Universiteit Leiden

Master Computer Science

Application of Multimodal Optimization Metaheuristics to Training of Artificial Neural Networks

Name:	Qinshan Sun
Student ID:	s3674320
Date:	2024-08-02
Specialisation:	Data Science
1st supervisor:	Niki van Stein
2nd supervisor:	Anna Kononova

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

Abstract

This thesis investigates the optimization of neural network parameters using evolutionary algorithms (EAs). While gradient-based methods are well-established in network training, EAs are also commonly used to aid optimization in this domain. However, researchers often overlook the multimodal optimization (MMO) techniques within EAs. This study aims to enhance EA population diversity and avoid local optima through MMO techniques to facilitate network parameter optimization. We primarily explore two niching methods, fitness sharing and dynamic fitness sharing, detailing their implementation, hyperparameter selection, and relationships with crossover and mutation. Additionally, we implement a hybrid algorithm combining EA with stochastic gradient descent (SGD). Comparative experiments indicate that incorporating MMO into traditional EAs and EA-SGD hybrid algorithms significantly improves their performance, approaching gradient-based methods such as SGD and ADAM. Nevertheless, these methods still underperform gradient-based methods in a fair comparison. The study concludes that MMO-enhanced EAs offer a powerful alternative to traditional methods, paving the way for further research in evolutionary deep learning and hybrid optimization techniques.

1 Introduction

Since the mid-20th century, an efficient and robust class of optimization algorithms known as evolutionary algorithms (EAs) has been widely applied to various optimization problems. By mimicking the process of biological evolution, EAs facilitate the evolution of a population of individuals toward an optimal state by iteratively conducting crossover/recombination, mutation, and selection. Although EAs cannot guarantee optimal solutions within a finite amount of time [7], they often provide reasonable solutions at a relatively low development cost. Utilizing various exclusive tools, EAs can be effective in search spaces of complex landscapes, addressing challenges posed by multimodality, discontinuity, constraints, and other optimization conditions [5, 14, 42].

Meanwhile, over the past decade, neural networks (NNs) have emerged as the most popular artificial intelligence model in computer science. They demonstrate outstanding learning capabilities and have found widespread applications. However, the tasks of data preparation, architecture design, parameter training, and model deployment for NNs are nontrivial and demand significant time and effort from practitioners. Accordingly, EAs become valuable tools for handling these tasks in NNs, such as optimizing the data features, the network architecture, and sometimes the network parameters [29]. The integration of EA and NN has evolved into a notable research direction known as Evolutionary Deep Learning (EDL) [29] or neuroevolution [16].

In this work, we focus on optimizing NN parameters, i.e., neural network training, which has a crucial impact on NNs' performance. Gradient-based methods based on backpropagation [45] (BP), such as stochastic gradient descent (SGD) and adaptive momentum estimation (ADAM), have long been the primary choice for training neural networks [8, 27, 44]. Alongside these methods, EA has also found its place in NN training, either training network parameters by itself [10, 24, 36] or in combination with gradient-based methods [12, 26, 51, 53].

Related works from [16, 29, 36] indicate that EAs are introduced to address the issue of gradient-based methods being trapped in local optima or saddle points, thereby finding the global optima. However, in most practical scenarios, there is not only a single global optimum for the parameters of neural networks. Instead, multiple distinct global optima exist, meaning that different sets of NN parameters exhibit similar performance and outperform each other on different test sets. Additionally, although each individual in the population of EAs represents a solution, there is a tendency for the diversity of solutions in the population to diminish gradually, converging towards a single solution, even when multiple optima exist in the search space [3, 4, 31].

Therefore, it is relevant to utilize multimodal optimization (MMO) metaheuristics to ensure the diversity of solutions in the population of evolutionary algorithms. Single-objective optimization is the task of obtaining the highest quality solution for a given objective function. In contrast, MMO aims to obtain a diverse set of solutions with fitness values exceeding a specified threshold [6].

As an extension of EAs, niching methods for MMO can obtain multiple optima within a single run by preserving the diversity of individuals in the population [46] based on their distances in the multimodal search space. With this approach, EAs can discover various structurally distinct parameter combinations, yielding favorable NNs. Moreover, different solutions from the population can be selected for different datasets or requirements in practice.

Among various niching methods available, we commence with two classical approaches, fitness sharing [18] and dynamic fitness sharing [34], and apply them to our designed evolution-based algorithms: an Evolution Strategy (ES), a Genetic Algorithm (GA), and GA_SGD. ES and GA represent the two fundamental types of EAs, while GA_SGD is our straightforward hybrid algorithm integrating SGD into GA.

In a series of experiments, the niching methods demonstrate significant improvements for both GA and GA_SGD, effectively diversifying the population and accelerating the optimization process of the algorithms. Although our evolutionary algorithms and hybrid algorithms are still slower than SGD in a fair comparison, the parallel optimization and diversified population of GA_SGD with niching present distinct advantages over SGD, showcasing superior performance in specific tests. These foundational results encourage further research into applying more MMO metaheuristics to complex EAs or EA-Gradient hybrid algorithms for NN training.

Through this thesis, we aim to explore the performance of EAs applied with multimodal optimization metaheuristics for neural network training. According to the taxonomy of EDL proposed in the survey [29], our work performs parameter optimization (PO) for neural networks using pure EAs or gradient-based EAs.

The remainder of this paper is organized as follows. Section 2 reviews existing research using evolution-based algorithms for training neural networks and provides motivation for this study. Section 3 elaborates on our EAs and niching methods, as well as the results of relevant parameter experiments and comparative experiments. Section 4 presents GA_SGD and its application of niching methods, showcasing and analyzing the results of comparative experiments and tests. Finally, Section 5 concludes the paper and discusses future work.

2 Related Work

Although related, the concepts of EDL and neuroevolution mentioned in the introduction are distinct. EDL encompasses the entire deep-learning process of data preparation, model generation, and model deployment [29]. For instance, it involves optimizing data features, optimizing network parameters or architecture, and pruning models using EAs.

On the other hand, neuroevolution primarily focuses on the model generation phase of deep learning. It utilizes EAs to optimize various components of neural networks, such as activation functions, hyperparameters, architectures, parameters (weights and biases), and even the optimization algorithms themselves [48].

In this discussion, we will focus on the work related to optimizing neural network parameters using EAs while disregarding other aspects to ensure the audience's understanding and engagement.

The research on using EAs to train neural network parameters emerged around the 1990s, yielding promising results as noted in section 2.1. However, as neural network models became more complex and gradient-based algorithms overcame challenges and achieved widespread success, researchers shifted their focus away from EAs in favor of methods like SGD. Despite this shift, the early successes of EAs have continued to inspire some researchers to employ EA-related methods for network training or to combine EAs with gradient-based algorithms to leverage their complementary strengths, as outlined in section 2.2 and 2.3. Additionally, when gradient information is challenging to obtain or inaccurate, EAs are effective. This advantage is mainly applied in training neural networks for reinforcement learning (RL), as detailed in section 2.4.

2.1 Early researches

In 1989, David J. Montana and Lawrence Davis utilized genetic algorithms (GA) to train feedforward neural networks (FNNs) [35]. They trained an FNN with only 126 parameters using 236 sonar data samples for classification. The results indicated that the GA optimization outperformed SGD regarding search efficiency. Although their evaluation method was biased in favor of GA by allowing more iterations, the study still demonstrated the potential of GA for optimizing small neural networks.

Their work inspired subsequent research in the 1990s. A 1995 study [41], using a similar sonar data classification task, included simulated annealing (SA) in addition to backpropagation and evolutionary algorithms for comparative experiments. The study showed that evolutionary algorithms outperformed gradient-based algorithms in pattern classification tasks over the same number of iterations.

Following that, Christian Goerick and Tobias Rodemann successfully employed the evolution strategy (ES, a subset of evolutionary algorithms) to optimize a model that backpropagation could not handle due to premature saturation of hidden neurons [17].

In 1999, Xin Yao summarized the integration of evolutionary algorithms and neural networks, focusing on optimizing network parameters, architectures, and input features [52]. The study analyzed various operators in evolutionary algorithms, highlighting the potential for combining evolutionary algorithms and neural networks to create powerful AI models.

Furthermore, M. Mandischer compared ES and backpropagation in training neural networks, concluding that ES could only compete with gradient-based algorithms on small-scale problems. ES was also adept at handling non-differentiable activation functions [32]. Additionally, the study found that as the NN parameters' dimensionality increased, ES's performance diminished, and the tuning requirements for ES became more stringent.

2.2 EAs for Training NNs

2.2.1 Genetic Algorithm

Karegowda *et al.* [24] employed the Pima Indians Diabetes Database (PIDD) to address the classification task of diabetes diagnosis using a simple GA-optimized multilayer perceptron (MLP) with a single hidden layer containing approximately 100 weights. The authors claimed that their GA-based approach outperformed gradient-based methods regarding accuracy; however, they did not specify the comparative methods, presumably based on the same number of generations.

In 2016, Morse and Stanley [36] proposed the Limited Evaluation Evolutionary Algorithm (LEEA), which, compared to traditional EAs, adopted minibatch training from gradient-based algorithms and introduced fitness inheritance of offspring from parents to mitigate the impact of varying fitness function, caused by different batches between generations. Their experimental results demonstrated that LEEA surpassed the traditional EA in learning efficiency when optimizing networks with over 1000 weights for three classic problems, performing comparably to SGD and RMSProp. However, their evaluation approach based on the same number of training examples neglected the complexity introduced by the population of EA, putting gradient-based methods at a disadvantage.

2.2.2 Others

Motivated by early studies such as [35, 41], EA variants akin to traditional EAs, such as Multi-phase Particle Swarm Optimization (MPPSO) [1], Artificial Bee Colony Optimization (ABC) [23], and Ant Colony Optimization (ACO) [47], have been utilized for the optimization of small-scale neural networks (typically with only around 100 to 200 weights or even fewer) in pattern classification tasks.

The results of [1] demonstrate that MPPSO exhibits better search efficiency and outcomes than backpropagation in classical pattern classification problems. [23] shows that ABC can achieve lower error rates than backpropagation given a longer time. Moreover, [47] combines ACO with the Levenberg-Marquardt (LM) algorithm, yielding superior performance compared to individual ACO, backpropagation, or LM.

Subsequent research also explored cooperative coevolution algorithms [10, 9], decomposing the original optimization problem to address subproblems. For instance, [10] merges the best-performing individuals from different subpopulations during cooperative evolutionary optimization, thereby treating a single hidden layer as a subcomponent and constructing new neural networks.

2.3 EAs Combined with Gradient for Training NNs

The preceding section shows that the standalone use of EAs often proves suitable only for small-scale problems. The challenges posed by increasing dimensions typically lead to their suboptimal performance when applied to larger-scale models. Consequently, researchers have increasingly integrated EAs with gradient-based methods, to alleviate issues such as local optima or saddle points encountered by gradient-based approaches, while leveraging gradient information to enhance the search efficiency. There are various ways to combine EAs with gradient-based methods, categorized as follows:

1. Embedding the gradient-based method at a specific location within the EA or integrating gradient information into an EA operator.

2. Utilizing the gradient-based method and the EA separately, regardless of order.

3. Utilizing the gradient-based method and the EA separately, regardless of order, in each generation of evolution.

Related works [13, 38, 51] fall into the first category. David and Greental [13] proposed a GA-assisted method for training autoencoders layer by layer from input to output. Experimental validation of their method on the MINIST dataset demonstrated lower reconstruction error and sparser networks compared to traditional backpropagation within the same runtime. Additionally, using Support Vector Machine (SVM) classification on the encoded embeddings from autoencoders with GA assistance yielded higher accuracy. Each generation of this method optimizes high-fitness individuals using BP, discarding low-fitness individuals and replacing their positions with high-fitness ones' offspring generated through BP, selection, crossover, and mutation. Utilizing uniform crossover helps avoid local optima, while the mutation of randomly setting weights to zeros ensures network sparsity.

Pawełczyk *et al.* [38] employed LeNet-4 for the multi-classification task on the MINIST dataset. While structurally similar to the [13] approach, the algorithm in this work incorporated different elitism methods, 1-point crossover, and 1-kernel mutation based on CNN (Convolutional NN) structures. Notably, it optimized the number of gradient descent steps alongside network parameters using GA, outperforming gradient-based learning.

Unlike the above two, the approach of Yang *et al.* [51] did not use gradient-based optimization methods directly. Instead, it incorporated gradient information into the crossover operator of EA (gSBX: Gradient-Based Simulated Binary Crossover), ensuring crossovers occur only in the direction of gradient descent. This method significantly improved the search efficiency of the EA, resulting in performance comparable to or even better than gradient-based methods.

Regarding the second category, examples like [22, 50] are notable. Ijjina *et al.* [22], for human action recognition, utilize GA to optimize CNN parameters initially, followed by gradient-based optimization, achieving higher classification accuracy than only gradient descent. In contrast, [50] applied gradient descent first, followed by GA optimization for facial expression recognition, demonstrating the effectiveness of EA in both cases.

The third category includes GADAM (Genetic Adaptive Momentum Estimation) and ESGD (Evolutionary Stochastic Gradient Descent) [53, 12]. In each generation, GADAM [53] optimizes the population of networks using ADAM, followed by further optimization through GA, selecting the best networks from both the populations after ADAM and GA for the next generation. On the other hand, ESGD [12], in each generation, selects an optimizer from a collection of gradient-based optimizers to optimize the population, followed by an evolution strategy. Unlike GADAM, the population for the next generation is only selected from the networks after ES optimization in ESGD. To ensure that the best fitness in the population never decreases, ESGD employs a back-off strategy during SGD steps and elitism within the evolution strategy.

Additionally, there are EA applications targeting the optimization of specific network structures, such as [30, 25] focusing on optimizing the attention layer and Modular Memory Unit (MMU).

2.4 EAs for Training NNs in Reinforcement Learning

In deep reinforcement learning (DRL), algorithms such as policy networks often rely on sparse or deceptive rewards to compute gradient information for network optimization. This results in inaccurate gradient information, adversely affecting the performance of gradient descent. Consequently, numerous studies [49, 40, 11, 26, 33] have utilized EAs to train policy networks in DRL.

In 2017, Such *et al.* [49] employed the genetic algorithm and novelty search (NS) to optimize convolutional neural networks (CNNs), demonstrating that simple, gradient-free, population-based GA could perform well on challenging deep RL tasks, including Atari games and humanoid locomotion. Using advanced computational settings and network parameter encoding based on random seeds, the GA successfully evolved networks with over four million free parameters, outperforming methods such as ES, Asynchronous Advantage Actor-Critic (A3C), and Deep Q-Network (DQN) in speed. The introduction of novelty search showed that following the gradient is not always the best choice for policy optimization, especially in tasks with deceptive or sparse reward functions, encouraging exploration and addressing high-dimensional problems where reward maximization algorithms like DQN, A3C, ES, and GA fail. Notably, their GA utilized only selection and mutation without crossover. Following this, three representative papers [40, 11, 26] in this field emerged in 2018.

Peng *et al.* [40] proposed a learning scheme utilizing an architecture comprised of a feature learning network and a policy learning network, where sensor inputs are first transformed into high-level features before being used for action prediction in the policy. With a population of networks, this scheme uses NeuroEvolution of Augmenting Topology (NEAT) to optimize the weights and architecture of the feature learning network and employs Policy Gradient search (PGS) algorithms to optimize the policy learning network. The three-stage PGS-NEAT-PGS training facilitated more effective knowledge sharing and learning across multiple agents. Experimental results on Atari games demonstrated that this novel learning scheme outperformed NEAT and several PGS algorithms' effectiveness and sample efficiency, particularly in large-scale and high-dimensional tasks.

Conti *et al.* [11] integrated novelty search and quality diversity (QD) into ES for optimizing deep policy networks, promoting directed exploration through a population of agents seeking novelty. Their proposed algorithms addressed environments with sparse or deceptive rewards, achieving higher overall performance by avoiding local optima. Experiments on simulated robotic control and Atari games showed that these hybrid algorithms outperformed standard ES and gradient-based methods like DQN and A3C.

khadka *et al.* [26] introduced EA into a classic policy gradient algorithm, DDPG (Deep Deterministic Policy Gradient), with an actor-critic mechanism to create the ERL (Evolutionary Reinforcement Learning) algorithm. ERL maintains a population of actors, along with an actor and a critic, throughout the learning process. In each generation, ERL uses GA's fitness evaluation, selection, crossover, and mutation to optimize the actor population, selecting policies that gain more rewards during episodes. Subsequently, the policy gradient updates the additional actor and critic, backward influencing the actor population. ERL also incorporates classic tricks such as replay buffer and target network to aid convergence. ERL demonstrated significant performance improvements over traditional policy gradient algorithms on six continuous control tasks simulated in Mujoco, achieving faster convergence and better final agent performance.

