac{1}{1 + e^{(||x-c_i||/\sigma^2)-\theta}}, \qquad \text{logistic}, \tag{3.7}$$

$$\varphi(x, c_i) = ||x - c_i||, \qquad \text{linear}, \tag{3.8}$$

The σ in Equations (3.5), (3.6), and (3.7) is used to control the smoothness of the interpolating function. The θ in Equation (3.7) is a bias that can be tuned. Equations (3.5), (3.6), and (3.7) are also localized radial basis functions, meaning that $\varphi \to 0$ as $||x - c_i|| \to \infty$. [38]

Because the neurons naturally have Gaussian-like receptive fields [38], the Gaussian is typically selected as the canonical RBF. Along with being compact and positive, it is also the only factorizable RBF, which makes it a very desirable candidate for hardware implementation.

The equation for the Gaussian RBF comes from the idea that we can approximate any function using a linear combination of Gaussians. Given the following Gaussian probability density function:

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.9}$$

the left-hand coefficient can be discarded, which will eventually be re-factored into the weights, and directly use the right-hand side as the activation function, setting $\mu = c_i$. This completes the construction of the simple Gaussian RBF, where the parameters, c_i and $\beta_i = \frac{1}{2\sigma_i^2}$, can be initialized to constants selected from a random distribution of the training set. These parameters are then adjusted accordingly to minimize the mean of squares loss function as the neural network proceeds to train. However, if the training set is insufficiently large or fails to capture the overall shape of the function, the model may produce unsatisfactory results.

K-Means Clustering

To improve the speed and accuracy of the neural network training, the starting values for β_i and c_i are ideally initialized to approximate the target function as closely as possible. This is achieved through clustering, a method to characterize the overall distributions and associations of a particular data set.

A common method used for clustering in the RBF Gaussian network is K-means clustering, which splits the dataset into k groups, with k being a parameter set by the user. Both Lloyd's or Elkan's algorithm achieves this partitioning in $O(knT)$ time, where n is the number of samples and T is the number of iterations [29]. After obtaining the partitioned model, we can then set c_i to be the cluster centers and

$$\beta_i = \frac{1}{2\sigma_i^2} \tag{3.10}$$

$$\sigma_i = \frac{1}{n}\sum_{j=1}^{N} ||x_j - c_i|| \tag{3.11}$$

indicating that σ_i is the average Euclidean distance between all data points that belong to a specific cluster and that cluster's center point.

3.2 Experiments & Methodology

3.2.1 MLP Neural Networks

Multilayer perceptrons were implemented with the Keras API [6] with Google Tensorflow backend. Several supplementary libraries were also used, namely scikit-learn [29], numpy, and matplotlib.

Multilayer perceptrons were tested on all 18 of the benchmarking functions. The training and testing sets were generated either uniformly within the bounds of the design space (F1 - F8, F15 - F18) or from a normal distribution with standard deviation $\sigma = 1$ and centers μ as the standard starting points given in [26] for each benchmark function (F9 - F14). In general, of the data that was generated, 70% was allocated for training and 30% reserved for validation testing.

Before training, the points and corresponding objective function values were normalized to have mean $\mu = 0$ and standard deviation $\sigma = 1$ using the `StandardScaler` function within the `scikit-learn.preprocessing` library. Other scalers and normalizers, including `MinMaxScaler`, `MaxAbsScaler`, and normalization based on the known bounds of the function were tested, but none performed appreciably differently from `StandardScaler`.

3.2.2 RBF Neural Networks

Radial basis functions (RBF) were also implemented using the Keras API [6] with Tensorflow backend. Because there was no existing built-in RBF layer, we wrote our own using a Gaussian activation function and k-means as the initializer for the μ and β. Scikit-learn [29] was used for k-means clustering and normalization, numpy was used for data wrangling, and matplotlib was used for generating graphs. To speed up testing of the parameters, we also used the `multiprocessing` Python module to run all our simulations in parallel, drastically reducing the computational time. The dataset was uniformly randomly generated, where 70% was allocated for training and 30% was allocated for testing.

We then computed the effectiveness of the radial basis function network across all of the benchmark functions. For each, we explored the following hyperparameters: number of data points, number of nodes in the hidden layer, number of epochs, batch size, the toggle for training μ, and the toggle for training the β values.

3.3 Results

3.3.1 MLP Neural Networks

For the purposes of our testing, the batch size, learning rate, and number of training points were varied. In general, for neural networks it is recommended to test parameters on a logarithmic scale, as linear increases in numeric parameters rarely demonstrate a noticeable effect and the range of possible values is wide [1].

Training Set Size

The most intuitive parameter to vary external to the neural network itself is the size of the training set. As a larger sample, a larger training set in general better reflects the overall traits of the whole population, the image of the objective function. We predicted that in some cases, some functions would be sufficiently complex that a very large number of points would required to accurately model them regardless of the internal parameters of the neural net. To that end, a MLP was trained on all eighteen canonical benchmarking functions with either 1,500 total points or 15,000 total points, and with the following parameters:

- Learning rate = 0.001
- Number of epochs = 260
- Batch size = 16
- Number of nodes per layer = 128

The exact threshold at which the training set becomes large enough to model accurately is naturally function specific, but a major improvement from increasing the training set size would be visible with 10,500 training points in comparison to 1,050 training points. More sensitive tuning could then be performed on functions demonstrating such an effect.

For twelve out of eighteen functions, 1,050 points was sufficient to predict the objective function with reasonable accuracy. For the most accurate, plotting predicted points against real values almost exactly reflected a the diagonal line $x = y$, as shown in Figure 3.2.

For others, the plots showed more variation, but retained the overall hierarchy of points. The increase in the size of the training set improved the performance of the neural network, grouping points closer to the $x = y$ line. However, the model trained on only $1,000$ points was sufficient to order the test points. On balance, the additional computation required for a larger training set outweighs an incremental gain over an already sufficient model. This is most significant for the purposes of incorporating a machine learning model into a genetic algorithm, since the absolute values of individual points are less important than their relative fitness. Preserving solely the hierarchy of points would allow the genetic algorithm to correctly converge. For two functions, the Quartic function (F4) and the Foxholes function (F5) increasing the size of the training set did affect improvement in the performance of the network. For four functions—Schwefel (F6), Rastrigin (F7), Levy (F16), and Schaffer No. 2 (F17)—a 10-fold increase in number of training points still resulted in poor predictions. At both training set sizes, the neural net was unable to learn the function and provide informative output.

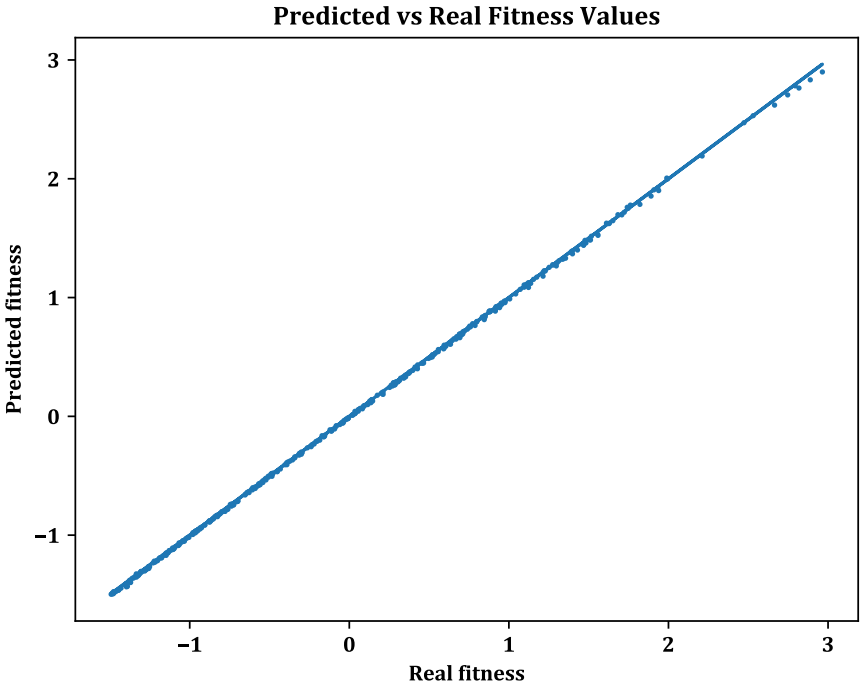


Figure 3.2: Using a set of 1000 randomly distributed points, an artifical neural network was trained on the Sphere function. The predicted fitness values are plotted against the true fitness values for 500 randomly selected points. Despite the relatively small training set, the predicted fitness values follow the $x = y$ line, reflecting almost perfect accuracy.

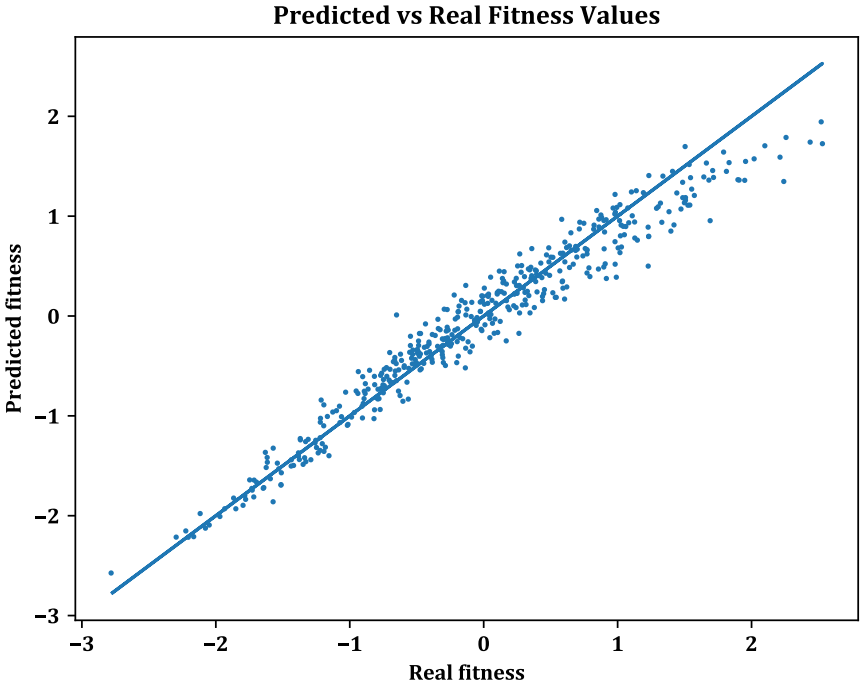


Figure 3.3: Using a set of 1000 randomly distributed points, an artifical neural network was trained on the Griewangk function. The predicted fitness values are plotted against the true fitness values for 500 randomly selected points. The predicted fitness values follow the $x = y$ line with reasonable error.

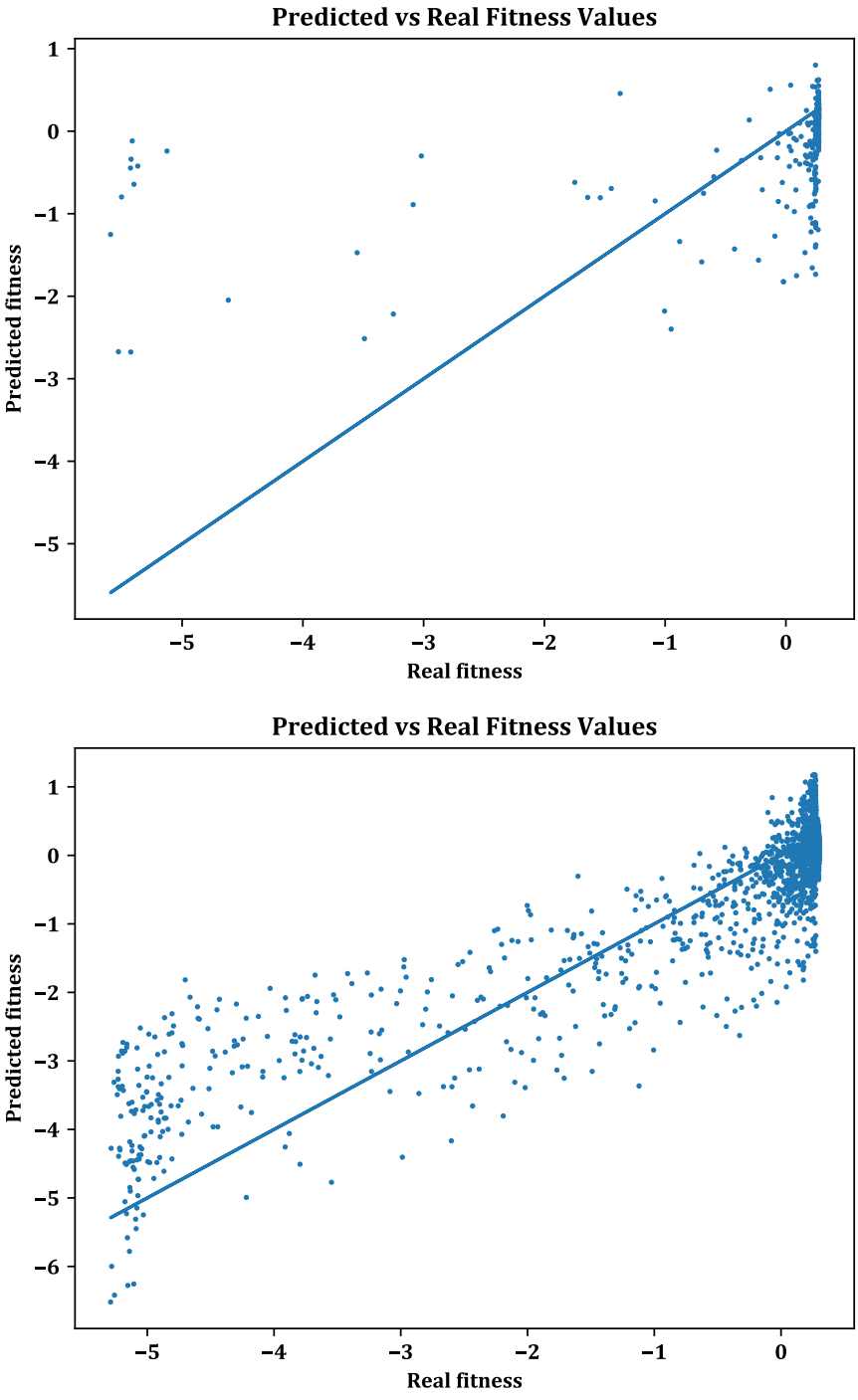


Figure 3.4: Plots of the predicted and real fitness values for two neural networks trained on the Foxholes function. The first was trained on 1,000 points and tested on 500; the second trained on 10,000 and tested on 5,000. The increase in the size of the training set substantially improved the accuracy of the neural network.

Learning Rate

As mentioned previously, typical learning rates for neural networks range from 0.1 to 1×10^{-6}. Learning rates of $0.1, 0.001, 0.0001$, and 0.000001 were tested for eighteen functions with the following remaining parameters.

- Number of points = 1,500 (1,000 training, 500 testing)
- Number of epochs = 260
- Batch size = 16
- Number of nodes per layer = 128

The resultant plot groups functions by two categories based on the results in the previous section: functions for which the neural network performs well and functions for which the neural network performs middling or poorly and may or may not be sensitive to parameter values. This division was motivated both by the different trends in both groups and by the different scales needed to plot the results accurately. In general, for the first category of

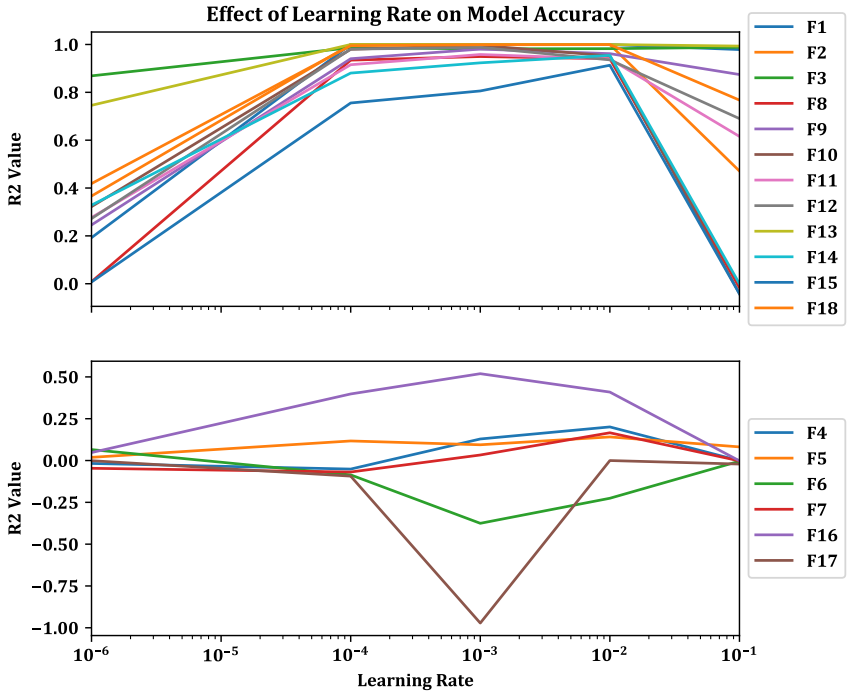


Figure 3.5: r^2 values plotted against learning rate for all eighteen benchmarking functions. Optimal values appear between 10^{-4} and 10^{-2} in most cases.

functions, values between 10^{-4} and 10^{-2} perform best, as seen in Figure 3.5. Plotting the loss of the neural network over its training, as in Fig 3.6, confirms that too large a learning rate results in large oscillations and even an increase in loss over training. On the other hand, too low a learning rate and the training of the model is considerably slowed, requiring a much longer duration to reach reasonable error.

For the second category of functions, varying the learning rate did not cause measurable improvement, with the exception of the Levy function (F16 in Figure 3.5). Despite the improvement, however, the neural network accuracy remains mediocre, peaking at slightly over 60%.

