
UNIVERSITY OF ICELAND

# Parallel and Scalable Hyperparameter Optimization for Distributed Deep Learning Methods on High-Performance Computing Systems

Marcel Aach

2025

FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL ENGINEERING AND COMPUTER SCIENCE

# Parallel and Scalable Hyperparameter Optimization for Distributed Deep Learning Methods on High-Performance Computing Systems

Marcel Aach

Dissertation submitted in partial fulfillment of a
*Philosophiae Doctor degree in Computational Engineering*

Supervisor
Prof. Dr.-Ing. Morris Riedel

Doctoral Committee
Prof. Dr.-Ing. Morris Riedel
Dr.-Ing. Andreas Lintermann
Prof. Dr. rer. nat. Helmut Neukirchen

Opponents
Prof. Dr. Matthias Feurer
Prof. Dr. Marco Aldinucci

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, January 2025

Parallel and Scalable Hyperparameter Optimization for Distributed Deep Learning Methods on High-Performance Computing Systems
(Parallel Hyperparameter Optimization on HPC)

Dissertation submitted in partial fulfillment of a *Philosophiae Doctor* degree in Computational Engineering

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Tæknigarður - Dunhagi 5
107 Reykjavik
Iceland

Telephone: 525-4000

Bibliographic information:
Marcel Aach (2025) *Parallel and Scalable Hyperparameter Optimization for Distributed Deep Learning Methods on High-Performance Computing Systems*, PhD dissertation, Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland, 148 pp.

ISBN 978-9935-9807-8-6

Copyright © 2025 Marcel Aach
All rights reserved

Printing: Háskólaprent
Reykjavik, Iceland, January 2025

# Abstract

The design of Deep Learning (DL) models is a complex task, involving decisions on the general architecture of the model (e.g., the number of layers of the Neural Network (NN)) and on the optimization algorithms (e.g., the learning rate). These so-called hyperparameters significantly influence the performance (e.g., accuracy or error rates) of the final DL model and are, therefore, of great importance. However, optimizing these hyperparameters is a computationally intensive process due to the necessity of evaluating many combinations to identify the best-performing ones. Often, the optimization is manually performed.
This Ph.D. thesis leverages the power of High-Performance Computing (HPC) systems to perform automatic and efficient Hyperparameter Optimization (HPO) for DL models that are trained on large quantities of scientific data. On modern HPO systems, equipped with a high number of Graphics Processing Units (GPUs), it becomes possible to not only evaluate multiple models with different hyperparameter combinations in parallel but also to distribute the training of the models themselves to multiple GPUs. State-of-the-art HPO methods, based on the concepts of early stopping, have demonstrated significant reductions in the runtime of the HPO process. Their performance at scale, particularly in the context of HPC environments and when applied to large scientific datasets, has remained unexplored. This thesis thus researches parallel and scalable HPO methods that leverage new inherent capabilities of HPC systems and innovative workflows incorporating novel computing paradigms. The developed HPO methods are validated on different scientific datasets ranging from the Computational Fluid Dynamics (CFD) to Remote Sensing (RS) domain, spanning multiple hundred Gigabytes (GBs) to several Terabytes (TBs) in size.

# Útdráttur

Að hanna Deep Learning (DL) kerfi er flókið verkefni, sem felur í sér ákvarðanir um almennan arkitektúr kerfisins (t.d. fjölda laga) og fínstillingu á breytum (t.d. við innleiðingu kerfisins). Þessar svokölluðu ofurfæribreytur hafa veruleg áhrif á frammistöðu staðbundna DL líkansins og eru því mjög mikilvægar. Hins vegar getur fínstilling þessa færibreyta verið auðlindafrekt (resource-intensive) ferli vegna þess að það þarf að meta margar samsetningar til að finna þær sem standa sig best og skila besta árangri.
Þessi Ph.D. ritgerð miðar að nýta kraftinn í ofurtölvu kerfum (High-Performance Computing/HPC) til að framkvæma skilvirka Hyperparameter Optimization (HPO) fyrir DL líkön sem eru þjálfuð á stórum vísinda-gagnasöfnum. Í nútíma HPC kerfum, búin fjölda grafískra vinnslueininga (GPU), verður ekki aðeins hægt að mæla margar gerðir með mismunandi samsetningar samhliða, heldur einnig að keyra þjálfun líkananna sjálfra á mörgum GPU einingum. Nýjustu HPO aðferðir, sem byggja á hugmyndinni um snemmbúna stöðvun, hafa sýnt verulega lækkun á keyrslutíma HPO ferlisins. Frammistaða þeirra í stærðargráðu, sérstaklega í tengslum við HPC umhverfi og þegar þau eru notuð í stórum vísindalegum gagnagrunnum, hefur hingað til verið órannsakað svið. Í þessari ritgerð er leitast við að brúa þetta bil með því að innleiða hliðstæðar og stigstærðar/skalanlegar HPO aðferðir sem nýta eðlislæga eiginleika HPC kerfisins og verkflæði sem fela í sér innlimun nýrra reiknivíðmiða. HPO aðferðirnar og virkni þeirra hafa verið staðfest (validated) á mismunandi vísindalegum gagnasöfnum og sviðum, allt frá Computational Fluid Dynamics (CFD) til fjarkönnunar (Remote Sensing/RS), sem spannar nokkur hundruð gígabæt til nokkurra terabæta að stærð.

*Complexity by way of simplicity*

# Contents

# List of Figures

# List of Tables

# List of Original Publications

**Paper I:** **M. Aach**, E. Inanc, R. Sarma, M. Riedel, and A. Lintermann. “Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks.” In: Journal of Big Data 10.1 (2023), p. 96, https://journalofbigdata.springeropen.com/articles/10.1186/s40537-023-00765-w

**Paper II:** **M. Aach**, E. Inanc, R. Sarma, M. Riedel, and A. Lintermann. “Optimal Resource Allocation for Early Stopping-based Neural Architecture Search Methods.” In: Proceedings of the Second International Conference on Automated Machine Learning (2023), https://proceedings.mlr.press/v224/aach23a.html

**Paper III:** **M. Aach**, R. Sarma, H. Neukirchen, M. Riedel, and A. Lintermann. "Resource-Adaptive Successive Doubling for Hyperparameter Optimization on Large Datasets on High-Performance Computing Systems", submitted to Future Generation Computer Systems (2024), https://arxiv.org/abs/2412.02729

**Paper IV:** **M. Aach**, R. Sarma, E. Inanc, M. Riedel, and A. Lintermann. “Short Paper: Accelerating Hyperparameter Optimization Algorithms with Mixed Precision.” In: Proceedings of the SC ’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W ’23. New York, NY, USA: Association for Computing Machinery (2023), pp. 1776–1779, https://doi.org/10.1145/3624062.3624259

**Paper V:** E. Wulff, J. P. Garcia Amboage, **M. Aach**, T. E. Gislason, TH. K. Ingolfsson, T. K. Ingolfsson, E. Pasetto, A. Delilbasic, M. Riedel, R. Sarma, M. Girone, and A. Lintermann. "Distributed Hybrid Quantum-Classical Performance Prediction for Hyperparameter Optimization", Quantum Machine Intelligence (2024), https://link.springer.com/article/10.1007/s42484-024-00198-5 (Note that the first three authors have equally contributed to Paper V)

**Paper VI:** **M. Aach**, R. Sedona, A. Lintermann, G. Cavallaro, H. Neukirchen, and M. Riedel. "Accelerating Hyperparameter Tuning of a Deep Learning Model for Remote Sensing Image Classification." In: IGARSS IEEE International Geoscience and Remote Sensing Symposium (2022), pp. 263–266, https://doi.org/10.1109/IGARSS46834.2022.9883257

# Other Publications

**Paper VII:** E. M, Sumner, **M. Aach**, A. Lintermann, R. Unnthorsson, and M. Riedel. "Speed-Up of Machine Learning for Sound Localization via High- Performance Computing." In: 26th International Conference on Information Technology (2022), pp. 1–4., https://doi.org/10.1109/IT54280.2022.9743519

**Paper VIII:** C. Barakat, **M. Aach**, A. Schuppert, S. Brynjólfsson, S. Fritsch, and M. Riedel. "Analysis of Chest X-ray for COVID-19 Diagnosis as a Use Case for an HPC-Enabled Data Analysis and Machine Learning Platform for Medical Diagnosis Support." In: Diagnostics 13.3 (2023), https://doi.org/10.3390/diagnostics13030391

**Paper IX:** X. Liu, M. Rüttgers, A. Quercia, R. Egele, E. Pfaehler, R. Shende, **M. Aach**, W. Schröder, P. Balaprakash, and A. Lintermann. "Refining Computer Tomography Data With Super-Resolution Networks to Increase the Accuracy of Respiratory Flow Simulations." In: Future Generation Computer Systems 159 (2024), pp. 474-488., https://doi.org/10.1016/j.future.2024.05.020

