Karlsruhe Institute of Technology

Scientific Computing Center

Speedup of Hyperparameter Optimization in Propulate Using Approximative Surrogate Models

Bachelor's Thesis of

Vito Dierksen

at the Department of Informatics
Scientific Computing Center (SCC)

Reviewer:	Prof. Dr. Achim Streit
Second reviewer:	Prof. Dr. Bernhard Neumair
Advisor:	Dr. Markus Götz
Second advisor:	Dr. Marie Weiel

25. November 2023 – 19. March 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

Hyperparameter optimization (HPO) is a critical aspect in machine learning (ML) that involves finding the most effective settings for a model's non-trainable parameters, which can significantly impact its performance. Automated approaches like Random Search, Grid Search, Bayesian Optimization (BO), or evolutionary algorithms (EAs) train the neural network (NN) over and over again, testing new hyperparameters (HPs) every time. This is exceptionally compute-intensive, especially as newer models get bigger and bigger.

Predicting the performance of HP configurations during the training process allows for early termination of less promising configurations. This work introduces surrogate models (SMs) into Propulate, a program designed for HPO in high performance computing (HPC) environments. SM have access to interim loss values from each evaluated NN's training during the HPO and decide about stopping it early.

Evaluating static and probabilistic SMs for HPO in Propulate with different datasets and NNs shows a significant decrease in total run time and energy consumption while still finding a loss within small bounds of the best loss found without early stopping.

The new SM implementation added to Propulate is universally usable. It enables finding and developing more powerful SMs in the future.

Zusammenfassung

Hyperparameteroptimierung (HPO) ist ein kritischer Aspekt im Bereich des Maschinellen Lernens (ML), der das Finden der effektivsten Konfigurationen für die nicht trainierbaren Parameter eines Modells umfasst. Schlecht gewählte Hyperparameter (HP) haben einen erheblichen Einfluss auf die Vorhersageleistung des Modells. Automatisierte Ansätze wie Zufallssuche, Rastersuche, Bayes'sche Optimierung (BO) oder evolutionäre Algorithmen (EA) trainieren das neuronale Netzwerk (NN) viele Male, um so unterschiedliche HP-Kombinationen zu testen. Dies ist außerordentlich rechenintensiv, insbesondere da neue NNs immer größer werden.

Die Vorhersage der Performanz von HP-Konfigurationen während des Trainingsprozesses ermöglicht eine frühzeitige Beendigung weniger vielversprechender Konfigurationen. Diese Arbeit führt sogenannte Surrogatmodelle (SMs) in PROPULATE, ein für HPO in Hochleistungsrechenumgebungen konzipiertes Programm, ein. SMs ermöglichen ein frühzeitiges Abbrechen des Trainings von NNs mit schlechten HP. Sie können auf Zwischenverlustwerte des Trainings jedes NNs während der HPO zugreifen und entscheiden basierend auf diesen über dessen vorzeitigen Abbruch.

Die Untersuchung eines statischen und eines probabilistischen SM in PROPULATE anhand verschiedener Datensätze und NNs zeigt eine signifikante Verringerung der Gesamtlaufzeit und des Energieverbrauchs, wobei immer noch ein Verlust innerhalb kleiner Grenzen des besten, ohne frühzeitige Beendigung gefundenen Verlusts erreicht wird.

Die neu in PROPULATE integrierte Implementierung ist universell einsetzbar und ermöglicht so den zukünftigen Einsatz leistungsfähigerer SMs.

Contents

Abstract i

Zusammenfassung iii

1. Introduction 1
   1.1. Structure of This Work 2

2. Foundations and Related Work 5
   2.1. Hyperparameter Optimization Methods 5
   2.2. Propulate 6
   2.3. Learning Curve Prediction 8
      2.3.1. Surrogate Models 8
      2.3.2. Static Termination 8
      2.3.3. Termination with Bayesian Optimization 9
      2.3.4. Ranking 10

3. Approximative Surrogates 13
   3.1. Implementation 13
      3.1.1. Goals 13
      3.1.2. Propulate Implementation 13
      3.1.3. User-Side Implementation 19
   3.2. Static approach 20
   3.3. Probabilistic approach 22

4. Evaluation and Discussion 25
   4.1. Environment and Setup 25
      4.1.1. Computational Environment 25
      4.1.2. Datasets 25
      4.1.3. Seed Variation 26
      4.1.4. Metrics Logged 26
   4.2. Neural Networks 27
      4.2.1. MNIST Classification 27
      4.2.2. CIFAR-10 classification 29
      4.2.3. Naming 30
   4.3. Surrogate Comparison 30
      4.3.1. Default Surrogate 30
      4.3.2. Run time Comparison 30
      4.3.3. Stopping Behavior 32
      4.3.4. Hardware Utilization 35

4.3.5. Energy Consumption and Carbon dioxide equivalent (CO_2e) . 38
5. Conclusion 41
Bibliography 43
A. Appendix 47

List of Figures

1.1. Loss deciles 2

2.1. Propulate Island-Worker structure 7

3.1. Abstract surrogate class 14
3.2. Sequence diagram for evaluate_individual call 17
3.3. Sequence diagram for receive_individuals call 18
3.4. Static Surrogate explanatory graphic 21
3.5. Dynamic Surrogate explanatory graphic 22

4.1. Total Propulate run time vs. best loss 31
4.2. Total run time for each surrogate and seed 31
4.3. Last epoch of every generation 33
4.4. Loss curves for MNIST 34
4.5. Loss curves for CIFAR 34
4.6. Static surrogate median loss vs. best loss 35
4.7. Dynamic surrogate median loss vs. best loss 35
4.8. Generation run time 36
4.9. CPU and memory utilization 38
4.10. GPU utilization 38
4.11. Energy consumption and emitted CO_2e for Propulate runs 39

List of Tables

4.1. Software Versions 25
4.2. Results CSV header 27
4.3. PERUN CSV header 28
4.4. MNIST training parameters 28
4.5. MNIST hyperparameters 28
4.6. CIFAR training parameters 29
4.7. CIFAR hyperparameters 30
4.8. Lowest loss per PROPULATE run 32
4.9. Min, max, and average run time for epoch 1 to 9 37
4.10. Min, max, and average run time between generations 37

A.1. Code Links 47

List of Source Code Snippets

1. Surrogate factory argument to Islands constructor 15
2. Creation of the local surrogate in Propulator init 15
3. Inside _evaluate_individual() 16
4. Example for possible yields in ind_loss() 19
5. User-side PROPULATE surrogate configuration 20

6. ResNet9 layer shuffle 29

Acronyms

CO_2e carbon dioxide equivalent. vi, vii, 1, 26, 28, 38, 39, 41

BO Bayesian Optimization. i, 1, 5, 6, 8–11, 41

CNN convolutional neural network. 1, 2, 27, 28

CSV comma-separated values. 26, 27

EA evolutionary algorithm. i, 1, 6

GA genetic algorithm. 6, 7

GP Gaussian processes. 5, 9, 10, 22, 23, 36

HP hyperparameter. i, 1, 2, 5–11, 19–21, 23, 27, 33, 34

HPC high performance computing. i, 5, 6, 26, 41

HPO hyperparameter optimization. i, 1, 2, 5, 6, 8–11, 13, 20, 29, 38, 41

ML machine learning. i, 5, 6, 8–10

MPI message passing interface. 6, 7, 14–16, 26, 37

NAS neural architecture search. 1

NN neural network. i, 1, 5, 8, 19, 25, 27, 41

ResNet residual neural network. 2, 29, 30

SM surrogate model. i, 2, 8, 11, 13–22, 25–27, 29–33, 35–39, 41, 47

1. Introduction

Recent advances in hardware and methodology have led to the development of large, complex neural networks that demonstrate significant improvements in accuracy for a variety of tasks. A pivotal factor in these advancements is automated hyperparameter optimization.

Traditionally, the process of HP tuning is predominately manual, relying on the expertise of human specialists to guess, test, and adjust HPs. This method involves continuous supervision, allowing interventions when certain configurations underperform. However, not only is this approach time-consuming and reliant on domain-specific knowledge, but it also does not scale well, especially as the complexity of NNs grows.

Automated HPO offers a promising solution to these challenges. Techniques such as Random Search, Grid Search, and BO have been shown to deliver promising results and good HP configurations [24, 5, 8, 33, 37, 4] while taking the human out of the loop and even outperforming human experts [32].

Mimicking natural selection, EAs represent another class of optimization methods that have shown the ability to find optimal or near-optimal solutions in complex search spaces, surpassing other automated approaches [18] and human expert performance [29]. One such program utilizing EAs is Propulate [36]. It will provide the basis for this work.

Nevertheless, all prior HPO methods entail training the same network repeatedly with different HPs, which consumes significant computational resources, meaning these improvements come at a cost, both financially and environmentally. The energy requirements for training state-of-the-art models are substantial, necessitating the use of specialized hardware like GPUs or TPUs.

This not only increases the financial burden due to hardware and electricity costs but also contributes significantly to the environmental impact in terms of carbon footprint. For instance, running neural architecture search (NAS) on a transformer model can result in a carbon footprint equivalent to approximately 284.019 kg of CO_2e [34]. These costs are further amplified when new models are developed, as the whole process repeats itself.

Furthermore, the environmental cost is exacerbated by the fact that a significant portion of the energy used is not derived from carbon-neutral sources. For example, training on a Google cloud server with the newest GPUs located within the EU requires using a server in the Netherlands [10] which only uses 57 % renewable energy whereas the rest has a carbon intensity of 317 gCO_2e/kWh [6].

Moreover, not all HP configurations perform equally well. Aggregating the final losses of all evaluated HP configurations from a Propulate HPO for classification of the MNIST dataset with a convolutional neural network (CNN) and the CIFAR-10 dataset with a residual neural network (ResNet) in Figure 1.1 shows that a significant portion of HPs perform badly.

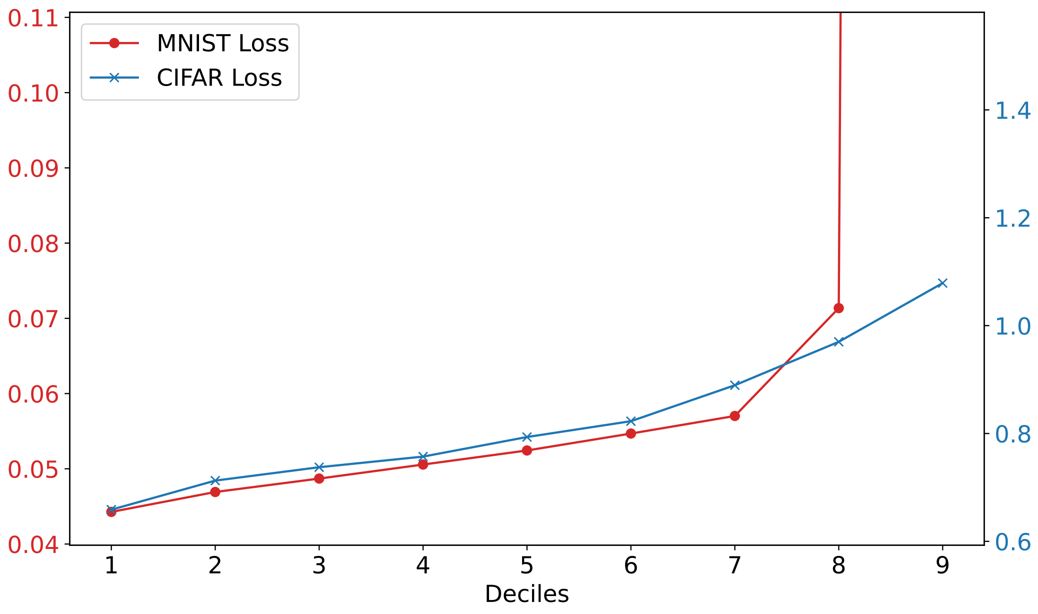


Figure 1.1.: Loss deciles from a Propulate HPO for a CNN classification of MNIST and a ResNet classification of CIFAR with over 192 generations (32 generations per execution on two parallel workers over three different seeds) each. Only the last / final average validation loss of a generation is noted. There is a wide range between the best and worst observed losses. In a best case scenario, only the top few generations are trained fully and all worse performing ones are stopped early to speed up HPO.

This work aims to address this inefficiency by integrating an early stopping mechanism into Propulate. The goal is to develop and test a surrogate-based approach within Propulate that can predict the performance of HP configurations early in the training process. This method aims to reduce computational resources by halting the training of low-performing configurations, thus speeding up the overall optimization process while still identifying high-performing HP sets effectively. Implemented SMs are based on prior research that showed promising results predicting (bad performing) loss curves [22, 35].

1.1. Structure of This Work

This work is structured so as to explore the application of early termination in HPO through the development and evaluation of a corresponding extension in Propulate. Chapter 2 delves into the foundations of HPO, introducing Propulate and presenting various approaches of learning curve prediction. Chapter 3 discusses the concrete SMs, detailing their working principles and practical implementation. Chapter 4 evaluates Propulate's performance using the different SMs, comparing the reduced run time against their achieved results and computational performance. Finally, Chapter 5 concludes by summarizing the findings, highlighting the contributions to PROPULATE, and suggesting directions for future development.