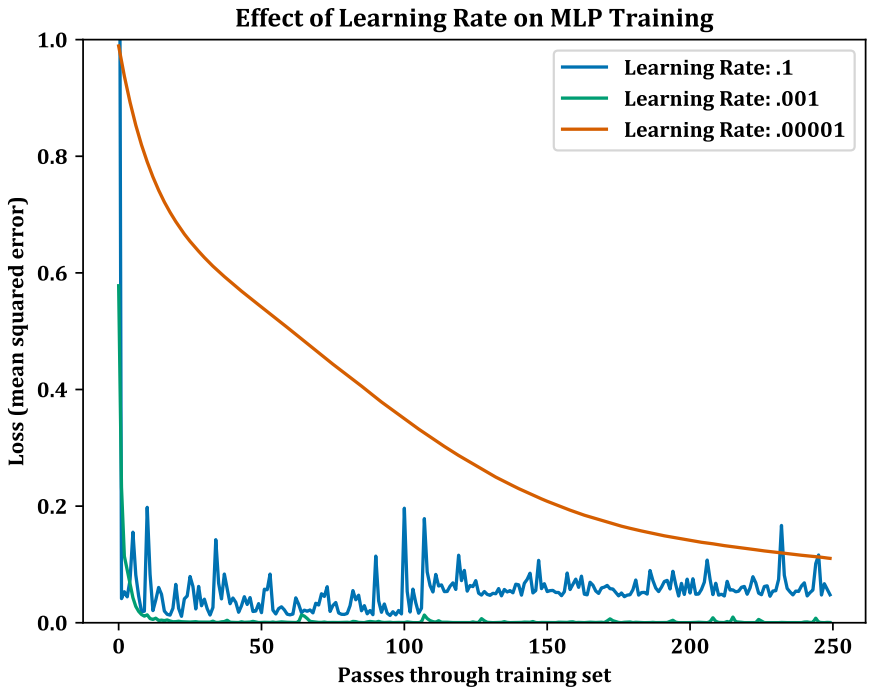


Figure 3.6: Neural network loss is plotted over epochs at different learning rates, as a visualization of training, for the Rosenbrock Function. All other parameters were kept constant. A high learning rate can result in an increase in loss over epochs, while too low a rate prevents convergence in a reasonable timeframe.

Batch Size

Theoretically, mini-batch size should affect mainly the speed of training rather than final accuracy, and therefore may be tested independently of other hyper-parameters [1]. All eighteen functions were tested at mini-batch sizes of $8, 16, 32, 64$, and 128. The results are plotted for the high-performing and medium- or low-performing functions in Figure 3.7. In general, mini-batch size shows little consistent effect. In no cases did an increase in mini-batch size result in a higher R^2 value than all lower mini-batch sizes. For some functions, an increase in mini-batch size corresponds with a slight decrease in performance, motivating the decision to retain a mini-batch size of 16 as the default.

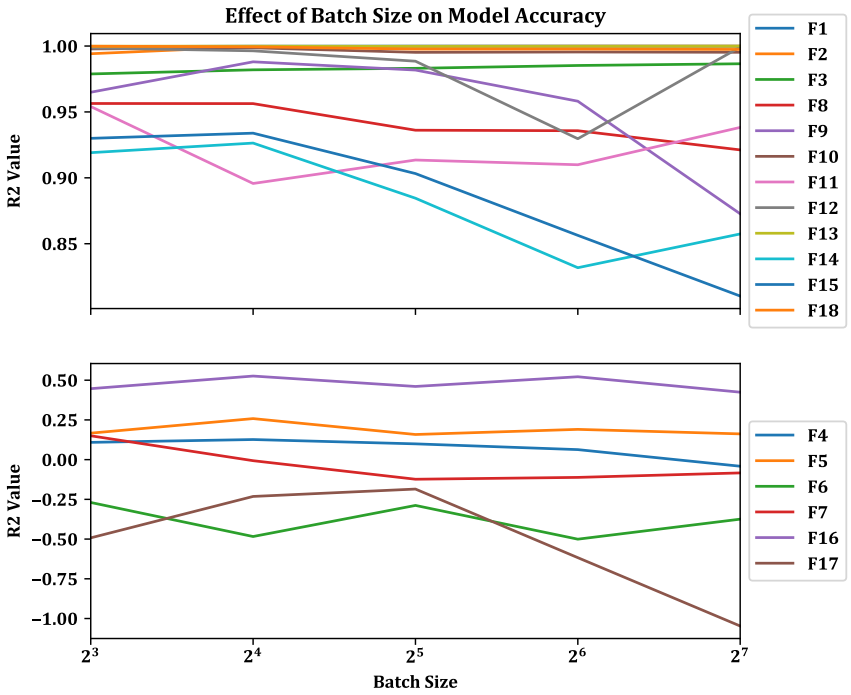


Figure 3.7: Model accuracy is plotted against different batch sizes for all 18 functions.

3.3.2 RBF Neural Networks

Optimizer Selection

We compared the effects of different optimizers from the Keras [6] package, specifically the stochastic gradient descent (SDG) optimizer, the Adadelta optimizer, the Adam optimizer, and the Nadam optimizer. Overall, Adadelta, Adam, and Nadam all demonstrate a faster convergence to low mean square errors in comparison to SDG, most likely due to the additional feature of adaptive learning rates based on a moving window of gradient updates. Adam stores another feature, the momentum, in order to more accurately correct bias, and excels over Adadelta in the more complicated multimodal functions. Nadam is a further improvement upon Adam, as it uses Nesterov accelerated gradient descent instead of the traditional gradient descent, which performs more accurate steps in the gradient direction by updating the parameters with momentum before computing the gradient.

Training Set Size

While having a large number of data points well distributed throughout the range of the function is crucial in allowing the neural network model to learn its shape, such a large amount of data is not always available and training may become computationally heavy. Selecting the optimal number of data points thus becomes a balance of accuracy versus speed. When we increase the number of data points from 1000 to 10000, the predicted data points fit much better on the $y = x$ line, where y is the predicted fitness and x is the real fitness. The model in learns comparatively faster with more training data, resulting in a much greater rate of mean square error loss decrease. However, training of the $N_{DATA} = 1000$ model is usually much slower than the $N_{DATA} = 10000$ model. For example, in the Step function, the former only took 8.11 seconds while training of the latter took 29.75 seconds—an approximate threefold slowdown.

Number of Hidden Nodes

We set up our RBF network such that the number of nodes in the hidden layer is directly equivalent to the k used in k-means clustering. This implies that if we have too few hidden nodes, we have fewer Gaussians, rendering the network difficult to train and unable to capture the overall shape of the function. In contrast, if we have too many hidden nodes, the network will be slowed and become increasingly susceptible to residual noise. We see this effect in Figure 3.8: when we increase the number of nodes from 32 to 128, the accuracy actually decreases as the points adapt a more erratic nature due to noise. This same pattern applies to our more complicated functions; regardless of the complexity of the function being approximated, we conclude that selecting a number of nodes towards the lower end contributes to higher accuracy.

Number of Epochs

Changing the number of epochs directly affects the quality of the model predictions as this parameter influences how long we leave the network to train. The correlation is not a linear one, however; in most of our benchmark functions, the output of the loss function plateaus around the $100 - 200$ epoch count. Figure 3.9 demonstrates the effects of increasing the number of epochs from 100 to 250 for the Bukin No. 6 function. From the difference plots, we see that the model training on 100 epochs does not differ significantly from one training on 250 epochs, as both demonstrate a convergence of the fitness differences around 0 (note that the graph with 100 epochs is scaled slightly differently than the graph with 250 epochs). We can thus set the training threshold to around 100 epochs without substantially sacrificing the goodness of the model predictions.

Batch Size

Depending on the number of hidden nodes we set for a particular function, we decreased and increased our batch sizes accordingly, although we did generally work with a range from 8 to 32, incremented in powers of 2. There was no distinct pattern in the relationship between batch sizes and quality of predictions, indicating that the ideal batch size would unfortunately need to be manually tuned by the user. In Figure 3.10 and 3.11, we see the effects of adjusting batch size on the Step function: a batch size of 1 appears to introduce too much noise, while a batch size of 64 appears to reduce the stochasticity of the gradient descent by more than is necessary. The optimal batch size, in this case, would lie somewhere between $8 - 16$, with 8 performing better at lower fitness values and 16 performing better at higher fitness values.

β and c Toggles

Although the initialized values for β and c were already set through k-means clustering, their respective degrees of influence on the model prediction can be adjusted by the assignment of weights. By toggling the training for c and β on/off, we can then control these weights, allowing the model to acquire a better fit. However, care needs to be taken while training with these weights—because the accumulated mean square errors may be extremely large while training β and c combined with a high number of hidden nodes, NaNs will sometimes be generated.

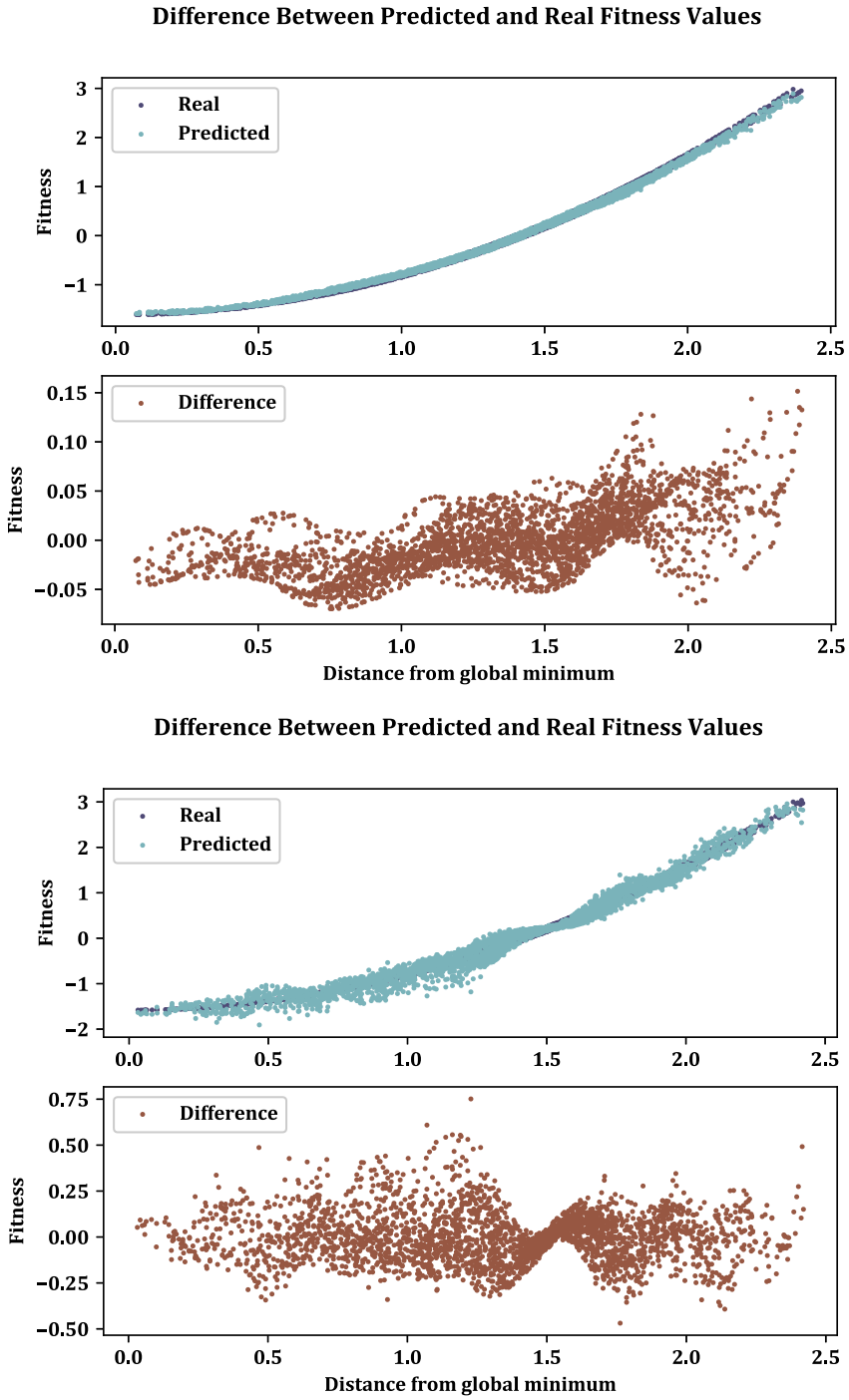


Figure 3.8: RBF Effect of hidden nodes on Sphere accuracy: changing number of hidden nodes from 32 (top) to 128 (bottom) on the Sphere function decreased the overall model accuracy as more residual noise was introduced to the model.

Handling NaNs

NaNs can occur in the loss function due to a variety of reasons: invalid values in the original dataset, incompatibility between the optimizer used and the problem type, and learning rates that are too large, causing an enormous mean square error that causes overflow. These issues may be resolved by running `df.isnull().any()` to check for NaNs or inf in the dataset, switching optimizers usually away from stochastic gradient descent, decreasing the learning rate by powers of 10, or decreasing the number of neurons by powers of 2.

Benchmark Function Analysis

Figure 3.12 shows the loss rates across all of the benchmark functions. These models were constructed with the following parameter ranges: 10000 data points, $32-128$ hidden nodes, 100 epochs, a batch size of $4-8$, center and β training mainly toggled on. The learning rates were not modulated from their default initial values, as we assumed they would eventually adapt with the model. The line plot demonstrates that other than the Schwefel (F6), the loss rates of the rest of the functions appear to plateau around the 20 epoch mark. From these, most of the convergence occur around a MSE of 0.0, with the only exception being the Schaffer No. 2 function (F17), which converged around the 0.08 mark.

The loss rate of the Schwefel function at a comparatively high value of 0.70 points to two conclusions: firstly, the structure of the function itself renders it a difficult function to approximate and secondly, the parameter settings we have chosen are not well adapted to model the function. We note that the other two machine learning methods we investigated, MLP and SVR, also produced similarly unsatisfactory results for this function.

Overall, the RBF performed particularly well on the unimodal equations, namely the Sphere (F1), Rosenbrock (F2), and Step (F3). It also excels at approximating some of the multimodal functions with discretized inputs, namely the Ackley (F15) and Bukin No. 9 (F18). However, it did fail to capture the intricacies and escape the local optima of the other continuous multimodal functions, particularly the Foxholes (F5) and Schwefel (F6), despite the Foxholes having a very low loss rate towards the end of the training session. It is also worth noting that the RBFs performs comparatively better on the Quartic (F4) function as compared to the MLP and SVR models.

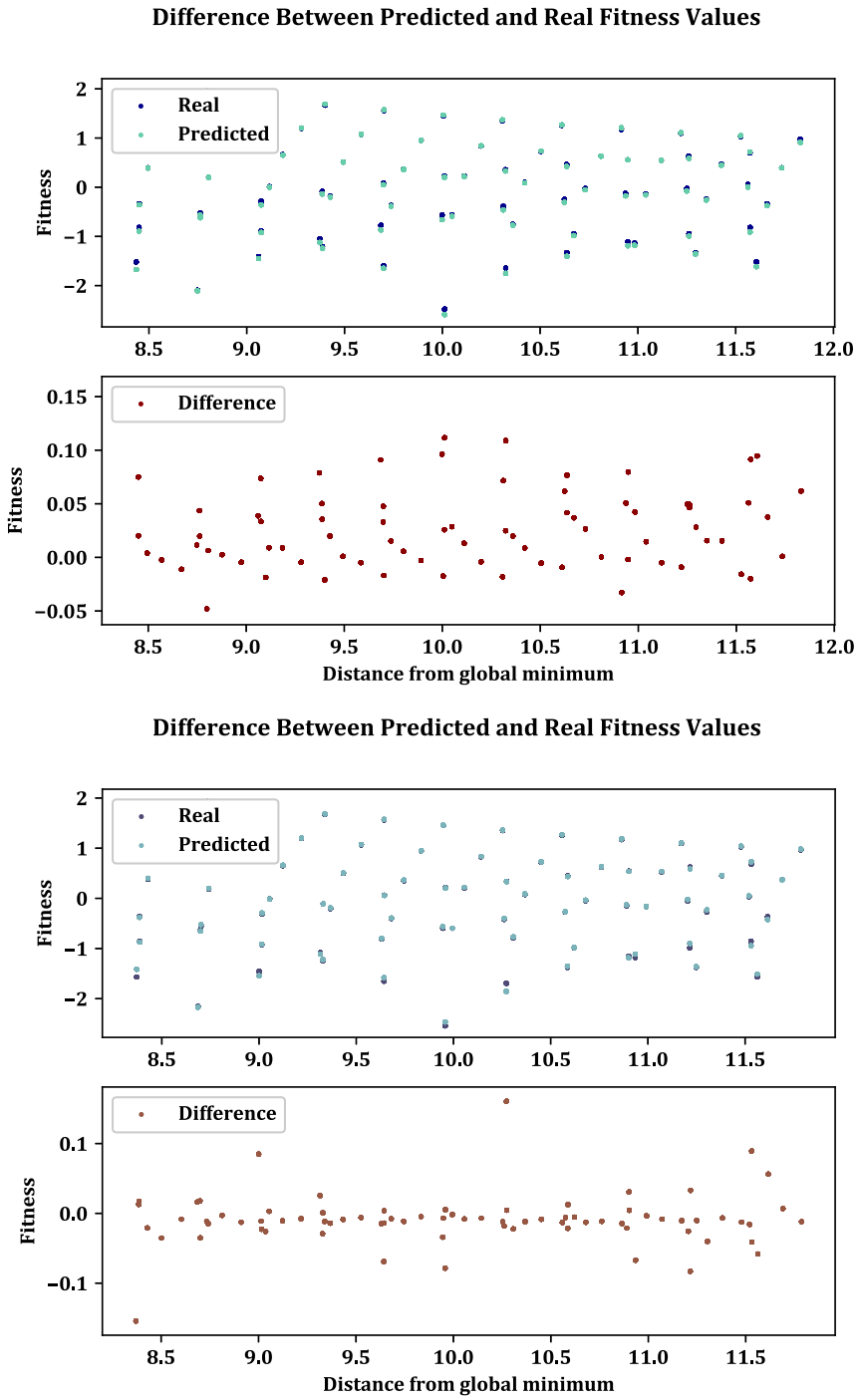


Figure 3.9: RBF Effect of epoch number on accuracy: changing number of epochs from 100 (top) to 250 (bottom) on the Bukin No. 6 function does not significantly impact the overall model accuracy.