# Abbreviations

**ADAM** Adaptive Moment Estimation

**AI** Artificial Intelligence

**AM** Additive Manufacturing

**AMP** Automatic Mixed Precision

**ASHA** Asynchronous Successive Halving Algorithm

**AutoML** Automated Machine Learning

**BO** Bayesian Optimization

**BOHB** Bayesian Optimization Hyperband

**BPMN** Business Process Model and Notation

**CFD** Computational Fluid Dynamics

**CNN** Convolutional Neural Network

**CPU** Central Processing Unit

**CT** Computer Tomography

**CV** Computer Vision

**DALI** NVIDIA Data Loading Library

**DL** Deep Learning

**EO** Earth Observation

**GB** Gigabyte

**GNS** Gradient Noise Scale

**GPU** Graphics Processing Unit

**HB** Hyperband

**HEP** High-Energy Physics

**HPC** High-Performance Computing

**HPO** Hyperparameter Optimization

**LES** Large Eddy Simulation

**MB** Megabyte

**ML** Machine Learning

**MLP** Multi-Layer Perceptron

**MPI** Message Passing Interface

**NAS** Neural Architecture Search

**NCCL** NVIDIA Collective Communications Library

**NLP** Natural Language Processing

**NN** Neural Network

**OpenMP** Open Multi-Processing

**PBT** Population Based Training

**QA** Quantum Annealer

**QC** Quantum Computer

**QT-SVR** Quantum Trained Support Vector Regression

**QUBO** Quadratic Unconstrained Binary Optimization

**RASDA** Resource-Adaptive Successive Doubling Algorithm

**RocM** Radeon Open Compute Platform

**RS** Remote Sensing

**SGD** Stochastic Gradient Descent

**SVR** Support Vector Regression

**TB** Terabyte

# Acknowledgments

First and foremost, I would like to thank my doctoral committee, consisting of Morris Riedel, Andreas Lintermann, and Helmut Neukirchen, for their excellent supervision. Their guidance, whether provided in Jülich or Iceland, in person or remotely, has been invaluable. I appreciate the autonomy they granted me to explore my own research ideas, as well as the opportunities they facilitated for me to present my work at conferences all around the globe. Their extensive feedback and a keen eye for detail have also played a major role in making my publications and this thesis possible.

I am also grateful to all the current and former colleagues of the SimLab Fluids & Solids Engineering in Jülich and the High-Productivity Data Processing group in Iceland for fostering a great and encouraging working atmosphere. I particularly wish to acknowledge Rakesh and Eray for the frequent discussions on research ideas and code implementations, as well as their thoughtful feedback, even under the pressure of tight deadlines. I am grateful to Rocco and Gabriele for their guidance at the beginning of my PhD journey, to Amer and Edoardo for our fruitful collaboration, from which I learned a lot, and to Chadi, Eric, Reza, Xin, and Mario for the work on joint research ideas. I am also thankful to my colleagues in the CoE RAISE project from across Europe for three research-intensive and, at the same time, often fun years.

Finally, I would also like to thank my family, my mother Elisabeth, my sister Linda, and Jonas, who have been a constant source of strength and encouragement through every step of this journey. I am especially grateful to Katja, who has supported me through these sometimes very stressful years, particularly during times when my work often extended well into the evening hours. I truly appreciate her presence by my side.

I dedicate this thesis to my late father, Til, who sparked my interest in the sciences at a very young age.

# 1 Introduction

This chapter provides an introduction to the thesis, laying out the motivation (Sec.1.1) and thesis objectives (Sec.1.2), as well as giving an overview of the outline (Sec. 1.3) and the main contributions made (Sec. 1.4).

## 1.1 Motivation

The exponential growth in data collection across all domains of science and engineering over recent decades has rendered their manual analysis infeasible because of sheer volume. This trend is likely to persist as the barriers to the acquisition of new data are continuously getting lower, e.g., due to an ever-increasing number of satellites monitoring the Earth from above or through more powerful measurement devices. Extraction of meaningful insights from this kind of "Big Data" therefore requires the adoption of more automized approaches, for which Machine Learning (ML) has become the main method of choice. Especially Deep Learning (DL) models and Neural Networks (NNs) have shown to be capable of accurate analysis, e.g., in the Computer Vision (CV)-, Biology-, High-Energy Physics (HEP), or Natural Language Processing (NLP) domain [1, 2, 3, 4, 5].
However, two challenges arise when working with NNs: First, the general design of the NN is complex. Selecting suitable hyperparameters, such as a proper network architecture or a suitable learning rate (see Fig. 1.1) to guide the optimizer in the right direction during the training process usually requires expert knowledge to create models with high accuracy. Many of these decisions are, therefore, still manually performed by human researchers, who try to select optimal hyperparameters based on experience. Second, training NNs on massive datasets requires sufficient hardware computing resources and software frameworks that can efficiently parallelize the training process to achieve results in a reasonable amount of time. Such hardware resources can be provided by High-Performance Computing (HPC) systems, which are nowadays mainly equipped with a large amount of Graphics Processing Units (GPUs), the primary type of accelerator for DL workloads. To take advantage of these systems, it is possible to split up the large dataset onto multiple GPU devices and then perform the NN training using parallel algorithms, a technique that is referred to as distributed DL or data-parallel DL [6, 7]. At the same time, also Hyperparameter Optimization (HPO) [8], which describes the systematic search for hyperparameters, can be performed in a distributed setting. Given a sufficient number of GPUs, multiple NN candidates with varying hyperparameters can be trained in parallel, which yields well-performing models much faster than training them all sequentially.


*Figure 1.1. Learning curves of training a small Convolutional Neural Network (CNN) with different initial learning rates and a cyclic learning rate scheduler on the cifar-10 dataset [11].*

While distributed methods for systematic HPO and Neural Architecture Search (NAS) have long existed [9], they are often not evaluated on a large scale and not tailored towards HPC usage. Consequently, these methods often neglect some of the inherent features of large-scale HPC systems, which include an expansive storage system for fast data loading, numerous accelerators for *fast computation*, and an optimized interconnection network structure that facilitates *fast communication*. Additionally, a large portion of literature in the field of Automated Machine Learning (AutoML) centers around benchmarking datasets from the CV domain and several scientific disciplines (such as the Remote Sensing (RS) or Computational Fluid Dynamics (CFD) domains) have so far not taken advantage of sophisticated HPO methods.
**The motivation of this thesis is therefore to leverage the inherent features of large-scale HPC systems, for performing efficient, distributed HPO to improve the performance of DL models on several scientific applications.** This thesis is motivated by use cases that are mainly embedded within the Center of Excellence "Research on AI- and Simulation-Based Engineering at Exascale" (CoE RAISE)[1], a collaborative effort between scientific and industrial researchers to develop novel, scalable Artificial Intelligence (AI) technologies. CoE RAISE provides numerous use cases from the engineering domain and the natural sciences, along with extensive open datasets that measure up to 8.3 Terabyte (TB) in size. The HPC systems used for the development and empirical evaluations of the HPO methods are mainly located at the Jülich Supercomputing Centre and include the JUWELS BOOSTER machine [10], which is among of the most powerful supercomputers in Europe. Apart from classical supercomputing, also the novel computing paradigm of Quantum Annealer (QA) is investigated in regard to performing efficient HPO.

[1] CoE RAISE: `https://www.coe-raise.eu/`

## 1.2 Thesis Objectives

The main research question this thesis seeks to answer is whether it is possible to leverage the inherent features of HPC systems to perform HPO efficiently. This is structured into four thesis objectives:

- **TO 1** – Evaluate and validate software frameworks that can be used for distributed DL and distributed HPO at large scale to identify toolsets utilizing HPC systems
- **TO 2** – Develop a scheduler facade to combine distributed DL and HPO to perform scalable and resource-efficient HPO on HPC systems
- **TO 3** – Implement a workflow to couple an emerging Quantum architecture with an HPC systems for HPO and exploit novel accelerator capabilities
- **TO 4** – Improve the performance of DL models from different scientific applications with HPO

While **TO1** serves as an introduction point to familiarize with the relevant software packages in an HPC environment, **TO2** and **TO3** are concerned with the development and optimization of algorithms and techniques to use at the intersection of distributed DL, HPO and HPC. In **TO4**, the methods and workflows developed are applied to use cases from different scientific domains to demonstrate their advantages.

## 1.3 Outline

This thesis follows a cumulative style, presenting the results of the conducted research as a collection of articles published in conferences and journals. Additionally, all code developed is released on Git repositories.

### 1.3.1 Thesis Structure

The thesis is organized as follows:

- **Chapter 1** provides an introduction and describes the motivation of this thesis.
- **Chapter 2** introduces relevant concepts on the topics of HPC, DL and HPO.
- **Chapter 3** summarizes the related work in the fields of large-scale, distributed DL and HPO, upon which this thesis builds and extends.
- **Chapter 4** presents the conference and journal publications in detail, including those currently under review at journals.
- **Chapter 5** concludes the thesis with a summary of the main findings and a brief outlook of directions of future research.

