SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Consensus-Based Optimization of Sampled Neural Networks

Melek Walha

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Consensus-Based Optimization of Sampled Neural Networks

Konsensbasierte Optimierung von gesampelten neuronalen Netzen

Author:	Melek Walha
Supervisor:	Prof. Dr. Felix Dietrich
Submission Date:	15.09.2024

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2024 Melek Walha

Acknowledgments

First and foremost, I dedicate this work to the memory of my beloved father, who always believed in me. Though he is no longer with us, his spirit continues to guide me, his words continue to inspire me and I owe any success to his enduring influence.

I would like to extend my deepest gratitude to my advisor, Prof. Dr. Felix Dietrich, for his invaluable advice and support throughout the course of this thesis. His insights and expertise have been instrumental in shaping my research and have greatly enriched this work.

I am also profoundly grateful to my family -my mother, my brother and my aunt- for their endless encouragement and love. Their constant support has been my foundation and I could not have reached this point without them.

Finally, I see this thesis as just the beginning of my journey. I look forward to the challenges and opportunities that lie ahead and I am hopeful that this is only the first of many accomplishments to come.

Abstract

This thesis presents a novel approach to neural network training, integrating the Sample Where It Matters (SWIM) algorithm with Consensus-Based Optimization (CBO) to create a derivative-free training methodology. SWIM strategically initializes network parameters by sampling regions of the data space that are most likely to contain optimal solutions, which effectively reduces the number of training epochs needed. The CBO method then refines these parameters by guiding multiple candidate solutions towards consensus through a combination of deterministic and stochastic updates. This combined SWIM-CBO approach is rigorously evaluated across various tasks, including regression and classification, using a simple sine function, a more oscillatory and complicated function and a simplified MNIST dataset. The results demonstrate that the combined method significantly improves convergence times and achieves lower final losses compared to the CBO-only approach. Moreover, the analysis of hyperparameter sensitivity highlights the importance of careful tuning to achieve optimal performance, offering insights into the potential of derivative-free optimization techniques in neural network training.

Contents

1 Introduction

Machine learning has emerged as one of the most influential technologies of the modern era, with applications spanning across various domains such as healthcare, finance, transportation and beyond. Its capability to learn from data and make informed decisions without explicit programming has revolutionized tasks like image recognition, natural language processing and predictive analytics. Within the broad spectrum of machine learning techniques, neural networks have attracted significant attention due to their ability to model complex, non-linear relationships, making them particularly powerful in solving challenging problems [1].

Neural networks, inspired by the neural structure of the human brain [2], consist of layers of interconnected artificial neurons that process input data to produce an output. These networks have demonstrated remarkable effectiveness in a wide range of applications, from simple regression tasks to more sophisticated classification problems. However, training neural networks, especially those with deep architectures, poses substantial challenges. The traditional approach to training neural networks relies on gradient-based optimization methods, such as stochastic gradient descent (SGD) or more advanced variants like Adam [3]. These methods iteratively adjust the network's parameters -namely weights and biases- to minimize a loss function, which quantifies the discrepancy between the predicted outputs and the actual data [4].

While gradient-based methods are effective, they come with significant drawbacks. These methods are highly sensitive to the initial parameter settings and often struggle with local minima in the optimization landscape, particularly in the context of deep, non-convex networks. Moreover, the computation of gradients can be computationally expensive, especially for large-scale neural networks, making these methods resource-intensive. To address these challenges, alternative approaches, such as derivative-free optimization methods, have been explored, offering a promising solution to the limitations of gradient-based training.

In this thesis, we introduce a novel approach to neural network training that combines two derivative-free methods: the Sample Where It Matters (SWIM) algorithm [5, 6] and Consensus-Based Optimization (CBO) [7]. The SWIM algorithm provides an informed initialization by sampling network parameters in regions of the data space that are likely to contain optimal solutions, thereby positioning the network closer to the target. Following this initialization, the CBO method refines these parameters by guiding multiple candidate solutions (particles) towards a consensus through a combination of deterministic and stochastic updates.

Our research primarily focuses on three training strategies within the combined SWIM-CBO framework. The first strategy, referred to as "all", involves initializing the parameters of all layers using SWIM and then training them using the CBO method. The second strategy, "lin", trains only the parameters of the final linear layer using CBO, under the assumption that SWIM has effectively optimized the earlier layer. The third strategy, "rand-lin", involves a combination where the dense layer is initialized using SWIM while the linear layer is initialized randomly and subsequently trained with CBO.

To thoroughly evaluate the proposed approach, we apply it to a diverse set of problems, including regression and classification tasks. These tasks are represented by a simple Sine function, a more oscillatory and complicated function and a simplified MNIST dataset. These experiments were chosen to test the robustness and adaptability of the SWIM-CBO approach across different problem types and complexities.

The results from these experiments demonstrated that the combined SWIM-CBO method significantly improved convergence times and achieved lower final losses compared to the CBO-only approach. This indicates that the SWIM-CBO approach not only accelerates training but also enhances the overall accuracy of the models, highlighting its potential for solving complex optimization problems in neural network training.

This thesis is organized in two main parts: The first part provides a comprehensive overview of the current state-of-the-art in neural network training methodologies, including in-depth explanations of sampled networks, SWIM and CBO as one of particle optimization methods. The second part delves into the proposed combined SWIM-CBO approach, offering insights into the underlying motivations and problem definition, implementation details and then presents the results of our computational experiments, highlighting the performance of our approach.

2 State of the Art

2.1 SWIM and sampled networks

Training a neural network has always been a challenging task, not only is it time consuming but also computationally expensive. The training process generally uses iterative, gradient-based methods such as Adam and the resulting parameters (weights and biases) are hard to interpret, which is why a neural network is often referred to as a black box [8]. This opacity is a significant drawback, especially in critical applications where interpretability and transparency are crucial. In this section, we will introduce the concept of sampled networks and the SWIM algorithm, which offer an alternative approach by constructing the network parameters directly from data point pairs, thereby eliminating the need for iterative optimization.

2.1.1 Sampled Networks

Sampled networks represent an innovative approach to constructing neural networks by linking the model parameters directly to specific data points from the input space. Unlike traditional neural networks, which rely on iterative optimization methods to adjust weights and biases, sampled networks determine each pair of weight and bias of all its hidden layers by two points from the input space [5]. This approach ensures that the network is inherently tied to the given dataset, providing a more intuitive and transparent model structure.

This method offers several benefits [5], from which we highlight the following:

- **Efficiency and Accuracy:** By avoiding the need for extensive iterative optimization, sampled networks can be constructed more rapidly. This data-driven method also provides a more precise and width-efficient approximation compared to traditional sampling methods that do not consider the dataset structure.
- **Interpretability:** With weights and biases directly derived from data points, sampled networks offer a clearer and more understandable connection between the model's parameters and the training data. This transparency aids in elucidating the model's decision-making processes and overall functionality.

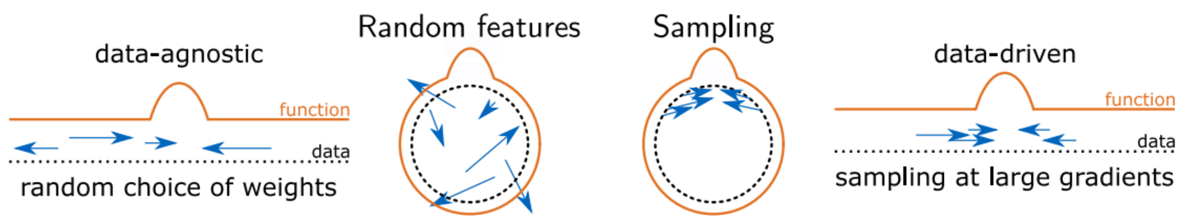


Figure 2.1: Random feature models choose weights in a data-agnostic way, compared to sampling them where it matters: at large gradients. The arrows illustrate where the network weights are placed. Figure taken from [5].

Definition:
In a sampled network, each neuron's weight vector and bias are determined by two distinct data points from the input space. Specifically, the weight is calculated as the normalized difference between these two points, divided by the squared distance between them. The bias, on the other hand, is computed as the inner product of this weight vector with one of the data points. This systematic approach ensures that the network's structure is directly linked to the training data, forming the foundation for sampled networks.

Formally, a neural network Φ with L hidden layers and an input space $X \subseteq \mathbb{R}^D$ is defined as a sampled network if, for each layer l from 1 to L and each neuron i from 1 to N_l, the weights and biases are determined by pairs of data points $(x_{0,i}^{(1)}, x_{0,i}^{(2)})$ sampled from $X \times X$. The weight and bias are given by

$$w_{l,i} = s_1 \frac{x_{l-1,i}^{(2)} - x_{l-1,i}^{(1)}}{\|x_{l-1,i}^{(2)} - x_{l-1,i}^{(1)}\|^2}, \quad b_{l,i} = \left\langle w_{l,i}, x_{l-1,i}^{(1)} \right\rangle + s_2,$$

where s_1 and s_2 are scalar constants, $x_{l-1,i}^{(1)}$ and $x_{l-1,i}^{(2)}$ represent the transformed data points in the $l-1$ layer and $x_{l-1,i}^{(1)} \neq x_{l-1,i}^{(2)}$. For the output layer, the weights W_{L+1} and biases b_{L+1} are chosen to minimize a specific loss function $\mathcal{L}$.