2. Foundations and Related Work

This chapter introduces HPO in the context of NNs and presents different approaches to it. After briefly explaining PROPULATE, an HPC-adapted program for large-scale HPO, all necessary components for implementing early stopping will be outlined.

2.1. Hyperparameter Optimization Methods

HPO is a fundamental aspect of ML that involves selecting the best set of HPs for a learning algorithm to maximize its performance on a given dataset. Unlike model parameters that are learned during training, HPs are set before the training process and control various aspects of the learning algorithm, such as the learning rate, the model's complexity, its number of hidden layers, or any other fixed parameter of the network itself. The goal of HPO is to find the HP values that yield the most accurate predictions, which is crucial because the choice of HPs can significantly impact the model's performance.

The process of HPO can be viewed as an optimization problem, where the objective function to be minimized (or maximized) is often the model's validation error (or accuracy). One common approach is Grid Search, which trains and validates a model on each possible HP combination on a predefined uniform grid of HP values within a fixed range. Although straightforward, Grid Search can be extremely computationally expensive, especially as the number of HPs and their possible values increase. Another approach is Random Search, which samples HP values from a specified probability distribution for a fixed number of iterations. Random Search is already more efficient than Grid Search, especially when only a few HPs affect the final performance of the model significantly [5].

Compared to Grid or Random Search, BO for HPO is a more efficient approach that is particularly suited for scenarios where evaluating the objective function is computationally expensive, such as training complex ML models [8]. Unlike Grid or Random Search, BO uses a probabilistic model, typically a Gaussian processes (GP), to predict and evaluate the performance of HPs in untested regions of the search space, incorporating both prediction accuracy and uncertainty into its decision-making process [33]. At its core, BO models the objective function (e.g., the validation error of a machine learning model) using this probabilistic model. This method is adept at balancing exploration of new HP areas with exploitation of known good configurations, a feature facilitated by the use of acquisition functions like Expected Improvement [31].

BO is particularly effective when the dimensionality of the HP space is manageable and the cost of function evaluations is high [9]. It can significantly reduce the number of evaluations needed to find optimal or near-optimal HPs, often requiring fewer iterations with better results than Grid or Random Search and even outperforming human experts in some cases [8, 33, 37, 4].

Another class of techniques for HPO are EAs, which draw inspiration from the process of natural selection and biological evolution. These algorithms work by generating a population of candidate solutions (i.e., sets of HPs), and then iteratively evolving this population over several generations to improve the performance of the model. At each iteration, individuals in the current population are evaluated based on a fitness function – usually, the model's performance on a validation set. The best-performing individuals are then selected to produce offspring for the next generation through operations such as crossover (mixing HPs from two parents) and mutation (randomly altering some HPs), with the aim of creating a new generation of solutions that is better adapted to the optimization problem. Genetic algorithms (GAs) are a subset of EAs, specifically designed to mimic the process of natural selection more directly [3].

One of the key strengths of EAs in HPO is their ability to explore the search space broadly and avoid getting trapped in local optima, a common problem in gradient-based optimization methods. This characteristic stems from their stochastic nature and mechanisms such as mutation and crossover, which introduce variability and allow for the exploration of diverse regions of the search space.

EAs are particularly effective when the objective function is noisy, discontinuous, or non-differentiable, as they do not rely on the gradient of the function being optimized. This makes them suitable for HPO of complex ML models where the relationship between HPs and model performance is a black box and thus not straightforward.

EAs can be easily parallelized, as the evaluation of each individual in the population can be performed independently. This parallelism significantly reduces the time required for HPO, making EAs a good option for problems with long training times.

The effectiveness of EAs for HPO has been demonstrated multiple times, they can outperform traditional approaches like Grid Search or BO [18] and human experts in many problems [29].

2.2. Propulate

Propulate is a massively parallel, population-based, general-purpose optimizer specifically designed for HPC environments. Traditional parallel GAs are hindered by the need for synchronization after each generation. Propulate omits this synchronization, maintaining a continuous population of evaluated individuals, and allows for asynchronous evaluation and propagation of populations. It features a message passing interface (MPI)-based implementation with variants of selection, mutation, crossover, and migration and is extendable with custom functionality. While Propulate supports multiple populate-based optimizer, the GA is the only variant relevant and utilized in this work. Hence, all future explanations will pertain solely to the GA. Experimental comparisons with the HPO tool Optuna showed that Propulate is

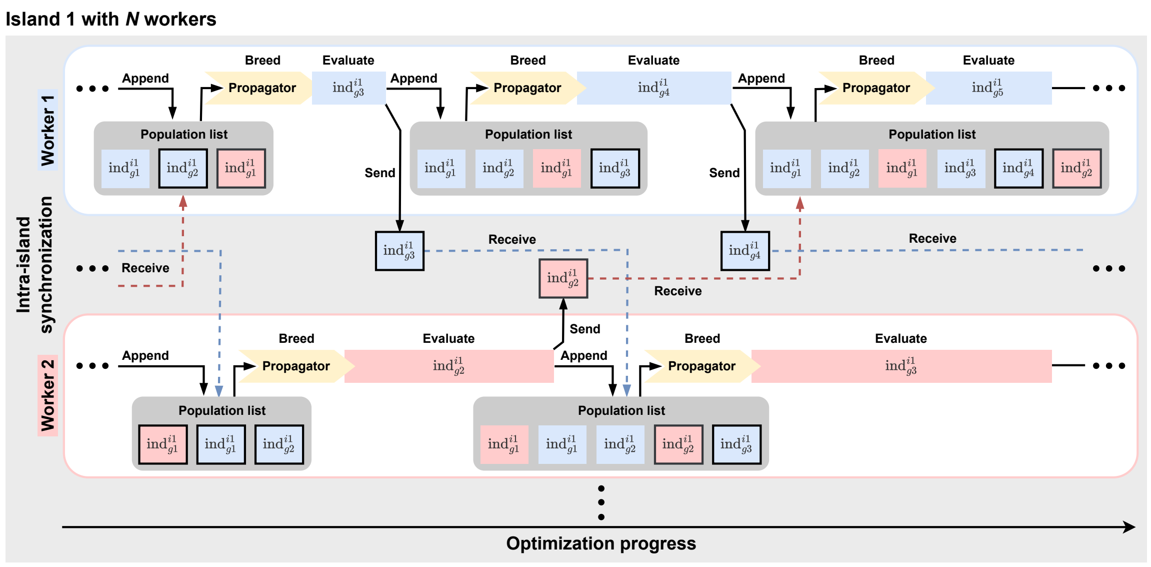


Figure 2.1.: PROPULATE figure from [36] showing the internal communication during the optimization process. PROPULATE is run with MPI. For every copy of PROPULATE started with MPI, a worker is created. Each worker maintains a local population list of evaluated HP configurations, stored as individuals (ind). Workers run for a specified number of generations. In each generation and when using the GA, they breed a new individual through selection, mutation, and crossovers. This new individual is evaluated and then shared with the other workers on the same island. Between these cycles, workers check for incoming individuals from other workers running in parallel and integrate them into their own population.

significantly faster (up to three orders of magnitude) without compromising solution accuracy, demonstrating the effectiveness of its asynchronous approach.

The core of PROPULATE's functionality lies in its novel parallel communication scheme, which is based on a fully asynchronous island model with independently processing workers. This allows for massive parallelism and efficient use of parallel hardware, minimizing idle times between evaluations. In Figure 2.1, the structure on a single island (usually equivalent to a node) with multiple workers is outlined. The algorithm leverages asynchronous propagation of continuous populations with interwoven worker-specific generations, avoiding explicit synchronization points. Workers on each island process and maintain populations, breeding and evaluating individuals and sharing results via MPI at the end of each evaluation for mutual population updates. Individuals from parallel workers are collected and integrated into the local population between evaluations. PROPULATE also implements asynchronous migration or pollination between islands, enhancing genetic diversity and algorithmic efficiency [36].

2.3. Learning Curve Prediction

Learning curve prediction is a technique in ML to forecast model performance over time, offering insights into future accuracy and efficiency. It can play a significant role in HPO by predicting the outcome of different HP configurations without fully training the model, thus saving computational resources and time.

Various methods for learning curve prediction include extrapolation of early performance metrics, Bayesian modeling, and the use of performance bounds. Each method aims to estimate the final performance of a model based on initial training epochs. These predictions allow for early stopping of unpromising training runs and focusing computational effort on configurations likely to yield better results.

Probabilistic modeling with Markov Chain Monte Carlo inference or BO from early training data offers a way to predict the full learning curve with high accuracy [7, 20].

A newer approach is using a prior-data fitted network, trained on common learning curve benchmarks, for learning curve extrapolation. This significantly speeds up the prediction process itself by only using a single forward inference, while still providing accurate extrapolations of learning curves [2].

2.3.1. Surrogate Models

SMs are simplified representations or approximations of complex real-world systems, used to reduce computational cost while retaining essential features of the represented system.

These models operate by emulating the behavior of a more complex model or system through a simpler, more computationally feasible framework. In optimization problems like NN training, SMs can be used to predict the outcome without the need for expensive and time-consuming full-scale models. They are particularly useful in situations where the number of possible simulations is vast, and running the actual model is impractical due to resource constraints.

SMs can be used in HPO to speed up the search for optimal ML model configurations. By approximating the performance of a given configuration, SMs can efficiently guide the search process towards areas that are likely to yield improved performance, thereby reducing the number of expensive evaluations of the actual model and speeding up the overall HPO process [15, 16].

2.3.2. Static Termination

This chapter introduces an approach known as static termination. It simplifies the early stopping of underperforming training runs by comparing their performance against predefined criteria, heuristics, or baselines, without the necessity for probabilistic forecasting. While being very effective, the traditional BO approach can demand significant computational resources and expertise in probabilistic modeling. Static termination emerges as a practical alternative, offering a straightforward and easier to implement method for early stopping.

The static early stopping criterion in Lakhmiri and Le Digabel [22] utilizes a baseline comparison approach. This involves comparing the performance of the currently evaluated network against a baseline set by the best configuration so far. The decision to stop early is based on whether the network's validation scores are within a predefined, problem-dependent error margin from the baseline.

Another approach established by Mahsereci et al. [25] eliminates the need for a validation set. Instead of relying on local statistics of computed gradients to decide when to halt the optimization process, the "Evidence-Based criterion" assesses if the gradient represents noise from the dataset's finiteness rather than conveying meaningful direction for optimization. The Evidence-Based criterion allows the optimizer to use all available training data.

Most approaches employ some kind of baseline comparison. However, there is no necessity to explicitly define said baseline. In Hyperband Li et al. [23], the HPO begins with a broad evaluation of numerous HPs with minimal resources before utilizing a successive halving algorithm to systematically narrowing down the pool based on performance metrics. Early stopping decisions are made adaptively, with lower-performing HP configurations halted early to reallocate resources to more promising candidates. This eliminates the need for a predefined error margin or baseline.

2.3.3. Termination with Bayesian Optimization

BO is a strategy for the global optimization of objective black-box functions that are expensive to evaluate. It is particularly useful in ML for HP tuning, where evaluations of the objective function, i.e., training and validating a model, are computationally costly and time-consuming.

The core idea of BO is to model the objective function $f(\boldsymbol{x})$ with a surrogate probabilistic model, typically a GP, and then iteratively select new points to evaluate by balancing exploration of the domain with the exploitation of current knowledge. The decision on where to sample next is guided by an acquisition function, which is derived from the posterior distribution of the surrogate.

A GP defines a distribution over functions, characterized by a mean function $m(\boldsymbol{x})$ and a covariance function $k(\boldsymbol{x}, \boldsymbol{x}')$, which together describe the relationships and variations in the data:

$$f(\boldsymbol{x}) \sim \mathcal{GP}\left(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}')\right) \tag{2.1}$$

The mean function $m(\boldsymbol{x})$ provides an estimate of the objective function's values and thus a baseline expectation for its values across its domain, essentially offering an initial guess before any observations are factored in:

$$m(\boldsymbol{x}) = \mathbb{E}\left[f(\boldsymbol{x})\right] \tag{2.2}$$

The covariance function $k(\boldsymbol{x}, \boldsymbol{x}')$, also called kernel, defines the similarity or "closeness" between points $\boldsymbol{x}$ and $\boldsymbol{x}'$ in a GP. The choice of kernel function encodes prior beliefs about the function being modeled and can drastically influence the GP's performance:

$$k\left(x, x'\right) = \mathbb{E}\left[\left(f\left(x\right) - m\left(x\right)\right)\left(f\left(x'\right) - m\left(x'\right)\right)\right] \tag{2.3}$$

Kernels can be combined in various ways to model different aspects of the data, such as periodicity, linearity, or smoothness, allowing for the construction of complex, composite kernels that can capture a wide range of behaviors in the data [30].

Prior work mostly uses Bayesian methods like Markov Chain Monte Carlo inference to extrapolate learning curves, which has yielded good results [7]. Building up on this, Bayesian neural networks have been used with even better results [21] on par with prior presented approaches like Hyperband in subsection 2.3.2.