### 1.3.2 Publications

The following publications are part of this thesis. The complete documents are included as appendices.

- **Paper I:**
  **M. Aach**, E. Inanc, R. Sarma, M. Riedel, and A. Lintermann. "Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks." In: Journal of Big Data 10.1 (2023), p. 96, https://journalofbigdata.springeropen.com/articles/10.1186/s40537-023-00765-w

- **Paper II:**
  **M. Aach**, E. Inanc, R. Sarma, M. Riedel, and A. Lintermann. "Optimal Resource Allocation for Early Stopping-based Neural Architecture Search Methods." In: Proceedings of the Second International Conference on Automated Machine Learning (2023), https://proceedings.mlr.press/v224/aach23a.html

- **Paper III:**
  **M. Aach** , R. Sarma, H. Neukirchen, M. Riedel, and A. Lintermann. "Resource-Adaptive Successive Doubling for Hyperparameter Optimization on Large Datasets on High-Performance Computing Systems", submitted to Future Generation Computer Systems (2024), https://arxiv.org/abs/2412.02729

- **Paper IV:**
  **M. Aach**, R. Sarma, E. Inanc, M. Riedel, and A. Lintermann. "Short Paper: Accelerating Hyperparameter Optimization Algorithms with Mixed Precision." In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23. New York, NY, USA: Association for Computing Machinery (2023), pp. 1776–1779, https://doi.org/10.1145/3624062.3624259

- **Paper V:**
  E. Wulff, J. P. Garcia Amboage, **M. Aach**, T. E. Gislason, TH. K. Ingolfsson, T. K. Ingolfsson, E. Pasetto, A. Delilbasic, M. Riedel, R. Sarma, M. Girone, and A. Lintermann. "Distributed Hybrid Quantum-Classical Performance Prediction for Hyperparameter Optimization", Quantum Machine Intelligence (2024), https://doi.org/10.21203/rs.3.rs-4270639/v1 (Note that the first three authors have equally contributed to Paper V)

- **Paper VI:**
  **M. Aach**, R. Sedona, A. Lintermann, G. Cavallaro, H. Neukirchen, and M. Riedel. "Accelerating Hyperparameter Tuning of a Deep Learning Model for Remote Sensing Image Classification." In: IGARSS IEEE International Geoscience and Remote Sensing Symposium (2022), pp. 263–266, https://doi.org/10.1109/IGARSS46834.2022.9883257

### 1.3.3 Code Repositories

The following code repositories were created during the thesis and relate to the publications.

- `https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/resnet-benchmarks` contains the code for reproducing the large-scale benchmarks of **Paper I**
- `https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/nas-allocation` contains the code for running the NAS experiments of **Paper II**
- `https://github.com/olympiquemarcel/rasda` contains the code for the Resource-Adaptive Successive Doubling Algorithm (RASDA) method of **Paper III**
- `https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/mixed-precision-hpo` contains the code for performing mixed-precision HPO of **Paper IV**
- `https://github.com/JP-Amboage/qtml-hybrid-workflow` contains the code for the hybrid quantum-classical workflow of **Paper V**
- `https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/switching_bs` contains the code for tuning the RS model of **Paper VI**

## 1.4 Contributions

All presented papers from Sec. 1.3.2 relate to one or more of the thesis objectives in Sec. 1.2 as shown in Tab. 1.1.

For the accomplishment of **TO1**, a large-scale evaluation of distributed DL frameworks was conducted in Paper I, scaling them up to 1024 GPUs on the JUWELS BOOSTER HPC machine. The communication and data loading processes were found to significantly impact runtime, highlighting the importance of a fast HPC network interconnect for the exchange of data between the file systems and the GPUs, as well as between GPUs for large-scale DL training. Additionally, the study confirmed the degradation of

*Table 1.1. Relation of publications to the TO.*

| | **Paper I** | **Paper II** | **Paper III** | **Paper IV** | **Paper V** | **Paper VI** |
|---|---|---|---|---|---|---|
| **TO1** | × | × | | | | × |
| **TO2** | | × | × | × | | × |
| **TO3** | | | | × | × | |
| **TO4** | | | × | × | × | × |

solution quality when training with large batch sizes and found that it could be addressed to some degree by scheduling of the learning rate.
In pursuit of **TO2**, a larger focus was put on the HPO process on HPC systems. Paper II presented an empirical evaluation of static GPU allocations for optimally balancing a fixed number of GPUs between the HPO and the distributed DL loop for commonly used NAS methods. The findings from **TO1** were utilized to implement distributed DL training with one of the evaluated software frameworks. An adaptive resource allocation strategy (RASDA) is presented in Paper III, which extends the investigations of Paper II by allocating more GPUs to the distributed training of more promising hyperparameter candidates during the HPO process. This method was then scaled to 1024 GPUs and tested on various use cases, effectively exploiting the main features of HPC: fast computation and fast network communication for the exchange of data. It is one of the first systematic HPO methods that is applied to a scientific dataset that spans multiple TBs.
**TO3** resulted in novel and first-of-its-kind workflows for quantum performance prediction and mixed-precision calculations combined with early stopping-based HPO methods. In the first case, a combination of a QA with a classical HPC system lowered the overall cost of the distributed HPO process. In the second case, the usage of mixed-precision calculations led to significant speed-ups for several HPO methods, but it was found that for evolutionary optimization the saving and loading of model weights becomes a bottleneck. To alleviate this issue, it was found beneficial to decrease the checkpoint frequency.
Finally, for **TO4**, the developed methods and workflows were applied to scientific use cases. Paper VI applies systematic HPO for enhancing the classification performance of a RS model using HPC resources. Papers III and IV improved the performance of a CFD model for flow reconstruction using the proposed workflows. In co-authored publications, the HPO workflow for a sound localization model was accelerated, while Paper VIII boosted the classification accuracy of a model for pneumonia and COVID-19 detection. Paper IX increased the accuracy of a super-resolution network for Computer Tomography (CT) data.
The timeline of the thesis, detailing the period from the beginning to the completion of the doctoral studies is shown in Fig. 1.2, following the Business Process Model and Notation (BPMN).

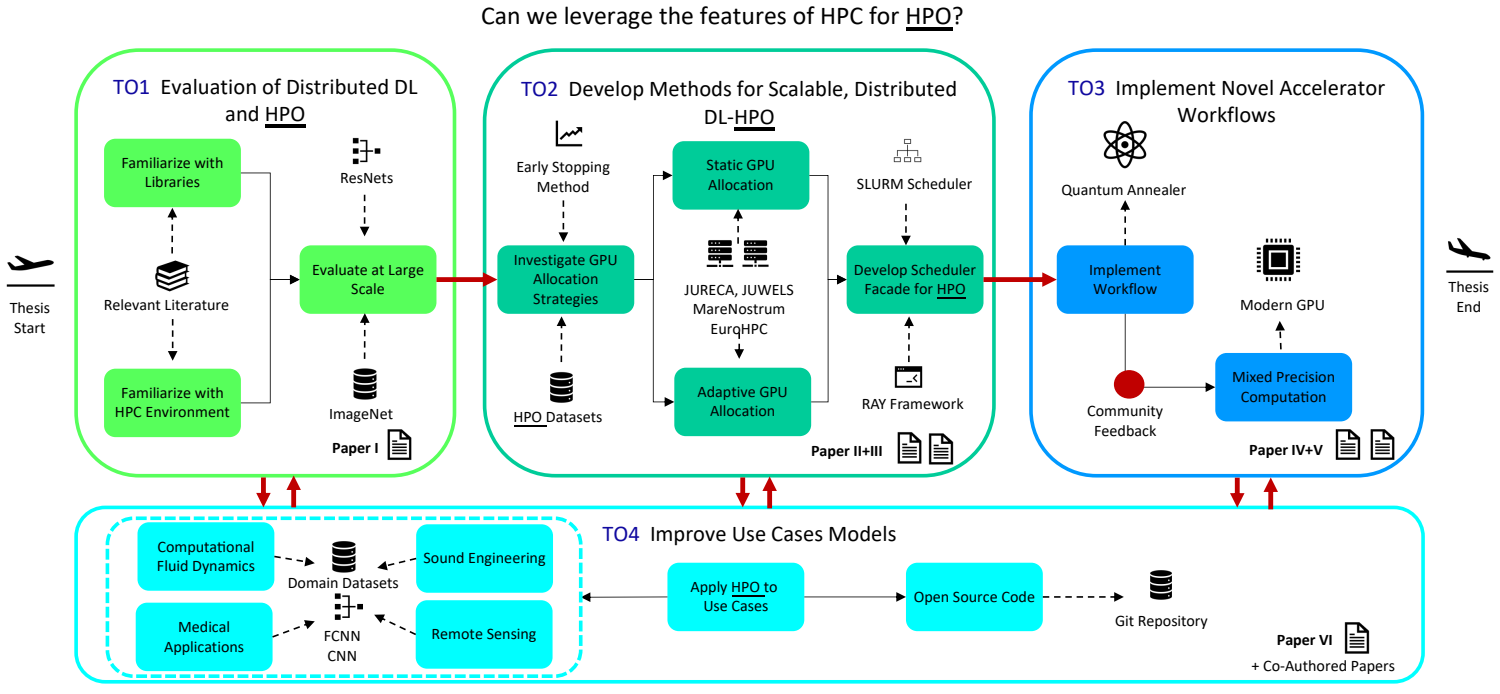



*Figure 1.2. BPMN flow diagram of the thesis. The structure follows the* ***TOs****.*

# 2 Background

This chapter establishes the background of this thesis, including the three main topics: High-Performance Computing (HPC) in Sec. 2.1, Deep Learning (DL) in Sec. 2.2 and Hyperparameter Optimization (HPO) in Sec. 2.3. In Sec. 2.4 the applications that are used as case studies are briefly introduced.