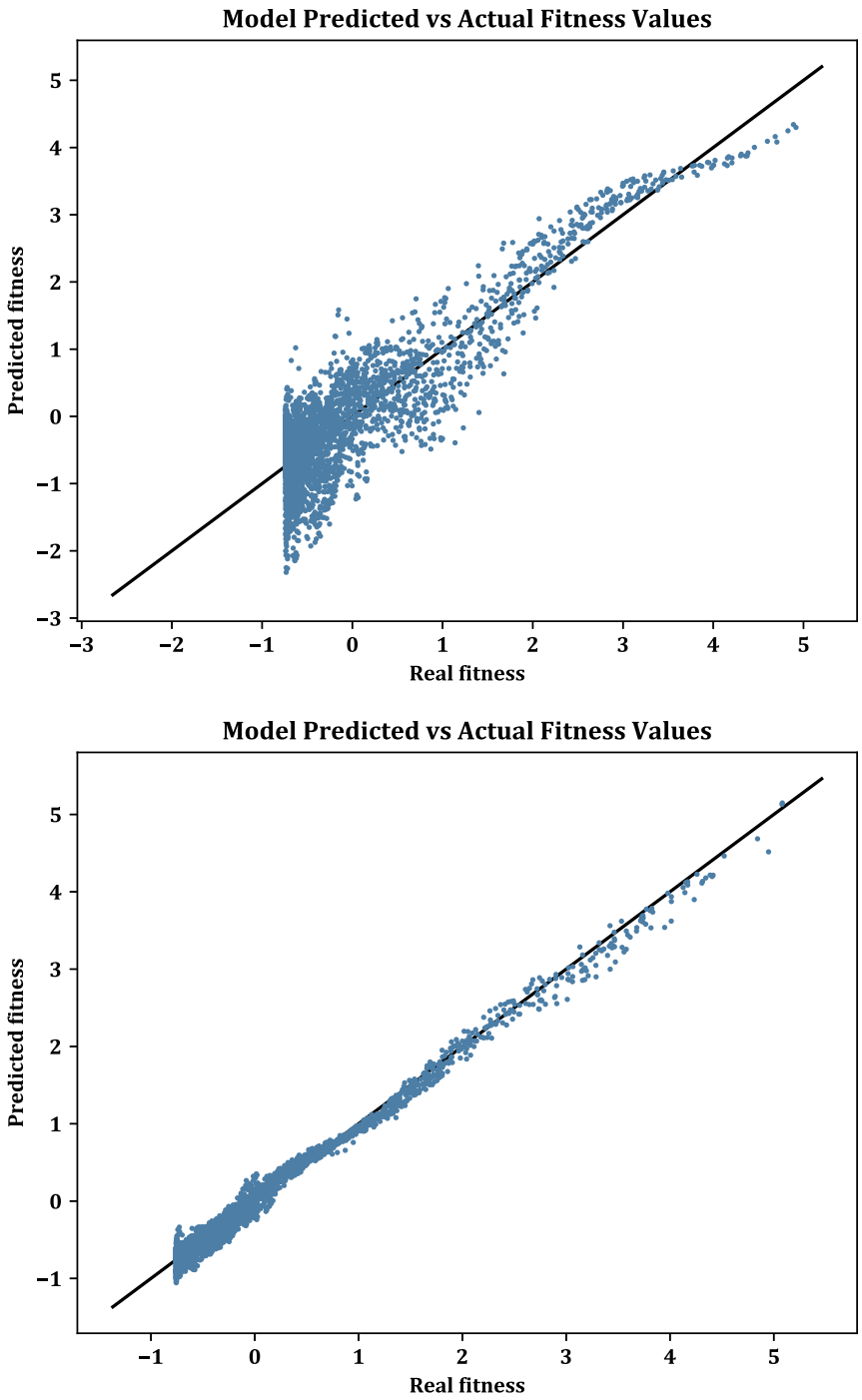


Figure 3.10: RBF Effect of batch size on accuracy (a): changing batch size from 1 (top) to 8 (bottom) on the Rosenbrock function increases model accuracy.

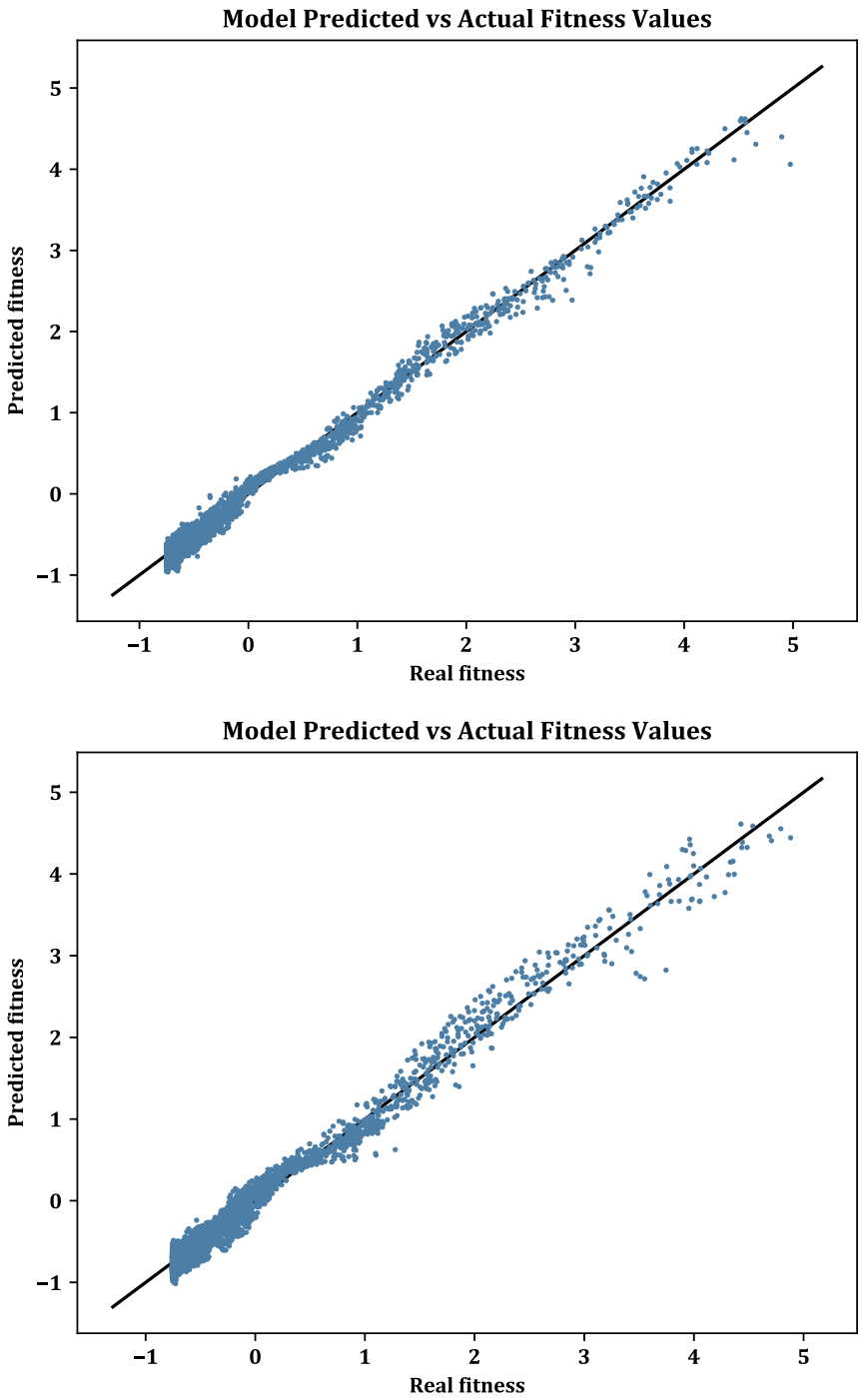


Figure 3.11: RBF Effect of batch size on accuracy (b): changing batch size from 16 (top) to 64 (bottom) on the Rosenbrock function decreases model accuracy.

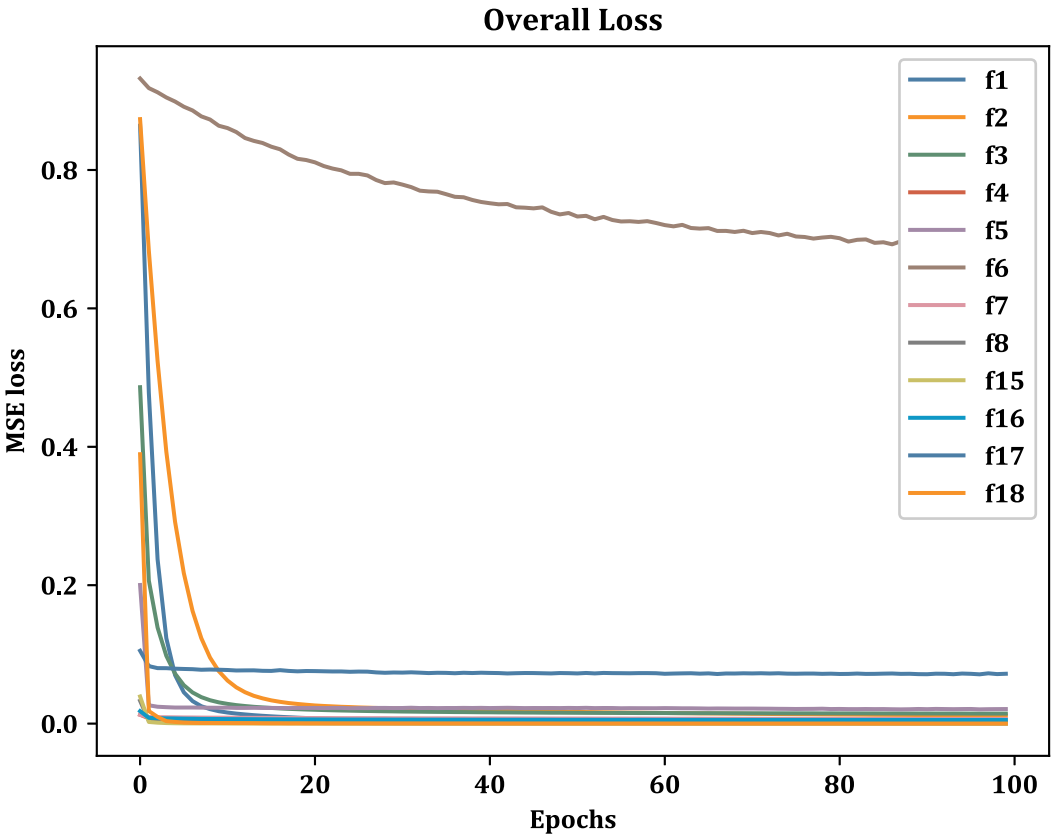


Figure 3.12: RBF mean square error loss rates: recorded across all benchmark functions over 100 epochs

Chapter 4

Machine Learning Genetic Algorithms

4.1 Background

4.1.1 Surrogate Models in Optimization

Surrogate models are a well-established method for improving the run time cost of optimization algorithms for complex, computationally expensive optimization problems [18], and have been found to be effective in improving algorithm speed [4]. While surrogates may be used for estimation of the distribution of potential solutions or to create a reverse mapping from the objective space to the input space, the most straightforward and established method is objective function estimation.

Just as there exists a trade-off between speed and accuracy of convergence in a choice of genetic algorithm parameters, the complexity of a surrogate model significantly affects the speed of implementation and usage. A mismatch between the complexity of the surrogate model implemented and the complexity of the objective function may decrease efficacy. One classical surrogate model is a polynomial model [18], yet for complex objective functions, such as Foxholes (see Table 1.1 in Chapter 1, a polynomial model would struggle to approach a good approximation. Similarly, Kriging has been used widely as surrogate, but suffers from the curse of dimensionality with large inputs or a large set of samples. As simpler models are in general more efficient both to create and to use, it is desirable to use the least complex model necessary in order to avoid needless computation and thereby maximize the speed increase from introducing a surrogate.

4.1.2 Model Management in Genetic Algorithms

The balance between speed and predictive accuracy inherent in a choice of surrogate must directly impact the manner in which it is implemented within an optimization algorithm. It is therefore necessary to carefully manage surrogate usage to ensure both reliability and speed in algorithm convergence. Model management strategies for surrogate functions within genetic algorithms are typically generation-based, population-based, or individual-based [4]. Surrogate models may alternate with the true objective function, may be used independently, or may filter the population for poor individuals before the true objective function is applied, among other approaches. A framework for implementing a surrogate model within a genetic algorithm, using a complete switch to the surrogate, is shown in

Figure 4.1.

4.1.3 Ensemble

To enhance the generalizability of surrogate model predictions, an ensemble of neural networks was implemented. An ensemble is a collection of surrogate models, trained independently. Their predictions are then considered as a unit with an averaging or weighted averaging procedure typically implemented to produce a single, final prediction. Since it has been demonstrated that different types of surrogate functions perform well in different situations, ensembling enables the strengths of certain surrogates on a specific problem to compensate for the weaknesses of other models [11]. Overall, this has been shown to provide a much more robust approach to surrogate approximation of complex objective functions [11].

In this report, for the estimation of the benchmark functions and DOP, different ensembles of MLP and RBF neural networks were implemented.

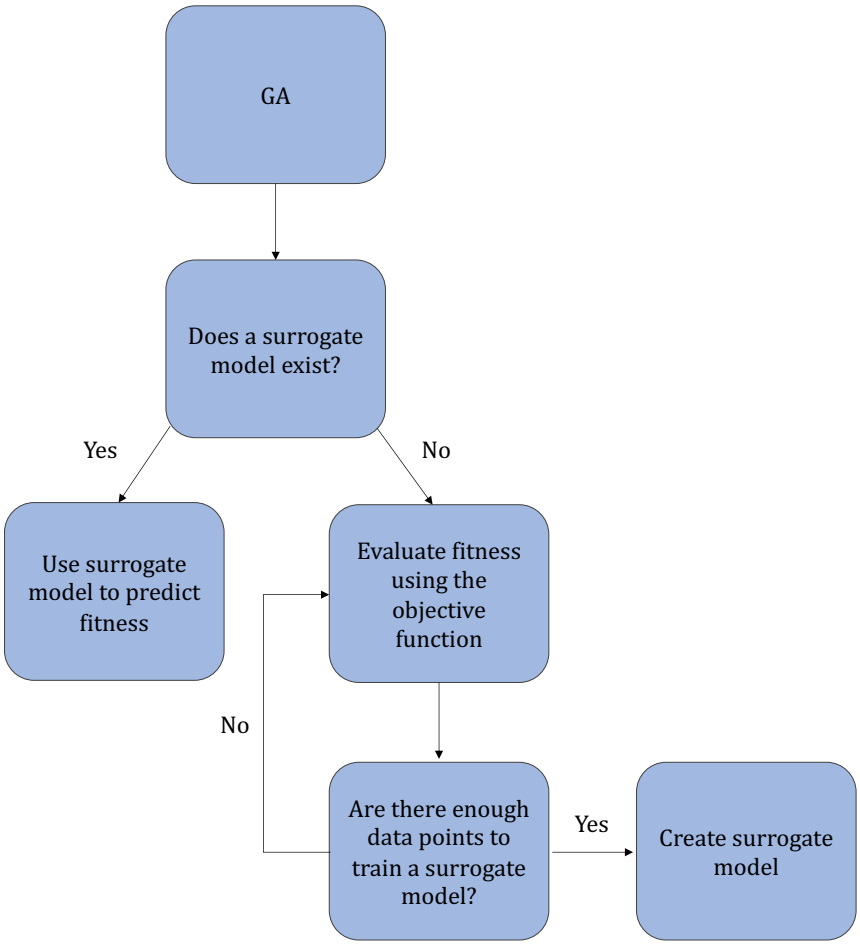


Figure 4.1: Outline of surrogate function usage within a genetic algorithm.

4.2 Experiments & Methodology

4.2.1 Model Management

Training Delay

In a genetic algorithm, as the distribution of individuals in the population evolves over generations, it is important that the surrogate model maintain predictive power in later generations. Based on the convergence plots of the implemented genetic algorithms, there is typically a steep decline in population diversity within the first few generations, greatly narrowing the search space. While maintaining a reasonable level of diversity is important for avoiding premature convergence on a false minimum, it is also helpful to exploit this smaller search space when training surrogate models. Due to this, in the MLGA, a neural network training delay of two to three generations is implemented to allow the GA population to somewhat stabilize before allowing the neural networks to intake training points. Although this training delay effectively translates to more evaluations of the true objective function, it also greatly aids the training and robustness of the neural networks' predictions over time.

Handling Model Inaccuracy

For many functions, in particular those that are high-dimensional or highly multimodal, a precise neural network surrogate requires an infeasible number of training points. While implementing the surrogate model would likely still introduce a significant increase in speed, this approach is ill-equipped to problems requiring a solution within hours, rather than days or weeks. In initial trials of machine learning on optimizing our benchmark functions, we observed that the models tended to preserve the overall hierarchy of points even in cases where they were inaccurate. Essentially, although the ML model predictions of individuals' fitnesses differ significantly from their true values, in general the ML predicts the correct individuals as being the preferred solutions to the objective function. That is, individuals that are fitter with the true objective function are, in general, also fitter with the surrogate. Evaluating the final population of the genetic algorithm enables the algorithm to leverage the efficiency of a ML surrogate to converge faster, while still resulting in an accurate fitness value for the final solution.

The forms of model management listed above are not exhaustive. Many other forms of model management were considered, in particular error estimation and model retraining. However, due to time constraints, they were not implemented. More information regarding some of these techniques can be found in Chapter 6.

4.2.2 Ensemble Creation

To generate a neural network ensemble, the bagging algorithm [2], which uses aggregated bootstrap sampling is used to create different training sets for the neural networks. Effron and Tibshirani provide background on bootstrapping [8] and Breiman provides an overview of how and why bagging works [2]. In neural network ensembles, boosting algorithms are also commonly used, although the neural networks in these ensembles must be generated sequentially [40], restricting future opportunities to train neural networks in parallel.