The values of s_1 and s_2 depend on the activation function used. For the ReLU activation function, s_1 is set to 1 and s_2 to 0, mapping $x^{(1)}$ to zero and $x^{(2)}$ to one. In the case of the tanh activation function, s_1 is set to twice s_2, with s_2 being $\frac{\ln(3)}{2}$, which maps $x^{(1)}$ and $x^{(2)}$ to $-\frac{1}{2}$ and $\frac{1}{2}$ respectively, with their midpoint mapping to zero.

2.1.2 SWIM

2.1.2.1 Overview of the SWIM Algorithm

The "Sample Where It Matters" (SWIM) algorithm is a method proposed to create sampled networks efficiently. The algorithm focuses on sampling the input space in regions that are critical for learning. For each hidden layer l, a conditional probability distribution $P^{(l)}$ over pairs $(x^{(1)}, x^{(2)})$ from $X \times X$ is constructed. This distribution prioritizes pairs of points that are close in the representation space in the l-th layer but exhibit significant differences in their true output values.

The SWIM algorithm can be summarized in the following steps:

1. **Compute Probability Distribution:** For each hidden layer l, compute the probability distribution $P^{(l)}$ based on the representations of the input points in the previous layer and their true output values.

2. **Sample Data Pairs:** Using the computed probability distribution, sample pairs of data points $(x^{(1)}, x^{(2)})$.

3. **Compute Weights and Biases:** For each neuron in the hidden layer, compute the weight vector as the normalized difference between the sampled points and compute the bias as the inner product of the weight vector with one of the sampled points.

4. **Optimize Output Layer:** Once all hidden layers are processed, optimize the weights and biases of the output layer to minimize a loss function $\mathcal{L}$, typically the Mean Squared Error (MSE), between the actual output values and those predicted by the network. This is a linear least-squares optimization problem that has a closed form solution and can be solved quickly and efficiently.

Algorithm 1 A possible implementation of the SWIM Algorithm for an activation function ϕ and a loss function $\mathcal{L}$

```
**Input:** $X = \{x_i : x_i \in \mathbb{R}^D, i = 1, 2, \ldots, M\}$, $Y = \{y_i : y_i \in \mathbb{R}^{N_{L+1}}, i = 1, 2, \ldots, M\}$
**Output:** $\{W_l, b_l\}_{l=1}^{L+1}$
**Constant:** $L \in \mathbb{N}_{>0}$, $\{N_l \in \mathbb{N}_{>0}\}_{l=1}^{L+1}$ and $s_1, s_2 \in \mathbb{R}$;
$\Phi^{(0)}(x) = x$;
**for** $l = 1$ to $L$ **do**
    $P^{(l)} = \text{ComputeProbabilityDistribution}(\Phi^{(l-1)}, X, Y)$;
    $W_l \in \mathbb{R}^{N_{l-1}, N_l}$, $b_l \in \mathbb{R}^{N_l}$;
    **for** $i = 1$ to $N_l$ **do**
        Sample $(x^{(1)}, x^{(2)})$ from $X \times X$, with probability proportional to $P^{(l)}$;
        $x^{(1)}_{l-1,i}, x^{(2)}_{l-1,i} \leftarrow \Phi^{(l-1)}(x^{(1)}), \Phi^{(l-1)}(x^{(2)})$;
        $W_{i,:} = s_1 \frac{x^{(2)}_{l-1,i} - x^{(1)}_{l-1,i}}{\|x^{(2)}_{l-1,i} - x^{(1)}_{l-1,i}\|^2}$;
        $b_i = \langle W_{i,:}, x^{(1)}_{l-1,i} \rangle + s_2$;
    **end for**
    $\Phi^{(l)}(\cdot) \leftarrow \phi(\Phi^{(l-1)} W_l(\cdot) - b_l)$;
**end for**
$W_{L+1}, b_{L+1} \leftarrow \arg\min \mathcal{L}(\Phi^{(L)}(X) W_{L+1} - b_{L+1}, Y)$;
**return** $\{W_l, b_l\}_{l=1}^{L+1}$;
```

2.1.2.2 Implementation: Training a Sampled Network in Python

To illustrate how the SWIM algorithm can be used to train a neural network, we provide a practical example using Python. Below is a simple code snippet that demonstrates the process of defining and training a sampled network.

Listing 2.1: Training a sampled network using the SWIM algorithm in Python [6]

```python
from sklearn.pipeline import Pipeline
from swimnetworks import Dense, Linear

# Define the SWIM model
steps = [
    ("dense", Dense(layer_width=512, activation="tanh",
                     parameter_sampler="tanh",
                     random_seed=42)),
    ("linear", Linear(regularization_scale=1e-5))
]
model = Pipeline(steps)
# Train the model
model.fit(x_train, y_train)
# Predict with the trained model
pred = model.transform(x_test)
```

As shown in Listing 2.1, a Python implementation is provided to demonstrate the model's architecture and training process of the SWIM algorithm. The model consists of a `Dense` hidden layer followed by a linear output layer. Specifically, the dense layer uses the Tanh activation function and contains 512 neurons, defining the core architecture. A sample seed is used for sampling data points during training, ensuring reproducibility and consistent behavior across different training runs.

The model is then trained using the `fit` method on the training dataset (`x_train` and `y_train`), which computes the parameters based on the provided data.

Once training is complete, the model can be used to make predictions on new data, by applying the `transform` method on `x_test` for example for evaluating performance.

2.2 Particle Optimization Methods

Particle optimization methods have gained significant attention due to their effectiveness in addressing complex, high-dimensional and nonconvex optimization problems. These challenges often prove difficult for traditional optimization techniques, which may struggle with issues such as local minima or the computational demands of high-dimensional search spaces.

One of the key advantages of particle optimization methods is their decentralized and distributed nature, which enhances the robustness of the approach [9]. In such systems, the failure of individual agents typically has minimal impact on the overall performance, making the methods resilient to disruptions. Furthermore, these methods are inherently scalable, enabling them to handle large-scale optimization tasks efficiently. The parallelism inherent in particle-based algorithms allows for the effective utilization of computational resources, particularly in high-dimensional spaces where traditional methods may falter.

Additionally, particle optimization methods strike a balance between exploration and exploitation, which is crucial for thoroughly searching the solution space and refining promising regions. This balance helps to reduce the risk of becoming trapped in local minima, thereby increasing the likelihood of converging towards a global optimum. Another important aspect is the simplicity and ease of implementation of many particle optimization algorithms. These methods do not require gradient information or other complex mathematical constructs, making them accessible and practical for a wide range of applications.

Due to these attributes, particle optimization methods have proven to be powerful tools for solving diverse real-world optimization problems across various fields, including finance, engineering and robotics [10].

2.2.1 Key Principles of Particle Optimization

Particle optimization methods are inspired by self-organization and collective behavior in nature or human society [11]. This concept can be seen in the swarming of the birds, the schools of fish, bacterial growth, ant colonies foraging and many more. The key principles of a particle optimization method include:

- **Decentralization**: The control is distributed among all agents in the swarm, meaning that there is no central authority dictating the actions of individual agents.
- **Self-Organization**: Agents in the swarm interact locally with each other and with their environment, which leads to the emergence of global behavior from simple local rules.
- **Adaptation**: Agents are allowed to modify their behavior based on feedback from the environment and/or other agents.
- **Collaboration**: Agents work together to explore the solution space, share information and converge towards optimal solutions.

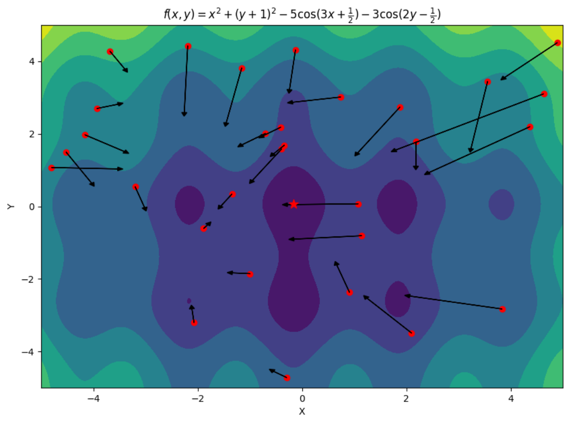


Figure 2.2: Depiction of particles (in red) and velocities (black arrows) according to particle swarm optimization algorithm. The minimum of the cost function is located in the center of the image (red star).

2.2.2 Consensus-Based Optimization (CBO)

Consensus-Based Optimization (CBO), being one of the particle optimization methods we discussed earlier, follows its guiding principles, such as decentralization, self-organization and collaboration, but it is of a much simpler nature and more amenable to theoretical analysis. CBO is a derivative-free, population-based optimization method designed to globally minimize nonconvex and non-smooth functions in high-dimensional spaces [12].