## 2.1 High-Performance Computing and Quantum Annealing

HPC refers to the use of supercomputers for solving advanced scientific problems with parallel processing techniques. The basic ingredient of an HPC system is a single node, which acts as a standalone computer and consists of several Central Processing Units (CPUs), often complemented by accelerators such as GPUs, and a shared memory that all processors can access. The processors on the nodes are responsible for performing fast calculations. Historically, computational capabilities were enhanced through the exponential increase in transistor density on processor chips, a trend described by Moore's law [12]. However, this approach has reached its physical limitations, and current advancements in computing capacity are primarily achieved by allocating more processors in parallel and by using hardware accelerators that are more specialized than general purpose CPUs. Apart from the processors for computation, HPC systems also feature high-bandwidth, low-latency network connections (so-called interconnects, e.g., via InfiniBand[2]) between the nodes to facilitate fast communication, which is crucial for solving large scientific problems. HPC environments typically also incorporate high-capacity storage systems to handle the saving and loading of the current state of the experiments (checkpoints) and massive datasets. To ensure efficient utilization of computing resources, specialized software is necessary for running jobs in parallel. For CPU workloads on a single-node level, this includes libraries like Open Multi-Processing (OpenMP) [13], while in the multi-node setting, Message Passing Interface (MPI) [14] is used. GPUs rely on vendor-specific libraries such as NVIDIA Collective Communications Library (NCCL)[3] or Radeon Open Compute Platform (RocM)[4] for this purpose. Scheduling and monitoring of parallel jobs are handled by software tools, such as SLURM [15] or LLView [16]. Large HPC systems additionally require advanced cooling mechanisms.

[2] InfiniBand: `https://www.infinibandta.org/`
[3] NCCL: `https://github.com/NVIDIA/nccl`
[4] RocM: `https://github.com/ROCm/ROCm`

In recent years, modular supercomputing [17] has seen a rise in popularity. In this approach, not all nodes of a cluster have the same architecture; instead, some parts of the cluster are targeted toward specific applications. For example, certain nodes may feature more GPUs per node to handle DL workloads, while other nodes feature more CPUs to handle cases from classical simulation sciences.
The performance of supercomputers is regularly assessed through the TOP500 list[5], which ranks the fastest systems in the world based on the LINPACK benchmark[6]. As of the latest ranking (06/2024), the Frontier System at Oak Ridge National Laboratory leads with a performance of 1 206 exaflops, which means it is capable of performing more than $10^{18}$ floating-point operations per second. The most energy-efficient cluster as of 06/2024 is the JEDI system at Jülich Supercomputing Centre.

A common metric for tracking the efficiency of a program or algorithm running on such an HPC system is to perform strong scaling tests. Here, a fixed problem size is chosen, while the number of workers is increased and the runtime of the program is measured at the difference scales. The speed-up can then be calculated by

$$S = \frac{T_1}{T_N}, \tag{1}$$

where $N$ is the number of workers, $T_1$ is the runtime of the single worker baseline case, and $T_N$ is the runtime on $N$ workers. For easier comparison, the parallel efficiency $E_N$ can then be expressed by

$$E_N = \frac{S_N}{N}, \quad E_N \in [0,1] \tag{2}$$

In case $E_N$ equals (or is close to) 1, the best possible scaling performance is achieved. Only in exceptional cases, values larger than 1 are possible. Analyzing $E_N$ allows for a standardized comparison of the different methods.
When the problem size is too large to fit onto a single node, and thus no $T_1$ baseline case runtime can be computed, a weak scaling analysis is performed. Here, the problem size is increased with the number of workers. Ideally, the runtime then stays constant across scales.
These scaling behaviors are theoretically bounded by two fundamental laws in parallel computing. Amdahl's law describes the maximum speedup achievable when parallelizing a program with both serial and parallel portions:

$$S_N = \frac{1}{(1-p) + \frac{p}{N}} \tag{3}$$

where $p$ is the proportion of the program that can be parallelized and $N$ is the number of processors. It is related to the strong scaling metric. Gustafson's law, on the other hand, considers that the problem size often scales with the number of processors available, leading to a different speedup formula:

$$S_N = (1-p) + p * N \tag{4}$$

[5]TOP500: `https://top500.org/`
[6]LINPACK: `https://www.netlib.org/benchmark/hpl/`

where $p$ is the proportion of the program that can be parallelized and $N$ is the number of processors. It relates to the weak scaling metric.

### Quantum Annealing

Apart from classical CPU- and GPU-based supercomputers, other types of hardware architectures are emerging. One example are Quantum Computers (QCs), which use the properties of quantum mechanics to perform calculations. While it has been shown that these gate-based QCs have some theoretical advantages over classical computers [18], it is unlikely to materialize in practice in the near future [19]. In contrast to gate-based QCs, QA [20] use a heuristical optimization process for solving Quadratic Unconstrained Binary Optimization (QUBO) problems [21], a special class of optimization problems, where problem variables can take values over a binary set. The general cost function $C(v_1, \ldots, v_M)$ of a QUBO problem with $M$ binary variables $v_i$, $i = 1, \ldots M$ is defined as:

$$C(v_1, \ldots, v_M) := \sum_{i \leq j} Q_{ij} v_i v_j, \tag{5}$$

where $Q$ is the QUBO upper diagonal weight matrix that stores the coefficients of the problem.
When solving the QUBO problem on QA systems manufactured by D-Wave[7], the $v_i$ are encoded to represent the values of the qubits. The optimization problem is then solved by cooling the system down, which tends to its minimum energy state. The qubits can then be read out and represent a minimum of the cost function and thus a solution to the original optimization problem. The evolution to a minimum energy state is probabilistic and therefore repeated several times, so the QA returns not one but a set of solutions. The JUPSI machine[8] is currently one of the largest D-Wave QA and is located at Jülich Supercomputing Centre. To leverage such devices for ML, it is necessary to rewrite the ML optimization problem as a QUBO problem, which in literature has been shown to work, e.g., for Support Vector Machine methods [22, 23, 24].

## 2.2 Deep Learning

In this sub-section, the concept of Deep Learning (DL) is introduced in Sec. 2.2.1 and the different methods of distributing the training are summarized in Sec. 2.2.2.

### 2.2.1 Introduction to Deep Learning

Deep NNs are ML models that consist of multiple layers of neurons. Input data to the network is processed through these layers, with each layer learning increasingly complex representations through its parameters. From a mathematical point of view, these parameters at each layer $l$ consist of a weight matrix $W^{(l)}$ and a bias vector $b^{(l)}$,

[7] D-Wave: `https://www.dwavesys.com/`

[8] JUPSI: `https://www.fz-juelich.de/en/ias/jsc/systems/quantum-computing/juniq-facility/juniq/d-wave-advantagetm-system-jupsi`

which are collectively denoted as $\theta = \{W^{(l)}, b^{(l)}{}_{l=1}^{L}\}$ for a network with $L$ layers. The layer-wise transformation follows:

$$h^{(l)} = \sigma(W^{(l)} h^{(l-1)} + b^{(l)}) \tag{6}$$

where $h^{(l)}$ represents the layer output and $\sigma$ denotes a non-linear activation function such as the Sigmoid function.