- **Bootstrap Sampling:** In a simplistic form, bootstrap sampling uses an original dataset to create a multitude of smaller data sets. First, a number of data sets is

specified, along with a sample size less than the original sample. A sample is randomly drawn from the original population and then replaced, and the process repeats until the small data set is filled. Then, the entire process is repeated until the specified number of data sets is generated. From the bootstrapped data sets, a statistic can be generated and then averaged across all data sets to estimate a parameter of interest about the entire population.

In bagging, the bootstrapped data sets are each used to train their own neural networks. For specifics on how each neural network is trained, refer back to Chapter 3. As described by [2], although bagging can degrade the performance of stable procedures, it also has the ability to help optimize unstable procedures. This is especially helpful in controlling the variability stemming from stochastic processes in GAs.

4.2.3 Ensemble Usage and Control

Classical ensemble methods take an average of the predictions of each surrogate model to formulate a final prediction. These were implemented, as well as a weighting procedure, as described by [11] to determine the influence of each neural network on the final prediction. Selection of weights should reflect confidence in the surrogate model as well as control the influence of bad models on the final prediction. In the MLGA implemented in this report, weights were determined as follows:

$$w_i{}^* = (E_i + \alpha E_{avg})^\beta \tag{4.1}$$

where E_i is defined as the square root of mean squared error ($RMSE_i$), and E_{avg} is

$$E_{avg} = (\sum_{i=1}^{N_{SM}} E_i)/N_{SM}; \quad \beta < 0, \alpha < 1 \tag{4.2}$$

where N_{SM} is the number of surrogate models, α and β are additional parameters affecting weight. Increasing α gives more importance to averaging than to individuals. Increasing β does the opposite, emphasizing the importance of individual surrogates more than the averaging processes [11]. Next, w_i is calculated,

$$w_i = w_i{}^*/(\sum_i w_i{}^*) \tag{4.3}$$

where w_i is the weight of each neural network, and the sum of the weights is 1. After obtaining these weights, each neural network's prediction is multiplied by its weight, and these products are summed to produce a final prediction value.

4.2.4 Ensemble Parameters

The MLGAs in this section consist of 10 neural networks. Three different ensemble models are tested: MLP only, RBF only, and a MLP + RBF hybrid. The training data consisted of three generations of objective function calculations (after a generation delay of either size 2 or size 3, function dependent, to offset the initial instability of the population). The GA had the following hyperparameters, many of which were determined through the Taguchi method, as specified in Chapter 2.

- Population size = 250
- Tournament Size = 12
- Crossover probability = 0.3
- Crossover operator = Uniform
- Mutation probability = 0.9
- Mutation operator = Gaussian
- Precision = 0.1
- Number of generations for running mean termination criterion = 5
- Maximum allowed generations = 1000

4.3 Results

4.3.1 Ensemble Models

The various ensemble models were tested on the following selected benchmark functions: Sphere (F1), Rosenbrock (F2), Step (F3), Foxholes (F5), four-dimensional Schwefel (F6), four-dimensional Griewangk (F8), Powell Badly Scaled (F12), Gulf Research and Development (F13), Schaffer No. 2 (F17), and the Bukin No. 6 (F18). These functions were selected because they share characteristics with our applied Walker constellation problem. However, this section primarily focuses on the Schwefel and the Griewangk functions since these are more similar to the objective function for dilution of precision.

4.3.2 Weighting Procedure

Figures 4.2 and 4.3 highlight the average final fitnesses for the three different ensemble models, as well as a breakdown of the two different averaging procedures that were implemented. It can be observed that for the Schwefel and four-dimensional Griewangk functions, in many cases, unweighted averaging performs better than weighted averaging. However, in the case of genetic algorithms, which are highly stochastic, weighted averaging allows us to penalize poor surrogate outliers that might throw off an entire ensemble in a non-weighted average.

4.3.3 Conclusions

For these functions, Levene's test for homogeneity of variance was conducted. For the Schwefel function, it was found that the variance of at least one of the ensemble models had a statistically significant difference from the others ($\alpha = 0.05$). From Figure 4.2, we can see that the ensemble composed only of RBF neural networks (both weighted and unweighted) performed better than the other models, perhaps with the exception of the unweighted MLP + RBF hybrid ensemble. Other functions that demonstrated statistically significant differences were the Foxholes and the Schaffer No. 2 functions.

On the other hand, when Levene's test was performed on the Griewangk function, no ensemble model was shown to have be significantly different from the others. In general, although hybrid ensemble models are shown to be more robust and adaptable to a variety of

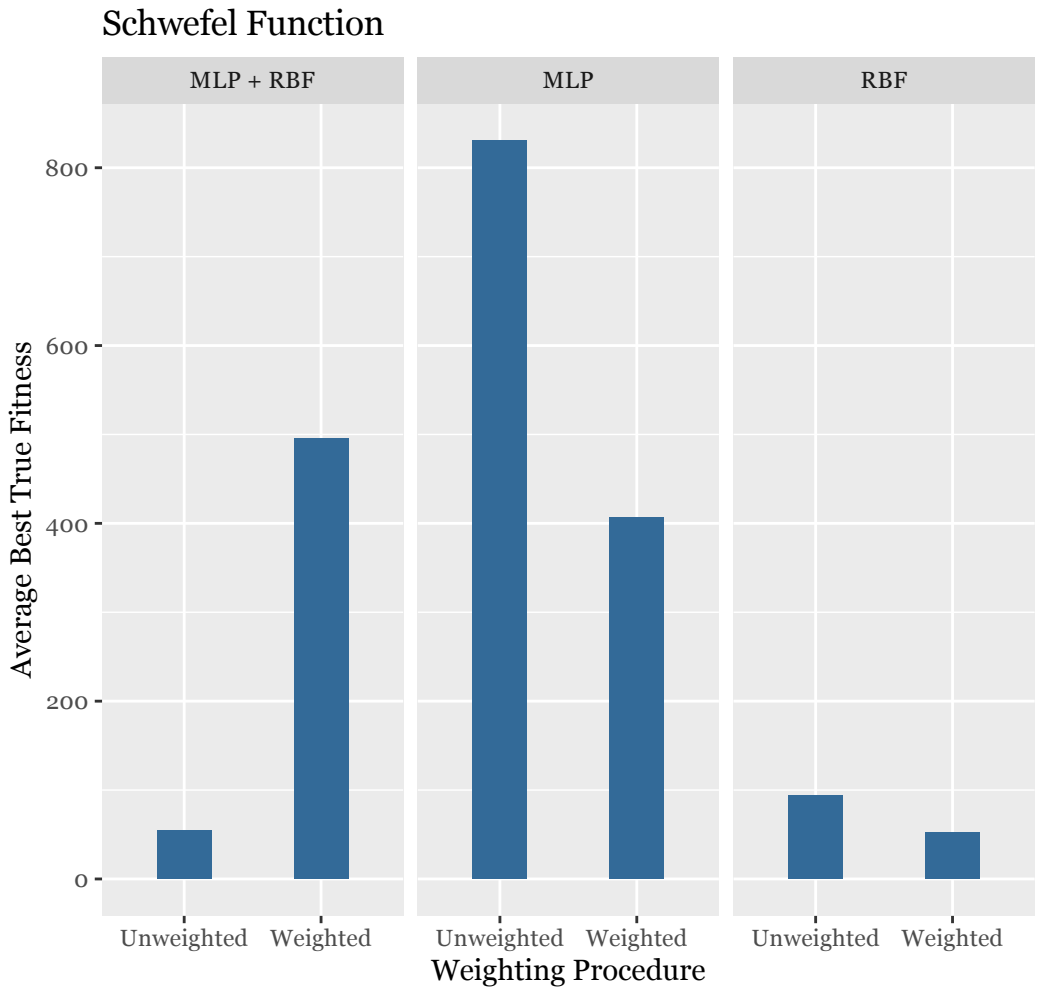


Figure 4.2: Schwefel function: ensemble results - average best fitness of different neural ensembles, tested on a four-dimensional version of the Schwefel function. The true minimum is 0, therefore, smaller bars represent better ensemble performance.

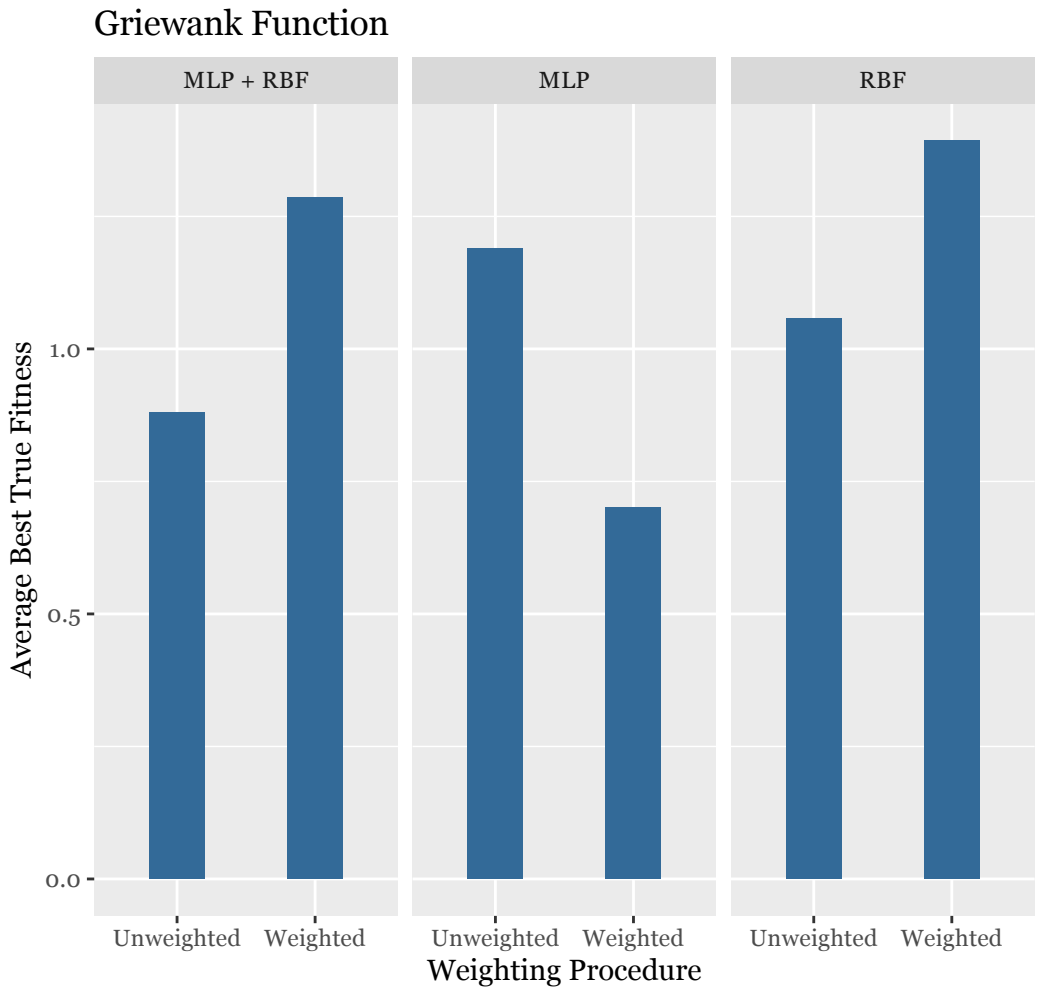


Figure 4.3: Rastrigin function: ensemble results - average best fitness of different neural ensembles, tested on a four-dimensional version of the Griewangk function. The true minimum is 0, therefore, smaller bars represent better ensemble performance.

objective functions [11], across the benchmark functions, the best ensembles were typically either MLP-only or RBF-only. In the next chapter, we tested all three ensemble models on the dilution of precision objective function to determine which would be the ideal choice moving forward. We also continued implementing weighted averaging, with more tuning done on the α parameter, which controls the averaging penalty for poor surrogates within the ensemble. Although the weighted averaging was not demonstrated to outperform unweighted averaging in all procedures, the option to tune α presents more opportunity to improve robustness than an unweighted ensemble. In general other applied problems, if it is too difficult to test and tune different ensemble models, the hybridized-neural model remains an acceptable and robust option.

Chapter 5

Dilution of Precision

5.1 Background

5.1.1 Walker Constellations

A Walker Constellation is a formation of satellites defined by three parameters: T, P, and F [36].

- T is the number of satellites in the constellation.
- P is the number of planes, or the flat, disk-shapes of the orbits.
- F is a phase parameter that gives the relative position of satellites within the same plane with value $0 \leq F \leq (P-1)$

Each plane has the same number of satellites, S, such that $S = T/P$. All orbital planes have the same inclination, or angle between the orbital and the reference planes (the reference plane is often the equatorial plane). This angle is measured at the ascending nodes of each orbit. A node is one of two points where the orbital plane intersects the reference plane, and the ascending node is the node at which is does so while the orbit is moving north. Walker defined a unit, the "Pattern Unit" (PU) to be $360°/T$. The ascending nodes of the orbits in a Walker constellation are placed every S PUs. When a satellite in a certain plane is at its ascending node, a satellite in an adjacent plane has traveled F PUs since passing through its respective ascending node.

5.1.2 Error Analysis

Pseudorange Measurements

GPS range measurements are known as pseudoranges, and a user may determine his or her position from four or more simultaneous pseudorange measurements [19, 22]. The formula to compute a pseudorange measurement given the location of the user, or receiver, and the location of the satellite is

$$PR_i = \sqrt{(X_i - X)^2 + (Y_i - Y)^2 + (Z_i - Z)^2} + cdt$$

where (X, Y, Z) is the position of the receiver and (X_i, Y_i, Z_i) is the position of satellite i, for $i \geq 4$. c is the speed of light in a vacuum, and dt is the clock offset, or the discrepancy between the satellite's time and the receiver's time.

Since this equation is non-linear, we apply linearization by instead trying to find the difference in position between an assumed location for the receiver and the receiver's actual location. Recall that we we have at least four satellites, so we can represent all of the equations we plan to solve simultaneously in matrix form:

$$A\boldsymbol{x} = \boldsymbol{b}$$

with

$$\boldsymbol{x} = \begin{bmatrix} \Delta X \\ \Delta Y \\ \Delta Z \\ cdt \end{bmatrix}$$

$$A = \begin{bmatrix} \frac{(X_1 - X)}{R_1} & \frac{(Y_1 - Y)}{R_1} & \frac{(Z_1 - Z)}{R_1} & -1 \\ \frac{(X_2 - X)}{R_2} & \frac{(Y_2 - Y)}{R_2} & \frac{(Z_2 - Z)}{R_2} & -1 \\ \frac{(X_3 - X)}{R_3} & \frac{(Y_3 - Y)}{R_3} & \frac{(Z_3 - Z)}{R_3} & -1 \\ \frac{(X_4 - X)}{R_4} & \frac{(Y_4 - Y)}{R_4} & \frac{(Z_4 - Z)}{R_4} & -1 \\ \vdots & \vdots & \vdots & \vdots \\ \frac{(X_k - X)}{R_k} & \frac{(Y_k - Y)}{R_k} & \frac{(Z_k - Z)}{R_k} & -1 \end{bmatrix}$$

where

$$R_i = \sqrt{(X_i - X)^2 + (Y_i - Y)^2 + (Z_i - Z)^2}$$

and k is the number of satellite range measurements. The vector $\boldsymbol{b}$ is the assumed position of the receiver.

The linearization of the problem allows us to apply the least-squares method to solve for the receiver's position and the clock offset. Our non-linear equation $A\boldsymbol{x} = \boldsymbol{b}$ becomes

$$\boldsymbol{x} = (A^T A)^{-1} A^T \boldsymbol{b}$$

once we use the least-squares method to solve for $\boldsymbol{x}$. Recall that $\boldsymbol{b}$ is the assumed position, and multiple iterations through this equation while updating the assumed position with the most recently found user position may be required if the initial assumed position is too inaccurate.

To quantify the effects of error on the value of $\boldsymbol{x}$, we compute the covariance matrix of $\boldsymbol{x}$ by using with the law of propagation of error:

$$C_{\boldsymbol{x}} = \Big[\big(A^T A\big)^{-1} A^T\Big] C_b \Big[\big(A^T A\big)^{-1} A^T\Big]^T = \big(A^T C_{\boldsymbol{b}}^{-1} A\big)^{-1} \tag{5.1}$$

where $C_{\boldsymbol{b}}$ is the covariance matrix of $\boldsymbol{b}$.

Assuming that all errors are the same for all measurements with standard deviation σ, $C_{\boldsymbol{b}} = I\sigma^2$ where I is the identity matrix. From Equation (5.1), we get

$$C_{\boldsymbol{x}} = \big(A^T C_{\boldsymbol{b}}^{-1} A\big)^{-1} = \Big(A^T \big(I\sigma^2\big)^{-1} A\Big)^{-1} = (A^T \sigma^2 A)^{-1} = (A^T A)^{-1}\sigma^2$$

Dilution of Precision

Dilutions of Precision (DOP) are metrics characterizing the errors caused by satellite geometry from the user's perspective [19, 22, 35]. If we designate the matrix $(A^T A)^{-1}$ as Q then

$$Q = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{xz} & \sigma_{xt} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{yz} & \sigma_{yt} \\ \sigma_{xz} & \sigma_{yz} & \sigma_z^2 & \sigma_{zt} \\ \sigma_{xt} & \sigma_{yt} & \sigma_{zt} & \sigma_t^2 \end{bmatrix}$$

and we can then define the different types of DOP as in Table 5.1. Table 5.2 reviews the quality of DOP values.

Varieties of DOP Values		
Name	**Measurements Impacted**	**Formula**
Horizontal (HDOP)	latitude, longitude	$\sqrt{\sigma_x^2 + \sigma_y^2}$
Vertical (VDOP)	altitude	$\sqrt{\sigma_z^2}$
Position (PDOP)	latitude, longitude, altitude	$\sqrt{\sigma_x^2 + \sigma_y^2 + \sigma_z^2}$
Temporal (TDOP)	time (clock variance)	$\sqrt{\sigma_t^2}$
Geometric (GDOP)	latitude, longitude, altitude, time	$\sqrt{\sigma_x^2 + \sigma_y^2 + \sigma_z^2 + \sigma_t^2}$

Table 5.1: The different varieties of DOP values that can be found from the covariance matrix Q.