ES applications in DRL are extensive, with researchers considering ES's optimization of direction selection in the search space as an estimate of gradient information. For instance, [33] improved the accuracy of ES's gradient estimation and thus the convergence rate by integrating the gradient descent directions of past optimization steps.

2.5 Motivation

Upon reviewing related work, it is evident that the majority of studies did not consider the use of multimodal optimization metaheuristics to assist EA in optimizing neural networks. Furthermore, most related work utilized specialized versions of EA, with some studies targeting particular problems or employing unfair evaluation methods when comparing EAs with gradient-based approaches. Therefore, we aim to apply MMO metaheuristics to classical EAs to optimize neural networks and address common problems. Additionally, we plan to design a fair evaluation method to compare the performance of the algorithms, demonstrating the effectiveness of multimodal approaches.

3 Quantitative Comparison of EAs for NN Training

In section 3.1, we explain the task for our experiments on neural networks, the dataset employed, and the specific settings used to train the NN with different algorithms. Following this, section 3.2 provides a detailed exposition of the gradient-free algorithms utilized in our study, including Nevergrad black-box optimization algorithms, an evolution strategy, a genetic algorithm, and the niching methods. Corresponding to these algorithms, section 3.3 describes the conducted experiments on them and analyzes the results. At last, section 3.4 summarizes and discusses the findings of this section.

3.1 BBOB Function Data for Regression Task

To compare the performance of NN parameter-optimization algorithms, we need to build NN problems for numerical comparison.

Similar to [36], we use function approximation for the NN to solve. The functions selected are from the BBOB (Black-Box Optimization Benchmark) function set [15], which have different characteristics and can fulfill our requirements of multimodality and complexity. The general experiment process is as follows:

1. Generate 5000 training data samples of a 2-dimension BBOB function (selected from all 24 functions) using Coco experimenter [20]. Selecting two as the number of dimensions allows the ability to visualize the function approximated and is the same as the function approximation problem in [36].

2. Build a neural network: (2, 50, 20, 1), using PyTorch [37], which does not change during the training of NN. Such an architecture is chosen because it is simple enough while being able to model the BBOB functions. The two inputs and one output correspond to the 2d BBOB function with x_1 and x_2 as variables (inputs) and y as value (output).

3. Train the neural network using the generated training data with different optimization algorithms: SGD, ADAM, (1+1)-EA, ES, GA, GA-SGD... Gradient-based methods, SGD and ADAM, utilize PyTorch's built-in default implementation (torch.optim) with MSE (mean squared error) as the loss function. At the same time, evolutionary algorithms optimize by using the network's MSE on the training samples as the objective function.

4. Evaluate the training results based on the averaged loss curve or statistics over repetitions or further tests on test data.

To demonstrate the effectiveness of function approximation on the BBOB benchmark, we utilize the (2, 50, 20, 1) neural network for function approximation across all 24 BBOB functions, illustrated in Figure 1.

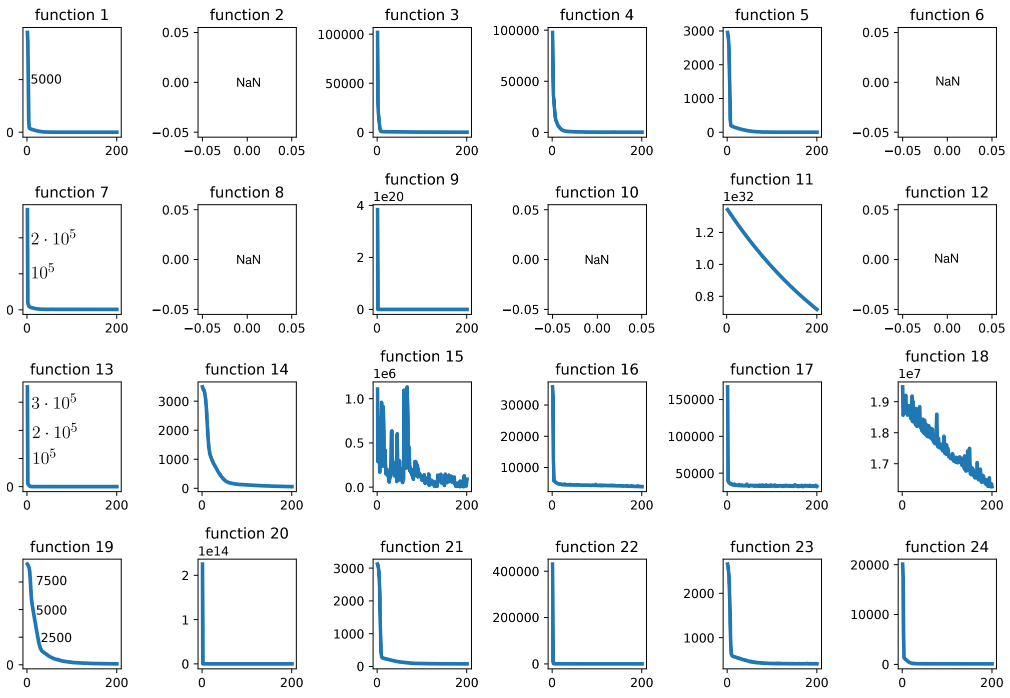


Figure 1: the learning loss curves for the function approximation of the 24 BBOB functions, optimizing MLP (2, 50, 20, 1) using SGD over 200 epochs, with the batch size as 64 and the learning rate as 0.00001.

The simple two-hidden-layer network can approximate most BBOB functions effectively, including functions 1, 3, 4, 5, 7, 13, 14, 16, 17, 19, 21, 22, 23, and 24. The learning curves for these functions show a rapid reduction in loss during the initial phase, indicating that the model can quickly and efficiently approximate these functions. In the later stages, the loss still decreases, although it descends too slowly to present a significant decline in the figure.

However, the losses during the learning process are very high for some functions, such as functions 9, 11, 15, 18, and 20. Although the loss curves decrease continuously over time or quickly at their beginning, they remain relatively high losses. This phenomenon indicates that these functions are particularly challenging for our simple network architecture. While the model approximates these functions to some extent, the accuracy is lower compared to other functions, highlighting the limitations of the network architecture.

Additionally, the approximation of functions 2, 6, 8, 10, and 12 encounters numerical instability issues, such as gradient explosion, resulting in NaN (Not a Number) losses and halting the training process. Further experiments reveal that these specific functions require extremely small learning rates. When an appropriate learning rate is used, the model's performance on these functions is similar to its performance on functions 9, 11, 15, 18, and 20, where the effective approximation is difficult to achieve.

In our subsequent experiments, we select a subset of BBOB functions that the (2, 50, 20, 1) network could approximate well for data generation, network training, and testing. The selected BBOB functions are functions 1, 3, 7, 13, 16, and 22, as depicted in Figure 2. Functions 1 and 3 are separable, while functions 7, 13, 16, and 22 each belong to different function categories [15]. The function names are provided below.

F1: Sphere Function; F3: Rastrigin Function; F7: Step Ellipsoidal Function; F13: Sharp Ridge Function; F16: Weierstrass Function; F22: Gallagher's Gaussian 21-hi Peaks Function.

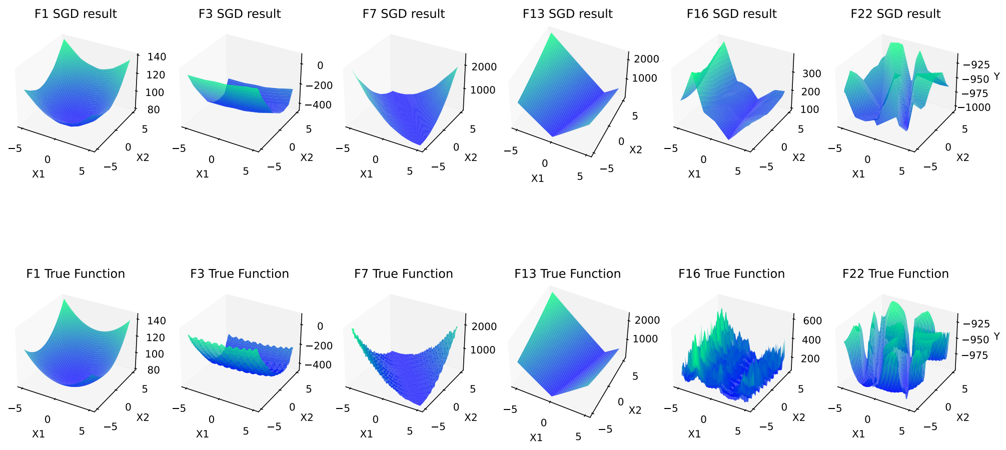


Figure 2: the visualization of NN-approximated functions and true functions, including F1, F3, F7, F13, F16, F22. The NNs here are trained for 1000 epochs, extending the experiment illustrated in Figure 1. This figure demonstrates the network architecture's ability to approximate these six functions.

To further investigate the reasons behind the network's inability to approximate certain BBOB functions, we examine the value ranges of all BBOB functions. We find that the BBOB functions where the network encounters approximation issues have significantly larger function values, whereas other functions successfully approximated have much smaller values.

Therefore, after applying min-max scaling to the training data's function values, we conduct our experiment in Figure 1 again, as illustrated in Figure 3. The results demonstrate that once the function value ranges are standardized to $[0, 1]$, our network architecture can approximate all 24 BBOB functions. However, in subsequent experiments, we do not standardize the function values and instead utilize the selected six functions that the network can already approximate without scaling their values.

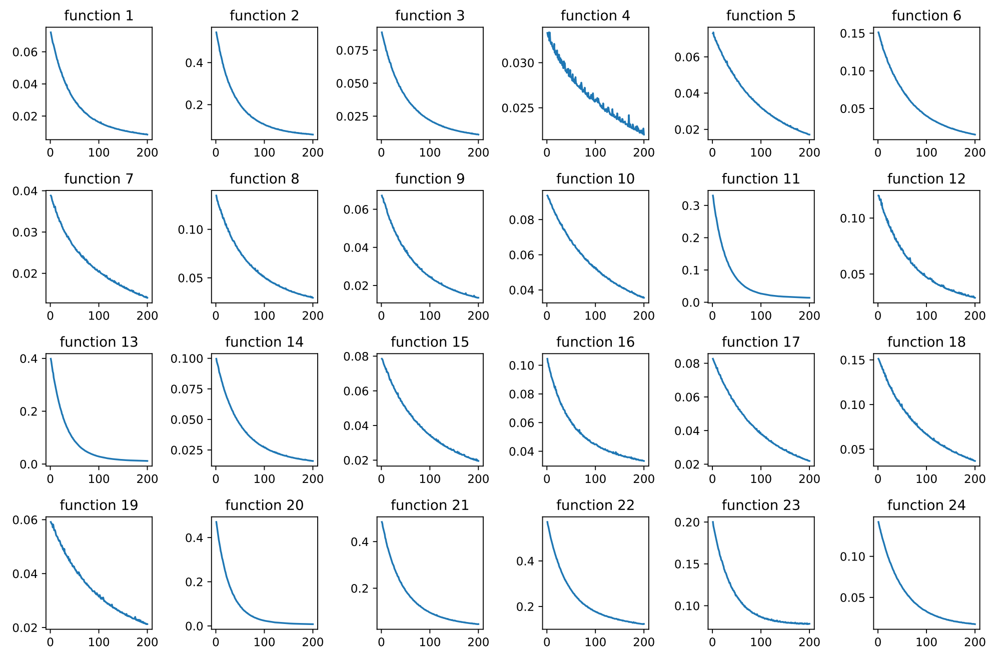


Figure 3: the learning loss curves for the function approximation of the 24 BBOB functions (with standardized values), optimizing MLP (2, 50, 20, 1) using SGD over 200 epochs, with the batch size as 64 and the learning rate as 0.00001.

3.2 Gradient-free Algorithms

Before implementing our evolutionary algorithms, we compare various black-box optimization algorithms implemented in Nevergrad with SGD, including evolutionary algorithms such as OnePlusOne and CMA, as described in section 3.2.1.

To explore the role of niching methods for MMO in NN parameter optimization, we apply them to two fundamental types of EA, the ES (Evolution Strategy) and GA (Genetic Algorithm). We utilize an ES and a GA of our custom design, detailed in sections 3.2.2 and 3.2.3. Subsequently, section 3.2.4 explains how we apply two niching methods, fitness sharing and dynamic fitness sharing.

3.2.1 Nevergrad

In this study, we utilize a selection of optimization algorithms from the Nevergrad [43] platform to compare their performance with SGD on NN optimization. The algorithms used are NGOpt (Nevergrad Optimizer), OnePlusOne ((1+1)-EA), CMA (Covariance Matrix Adaptation), TwoPointsDE (Two Points Differential Evolution), TBPSA (Test-based population-size adaptation), and RandomSearch.

NGOpt is a versatile, adaptive algorithm designed to dynamically select and combine different optimization techniques based on the characteristics of the problem, encompassing various other algorithms. OnePlusOne is a simple yet effective evolutionary algorithm that maintains a single solution and iteratively improves it through mutation. CMA is a robust evolution strategy that adjusts the covariance matrix for the mutation, enabling it to navigate complex and high-dimensional search spaces efficiently. TwoPointsDE is a variant of the Differential Evolution algorithm that optimizes using two-point crossover and differential mutation. TBPSA is an algorithm that adjusts the population size during optimization based on performance tests to balance exploration and exploitation. RandomSearch is a baseline algorithm that randomly samples solutions from the search space, providing a comparison point to evaluate the effectiveness of more complex algorithms.

Each algorithm offers unique strengths and weaknesses, making the algorithms well-suited for a comparative analysis surrounding NN optimization.

3.2.2 ES

Our evolution strategy comprises initialization, recombination, mutation, and selection. Each operator can be realized in various ways, while we opt for relatively simple implementations. Each operator of our ES is described below in detail.

Initialization: The initialization step generates the initial population of networks along with their corresponding standard deviations (σ). Each individual in the population is represented by an 1191-dimensional variable corresponding to the 1191 parameters of the utilized network. Thus, each individual represents a network and has an associated σ. In the initial population of size μ, network parameters are set to 0, and every σ is set to 1.

Recombination: During the recombination step, the algorithm employs discrete recombination for the network parameters and intermediate recombination for σ to generate an offspring population of size λ. Specifically, two parents are randomly selected for each offspring. Subsequently, each network parameter of the offspring is randomly chosen from the two corresponding parameters of the two parents. The σ value for the offspring is computed as the average of its two parents' σ values.

Mutation: The mutation step is carried out by applying the one-σ mutation. Initially, the mutation occurs on all σ as shown in Equation (1). Subsequently, each network in the population undergoes mutation by adding normally distributed random values scaled by its respective σ to each parameter, as described in Equation (2).

$$\sigma' = \sigma \exp(\tau_0 \mathcal{N}(0,1)) \tag{1}$$

$$x_i' = x_i + \sigma' \mathcal{N}(0,1) \tag{2}$$

The learning rate τ_0 is set to the recommended value of $\frac{1}{\sqrt{1191}}$. $\mathcal{N}(0,1)$ denotes the standard normal distribution, and x_i represents the i-th parameter of the network x.

Selection: The selection process employs the $(\mu + \lambda)$ selection mechanism. The parent and offspring populations are merged and evaluated using the objective function. Subsequently, the merged population is sorted based on the objective function values, and the top μ networks are selected for the next generation while retaining their corresponding σ values.

Evolution Strategy: The evolutionary strategy process begins with initializing a population and σ values. Subsequently, throughout each iteration, the algorithm undertakes recombination to generate offspring, mutation to introduce variation, and selection to identify the best individuals for the next generation. Throughout this process, the loss for each iteration is the objective function value from the best network in the entire population, with the number of iterations specified as the budget.

3.2.3 GA

The genetic algorithm includes initialization, fitness evaluation, parent selection, crossover, mutation, and selection. Each of these is explained below in detail.

Initialization: As the evolution strategy's initialization, the GA initializes a population of μ neural networks, with each network represented by 1191 network parameters. During the initialization process, the parameters of each network are randomly generated according to the standard normal distribution $\mathcal{N}(0,1)$. Unlike ES, GA does not utilize standard deviation for mutation, so it only initializes these network parameters.