CBO uses N particles $V^1, ..., V^N$,which are independently initialized according to some law $\rho_0 \in \mathcal{P}(\mathbb{R}^d)$, to explore the domain and to form a global consensus about the minimizer v^* as time passes. The dynamics of each particle are governed by a combination of deterministic and stochastic components, which guide the particles towards a consensus point while allowing for sufficient exploration of the solution space. This consensus point is a weighted average of the particles' positions and represents the collective opinion of the particle swarm about the location of the global minimum.

2.2.2.1 Dynamics and Formulation

The CBO method is described by the following stochastic differential equation (SDE) for each particle i at time t

$$dV_i^t = -\lambda(V_i^t - v_\alpha(\rho_t^N))dt + \sigma D(V_i^t - v_\alpha(\rho_t^N))dB_i^t, \tag{2.1}$$

where V_i^t represents the position of particle i at time t.

The term $-\lambda(V_i^t - v_\alpha(\rho_t^N))$ in the SDE above is a drift term that promotes the convergence towards lower objective function values, meaning that λ controls the strength of this drift and influences the convergence speed of the particles. This ensures that particles are guided towards the consensus point, $v_\alpha(\rho_t^N)$, which is a weighted average of the particles' positions and is computed as

$$v_\alpha(\rho_t^N) := \int v \frac{\omega_\alpha(v)}{\|\omega_\alpha\|_{L_1(\rho_t^N)}} \, d\rho_t^N(v), \quad \text{with} \quad \omega_\alpha(v) := \exp(-\alpha\mathcal{E}(v)), \tag{2.2}$$

where $\alpha > 0$ determines the sharpness of the weighting and $\mathcal{E}(v)$ is the objective function.

In order to avoid the particles getting stuck in local minima, the diffusivity term $\sigma D(V_i^t - v_\alpha(\rho_t^N))$ is incorporated into the dynamics, which introduces randomness into the dynamics through independent standard Brownian motions $\left((B_t^i)_{t\geq 0}\right)_{i=1,\dots,N}$, allowing particles to explore the solution space and to explore the energy landscape of $\mathcal{E}$. The two commonly studied diffusion types are isotropic [12] and anisotropic [11, 13] diffusion with

$$D\left(V_t^i - v_\alpha(\hat{\rho}_t^N)\right) = \begin{cases} \|V_t^i - v_\alpha(\hat{\rho}_t^N)\|_2 \mathrm{Id}, & \text{for isotropic diffusion,} \\ \mathrm{diag}\left(V_t^i - v_\alpha(\hat{\rho}_t^N)\right), & \text{for anisotropic diffusion,} \end{cases}$$

where $\mathrm{Id} \in \mathbb{R}^{d\times d}$ is the identity matrix and diag(.) maps a vector onto a diagonal matrix with the vector as its diagonal. The term's scaling σ controls the intensity of the diffusion and encourages in particular particles far from $v_\alpha(\hat{\rho}_t^N)$ to explore larger regions.

2.2.2.2 Implementation: Training a Neural Network using Consensus-Based Optimization

To demonstrate how CBO can be applied to train a neural network, we present a simplified Python code snippet. This example illustrates the process of defining a neural network, setting up the CBO optimizer and executing the training loop.

Listing 2.2: Training a neural network using CBO in Python [7]

```python
from cbo import Optimizer, Loss

# Define a neural network by extending nn.Module
class MyNeuralNet(nn.Module):
    def __init__(self):
        # Define the architecture of the network

    def forward(self, x):
        # Define the forward pass of the network

# Initialize the model, CBO optimizer and loss function
model = MyNeuralNet()
optimizer = Optimizer(model, n_particles=100, alpha=50, sigma=0.4**0.5,
                      l=1, dt=0.1, anisotropic=True, eps=1e-2, device='cuda')
loss_fn = Loss(F.nll_loss, optimizer)
```

```python
# Training loop
for epoch in range(100):
    for X, y in train_dataloader:
        loss_fn.set_batch(X, y)
        optimizer.step()
```

As shown in Listing 2.2, the Python implementation provides an overview of how Consensus-Based Optimization (CBO) is applied to train a neural network. The `MyNeuralNet` class extends `nn.Module`, where the neural network structure and forward pass are defined.

The `Optimizer` class is initialized with the CBO dynamics, including important parameters such as `n_particles` (number of particles), `alpha`, `sigma`, and `dt`. These parameters control how the CBO optimizer guides particles toward a consensus solution.

The training loop follows a structure similar to traditional gradient-based methods, iterating over a specified number of epochs. For each batch of training data, the optimizer updates the model parameters based on the CBO dynamics, allowing the model to learn from the data without the need for gradient calculations.

3 Consensus-Based Optimization of Sampled Neural Networks

3.1 Motivation and problem definition

In the context of optimizing neural networks, the combination of Consensus-Based Optimization and the SWIM algorithm presents a compelling approach to address several challenges associated with high-dimensional, nonconvex optimization tasks.

CBO, being a derivative-free optimization method, has already shown promise by achieving around 97% accuracy in the MNIST dataset of handwritten digits after few epochs with just 100 particles [13]. However, one can easily notice that the effectiveness of CBO is significantly influenced by the initial distribution of its particles.

To enhance the initialization process and improve the performance of CBO, we propose using the SWIM algorithm, which is also a derivative-free method, that samples weights and biases more strategically based on the underlying data distribution. By initializing the particles with SWIM, the method ensures that the optimization starts from a more informed position, potentially closer to the global minimum and giving it a head start compared to randomly distributed particles. This combination might not only accelerate the convergence but also improve the overall robustness of the training process, particularly in challenging scenarios such as nonconvex and high-dimensional spaces.

In this thesis we aim to address the challenge of linking both algorithms and efficiently training neural networks in a way that both explores the solution space effectively and converges rapidly to a high-quality solution. The combined SWIM-CBO approach seeks to optimize this process by leveraging its derivative-free nature, strategically initializing network parameters and then refining them through the robust consensus-driven dynamics of CBO. This method is particularly useful in scenarios where traditional gradient-based methods may struggle, such as in the presence of noisy gradients or complex loss landscapes. By integrating SWIM with CBO, we aim to overcome these challenges, providing a more effective and theoretically grounded approach to neural network optimization. The following sections will delve into the details of this combined approach, its implementation and the results of computational experiments demonstrating its efficacy.

3.2 Combined SWIM-CBO Approach

In this section, we present the combined SWIM-CBO approach for optimizing neural networks. For that we will use a model that consists of:

- A dense (fully connected) layer that maps the input to a high-dimensional hidden space.
- A Tanh activation function to introduce non-linearity.
- A linear output layer that maps the hidden representation to the final output.

This architecture seems to work well with sampled network and it allows for flexible initialization and optimization, making it ideal for exploring different strategies in the combined SWIM-CBO approach.

We will explore the following three cases:

- **Case 1:** Reinitialize each particle's model parameters for both layers using SWIM, with a different random seed For each particle. Then, train all parameters in both layers using CBO.
- **Case 2:** Similar to Case 1, but train only the parameters of the final layer (linear layer) using CBO after reinitializing both layers with SWIM.
- **Case 3:** Reinitialize only the dense layer using SWIM, leaving the particles in the linear layer randomly distributed. Then, use CBO to train only the parameters of the final linear layer.

Remark: The term "reinitialize" is used here because the particles' parameters in CBO algorithm are initially randomly distributed, as can be seen in the code snippet below.

Listing 3.1: Particle Initialization in CBO [7]

```python
class Particle(nn.Module):
    def __init__(self, model):
        """
        Represents a particle in the consensus-based optimization.
        Stores a copy of the optimized model.
        :param model: the underlying model.
        """
        super(Particle, self).__init__()
        self.model = deepcopy(model)
        for p in self.model.parameters():
            with torch.no_grad():
                p.copy_(torch.randn_like(p))
```

3.2.1 Converting a Sampled Network to a Neural Network (nn.Module)

In this subsection, we describe the Python code developed to convert a sampled network, generated using the SWIM algorithm, into a neural network from PyTorch's `nn.Module` so that it is compatible with CBO algorithm.

The conversion process involves two main classes:

- **SwimNet Class:** This class defines the neural network architecture, which consists of a dense layer, a Tanh activation function, and a linear output layer. The `load_params_from_swim` method in this class imports the weights and biases derived from the SWIM algorithm into the model's layers.
- **FirstPartModel and SecondPartModel Classes:** These classes are used to split the main model into two separate parts, one containing the dense layer and the other containing the linear output layer. This split allows for the cases where only the final linear layer is to be trained using CBO, while the dense layer parameters remain fixed.

The converted model can then be utilized in the CBO framework, where it can be trained and optimized as part of a particle optimization, leveraging the strengths of both SWIM and CBO.

3.2.2 Reinitializing Particles with the _reinitialize_particles Function

The `_reinitialize_particles` function plays a crucial role in the combined SWIM-CBO approach by reinitializing each particle's model parameters based on different training strategies. This function allows for flexibility in how particles are initialized, whether fully or partially and significantly impacts the effectiveness of the optimization process.