DOP Value	**Rating**
1	Ideal
1-2	Excellent
2-5	Good
5-10	Moderate
10-20	Fair
>20	Poor

Table 5.2: Meaning of DOP Values

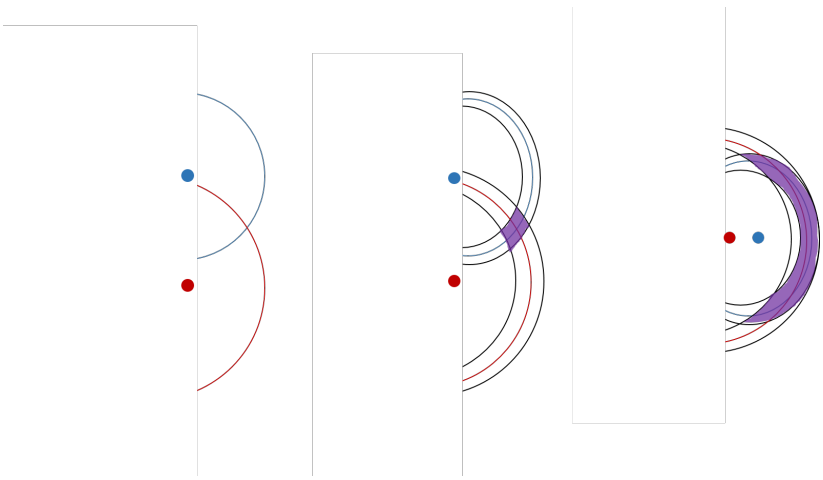


Figure 5.1: Illustration of DOP: The red and blue curves represent satellite ranges, and the black curves around them in the second and third figures are the error bounds for each range. In the first image, the desired location can be found at the intersection of the two ranges. In the second figure, when error bounds are introduced, the desired location may be found anywhere within the purple area. Here, dilution of precision (DOP) is relatively low. The third figure demonstrates a satellite geometry in which the area of uncertainty is much larger, resulting in a higher DOP value.

5.1.3 Satellite Geometry

The calculation of any DOP value requires measurements from at least four satellites. As a result, we must check that four satellites are in view from the perspective of the user at the time that we wish to calculate DOP. Here, we describe the trigonometry required to do so.

First, we assume a spherical Earth. Taking the center of the Earth to be at point (0,0,0), fix a user at position (x_u, y_u, z_u) on the surface of the Earth and consider a satellite at position (x_s, y_s, z_s). Let $\boldsymbol{a}$ be the vector from (0,0,0) to (x_u, y_u, z_u), and $\boldsymbol{b}$ be the vector from (0,0,0) to (x_s, y_s, z_s). Then let

$$\boldsymbol{x} = \boldsymbol{b} - \boldsymbol{a}$$

represent the vector from (x_u, y_u, z_u) to (x_s, y_s, z_s). The vector $\boldsymbol{a}$ is orthogonal to a plane, which we will call L, that is tangent to the surface of the Earth.

Define θ as the angle between $\boldsymbol{a}$ and $\boldsymbol{x}$. Then

$$\theta = \arccos\left(\frac{\boldsymbol{a} \cdot \boldsymbol{x}}{\|\boldsymbol{a}\| \, \|\boldsymbol{x}\|}\right)$$

where $\boldsymbol{a} \cdot \boldsymbol{x}$ is the dot product of $\boldsymbol{a}$ and $\boldsymbol{x}$. If $\theta \leq \frac{\pi}{2}$, then the satellite is above plane L, and is in view of the user. See the paper by Washburn [37] for a more detailed explanation of the geometry behind satellite coverage and more complex cases in which the field of view of the satellite is limited or the user is at an elevated point on the surface of the Earth.

5.1.4 Coordinate Transformations

In order for the above calculation of the satellites in view to work correctly, both satellite positions and user positions must be in the same coordinate frame. Typically, user positions are generated in an Earth-centered, Earth-fixed frame, also known as an Earth-centered relative (ECR or ECEF) frame. In Cartesian coordinates, the origin is located at the center of the earth, the X-axis runs through the intersection of the Prime Meridian and the Equator, the Z-axis runs through the North Pole, and the Y-axis is orthogonal to both

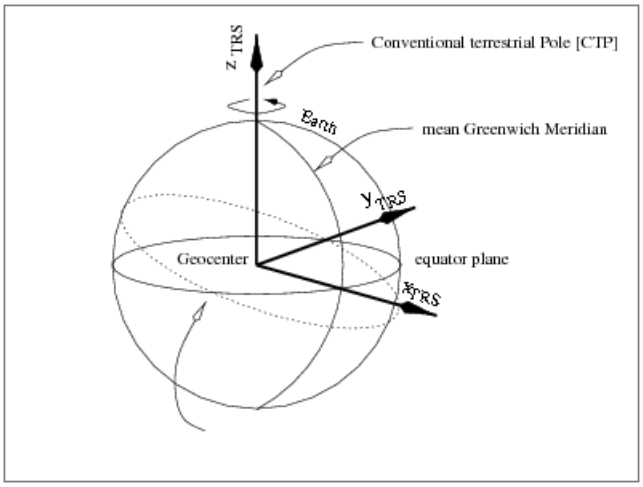


Figure 5.2: Earth-centered relative coordinate frame, also called a terrestrial reference system. Here, the axes rotate with the Earth's movement.

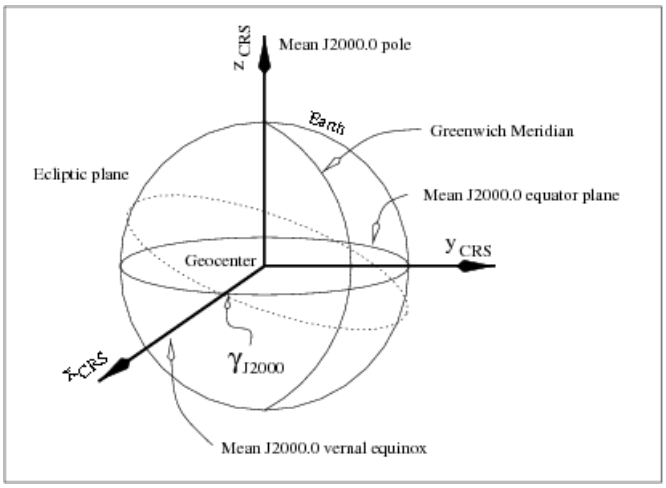


Figure 5.3: Earth-centered inertial coordinate frame, also called a celestial reference system. Here, the coordinate axes do not rotate with the Earth.

other axes (Figure 5.2). ECEF coordinates rotate with the rotation of the earth, so the location of a point on the earth's surface is fixed.

Satellite orbits, however, are propagated in an Earth-centered inertial frame (ECI), shown in Figure 5.3. The axes of this coordinate frame are fixed to be equivalent to the ECR axis at January 1, 2000, 12:00:00 (J2000). This frame does not shift with the rotation of the Earth, and fixes the positions of stars in the sky.

5.1.5 Orbits

Two-body orbits are defined by six classical orbital elements:

- a is the semi-major axis. For circular orbits, this is the distance between the centers of the two bodies
- e is the eccentricity: the amount of deviation in the shape of the orbit from a circle.
- i is the inclination: the angle between the reference plane (for geocentric orbits, the equatorial plane) and the plane of the orbit at the orbit's ascending node.

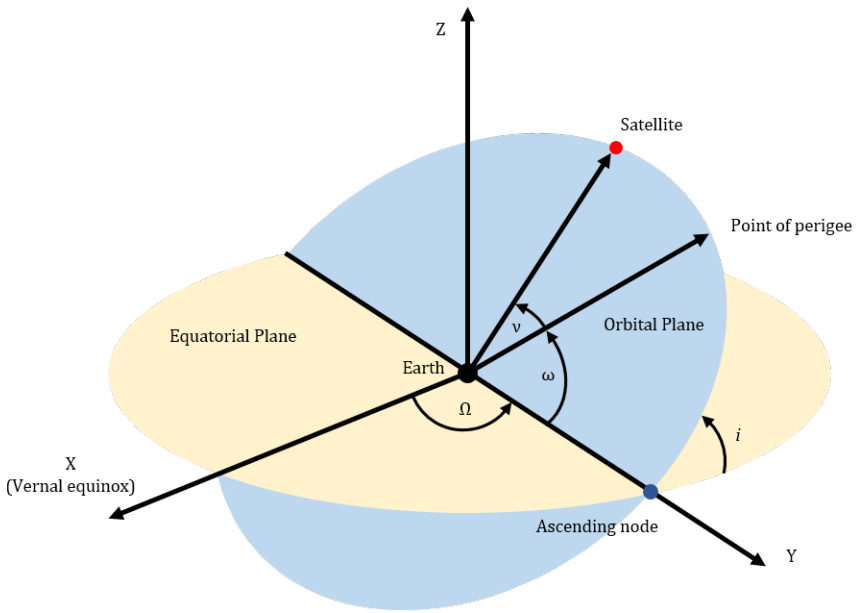


Figure 5.4: Classical orbital elements

- Ω is the right ascension of the ascending node: the angle between the vernal equinox and the ascending node of the orbit.
- ω is the argument of perigee: the angle between the ascending node and the point of periapsis of the orbit, which is the point at which the orbit is the closest to the Earth. This is undefined for circular orbits, but often set to 0, placing the point of periapsis for circular orbits at the same place as the ascending node of the orbit.
- ν is the true anomaly: the angle between the point of perigee and the current position of the orbiting body.

For an illustration of these parameters, see Figure 5.4.

5.1.6 Fibonacci Lattice

In order to evaluate the quality of global coverage provided by a certain Walker constellation, we must compute DOP values at evenly spaced positions representing around the globe. One method for placing points evenly on the sphere is the Fibonacci lattice [12]. To construct a Fibonacci lattice, choose an $N \in \mathbb{N}$ such that the desired number of points is $P = 2N + 1$. Let $i \in [0, 2N]$. Then the spherical coordinates, in radians, of the ith point are given by

$$\text{lat}_i = \arccos\left(1 - \frac{2i}{2N+1}\right) \tag{5.2}$$

$$\text{lon}_i = \pi i(1 + \sqrt{5}) \tag{5.3}$$

5.2 Experiments & Methodology

5.2.1 The Objective Function

The goal of the application problem is to minimize 98% global PDOP. If a user has a 98% PDOP of p, then 98% of the time, PDOP is less than or equal to p. A potential solution to

the optimization problem is a Walker constellation with parameters $T/P/F$ and inclination i. The objective function we wish to minimize is

$$f(T, P, F, i) = \sum_{i=1}^{M} \text{PDOP}_i^2 \tag{5.4}$$

where M is the number of evenly spaced users on the Earth's surface, and PDOP_i is the all-in-view, 98% average PDOP calculated at the i-th user. This value was found by averaging the PDOP calculated at the user location every 5 minutes over the course of 24 hours, discarding the worst 2% of values, and averaging the remaining values. If at any time point, fewer than 4 satellites (the minimum required to calculate PDOP) were in view, PDOP at that time was set to 50.

We used poliastro [32], a Python astrodynamics package, to set up our satellites as orbital objects after computing their initial positions ourselves. We then used pykep [15], another Python astrodynamics package, to propagate the satellite orbits because of the simplicity of its method for propagating Keplerian motion for circular orbits.

Satellite positions were transformed to ECR coordinates using the astropy [9] module to enable direct comparison with user positions on the Earth.

For our experiments, we set $M = 501$, but M can be increased for greater accuracy in computing global PDOP. Squaring the PDOP at each user ensures that small PDOP quantities will not contribute heavily to the objective function, while large values of PDOP will be severely penalized.

5.2.2 Constraints

We limited potential solutions to Walker Constellations of Medium Earth orbit (MEO) satellites in order to reflect the placement of actual GPS satellites. The altitude of all satellites was set to 20,200 kilometers. Orbital planes are evenly spaced around the reference plane (equatorial plane). Table 5.3 shows the bounds for each variable of a potential solution.

T	8-32
P	3-9
F	$1 \leq F \leq P - 1$
i	30-120°

Table 5.3: Range of Walker constellation parameters

5.2.3 Model Assumptions

We made the following simplifying assumptions when modeling the Earth and its satellite orbits:

- A spherical Earth with a radius of 6,378 kilometers
- A smooth surface of the Earth, meaning there are no changes in user elevation
- Circular, Keplerian orbits for all satellites with no orbital perturbations

5.3 Results

We ran a combination of hybrid, MLP, and RBF ensemble models, both weighted and unweighted, in place of the real PDOP fitness evaluation step after 3 generations: 1 generation was skipped and 2 generations were used as the ML training set. The genetic parameters were set to the ideal ones identified from our Taguchi results. After the genetic algorithm had converged, we then returned to the true objective function to evaluate that last population. The best fitnesses found in each model can be observed in Table 5.4.

	Weighted	**Unweighted**
Hybrid	895.01	911.97
MLP	938.80	9346.66
RBF	11297.14	898.4

Table 5.4: PDOP: best final fitness values across ensemble models

Here, we can see that the best fitness value, 895.01, was obtained from the weighted hybrid ensemble. As we were summing PDOP values from a total of approximately 500 users, we note that a PDOP of $895.01/500 = 1.79$ falls into the "excellent" category of overall PDOP ratings, resulting in good global coverage (see Figure 5.5). The unweighted model is only marginally worse, at a PDOP value of $911.97/500 = 1.82$, indicating that the hybrid model is more generalizable, allowing the genetic algorithm to accurately converge. In contrast, we note that while the weighted MLP ensemble and the unweighted RBF

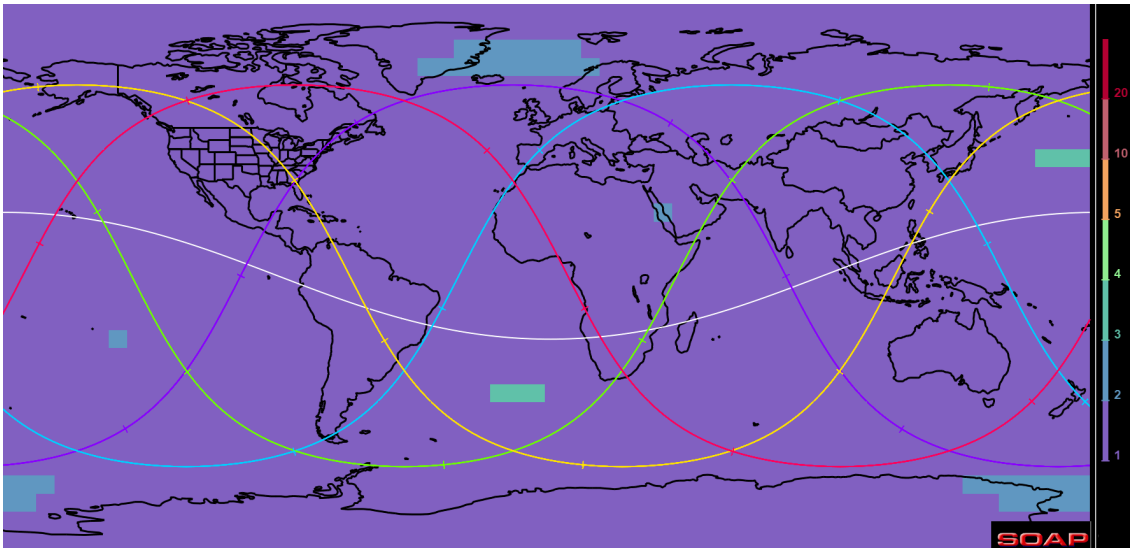


Figure 5.5: With good global PDOP, all areas of Earth have PDOP less than 5, indicated by the purple, blue, and blue-green areas. Obtained by the weighted hybrid ensemble GA, this constellation has excellent PDOP, i.e. values between 1 and 2, over most of the Earth.

ensemble also converged to a good fitness value, they were more susceptible to getting stuck in local optima when we toggled the weighted/unweighted parameters. While the unweighted MLP and weighted RBF occaisionally converged to a good constellation, in general they resulted in final solutions with poor global PDOP (see Figure 5.6).

Table 5.5 shows the total time in seconds necessary to run the entire MLGA algorithm on each model. The average across all models, 5431.62 seconds, demonstrates that using any version of the MLGA model still presents a significant time speedup over the base

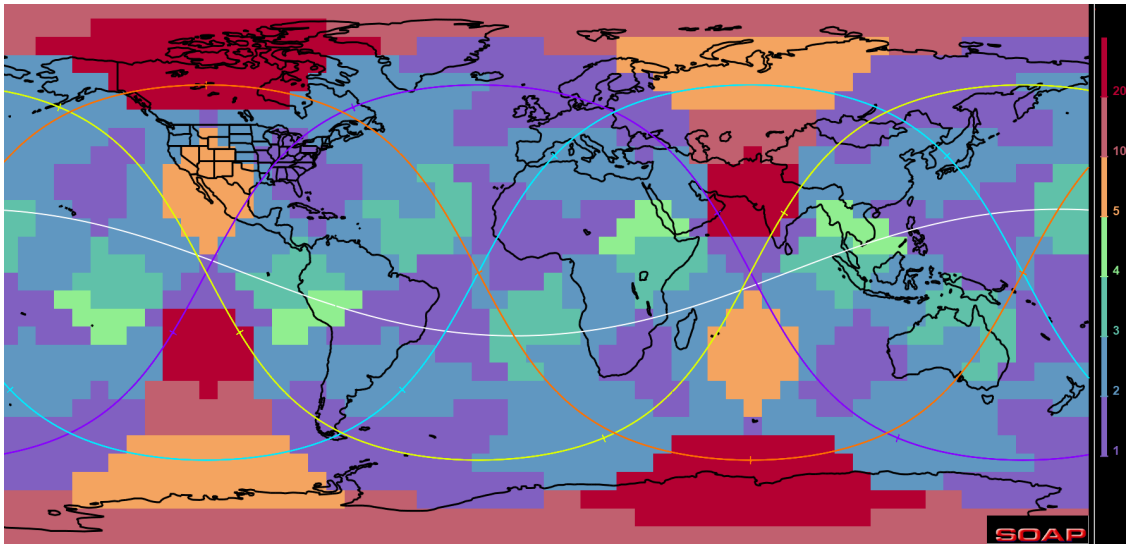


Figure 5.6: With poor global PDOP, many areas have PDOP values above 5, 10, or 20, indicated by red, pink, and orange areas.

genetic algorithm, which takes approximately 7 hours to converge. The five-fold difference in speedup will only increase with a greater number of users, greater range of allowed satellites, and overall more realistic constellation assumptions, indicating that the MLGA approach becomes increasingly vital in reducing computational time.