Fitness Evaluation: After initialization, the algorithm evaluates the fitness of each network in the population by calculating the objective function value, that is, the MSE loss. Fitness is determined as the reciprocal of the objective function value. The optimization goal is to achieve lower objective function values corresponding to higher fitness scores.

Parent Selection: After evaluating fitness, the GA selects networks from the current population as parents for producing offspring. In this implementation, the roulette wheel selection (RWS) [19] with fitness scaling is used to choose parents based on their fitness, ensuring that better networks are more likely to be selected.

The algorithm first sorts the population in descending order based on fitness scores. It then elitistically selects the top 40% of the population for further selection. Before the RWS, the fitness values of these individuals are scaled: the minimum fitness value among them is subtracted from all fitness values, which are then divided by the total sum of the adjusted fitness values to convert them into selection probabilities ranging from 0 to 1. Finally, the roulette wheel selection is performed by generating random values and comparing them to the selection probabilities to simulate the roulette wheel. This process selects μ parents by spinning the roulette wheel μ times.

Crossover: Once the parent generation is selected, the GA proceeds with crossover and mutation to generate offspring. The offspring population generated by crossover is the input population of mutation. Three crossover types are implemented here, which can be described as uniform crossovers of network parameters, nodes, and layers. We default to using the uniform crossover of network parameters.

Through this process, each parent in the population is traversed precisely once to crossover, where two offspring are randomly generated from every two adjacent parents. Figure 4 depicts the generation of offspring. In each dimension of the individual, when one offspring selects one unit from one parent, the other offspring always selects the other unit from the other parent. Uniform crossover of network nodes involves treating the weights and the bias of each node as a single unit that will not be crossed, while uniform crossover of network layers treats the weights and biases of each layer as a single unit.

Mutation: After crossover, the mutation introduces genetic diversity into the population by making small random changes to the network parameters. In this implementation, each network has a mutation probability of 4%. Upon mutation, each parameter of the network is added with a value sampled from the uniform distribution $\mathcal{U}(-0.1, 0.1)$.

Selection: Finally, the GA selects the top μ networks with the highest fitness scores

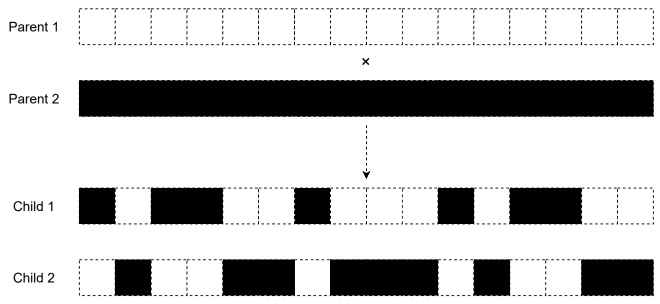


Figure 4: the uniform crossover.

from the offspring resulting from parent selection, crossover, and mutation, as well as from the initial population of this generation that did not undergo these operations, to form the population for the next generation.

Genetic Algorithm: The genetic algorithm process starts with initializing a population and calculating its networks' losses. Subsequently, in each iteration, the algorithm performs fitness evaluation using losses, parent selection based on fitness scores, large-scale changes through crossover, small-scale changes through mutation, and selection based on offspring evaluation to update the population. Like the ES, the objective function value of the best network in the population represents the loss of the GA in each iteration, with the number of iterations specified as a budget.

3.2.4 Niching

Our research primarily employs two classical niching methods: fitness sharing and dynamic fitness sharing. These methods are integrated with both the ES and the GA, merely modifying all the evaluations of individuals within ES and GA, specifically the objective function evaluations in the ES and the fitness evaluations in the GA.

Since the ES is not based on fitness and only minimizes the objective function, i.e., the MSE loss, we modify its objective function to the product of the (dynamic) niche count and the original objective function value to maintain the same logic as the niching methods described below.

In addition, during the selection process in ES and GA, the (dynamic) shared fitness is used to evaluate the current population and its offspring in one population, replacing the original evaluation.

Before explaining the sharing method, we define the term ”landscape”. A landscape is determined by a function and a search space, representing the surface of the function's values as they vary according to the varying search points in the search space.

In the context of NN parameter optimization, the search space of the landscape corresponds to the value range of network parameters, and the function of the landscape is the MSE loss of the network on the training data, i.e., the loss function of the gradient-based methods and the objective function of the evolutionary algorithms. Furthermore, the fitness landscape differs from the general landscape in that its function represents fitness values, which are to be maximized, rather than the MSE loss to be minimized.

Fitness Sharing: Sharing, a concept introduced in 1975 [21], is one of the earliest concepts of niching. Goldberg and Richardson [18] later utilized it as a niching technique for genetic algorithms, demonstrating its practical application. As a classical niching method primarily employed in GAs, fitness sharing has laid the foundation for various subsequent successful MMO niching methods and continues to be effectively applied in the field.

The core idea of fitness sharing is to treat the fitness of the fitness landscape as a resource shared among individuals in the population. By distributing this resource to individuals as evenly as possible, fitness sharing aims to reduce the prevalence of similar individuals within the population, where similarity is measured based on distance.

In abstract terms, fitness sharing creates a niche around each individual in the search space with a radius. Then the fitness of each individual varies based on the number of individuals contained within its niche, resulting in a reduction of fitness resources for closely situated individuals compared to their original fitness resources, where the varied fitness is the shared fitness. The following equations define the sharing function, niche count, and shared fitness:

$$sh(d_{i,j}) = \begin{cases} 1 - \left(\frac{d_{i,j}}{R}\right)^{\alpha_{sh}} & \text{if } d_{i,j} < R \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$m_i = \sum_{j=1}^{\mu} sh(d_{i,j}) \tag{4}$$

$$f_i^{sh} = \frac{f_i}{m_i} \tag{5}$$

In Equation (3), $d_{i,j}$ represents the distance between individuals i and j, implemented as the Euclidean distance between the parameters of networks i and j. The parameter R denotes the radius of each niche within the search space, and $\alpha_{sh} \geq 1$ is a control parameter set to the commonly used value of 1. When $d_{i,j}$ is greater than or equal to the niche radius, the sharing function value is 0. Otherwise, the closer individuals i and j are, i.e., the more similar networks i and j are, the higher the value of the sharing function, approaching 1.

The m_i in Equation (4) is the niche count, representing the sum of the sharing function values for individual i with respect to all individuals in the population. Its value is greater than or equal to 1 because the sharing function value of an individual with itself is 1. The m_i can be understood as the number of individuals, or networks, that are similar to individual i within its niche.

Equation (5) expresses the original fitness of individual i as f_i, specifically the reciprocal of the MSE loss of network i on the training samples. The shared fitness, f_i^{sh}, is then defined as the raw fitness divided by the niche count m_i. The shared fitness is strictly positive and to be maximized. Consequently, the larger the m_i, indicating more networks in the population similar to network i, the smaller the shared fitness, making it less likely to be selected by the evolutionary algorithms, aligning with the goal of multimodal optimization.

By introducing a fixed parameter niche radius R and assuming that the distances between optimal solutions are sufficiently large relative to R, fitness sharing can reduce redundancy in the gene pool, particularly around the peaks of the fitness landscape [46].

Dynamic Fitness Sharing: The dynamic fitness sharing [34] is an extension of the fitness sharing. In each generation, it dynamically identifies N peaks forming N niches within their niche radius R by Algorithm 1, and categorizes individuals as members of one of these niches or the non-niche domain according to their distances to the peaks. If the distance of an individual to a peak is smaller than R, the individual belongs to the niche formed by the peak. Please note that when using dynamic fitness sharing in our ES, we use Dynamic Peak Identification (DPI) to identify peaks only in the parent population. In contrast, in the GA, we apply DPI to the population integrating both the parent and offspring populations.

The sharing function of dynamic fitness sharing is the same as Equation (3). The dynamic niche count and the dynamic shared fitness are defined as follows:

$$m_i^{\text{dyn}} = \begin{cases} n_j & \text{if individual } i \text{ belongs to dynamic niche } j \\ m_i & \text{otherwise (individual i is non-niche)} \end{cases} \tag{6}$$

$$f_i^{\text{dyn}} = \frac{f_i}{m_i^{\text{dyn}}} \tag{7}$$

Algorithm 1 Dynamic Peak Identification (DPI) [34]

```
Require: $P$: population
Require: $N$: number of niches
Require: $R$: niche radius
 1: Sort $P$ in decreasing fitness order    ▷ Identify peaks in decreasing fitness order
 2: $i \leftarrow 1$    ▷ The index of individual in $P$ ranging from 1 to PopSize
 3: NumPeaks $\leftarrow 0$    ▷ The number of peaks identified
 4: DPS $\leftarrow \emptyset$    ▷ Dynamic Peak Set collecting peaks in the population
 5: while NumPeaks $\neq N$ and $i \leq$ PopSize do
 6:     if $P_i$ is not within sphere of radius $R$ around any peak in DPS then
 7:         DPS $\leftarrow$ DPS $\cup\{P_i\}$
 8:         NumPeaks $\leftarrow$ NumPeaks $+1$
 9:     end if
10:     $i \leftarrow i + 1$
11: end while
12: return DPS
```

The dynamic niche count m_i^{dyn} of individual i depends on its relationship with the peaks in the Dynamic Peak Set (DPS). When the distance between individual i and dynamic peak j is less than R, meaning that individual i belongs to dynamic niche j, the value of m_i^{dyn} is n_j, which is the number of individuals contained in dynamic niche j. Otherwise, if individual i does not belong to any peak, the value of m_i^{dyn} reverts to the standard niche count m_i as defined by Equation (4).

The logic of dynamic shared fitness f_i^{dyn} is analogous to shared fitness f_i^{sh}, incentivizing the evolutionary algorithm to explore more diverse individuals.

3.3 Experiments and Results

3.3.1 Nevergrad on BBOB F3

The built-in "ask" and "tell" functions in Nevergrad are employed for training. During each iteration, the code requests a single individual of recommended network parameters from the optimizer using "ask" and subsequently provides the optimizer with the objective function value (MSE) of the network via "tell", through which Nevergrad optimizes the network parameters using the configured black-box algorithms. This approach is consistent with the SGD training method, where the network parameters are updated once per iteration based on gradients derived from the whole dataset of 5000 data samples. For our experiments with Nevergrad, all Nevergrad algorithm parameters are kept at their default settings.

In the initial experiments, the learning rate for SGD was 0.0001, which was not optimal. This resulted in poorer performance compared to the OnePlusOne, as depicted in Figure 5. However, when the learning rate is 0.00005, SGD's performance surpasses that of all the Nevergrad black-box optimization algorithms, as shown in Figure 6.

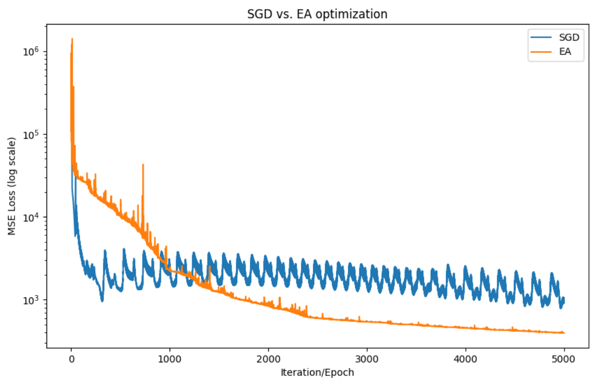


Figure 5: the loss curves of the NN learning process using SGD and (1+1)-EA for the (2,50,20,1) network on 5000 BBOB F3 data samples. We utilize the SGD and (1+1)-EA on the dataset without other training tricks like data standardization or data batches. The learning rate of SGD is 0.0001, and the (1+1)-EA is the default version of Nevergrad. The curves are from a single training run of the algorithms.

Experiments conducted on Nevergrad indicate that SGD requires an appropriate learning rate. For instance, a learning rate of 10^{-4} can cause fluctuations in the learning curve, making it challenging to continue reducing the loss. Conversely, a learning rate of 10^{-6} can decrease learning efficiency or lead to local optima.

The black-box algorithms in Nevergrad, while powerful, do not leverage gradient information from the parameters of the optimized network. This limitation, while posing a disadvantage during the optimization process, also presents an intriguing challenge for further research and improvement. When an evolutionary algorithm is only provided with the objective function value of a single individual per iteration, as SGD, its performance after the same number of iterations is inferior to that of SGD.

OnePlusOne stands out as the top performer among the Nevergrad algorithms tested, besides SGD. Its unique and straightforward logic, with a population size of one, is particularly well-suited to this training method, sparking interest in its potential applications.

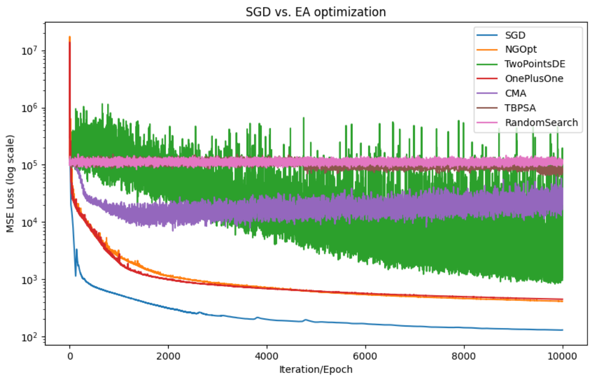


Figure 6: the loss curves of the NN learning process using SGD and Nevergrad algorithms for the (2,50,20,1) network on 5000 BBOB F3 data samples. We utilize the SGD and Nevergrad algorithms on the dataset without other training tricks like data standardization or data batches. The learning rate of SGD is 0.00005, and the Nevergrad algorithms are the default versions of Nevergrad. The curves represent the average loss over 10 independent training runs.

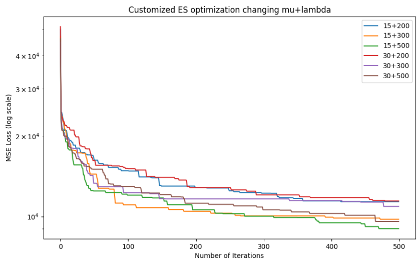
3.3.2 ES & Niching on BBOB F3

In the experiments with our ES, we initially tune the parameters μ and λ using the basic ES approach, as depicted in Figure 7a. Subsequently, for further parameter experiments with ES, we choose $(15 + 300)$ as the value for $(\mu + \lambda)$.

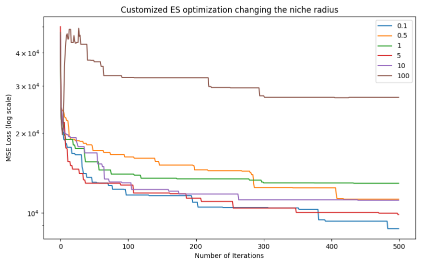
Next, we conduct experiments with fitness sharing on ES, revealing that a niche radius R of 0.1 yields the best performance, as shown in Figure 7b. Upon closer inspection, it is observed that the second best performance for ES_sharing occurs with R set to 5, outperforming other values between 0.1 and 5. As a result, it is conjectured that fitness sharing is nearly ineffective when $R = 0.1$, exerts some influence with R values between the two best settings, and becomes effective at $R = 5$.

Figure 7c presents the experiment results varying the number of niches for ES_dynamic, with R set to the previously best value of 0.1. The results indicate that ES performs best when the number of niches N is set to the maximal 15.

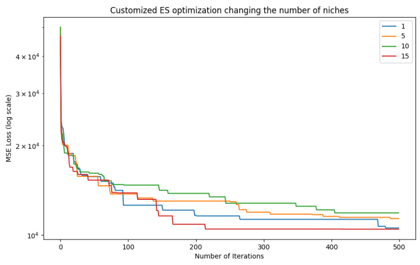
Based on these parameter experiments, both ES_sharing and ES_dynamic do not demonstrate significant performance enhancements over plain ES. This unfavorable outcome could be due to the algorithms not converging sufficiently yet requiring a longer training time. Alternatively, setting R too small may render fitness sharing ineffective. Or fitness sharing might require a larger population size μ to maintain multiple niches.