Freeze-Thaw Bayesian Optimization [35] is the main work directly using BO for loss curve prediction. In this approach, two GPs are utilized to efficiently explore the HP space by predicting the performance of ML models without the need to fully train them. The first GP models the global trend of how model performance varies with HPs, capturing the overall behavior across the entire HP space. The second GP models the local training dynamics of individual models, predicting how the performance of a model will evolve as training progresses. The setup allows the optimization process to “freeze” models that are unlikely to perform well and “thaw” and continue training for those with promising early performances, thus focusing computational resources more effectively.

The local GP relies on a kernel designed to capture the training curves’ characteristics, assuming an exponential decay pattern towards an asymptotic loss value.

The division into local and global GPs is primarily driven by the need to manage computational complexity. GPs are computationally expensive, with a prediction complexity of $O\left(N^3\right)$ where N is the number of training observations. This computational demand increases quickly for HPO settings dealing with multiple HP configurations with many loss values each. If a naive approach were applied, where a GP model encompassed every observed training loss over time with T training iterations per setting, the computational complexity would be up to $O\left(N^3T^3\right)$, making it prohibitively expensive.

2.3.4. Ranking

In the context of HPO, ranking refers to the method of ordering or prioritizing HP configurations based on certain criteria or metrics and allocate computational resources to promising configurations. This can be extended to halt and eventually resume or completely abandon past configurations. By evaluating and comparing the performance or potential of each entity, ranking allows for the identification and selection of the most promising or effective options. Ranking can be used as standalone speedup technique or combined with other early stopping mechanisms to focus resources on exploring and exploiting the most beneficial configurations.

Ranking does not necessarily need to create a complete order. With the prior presented successive halving algorithm, it is sufficient to divide running configurations in two categories, i.e., halt and continue. This method is especially efficient with a limited computational budget, as it maximizes the utilization of available resources by ensuring only the most promising configurations receive extensive evaluation [23].

The “Freeze-Thaw Bayesian Optimization” paper introduces a ranking system based on BO to forecast the eventual performance of HP configurations from their early results. This system prioritizes configurations for further evaluation not merely on current performances but on their predicted future outcomes, allowing for an early stop of less promising configurations and the continuation of those with potential. The resumption occurs when, upon re-evaluation, previously halted configurations show promise based on updated predictions or when the optimization process necessitates a broader exploration of the HP space [35].

Another approach is to create a pool of HP configurations and rank them directly after creation before sequentially evaluating them from the highest to lowest rank. The process is halted as soon as a candidate exhibits superior performance compared to the current best, thereby ignoring the remaining candidates in the pool. This approach favors fast improving configurations. Coupled with early-stopping SMs, this strategy can significantly speed up HPO [22].

A disadvantage of ranking mechanisms is that they need to be integrated into the HPO framework directly as they require a synchronized view on all running instances, adding complex communication infrastructure, and – when future resuming should be realized – access to control the HPO framework’s candidate creation and evaluation. This means ranking is inherently incompatible with retrospectively added SMs without altering big parts of the original program. As this problem especially halts true for Propulate, ranking is not considered further in this work.

3. Approximative Surrogates

Chapter 2 outlined various approaches to HPO. The goal of this work is to bring and test some of these methods to and in PROPULATE. Three steps are needed to achieve this:

1. Introducing and implementing an early stopping mechanism to stop training in PROPULATE. This is explained thoroughly in section 3.1.
2. Testing the implementation and giving a reference for future SM implementation with two examples. Both are building on prior explained concepts and are explained in section 3.2 and section 3.3.
3. Evaluating the SMs in the next chapter 4.

3.1. Implementation

3.1.1. Goals

Apart from the pure functional details, there are three major goals for the implementation of SMs in PROPULATE.

Firstly, PROPULATE should remain usable without any SMs. Users should be able to run legacy code without any changes, requiring backwards compatibility. In addition, future projects should be feasible without any knowledge of SMs. As the current implementation of SMs depends on intermediate results of the loss function from each individual evaluation, some optimization problems without iterative evaluations are inherently incompatible with it and can not be speed up by SMs.

Secondly, the SMs should be as extensible as possible. The only limitation is the information given during training runs as the surrogate method calls can not be changed. However, users should be allowed to control every aspect within the SMs.

Lastly, the SMs should be easy to use. Users should be able to employ SMs without any knowledge of their inner workings and integrate them into their projects with minimal effort. The implementation of the models itself is exempt from this goal as simplicity can come into conflict with extensibility.

Further implementation details are discussed in the following sections and will refer back to these goals to show how they are achieved.

3.1.2. Propulate Implementation

To illustrate the implementation of SMs within PROPULATE, the subsequent explanation follows the path of a surrogate through the PROPULATE code.

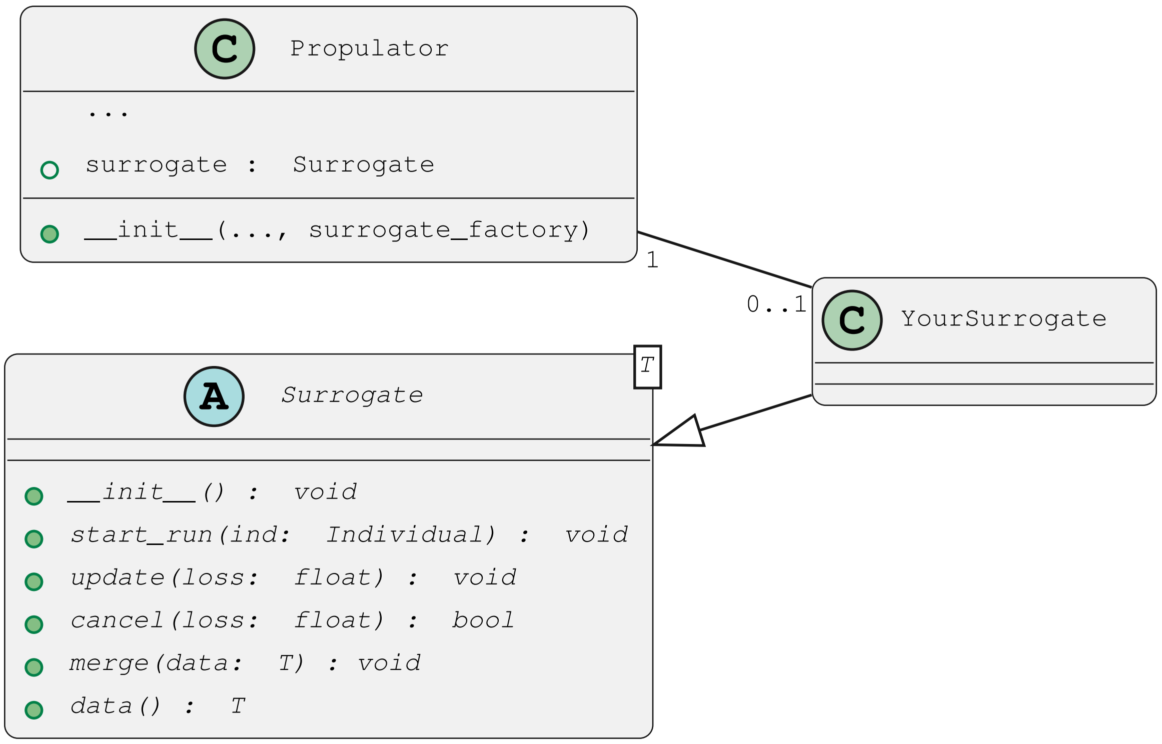

C Propulator
...
surrogate : Surrogate
__init__(..., surrogate_factory)
1
0..1
C YourSurrogate
A Surrogate
T
__init__() : void
start_run(ind: Individual) : void
update(loss: float) : void
cancel(loss: float) : bool
merge(data: T) : void
data() : T

Figure 3.1.: Inside the PROPULATE core functionality is only a fully abstract `Surrogate` class. Concrete subclasses have to be implemented by the user or externally provided, here depicted as "YourSurrogate". `Propulator`, the class that handles the processes' local population, creation of new individuals, and actual training can hold exactly one SM in a class variable. It gets initialized with the passed down surrogate factory as shown in Code snippet 2.

The first step is to create an SM itself. This is done by creating a class that inherits from the abstract Surrogate class and implements all abstract methods. Everything else inside the class is up to the user, allowing for maximum flexibility in future SMs and effectively ensuring the second goal.

SMs are structured as abstract classes comprising five essential methods. Initially, the method `start_run()` is invoked once immediately after breeding a new configuration, which is stored as an individual and passed as an argument. After completing the evaluation of this individual, the `update()` method is called to mark the conclusion of the training and convey the final loss value. Throughout the iterative training phase, the `cancel()` method is called at periodic intervals with intermediate loss values. This particular step is crucial as it determines the decision for early stopping based on the given losses. To synchronize SMs across the MPI processes, the methods `data()` and `merge()` are used to export and import all relevant surrogate data at any given time, enabling synchronization at any moment without predefined checkpoints.

A more detailed insight into two possible SMs and their implemented methods is given in section 3.2 and section 3.3.

The only limitation for SMs is that the actual calls to the SM can not be changed as they are defined within PROPULATE's core functionality. During `ind_loss()` runs, the SM thus has to work with only the prior given information plus the repeatedly yielded intermediate loss itself.

```
islands = Islands(  # Set up island model.
    ...
    surrogate_factory=lambda: YourSurrogate(args: your_args),
)
```

Code Snipped 1: A surrogate factory is given as an argument to create Islands during the PROPULATE configuration. This can be as simple as defining a lambda or passing the `__init__()` method. Additionally, possible surrogate specific arguments in the constructor are defined here.

PROPULATE is usually run with MPI so it has multiple processes running in parallel. Each process maintains its own population and runs independently from the others. To keep the resulting asynchronous nature of PROPULATE, each process needs to create its own instance of the SM. Accordingly, the first step is to create a factory function that returns a new instance of the SM to be passed to the Islands class constructor as illustrated in code Code snippet 1. The Islands class is instantiated by the user and the main entry point for PROPULATE with its `evolve()` method. Because the factory function is created outside of PROPULATE and the only place where the SM constructor is defined, the user can override the default surrogate constructor with additional arguments. From the Islands class, the surrogate factory is then passed down to the Propulator class, which handles the population, generating new individuals and evaluating them. There, a new SM is created as shown in Code snippet 2.

```
class Propulator:
    def __init__(
        ...
        surrogate_factory: Callable[[], Surrogate] = None,
    ) -> None:
        ...
        self.surrogate = None if surrogate_factory is None \
            else surrogate_factory()
```

Code Snipped 2: The surrogate factory is called within Propulator's `__init__()` at the start of PROPULATE. Each Propulator (i.e., each process) has its own instance of an SM.

When `_evaluate_individual()` is called, Propulator checks whether the loss function is a generator (see Code snippet 3). This preserves PROPULATE's original functionality, allowing the loss function to be implemented as a conventional function, which is the main reason for achieving the first goal. Only if the loss function is a generator and an SM is given, its `cancel()` method is called with the new interim loss value. When the model decides to cancel the run, the loss function is stopped and the last yielded loss value is returned as the final result. Lastly, `update()` is called to signal the SM that the run has finished (see Figure 3.2).

```python
class Propulator:
    ...

    def _evaluate_individual(self) -> None:
        ...

        if inspect.isgeneratorfunction(self.loss_fn):
            last: float = 0.0
            for last in self.loss_fn(ind):
                if self.surrogate is not None:
                    if self.surrogate.cancel(last):
                        break
            ind.loss = last
        else:
            ind.loss = self.loss_fn(ind)
```

Code Snipped 3: If the loss function is a generator and a local SM exists, its `cancel()` method is called any time the generator yields a new loss value. The else block evaluates the loss function exactly like before, making it possible to not use any SMs.

Once the evaluation is complete, the final loss is communicated back to the surrogate, where it can be incorporated into its inner model. It is important to note that each process operates with its own surrogate. Despite this individualized approach, there is still a necessity to synchronize the processes. Otherwise, different surrogates would find the same best configuration multiple times. To streamline this synchronization without introducing error-prone and challenging-to-test additional communication code, the surrogate is simply attached to the Individual. The Individual inherits from Dictionary, making it easy to append the data here. This Individual, along with its attached surrogate data, is then sent to the other PROPULATE processes via the already implemented MPI communication as depicted in the last loop in Figure 3.2.

The receiving side of this process is shown in Figure 3.3. The existing code to check for incoming individuals from other processes is again just extended. If and only if surrogate data is stored with the incoming individual, the new data is merged into the receiving processes' own SM.