The function supports the three main strategies discussed earlier for initializing particles, that can be specified using the `train_strategy` parameter with one of the following:

- **all:** In this strategy, since all parameters will be trained, the main model and particles are instances of the 'SwimNet' class. We iterate over the particles and for each particle we create a sampled network using SWIM algorithm with a different random seed. Then, we use the `load_params_from_swim` method to load the parameters in both dense and linear layers.
- **lin:** In this strategy, the main model is composed of 'FirstPartModel' (dense layer) and 'SecondPartModel' (linear layer). Since only the parameters of the linear layer will be trained using CBO, the particles are instances of 'SecondPartModel'. We create a sampled network using SWIM, convert it to 'FirstPartModel' and load the parameters. Afterwards, we reinitialize the linear layer of each particle.
- **rand-lin:** Similar to the 'lin' strategy, the main model is composed of 'FirstPartModel' and 'SecondPartModel'. Here only the 'FirstPartModel' is trained using SWIM, leaving the particles randomly distributed in the linear layer.

3.3 Evaluation and Validation

3.3.1 Assessment of Approach Effectiveness

To evaluate the effectiveness of the combined SWIM-CBO approach or any neural network optimization method in general, we consider several key factors that indicate whether the method is performing well.

One of the primary indicators is the evolution of the loss function over the course of training, in our case the Mean Squared Error (MSE) loss. Monitoring how the loss decreases over time gives insight into how well the model is learning. A steady reduction in MSE suggests that the model is fitting the data increasingly well, while any plateau or increase may signal issues such as overfitting or insufficient training.

Convergence time is another critical aspect. This refers to how quickly the model reaches a point where further training does not significantly improve performance. Ideally, the model should converge in a reasonable number of epochs, indicating that the optimization process is efficient. A model that requires fewer epochs to achieve a low MSE is considered more effective, as it demonstrates that the approach is making the most of each training iteration.

In addition to loss and convergence, it is essential to assess the model's robustness. This involves testing the SWIM-CBO approach across different datasets, initial conditions and various types of tasks, such as regression and classification. The method should consistently perform well across these scenarios, not just in specific cases or with a particular loss function. Robustness is crucial in a neural network because it demonstrates that the approach is not only effective under ideal conditions but also reliable in more challenging and varied contexts.

Finally, practical considerations like computational efficiency are also important. This includes evaluating the time required for training and how well the approach scales with more complex models or larger datasets. In fact, the paper "Large Scale Distributed Deep Networks" [14] discusses the trade-offs between model performance and computational resources and emphasizes the importance of computational efficiency in large-scale neural network training. Achieving a balance between low loss, quick convergence, robustness across tasks and computational efficiency is key to determining the overall success and applicability of the method. Ensuring that the method remains practical and scalable while delivering strong performance across different scenarios is essential for its adoption in real-world applications.

3.3.2 Hyperparameter Tuning

Hyperparameters play a crucial role in determining how effectively a model will be trained and how well it will predict. Even a small adjustment to one of these parameters can significantly impact the model's performance. Given their importance, we will discuss in this subsection each of the hyperparameters used in our approach in detail, examining their potential individual effects on the training process [12, 11].

- The **Number of Particles** controls how thoroughly the solution space is explored. More particles allow for a more comprehensive search, increasing the chances of finding the global minimum. However, too many particles can lead to increased computational overhead and slower training, while too few may result in insufficient exploration, potentially causing the model to converge to a local minimum.
- The **Number of Neurons in the Hidden Layer** affects the model's capacity to capture complex patterns in the data. A larger number of neurons allows the model to learn more intricate relationships, potentially improving accuracy. However, using too many

neurons can lead to overfitting, where the model performs well on the training data but poorly on unseen data. Therefore, it is important to find a balance between having enough neurons to capture the complexity of the data without making the model overly complex.

- The **Alpha (α)** parameter controls the weight of the consensus term. Higher values of this parameter make the particles converge faster to the consensus point. However, if set too high, it can lead to instability and premature convergence to suboptimal solutions. In practice, α needs to be carefully tuned to balance the trade-off between rapid convergence and maintaining diversity among particles.

- The **Sigma (σ)** parameter governs the amount of noise in the updates of particle positions. Introducing noise is essential for ensuring that particles can explore the solution space and avoid getting trapped in local minima. However, too much noise can lead to divergence and erratic behavior, while too little noise might prevent sufficient exploration, leading to poor optimization. Since the particles are already well-initialized with the SWIM algorithm, σ should be kept at a moderate level to prevent losing the benefits of the initial setup.

- The **Lambda (λ)** parameter dictates the strength of the consensus term, which encourages particles to stay close to the consensus point. A higher λ value results in particles clustering more tightly around the consensus, which can accelerate convergence but might also reduce the ability to explore diverse areas of the solution space. Choosing the right value for λ is crucial for maintaining a balance between exploitation of known good regions and exploration of new ones.

- The **Time Step (dt)** determines the rate at which particle positions are updated. A smaller dt allows for finer updates and smoother trajectories, potentially leading to more stable convergence. However, this might also slow down the overall training process. Conversely, a larger dt can speed up convergence but may risk overshooting optimal solutions, especially if combined with high σ values.

- The **Epsilon (ϵ)** parameter sets the threshold for applying random drift. Smaller values of ϵ mean that random drift is applied more frequently, which can help the particles escape local minima and explore the solution space more thoroughly. However, if applied too frequently, it could disrupt convergence. Careful tuning of ϵ is necessary to ensure that random drift is used effectively to enhance exploration without hindering overall convergence.

The objective is to find a set of hyperparameters that lead to the best possible performance in terms of loss minimization, convergence speed and model stability. It is also important to validate the tuning process on a separate validation set or through cross-validation to ensure that the chosen parameters generalize well to unseen data.

3.3.3 Regularization

In this subsection, we will talk briefly about **regularization**, a technique used when training neural networks to prevent overfitting, ensuring that models generalize well to unseen data. Overfitting occurs when a model learns not only the underlying patterns in the training data but also the noise, leading to poor performance on new data. Regularization adds a penalty to the loss function during training, which discourages the model from becoming too complex and overly fitted to the training data.

One of the most commonly used regularization methods, which will also be used later in the experiments, is **L1-regularization**. Introduced by Robert Tibshirani [15] and also known as **Lasso (Least Absolute Shrinkage and Selection Operator)**, L1-regularization adds a penalty term to the loss function that is proportional to the sum of the absolute values of the model's weights:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{original}} + \frac{\lambda}{\sqrt{N}} \sum_i |w_i|$$

Here, $\mathcal{L}_{\text{total}}$ represents the total loss, $\mathcal{L}_{\text{original}}$ is the original loss (such as Mean Squared Error), w_i denotes the model's weights, λ is the regularization parameter that controls the strength of the penalty and N stands for the number of neurons. The division by $\sqrt{\text{N}}$ helps to normalize the regularization term relatively to the size of the model.

One of the benefits of L1 regularization is that it encourages **sparsity** in the model parameters by shrinking some of the weights to zero. And by driving some of the weights to zero, it inherently performs a **feature selection**, excluding less important features from the model [16]. This is especially useful in high-dimensional datasets where many features may be irrelevant or redundant. Additionally, after training, the neurons corresponding to the weights that have been shrunk to zero are inactive and can be removed, leading to a simpler and more efficient model. To ensure consistency across our analysis and effectively leverage this sparsity, a fixed value of $\lambda = 0.01$ was applied for L1 regularization in all experiments that will be discussed later in this thesis.

3.4 Computational Experiments

In this section, we describe three different computational experiments, that we conducted and were designed to evaluate the effectiveness of the combined SWIM-CBO approach. These experiments span a range of complexities, from a simple sine function to a modified MNIST dataset for binary classification. Then, we will go over the numerical results and compare them to assess the performance of the SWIM-CBO method across different tasks and datasets.

As discussed earlier, each experiment will be tested using three different training strategies:

1. Reinitializing all model parameters for each particle using SWIM and training them using CBO.
2. Reinitializing both layers with SWIM and training only the linear layer using CBO.

3. Reinitializing only the dense layer with SWIM, leaving particles in the linear layer randomly distributed and training only the linear layer using CBO.

3.4.1 Sine Function

The first experiment involves predicting the sine of a given input, starting with $\sin(x)$ and progressively increasing the frequency by considering $\sin(2x)$, $\sin(3x)$ and so on, to evaluate the model's ability to generalize across varying frequencies. We generate 100 data points over the interval $[-\pi, \pi]$ and split them into training and test sets using the `train_test_split` method from `sklearn.model_selection`. We opted for a 60% test split to challenge the model with more test data and evaluate its performance under less training data. However, typical splits often involve around 2/3 of the data for training and 1/3 for testing.

- **Data Generation:** The data is generated starting with a simple sine function $\sin(x)$ and then progressively using higher frequencies to observe the effect on model performance.
- **Model Configuration:** A neural network with an input dimension of 1, a hidden layer with 15 neurons and an output dimension of 1 is configured.

3.4.2 A More Complicated Function

The second experiment tests the approach on a more oscillatory function defined as $\sin(2x_1) \cdot \cos(3(x_1 + x_2))$. This function introduces more complexity to assess how well the model handles multiple input features and intricate patterns.

- **Data Generation:** We generate 10,000 data points by creating a grid over the interval $[-\pi, \pi]$ for two variables x_1 and x_2, then apply the combined sine-cosine function.
- **Model Configuration:** The neural network model is adjusted to an input dimension of 2, with a hidden layer of 750 neurons and an output dimension of 1.