	Weighted	**Unweighted**
Hybrid	5379.13	5534.83
MLP	5392.97	5461.64
RBF	5454.36	5366.80

Table 5.5: PDOP: Runtime in seconds across ensemble models

Chapter 6

Future Work

6.1 Adaptive GA Parameters

Our current set of ideal GA parameters, identified by the Taguchi method, is maintained constant throughout the entire run of the GA. Although our algorithm converged rather quickly for the majority of the benchmark functions using these parameters, the same may not necessarily occur in a real-world problem with noisy data. There thus arises a need to adapt the GA parameters to the particular distribution the data set morphs into after several generational runs of the GA. One way to achieve this, as suggested by Mahmoodabadi and Nemati [23], is to introduce new adaptive crossover and mutation operators. The proposed crossover scheme combines the traditional method with the particle swarm optimization (PSO) operator, while the proposed mutation scheme uses sliding mode control to escape from local minima.

6.2 Deep Learning

Another way to further improve our neural network models is to incorporate new hidden layers. While this is impractical for the radial basis function network, which only operates on a single hidden layer, we can expand on the multilayer perceptron network instead, which has no limit on the number of hidden layers. Care needs to be taken, however, to ensure that the additional layers are not haphazardly added, as this would inevitably lead to overfitting and conversely, an undesirable decrease in accuracy.

6.3 Machine Learning Genetic Algorithm (MLGA) Model Improvements

6.3.1 Retraining of Neural Networks

In the initial stages of our MLGA, we estimated the confidence of each of our surrogate models' predictions and maintained a constant, user-defined margin of error. If the confidence level of the surrogate prediction dropped below a certain threshold, we initiated a counter. If enough instances of low confidence occurred, the algorithm triggered a retraining of the neural networks. With the additional hyperparameters needed to successfully implement this strategy (such as a user-defined margin of error, lower confidence bound, and number of

uncertain predictions allowed before retraining), we sought out other solutions. However, in the future, with more time allotted to tune such hyperparameters, this solution could guide our surrogate models to better predict the objective function, especially in later generations as the GA starts to hone in on smaller and more specific search regions.

6.4 Additional Parallelization

We have already parallelized the hyperparameter testing of the genetic algorithms and machine learning models. To obtain a sufficiently large dataset of potential data points and their corresponding PDOP values, we have also parallelized the objective function evaluations across each generation of the GA. One possible way to further speed up the overall computational time of our combined MLGA is to also parallelize the training of the neural networks across our ensemble. While our current setup of $N_{NETS} = 10$, trained sequentially, may not consume an enormous amount of time, this time elapsed will only increase linearly with the incorporation of additional ML models into the ensemble. Training in parallel will also be compatible with the retraining framework, overall producing a more accurate model without significantly sacrificing time.

6.5 Multi-Objective Optimization

For the purposes of this project, we treated satellite constellation design as a single-objective function, only minimizing PDOP. Actually, constellation design is inherently a multi-objective problem, since constellations with fewer satellites are preferable. Since our MLGA is designed for single-objective problems, this could be adjusted for by introducing a penalty for the number of satellites in the objective function and varying its magnitude to approximate the Pareto front.

6.6 More Complex Satellite Constellation Models

We made several simplifying assumptions about the Earth and the satellites orbiting it in order to model Walker constellations. Some of the most major assumptions were

- a spherical, smooth Earth
- no satellite rotation while in orbit
- circular orbits
- no orbital perturbations.

All of these assumptions do not affect the reality of the Earth and satellite orbits, and future work should focus on integrating more accurate information into the objective function.

Additionally, there are other satellite configurations to explore besides the popular Walker constellation, which requires the inclination and altitude for all orbits to be the same.

Appendix A

Genetic Algorithms

A.1 Alternative Selection and Crossover Schemes

A.1.1 Selection

- **Roulette-Style**: This strategy is modeled after a roulette wheel, where individuals are selected with a random "spin." Probabilities of selection are directly related to individual fitness values, with fitter members occupying a larger region of the roulette wheel. In the case of minimization problems, individuals with the lowest fitness (closer to the global minimum), possess a greater probability of selection. One of the advantages of this technique is that even weak population members are given a chance to be selected at each stage. One disadvantage of this technique is that selection is strongly biased in the beginning, which can cause early convergence and the rapid loss of diversity. Another disadvantage is that the entire populations fitness must be evaluated via the objective function in order to determine the proportional probability of each individual being chosen.

  While roulette-style selection is a widely used selection scheme, we decided to veto this technique since our goal is to reduce the number of times that our objective function must be evaluated. With rapid initial convergence, it is highly possible that our GAs will get trapped in local minima, leading us to prefer tournament-style selection.

A.1.2 Crossover

- **Two-Point Crossover**: This crossover scheme operates similarly to one-point crossover, except two common intersection points are now randomly selected. The values in between these intersection points are swapped to form two offspring.

- **Rank Proximity Based Crossover**: A specialized crossover scheme devised by Chakraborty [3], this technique aims to improve the speed and quality of the new offspring by increasing the probability of crossover when the rank in fitness of two individuals are high and they are closely related in the search space. The cutoff of what "high" is will be determined by the user, depending on whether they want more exploration of the gene pool or more selective pressure, and "closeness" will be determined by the Manhattan distance between the two individuals. This relationship between exploration and selective pressure can be further tuned by adjusting α, a

parameter that controls variance in the probability of crossovers. While $\alpha > 1$, we force extensive exploration in the beginning then rapid convergence towards the end of the iterations. While $\alpha < 1$, selective pressure is high from the start, almost forgoing exploration entirely. This algorithm has been already demonstrated to be effective in speeding up the computation and rate of convergence for the generalized Rosenbrock function, Schwefel's function, and the generalized Ackley's function. One caveat of using rank-proximity based crossover, however, is that the algorithm requires knowledge of the global minimum and maximum of the main objective fitness function during its computation. These values are not always feasible to obtain; while they can be derived from the benchmark functions we tested our genetic algorithms on, we will not always have a closed-form function when optimizing for fitness.

A.2 Taguchi Results

A.2.1 Experimental Orthogonal Array

Parameters	**Level 0**	**Level 1**	**Level 2**
Mut_Op	Gaussian	Cauchy	NA
Recomb	No	Yes	NA
Arch	No	Yes	NA
Cx_Op	One-Point	Uniform	NA
Pop_Size	250	500	1000
Tourn_Size	3%	4%	5%
Cxpb	0.3	0.6	0.9
Mutpb	0.3	0.6	0.9
Sc_Gen	5	15	25
Mut_Sig	0.1	1	10
Prec	0.001	0.01	0.1

Table A.1: Displays the parameters being tested in the Taguchi experiments, as well as their "levels" or settings.

Experiment	Mut_Op	Recomb	Arch	Cx_Op	Pop_Size	Tourn_Size	Cxpb	Mutpb	Sc_Gen	Mut_Sig	Prec
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	2	2	2	2	2	2	2
4	0	0	0	0	0	0	0	1	1	1	1
5	0	0	0	0	1	1	1	1	2	2	2
6	0	0	0	0	2	2	2	2	0	0	0
7	0	0	1	1	0	0	1	2	0	1	2
8	0	0	1	1	1	1	2	0	1	2	0
9	0	0	1	1	2	2	0	1	2	0	1
10	0	1	0	1	0	0	2	1	0	2	1
11	0	1	0	1	1	1	0	2	1	0	2
12	0	1	0	1	2	2	1	0	2	1	0
13	0	1	1	0	0	1	2	0	2	1	0
14	0	1	1	0	1	2	0	1	0	2	1
15	0	1	1	0	2	0	1	2	1	0	2
16	0	1	1	1	0	1	2	1	0	0	2
17	0	1	1	1	1	2	0	2	1	1	0
18	0	1	1	1	2	0	1	0	2	2	1
19	1	0	1	1	0	1	0	2	2	2	0
20	1	0	1	1	1	2	1	0	0	0	1
21	1	0	1	1	2	0	2	1	1	1	2
22	1	0	1	0	0	1	1	2	2	0	1
23	1	0	1	0	1	2	2	0	0	1	2
24	1	0	1	0	2	0	0	1	1	2	0
25	1	0	0	1	0	2	1	0	1	2	2
26	1	0	0	1	1	0	2	1	2	0	0
27	1	0	0	1	2	1	0	2	0	1	1
28	1	1	1	0	0	2	1	1	1	0	0
29	1	1	1	0	1	0	2	2	2	1	1
30	1	1	1	0	2	1	0	0	0	2	2
31	1	1	0	1	0	2	2	2	1	2	1
32	1	1	0	1	1	0	0	0	2	0	2
33	1	1	0	1	2	1	1	0	1	0	0
34	1	1	0	0	0	2	0	1	2	1	2
35	1	1	0	0	1	0	1	2	0	2	0
36	1	1	0	0	2	1	2	0	1	0	1

Table A.3

A.2.2 Parameter Tables

Fitness-Geared GA (SNR)			
Sphere Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	-14.046	-30.229	-10.036
TOURN_SIZE	-18.363	-25.970	-18.068
CXPB***	-6.251	-25.251	-22.809
MUTPB***	-21.173	-21.893	-11.305
MUT_SIG***	16.396	-23.006	-50.984
CX_OP	-18.254	-17.953	NA
MUT_OP***	6.641	-42.848	NA
PREC***	-13.376	-30.239	-10.696
SC_GEN***	-7.953	-15.567	-30.157
RECOMB	-18.282	-17.926	NA
ARCH	-17.838	-18.370	NA
Run Time-Geared GA			
Sphere Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	0.168	0.358	1.048
TOURN_SIZE**	0.503	0.556	0.627
CXPB***	0.444	0.591	0.540
MUTPB***	0.588	0.489	0.489
MUT_SIG	0.566	0.488	0.514
CX_OP***	0.478	0.572	NA
MUT_OP***	0.603	0.447	NA
PREC**	0.535	0.549	0.490
SC_GEN***	0.170	0.524	0.851
RECOMB***	0.473	0.576	NA
ARCH***	0.553	0.496	NA

Table A.4: Displays the results of Taguchi analysis. In the parameters column, * represents a parameter found to be statistically significant ($* : \alpha = 0.05$, $** : \alpha = 0.01$, $*** : \alpha = 0.001$). For the fitness-geared genetic algorithms, the parameter level with the greatest SNR value represents the ideal parameter setting. For the speed-geared genetic algorithms, the parameter level with the lowest average run time is found to be the ideal setting.

Fitness-Geared GA (SNR)			
Rosenbrock Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	-110.103	-119.407	-102.968
TOURN_SIZE	-110.984	-114.520	-110.338
CXPB***	-101.683	-116.844	-113.951
MUTPB***	-112.434	-113.455	-106.674
MUT_SIG***	-83.280	-119.487	-132.728
CX_OP	-110.843	-110.809	NA
MUT_OP***	-94.386	-127.266	NA
PREC***	-108.493	-119.422	-104.562
SC_GEN***	-103.320	-109.221	-119.445
RECOMB	-110.829	-110.823	NA
ARCH	-110.321	-111.330	NA
Run Time-Geared GA			
Rosenbrock Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	0.189	0.380	1.125
TOURN_SIZE***	0.532	0.589	0.680
CXPB***	0.471	0.647	0.576
MUTPB***	0.643	0.522	0.518
MUT_SIG*	0.607	0.541	0.518
CX_OP***	0.504	0.624	NA
MUT_OP***	0.648	0.480	NA
PREC***	0.654	0.558	0.481
SC_GEN***	0.178	0.567	0.915
RECOMB***	0.508	0.621	NA
ARCH***	0.606	0.523	NA

Table A.5: Displays the results of Taguchi analysis. In the parameters column, * represents a parameter found to be statistically significant ($* : \alpha = 0.05$, $** : \alpha = 0.01$, $*** : \alpha = 0.001$). For the fitness-geared genetic algorithms, the parameter level with the greatest SNR value represents the ideal parameter setting. For the speed-geared genetic algorithms, the parameter level with the lowest average run time is found to be the ideal setting.

Fitness-Geared GA (SNR)			
Step Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	-66.811	-66.752	-66.948
TOURN_SIZE	-66.793	-66.953	-66.768
CXPB***	-66.974	-66.968	-66.569
MUTPB***	-67.364	-66.563	-66.517
MUT_SIG***	-67.323	-67.149	-66.025
CX_OP***	-66.706	-66.968	NA
MUT_OP***	-66.736	-66.938	NA
PREC	-67.049	-66.640	-66.822
SC_GEN***	-66.707	-67.219	-66.542
RECOMB***	-66.749	-66.925	NA
ARCH***	-66.715	-66.959	NA
Run Time-Geared GA			
Step Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	0.224	0.465	1.347
TOURN_SIZE***	0.633	0.716	0.831
CXPB*	0.616	0.747	0.673
MUTPB**	0.728	0.652	0.649
MUT_SIG***	0.784	0.621	0.617
CX_OP***	0.611	0.746	NA
MUT_OP***	0.817	0.540	NA
PREC**	0.675	0.728	0.632
SC_GEN***	0.263	0.670	1.069
RECOMB***	0.627	0.730	NA
ARCH*	0.703	0.654	NA

Table A.6: Displays the results of Taguchi analysis. In the parameters column, * represents a parameter found to be statistically significant ($* : \alpha = 0.05$, $** : \alpha = 0.01$, $*** : \alpha = 0.001$). For the fitness-geared genetic algorithms, the parameter level with the greatest SNR value represents the ideal parameter setting. For the speed-geared genetic algorithms, the parameter level with the lowest average run time is found to be the ideal setting.

Fitness-Geared GA (SNR)			
Quartic Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	-102.922	-108.769	-95.875
TOURN_SIZE	-104.098	-104.440	-102.216
CXPB***	-97.150	-107.288	-103.129
MUTPB***	-103.649	-104.296	-99.675
MUT_SIG***	-80.913	-109.846	-119.219
CX_OP	-102.739	-102.305	NA
MUT_OP***	-88.811	-116.233	NA
PREC***	-100.887	-109.854	-96.825
SC_GEN***	-96.714	-100.881	-109.624
RECOMB*	-103.632	-101.412	NA
ARCH	-102.299	-102.745	NA
Run Time-Geared GA			
Quartic Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	18.811	39.491	80.025
TOURN_SIZE***	41.890	52.405	53.509
CXPB*	40.669.	51.670	45.988
MUTPB	50.524	38.155	48.617
MUT_SIG***	54.336	42.330	40.661
CX_OP**	42.899	49.319	NA
MUT_OP	45.956	46.262	NA
PREC***	100.612	31.837	5.879
SC_GEN	41.649	52.938	42.799
RECOMB	45.048	47.171	NA
ARCH***	54.328	37.891	NA

Table A.7: Displays the results of Taguchi analysis. In the parameters column, * represents a parameter found to be statistically significant ($* : \alpha = 0.05$, $** : \alpha = 0.01$, $*** : \alpha = 0.001$). For the fitness-geared genetic algorithms, the parameter level with the greatest SNR value represents the ideal parameter setting. For the speed-geared genetic algorithms, the parameter level with the lowest average run time is found to be the ideal setting.

Fitness-Geared GA (SNR)			
Foxholes Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	-77.168	-88.812	-72.752
TOURN_SIZE	-80.326	-84.560	-79.195
CXPB***	-71.344	-84.002	-83.384
MUTPB***	-80.808	-83.716	-74.450
MUT_SIG***	-54.586	-80.291	-105.996
CX_OP	-79.438	-79.717	NA
MUT_OP***	-59.659	-99.496	NA
PREC***	-75.565	-88.650	-74.517
SC_GEN***	-73.697	-76.226	-88.598
RECOMB	-80.133	-79.021	NA
ARCH	-79.504	-79.650	NA
Run Time-Geared GA			
Foxholes Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	0.629	1.138	2.522
TOURN_SIZE	1.502	1.518	1.504
CXPB***	1.253	1.524	1.512
MUTPB***	1.506	1.415	1.361
MUT_SIG	1.540	1.332	1.400
CX_OP***	1.314	1.545	NA
MUT_OP*	1.480	1.383	NA
PREC**	1.464	1.476	1.350
SC_GEN***	0.526	1.483	2.201
RECOMB***	1.357	1.502	NA
ARCH**	1.481	1.379	NA

Table A.8: Displays the results of Taguchi analysis. In the parameters column, * represents a parameter found to be statistically significant ($* : \alpha = 0.05$, $** : \alpha = 0.01$, $*** : \alpha = 0.001$). For the fitness-geared genetic algorithms, the parameter level with the greatest SNR value represents the ideal parameter setting. For the speed-geared genetic algorithms, the parameter level with the lowest average run time is found to be the ideal setting.

Fitness-Geared GA (SNR)			
Schwefel Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	-88.234	-85.483	-51.540
TOURN_SIZE***	-80.205	-79.604	-64.850
CXPB***	-82.158	-69.552	-73.547
MUTPB***	-82.158	-69.552	-73.547
MUT_SIG***	-55.155	-77.401	-94.555
CX_OP***	-81.457	-68.714	NA
MUT_OP***	-50.821	-99.351	NA
PREC*	-71.117	-82.868	-71.271
SC_GEN*	-71.875	-76.113	-76.916
RECOMB	-75.539	-74.632	NA
ARCH	-74.934	-75.237	NA
Run Time-Geared GA			
Schwefel Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	1.737	2.218	6.673
TOURN_SIZE***	3.541	3.258	4.247
CXPB***	3.965	3.300	3.364
MUTPB***	2.843	3.201	4.614
MUT_SIG***	6.039	2.125	2.138
CX_OP*	3.698	3.387	NA
MUT_OP***	5.082	2.003	NA
PREC	3.830	3.217	3.581
SC_GEN**	3.285	3.550	3.772
RECOMB	3.514	3.571	NA
ARCH	3.561	3.524	NA

Table A.9: Displays the results of Taguchi analysis. In the parameters column, * represents a parameter found to be statistically significant ($* : \alpha = 0.05$, $** : \alpha = 0.01$, $*** : \alpha = 0.001$). For the fitness-geared genetic algorithms, the parameter level with the greatest SNR value represents the ideal parameter setting. For the speed-geared genetic algorithms, the parameter level with the lowest average run time is found to be the ideal setting.