Given a Neural Network (NN), consisting of weights and biases $\theta$, the prediction performance on a training dataset $\mathscr{D}$, consisting of input and target values can be computed by measuring the distance between the predicted and actual target values with a loss function $L(\theta)$. Prominent choices for $L$ include the mean-squared error in a regression case or the cross-entropy in the classification case. To minimize $L(\theta)$ (and optimize the $\theta$ values), the Stochastic Gradient Descent (SGD) technique [25] is used. Here, at timestep $t$, a certain amount of samples from the training dataset $\mathscr{D}$ is chosen (batch size $BS$), over which the gradient of the loss $G(\theta_t) = \nabla L(\theta_t)$ is computed through backpropagation, by recursive application of the chain rule. The NN parameters are then updated in the direction of the steepest descent of the gradient with step-size (learning rate) $\varepsilon$:

$$\theta_{t+1} = \theta_t - \varepsilon \sum_{BS} G(\theta_t) \tag{7}$$

This process is repeated iteratively on a single device (e.g., a single GPU). While SGD is not guaranteed to converge to a global minimum, experience has shown it can find local minima in a reasonable amount of time [26]. Other variations of gradient descent optimization methods include Adaptive Moment Estimation (ADAM) [27] or RMSProp[9].

### 2.2.2 Distributed Deep Learning

Several methods for distributing the training of deep NNs across different workers exist that can be categorized into three approaches: data-parallel training where each worker holds a copy of the model and trains it on a subset of data, model-parallel training, where each worker holds a part of the model weights, and federated learning, where the data is split across different clients. The methods are introduced in the following.

#### Data-Parallel Deep Learning

When the training dataset is large, it is possible to accelerate the process by using multiple devices at the same time and performing data-parallel training. Here, the training dataset $\mathscr{D}$ is partitioned across the total number of workers $N$, and each worker then trains an identical copy of the model on its distinct subset of the data $\mathscr{D}_1 \cup \mathscr{D}_2 \cup \ldots \cup \mathscr{D}_N$. Specifically, each worker $i = 1 \ldots N$ runs one model forward and backward pass with a predefined number of samples (local batch size $BS_{local}$) of its subset of data to compute its local gradients $G_{local}(\theta)$ concerning the model parameters $\theta$. After the

[9] RMSProp: `https://keras.io/api/optimizers/rmsprop/`

backward pass, these local gradients are aggregated and averaged across all workers to compute the global gradient $G_{global}$ as follows:

$$G_{global}(\theta) = \frac{1}{N}\sum_{i=1}^{N} G_i(\theta) \tag{8}$$

The averaged global gradient is then used to update the model parameters on all workers every $BS_{global} = BS_{local} \times N$ samples [6]. Data-parallel training can be performed with synchronous or asynchronous communication, and with a centralized parameter server or and decentralized approach [7].

### Model-Parallel Deep Learning

When the NN itself is too large to fit into the memory of a single device, model-parallel training becomes essential. Here, the model parameters $\theta$ are partitioned across the total number of workers $N$, such that $\theta_1 \cup \theta_2 \cup \ldots \cup \theta_N$ (e.g., each NN layer could reside on a different worker). As the input data flows though the network during the forward pass and the gradients during the backward pass, the output and input of the layers needs to be passed from device to device. While this distribution enables the training of larger models, it introduces communication overhead and can result in worker idle time as devices wait for data from previous layers. Different versions of model-parallel training include pipeline [28] or tensor-parallelism [29]. Model-parallel training is essential for NN with billions of parameters.

### Federated Deep Learning

The federated learning approach becomes necessary, when the training data $\mathscr{D}_1 \cup \mathscr{D}_2 \cup \ldots \cup \mathscr{D}_N$ is distributed across $N$ different facilities and can not be shared, e.g., due to privacy regulations or practical constraints. In this case, at each facility one client is launched that performs one or multiple local NN training steps on its private dataset to obtain locally updated model parameters $\theta_i$. In the FedAvg algorithm [30], these local models are then aggregated and a weighted average over all model parameters $\theta_i$ of the local clients $i = 1 \ldots N$ is computed. The averaged global model is then redistributed to all clients for the next round of training. A potential bottleneck are the communication costs between clients [31]. While the primary focus of federated learning is the preservation of data privacy [32], recent results have shown it can also be leveraged to train large NNs across geographically different HPC systems [33]. This distributed approach effectively reduces the time-to-solution of NN training.

## 2.3 Hyperparameter Optimization

In DL, the hyperparameters refer to the kind of values that are usually set by the user before the training run. They are distinct from the actual parameters of the NN, as they usually remain fixed throughout the training run and are not optimized during the forward and backward passes of the training. Still, the types of hyperparameters vary a

lot. On the one hand, they include architectural parameters, such as the number and type of layers, number of neurons per layer, or activation functions used in between layers. The search for optimal parameters related to the NN architecture is also described as NAS. On the other hand, the classical hyperparameters include parameters of the optimization process, such as the choice of the optimizer (e.g., SGD or ADAM), the learning rate, or the weight decay. Sometimes also data pre-processing steps are optimized, e.g., the crop size in image processing. As a whole, this process is also referred to as AutoML [8].
Mathematically speaking, the performance of a NN regarding a metric or loss function (e.g., the error on a validation data set) can be described by

$$f(x) : \mathscr{X} \to \mathbb{R} \tag{9}$$

where $\mathscr{X}$ is the space of all possible hyperparameters and model architectures. Then main goal of HPO is to systematically minimize the objective function $f(x)$ by finding hyperparameters $x^* \in \mathscr{X}$ such that

$$x^* \in \arg\min_{x \in \mathscr{X}} f(x) \tag{10}$$

Performing this kind of minimization is challenging due to several reasons [34]. The evaluation of the objective function $f$ for a single parameter combination $x$ is expensive, as it requires a complete model training run, which for large models or datasets, demands high computational resources. Furthermore, the search space $\mathscr{X}$ that the hyperparameters are sampled from is complex and high-dimensional. This is due to the diverse nature of hyperparameters, e.g., the number of layers in a NN is an integer while the learning rate is of floating point value type. Classical optimization techniques, such as gradient descent can usually not be applied in this setting. In order to perform systematic HPO, it is, therefore, necessary to rely on a low-cost approximation of $f(x)$ and sample configurations to be evaluated efficiently.

## 2.4 Applications

In this sub-section, the applications from different fields of science are introduced. They are used as case studies in this thesis.

### 2.4.1 Computer Vision Benchmarking

Computer Vision (CV) has been an active field of research in the past decades. To evaluate newly proposed methods in a standardized way, several widely accessible benchmarking datasets exist. Two examples of smaller datasets that are a few hundred Megabyte (MB) in size include the cifar-10 dataset [11] (consisting of 60,000 images from 10 different classes, such as dog or cat) and the TinyImageNet dataset [35] (consisting of 100,000 images, split into 200 classes). An example of a larger benchmarking dataset is ImageNet ($\approx$ 300 GB in size when uncompressed), originally introduced for

the ImageNet Large Scale Visual Recognition Challenge[10], which features 1,281,167 training images and 50,000 validation images divided into 1,000 object classes [36]. One way to measure the performance of ML methods that have the goal of image recognition on these datasets is through the validation accuracy [37], which is computed from correctly classified validation images. The datasets have all been widely used as reference benchmarks in HPO and NAS literature [38, 39, 40].

### 2.4.2 OpenML Platform

OpenML is a collaborative, open-source platform designed to facilitate the sharing and reproducibility of ML datasets and experiments [41]. It provides a structured framework for organizing and sharing various components of the ML workflow, centered around datasets, the type of task (e.g., classification or regression), the flow to apply to the data sets (which includes the actual algorithm to run), and the runs (containing the actual results of applying a flow to a task). This structure allows for comprehensive tracking and sharing of experiments, enabling researchers and practitioners to easily access, reproduce, and build upon others' work. OpenML provides integration of various programming languages, including Python and R [42, 43]. Currently, the platform features more than 5,700 datasets and has become a foundation for several standardized benchmarks, particularly in the field of HPO [44, 45].

### 2.4.3 Remote Sensing

Remote Sensing (RS) describes the process of acquiring information about an object, such as the Earth's surface, atmosphere, or oceans, from a distance without direct physical contact with the object. The technique relies on advanced measurement devices, typically mounted on satellites or other types of aircraft. These sensors capture electromagnetic radiation, containing light, radio, and other wavelengths emitted from the object. In the case of Earth Observation (EO), visible and infrared sensors, for instance, can be leveraged to produce high-resolution images for land cover mapping or monitoring of vegetation. Radar sensors can offer valuable information on terrain elevation or soil moisture levels. Due to several satellite missions, such as the Copernicus Sentinel-2[11] the same area of the Earth is observed every few days, generating a large amount of raw observation data. To gain insights from this data, it is necessary to apply automated data analytics methods, such as Support Vector Machines [46] or NNs [47]. Benchmarking datasets for assessing the performance of these analytics methods include the BigEarthNet [48], which is 66 Gigabytes (GBs) in size. In this thesis, the goal is to improve the performance of image classification DL models which are trained on BigEarthNet.