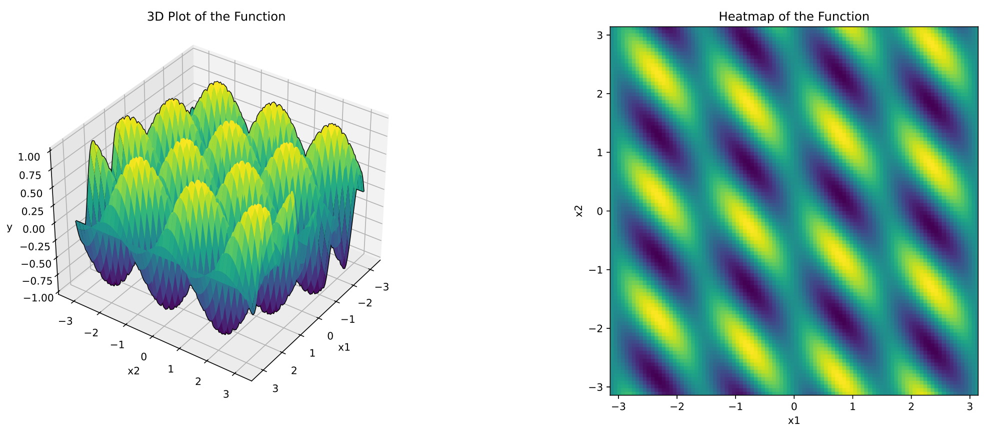


Figure 3.1: 3D Plot and Heatmap of the Function $\sin(2x_1) \cdot \cos(3(x_1 + x_2))$.

3.4.3 Simplified MNIST Dataset

The third experiment applies the SWIM-CBO method to a simplified version of the MNIST dataset, focusing on binary classification between the digits 1 and 8. This experiment is intended to evaluate the approach on categorical data and classification tasks.

- **Data Preparation:** The MNIST dataset is filtered to include only the digits 1 and 8 and the labels are one-hot encoded for binary classification. We again used the `train_test_split` function, allocating 60% of the data to the test set to emphasize model evaluation under limited training conditions.
- **Model Configuration:** The model is defined with an input dimension of 784, a hidden layer with 400 neurons and an output dimension of 2.

The following code snippet demonstrates the data generation process:

Listing 3.2: Data Generating in Simplified MNIST Dataset

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Fetch the MNIST dataset
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist["data"], mnist["target"].astype(np.int8)

# Filter out only the digits 1 and 8
mask = (y == 1) | (y == 8)
x_dataset = np.array(X[mask]).astype(np.float32)
y_dataset = np.array(y[mask]).astype(np.int64)

# One-hot encode the labels
onehot_encoder = OneHotEncoder(sparse_output=False)
y_dataset = onehot_encoder.fit_transform(y_dataset.reshape(-1, 1))