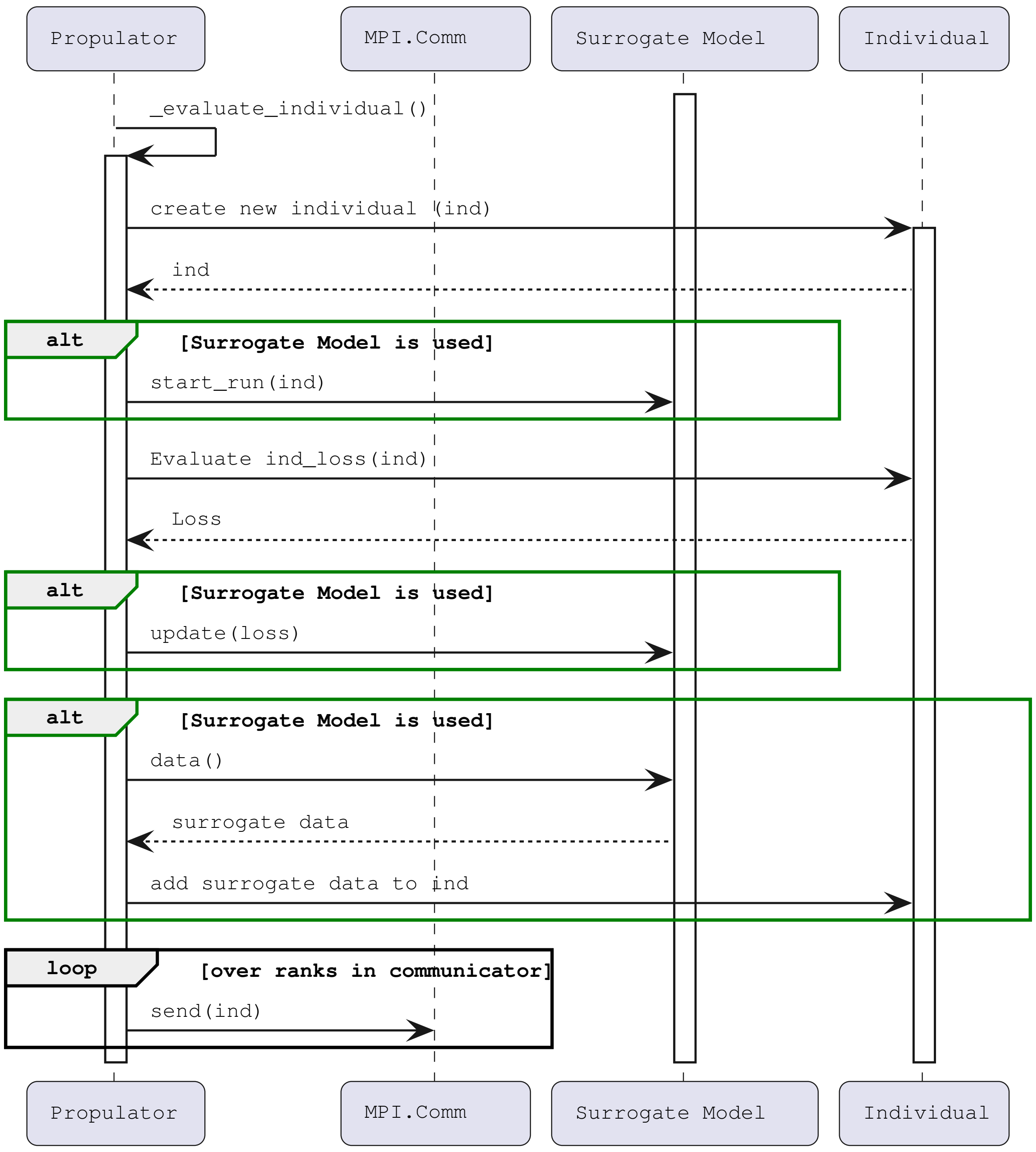


Figure 3.2.: Sequence diagram for all important calls during the `_evaluate_individual()` execution. Each green box represents new code introduced along with the SMs and conditionally called when the latter are used. Details of the "Evaluate `ind_loss()`" step are shown in Code snippet 3. In the end, the SM is appended to the newly created and evaluated Individual that is then sent to the other processes.

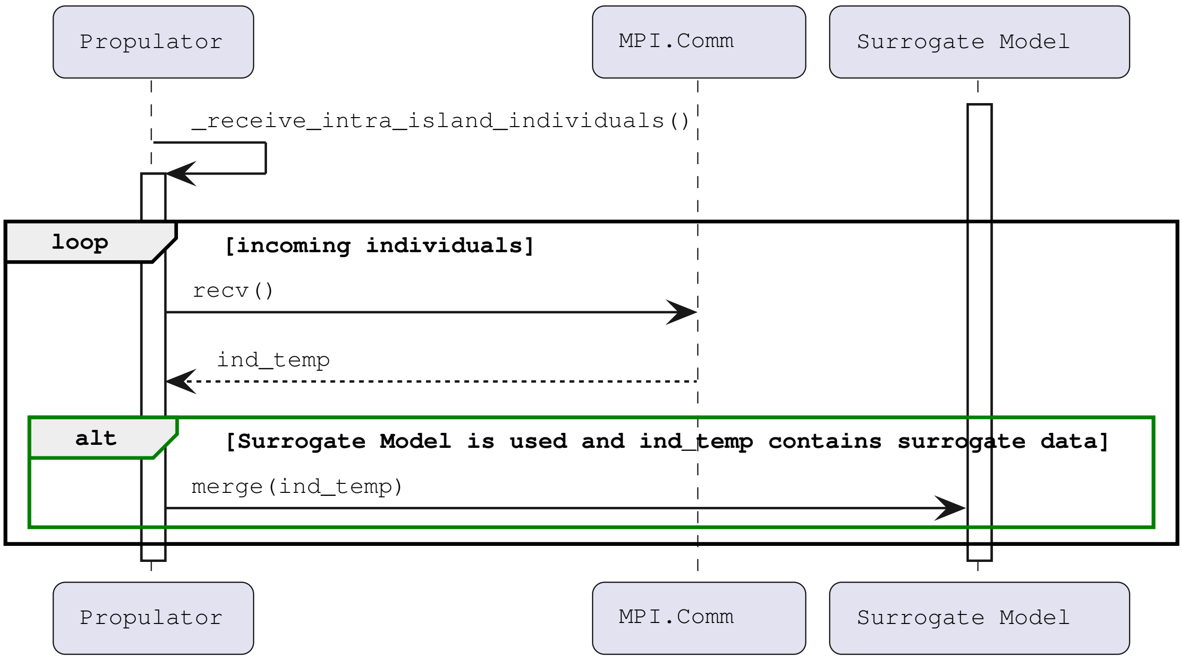


Figure 3.3.: Sequence diagram for the receiving side of sending individuals to the other Propulators on the same island. Shown is the first part of the `_receive_intra_island_individuals()` method. The green box is the added code that checks if a local SM exists and the incoming individual has surrogate data attached to it. If this is the case, `merge()` is called to synchronize the local SM.

3.1.3. User-Side Implementation

At the core of PROPULATE's user-defined functionality is a loss function that handles creating the NN with the given HPs and running the training plus validation for the predefined number of epochs. Thus, users have fine-grained control over what happens during training. The newly added SMs interfere exactly here. The loss function can now be implemented either as conventional function returning the final average validation loss or as generator yielding interim loss values. This adaptability allows PROPULATE to behave unchanged in absence of a surrogate while at the same time giving the flexibility to cater to as many potential surrogate models as possible.

Only when the loss function is implemented as a generator, PROPULATE activates its SM mechanism. As the only value yielded is the loss, SMs have to autonomously generate unique identifiers for each yielded value if a running index is needed. This means it is crucial to maintain regularity in the yield points within the loss function. Irregular yielding can disrupt the indexing process of some SMs, potentially leading to inaccurate predictions (see 3).

A simple example of this is shown in Code snippet 4. The average validation loss is yielded once per epoch. As this quantity is usually calculated anyway, this approach would require the user to change only one line of code. It should be noted that depending on the number of epochs in the training process, this could be too few or too many yields and the frequency of yield should always be adjusted accordingly.

```python
def ind_loss(
        params: Dict[str, Union[int, float, str]],
) -> Generator[float, None, None]:
    # ...

    for epoch in range(epochs):
        # ...

        # Validation loop
        # ...
        for batch_idx, (data, target) in enumerate(val_loader):
            ...

        avg_val_loss = total_val_loss / len(val_loader)
        yield avg_val_loss
```

Code Snipped 4: Example illustrating where to put a possible yield in `ind_loss()` during the training process. At the end of every epoch, the average validation loss will be yielded.

The SMs require careful handling due to their potential computational intensity. Excessive yielding, especially with complex models, can lead to significant processing demands. To steer the creation of new HPs even more towards the already successful configurations, the last yielded loss value is considered the definitive result of the run even if the process is terminated prematurely.

```
    limits = {
        "conv_layers": (2, 10),
        "activation": ("relu", "sigmoid", "tanh"),
        "lr": (0.01, 0.0001),
    }
    propagator = get_default_propagator(
        # ...
        limits=limits,
        # ...
    )
    islands = Islands(
        loss_fn=ind_loss,
+       surrogate_factory=lambda: DefaultSurrogate(),
        # ...
    )
    islands.evolve(
        # ...
    )
```

```
    def ind_loss(
        params: Dict[str, Union[int, float, str]],
+   ) -> Generator[float, None, None]:
```

Code Snipped 5: User-side Propulate surrogate configuration for the exemplary use case presented in subsection 4.2.1. Lines differing from usage without SM are highlighted in green.

As clearly illustrated in Code snippet 5, the adjustment required to configure Propulate with SMs on the user side is minimal. Specifically, after incorporating the yield into the training process, only a single line needs to be modified so as to add the surrogate factory as an argument. This streamlined approach ensures that users can easily integrate and leverage the functionality of SMs without the need for extensive modifications to their existing setup.

3.2. Static approach

The first SM implementation provided is based on the 2022 publication “Use of Static Surrogates in Hyperparameter Optimization” by Lakhmiri et al. [22]. It introduces two mechanisms to speed up HPO, i.e., firstly the early stopping of poorly performing candidates and secondly a ranking strategy to allocate more computational resources to well performing HPs.

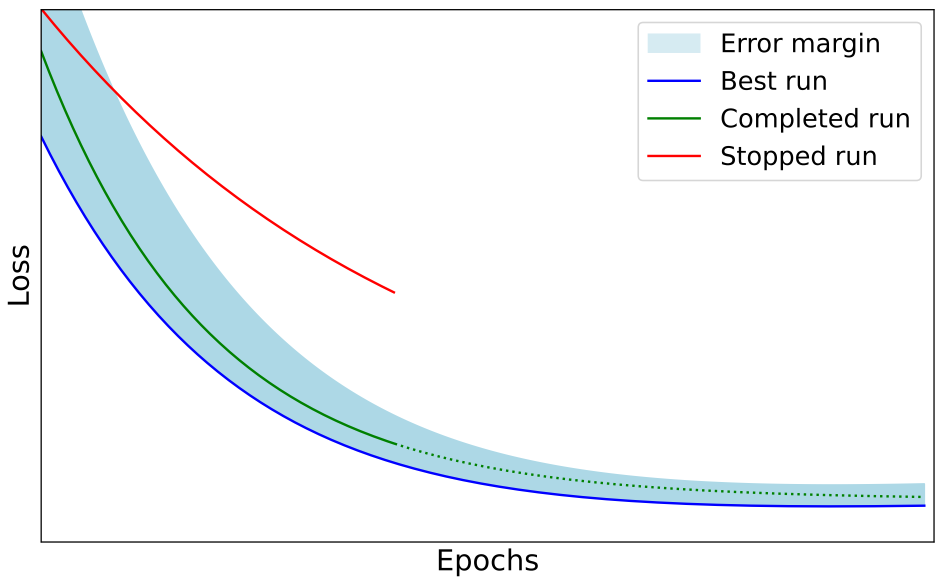


Figure 3.4.: Static Surrogate explanatory graphic. When the SM gets a new loss value from an active training run, it compares the value to an existing baseline, called the best run (in blue). Around the best run is a relative dynamic error margin. When a new loss lies above the error margin, like the end of the red line, the training run is stopped. When it is inside, like the solid part of the green line, the training run continues.

The implementation for Propulate only considers the early stopping mechanism as ranking would need a deeper integration within Propulate to not only stop but also start new training runs with specific HPs.

In the outlined method, the early stopping decision is based on the comparison of a candidate's performance against a predefined baseline, which represents the best performance observed so far. This candidate is adjusted with a dynamic error margin, allowing for some flexibility in performance variance. The training of networks that do not meet this criterion is stopped early, ensuring only candidates with a realistic chance of surpassing the baseline continue to consume resources.

This functionality can be fitted into the abstract Surrogate class. Firstly, the surrogate needs to gather a baseline, making the first training run after starting Propulate always run through completely. During the second and all following training runs, the static surrogate has to keep track of the current training run's loss series. Should it turn out to perform better than the baseline in terms of the final training loss, the baseline is replaced with the current training run's loss series. The `start_run()` call in the beginning resets the current loss series. During the training run, `cancel()` continuously stores and checks the newest loss value against the loss in the baseline at the exact same point and cancels according to the prior condition.