[10] ImageNet Large Scale Visual Recognition Challenge: `https://www.image-net.org/challenges/LSVRC/`

[11] Sentinel-2: `https://dataspace.copernicus.eu/explore-data/data-collections/sentinel-data/sentinel-2`

### 2.4.4 Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) is a branch of fluid mechanics that involves solving problems concerning fluid flows via numerical approximation. At its core, CFD involves the discretization of the continuous fluid domain into single cells, forming a mesh. The governing equations of fluid dynamics (typically the Navier-Stokes equations) are then solved numerically with the application of boundary conditions within these cells in an iterative fashion.
For the numerical solution, mainly three methods are considered: the finite difference, finite element, or finite volume method. As such methods are computationally expensive, HPC systems are necessary to perform fast and accurate simulations. Still, especially turbulence modeling remains a significant challenge in CFD due to the wide range of length and time scales involved. The spectrum of turbulence modeling approaches includes methods like Large Eddy Simulation (LES), with different approaches striking a balance between computational feasibility and accuracy of the simulation. As especially fine-grained simulations are still very computationally costly, there has been growing interest in the field of scientific machine learning, which aims at augmenting traditional CFD solvers with data-driven ML methods.
In this thesis, an 8.3 TB dataset, generated from a high-fidelity LES of an actuated Turbulent Boundary Layer is used for training DL models, with the goal of learning how to reconstruct the flow field. The actuation of the Turbulent Boundary Layer is achieved through a sinusoidal moving geometry, with the movement controlled by specific actuation parameters [49]. These simulations are executed using the m-AIA code (an extension of the Zonal Flow Solver [50]), a simulation code developed at RWTH Aachen University.

## 2.5 Deep Learning Models

For the applications from Sec. 2.4, several types of NNs are used as base models, which are then optimized by modifying their architecture and optimizer-related parameters. This includes simple Multi-Layer Perceptron (MLP) models [51], which are feedforward NNs usually consisting of just an input, several hidden, and an output layer. Each of the layers consists of multiple neurons, combined with a non-linear activation function (such as ReLU or Sigmoid). In this kind of structure, all neurons inside a layer are connected to all neurons in the following layer (a fully-connected NN architecture), which increases the number of connections and makes the analysis of image-like input data infeasible.
To address this limitation, Convolutional Neural Networks (CNNs) were developed, introducing an inductive bias for the analysis of image data that emphasizes locality. Unlike MLPs, CNNs use convolutional layers that apply filters (or kernels) to local regions of the input. This approach significantly reduces the number of parameters compared to fully connected networks while at the same time helps to capture spatial relationships within the data. The convolutional layers are typically followed by pooling layers, which further reduce the spatial dimensions of the data. Stacking several convolutional layers on top of each other allows them to detect features at different

levels of detail. Lower layers extract simple information like shapes or colors, while deeper layers can recognize more complex patterns. This hierarchical structure learning makes CNNs well-suited for tasks such as image classification or semantic segmentation. However, stacking too many convolutional layers led to degraded performances, as the gradients backpropagated through these layers during the training became too small and vanished. This issue is addressed by introducing residual skip-connections in the ResNet architecture [1]. These shortcuts allow the input to a layer to skip one or more layers and be added directly to the output of a later layer, improving the gradient flow and enabling the training of deep NN. Building upon these advancements, EfficientNet [52] further optimizes the NN structure by introducing a compound scaling method that uniformly scales width, depth, and resolution of the network.

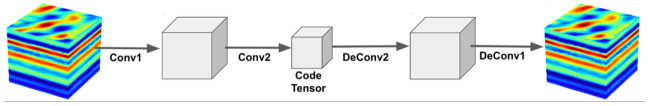



*Figure 2.1. Structure of an autoencoder model for the reconstruction of a flow field, with the encoder on the left side, the decoder on the right side, and the latent space in between. Taken from [53].*

The goal of autoencoder models [54] is to achieve compression by down- and subsequently up-sampling the input data. They consist of two main components: an encoder and a decoder, see Fig. 2.1. The encoder compresses the input data into a lower-dimensional representation, referred to as the latent space. The decoder then attempts to reconstruct the original input from this compressed representation. By forcing the network to reconstruct the input through this bottleneck, autoencoders learn to capture the most important features of the data while discarding less important ones. After the autoencoder has been trained, the low-dimensional representation of the input data in the latent space can then be used for the extraction of information.

In this thesis, CNNs are used within the CV and RS application cases (see Sec. 2.4.1 and Sec. 2.4.3), fully-connected NNs for the OpenML datasets (see Sec. 2.4.2), and an autoencoder model is used within the CFD application case (see Sec. 2.4.4).

# 3 Related Work

This chapter summaries the related work and introduces the current state-of-the-art algorithms in the field of large-scale DL in Sec.3.1 and in large-scale HPO in Sec. 3.2. The advancements beyond the related work are then presented in Sec. 3.3.

## 3.1 Efficient Large-Scale Deep Learning

Ever-increasing model- and dataset sizes require efficient methods for scaling the DL training. This thesis focuses on the data-parallel training techniques (see Sec. 2.2.2), as the GPU memory on current generation chips is large enough for holding common models used in the scientific application fields, but not large enough for holding the whole set of training data.
Several problems are associated with scaling distributed, data-parallel DL to a large number of workers. On the one hand, the computational efficiency needs to be kept high. It usually degenerates as the amount of computation, which involves the forward- and backward-passes of the NN, reduces in relation to the amount of communication, which involves the exchange and averaging of the gradients [7]. Initially, one relied on performing the averaging operation via a central parameter server for handling the communication of the gradients. The approach is however bottlenecked by the network bandwidth of the connection to the central parameter server. Instead, one has switched to the `RingAllReduce` approach[12], where each worker sends its chunk of data to an adjacent worker, which performs a reduction operation (sum, max, min etc.) locally and sends the chunk to the next worker. This alleviates the bandwidth bottleneck and is thus suitable for large setups, see Fig. 3.1. The algorithm is now also used by the NCCL library. By modifying the SGD optimizer, it is also possible to perform asynchronous communication [55], which is helpful in cases where the speed of the single workers is not consistent.
On the other hand, the degradation of the quality of the model performance (e.g., in terms of validation accuracy) is problematic. To keep GPU throughput high, the global batch size $BS_{global}$ of the NN training is increased linearly with the number of workers (see Eq. 8). This relationship stems from the mathematical requirements of distributed training under gradient averaging. Specifically, when using $N$ workers, each processing a local batch size $BS_{local}$, the global effective batch size becomes $BS_{global} = BS_{local} \times N$. This ensures that the distributed training remains mathematically equivalent to sequential training with a single large batch size. Experience has shown training with such large

[12] `RingAllReduce`: `https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/`

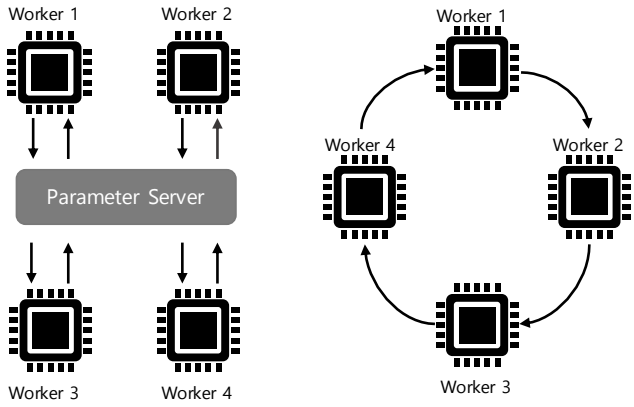



*Figure 3.1. Central parameter server approach (left) vs.* `RingAllReduce` *(right). Black arrows show communication between workers.*

batch sizes can result in much lower model scores and even in divergence [6, 56]. For example, on the ImageNet dataset [36] models start diverging when trained on $BS_{global} > 8000$ samples if no countermeasures are taken. This phenomenon occurs due to two primary factors: First, as $BS_{global}$ increases, the number of optimizer steps decreases, necessitating larger step sizes to reach a minimum within the same timeframe. To address this issue, several strategies have been proposed: employing learning rate schedulers that gradually increase the learning rate at the beginning (warm-up) and then decay it over time; scaling the learning rate with the number of workers (linearly or using square-root scaling) [57, 58]; or utilizing specialized optimizers such as LARS [59] or LAMB [60], which introduce layer-wise adaptive scaling mechanisms for the learning rate. Leveraging such techniques, it becomes possible to train on ImageNet with batch sizes ranging up to 80k samples [61]. The second reason for the decay in solution quality is that optimizers with large batch sizes tend to converge to sharp minima [62], which can negatively impact generalization [56]. In literature this has been addressed by performing a sufficient amount of HPO [63].
Another approach for efficient data-parallel training at scale without accuracy degradation involves adjusting the batch size itself during the training process. It has been shown empirically that increasing the batch size dynamically has a similar effect as decaying the learning rate over time while using fewer parameter updates [64]. A metric to predict when and how much to increase the batch size is the Gradient Noise Scale (GNS) [65]. It measures the amount of variance in the gradient in relation to the size of the gradient and can be approximated with sufficient precision following Eq. 11, where $G$ is the gradient and $\Sigma$ the covariance matrix of the gradient. The authors of [65] also propose a method on how to calculate the GNS without overhead during the data-parallel training.