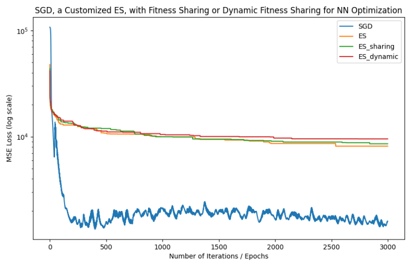
(a) Basic ES tuning the population size μ and offspring size λ



(b) ES_sharing (ES with fitness sharing) tuning the niche radius R



(c) ES_dynamic (ES with dynamic fitness sharing) tuning the number of niches N



(d) SGD vs. the basic ES, ES_sharing ($R = 5$), ES_dynamic ($R = 5, N = 15$)

Figure 7: the loss curves of the experiments on our evolution strategies training the (2,50,20,1) network on 5000 BBOB F3 data samples. We utilize the SGD and evolution strategies on the dataset without other training tricks like data standardization or data batches. The learning rate of SGD is 0.0001, and the evolution strategies are described in section 3.2. In Figures 7b and 7c, the $(\mu + \lambda)$ values for ES_sharing and ES_dynamic are both set to $(15+300)$, with the R for ES_dynamic in Figure 7c being 0.1. In Figure 7d, the $(\mu+\lambda)$ values for ES, ES_sharing, and ES_dynamic are $(15 + 300), (50 + 300), (50 + 300)$, respectively. The curves in the first three subfigures represent the average loss over 5 independent training runs, while the curves in the last subfigure represent the average loss over 10 independent training runs.

Therefore, in the comparative experiment shown in Figure 7d, the parameters μ and R for ES_sharing and ES_dynamic are increased to 50 and 5, respectively. However, these niching methods still do not benefit ES here.

Our ES and GA's training methodology differs from the "Ask&Tell" approach used in Nevergrad experiments. Here, they evaluate and update the entire population of networks per iteration, as described in the algorithm section.

3.3.3 GA & Niching on BBOB F3

Unlike the parameter experiments conducted for ES, the parameter experiments for GA omit the search for the population size μ. Instead, they only sequentially adjust the niche radius R in GA_sharing and the number of niches N in GA_dynamic. Here, we omit the parameter experiment plots and only present the final comparison results among SGD and GAs. As shown in Figure 8, with appropriate parameter settings, both GA_sharing and GA_dynamic enhance the performance of GA. Moreover, with the same number of iterations, GAs achieve a lower loss compared to SGD, although this comparison is biased due to the utilization of the population in GAs.

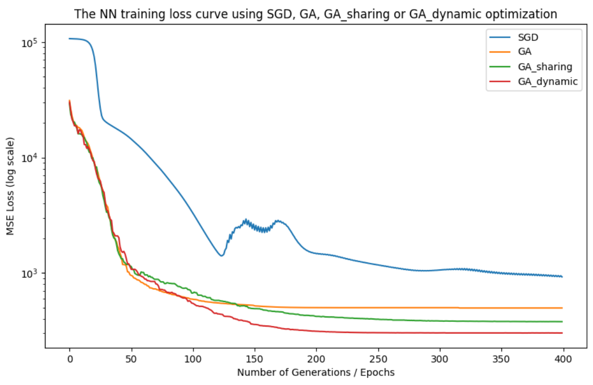


Figure 8: the loss curves of the NN learning process using SGD and GAs for the (2,50,20,1) network on 5000 BBOB F3 data samples. We utilize the SGD and GAs on the dataset without other training tricks like data standardization or data batches. The learning rate of SGD is 0.00005, and the GAs are described in section 3.2 with population size $\mu = 1000$. For the GA_sharing (GA with fitness sharing), the niche radius $R = 5$. For the GA_dynamic (GA with dynamic fitness sharing), the niche radius $R = 5$ and the number of niches $N = 50$. The curves represent the average loss over 3 independent training runs.

Additionally, Table 1 presents the statistics on the losses of all networks obtained from three repeated runs for GA, GA_sharing, and GA_dynamic. The table demonstrates the successful application of the niching method by showing the number of peaks (n_solution) in each algorithm's population, indicating effective dispersion of individuals within the population. Furthermore, based on other statistics, it is evident that GA_sharing is more stable than the other two methods, while GA_dynamic achieves the lowest losses among the three.

Table 1: the table of loss statistics, calculated on the collection of $3000 = 3 * \mu$ final solutions from 3 repeated runs, including the GA, GA_sharing, and GA_dynamic. The collections are first filtered by DPI ($R = 5, N = 3000$), which only saves the peak set of solutions (discards others) for statistical analysis of their losses. The n_solution represents the number of solutions, i.e., the size of the peak set.

Algorithm	mean_loss	best_loss	std_deviation	std_error	1Q	2Q	3Q	n_solution
GA	497.20	470.91	40.60	23.44	473.82	476.72	510.34	3
GA_sharing	384.38	337.17	**31.65**	**0.58**	345.61	388.74	420.66	**2960**
GA_dynamic	**304.03**	**240.00**	46.51	0.85	**241.52**	**318.11**	**352.50**	2959

3.3.4 GA & Niching on BBOB F1, F3, F7, F13, F16, F22

Before starting the experiment, we modify the experimental setup to ensure a more fair and comprehensive comparison of various optimization algorithms. First, we standardize the input variables of the 5000 training samples using the *StandardScaler* from the *scikit-learn* package [39]. Additionally, we introduce ADAM optimization and incorporate batch learning with batch size 64 for both SGD and ADAM. To ensure that gradient-based methods perform well when training the network to approximate various BBOB functions, we fix the learning rate at 0.00001.

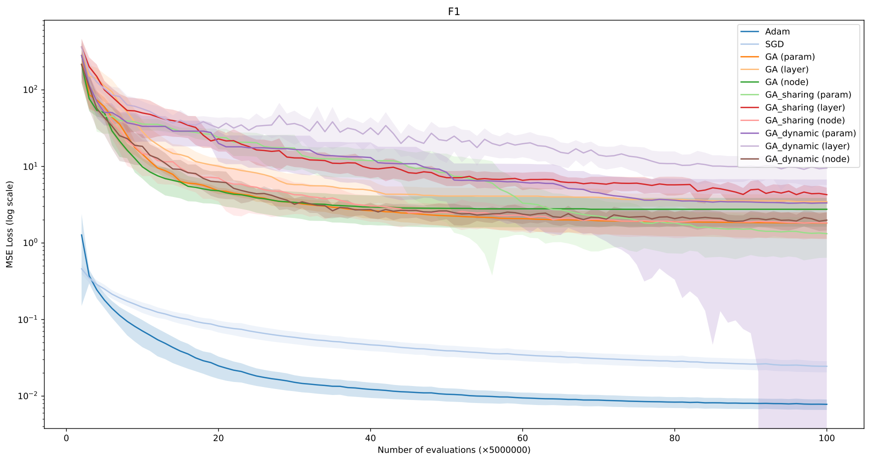
For the GAs, we maintain their parameter settings the same as those described in section 3.3.3, including population size ($\mu = 1000$), mutation rate ($p_m = 0.04$), niche radius ($R = 5$), and the number of niches ($N = 50$). However, we introduce different crossover types into the GAs. Besides the default crossover, which treats each network parameter as a unit, we also include crossover methods that treat the parameters of a network node or layer as a unit, as described in section 3.2.3.

More importantly, we change the comparison between gradient-based and evolution-based algorithms from being based on the same number of iterations to being based on the same number of network traversals. The x-axis of the loss curves now represents network traversal times instead of iterations.

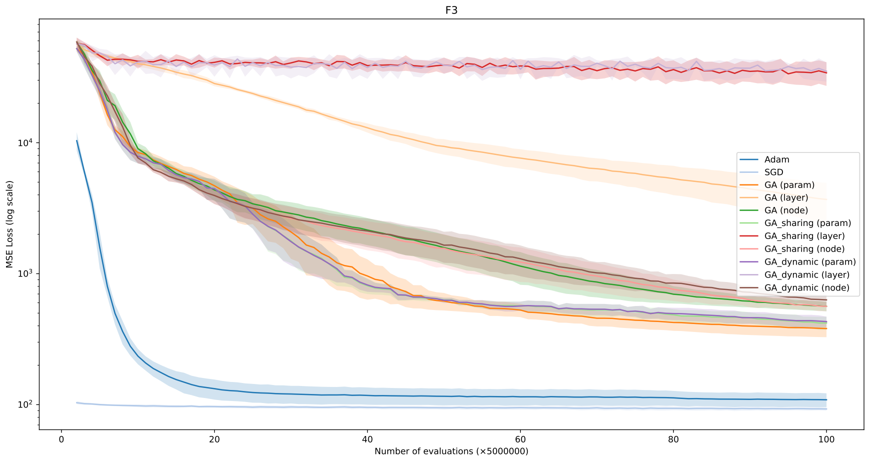
For gradient-based methods, during each epoch, the 5000 samples in the training set undergo one forward pass and one backward pass through the network, resulting in 10000 network traversals per epoch. For GAs, during each generation, the 1000 networks in the population each undergo a forward pass of the 5000 samples in the training set, resulting in 5000000 network traversals per generation. Notably, the first generation of the GAs requires an additional 5000000 network traversals to evaluate the initial population.

Therefore, in the plots, the loss of the 1st generation of the GAs is aligned with the loss of the 1000th epoch of the gradient-based methods, and the loss of each subsequent generation is aligned with the loss every 500 epochs. This comparison method still has a problem, discussed in section 4.2.

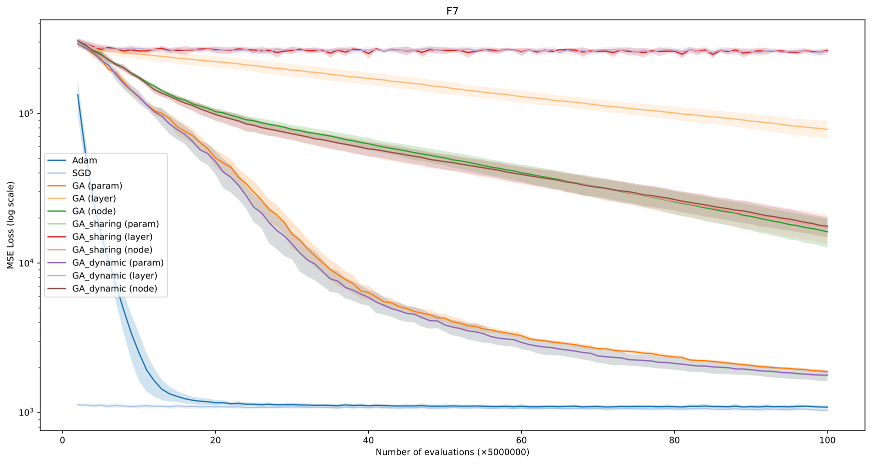
Based on previous results, we now aim to explore further the performance of the GA combined with fitness sharing across the entire problem set, approximating F1, F3, F7, F13, F16, and F22 using the neural network. Figure 9 corresponds to the training process of various optimization algorithms for approximating these six functions.



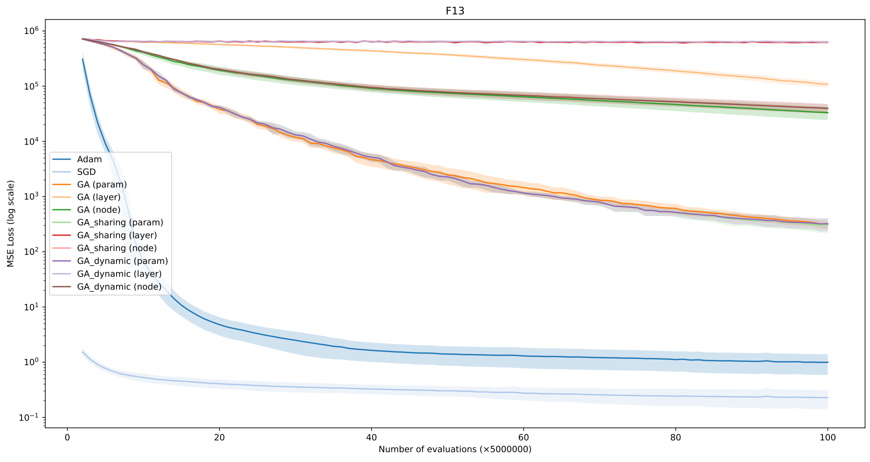
(a) F1



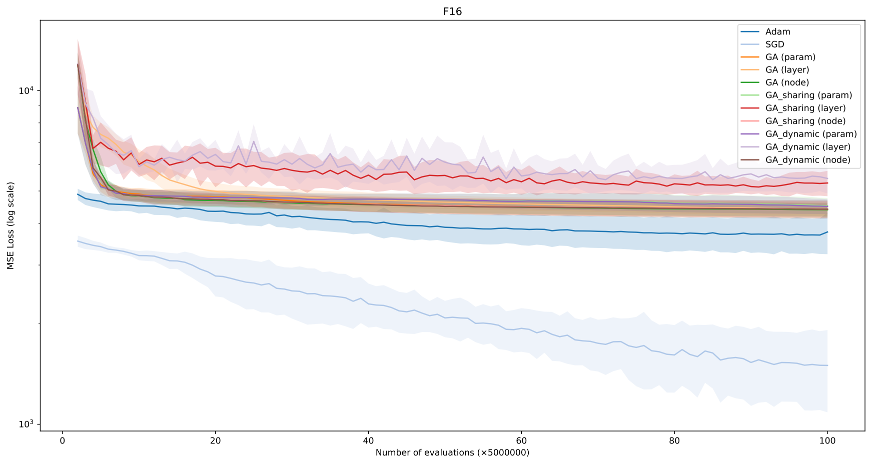
(b) F3



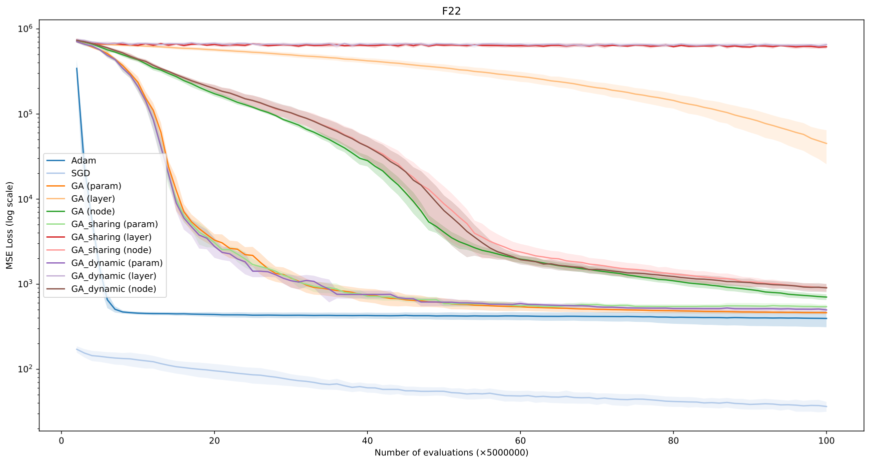
(c) F7



(d) F13



(e) F16



(f) F22

Figure 9: the loss curves of the NN learning process using ADAM, SGD, GA, GA_sharing, and GA_dynamic for the (2,50,20,1) network on 5000 standardized data samples of BBOB function 1, 3, 7, 13, 16, and 22. The learning rate of the gradient-based methods is 0.00001, and their batch size is 64. The GA, GA_sharing, and GA_dynamic of different crossover types are described in section 3.2 with population size $\mu = 1000$. For GA_sharing, the niche radius $R = 5$. For GA_dynamic, the niche radius $R = 5$ and the number of niches $N = 50$. The curves represent the average loss over 5 independent training runs, and the shaded areas represent the standard deviation confidence intervals across runs.

Using the same hyperparameters, SGD consistently outperforms Adam except for F1. Because these two gradient-based algorithms require different learning rate settings, and the settings used here are more suitable for SGD. Under the same number of network traversals, both SGD and Adam outperform GAs on all the BBOB function approximation problems.

Additionally, crossover at the layer or node level performs poorly. Layer-level crossover has a more negative impact than node-level crossover, as evidenced by the highest loss curves corresponding to GAs using layer-level crossover. This negative effect is more pronounced for GA_sharing and GA_dynamic than the plain GA.

For GAs using node crossover, the learning process significantly slows down in approximating functions 3, 7, 13, and 22 compared to the default parameter crossover. However, for functions 1 and 16, node-level crossover performs similarly to parameter-level crossover.