Fitness-Geared GA (SNR)			
Rastrigin Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	-98.151	-99.767	-93.221
TOURN_SIZE**	-98.313	-97.165	-96.670
CXPB***	-95.428	-98.419	-97.292
MUTPB***	-94.833	-98.275	-98.318
MUT_SIG***	-87.483	-99.333	-105.311
CX_OP***	-98.479	-95.614	NA
MUT_OP***	-91.270	-102.822	NA
PREC***	-95.948	-100.042	-95.149
SC_GEN***	-95.546	-95.611	-99.976
RECOMB***	-97.784	-96.309	NA
ARCH	-97.015	-97.078	NA
Run Time-Geared GA			
Rastrigin Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	0.875	1.645	3.865
TOURN_SIZE***	2.043	2.112	2.636
CXPB**	2.010	2.128	2.246
MUTPB***	1.754	1.844	2.794
MUT_SIG***	2.536	1.913	1.883
CX_OP*	2.063	2.194	NA
MUT_OP***	2.801	1.455	NA
PREC*	2.153	1.992	1.240
SC_GEN***	1.382	2.203	2.731
RECOMB*	2.067	2.189	NA
ARCH***	2.244	2.012	NA

Table A.10: Displays the results of Taguchi analysis. In the parameters column, * represents a parameter found to be statistically significant ($* : \alpha = 0.05$, $** : \alpha = 0.01$, $*** : \alpha = 0.001$). For the fitness-geared genetic algorithms, the parameter level with the greatest SNR value represents the ideal parameter setting. For the speed-geared genetic algorithms, the parameter level with the lowest average run time is found to be the ideal setting.

Fitness-Geared GA (SNR)			
Griewangk Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE*	-29.624	-22.736	-21.619
TOURN_SIZE***	-28.972	-21.777	-18.393
CXPB*	-29.345	-22.568	-22.066
MUTPB*	-23.292	-20.724	-29.748
MUT_SIG***	-28.404	-31.3103	-14.507
CX_OP	-26.484	-22.835	NA
MUT_OP***	-18.857	-30.463	NA
PREC	-27.503	-25.117	-21.359
SC_GEN	-25.129	-27.657	-20.983
RECOMB	-24.578	-24.741	NA
ARCH	-24.060	-25.259	NA
Run Time-Geared GA			
Griewangk Function			
Parameters	Level 0	Level 1	Level 2
POP_SIZE***	1.245	1.718	4.727
TOURN_SIZE	2.585	2.618	3.013
CXPB	2.611	2.503	2.576
MUTPB	2.739	2.158	2.745
MUT_SIG*	3.463	2.026	2.081
CX_OP	2.735	2.391	NA
MUT_OP*	-18.857	-30.463	NA
PREC*	-27.503	-25.117	-21.359
SC_GEN**	-25.129	-27.657	-20.983
RECOMB	-24.578	-24.741	NA
ARCH	-24.060	-25.259	NA

Table A.11: Displays the results of Taguchi analysis. In the parameters column, * represents a parameter found to be statistically significant ($* : \alpha = 0.05$, $** : \alpha = 0.01$, $*** : \alpha = 0.001$). For the fitness-geared genetic algorithms, the parameter level with the greatest SNR value represents the ideal parameter setting. For the speed-geared genetic algorithms, the parameter level with the lowest average run time is found to be the ideal setting.

Appendix B

Support Vector Regression

B.1 Theoretical Background

Support vector regression (SVR) is an application of support vector machines (SVMs), which are a type of supervised machine learning model.

B.1.1 SVR as an Optimization Problem

Suppose we have a set of training data $\{(x_1, y_1), .., (x_m, y_m)\} \subset \mathbb{X} \times \mathbb{R}$ where $\mathbb{X}$ is the space of the inputs of the function we wish to approximate. The goal of support vector regression is to find a function $f(x)$, as flat as possible, of the form

$$f(x) = w \cdot x$$

with $x \in \mathbb{X}$ where $w \cdot x_i$ is the dot product of w and x $\in \mathbb{X}$, satisfying the problem

$$\text{minimize } \frac{1}{2}\|x\|^2 \tag{B.1}$$

$$\text{s.t. } \begin{cases} y_i - w \cdot x_i - b \leq \varepsilon \\ w \cdot x_i + b - y_i \leq \varepsilon \end{cases} \tag{B.2}$$

Qualitatively, this means that all training points lie within a tube of width ε around $f(x)$. However this may be infeasible, slack variables are introduced ξ_i, ξ_i^* that allow for points outside of the ε-tube. The model is penalized for each point that is more than ε away from $f(x)$ according to C, a user-defined penalty parameter. Instead, the following optimization problem is solved

$$\text{minimize } \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{l}(\xi_i + \xi_i^*) \tag{B.3}$$

$$\text{s.t. } \begin{cases} y_i - w \cdot x_i - b \leq \varepsilon + \xi_i \\ w \cdot x_i + b - y_i \leq \varepsilon + \xi_i^* \\ \xi_i, \xi_i^* \geq 0 \end{cases} \tag{B.4}$$

A visualization of this convex optimization problem in two dimensions is shown in Figure B.1.

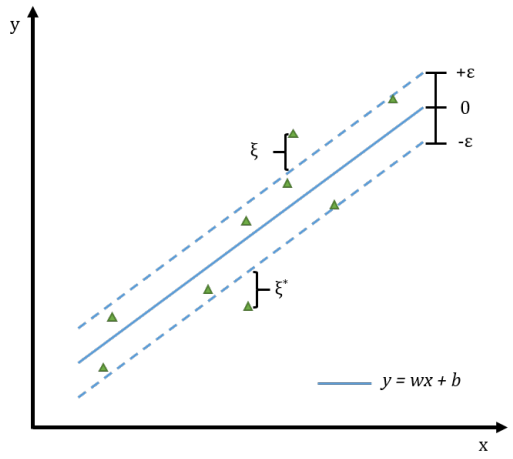


Figure B.1: Two dimensional SVR illustration: ϵ defines the ϵ-tube, while ξ and ξ^* are the slack variables.

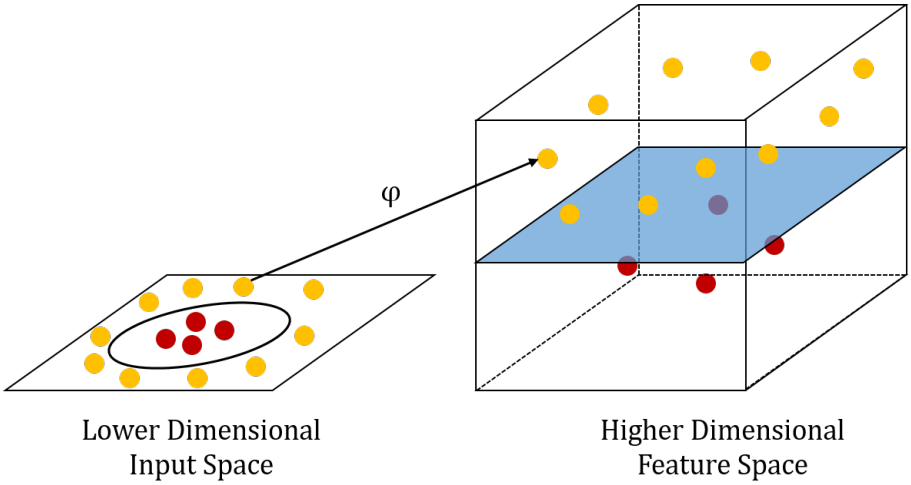


Figure B.2: Illustration of the concept of a kernel trick: The kernel is a function that that allows points in a space to be mapped to a higher dimensional, or feature space.

Instead of solving the optimization problem in this form, the primal, solving it in the dual form is preferred. The derivation of the dual can be found in the tutorial on SVR by Smola [34].

B.1.2 Kernels

Kernel is a function that allows points in a space to be mapped to a higher dimensional, or feature space. Figure B.2 demonstrates this concept. Explicitly computing the values of points in the higher dimensional space is computationally intensive, and kernel tricks allow the avoidance of these computations while still representing data in a space in which it may be linearly separable. For requirements of a kernel and how to derive one, see the Smola tutorial on SVR.

B.1.3 SVR Parameters

In their paper on selecting parameters for SVR, Cherkassy and Ma claim that "parameters C and ε are selected by users based on a priori knowledge and/or user expertise" [5]. SVR models are highly sensitive to parameter tuning, so hand-tuning is often preferred to a computationally intensive grid-search.

- **Size of training set**: The number of training samples heavily influences the ability of the SVR model to learn to approximiate a function. If there are not enough representative points from
- **Kernel**: The choice of kernel is essential for a good fit in an SVR model, as SVR is dependent on the data being generally linearly separable in some feature space. Some popular choices are radial basis functions, polynomial, and sigmoidal kernels. The linear kernel often only works for lower-dimensional problems with little complexity. Within the choice of kernel itself are more parameters; for instance, the degree and coefficient of the polynomial kernel must also be specified by the use.
- ε: The ε that determines the width of the tube around a potential solution to the optimization problem must be chosen carefully. An ε that is too small may make the optimization problem infeasible, but an ε that is too large allows for too much deviation from a potential solution $f(x)$ and will result in a poor fit.
- **C**: Similar to ε, the penalty parameter C must be selected carefully due to its direct influence on the final solution to the optimization problem. A large C can cause overfitting by forcing the function $f(x)$ to adhere too closely to the points in the training set, while a small C could result in a poor fit by allowing $f(x)$ to ignore training points with little penalty.

B.2 Experiments & Methodology

Support vector regression (SVR) was implemented in Python using the `sklearn.svm` module of scikit-learn [29]. It was tested on 12 of the benchmarking functions, F1 - F8 of the continuous functions and F15 - F18, which are all discrete. Data was randomly generated within the function bounds (except for the Foxholes model, which is described in the discussion of the results). The kernel coefficient, was set to $1/n$ for all trials, where $n =$ individual size, in order to reduce the number of parameters that required tuning.

To demonstrate the importance of parameter tuning and to give insight into our hand-tuning process, we present accuracy plots for SVR models trained on data from one of the benchmarking functions, the sphere function, in Figures B.3 and B.4.

B.3 Results

Unsurprisingly, SVR performs excellently on low-dimensional data from functions that are simple, such as the sphere and Rosenbrock functions. On the discretized functions, SVR performs well on Bukin No. 6, perhaps because it is convex. On highly multimodal functions such as the Ackley and Schwefel functions, SVR fails to learn the function at all.

For the Schwefel function, the training set size is small because a larger set took an unfeasible amount of time to train. We suspect that the SVR model for approximating the

Parameters for Functions by Best R^2 Value							
Function	**R^2**	**n_train**	**kernel**	**C**	**epsilon**	**degree**	**coeff**
Rosenbrock	1	700	poly	100	0.1	4	1
Bukin N. 6	0.999996	5000	rbf	50	0.1		
Sphere	0.999974	700	poly	100	0.1	3	1
Shekel's Foxholes	0.999621	15000	rbf	100	0.1		
Step	0.990703	700	linear	100			
Quartic	0.877507	7000	rbf	100	1		
Levy	0.413601	700	poly	1	10	4	1
Schaffer N. 2	0.228614	5000	rbf	1	0.1		
Rastrigin	0.164871	4000	poly	100	0.1	5	1
Schwefel	0.050135	700	linear	10			
Griewangk	-0.00012	7000	rbf	100	0.1		
Ackley	-0.00016	5000	rbf	1	0.1		

Table B.1: R^2 is the coefficient of determination for each regression model. N_train is the number of training data points, kernel is the type of kernel function used, C is the penalty parameter, and epsilon is the allowed error. Degree and coeff are the additional parameters for polynomial kernels. Each SVR model was tested on 300 randomly generated data points, except for the model for Shekel's Foxholes, which was tested on 5000 points.

Schwefel function requires a larger training set, but because we are trying to reduce the time required to computer or estimate the objective functions in our genetic algorithms, it is not sensible or valuable to run trials with a larger training set.

Unexpectedly, on Shekel's Foxholes function, SVR outperforms both neural network implementations, Multilayer Perception (MLP) and Radial Basis Function (RBF) Networks. However, the model required a training set of 15,000 points, over 20 times larger than our smallest training sets, and it required a reduction of bounds for the initial generation of points. We believe this suggests that the model requires many training points located in the Foxholes in order to learn the function. Collecting a training set of 15,000 or more points would require just as many objective function evaluations once we combine SVR with our genetic algorithms, so this also does not appear to be a practical method for approximating a complicated objective function, even if parameters do exist that allow the model to learn the function very well.

Table B.1 presents a chart of the parameter sets for each function that yielded the best R^2 values. Note that a poor R^2 value may indicate that we have simply not found the correct parameter set for that SVR model, but because we want to reduce computational time, searching for those parameters may be impractical. We have also included the difference and accuracy plots for the Foxholes function and the Schaffer No. 2 function, which has discrete inputs.

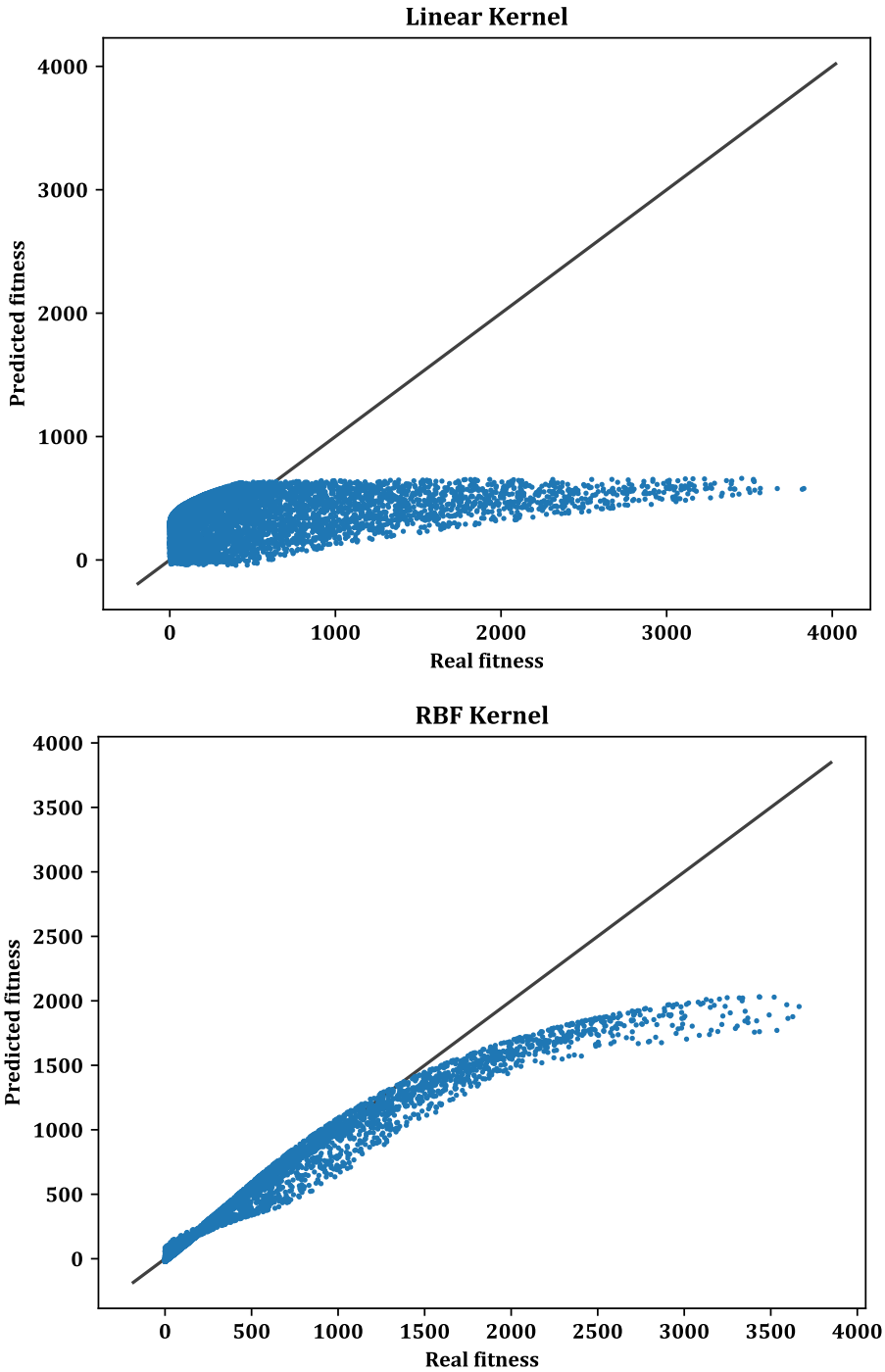


Figure B.3: Plots of the predicted vs. actual fitness values for the Sphere function when kernel type is varied.