$$GNS_{simple} = \frac{tr(\Sigma)}{|G|^2} \tag{11}$$

When using SGD with a small batch size, the gradient update is a noisy estimate of the true gradient. A big batch of data matches the true gradient much better. Following this intuition, the GNS can also be seen as a measure of the ratio of noise to signal of the gradient. A large noise-to-signal ratio indicates that a bigger batch size should be used and vice versa. Empirically it has been shown the GNS increases as the training progresses which is an indication to use larger batch sizes in the later part of the training run [65]. Different libraries, such as Pollux [66] or KungFu [67] already take the GNS and similar metrics as a basis for systematic optimization of the scheduling and throughput of DL models on HPC systems.
Apart from the efficient training, also the data loading plays an important role. As the datasets are at least several hundred GBs in size, an effective approach is necessary to prevent bottlenecks in the training process, which can occur when GPUs idle while waiting for data. Different libraries tackle this issue, e.g., the NVIDIA Data Loading Library (DALI) data loader[13] offers the option to directly perform preprocessing steps on the GPU instead of the CPU, eliminating the need to copy the data from CPU memory to GPU memory first. The FFCV library [68] uses a special data format to accelerate the reading of the data. Another factor is the format the data is stored in. Reading too many small files (such as millions of .jpeg images) can cause a strain on the underlying filesystem of the cluster. Therefore, formats that compress the data into one file (and support parallel I/O), such as HDF5[14] are more favorable.

### Mixed Precision

To increase the performance of the calculations on the single GPUs, techniques such as mixed precision calculations can be used. Usually, the parameters of NNs, e.g., during forward and reverse passes, are computed and stored using single precision (32-bit floating point or `FP32`). However, as the size of the model and the dataset increases, it was found that not all parameters require such high precision [69], potentially reducing computation time and disk space. One option is to use half-precision (16-bit floating point or `FP16`) calculations. However, since values smaller than $2^{-24}$ cannot be represented in `FP16` and already a few non-representable values cause divergence, a master copy of the weights is kept in memory in `FP32`. To detect non-representable values and to avoid propagation and deterioration of model accuracy, there is the Automatic Mixed Precision (AMP) training workflow with gradient scaling, see Algo. 1. In literature, it has been reported that models trained with AMP can even reach higher accuracy than their single precision counterparts, due to regularization effects of the lower precision [70].

### Frameworks

Several frameworks for efficient scaling of data-parallel training exist. Horovod is an open-source distributed DL library originally developed by Uber for *TensorFlow* [71] and was one of the first libraries to make use of the `RingAllReduce` approach. It is compatible with all major ML frameworks. PyTorch-DDP (Distributed Data-

[13] DALI: `https://developer.nvidia.com/dali`
[14] HDF5: `https://www.hdfgroup.org/solutions/hdf5/`

**Algorithm 1** PyTorch AMP Workflow Pseudocode

```
1: compute network forward pass in FP16
2: compute loss in FP32; scale by a large factor to ensure representability in FP16
3: compute backward pass in FP16 and check for NaN/Inf
4: if result contains no NaN/Inf then
5:     unscale gradients and update optimizer
6: else
7:     skip optimizer update
8: end if
```

Parallel) [6] is another option, mainly developed by Facebook AI Research at Meta. It uses a similar approach to Horovod for the reduction of the gradients and can work on top of different communication libraries (MPI, NCCL or Gloo[15]). The focus of Microsoft Research's DeepSpeed library [72] is on the training of massive models that do not fit into a single GPU. The ZeRO optimizer [73] distributes the training data, gradient, and model parameters across several GPUs to avoid memory redundancy and is thus a suitable choice if model- and data-parallel training is needed at the same time.

## 3.2 Distributed Large-Scale Hyperparameter Optimization

As HPO requires searching through a vast space of possible hyperparameter candidates to find optimal ones, this process is computationally expensive and there has been growing interest in reducing the runtime of the process. Two approaches exist to do so from the software side. The first one focuses on choosing better values for the hyperparameters, for an evaluation run, here called a "trial", by learning promising hyperparameter values from earlier runs. The second approach aims at reducing the runtime of HPO either by reducing the runtimes of the single trials by employing better scheduling techniques or by estimating the performance of a hyperparameter configuration with less than a full training run and terminating unpromising trials early on. From the hardware side, the HPO process can also be distributed to multiple devices to run in parallel. By using parallelism, multiple configurations can be evaluated concurrently, which significantly reduces the total runtime to find candidates that perform well.
The simplest way to perform distributed HPO is to use the Random Search or Grid Search approach [74]. In the Random Search case, a fixed number of configurations from the hyperparameter search space are sampled and evaluated randomly, while in the Grid Search case, every possible configuration is evaluated, see Fig. 3.2. As in both cases, each training run is independent of others, and apart from reporting the final performance of a configuration no communication between trials is required, this approach is highly parallel, but also computationally expensive. The convergence of

[15] Gloo: https://github.com/facebookincubator/gloo

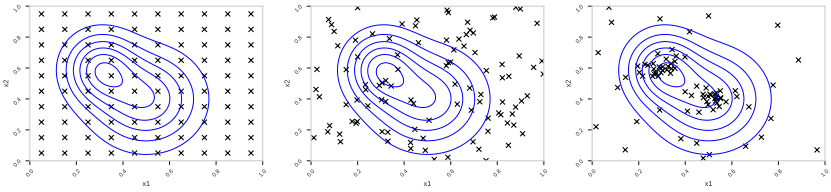


*Figure 3.2. Grid Search (left) and Random Search (center) and Bayesian Optimization (BO) (right) visualized for two hyperparameters $x_1, x_2$. A cross marks an evaluation of a hyperparameter configuration and the blue contours show regions with promising values.*

these methods relies on the principle that as the number of sampled configurations increases, the likelihood of identifying the optimal hyperparameter configuration also increases.

### Successive Halving and Hyperband

One approach to reduce the computational burden of Random Search is to approximate the objective function $f$ (see Sec. 2.3) by evaluating it with a lower fidelity, e.g., by running a trial with fewer epochs. Worse-performing trials are then cut off early and only the best-performing trials are allowed to continue training. This early stopping procedure is then repeated at multiple points in time and is known as Successive Halving [75]. It frames the HPO problem as a multi-armed bandit, where each arm represents a hyperparameter configuration. Pulling an arm then means training the configuration for a set number of iterations. The objective is to identify the arm with the highest payoff with a given budget $B$. Successive halving approaches this efficiently by first distributing an initial budget $B_0$ evenly across $n_{arms}$ configurations. After evaluating their performance at a milestone defined by $B_0/n_{arms}$ budget units, it eliminates a fraction $(1 - \frac{1}{\eta})$ of the poorest-performing configurations and continues training the remaining ones, by multiplying their computational budget by $\eta$. The reduction factor $\eta$ typically ranges from 2 to 4, see Fig. 3.3. This process creates a series of rungs, where each rung represents a selection and promotion cycle. Through successive iterations of this procedure across rungs, the algorithm gradually narrows down to a single configuration that demonstrated the best overall performance.

The theoretical foundation of successive halving rests on the assumption that eventually the losses of all sampled hyperparameter configurations $x_i, i = 1 \dots n_{arms}$ converge to terminal loss values $\lambda_i$. For any configuration, one can then define an envelope function $\gamma_i(\kappa_i)$ that represents the maximum possible deviation between the intermediate loss at budget $\kappa_i$ and the configuration's terminal loss $\lambda_i$. Assuming the first arm to result in the lowest loss, i.e. $\lambda_1 < \lambda_2 \leq \dots \leq \lambda_n$, this convergence property enables one to differentiate between two configurations $x_1$ and $x_2$ with different terminal losses already after budget $\tau$. Specifically, one can distinguish between them once their envelope functions $\gamma_1$ and $\gamma_2$ no longer overlap, which is once the widths of the envelopes become

smaller than the differences of the terminal losses $\frac{\lambda_2-\lambda_1}{2}$ [9]. This property ensures that successive halving will identify the optimal configuration, as long as no halving step is performed before budget $\tau$ that could discard configuration $x_1$. In practice, however, the convergence behaviour of the configurations is not known beforehand, making it difficult to choose how to allocate the total budget $B$. In cases where configurations are easily distinguishable early on, it is advisable to spend more budget on sampling a larger number of configurations $n_{arms}$. When configurations are hard to distinguish, it is more advisable to train configurations for longer, to avoid wrong terminations, at the cost of a reduced number of configurations.