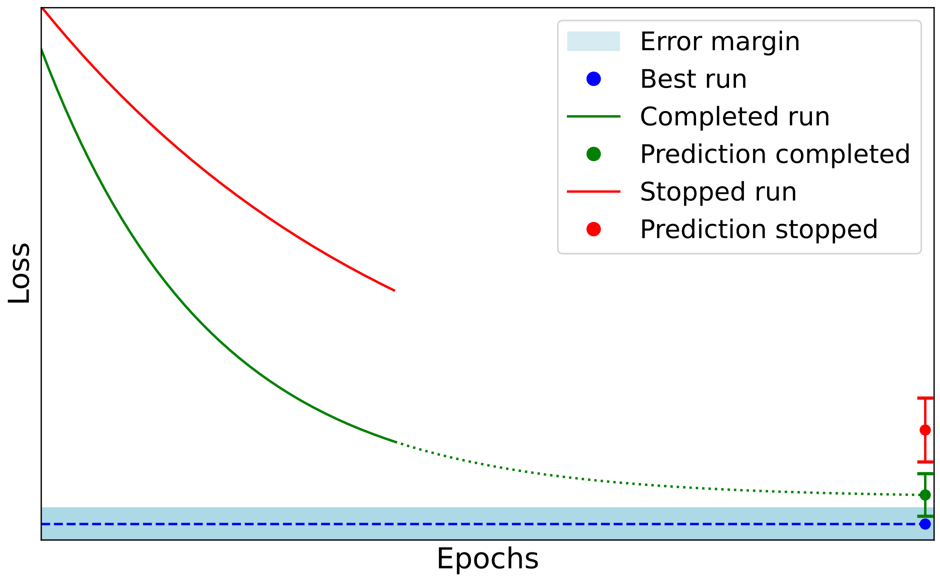


Figure 3.5.: Dynamic Surrogate explanatory graphic. When the SM gets a new loss value from an active training run, it feeds this value into its local GP and predicts the final loss. This loss minus the variance of the results is compared to an existing baseline, called the best run (in blue). Around the best run is an error margin. When a new predicted loss minus its variance lies above the error margin, like the red dot corresponding to the predicted loss of the red line, the training run is stopped. When it is inside, like the variance bars of the green dot that overlap with the error margin, the training run continues.

3.3. Probabilistic approach

The second SM implementation is inspired by the early stopping mechanism presented in the 2014 publication “Freeze-Thaw Bayesian Optimization” by Swersky et al. [35].

The implementation of the dynamic surrogate differs from Freeze-Thaw Bayesian Optimization as the exact details are complicated and no usable implementation is provided by the authors. While keeping the general structure of two separate GPs, the implementation for the dynamic surrogate mainly diverges by choosing slightly different kernels for the global and local GP. The provided dynamic surrogate uses a combination of a Matérn 3/2 kernel with a white noise kernel for the global GP and a combination of a Matérn 5/2 kernel with a white noise kernel for the local GP. It keeps the prior mean for the local GP drawn from the global GPs prediction for the new model. Additionally, an error margin for the prediction of the local GP is introduced.

The general implementation is quite similar to the prior static surrogate in section 3.2. Over the complete run time of Propulate, the best final loss of any training run is kept. As there is no prior loss to draw from for a prediction during the first generation, `cancel()` effectively only collects loss values and never stops until the second generation. The loss margin is multiplied with the final (predicted) loss to decide on stopping a run. The implementation for the dynamic surrogate differs in the actual stopping algorithm.

The global GP is created in `__init__()` of the dynamic surrogate. When calling `start_run()`, the global GP makes a prediction for a loss with the new HPs. The prediction is then used as a constant prior mean for the local GP. During each `cancel()` call, the local GP is optimized with the new loss value. Afterwards, it is used to make a prediction for the final loss of the currently running configuration. A decision to stop is made based on whether the predicted loss minus variance overlaps with the best final loss observed so far. This behavior is depicted in Figure 3.5. Here, the red loss curve represents a stopped training run as its predicted loss is far from the best run. The green curve corresponds to a to be completed run as the final loss prediction minus variance is well within the error margin of the best run.

4. Evaluation and Discussion

This chapter explores how the prior presented SMs perform. They are benchmarked with different NNs (section 4.2) and datasets (subsection 4.1.2) under various metrics such as run time, best found loss, hardware utilization and energy consumption (section 4.3) using Propulate.

4.1. Environment and Setup

4.1.1. Computational Environment

The experimental evaluation was conducted on virtual servers provisioned from IBM Cloud, specifically the gx2-16x128x2v100 virtual server instance. This instance provides 16 virtual CPUs, 128 GiB of RAM, and two NVIDIA V100 GPUs with 16 GB of memory each.

	Version
Operating System	Ubuntu 22.04.3 LTS
Kernel	5.15.0-1046-ibm
Python	3.10.12
MPI	OpenMPI 4.1.2
PyTorch	2.2.0+cu121
NumPy	1.26.4
NVIDIA Driver	535.154.05
CUDA	12.2

Table 4.1.: Software versions of all important components used. The other specific Python package versions can be obtained from the *requirements.txt* in the GitHub repository under the *ibmcloud* branch (Appendix A).

4.1.2. Datasets

The MNIST [28] dataset is a collection of handwritten digits. It consists of 60,000 training images and 10,000 testing images, each a 28×28 pixel grayscale image of a single digit. The CIFAR-10 [26] dataset is a set of 60,000 32×32 photo-realistic color images in 10 classes, with 5,000 training and 1,000 testing images per class. Both MNIST and CIFAR-10 belong to the most popular and widely used datasets for benchmarking and developing various types of neural network architectures and learning techniques [27].

4.1.3. Seed Variation

Propulate runs are inherently not reproducible, i.e. the underlying communication infrastructure of HPC can behave non-deterministically. Nevertheless, to make runs of Propulate more reproducible, a seeding function is called before initializing Propulate. This function takes a seed as an input argument and standardizes the initialization across Python, Numpy, and PyTorch environments to ensure the same initial conditions across all runs.

Propulate is called with MPI and usually runs multiple processes on a single node. Initializing each process with the same seed would result in the exact same result for every process, nullifying any advantage gained through the parallel execution. To account for this, each seed is multiplied by the current MPI rank, making seed values passed to the seeding function different on each process while still remaining reproducible.

Multiple seeds allow to mitigate the impact of "lucky" seeds that could bias results [17, 14]. Averaging outcomes across these seeds can neutralize randomness and provides a more accurate assessment of the SMs' impact on optimization efficiency.

Specifically, Propulate runs are conducted with three distinct seed values of 42, 271, and 3141. If not stated otherwise, further results in this chapter are always averaged over these seeds.

4.1.4. Metrics Logged

4.1.4.1. Logging Training Results

Training results are directly written to a comma-separated values (CSV) file from the surrogates. For this, a new method `_log_to_csv(...)` is introduced to each SM. This method writes all relevant fields into a CSV and is called at the beginning of `cancel()`. Additionally, new surrogate class variables are introduced to keep track of the dataset, rank, epoch, individual, and the own surrogate name. To avoid write conflicts, each process (rank) logs into its own file. This approach intentionally leaves the code inside Propulate itself unchanged as modifying Propulate code would be more error-prone and SMs have a more narrowly confined environment. The final results from all 18 runs are collected into a single CSV as described in Table 4.2.

The field "avg_validation_loss" has to be calculated at the end of each epoch as:

$$\text{avg\_validation\_loss} = \frac{\text{total\_validation\_loss}}{\text{len(validation\_loader)}} \tag{4.1}$$

4.1.4.2. Logging System Metrics

System metrics like the total run time and estimated CO_2e are collected with Perun. Perun is a tool specifically designed to efficiently gather usage statistics from CPU, GPU, memory, and other hardware components, which can then be used to calculate energy cost and CO_2e from current local prices and carbon intensity [11].

Table 4.2.: Collected training metrics for each PROPULATE run via direct writes to a CSV. Unambiguous identification of a result works through the combination of rank, surrogate, generation, epoch, dataset, and seed. All results are merged in a single CSV which is why time cannot be used as an index. Note that HP configurations can also appear multiple times and are not interchangeable with generation.

Header	**Type**	**Description**
time	float	Local unix time
rank	int	MPI rank
surrogate	string	“default”, “static”, or “dynamic”
generation	int	PROPULATE generation
epoch	int	Epoch
paramsID	string	Hyperparamter configuration
avg_validation_loss	float	Average validation loss
dataset	string	“mnist” or “cifar”
seed	int	42, 271 or 3141

Results are given in a short written, human-readable evaluation as plain text file and a detailed evaluation with all measured data points in a md5 or json format. All relevant data to evaluate the SMs can be found in the provided text file, meaning that the detailed evaluation is not used. The results are collected via a regular expression and then merged into a single CSV as specified in Table 4.3.

4.2. Neural Networks

The following two NNs are implemented in PyTorch and use its default build in loss functions, optimizers, and activation functions.

4.2.1. MNIST Classification

A CNN is used for the MNIST dataset classification. To enable a direct comparison between a version with and without SMs, the network architecture is the same as in the PROPULATE GitHub repository tutorial `torch_example.py` (see section A).

The network initiates with an input layer designed to process 28×28 pixel grayscale images, matching the MNIST dataset's single input channel. The convolutional layers are configurable with the number of layers ranging from two to ten (Table 4.5). Each convolutional layer has a kernel size of 3×3 and is followed by an activation function, selectable from ReLU, Sigmoid, or Tanh (Table 4.5). Following the convolutional layers, the network has a fully connected layer that maps the flattened convolutional features to ten output nodes, corresponding to the ten classes of the MNIST digits (0 through 9). The last HP is the learning rate which varies between 0.01 and 0.0001 (Table 4.5).

Table 4.3.: Collected system metrics and estimated energy / CO_2eq for each Propulate run via Perun. Unambiguous identification of a result works through the combination of dataset, surrogate, and seed. The server hostname is not an identifying attribute.

Header	**Type**	**Description**
dataset	string	"mnist" or "cifar"
surrogate	string	"default", "static", or "dynamic"
seed	string	"s42", "s271", or "s3141"
server	string	Server hostname
run time	float	Total run time in seconds
energy	float	Total energy in MJ
power	float	Average power consumption in watt
cpu_util	float	Average CPU utilization in %
gpu_power	float	Average GPU power consumption in watt
gpu_mem	float	Average GPU memory utilization
mem_util	float	Average memory utilization
kwh	float	Estimated kWh used
kgco2e	float	Estimated CO_2e equivalent in kg
price	float	Estimated price of electricity

Stochastic gradient descent is used as optimizer. The training runs with a batch size of 32 and through ten epochs in total. The loss function is cross entropy loss (Table 4.4). The seed is set externally as described in section 4.1.3.

Table 4.4.: Training parameters for MNIST classification with small CNN

Parameter	**Value**
Batch Size	32
Epochs	10
Optimizer	SGD
Loss function	CrossEntropyLoss

Table 4.5.: Hyperparameters to optimize for MNIST classification with small CNN

Hyperparameter	**Range**
convolution_layers	2 to 10
activation	relu, sigmoid, and tanh
learning_rate	0.01 to 0.0001

4.2.2. CIFAR-10 classification

For the CIFAR-10 classification, a ResNet [13] is employed. The ResNet implementation for this use case is based on [1]. It has a nine-layer structure as described in [12] and thus will be called ResNet9 in the following.

The network starts with an initial convolutional block, followed by a sequence of increasing channel-size convolutional blocks, some of which include pooling to reduce spatial dimensions. It has residual connections in two sections, after the second and fourth convolutional block. The model concludes with a classifier that consists of a max-pooling layer and a fully connected layer to output predictions across the CIFAR-10 classes. Gradient clipping is used during training.

HPO runs over the learning rate from 0.01 to 0.0001 and a permutation of the layers in the convolution block (Table 4.7). The permutation reorders the Conv2d, BatchNorm2d, and ReLU layers according to the permutation of the current configuration as shown in Code snippet 6. The permutation is given as an enum with six possible values “ABC”, “ACB”, “BAC”, “BCA”, “CAB”, and “CBA” where each letter stands for one of the layers. “ABC” is the original sequence of the layers.

Shuffling is done intentionally to worsen the network’s performance for some configurations. As no configuration performs particularly bad (Figure 4.5), this change results in a wider spread between loss curves of different configurations. This is expected to facilitate the analysis of the behavior of SMs.

```
layers = shuffle_array(
    [nn.Conv2d(in_channels, out_channels,
              kernel_size=3, padding=1),
        nn.BatchNorm2d(batch_norm_in),
        nn.ReLU(inplace=True)], perm)
```

Code Snipped 6: Shuffling within a convolutional block in the ResNet9. The parameter perm refers to one permutation as described in Table 4.7. The function `shuffle_array()` rearranges the given array according to this permutation.

The Adam optimizer [19] with weight decay is used. Training uses a batch size of 32 and runs through 10 epochs in total. The loss function is cross entropy loss (Table 4.6).

Table 4.6.: Training parameters for CIFAR classification with ResNet9

Parameter	**Value**
Batch size	32
Epochs	10
Optimizer	Adam
Loss function	CrossEntropyLoss

Table 4.7.: Hyperparameters to optimize in CIFAR classification with ResNet9

Hyperparameter	**Range**
Permutation	abc, acb, bac, bca, cab, cba
Learning rate	0.01 to 0.0001

4.2.3. Naming

As MNIST and CIFAR-10 classification is implemented with only one network each, the names are used interchangeably. In the following results and figures, classification with the CNN on MNIST as described in subsection 4.2.1 is referred to as **MNIST**, while classification with the ResNet9 on CIFAR-10 as described in subsection 4.2.2 is referred to as **CIFAR**.