# Split the data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(x_dataset, y_dataset,
                                                   test_size=0.6, random_state=123)
```

As shown in Listing 3.2, this Python implementation outlines the data preparation process for a simplified MNIST dataset, focusing solely on the digits 1 and 8.

The `fetch_openml` function retrieves the full MNIST dataset, which includes all digits from 0 to 9. We then filter the dataset to keep only the digits labeled as 1 and 8 by applying a mask. This selective filtering reduces the complexity of the classification task, enabling a more focused binary classification between these two digits.

Next, the `OneHotEncoder` is used to transform the labels into a one-hot encoded binary format, where each label is represented by a binary vector.

Finally, the dataset is split into training and testing sets using the `train_test_split` function, with 60% of the data allocated to testing. This higher testing proportion is designed to challenge the model more rigorously and provide a robust evaluation compared to the standard 1/3 testing split. The random seed ensures reproducibility in the dataset splitting process.

3.4.4 Results

All computational experiments described earlier were conducted on a system equipped with an AMD Ryzen 7 7840HS CPU 3.80 GHz, 32 GB of DDR5 RAM and an NVIDIA GeForce RTX 4070 Mobile GPU with 8 GB RAM and CUDA support utilized in the CBO algorithm.

The following subsections will detail the results obtained from the various experiments conducted using the three different training strategies.

3.4.4.1 Reinitialization Impact

In this subsection, we aim to compare the impact of different reinitialization strategies on the optimization process across the three experiments. We used the same hyperparameters across the CBO-only, all, lin and rand-lin strategies to observe the general tendencies of each approach under consistent conditions.

Hyperparameter	**Value Used in Sine Function**	**Value Used in Complicated Function**	**Value Used in Simplified MNIST**
Number of Epochs	10	10	25
Neurons in Hidden Layer	18	750	400
Number of Particles	30	75	100
Batch Size	7	100	200
Alpha (α)	50	50	75
Sigma (σ)	5e-6	5e-6	5e-7
Lambda (λ)	1	1	1
Time Step (dt)	1e-3	5e-3	5e-3
Epsilon (ϵ)	1e-3	1e-3	1e-3

Table 3.1: Hyperparameters Values Used Across Different Experiments.

Experiment	Strategy	Final MSE Loss	Reinitialization Time	Training Time
Sine Function	CBO-only	0.7582	0s	1.431s
	all	0.00904	0.0142s	1.282s
	lin	6.91e-7	0.009s	0.975s
	rand-lin	0.2503	0.0011s	0.9721s
Complicated Function	CBO-only	1.672168	0s	17.524s
	all	0.001635	34.707s	18.437s
	lin	0.026376	30.348s	11.858s
	rand-lin	7.952826	0.0015s	11.7787s
Simplified MNIST	CBO-only	6.5949	0s	58.567s
	all	0.023731	19.7s	64.0818s
	lin	0.012755	14.285s	32.884s
	rand-lin	0.23454	0.0324s	29.663s

Table 3.2: Combined Results for All Experiments: Final MSE Loss, Reinitialization and Training Time.

Sine Function Experiment

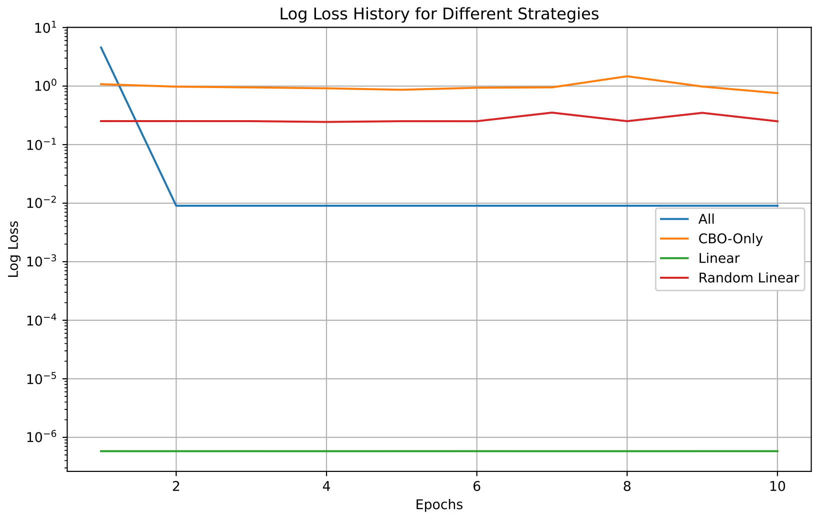


Figure 3.2: Log Loss Evolution Comparison for Sine Function Experiment.

Discussion: The results in Table 3.2 clearly demonstrate the significant impact of the SWIM-CBO approach on both training efficiency and model accuracy. The "all" and "lin" reinitialization strategies, in particular, achieved final losses several orders of magnitude lower than the CBO-only approach. This is due to SWIM's effectiveness in initializing the network parameters close to the optimal solution. The "rand-lin" strategy, while not as effective as the

others, still outperforms CBO-only in both final loss and time efficiency.
The log loss evolution graph (Figure 3.2) further illustrates this point, showing a rapid drop in loss for the "all" strategy right from the first epoch, indicating that these models are already near-optimal at initialization. The "rand-lin" strategy and especially the "lin" strategy show a better starting point than CBO-only approach, which starts with a much higher initial loss and converges more slowly. This highlights the impact of the reinitialization using SWIM for a more efficient optimization.

Complicated Function Experiment

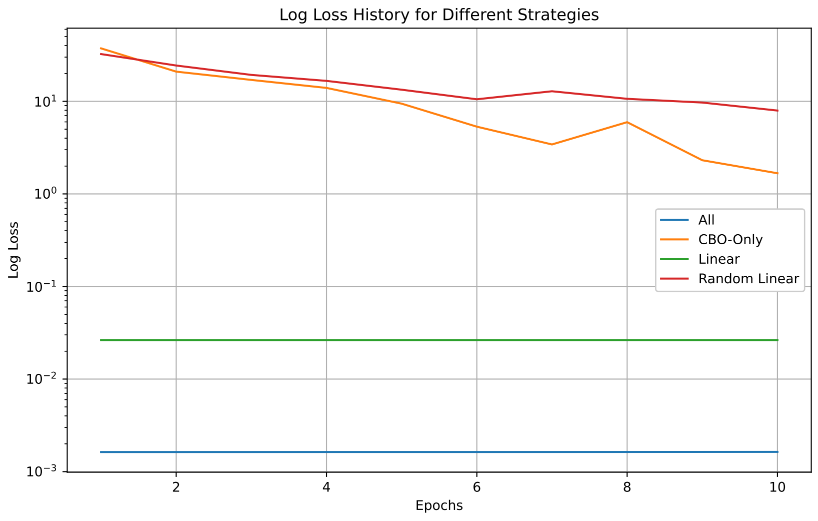


Figure 3.3: Log Loss Evolution Comparison for Complicated Function Experiment.

Discussion: The results in Table 3.2 once again highlight the impact of the SWIM reinitialization on the final model performance. The "all" and "lin" strategies yielded significantly lower final MSE losses compared to the CBO-only approach, showcasing the effectiveness of SWIM in providing a more informed initialization.

Although the "rand-lin" strategy started with a better initial loss compared to CBO-only, as it can be seen in the log loss evolution graph (Figure 3.3), it ended up with a higher final loss after 10 epochs. This suggests that while SWIM reinitialization provides a good starting point, it is not always sufficient on its own without adequate training epochs. The "all" and "lin" strategies on the other hand show from the beginning a constant low loss and maintain it consistently throughout the training process.

Simplified MNIST Dataset Experiment

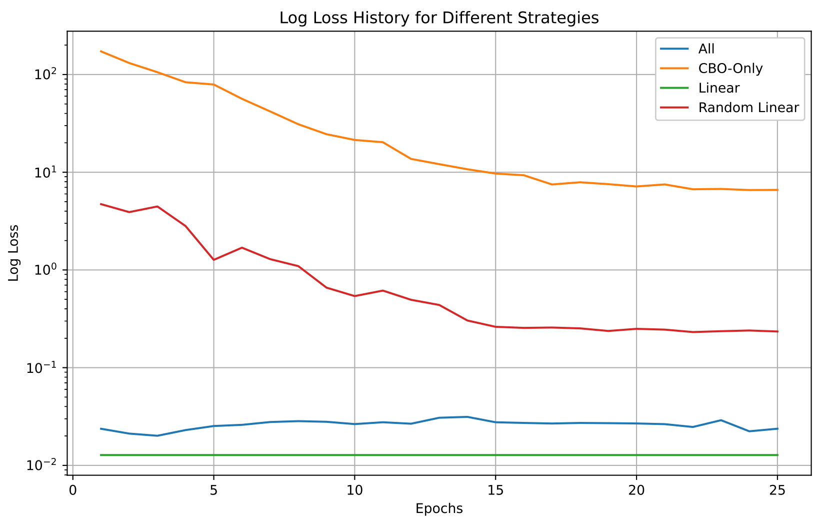


Figure 3.4: Log Loss Evolution Comparison for Simplified MNIST Dataset Experiment.

		Predicted	
		1	8
Actual	1	3440	1268
	8	1987	2127

(a) CBO-only Strategy

		Predicted	
		1	8
Actual	1	4673	35
	8	79	4035

(b) All Strategy

		Predicted	
		1	8
Actual	1	4674	34
	8	37	4077

(c) Lin Strategy

		Predicted	
		1	8
Actual	1	4453	255
	8	1411	2703

(d) Rand-Lin Strategy

Figure 3.5: Confusion Matrices for Training Strategy Used in The Simplified MNIST Dataset Experiment: True labels vs. Predicted labels for digits 1 and 8.

Discussion: As with the previous experiments, the results in Table 3.2 show that the SWIM-CBO approach achieves significantly better outcomes compared to the CBO-only strategy. Both the "all" and "lin" strategies produced much lower final MSE losses, highlighting the effectiveness of the SWIM algorithm in providing a well-informed initialization.

The log loss evolution graph in Figure 3.4 further supports this observation, showing that the SWIM-CBO approaches start with a much lower loss than the CBO-only method. This improved start is crucial in setting the trajectory for more effective training.

To quantify the performance in terms of classification accuracy, we use the following formula

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Samples}}$$

- **CBO-only**: Accuracy $= \frac{3440+2127}{8822} \approx 63.11\%$
- **All**: Accuracy $= \frac{4673+4035}{8822} \approx 98.71\%$
- **Lin**: Accuracy $= \frac{4674+4077}{8822} \approx 99.19\%$
- **Rand-Lin**: Accuracy $= \frac{4453+2703}{8822} \approx 81.12\%$

These results clearly demonstrate that the SWIM-CBO strategies, particularly "all" and "lin", yield higher classification accuracy compared to the CBO-only and "rand-lin" strategies, highlighting the advantage of SWIM reinitialization. By effectively skipping the initial struggle of finding a good starting point, the "all" and "lin" strategies achieve excellent results in fewer epochs, demonstrating their efficiency in reaching optimal solutions more rapidly.

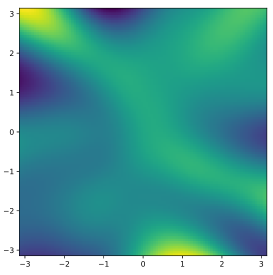
3.4.4.2 Number Of Epochs And Convergence Time

While the SWIM-CBO approach demonstrated superior performance with the specific hyperparameters used in the previous experiments, this might not be universally true across all scenarios. Each experiment and training strategy may require different hyperparameter configurations to achieve optimal results. Therefore, to ensure a fair comparison, we will train each strategy with its optimal hyperparameters until convergence. We will then compare the time required, the number of epochs and the overall efficiency of the training process.

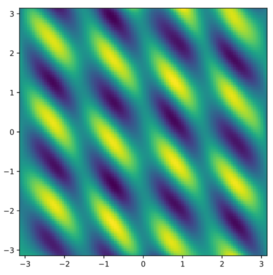
Since this analysis requires substantial computational resources and time, we will focus solely on the complicated function experiment with the model described in Section 3.2 using 750 neurons in the hidden layer.

Strategy	Optimal Hyperparameters	Time Spent	Number of Epochs	Final MSE Loss
CBO-only	Particles: 100, Alpha: 50 Sigma: 5e-2, Lambda: 1 dt: 0.2, Epsilon: 1e-2	166.022s	75	0.2345
all	Particles: 75, Alpha: 75 Sigma: 5e-6, Lambda: 1 dt: 5e-3, Epsilon: 1e-3	40.184s	5	0.001629
lin	Particles: 75, Alpha: 50 Sigma: 5e-2, Lambda: 1 dt: 0.2, Epsilon: 1e-2	35.507s	5	0.02638
rand-lin	Particles: 100, Alpha: 50 Sigma: 3e-2, Lambda: 1 dt: 0.5, Epsilon: 1e-2	222.127s	120	0.1996

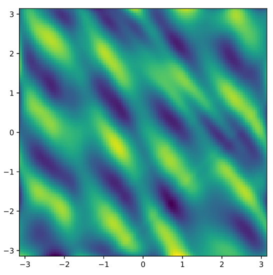
Table 3.3: Optimal Hyperparameters, Time Spent, Number of Epochs and Final Loss for Each Strategy in the Complicated Function Experiment.



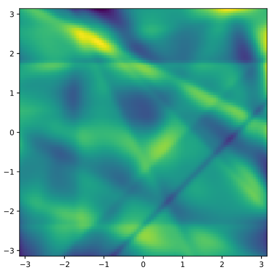
(a) CBO-only Strategy



(b) All Strategy



(c) Lin Strategy



(d) Rand-Lin Strategy

Figure 3.6: Final Heatmap of The Predicted Functions for Each Training Strategy under Optimal Conditions in the Complicated Function Experiment.