When the number of generations is less than 100, fitness sharing methods do not show an apparent improvement for GAs, although there is a slight enhancement for some functions like F1 and F7. This implies that the multimodal optimization approach requires more generations to be effective in the later optimization stages. Meanwhile, the niche radius and the number of niches need to be further tailored to each problem, as values suitable for F3 may not suit other function approximation problems.

Therefore, after modifying the experimental details as previously mentioned and extending the time to allow GAs to evolve for 400 generations, we reproduce the results for section 3.3.3 to confirm that fitness sharing methods still work. As shown in Figure 10, the curves indicate that after adding data standardization, fitness sharing methods continue to improve GA performance, while the difference between dynamic fitness sharing and fitness sharing becomes negligible.

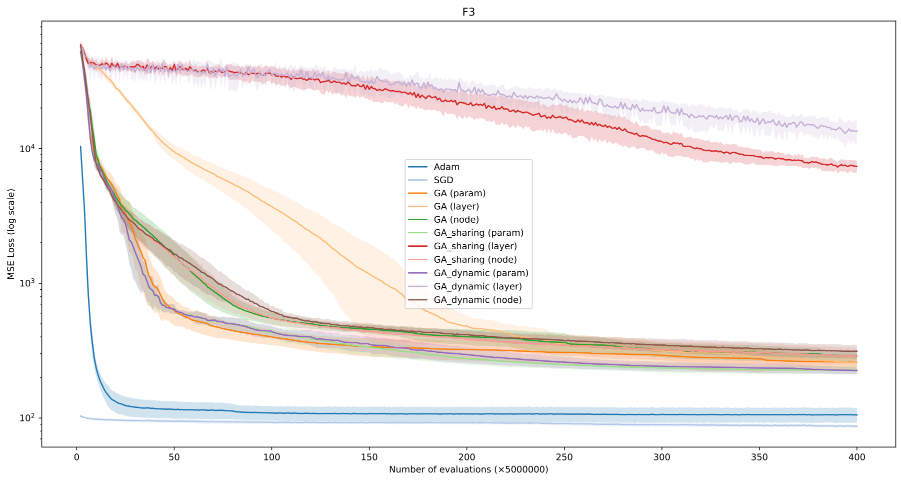


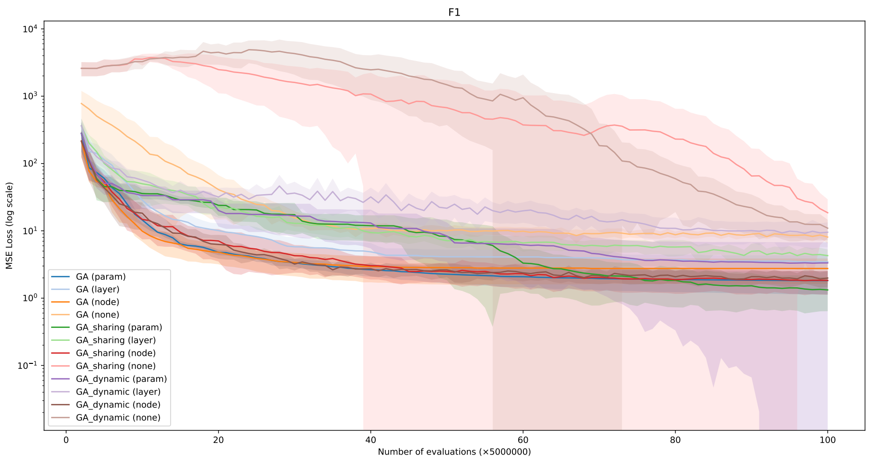
Figure 10: the loss curves for the ADAM, SGD, and GAs, as an extension of the experiment shown in Fig. 9b, with its time budget extended to 400 GA generations.

To investigate the impact of crossover further, we conduct ablation experiments on the crossover, exploring the performance of GAs without crossover. The results are as Figure 11.

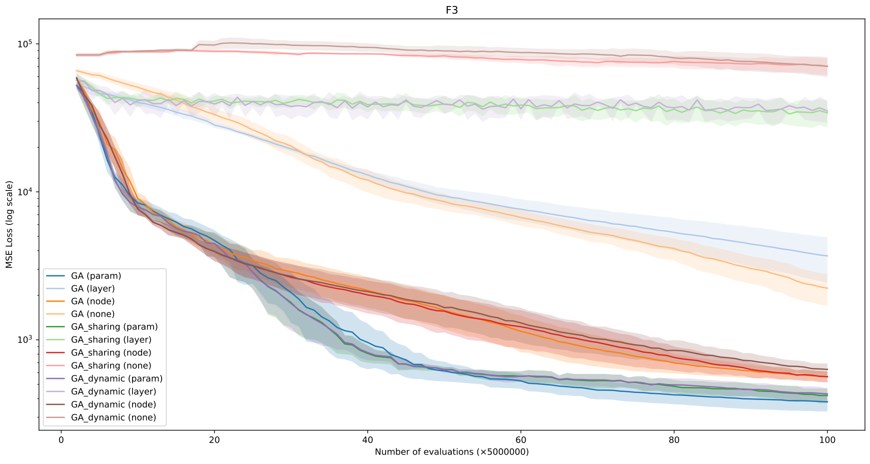
When not utilizing a crossover, GA_sharing and GA_dynamic exhibit the poorest performance compared to other crossover types. However, the plain GA without crossover still often performs slightly better than using crossover at the network layer level, which means both yield better results than the layer-crossover GA_sharing and GA_dynamic.

Analyzing the relationships among the performances of various crossover types or GAs, we observe a strong correlation between the use of fitness sharing and the presence of crossover. For fitness sharing to perform well, the crossover must be employed, preferably happening on each network parameter.

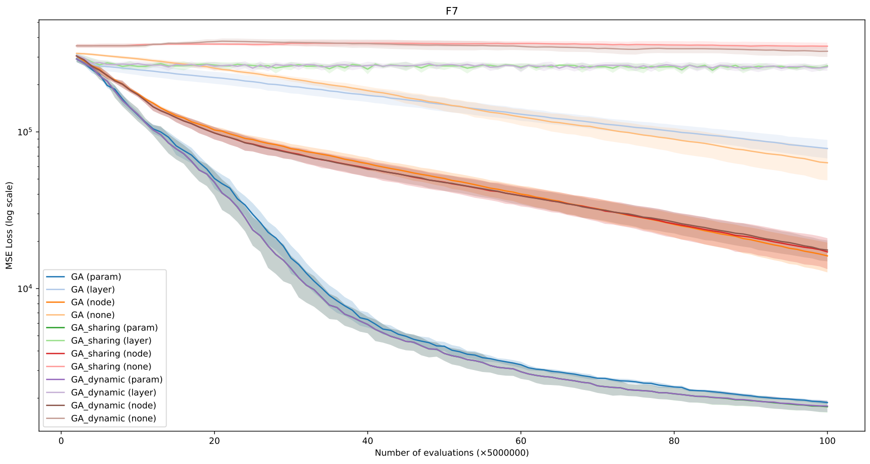
According to our analysis, fitness sharing's intention of dispersing the population within the search space can partially hinder mutation, thereby reducing the algorithm's exploitation of local areas. If there is not a sufficiently thorough crossover to escape local regions and conduct global searches, an algorithm that continues to use fitness sharing may become self-limiting and struggle to improve. Fitness sharing can only effectively disperse the population across a broader space to aid optimization when used in conjunction with crossover.



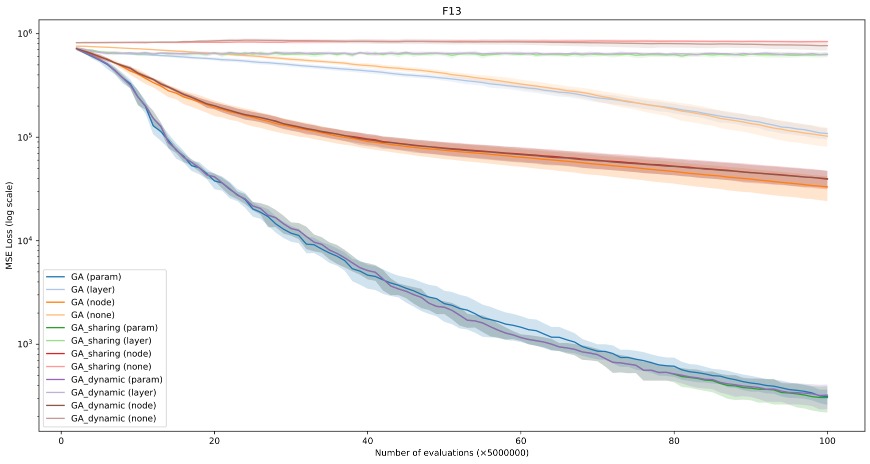
(a) F1



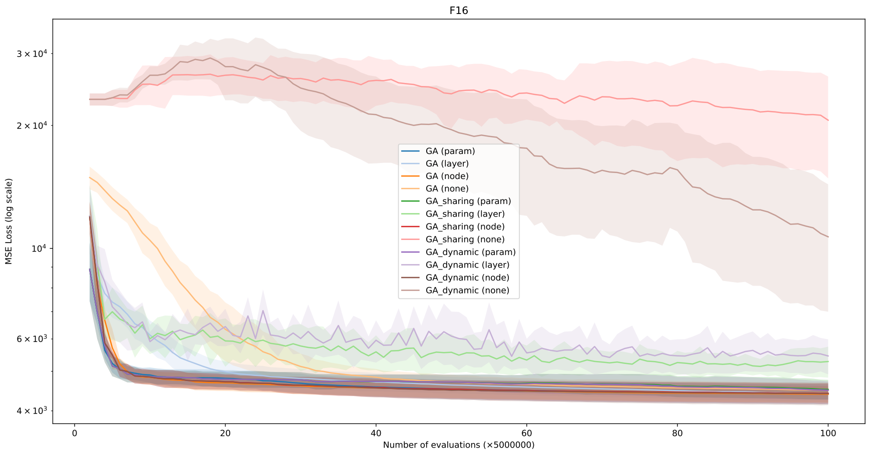
(b) F3



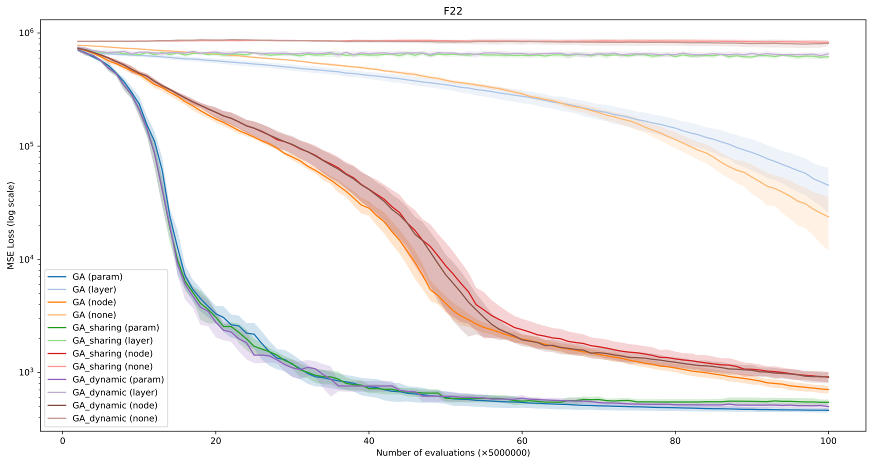
(c) F7



(d) F13



(e) F16



(f) F22

Figure 11: the loss curves of the NN learning process using GA, GA_sharing, and GA_dynamic with layer, node, parameter, or no crossover for the (2,50,20,1) network on 5000 standardized data samples of BBOB function 1, 3, 7, 13, 16, and 22. The GA, GA_sharing, and GA_dynamic of different crossover types are described in section 3.2 with population size $\mu = 1000$. For GA_sharing, the niche radius $R = 5$. For GA_dynamic, the niche radius $R = 5$ and number of niches $N = 50$. The curves represent the average loss over 5 independent training runs, and the shaded areas represent the standard deviation confidence intervals across runs.

3.4 Discussion

In comparative experiments with the commonly used black-box algorithms in Nevergrad, the simple evolution strategy, and the simple genetic algorithm, the gradient-based methods demonstrate absolute superiority in our network optimization problems. This finding underscores the crucial role of precise gradient information in optimizing efficiency. Notably, among the commonly used black-box algorithms in Nevergrad with default settings, OnePlusOne exhibits the highest network optimization efficiency, performing similarly to SGD with a suboptimal learning rate.

When employing our simple ES, the network losses are higher than those of OnePlusOne, GAs, and SGD, and the niching methods do not enhance its performance. This could be attributed to the evolution strategy's homogeneous initialization, which restricts the optimization path. Meanwhile, the one-sigma mutation is simplistic, limiting the variation across different variable dimensions. Additionally, the appropriate population size for ES is small, and its discrete recombination with no filtering of flawed individuals is too random to be effective. These factors and the data without standardization constrain the ES performance and make niching ineffective in dispersing the limited population's distribution to improve outcomes.

On the other hand, our simple GA has shown promise. Its advantage of a large population size enables the network optimization performance to surpass SGD within the same number of iterations. However, it still lags behind gradient-based methods with batch learning and data standardization during the same number of network traversals. When applied with appropriate parameters, the niching methods help the GA explore the search space more extensively, resulting in lower training losses in the later optimization stages. Furthermore, uniform crossover for each network parameter proves to be the most effective in all crossover types. Adequate crossover is crucial to ensuring the effectiveness of the niching methods.

Our GA's significant improvement over our ES is attributed to its larger population size, initialization based on a normal distribution, fitness-based parent selection mechanism, uniform crossover compared to discrete recombination, and different mutation using a uniform distribution.

In summary, due to the low exploration efficiency of mutations in evolutionary algorithms and the simplicity, speed, and effectiveness of gradient-based methods, incorporating gradients into GA to assist population updates undoubtedly will enhance network parameter optimization in GA, further supported by utilizing GA's crossover and niching methods. The proposed algorithm in section 4 encapsulates this promising idea.

4 Proposed GA_SGD_sharing

4.1 Algorithm

Algorithm 2 outlines the GA_SGD, which integrates the previous GA and SGD. This algorithm is similar to the algorithms proposed in the related works [13, 38, 51] as described in section 2.3, where gradient-based methods are embedded within GA for neural network optimization. However, unlike these works, our algorithm utilizes our custom GA with traditional simple operators and does not employ any operator introducing stochastic mutation, making it relatively simpler.

Algorithm 2 GA_SGD

```
Require: $\mu$: population size
Require: Net: network (2,50,20,1)
Require: $F_{obj}$: objective function
Require: $N_g$: number of generations
Require: $N_e$: number of epochs
Require: $lr$: learning rate
Require: $B$: batch size
Require: $D$: training data
Require: $C$: criterion such as MSE
 1: $P \leftarrow$ INITIALIZATION($\mu, Net$)
 2:                                            ▷ Initialize the population $P$
 3: PopLoss $\leftarrow F_{obj}(P)$
 4:                              ▷ Calculate the losses of the individuals in $P$
 5: for $i = 1$ to $N_g$ do
 6:                                            ▷ Start evolving generations
 7:     PopFitness $\leftarrow$ FITNESS_EVALUATION(PopLoss, $P$)
 8:                      ▷ Evaluate the fitness values of the individuals in $P$
 9:     Parents $\leftarrow$ PARENT_SELECTION(PopFitness, $P$)
10:          ▷ Select the parents from the individuals in $P$ according to PopFitness
11:     Parents $\leftarrow$ SGD_STEPS(Parents, Net, $D, C, N_e, lr, B$)
12:                      ▷ Optimize the parents using SGD with specified settings
13:     Offspring $\leftarrow$ CROSSOVER(Parents)
14:                        ▷ Generate Offspring by the crossover of Parents
15:     OffspringLoss $\leftarrow F_{obj}$(Offspring)
16:                      ▷ Calculate the losses of the individuals in Offspring
17:     PopLoss, $P \leftarrow$ SELECTION(OffspringLoss, PopLoss, Offspring, $P$)
18:             ▷ Evaluate the fitness values of the individuals in $P$ and Offspring
19:                    ▷ Select $P$ and update PopLoss for the next generation
20: end for
21: return PopLoss, $P$
```

The INITIALIZATION, FITNESS_EVALUATION, PARENT_SELECTION, CROSSOVER, and SELECTION in GA_SGD are consistent with the operators described in section 3.2.3 for GA. After parent selection and before crossover, the selected parents undergo SGD optimization for the specified number of epochs N_e with learning rate lr and batch size B. Consistent with the application of (dynamic) fitness sharing in GA as described in section 3.2.4, GA_SGD_sharing or GA_SGD_dynamic incorporates (dynamic) shared fitness calculation during FITNESS_EVALUATION and SELECTION within GA_SGD, in the lines 7 and 17 of Algorithm 2.