4.3. Surrogate Comparison

4.3.1. Default Surrogate

The overhead introduced by using SMs in Propulate might slow down parts of the program. Comparing the two SMs introduced in chapter 3 against an implementation without any SM bears the risk of skewing run time measurements and the calculated speedup. That is why a third default SM is introduced as a baseline. This default SM implements all methods of the abstract Surrogate class but does nothing. Its `cancel()` method always returns False so it will never stop a training.

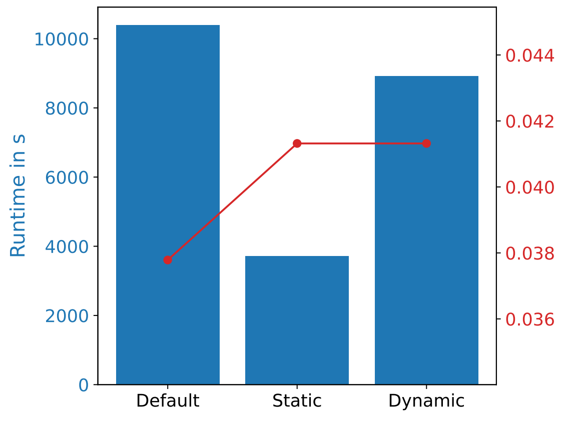
SMs are referred to as **Default** for the default SM or DefaultSurrogate, **Static** (as described in section 3.2) and **Dynamic** (as described in section 3.3) in the following.

4.3.2. Run time Comparison

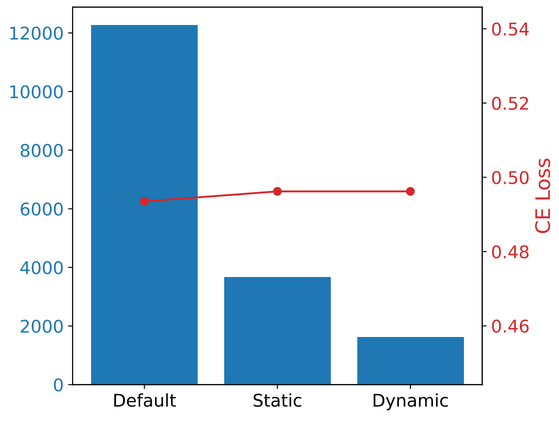
The evaluation of the proposed SMs starts with an analysis of the total run time of Propulate across the datasets and SMs. This initial step aims to discern efficiency improvements provided by SMs when applied to different datasets. No or statistically insignificant speedups would require fine-tuning the SMs, as further analyses would not provide any interesting insight compared to the default. The other comparative metric captured is the best loss found for each SM. Large discrepancies between the best found loss possible and the actually found best loss would make the presented SMs unsuitable for further use outside this evaluation.

As shown in Figure 4.1, a significant reduction in run time was observed across all surrogates and datasets. To quantify these improvements, the speedup achieved by employing an SM is calculated with respect to the default SM:

$$\text{Speedup} = \frac{\text{Run time of default}}{\text{Run time of SM}} \tag{4.2}$$



(a) MNIST



(b) CIFAR

Figure 4.1.: Total PROPULATE run time vs. the best loss for each SM. Lower is better for both.

A speedup of 2.8 was achieved for the MNIST dataset using a static surrogate, and a speedup of 3.3 for the CIFAR dataset under the same surrogate conditions. Conversely, the dynamic SM achieved a speedup of 1.1 for the MNIST dataset and a speedup of 7.6 for the CIFAR dataset, highlighting a pronounced variance in efficiency gains, particularly with dynamic surrogates. Between the seeds, the difference in run time varies at most by a factor of 1.6 between the fastest and slowest PROPULATE run for the static surrogate on MNIST and at the very least by a factor of 1.01 between the fastest and slowest PROPULATE run for the dynamic surrogate on CIFAR. This is also apparent when looking at Figure 4.2.

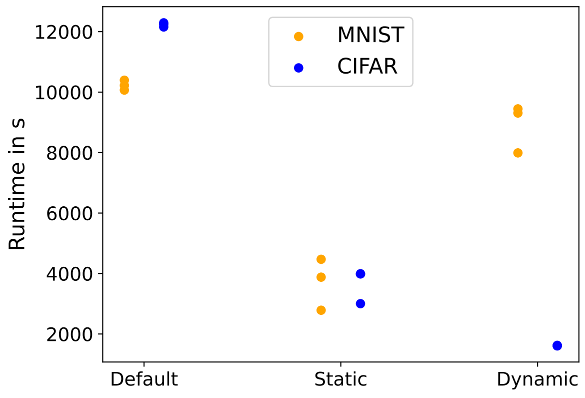


Figure 4.2.: Total run time for every PROPULATE run by dataset, surrogate, and seed.

The speedup metrics reveal a relatively consistent performance for the static SM across datasets, whereas a notable disparity is observed with dynamic surrogates, underscoring the impact of SM selection on computational efficiency. Across both MNIST and CIFAR datasets, the SMs identify the same best loss values. However, these are not the optimal loss values that could potentially be achieved without the use of surrogates with the same PROPULATE parameters.

Surrogate	Seed	Loss
	42	0.037786
Default	271	0.041425
	3141	0.041318
	42	0.042974
Static	271	0.045274
	3141	0.041318
	42	0.041796
Dynamic	271	0.043200
	3141	0.041318

(a) MNIST

Surrogate	Seed	Loss
	42	0.501943
Default	271	0.493513
	3141	0.465373
	42	0.507385
Static	271	0.496207
	3141	0.465373
	42	0.507385
Dynamic	271	0.496207
	3141	0.465373

(b) CIFAR

Table 4.8.: Lowest loss for every PROPULATE run, split by seed, grouped by dataset and SM. The row with the lowest loss for each dataset-surrogate combination is highlighted.

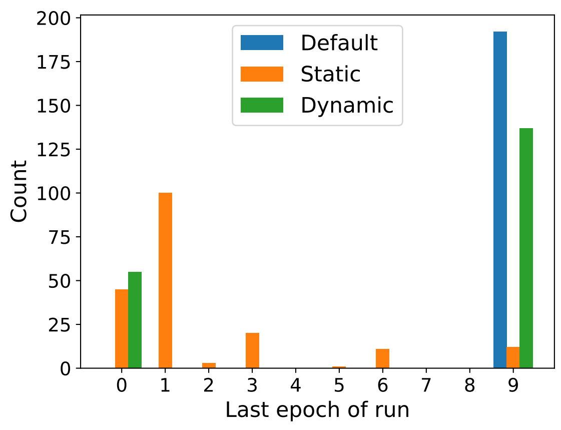
This behavior does not differ when looking at the different seeds on their own. The shown discrepancy reveals a limitation in the SMs' ability to identify the absolute best loss consistently, with the actually found loss being 8.5% and 0.05 % higher for MNIST and CIFAR, respectively. Despite this, the total numerical difference for MNIST is minimal, suggesting that the found loss was already approaching the lower bounds of the optimization potential.

Considering the minimum loss found by every seed individually in Table 4.8, this becomes clearer: The best loss found for CIFAR is the same across all SMs for the seed 3141. That mean that for some seeds the best loss is found, but not for all. SMs introduce a slight increase in variability for the performance over different seeds.

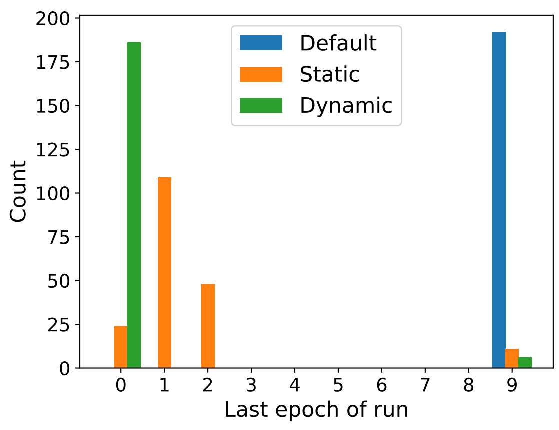
4.3.3. Stopping Behavior

A closer examination of the SMs reveals their varying effectiveness across different scenarios. Firstly, it is crucial to determine the exact point when each SM stops. This is illustrated in Figure 4.3, which shows the final epoch of each generation, delineated by dataset and SM type. When an SM terminates a run at, e.g., epoch 2, this is counted towards the epoch 2 tally but not for epochs 0 or 1.

By design, the default surrogate does not halt prematurely, allowing all training runs to conclude at epoch 9. The static SM exhibits consistent behavior across datasets, predominantly halting after two epochs (mode at epoch 1), with few exceptions before or after this point and rarely allowing runs to complete. The dynamic SM, acting more aggressively in its approach, stops either immediately at the first epoch or not at all. This aggressive stopping behavior is especially noticeable for the CIFAR dataset, where almost all runs are terminated early, in stark contrast to the MNIST dataset,



(a) MNIST



(b) CIFAR

Figure 4.3.: Last epoch of every generation by dataset and SM.

which sees fewer stopped runs and consequently a longer run time as evident from Figure 4.1.

Further analysis focused on the instances where training was halted and identifying the runs that were allowed to complete. Figures 4.4 and 4.5 provide insight into all observed loss curves, highlighting the top-ten runs from the default surrogate as a benchmark. These runs are marked in green across all SMs, though their performance varies with each model. The interruption of a run leads to different loss values being given to PROPULATE compared to those from completed runs, causing a divergence in the generation of new HP configurations from the default surrogate. Consequently, not every green marked run of the default surrogate appears for the other two SMs.

The MNIST classification reveals a categorization of loss curves into four distinct groups, i.e., those with consistently high loss, those that start high but improve over time, those that start low but do not improve further, and those that start low and keep getting lower. While it would be the most beneficial to let the second and last group run completely, both SMs effectively halt the first two groups early but struggle to identify runs that start low and do not improve over time.

The CIFAR classification presents a different scenario, with a clear division between the best performing curves and the rest, accompanied by numerous outliers not fully depicted in Figure 4.5. Here, the dynamic surrogate's aggressive stopping policy contrasts with the static surrogate's more uniform behavior, which tends to stop runs after two epochs.

Lastly, the median loss values at the point of interruption offer concrete insights. Figures 4.6 and 4.7 compare these values against the best performing run for each surrogate. Here, only the loss at the epoch of termination of each training run contributes to the median for that epoch. The static surrogate's median losses trace the best run's curve from a distance, indicating an error margin that narrows as the median loss approaches the best run's performance. Interestingly, for MNIST at epoch 3, the median stopped loss dips below that of the best run, attributed to an earlier/parallel run with lower loss. The dynamic surrogate, in contrast, exhibits

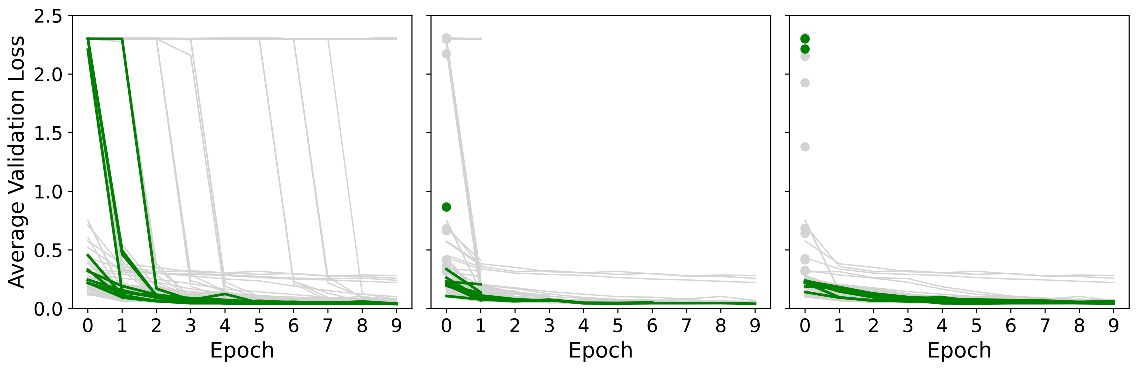


Figure 4.4.: MNIST loss curves. Left default surrogate, middle static surrogate, and right dynamic surrogate. The ten lines with the best final average validation loss for the default surrogate are colored green. For the two other surrogates, corresponding lines with the same generation, rank, and seed are colored green as well. Because Propulate's HP configuration generation diverges, these do not necessarily look the same. Loss curves that only have one value, i.e., training runs stopped after epoch 0, are shown as points, all others are shown as lines.