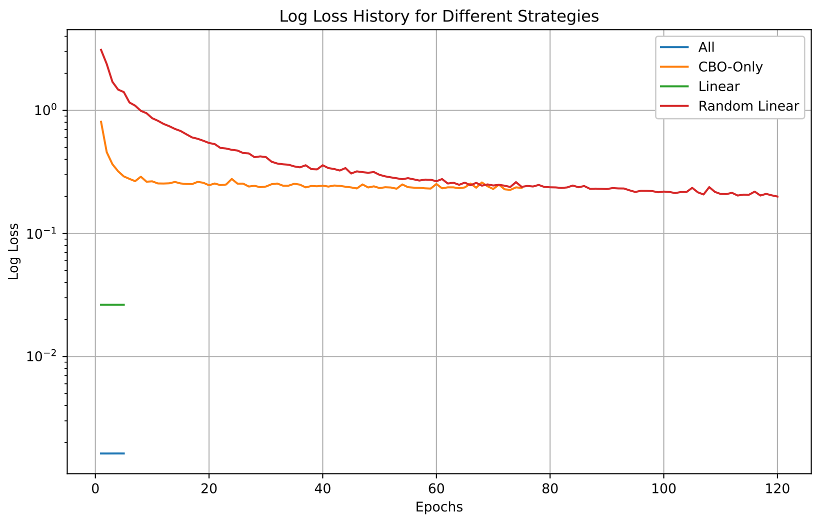


Figure 3.7: Log Loss Evolution Comparison for Each Training Strategy under Optimal Conditions in the Complicated Function Experiment.

Discussion: The results confirm that the SWIM-CBO approach consistently outperforms the CBO-only strategy, even under optimal conditions for each training strategy. This conclusion is supported by the data presented in Table 3.3, where both "all" and "lin" strategies achieved significantly lower MSE losses compared to both the CBO-only and "rand-lin" strategies, though "rand-lin" still benefits from the SWIM reinitialization but was less effective than the other combined approaches.

Analyzing the results further, the heatmaps in Figure 3.6 visually illustrate the superior performance of the combined approaches. The "all" and "lin" strategies not only maintained a low loss and required fewer epochs but produced heatmaps that more closely resemble the true function. As seen in Figure 3.7, their loss remained stable and almost constant, indicating that additional epochs would be unnecessary as the models had already efficiently converged.

However, it is important to note that the "rand-lin" strategy exhibited a slower start compared to CBO-only, requiring more time and epochs to converge. Despite this initial lag, it ultimately produced better results than CBO-only, as evidenced in both the heatmaps and the final MSE losses. This suggests that even with the same number of particles, the random linear strategy benefits from the SWIM reinitialization and eventually surpassing the CBO-only approach in performance.

Interestingly, the CBO algorithm on the "lin" strategy did not show further improvement after reinitialization, resulting in a final loss similar to that achieved by SWIM alone. This suggests that SWIM already provides a strong approximation and since the "lin" strategy only refines the linear layer, there is limited room for further enhancement.

Overall, the combined SWIM-CBO approaches, particularly "all" and "lin", demonstrate a more efficient optimization process, reaching optimal solutions more rapidly and with fewer resources compared to CBO-only and "rand-lin." This confirms the robustness and effectiveness of SWIM reinitialization across different configurations.

3.4.4.3 Hyperparameter Sensitivity and Loss Evolution

Since each scenario is unique and requires its own set of optimal hyperparameters, we explore, in this subsection, the role of individual hyperparameters by varying them one at a time while keeping the others fixed, in order to observe their impact on the model's performance. In other words, we will analyze how specific changes in hyperparameters affect the loss evolution and, consequently, the overall effectiveness of the training strategy.

Impact of Sigma on Convergence and Loss

We begin our sensitivity analysis by focusing on the σ parameter in the Complicated Function Experiment using the "all" training strategy. σ is chosen because it directly influences the amount of noise introduced during the training process, playing a critical role in preventing the particles from getting trapped in local minima and affecting the exploration of the solution space. To observe its impact on the model performance, we will fix all other hyperparameters at their optimal values, as determined in Subsection 3.4.4.2, but with an increased number of epochs to better observe the loss evolution. We will vary σ across a range of values $\{10^{-1}$, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, $10^{-7}\}$.

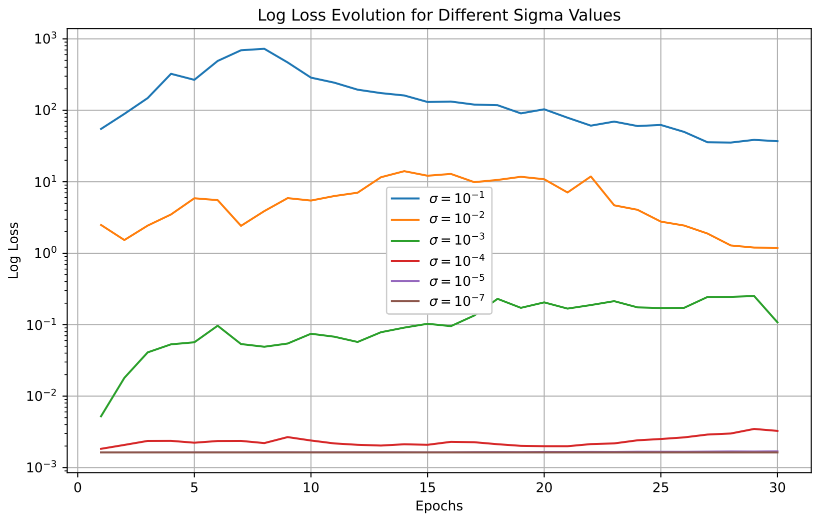


Figure 3.8: Log Loss Evolution Comparison for Different Sigma Values in the Complicated Function Experiment.

Impact of Number of Particles on Convergence and Loss

We now shift our focus to analyzing the impact of the number of particles in the Complicated Function Experiment using the "all" training strategy. The number of particles is a crucial parameter as it determines the extent to which the solution space is explored. We expect that a higher number of particles lead to more thorough exploration and potentially faster convergence or better final results. To evaluate its impact, we will maintain all other hyperparameters at their optimal values, as determined in Subsection 3.4.4.2, while varying the number of particles across a range $\{50, 75, 100, 150, 200\}$.

Number of Particles	Reinitialization Time	Training Time
50	21.529s	20.174s
75	31.744s	24.676s
100	40.965s	32.527s
150	63.894s	45.736s
200	91.422s	61.332s

Table 3.4: Reinitialization and Training Times for Different Numbers of Particles in the Complicated Function Experiment.

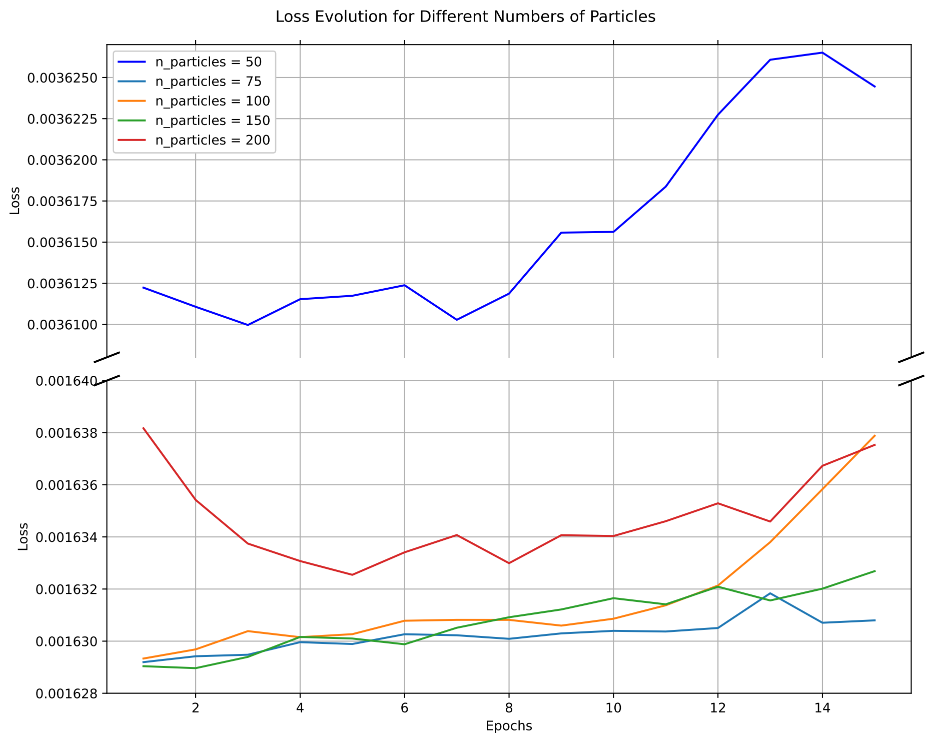


Figure 3.9: Loss Evolution Comparison for Different Numbers of Particles in the Complicated Function Experiment.

Discussion: The results demonstrate that increasing the number of particles initially leads to better performance, as seen ine the Figure 3.9 when comparing 50 and 75 particles. The model with 75 particles outperformed the one with 50, indicating that a greater number of particles allows for more thorough exploration of the solution space, resulting in faster convergence. However, this trend did not continue with higher particle counts. The performance with 100, 150 and 200 particles was nearly the same as with 75 particles and in some cases, particularly with 200 particles, the results were worse. This suggests that beyond a certain point, adding more particles does not necessarily improve the model's performance. Instead, it increases the computational cost and time, leading to inefficient training without significant gains in accuracy.

As shown in Table 3.4, the reinitialization and training times increased substantially with the number of particles. Therefore, while a sufficient number of particles is essential for effective optimization, there is a point of diminishing returns where more particles do not contribute to better results but instead waste computational resources. The goal should be to find a balanced number of particles that provides efficient training and optimal performance without unnecessary overhead.

4 Conclusion

4.1 Summary

This thesis explored the challenges of optimizing neural networks, particularly in complex, high-dimensional, nonconvex spaces and proposed a solution by combining two derivative-free approaches, leveraging the strength of both Sample Where It Matters (SWIM) and the Consensus-Based Optimization (CBO) to enhance the training of neural networks.

In the first part of the thesis, we provided an overview of the current state-of-the-art in neural network training methodologies, including sampled networks, SWIM and CBO as one of the particle optimization methods. The second part focused on the proposed combined SWIM-CBO approach, offering insights into the underlying motivations, problem definition, implementation details and presenting the results of computational experiments to evaluate the performance of the approach.

Three distinct training strategies -"all", "lin" and "rand-lin"- were presented in Section 3.2 and examined throughout the thesis. In the subsequent section, we assessed the effectiveness of the approach and conducted a detailed analysis of the hyperparameters to understand their impact on the model's performance.

The experiments and results were presented in Section 3.4, where the SWIM-CBO approach was evaluated across various tasks, including regression and classification. The results consistently demonstrated the effectiveness of the SWIM-CBO methods compared to the CBO-only approach, showing significant improvements in convergence time, final loss and overall performance. Additionally, we analyzed the sensitivity of the model to changes in hyperparameters, such as the number of particles and the noise parameter σ, highlighting their critical role in the optimization process.

4.2 Discussion

The combined SWIM-CBO approach has demonstrated its potential to improve the optimization process when compared to the CBO-only method. Results from the computational experiments consistently highlighted the key strengths of this approach, particularly in terms of reduced convergence time and lower final loss, particularly in cases where noisy gradients or complex loss landscapes hinder performance, in which traditional methods often face difficulties.

A key insight from this work is the critical role played by SWIM in providing a more informed initialization and better initial conditions. By strategically sampling the parameter space, SWIM allowed the model to begin optimization from a more advantageous starting point, leading to faster convergence. Both the "all" and "lin" strategies, which fully reinitialize the parameters using SWIM, clearly outperformed the CBO-only approach in terms of convergence speed and final loss values. This advantage is especially pronounced in high-dimensional problems, where random initialization often struggles to provide an optimal starting point for optimization. The promising results from these experiments suggest that SWIM's impact is not problem-specific but could be generalized to a wider range of tasks.