However, unlike section 3.2.4, in GA_SGD_sharing or GA_SGD_dynamic, the current population P and its offspring Offspring in SELECTION are not first merged into a single population for shared fitness calculation. Instead, they are treated as two separate populations, with shared fitness calculated individually for each. This method eliminates the mutual influence between parents and offspring when considered together, reducing the demand for sparse distribution search. Consequently, it enhances the exploration of the local optimization space, allowing the selection of more individuals with lower loss values to advance to the next generation.

4.2 Experiments and Results

The experimental setup for GA_SGD is identical to that in section 3.3.4, using the same dataset, number of network traversals, and other relevant configurations. Algorithms are employed to optimize the NN to approximate BBOB functions with GA_SGD's default parameter settings as specified in Table 2.

Table 2: the table of hyperparameters for Algorithm 2 in the experiments.

N_g	μ	N_e	lr	B	C	D	F_{obj}	Net
100	200	2	0.00001	64	MSE	5000 standardized BBOB F1/F3/F7/F13/F16/F22 samples	MSE on D	(2, 50, 20, 1)

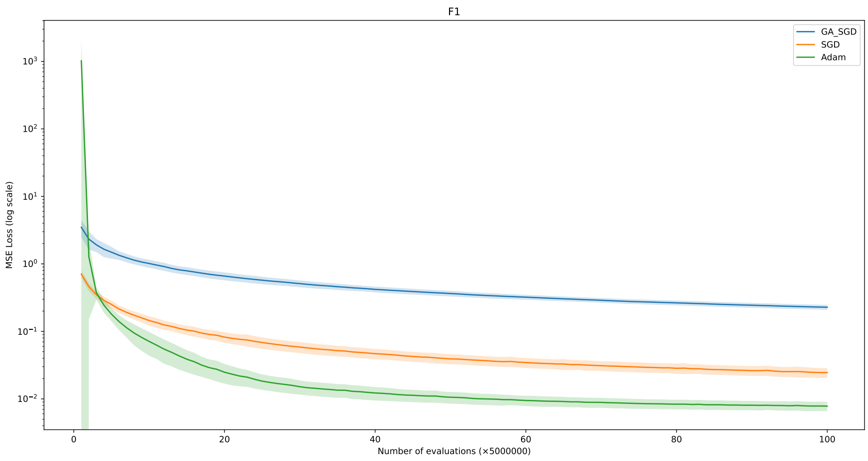
4.2.1 GA_SGD on BBOB F1, F3, F7, F13, F16, F22

Initially, we compare GA_SGD with SGD and ADAM across the entire problem set, ensuring that SGD and ADAM use the same learning rate, batch size, and loss function as GA_SGD. The results are depicted in Figure 12.

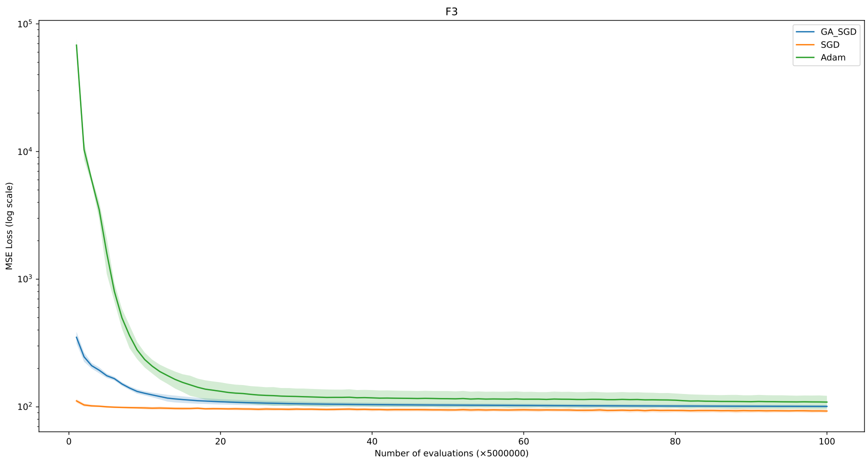
While GA_SGD's performance slightly lags behind SGD for the same number of network traversals, it nearly matches SGD on specific functions like F3 and F7. Meanwhile, GA_SGD generally outperforms the non-optimal ADAM except for F1 and F13.

Specifically, the simple GA_SGD, which effectively runs SGD optimization in parallel over 200 ($= N_g \cdot N_e$) epochs by utilizing GA, competes with the SGD or ADAM running over 50000 epochs in the experiments. By mitigating SGD's randomness in search paths with GA's population and operators, GA_SGD improves the optimization efficiency per epoch in parallel while decreasing the optimization efficiency per epoch or network traversal. Because the GA_SGD expends 400 ($= \mu * N_e$) epochs during a single generation for only 2 ($= N_e$) epochs in parallel, while the SGD and ADAM utilize this budget to run 500 epochs straightly.

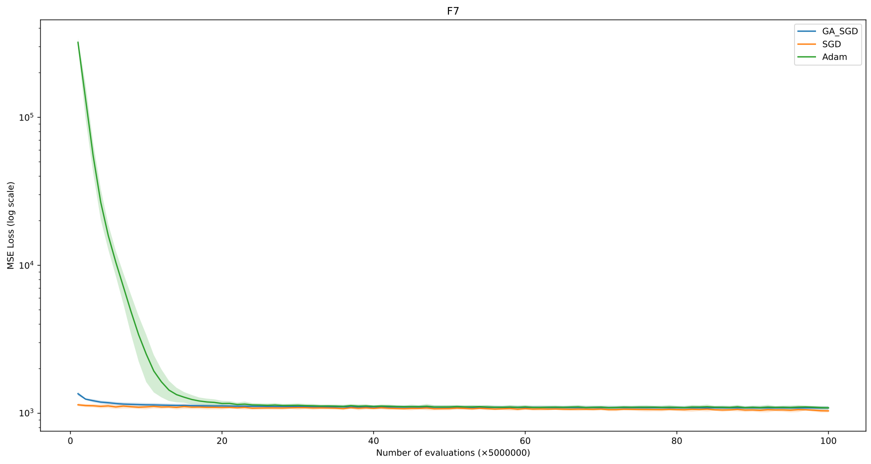
The parallel optimization characteristic of GA_SGD, based on its population and consuming more computational resources, is also one of the reasons why it is difficult for GA_SGD to surpass SGD under the current fair evaluation.



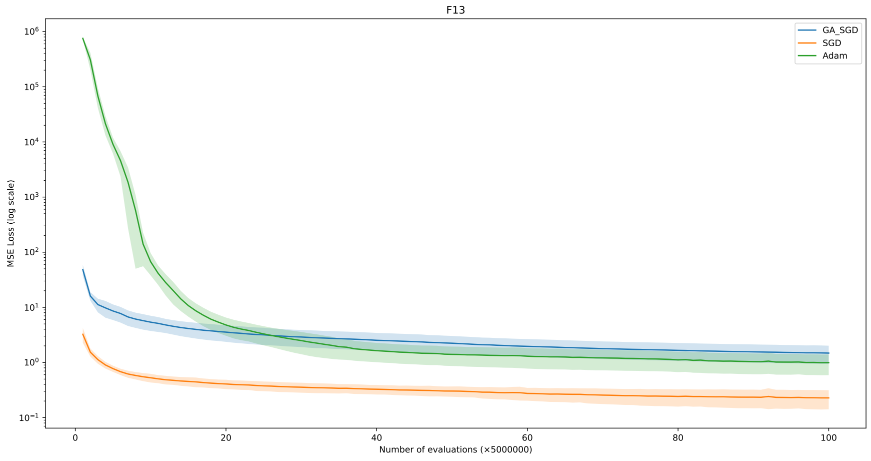
(a) F1



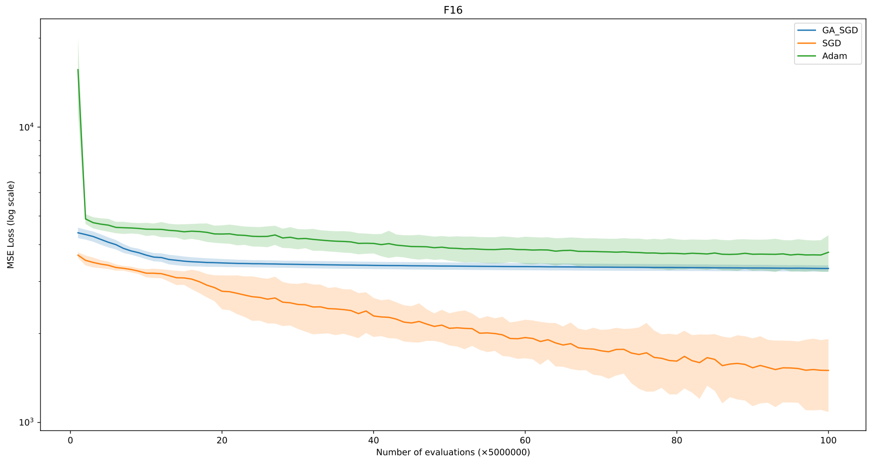
(b) F3



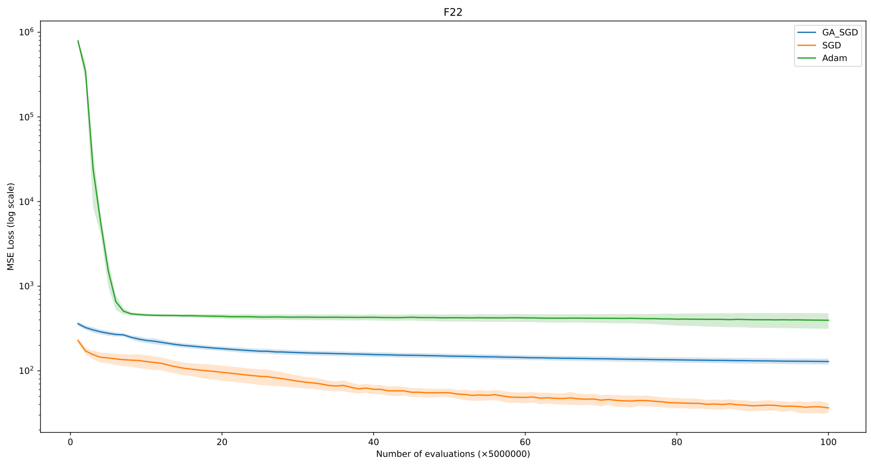
(c) F7



(d) F13



(e) F16

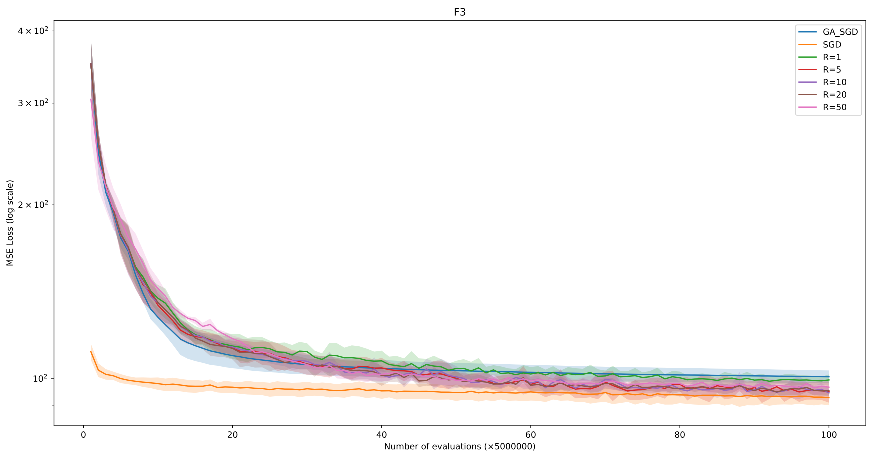


(f) F22

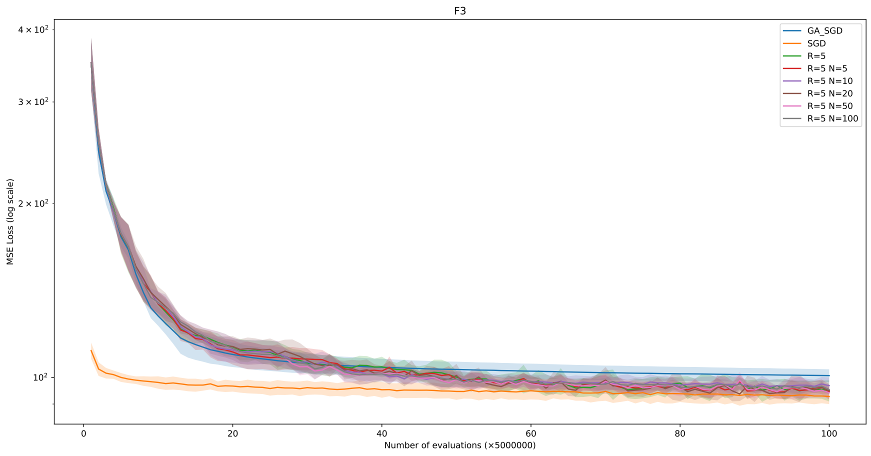
Figure 12: the loss curves of the NN learning process using GA_SGD, SGD, and ADAM for the (2,50,20,1) network on 5000 standardized data samples of BBOB function 1, 3, 7, 13, 16, and 22. The GA_SGD is described in section 4.1 with default parameter settings in Table 2. The learning rate of all SGD and ADAM steps is 0.00001, and their batch size is 64. The curves represent the average loss over 5 independent training runs, and the shaded areas represent the standard deviation confidence intervals across runs.

4.2.2 GA_SGD & Niching on BBOB F3

To explore the impact of niching methods on GA_SGD, we first conduct experiments on BBOB F3 by incorporating (dynamic) fitness sharing into GA_SGD, as in Figure 13.



(a) GA_SGD_sharing varying niche radius R on BBOB F3 data.



(b) GA_SGD_dynamic varying number of niches N on BBOB F3 data.

Figure 13: the loss curves of the NN learning process using GA_SGD, GA_SGD_sharing, GA_SGD_dynamic, and SGD for the (2,50,20,1) network on 5000 standardized data samples of BBOB F3. The GA_SGD, GA_SGD_sharing (R=*), and GA_SGD_dynamic (R=* N=*) are described in section 4.1 with default parameter settings in Table 2. The learning rate of all SGD steps is 0.00001, and their batch size is 64. The curves represent the average loss over 5 independent training runs, and the shaded areas represent the standard deviation confidence intervals across runs.

Figure 13a illustrates that when the niche radius R for fitness sharing is between 1 and 50, it enhances the optimization performance of GA_SGD. This results in a later convergence with a lower training loss, approaching that of SGD, while SGD converges to a slightly lower loss than other algorithms much more quickly. We select $R = 5$ as the optimal setting for our dynamic fitness sharing experiments, as shown in Figure 13b. The loss curves of GA_SGD_dynamic with different numbers of niches are almost identical to the GA_SGD_sharing curve, showing no significant variation.

On BBOB F3 data, our proposed algorithms achieve training losses comparable to those of SGD, motivating us to conduct further tests for performance comparison.

For the test, we first generate 100 test sets, each containing 5000 test samples, determined by a seed. For each test set, the function variables x_1 and x_2 are randomly generated within the interval (-5, 5), and the function value y is the corresponding BBOB function value for variables x_1 and x_2. The BBOB function is specified by the instance in the Coco experimenter, with the instance fixed at 1, ensuring the BBOB function does not change with different seeds. Since the training is performed using standardized data, the test data are also standardized before testing.

Each algorithm obtains 100 losses corresponding to the 100 test sets, and we compute the statistics of the 100 losses as the algorithm's test results. During testing, two different methods are employed to obtain the loss of an algorithm on a test set.

1. The first method collects all the final network individuals obtained from the five repeated runs of the algorithm during training into a single population, such as a population of 1000 ($= 5 * \mu$) networks from 5 independent runs of GA_SGD. The algorithm's loss on this test set is the lowest in this population, calculated by testing each network from this population with this test set and selecting the lowest test loss. This method reduces the impact of overfitting on the test results.
2. The second method considers the five networks, with the best training loss in the five final populations from the five repeated runs of the algorithm during training, respectively, as a population. The algorithm's loss on this test set is the average of this population's test losses, calculated by testing the five networks with this test set and averaging their five test losses.