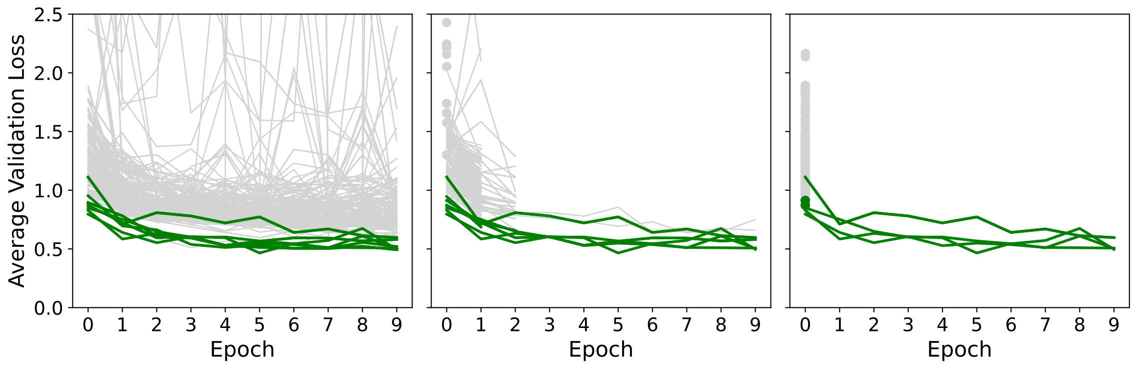
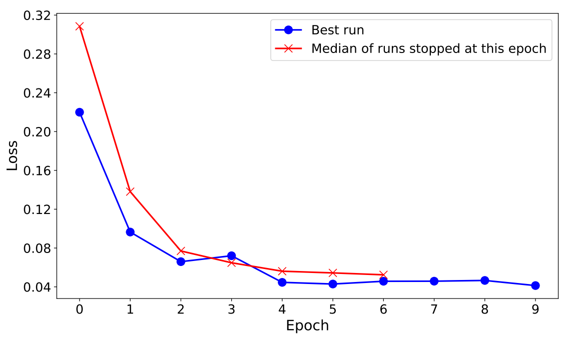
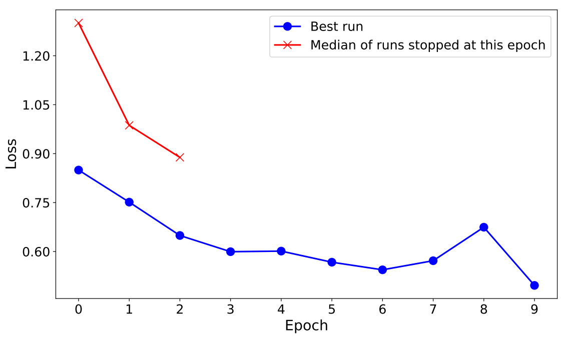


Figure 4.5.: CIFAR loss curves. Left default surrogate, middle static surrogate, and right dynamic surrogate. The ten lines with the best final average validation loss for default surrogate are colored green. Lines with the same generation, rank and seed are colored green in the other two surrogates. Because Propulate's HP configuration generation diverges, these do not necessarily look the same. Loss curves that only have one value, i.e., training runs stopped after epoch 0, are shown as points, all others are shown as lines.

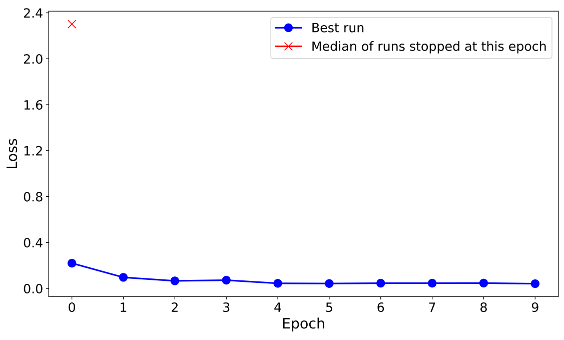


(a) MNIST

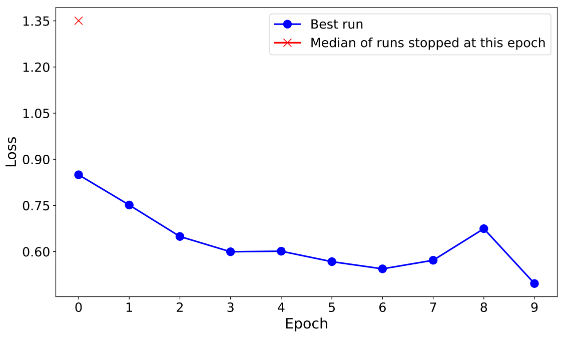


(b) CIFAR

Figure 4.6.: Median loss at the last epoch over all early stopped generations for the static surrogate by dataset. As a reference, the best generation with the lowest final loss is shown in blue.



(a) MNIST



(b) CIFAR

Figure 4.7.: Median loss at the last epoch over all early stopped generation for the dynamic surrogate by dataset. As a reference, the best generation with the lowest final loss is shown in blue.

a more aggressive approach, stopping only runs with very high values for MNIST. In contrast, the median loss at epoch 0 in CIFAR closely matches that of the static surrogate but without any stops beyond that point.

This analysis showcased the significant impact that SMs can have on the handling of different loss curve distributions, influenced by the distinct characteristics of the datasets. Note that subsection 4.3.2 did not deal with potential differences in outcomes between the SMs, as they ultimately identified the same optimal loss. The presented stopping behavior analysis revealed the nuances of their stopping mechanisms amidst varying loss curve distributions.

4.3.4. Hardware Utilization

The next section investigates the hardware utilization efficiency with SMs, particularly focusing on whether these models introduce a significant overhead. Figure 4.1 reveals that all SMs actually reduced the run time, indicating an overall positive net gain

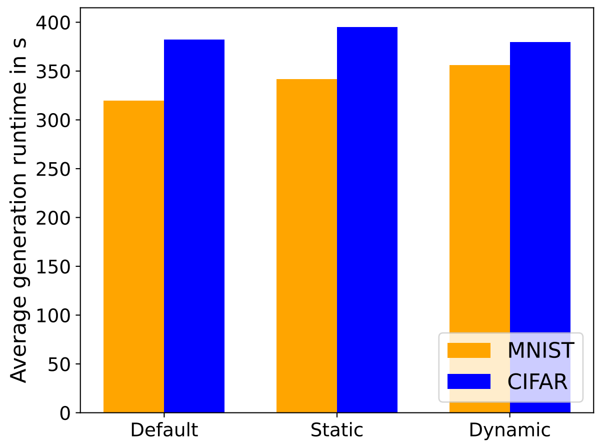


Figure 4.8.: Average run time for a full generation in seconds per SM.

in efficiency. However, what happens when an SM does not halt any run while still performing its computation for cancellation? At best, such a behavior does not (significantly) influence the total run time. This becomes especially pertinent when considering SMs designed to be less aggressive in stopping, as the benefits of early stopping might be negated by increased run time per generation.

For this, the most interesting metric is the run time per generation. This is depicted in Figure 4.8, which shows the sum of the average run time from the end of the second epoch to the end of the last (ninth) epoch and the average run time between generations, i.e., the time from the end of the last epoch of a generation to the end of first epoch of the following generation.

There is a run time increase across all datasets and SMs with one exception. The run time on the MNIST dataset increased by 22 seconds (6.8%) and 37 seconds (11.4%) for the static and dynamic surrogate, respectively. Conversely, for the CIFAR dataset, there is an increase of 13 seconds (3.3%) for the static surrogate, while the dynamic surrogate sees a decrease of three seconds (0.6%). The latter observation for the CIFAR dataset's dynamic surrogate is caused by missing data, since there are almost no runs that actually finish as shown in Figure 4.3.

There are two potential positions where SMs could slow the execution of a generation down: firstly when calling the `cancel()` method, and secondly between the generations when calling `update()` and `start_run()`.

Considering the impact of `cancel()`, SMs can acquire more data with increasing number of generations, which could potentially slow down the training process. This slowdown could particularly be noticeable with the dynamic surrogate, which requires retraining of the local GP with each new loss value. However, the run time comparison across the second to the last epoch in Table 4.9 shows that the SMs do not significantly slow down the training process. For both SMs, the minimum differs by less than one second for CIFAR and by no more than ten seconds for MNIST. At the maximums, both SMs perform even better than the default options, with minimal differences in average run time, suggesting no overall slowdown during training and a negligible impact.

Surrogate	Dataset	Min	Max	Average
Default	MNIST	235.936632	307.739633	280.089069
	CIFAR	337.981799	345.684469	341.822665
Static	MNIST	246.271889	296.196918	279.490288
	CIFAR	337.935431	343.411871	340.746359
Dynamic	MNIST	238.372108	302.594393	270.896124
	CIFAR	337.763292	342.326477	340.203104

Table 4.9.: Run time in seconds from epoch 1 to 9 for one generation, excluding the time for the first epoch and between generations.

Surrogate	Dataset	Min	Max	Average
Default	MNIST	26.255368	91.172827	39.606222
	CIFAR	39.047258	44.357307	40.643902
Static	MNIST	26.529503	329.466086	62.141974
	CIFAR	38.770828	340.248937	54.342863
Dynamic	MNIST	26.850699	335.423860	85.366121
	CIFAR	39.065605	43.886204	39.691245

Table 4.10.: Run time in seconds between generations from the last epoch of the prior generation to the end of the first epoch of the next generation.

Considering the computational complexity of the `start_run()` or `update()` methods, it is important to assess the run time between generations, in particular because some Propulate runs with SMs do not complete many late full generations. Table 4.10 reveals minimal differences in minimum run time between epochs but significant disparities in maximum and average run time. While the minimum differs by not more than one second across datasets and SMs, there appear to be large differences for maximum and average values. The average for CIFAR more than doubles for the dynamic surrogate, adding 45 seconds. The biggest differences appear in the maximum value, jumping from 44 seconds to 340 seconds for the static surrogate on CIFAR. This increase is just 40 seconds shy of the run time for the nine other epochs combined (Table 4.9).

This discrepancy is largely due to a bug in the current implementation affecting parallel processing, causing delays that falsely inflate the perceived run time between generations. Propulate is called with MPI to run two processes for each GPU. When one process finishes early, the data-loader falsely waits for the other process to finish after each generation, causing the big increase. When working properly, there is only a negligible difference as can be seen in the maximum for the dynamic surrogate on CIFAR, which only differs by one second from the default. The original code can

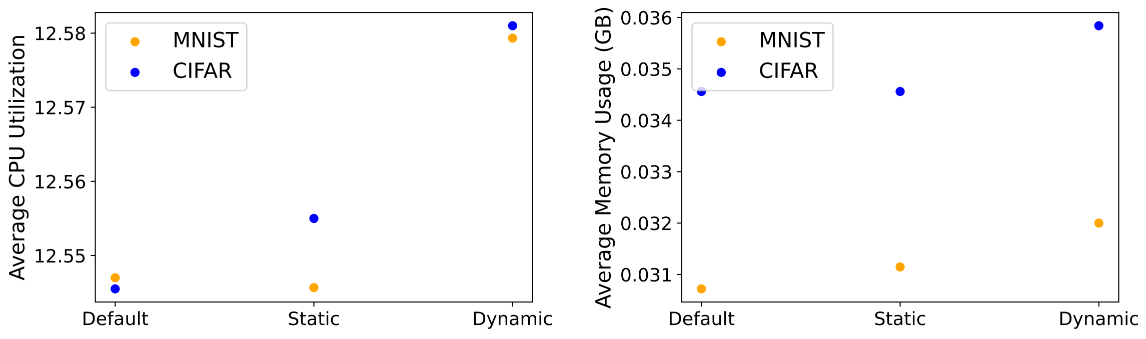


Figure 4.9.: Average CPU utilization in percent and memory utilization in total memory used by PROPULATE by dataset and surrogate.

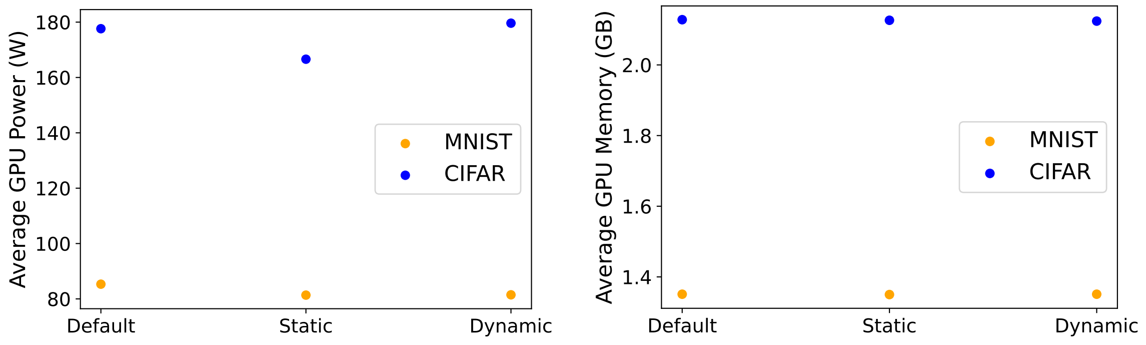


Figure 4.10.: Average GPU power in watt and memory utilization in GB by dataset and surrogate.

be found in an release (see Appendix A) but the bug is fixed in newer commits, as well as inside the original PROPULATE repository.

Lastly, GPU usage statistics, as shown in Figure 4.10, indicate a slight decrease in average GPU power for both surrogates across datasets. The only exception is the CIFAR dynamic surrogate, which is expected behavior following the results from Table 4.10. This finding is corroborated by Figure 4.9, which shows an increase in CPU and memory utilization across surrogates and datasets. However, the impact remains negligible on modern hardware as the CPU utilization went up by less than 0.05% pt. and the ram utilization by less than 5MB. Note that SMs as implemented here run fully on the CPU and increase its utilization, so for more complex SMs or more generations the hardware has to be scaled accordingly.