The performance of the "rand-lin" strategy, while an improvement over CBO-only, underscored the importance of effectively initializing all layers of the network. Leaving the linear layer randomly initialized introduced variability that hindered the overall performance of the model. This suggests that SWIM's advantage extends across both non-linear and linear layers and and initializing all layers could be beneficial in more complex neural networks. However, the extent to which this holds in deeper and more complex networks remains to be explored.

Beyond initialization, the analysis of hyperparameters revealed key trade-offs that must be managed to achieve optimal performance. Increasing the number of particles in the CBO algorithm improves exploration of the solution space by allowing for a more exhaustive search, thereby reducing the risk of becoming trapped in suboptimal solutions. However, this benefit comes with higher computational costs. As seen in the experiments, beyond a certain threshold, increasing the number of particles resulted in diminishing returns, with increased training times and minimal improvement in final loss. This highlights that while particle count is crucial for exploration, there is a point where the computational cost outweighs the benefits. Achieving the right balance between particle number and computational efficiency is essential, depending on the problem at hand.

The noise parameter σ also plays a crucial role in determining the effectiveness of the optimization process. Noise helps promote exploration and prevents particles from getting stuck in local minima, but excessive noise can disrupt the optimization process, leading to inefficient or unstable convergence. Tuning σ is therefore essential for balancing exploration and exploitation during training.

These findings reinforce the importance of careful hyperparameter tuning. Each hyperparameter has a distinct impact on the model's performance and the combined SWIM-CBO approach shows that optimal hyperparameters must be tailored to the specific task. In real-world applications, where problem complexity and data characteristics vary widely, careful tuning is crucial to balance the trade-offs between exploration, convergence and computational cost.

In conclusion, the SWIM-CBO approach presents a promising method for improving neural network training, particularly in terms of convergence speed and final performance. However, its full potential has yet to be realized, as more extensive testing is required on larger models and more challenging datasets. The method's scalability and effectiveness in more demanding scenarios remain areas for future exploration.

The next section will outline potential directions for future work, including ways to further test and improve the SWIM-CBO approach, as well as its potential application to more complex neural network architectures and broader optimization challenges.

4.3 Future Work

Building on the findings of this thesis, several promising avenues for future research could further enhance and extend the SWIM-CBO approach. One significant direction involves extending the SWIM-CBO method to larger and more complex neural architectures. While this study focused on relatively simple models to demonstrate the approach's effectiveness, future work could explore its application to more intricate models. This exploration is crucial because mastering SWIM-CBO's scalability and efficiency in complex architectures would lay the foundation for its application to real-world datasets, which often involve high complexity and large-scale data.

Further research should also consider a deeper investigation into the impact of additional hyperparameters. While this thesis analyzed the sensitivity of a select few, other parameters such as timestep, L1-constant or alpha could also play a significant role in optimizing the SWIM-CBO approach. Understanding the influence of these parameters could lead to improved guidelines for their selection, thereby enhancing the robustness and effectiveness of the combined optimization method.

Additionally, experimenting with different activation functions in the combined approach could yield valuable insights. Activation functions play a crucial role in determining the output of neural networks and testing the SWIM-CBO method across various activation functions could reveal which combinations are most effective in different scenarios.

Another important area for future research is the exploration of alternative regularization techniques. This thesis employed L1 regularization to encourage sparsity in model parameters, aiding in feature selection. However, future studies could investigate the effects of L2 regularization, which penalizes the square of the weights or elastic net regularization, which combines both L1 and L2 penalties. Such an investigation could provide valuable insights into the trade-offs between sparsity and generalization, leading to more comprehensive hyperparameter tuning strategies that are better suited to specific tasks.

Ultimately, the work on more complex neural architectures and the deeper understanding of hyperparameters are crucial preparatory steps for applying the SWIM-CBO approach to real-world datasets. Testing this method on complex, noisy and large-scale datasets from domains such as natural language processing, computer vision, and bioinformatics would validate its practical utility and assess its performance across diverse scenarios. Addressing these real-world challenges would not only demonstrate the approach's robustness but could also identify areas for further refinement, contributing to the broader field of derivative-free optimization in machine learning.

List of Figures

List of Tables

Bibliography

[1] J. Kufel, K. Bargieł-Łączek, S. Kocot, M. Koźlik, W. Bartnikowska, M. Janik, Ł. Czogalik, P. Dudek, M. Magiera, A. Lis, I. Paszkiewicz, Z. Nawrat, M. Cebula, and K. Gruszczyńska. "What Is Machine Learning, Artificial Neural Networks and Deep Learning?—Examples of Practical Applications in Medicine". In: *Diagnostics* 13.15 (2023). ISSN: 2075-4418. DOI: 10.3390/diagnostics13152582. URL: https://www.mdpi.com/2075-4418/13/15/2582.

[2] M. Islam, G. Chen, and S. Jin. "An overview of neural network". In: *American Journal of Neural Networks and Applications* 5.1 (2019), pp. 7–11.

[3] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[5] E. L. Bolager, I. Burak, C. Datar, Q. Sun, and F. Dietrich. *Sampling weights of deep neural networks*. 2023. arXiv: 2306.16830 [cs.LG]. URL: https://arxiv.org/abs/2306.16830.

[6] F. Dietrich. *swimnetworks*. https://gitlab.com/felix.dietrich/swimnetworks. Python code repository. 2023.

[7] I. Tukh. *cbo-in-python*. https://github.com/Igor-Tukh/cbo-in-python. Python code repository. 2022.

[8] J. E. Dobson. "On Reading and Interpreting Black Box Deep Neural Networks". In: *International Journal of Digital Humanities* 5.2 (2023), pp. 431–449. DOI: 10.1007/s42803-023-00075-w. URL: https://doi.org/10.1007/s42803-023-00075-w.

[9] A. Gad. "Particle Swarm Optimization Algorithm and Its Applications: A Systematic Review". In: *Archives of Computational Methods in Engineering* 29 (Apr. 2022), pp. 2531–2561. DOI: 10.1007/s11831-021-09694-4.

[10] C.-H. Chen, T.-K. Liu, and J.-H. Chou. "A Novel Crowding Genetic Algorithm and Its Applications to Manufacturing Robots". In: *IEEE Transactions on Industrial Informatics* 10 (Aug. 2014). DOI: 10.1109/TII.2014.2316638.

[11] M. Fornasier, T. Klock, and K. Riedl. "Convergence of Anisotropic Consensus-Based Optimization in Mean-Field Law". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2022, pp. 738–754. ISBN: 9783031024627. DOI: 10.1007/978-3-031-02462-7_46. URL: http://dx.doi.org/10.1007/978-3-031-02462-7_46.

[12] M. Fornasier, T. Klock, and K. Riedl. *Consensus-Based Optimization Methods Converge Globally*. 2024. arXiv: `2103.15130 [math.NA]`. URL: `https://arxiv.org/abs/2103.15130`.

[13] M. Fornasier, H. Huang, L. Pareschi, and P. Sünnen. *Anisotropic Diffusion in Consensus-based Optimization on the Sphere*. 2021. arXiv: `2104.00420 [math.OC]`. URL: `https://arxiv.org/abs/2104.00420`.

[14] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. "Large Scale Distributed Deep Networks". In: *Advances in neural information processing systems* (Oct. 2012).

[15] R. Tibshirani. "Regression Shrinkage and Selection via the Lasso". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 58.1 (1996), pp. 267–288. ISSN: 00359246. URL: `http://www.jstor.org/stable/2346178` (visited on 08/25/2024).

[16] I. Nusrat and S.-B. Jang. "A Comparison of Regularization Techniques in Deep Neural Networks". In: *Symmetry* 10.11 (2018). ISSN: 2073-8994. DOI: `10.3390/sym10110648`. URL: `https://www.mdpi.com/2073-8994/10/11/648`.