The test results for the BBOB F3 using both methods are presented in Table 3. Regardless of the test method, fitness sharing and dynamic fitness sharing enhance the performance of GA_SGD, consistent with the training loss relationships shown in Figure 13, indicating that GA_SGD, GA_SGD_sharing, and GA_SGD_dynamic do not suffer from overfitting in this budget.

For the first test method, GA_SGD_sharing outperforms SGD, leveraging the advantage of a large population brought by GA. However, SGD significantly outperforms other algorithms for the second test method, which aligns with the training loss curves. When SGD has more than 50000 epochs, it presents noticeable overfitting with poorer test results shown in the statistics of the second test method, even worse compared to GA_SGD_sharing and GA_SGD_dynamic.

Table 3: the statistics of 100 test losses on 100 test sets, each having 5000 standardized BBOB F3 data samples, using the two test methods for GA_SGD, GA_SGD_sharing ($R = 5$), GA_SGD_dynamic ($R = 5$ $N = 20$), and SGD.

Test Method	**Algorithm**	**mean_loss**	**best_loss**	**worst_loss**	**std_deviation**	**std_error**	**1Q**	**2Q**	**3Q**
1	GA_SGD	107.65	100.31	138.12	6.92	0.69	103.37	105.60	108.82
	GA_SGD_sharing	**101.25**	91.20	**125.57**	6.74	0.67	**96.06**	100.55	**103.80**
	GA_SGD_dynamic	102.80	95.90	127.60	**6.44**	**0.64**	98.73	100.67	104.12
	SGD (50000 epochs)	105.57	**90.22**	155.15	13.61	1.36	97.09	101.51	107.03
	SGD (59689 epochs)	102.68	92.18	137.77	8.60	0.86	97.06	**100.37**	104.48
2	GA_SGD	117.69	110.21	124.27	5.48	2.24	115.09	116.40	122.27
	GA_SGD_sharing	115.43	108.86	122.95	6.14	2.51	110.24	114.90	120.44
	GA_SGD_dynamic	115.20	106.15	121.58	5.51	2.25	112.70	116.43	118.45
	SGD (50000 epochs)	**106.29**	101.87	**114.54**	**4.32**	**1.76**	**104.60**	**105.25**	**106.17**
	SGD (59689 epochs)	116.27	**98.36**	143.17	15.79	6.44	106.81	113.44	121.81

Note the two SGD variations in the table: one with 50000 epochs, calculated based on the network traversal counting method described in section 3.3.4, ensuring the same number of network traversals as GA_SGD and other algorithms. Unfortunately, this counting method has a flaw. As in SGD, after the forward propagation of 5000 data samples, the backward propagation does not necessarily result in 5000 backward network traversals. Instead, with a batch size of 64, there is one gradient update per 64 data samples, which is counted as one backward network traversal by the new counting method. In this new method, 59689 SGD epochs match the budget of other algorithms. However, the batch size-based method still has an issue preferring SGD, as one gradient update on a batch is significantly more time-consuming than a single network traversal. We can regard these two counting methods as two extremes of a fair comparison.

4.2.3 GA_SGD_sharing on BBOB F1, F3, F7, F13, F16, F22

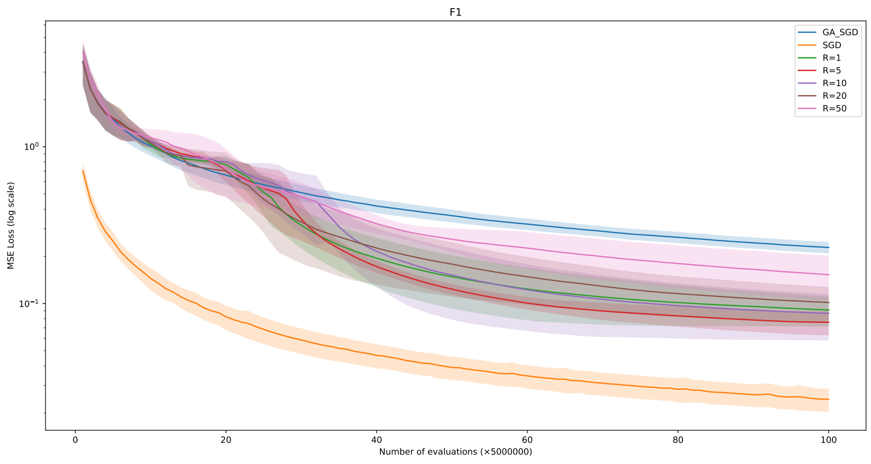
Based on the current results, GA_SGD_dynamic does not show a significant improvement over GA_SGD_sharing and requires the consideration of an additional parameter, the number of niches N. Therefore, we choose the simpler GA_SGD_sharing as the better algorithm for further experiments on other BBOB functions, with results shown in Figure 14.

Table 4: the statistics of 100 test losses on 100 test sets, each having 5000 standardized BBOB F7 data samples, using the two test methods for GA_SGD, GA_SGD_sharing ($R = 20$), and SGD.

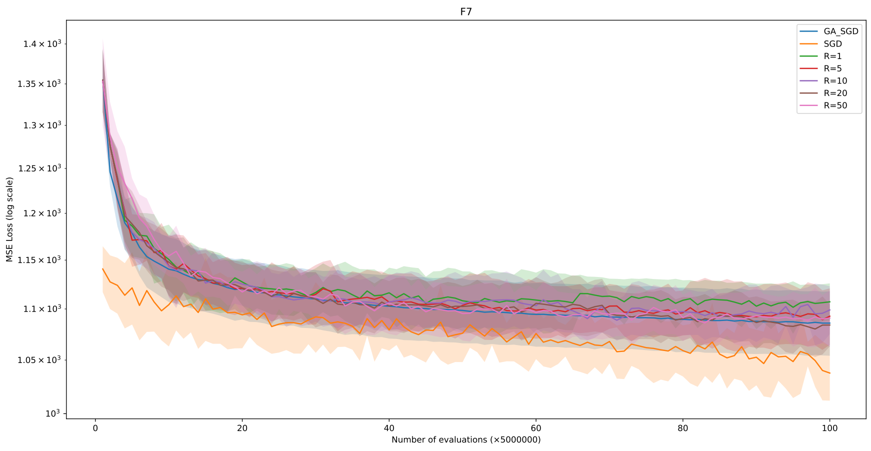
Test Method	Algorithm	mean_loss	best_loss	worst_loss	std_deviation	std_error	1Q	2Q	3Q
1	GA_SGD	1124.56	1034.61	1239.90	43.87	4.39	**1093.36**	1124.79	1149.69
	GA_SGD_sharing	**1121.21**	1052.76	**1213.65**	**37.58**	**3.76**	1094.63	1122.58	**1143.09**
	SGD (50000 epochs)	1134.58	1042.70	1400.15	59.38	5.94	1100.34	**1120.46**	1162.49
	SGD (59689 epochs)	1157.40	**1008.84**	1314.82	69.56	6.96	1113.94	1155.24	1209.67
2	GA_SGD	1251.25	**1122.72**	1464.24	130.82	53.41	**1151.91**	1222.23	1316.27
	GA_SGD_sharing	1321.89	1155.17	1695.08	206.52	84.31	1172.17	1259.91	1378.76
	SGD (50000 epochs)	**1222.65**	1144.21	**1437.44**	**110.68**	**45.19**	1154.20	**1189.49**	**1221.98**
	SGD (59689 epochs)	1363.72	1162.34	1655.48	210.03	85.74	1194.57	1304.00	1524.80

Table 5: the statistics of 100 test losses on 100 test sets, each having 5000 standardized BBOB F13 data samples, using the two test methods for GA_SGD, GA_SGD_sharing ($R = 5$), and SGD.

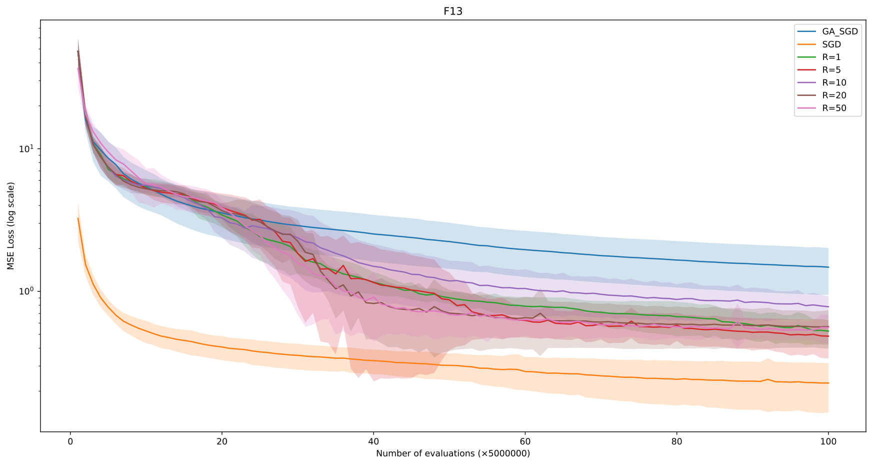
Test Method	Algorithm	mean_loss	best_loss	worst_loss	std_deviation	std_error	1Q	2Q	3Q
1	GA_SGD	52.93	3.67	500.25	68.73	6.87	17.59	29.75	68.27
	GA_SGD_sharing	**47.63**	2.35	**480.93**	**65.73**	**6.57**	**15.34**	**26.84**	**60.80**
	SGD (50000 epochs)	53.65	2.22	503.94	69.55	6.95	17.64	**30.46**	69.51
	SGD (59689 epochs)	53.91	**2.16**	506.32	69.97	7.00	18.09	29.99	69.47
2	GA_SGD	169.91	30.32	577.07	206.64	84.36	46.42	102.75	159.86
	GA_SGD_sharing	169.90	30.71	574.74	205.53	83.91	46.58	103.95	159.91
	SGD (50000 epochs)	**168.59**	30.81	**570.32**	**203.99**	**83.28**	45.83	103.04	**158.94**
	SGD (59689 epochs)	169.58	**30.00**	577.81	207.23	84.60	**45.30**	**102.16**	159.82


F1
GA_SGD
SGD
R=1
R=5
R=10
R=20
R=50
MSE Loss (log scale)
Number of evaluations (×5000000)

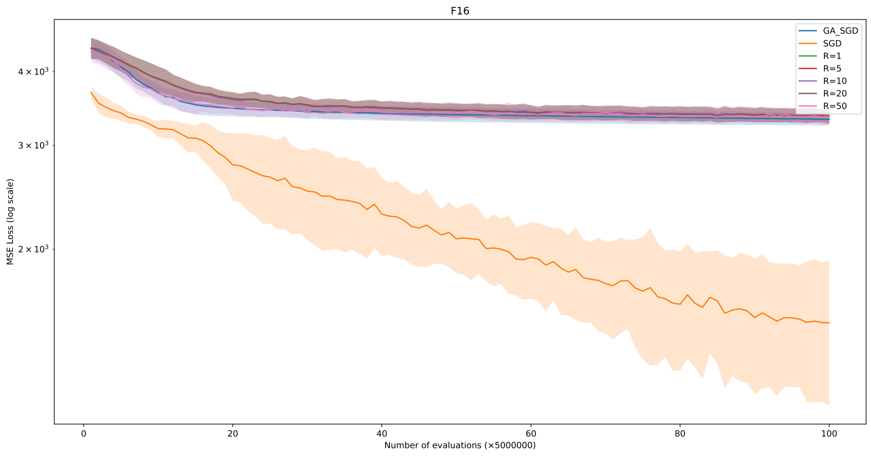
(a) F1


F7
GA_SGD
SGD
R=1
R=5
R=10
R=20
R=50
MSE Loss (log scale)
Number of evaluations (×5000000)

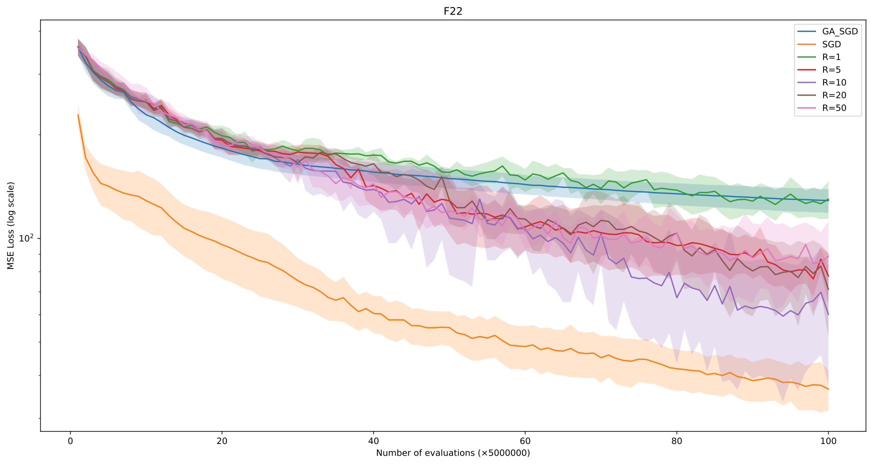
(b) F7


F13
GA_SGD
SGD
R=1
R=5
R=10
R=20
R=50
MSE Loss (log scale)
Number of evaluations (×5000000)

(c) F13



(d) F16



(e) F22

Figure 14: the loss curves of the NN learning process using GA_SGD, GA_SGD_sharing, and SGD for the (2,50,20,1) network on 5000 standardized data samples of BBOB function 1, 7, 13, 16, 22. The GA_SGD and GA_SGD_sharing (R=*) are described in section 4.1 with default parameter settings in Table 2. The learning rate of all SGD steps is 0.00001, and their batch size is 64. The curves represent the average loss over 5 independent training runs, and the shaded areas represent the standard deviation confidence intervals across runs.

Figure 13 and 14 indicate that fitness sharing significantly enhances GA_SGD in most of our function approximation problems, including BBOB F1, F3, F13, and F22. Fitness sharing aids the GA_SGD optimization of NNs on F3 and F13, making their performance in the final stages of training similar to that of SGD. However, fitness sharing does not have a notable impact on F7 and F16.

The function approximation of F7 by the NN may be sufficiently straightforward that GA_SGD can perform well without the need for fitness sharing. On the other hand, for F16, the limited capability of GA_SGD itself constrains the effectiveness of fitness sharing. The finite epochs of SGD within GA_SGD might not be sufficient for F16, causing both GA_SGD and GA_SGD_sharing to converge prematurely while SGD continues to improve. Moreover, experimented on F1 and F22, neither the faster-learning SGD nor the slower GA_SGD_sharing converge by the end of training, resulting in GA_SGD_sharing still trailing behind SGD in performance.

According to the findings, we opt to continue testing the algorithms using the two test methods on F7 and F13, where the training loss curve of GA_SGD_sharing is close to that of SGD. The test results are presented in Tables 4 and 5.

Similar to the test results on BBOB F3, the statistics of GA_SGD_sharing on F7 and F13 present superior performance to those of SGD when using the first test method. However, GA_SGD and GA_SGD_sharing underperform compared to SGD when using the second test method, although GA_SGD_sharing outperforms the overfitting SGD with more epochs on BBOB F7, as on F3. With the second method to test on F13, the performance of GA_SGD and GA_SGD_sharing is only marginally worse than that of SGD.

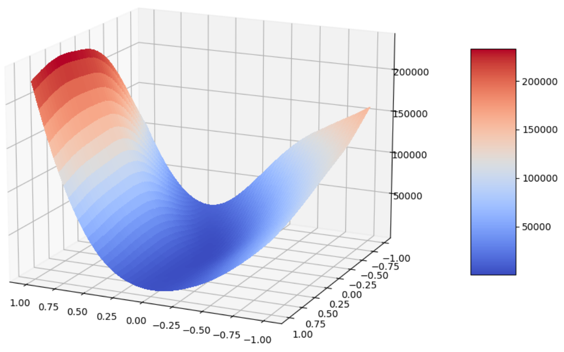
4.3 Discussion

After introducing SGD into GA, the optimization efficiency of GA_SGD significantly improves compared to the original GAs. However, it remains lower than the efficiency of SGD alone while generally surpassing that of ADAM under the same experimental settings, potentially because these settings are not ideal for ADAM. Notably, the parallel optimization feature of GA_SGD enhances the efficiency of each parallel SGD epoch, making it suitable for advanced computing environments such as parallel computing.