4.3.5. Energy Consumption and CO_2e

HPO can demand significant energy as explained in the motivation in chapter 1. PERUN can track GPU as well as CPU power consumption. However, Figure 4.9 indicates no substantial rise in CPU and memory usage which only leaves GPU power consumption as interesting metric. GPU utilization remained relatively unchanged, as shown in Figure 4.10. Thus, power consumption reductions are not to be expected on these accounts. Expected reduction in power consumption are only to be attributed to a reduced total run time (Figure 4.1). Perun's CO_2e calculations are based on the current average network carbon intensity [11]. Since the carbon intensity for all experiments was the same, there is a linear relationship between used kilowatt hours (kWh) and CO_2e. According to Figure 4.11, across different datasets and SMs, both energy consumption and CO_2e showed a decrease.

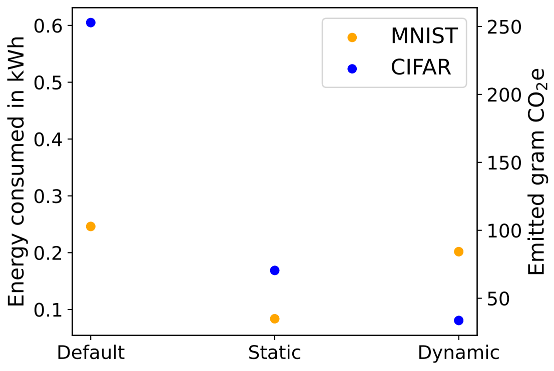


Figure 4.11.: Energy consumption and emitted CO_2e for complete PROPULATE runs by dataset and SM. Kilowatt hours and CO_2e are linearly related to each other as electricity carbon intensity for all runs was the same.

5. Conclusion

This work aimed to integrate early stopping mechanisms into the HPC-adapted optimizer Propulate so as to reduce the computational resources required for large-scale HPO and thus the environmental impact for obtaining powerful state-of-the-art NN models. The presented implementation of SMs within Propulate has demonstrated substantial potential for speeding up HPO. This conclusion is derived from the evaluation and comparison of these SMs across different classification problems, seeds, and early stopping mechanisms.

SMs, specifically static and probabilistic approaches, were tested to predict performance and terminate unpromising training runs early. The evaluation showed that both tested SMs significantly reduced the total run time of Propulate for all presented classification use cases of the MNIST and CIFAR datasets. Specifically, the static SM consistently demonstrated a speedup between 2.8 and 3.3, while the dynamic SM exhibited variable performance with particularly notable efficiency on the CIFAR dataset with a speedup of 7.6 but only 1.1 on the MNIST dataset. At the same time, the average loss obtained was within small margins of the loss that would have been found without SMs. With some seeds, the SMs even found this best loss.

Integrating SMs into HPO frameworks like Propulate paves the way to a significant increase in computational efficiency, which is critical given the environmental and financial costs. It constitutes a major contribution to reducing the CO_2e and computational cost associated with HPO research.

Despite these promising results, the SMs also introduced new challenges. Both the static and dynamic model showed aggressive stopping behaviors that overlooked configurations that would have performed well if allowed to train longer. Additionally, the variations in speedup across different NNs and datasets indicate that further refinement and context-specific tuning of the SMs is necessary to optimize their effectiveness for any new dataset, presenting a kind of chicken-and-egg problem: Effective tuning of an SM requires data that can only be obtained by running the model, yet the desire is to avoid extensive model runs without an efficiently tuned SM in place.

Future research could focus on refining these SMs, particularly on enhancing the dynamic model's ability to balance between terminating early and allowing potentially successful configurations to train sufficiently. Exploring additional SM architectures and incorporating more complex decision-making algorithms, like a prior-data fitted network for predicting learning curves as a way to bootstrap SM performance as well as other BO approaches, could also improve SM performance and applicability and overcome the chicken-and-egg problem.

While this work employed a generator-based approach for SM integration into Propulate, an alternative method using callbacks offers more flexibility in retrieving additional, more nuanced, and detailed metrics from the `ind_loss` function during training. Such an approach would necessitate a more complex setup by users potentially excluding researchers with limited programming experience from using Propulate.

Bibliography

[1] *05b Cifar10 Resnet - Notebook by Aakash Rao N S (aakashns) | Jovian.* URL: https://jovian.com/aakashns/05b-cifar10-resnet (visited on 02/21/2024).

[2] Steven Adriaensen et al. “Efficient Bayesian Learning Curve Extrapolation using Prior-Data Fitted Networks”. In: *Advances in Neural Information Processing Systems* 36 (Dec. 15, 2023), pp. 19858–19886.

[3] Thomas H. W. Bäck et al. “Evolutionary Algorithms for Parameter Optimiza-tionThirty Years Later”. In: *Evolutionary Computation* 31.2 (June 1, 2023), pp. 81–122. ISSN: 1063-6560. DOI: 10.1162/evco_a_00325.

[4] J. Bergstra, D. Yamins, and D. D. Cox. “Making a science of model search: hyperparameter optimization in hundreds of dimensions for vision architectures”. In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML’13. Atlanta, GA, USA: JMLR.org, June 16, 2013, pp. I–115–I–123.

[5] James Bergstra and Yoshua Bengio. “Random search for hyper-parameter optimization”. In: *The Journal of Machine Learning Research* 13 (null Feb. 1, 2012), pp. 281–305. ISSN: 1532-4435.

[6] *Carbon free energy for Google Cloud regions.* Google Cloud. URL: https://cloud.google.com/sustainability/region-carbon (visited on 11/23/2023).

[7] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. “Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves”. In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI’15. Buenos Aires, Argentina: AAAI Press, July 25, 2015, pp. 3460–3468. ISBN: 978-1-57735-738-4.

[8] Katharina Eggensperger. “Towards an Empirical Foundation for Assessing Bayesian Optimization of Hyperparameters”. In: 2013.

[9] Peter I. Frazier. *A Tutorial on Bayesian Optimization.* July 8, 2018. DOI: 10.48550/arXiv.1807.02811. arXiv: 1807.02811[cs,math,stat].

[10] *GPU regions and zones availability | Compute Engine Documentation.* Google Cloud. URL: https://cloud.google.com/compute/docs/gpus/gpu-regions-zones (visited on 11/23/2023).

[11] Juan Pedro Gutiérrez Hermosillo Muriedas et al. “perun: Benchmarking Energy Consumption ofăHigh-Performance Computing Applications”. In: *Euro-Par 2023: Parallel Processing.* Ed. by José Cano et al. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 17–31. ISBN: 978-3-031-39698-4. DOI: 10.1007/978-3-031-39698-4_2.

[12] Hay+Rice. *How to Train Your ResNet*. Myrtle. Sept. 24, 2018. URL: https://myrtle.ai/learn/how-to-train-your-resnet/ (visited on 02/21/2024).

[13] Kaiming He et al. *Deep Residual Learning for Image Recognition*. Dec. 10, 2015. DOI: 10.48550/arXiv.1512.03385. arXiv: 1512.03385[cs].

[14] Peter Henderson et al. *Deep Reinforcement Learning that Matters*. Jan. 29, 2019. DOI: 10.48550/arXiv.1709.06560. arXiv: 1709.06560[cs,stat].

[15] Jeroen van Hoof and Joaquin Vanschoren. *Hyperboost: Hyperparameter Optimization by Gradient Boosting surrogate models*. Jan. 6, 2021. DOI: 10.48550/arXiv.2101.02289. arXiv: 2101.02289[cs,stat].

[16] Ilija Ilievski et al. “Efficient hyperparameter optimization of deep learning algorithms using deterministic RBF surrogates”. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI'17. <conf-loc>, <city>San Francisco</city>, <state>California</state>, <country>USA</country>, </conf-loc>: AAAI Press, Feb. 4, 2017, pp. 822–829.

[17] Riashat Islam et al. *Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control*. Aug. 10, 2017. DOI: 10.48550/arXiv.1708.04133. arXiv: 1708.04133[cs].

[18] Max Jaderberg et al. *Population Based Training of Neural Networks*. Nov. 28, 2017. DOI: 10.48550/arXiv.1711.09846. arXiv: 1711.09846[cs].

[19] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. Jan. 29, 2017. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980[cs].

[20] Aaron Klein et al. “Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets”. In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Artificial Intelligence and Statistics. ISSN: 2640-3498. PMLR, Apr. 10, 2017, pp. 528–536.

[21] Aaron Klein et al. “LEARNING CURVE PREDICTION WITH BAYESIAN NEU- RAL NETWORKS”. In: (2017).

[22] Dounia Lakhmiri and Sébastien Le Digabel. “Use of Static Surrogates in Hyperparameter Optimization”. In: *Operations Research Forum* 3.1 (Feb. 16, 2022), p. 11. ISSN: 2662-2556. DOI: 10.1007/s43069-022-00128-w.

[23] Lisha Li et al. “Hyperband: a novel bandit-based approach to hyperparameter optimization”. In: *The Journal of Machine Learning Research* 18.1 (Jan. 1, 2017), pp. 6765–6816. ISSN: 1532-4435.

[24] Petro Liashchynskyi and Pavlo Liashchynskyi. *Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS*. Dec. 12, 2019. DOI: 10.48550/arXiv.1912.06059. arXiv: 1912.06059[cs,stat].

[25] Maren Mahsereci et al. *Early Stopping without a Validation Set*. June 6, 2017. DOI: 10.48550/arXiv.1703.09580. arXiv: 1703.09580[cs,stat].

[26] *Papers with Code - CIFAR-10 Dataset*. URL: https://paperswithcode.com/dataset/cifar-10 (visited on 12/10/2023).

[27] *Papers with Code - Machine Learning Datasets.* URL: https://paperswithcode.com/datasets (visited on 02/27/2024).

[28] *Papers with Code - MNIST Dataset.* URL: https://paperswithcode.com/dataset/mnist (visited on 12/10/2023).

[29] Esteban Real et al. *Regularized Evolution for Image Classifier Architecture Search.* Feb. 16, 2019. DOI: 10.48550/arXiv.1802.01548. arXiv: 1802.01548[cs].

[30] Eric Schulz, Maarten Speekenbrink, and Andreas Krause. "A tutorial on Gaussian process regression: Modelling, exploring, and exploiting functions". In: *Journal of Mathematical Psychology* 85 (Aug. 1, 2018), pp. 1–16. ISSN: 0022-2496. DOI: 10.1016/j.jmp.2018.03.001.

[31] Bobak Shahriari et al. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: *Proceedings of the IEEE* 104.1 (Jan. 2016). Conference Name: Proceedings of the IEEE, pp. 148–175. ISSN: 1558-2256. DOI: 10.1109/JPROC.2015.2494218.

[32] Benjamin J. Shields et al. "Bayesian reaction optimization as a tool for chemical synthesis". In: *Nature* 590.7844 (Feb. 2021). Publisher: Nature Publishing Group, pp. 89–96. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03213-y.

[33] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. *Practical Bayesian Optimization of Machine Learning Algorithms.* Aug. 29, 2012. DOI: 10.48550/arXiv.1206.2944. arXiv: 1206.2944[cs,stat].

[34] Emma Strubell, Ananya Ganesh, and Andrew McCallum. *Energy and Policy Considerations for Deep Learning in NLP.* June 5, 2019. DOI: 10.48550/arXiv.1906.02243. arXiv: 1906.02243[cs].

[35] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. *Freeze-Thaw Bayesian Optimization.* June 15, 2014. DOI: 10.48550/arXiv.1406.3896. arXiv: 1406.3896[cs, stat].

[36] Oskar Taubert et al. "Massively Parallel Genetic Optimization Through Asynchronous Propagation ofăPopulations". In: *High Performance Computing.* Ed. by Abhinav Bhatele et al. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 106–124. ISBN: 978-3-031-32041-5. DOI: 10.1007/978-3-031-32041-5_6.

[37] Chris Thornton et al. *Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms.* Mar. 6, 2013. DOI: 10.48550/arXiv.1208.3719. arXiv: 1208.3719[cs].

A. Appendix

Supplementary Information

Acknowledgments

The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

Code

The exact version of the code used for this work can be found as release on the PROPULATE fork by vtotiv. The linked branches in Table A.1 are updated to incorporate some bugfixes that improve the performance, thus will not replicate the same results presented in chapter 4.

Link	Description
github.com/Helmholtz-AI-Energy/perun	PERUN
github.com/Helmholtz-AI-Energy/propulate	PROPULATE
github.com/vtotiv/propulate/tree/ibmcloud	PROPULATE with SMs with results
github.com/vtotiv/propulate/tree/bwcloud	PROPULATE with SMs on BwUniCluster2.0

Table A.1.: Code Links

Tools

The following section declares how and where noteworthy tools were used in this work.

The bibliography is auto-generated by Zotero.

ChatGPT (GPT-4) was used to help create the Python code to create Figure 1.1, the two explanatory graphics Figure 3.4 and Figure 3.5, and all figures in chapter 4. ChatGPT (GPT-4) was used on every text for spell, grammar, and style checking and to correct unnecessary and avoidable repeating words and sentences.

The abstract was initially translated from English to German using DeepL.