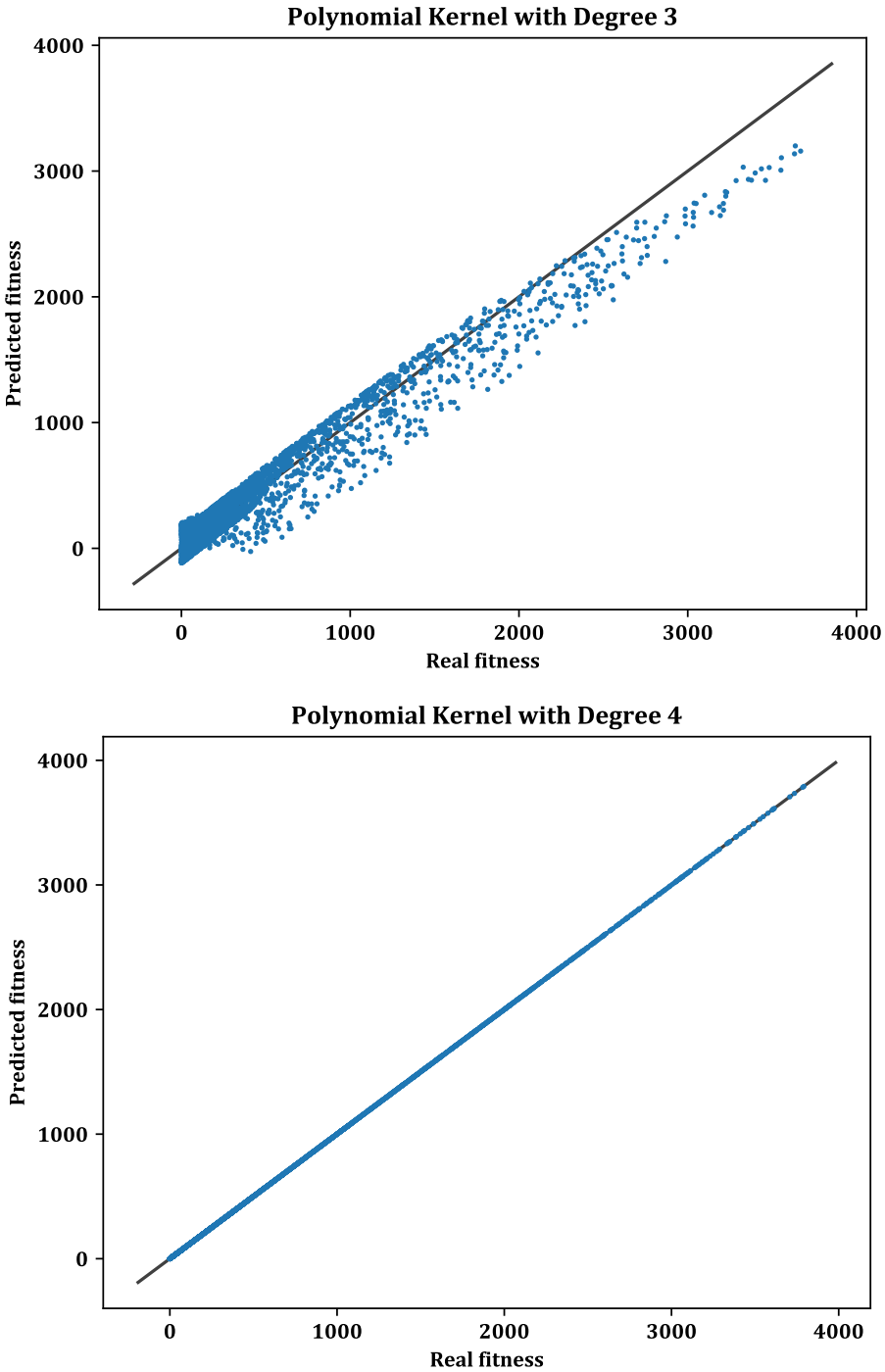


Figure B.4: Plots of the predicted vs. actual fitness values for the Sphere function when kernel type is varied.

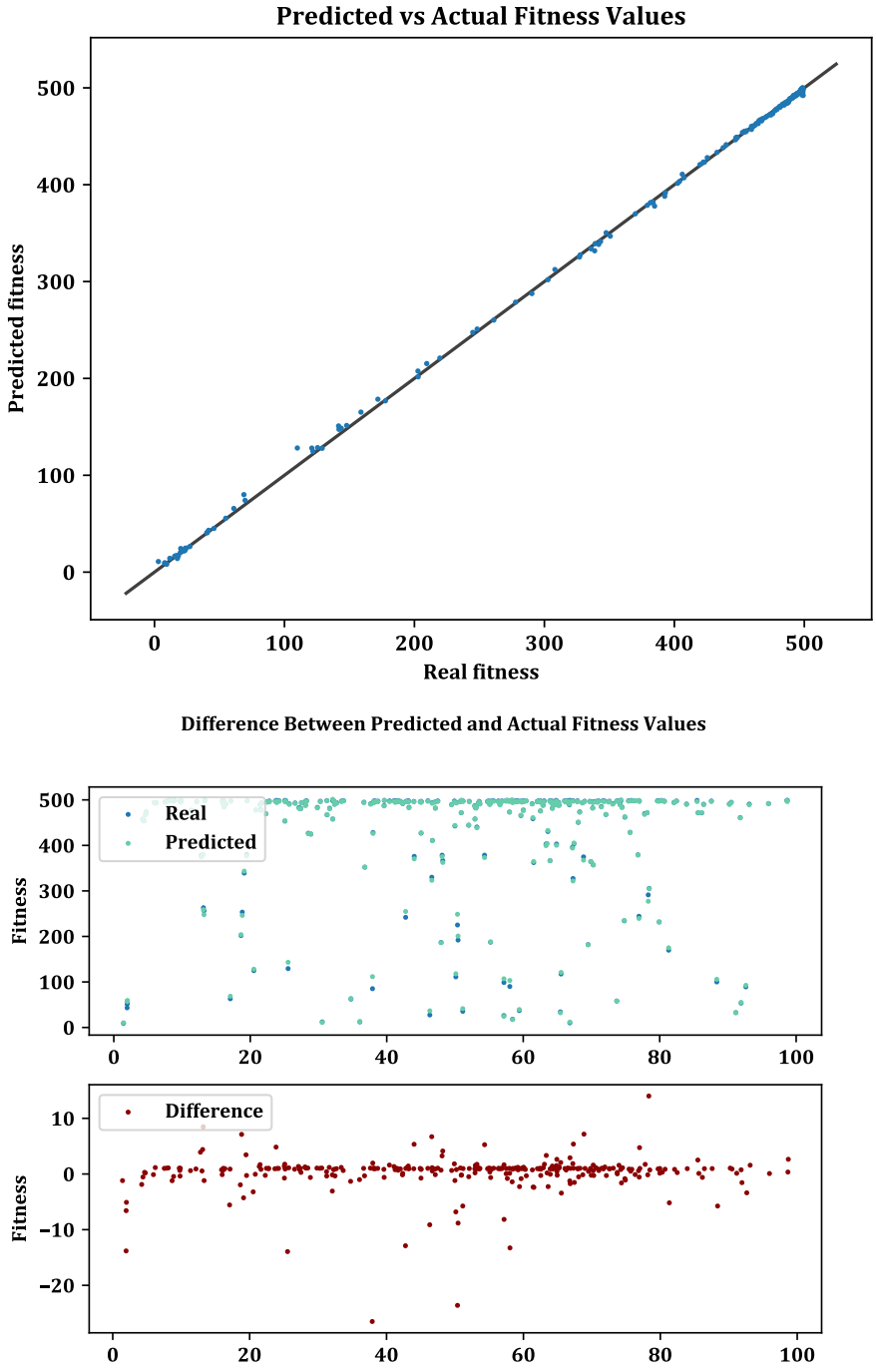


Figure B.5: Plots of the difference between the predicted and actual fitness values for Shekel's Foxholes function. Training set size was 300 points. Additional parameter values can be found in table B.1.

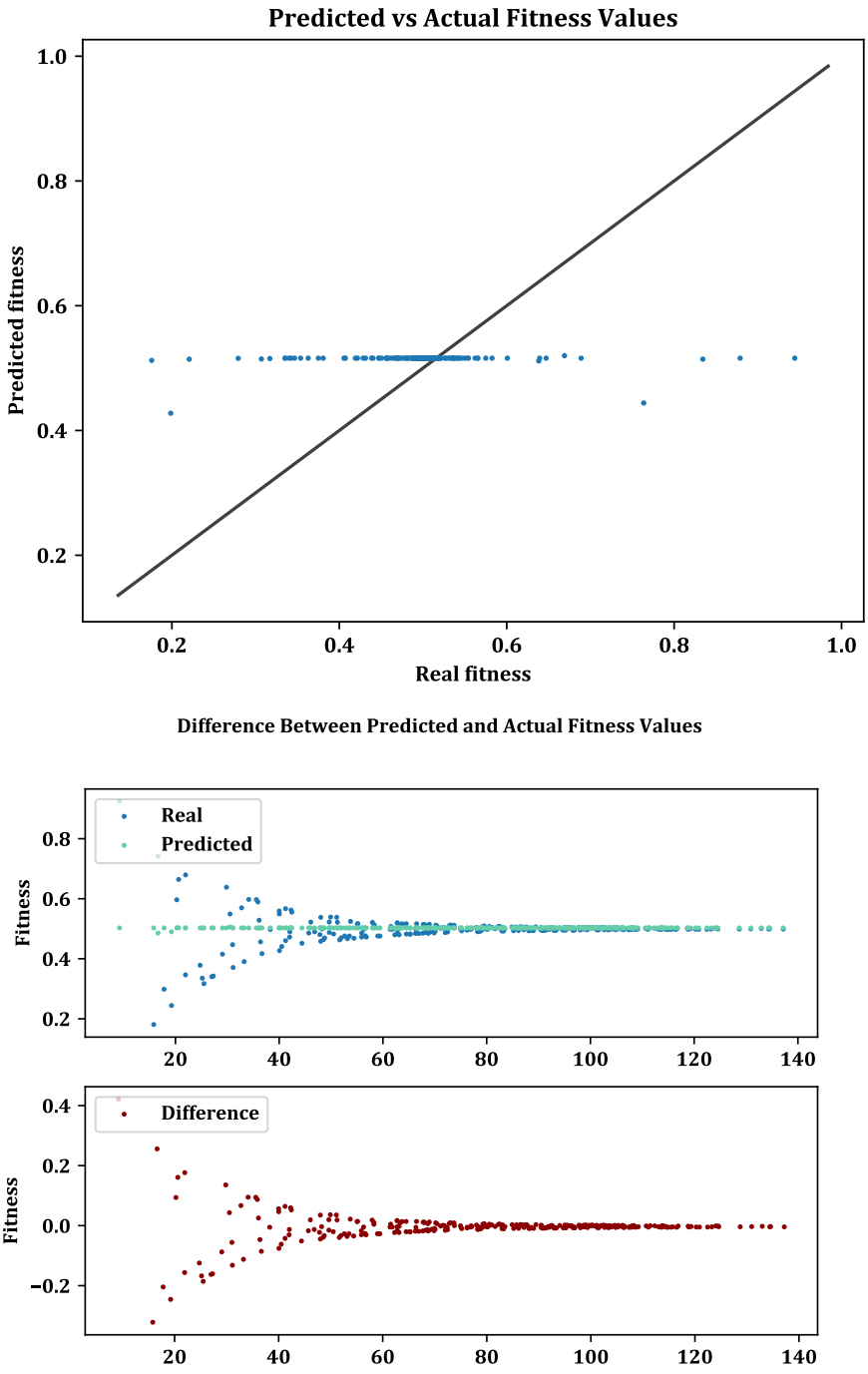


Figure B.6: Plots of the difference between the predicted and actual fitness values for the Schaffer No. 2 function. Training set size was 300 points. Additional parameter values can be found in table B.1.

Appendix C

Abbreviations

API. Application programming interface.

ANOVA. Analysis of variance.

DEAP. Distributed Evolutionary Algorithms in Python.

DOP. Dilution of precision.

ECEF. Earth-centered earth focus.

ECR. Earth-centered relative.

GA. Genetic algorithm.

GPS. Global positioning system.

IPAM. Institute for Pure and Applied Mathematics.

MEO. Medium Earth orbit.

MLGA. Machine learning genetic algorithm.

MLP. Multilayer perceptron.

MSE. Mean squared error.

PDOP. Positional dilutiono of precision.

PU. Pattern unit.

RBF. Radial basis function.

RIPS. Research in Industrial Projects for Students.

RMSE. Root mean squared error.

SGD. Stochastic gradient descent.

SNR. Signal to noise ratio.

SVR. Support vector regression.

Bibliography

[1] Y. Bengio, *Practical recommendations for gradient-based training of deep architectures*, CoRR, abs/1206.5533 (2012).

[2] L. Breiman, *Bagging predictors*, Machine Learning, 24 (1996), pp. 123–140.

[3] G. Chakraborty and B. Chakraborty, *Rank and proximity based crossover (RPC) to improve convergence in genetic search*, IEEE Congress on Evolutionary Computation, (2005), pp. 1311–1316.

[4] R. Cheng, C. He, Y. Jin, and X. Yao, *Model-based evolutionary algorithms: a short survey*, Complex & Intelligent Systems, 4 (2018), pp. 283–292.

[5] V. Cherkassy and Y. Ma, *Practical selection of SVM parameters and noise estimation for svm regression*, Neural Networks, 17 (2003), pp. 113–126.

[6] F. Chollet et al., *Keras*. https://keras.io, 2019.

[7] J. G. Digalakis and K. G. Margaritis, *On benchmarking functions for genetic algorithms*, International Journal of Computer Mathematics, 00 (2001), pp. 1–27.

[8] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*, no. 57 in Monographs on Statistics and Applied Probability, Chapman & Hall/CRC, Boca Raton, Florida, USA, 1993.

[9] P. et al., *The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package*, , 156 (2018), p. 123.

[10] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, *DEAP: Evolutionary algorithms made easy*, Journal of Machine Learning Research, 13 (2012), pp. 2171–2175.

[11] T. Goel, R. T. Haftka, W. Shyy, and N. V. Queipo, *Ensemble of surrogates*, Structural and Multidisciplinary Optimization, 33 (2007), pp. 199–216.

[12] Á. González, *Measurement of areas on a sphere using fibonacci and latitude–longitude lattices*, Mathematical Geosciences, 42 (2009), pp. 49–64.

[13] B. M. Gopalsamy, B. Mondal, and S. Ghosh, *Taguchi method and ANOVA: An approach for process parameters optimization of hard machining while machining hardened steel*, Journal of Scientific & Industrial Research, 68 (2009), pp. 686 – 695.

[14] A. S. Hedayat, N. J. A. Sloane, and J. Stufken, *Introduction*, Springer New York, New York, NY, 1999, pp. 1–9.

[15] D. Izzo and F. Biscani, *pykep*. https://zenodo.org/record/2575462.XVrhD-hKgjw, 2006–2019.

[16] B. Jain, H. Polheim, and J. Wegener, *On termination criteria of evolutionary algorithms*, in GECCO'2001 - Proceedings of the Genetic and Evolutionary Computation Conference, L. Spector, ed., San Francisco, CA, 2001, Morgan Kaufmann Publishers, p. 768.

[17] Y. Jin, *Surrogate-assisted evolutionary computation: Recent advances and future challenges*, Swarm and Evolutionary Computation, 1 (2011), pp. 61 – 70.

[18] Y. Jin and B. Sendhoff, *Fitness approximation in evolutionary computation - a survey*, in Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation, GECCO'02, San Francisco, CA, USA, 2002, Morgan Kaufmann Publishers Inc., pp. 1105–1112.

[19] R. B. Langley, *Dilution of precision*, GPS World, (1999), pp. 52–59.

[20] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin, *Exploring strategies for training deep neural networks*, J. Mach. Learn. Res., 10 (2009), pp. 1–40.

[21] M. Laumanns, E. Zitzler, and L. Thiele, *On the effects of archiving, elitism, and density based selection in evolutionary multi-objective optimization*, in Evolutionary Multi-Criterion Optimization, E. Zitzler, L. Thiele, K. Deb, C. A. Coello Coello, and D. Corne, eds., Berlin, Heidelberg, 2001, Springer Berlin Heidelberg, pp. 181–196.

[22] V. Lin, *An introduction to GPS and dilution of precision (DOP)*. Aerospace Corporation Material, July 2019.

[23] M. Mahmoodabadi and A. Nemati, *A novel adaptive genetic algorithm for global optimization of mathematical test functions and real-world problems*, Engineering Science and Technology, an International Journal, (2016), pp. 2001–2021.

[24] N. J. Majaj and D. G. Pelli, *Deep learning: Using machine learning to study biological vision*, bioRxiv, (2017).

[25] D. Masters and C. Luschi, *Revisiting small batch training for deep neural networks*, ArXiv, abs/1804.07612 (2018).

[26] J. J. Moré, B. S. Garbow, and K. E. Hillstrom, *Testing unconstrained optimization software*, ACM Trans. Math. Softw., 7 (1981), pp. 17–41.

[27] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*, MIT Press, Cambridge, Massachusetts, 2012.

[28] H. M. Pandey, *Performance evaluation of selection methods of genetic algorithm and network security concerns*, Procedia Computer Science, 78 (2016), pp. 13 – 18. 1st International Conference on Information Security Privacy 2015.

[29] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, *Scikit-learn: Machine learning in Python*, Journal of Machine Learning Research, 12 (2011), pp. 2825–2830.

[30] R. Poli and W. Langdon, *On the search properties of different crossover operators in genetic programming*, in University of Wisconsin, Morgan Kaufmann, 1998, pp. 293–301.

[31] R. Rao, R. Prakasham, K. Prasad, S. Rajesham, P. Sarma, and L. Rao, *Xylitol production by candida sp.: parameter optimization using taguchi approach*, Process Biochemistry, 39 (2004), pp. 951 – 956.

[32] J. L. C. RodrÃguez, *poliastro*. https://github.com/poliastro/poliastro, 2013-2019.

[33] S. Ruder, *An overview of gradient descent optimization algorithms*, CoRR, abs/1609.04747 (2016).

[34] A. J. Smola and B. Schölkopf, *A tutorial on support vector regression*, Statistics and Computing, 14 (2004), pp. 199–222.

[35] A. T. Takano, *An extremely brief introduction to orbits and constellations*. Aerospace Corporation Material, July 2019.

[36] J. G. Walker, *Continuous whole-earth coverage by circular-orbit satellite patterns*, Tech. Rep. 77044, Royal Aircraft Establishment, 1977.

[37] A. R. Washburn, *Earth coverage by satellites in circular orbit*, 2004.

[38] Y. Wu, H. Wang, B. Zhang, and K.-L. Du, *Using radial basis function networks for function approximation and classification*, ISRN Applied Mathematics, 2012 (2012).

[39] X. Yao, Y. H. Liu, and G. Lin, *Evolutionary programming made faster*, IEEE Trans. Evolutionary Computation, 3 (1999), pp. 82–102.

[40] Z.-H. Zhou, J. Wu, and W. Tang, *Ensembling neural networks: Many could be better than all*, Artificial Intelligence, 137 (2002), pp. 239 – 263.