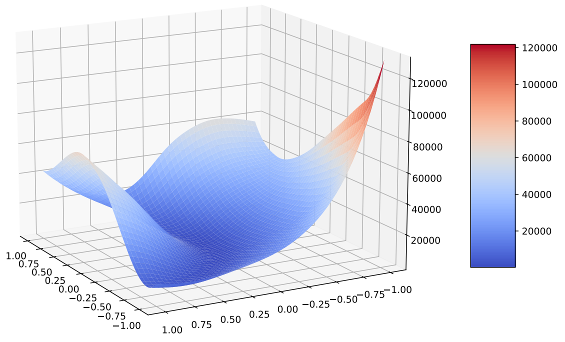
Incorporating fitness sharing into GA_SGD results in GA_SGD_sharing, which shows a marked improvement over GA_SGD in most of our NN function approximation problems. Despite this improvement, it still struggles to surpass the efficiency of SGD and shows no significant difference in performance on the overly simple F7 and the excessively difficult F16. Specifically, GA_SGD_sharing's performance on F3, F7, and F13 is very close to SGD in the end, and it still has not converged on F1 and F22 (SGD either), while failing to help GA_SGD breakthrough on F16.

Furthermore, consistent with the results from section 3.3.4, the GA_SGD_dynamic with dynamic fitness sharing does not outperform GA_SGD_sharing with fitness sharing on F3, indicating that fitness sharing adequately addresses difficulties with no need for its dynamic version after data standardization. The effectiveness of fitness sharing in enhancing GA_SGD performance is insensitive to niche radius R and number of niches N, with R settings between 1 and 50 being generally effective.

The lower efficiency of GA_SGD compared to SGD in training can be attributed to the inherent nature of GA, which introduces an entire population of networks. This parallel optimization mechanism of GA_SGD does not enhance the efficiency of each epoch or network traversal above SGD while allowing more budget for crossover and selection among different optimization paths brought by the randomness of SGD optimization in the population of networks. Thus, GA_SGD improves the efficiency of each parallel SGD epoch. In experiments, this manifests as GA_SGD running 200 SGD epochs in parallel for 200 individuals in the population, while SGD can run 50000 epochs or more with the same number of network traversals.



(a) the 1st partial landscape.



(b) the 2nd partial landscape.

Figure 15: the partial landscapes of NN parameter optimization for the BBOB F3 approximation, using the implementation and methodology from [2, 28]. The vertical axis represents the MSE loss and the other two axes refer to the 2d projection of network parameters.

Figure 15 visualizes the landscapes of NN parameter optimization for the BBOB F3 approximation, only representing a partial landscape view on the entire search space.

In these landscapes, a valley region is discernible. Typically, algorithms initiate optimization from a random peak, descending along the gradient to reach a point within the valley region. From there, they continue a gradual descent to find the global optimum. Although frequently hindering the algorithms, the local optima on the slopes are relatively easy for the algorithms to overcome. In contrast, the local optima in the valley are more challenging to surmount. Therefore, it is advantageous for the optimization algorithm to reach areas of relatively lower losses within the valley. This target cannot be reached simply by random restarts of SGD. Instead, employing EAs and SGD to balance exploration and exploitation with a population might be more beneficial, which is also why we are trying GA_SGD.

During SGD optimization, the loss quickly converges to a point in the valley and slowly decreases for a lower region. On the other hand, GA_SGD_sharing first descends the hill more slowly than SGD and reaches the valley later, vertically close but still higher to SGD, with fewer SGD epochs and an equivalent number of network traversals. Besides F3, the optimization of SGD and GA_SGD_sharing on F7 and F13 have similar processes, while the algorithms perhaps have not reached the valley in F1 and F22 experiments.

Testing on F3, F7, and F13, where GA_SGD_sharing and SGD show comparable training performance, demonstrate that GA_SGD_sharing yields the best test results when the best-performing individuals in the population for testing are selected for testing, presenting the advantage of a large dispersed population. Conversely, when the best-performing individuals in training are selected for testing, SGD provides the best test results, aligning with its training performance.

In the function approximation of F16, GA_SGD_sharing gets stuck midway. This failure suggests that when the landscape on the search space requires many SGD epochs to find the correct direction, the limited number of SGD epochs in GA_SGD restricts its ability to break through traps and reach the same level as SGD.

Overall, the efficiency of GA_SGD is inherently lower than that of SGD but higher than ADAM under suboptimal settings. Using niching methods such as fitness sharing and dynamic fitness sharing, originally intended for MMO, significantly enhances the performance of GA_SGD, bringing it closer to SGD. Furthermore, in the valley region of the landscape, the dispersed network population of GA_SGD_sharing likely contains better-performing networks for testing than the single network of SGD, although in a lower landscape region, after the same time budget at the late optimization stages.

5 Conclusion

This thesis aims to apply multimodal optimization (MMO) metaheuristics to classical evolutionary algorithms (EAs) and explore their performance in optimizing neural network (NN) parameters.

As a result, we implement simple evolution-based algorithms such as ES, GA, and GA_SGD, apply fitness sharing and dynamic fitness sharing from niching methods to them, and conduct a series of experiments. These evolution-based algorithms, along with gradient-based methods, are used to train a simple neural network architecture (2, 50, 20, 1) to approximate six BBOB functions of appropriate difficulty.

Our experimental results consistently demonstrate the efficiency of gradient-based methods, which achieve lower loss with fewer network traversals compared to the larger computational demands of population-based evolutionary algorithms. The OnePlusOne algorithm from the Nevergrad black-box optimization suite, utilizing its population of size one, exhibits the highest optimization efficiency among the Nevergrad algorithms under default settings, while slightly inferior to SGD.

In experiments with basic EAs, ES performs significantly worse than both GA and SGD. Fitness sharing and dynamic fitness sharing, designed for GA, improve GA's optimization performance but do not affect ES. Our GA, with a large population, outperforms basic SGD over the same number of iterations but remains inferior to SGD when compared based on network traversals. Additionally, a condition for sharing to improve GA's optimization performance is sufficient crossover in GA, such as uniform crossover on each network parameter.

Based on our experiments with basic EAs, we introduce gradient-based methods into the basic GA, termed GA_SGD. GA_SGD shows a marked improvement in optimization efficiency over GAs, approaching the performance of SGD and outperforming non-optimally configured ADAM. Incorporating fitness sharing into GA_SGD, resulting in GA_SGD_sharing, further enhances optimization performance, bringing it closer to SGD, though still not surpassing it. We also observe that dynamic fitness sharing does not improve performance over fitness sharing because fitness sharing is already effective in tackling most problems.

Following this, we select three of the six BBOB functions for further testing, where GA_SGD_sharing's performance is close to SGD. Using standard testing methods with the best network from training, SGD slightly outperforms GA_SGD_sharing, consistent with the training results. However, using a special testing method that selects the network with the best test performance from the population, GA_SGD_sharing outperforms SGD, indicating that its population likely ($> 50\%$) contains networks with lower test losses than SGD's single network.

In summary, our research demonstrates that distance-based niching methods such as fitness sharing and dynamic fitness sharing significantly enhance the performance of evolution-based GA and hybrid GA_SGD in NN parameter optimization. We also explore the hyperparameters, implementation, and relationship with crossover for these niching methods. Although GA_SGD_sharing does not surpass SGD in overall performance, its superior results in specific tests validate the advantages of a niching-enabled network population.

Based on our study, we propose the following directions for future research:

- Incorporating distributed and parallel computing techniques into GA_SGD_sharing to alleviate its inherent redundant computations while leveraging the advantages of population-based parallel evolution.
- Integrating fitness sharing to enhance the training frameworks proposed in related works [13, 38, 51], which are similar to our GA_SGD_sharing.
- Introducing other niching methods or MMO metaheuristics into evolutionary and hybrid evolutionary-gradient optimization algorithms, such as clearing and crowding. (Clustering has poor results in our trials.)
- Further refining the GA_SGD_sharing algorithm with additional features, such as mutation and parameter tuning.
- Testing the niching methods on larger problems with more complicated network architectures such as CNN.

References

[1] Buthainah Al-kazemi and Chilukuri K Mohan. Training feedforward neural networks using multi-phase particle swarm optimization. In *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP'02.*, volume 5, pages 2615–2619. IEEE, 2002.

[2] artur deluca. Visualizing the loss landscape of neural networks. `https://github.com/artur-deluca/landscapeviz`. Accessed: 2024-04-04.

[3] Thomas Back. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *Proceedings of the first IEEE conference on evolutionary computation. IEEE World Congress on Computational Intelligence*, pages 57–62. IEEE, 1994.

[4] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[5] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1):1–23, 1993.

[6] Thomas HW Bäck, Anna V Kononova, Bas van Stein, Hao Wang, Kirill A Antonov, Roman T Kalkreuth, Jacob de Nobel, Diederick Vermetten, Roy de Winter, and Furong Ye. Evolutionary algorithms for parameter optimization—thirty years later. *Evolutionary Computation*, 31(2):81–122, 2023.

[7] Thomas Bartz-Beielstein, Jürgen Branke, Jörn Mehnen, and Olaf Mersmann. Evolutionary algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4(3):178–195, 2014.

[8] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, pages 177–186. Springer, 2010.

[9] Rohitash Chandra. Competition and collaboration in cooperative coevolution of elman recurrent neural networks for time-series prediction. *IEEE transactions on neural networks and learning systems*, 26(12):3123–3136, 2015.

[10] Rohitash Chandra and Mengjie Zhang. Cooperative coevolution of elman recurrent neural networks for chaotic time series prediction. *Neurocomputing*, 86:116–123, 2012.

[11] Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. *Advances in neural information processing systems*, 31, 2018.

[12] Xiaodong Cui, Wei Zhang, Zoltán Tüske, and Michael Picheny. Evolutionary stochastic gradient descent for optimization of deep neural networks. *Advances in neural information processing systems*, 31, 2018.

[13] Omid E David and Iddo Greental. Genetic algorithms for evolving deep neural networks. In *Proceedings of the companion publication of the 2014 annual conference on genetic and evolutionary computation*, pages 1451–1452, 2014.

[14] Saber M Elsayed, Ruhul A Sarker, and Daryl L Essam. Multi-operator based evolutionary algorithms for solving constrained optimization problems. *Computers & operations research*, 38(12):1877–1896, 2011.

[15] Steffen Finck, Nikolaus Hansen, Raymond Ros, and Anne Auger. Real-parameter black-box optimization benchmarking 2009: Presentation of the noiseless functions. Technical report, Citeseer, 2010.

[16] Edgar Galván and Peter Mooney. Neuroevolution in deep neural networks: Current trends and future challenges. *IEEE Transactions on Artificial Intelligence*, 2(6):476–493, 2021.

[17] Christian Goerick and Tobias Rodemann. Evolution strategies: an alternative to gradient-based learning. In *Proceedings of the International Conference on Engineering Applications of Neural Networks*, volume 1, pages 479–482, 1996.

[18] David E Goldberg, Jon Richardson, et al. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, volume 4149, pages 414–425. Cambridge, MA, 1987.

[19] DE Goldberg. Genetic algorithms in search, optimization and machine learning. addison-wesley longman publishing co., inc. 1989.

[20] Nikolaus Hansen, Dimo Brockhoff, Olaf Mersmann, Tea Tusar, Dejan Tusar, Ouassim Ait ElHara, Phillipe R Sampaio, Asma Atamna, Konstantinos Varelas, Umut Batu, et al. Comparing continuous optimizers: numbbo/coco on github, march 2019. *URL https://doi. org/10*, 5281, 2019.

[21] John H Holand. Adaptation in natural and artificial systems. *Ann Arbor: The University of Michigan Press*, page 32, 1975.

[22] Earnest Paul Ijjina and Krishna Mohan Chalavadi. Human action recognition using genetic algorithms and convolutional neural networks. *Pattern recognition*, 59:199–212, 2016.

[23] Dervis Karaboga, Bahriye Akay, and Celal Ozturk. Artificial bee colony (abc) optimization algorithm for training feed-forward neural networks. In *Modeling Decisions for Artificial Intelligence: 4th International Conference, MDAI 2007, Kitakyushu, Japan, August 16-18, 2007. Proceedings 4*, pages 318–329. Springer, 2007.

[24] Asha Gowda Karegowda, AS Manjunath, and MA Jayaram. Application of genetic algorithm optimized neural network connection weights for medical diagnosis of pima indians diabetes. *International Journal on Soft Computing*, 2(2):15–23, 2011.

[25] Shauharda Khadka, Jen Jen Chung, and Kagan Tumer. Neuroevolution of a modular memory-augmented neural network for deep memory problems. *Evolutionary computation*, 27(4):639–664, 2019.

[26] Shauharda Khadka and Kagan Tumer. Evolution-guided policy gradient in reinforcement learning. *Advances in Neural Information Processing Systems*, 31, 2018.

[27] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[28] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems*, 31, 2018.

[29] Nan Li, Lianbo Ma, Guo Yu, Bing Xue, Mengjie Zhang, and Yaochu Jin. Survey on evolutionary deep learning: Principles, algorithms, applications, and open issues. *ACM Computing Surveys*, 56(2):1–34, 2023.

[30] Youru Li, Zhenfeng Zhu, Deqiang Kong, Hua Han, and Yao Zhao. Ea-lstm: Evolutionary attention-based lstm for time series prediction. *Knowledge-Based Systems*, 181:104785, 2019.

[31] Samir W Mahfoud. *Niching methods for genetic algorithms.* University of Illinois at Urbana-Champaign, 1995.

[32] Martin Mandischer. A comparison of evolution strategies and backpropagation for neural network training. *Neurocomputing*, 42(1-4):87–117, 2002.

[33] Florian Meier, Asier Mujika, Marcelo Matheus Gauy, and Angelika Steger. Improving gradient estimation in evolutionary strategies with past descent directions. *arXiv preprint arXiv:1910.05268*, 2019.

[34] Brad L Miller and Michael J Shaw. Genetic algorithms with dynamic niche sharing for multimodal function optimization. In *Proceedings of IEEE international conference on evolutionary computation*, pages 786–791. IEEE, 1996.

[35] David J Montana, Lawrence Davis, et al. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pages 762–767, 1989.

[36] Gregory Morse and Kenneth O Stanley. Simple evolutionary optimization can rival stochastic gradient descent in neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 477–484, 2016.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[38] Krzysztof Pawełczyk, Michal Kawulok, and Jakub Nalepa. Genetically-trained deep neural networks. In *Proceedings of the genetic and evolutionary computation conference companion*, pages 63–64, 2018.

[39] 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*, 12:2825–2830, 2011.

[40] Yiming Peng, Gang Chen, Harman Singh, and Mengjie Zhang. Neat for large-scale reinforcement learning through evolutionary feature learning and policy gradient

search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 490–497, 2018.

[41] Vincent W Porto and David B Fogel. Alternative neural network training methods [active sonar processing]. *IEEE Expert*, 10(3):16–22, 1995.

[42] Mike Preuss. *Multimodal optimization by means of evolutionary algorithms.* Springer, 2015.

[43] J. Rapin and O. Teytaud. Nevergrad - A gradient-free optimization platform. `https://GitHub.com/FacebookResearch/Nevergrad`, 2018.

[44] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[45] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation, parallel distributed processing, explorations in the microstructure of cognition, ed. de rumelhart and j. mcclelland. vol. 1. 1986. *Biometrika*, 71:599–607, 1986.

[46] Ofer Michael Shir. *Niching in derandomized evolution strategies and its applications in quantum control.* Leiden University, 2008.

[47] Krzysztof Socha and Christian Blum. An ant colony optimization algorithm for continuous optimization: application to feed-forward neural network training. *Neural computing and applications*, 16:235–247, 2007.

[48] Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1(1):24–35, 2019.

[49] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[50] Min Wu, Wanjuan Su, Luefeng Chen, Zhentao Liu, Weihua Cao, and Kaoru Hirota. Weight-adapted convolution neural network for facial expression recognition in human–robot interaction. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(3):1473–1484, 2019.

[51] Shangshang Yang, Ye Tian, Cheng He, Xingyi Zhang, Kay Chen Tan, and Yaochu Jin. A gradient-guided evolutionary approach to training deep neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 33(9):4861–4875, 2021.

[52] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[53] Jiawei Zhang and Fisher B Gouza. Gadam: genetic-evolutionary adam for deep neural network optimization. *arXiv preprint arXiv:1805.07500*, 2018.