The Hyperband (HB) algorithm [76] solves this issue of trading off the exploration of more hyperparameter combinations by initially launching more trials, with the deeper exploitation of the trials by training them for longer, by iterating over different numbers of initial trials in so-called brackets $s$. Given a total amount of resources $R$ to spend per configuration, the total number of brackets is computed as:

$$s_{max} = \lfloor \log_\eta(R) \rfloor \tag{12}$$

HB then loops over the different brackets, performing successive halving within each one, starting with $s = s_{max}$ as the most aggressive bracket, where exploration is maximized. The most conservative bracket $s = 0$ with the least amount of aggression is equal to pure Random Search. In the worst-case scenario, this is the only bracket that produces useful results, as the other brackets all terminated promising configurations too early. Theoretically, this makes HB a constant factor worse than plain Random Search in terms of runtime. Empirically it has been shown, however, to be much more efficient and to discover better solutions than Random Search when given the same amount of runtime [9].
When using HB, in most cases the number of epochs, training time or data subsets are used as fidelities. Some schedulers (HyperScher [76], Rubberband [77]) also treat the number of computational resources (such as the number of GPUs) as fidelity, but take advantage of the elasticity of the Cloud for allocation and de-allocation of these types of resources. On HPC systems, where the number of workers allocated to an HPO job usually stays fixed, this is not feasible.

### Bayesian Optimization Hyperband

As gradient-based optimization techniques are hard to apply in a diverse hyperparameter space [34], Bayesian Optimization (BO) has become one of the main methods in HPO to find the minimum of the objective function $f$. The idea behind BO is to use a probabilistic surrogate model of the objective function $f$ based on data points observed in the past. In the case of HPO, this means finding promising new hyperparameter configurations based on the performance of past trials. To discover new configurations, an acquisition function is used, and the following process is iterated:

1. Find the value that maximizes the acquisition function.
2. Evaluate the objective function at this maximum.
3. Include the new data point in the observation pool and refit the model.

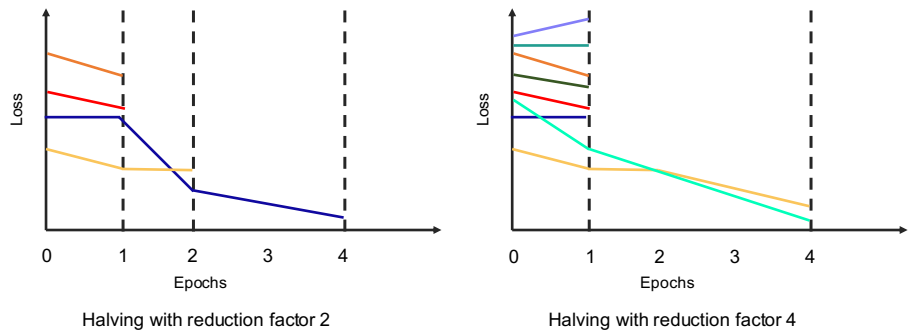



*Figure 3.3. Successive halving with different numbers of initial trials and different reduction factors. Dotted vertical lines represent decision points.*

BO can be combined with the HB scheduling method: In Bayesian Optimization Hyperband (BOHB), HB chooses the number of hyperparameter configurations and their assigned budget. For the BO part, BOHB employs a modified version of the tree parzen estimator [78]. The model is fitted once enough observations are available from trials running with the lowest budget. Combining BO and HB has several advantages: good results are achievable fast and on smaller budgets while on larger budgets the performance is better than other methods, such as plain HB.

### Asynchronous Successive Halving Algorithm

Asynchronous Successive Halving Algorithm (ASHA) [40] addresses the problem of large-scale HPO by improving the scalability of HB. To assess the most promising trials, HB waits for *all* trials to reach a certain threshold in time before applying successive halving. This leads to idling workers as some trials will be faster than others. ASHA circumvents this by deciding on a rolling basis which trials are promising. When two trials reach a successive halving point, the trial with the higher performance is allowed to proceed while the other trial is temporarily halted until the performance of the next completed trial can be juxtaposed. Even though this means some sub-optimal trials are promoted, the asynchronous comparison technique results in large speed-ups on up to 500 GPUs. As HB, ASHA concentrates solely on the scheduling aspects of HPO, and new hyperparameter configurations are randomly sampled. It inherits the theoretical convergence guarantees of successive halving and HB, is, however, in practice only configured to run with the three most aggressive brackets and for large-scale problems with the single most aggressive HB bracket [40]. ASHA also focuses not only on identifying the optimal hyperparameter configuration but also on delivering the fully trained model. In contrast to HB, ASHA requires as input not the total budget $B$, but instead a minimum $min_t$ and maximum $max_t$ resource amount to spend per configuration.

### Evolutionary Methods

Evolutionary methods in HPO seek to mimic the process of evolution and natural selection for finding optimal hyperparameters. At the beginning, an initial population of different ML models with randomly sampled hyperparameters is initialized and trained for a few epochs (a generation). Then the performance is measured and the models are ranked according to their results. In the next step, different genetic operations such as mutations are applied. In the case of mutation, the worst-performing trials copy the state and hyperparameters of the best-performing models and apply small perturbations to these parameters. This way, only the best-performing models continue training. The worst performing ones are early-stopped and their resources are reassigned to the perturbed configurations. One of the most commonly used frameworks for evolutionary HPO is the Population Based Training (PBT) algorithm, introduced in [79]. As multiple models are trained in parallel and the performance evaluation only takes place at certain points in time, the framework is highly scalable. Due to the iterative nature of the optimization process, the framework is also suitable for finding not just optimal steady hyperparameters but also unsteady series of hyperparameters, such as learning rate schedules. The early-stopping of sub-par configurations makes it additionally computationally efficient.

### Performance Prediction in Hyperparameter Optimization

Automatic performance prediction seeks to mimic the process of human experts developing ML models. By looking at the early results of the iterative training of NNs they can often tell if the current configuration of hyperparameters will converge to an optimal solution. If it does not, the expert terminates the training and starts a new training run with different hyperparameters. The main challenge is to make good predictions for the learning curve, based on a few evaluations [80]. One of the first approaches to automate the prediction was introduced in [81] where the combination of several probabilistic models is used to extrapolate learning curves, like the ones shown in Fig. 1.1 in Sec. 1.1. Integrated into HPO algorithms, it sped up the process of finding the optimal configuration of hyperparameters by a factor of 2. Based on Gaussian Processes, [82] introduces a BO method that halts unpromising models (based on the extrapolation of their learning curve) and trains more promising models instead. In [83], the authors use a Bayesian NN to perform learning curve extrapolation for different types of neural networks. The method is then integrated into another HPO method as a surrogate model for performance prediction. Another approach is taken in [84] where again with the help of BO the performance of neural networks trained on small parts of the training data is extrapolated to the full training set. The focus of [85] is computationally cheap regression methods to perform the learning curve extrapolation. The authors train linear, random forest, and support vector regressors on hyper and architectural parameters, as well as time series features derived from the learning curve of the neural networks. The results show that when training these regression methods with hundreds of full learning curves, they can accurately predict the final performance of a NN.

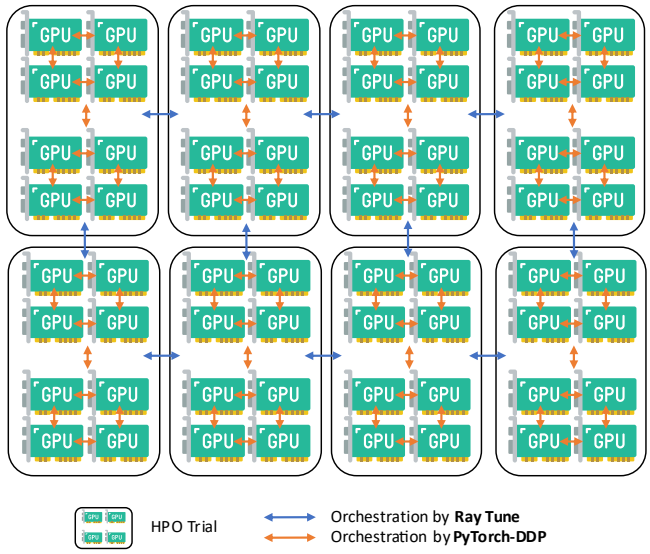



*Figure 3.4. Ray Tune in combination with PyTorch-DDP, performing distributed HPO with distributed DL.*

### Software Frameworks

Apart from the algorithms, several software frameworks for performing distributed HPO exist. One of the main frameworks that is used for HPO is the open-source library Ray[16]. Its subpackage Ray Tune [86] can run distributed HPO at scale. As HPO involves running a lot of trials with different sets of hyperparameters, allocating and launching each trial manually is inefficient. With Ray Tune, one head node and several worker nodes need to be launched via a SLURM script. The head node then connects to the worker nodes and launches the trials. During training, the worker nodes report their current status, including performance metrics, to the head node that terminates low-performing trials or launches new ones. The user specifies the number of resources to use per trial, the hyperparameters and their range, and a scheduling or optimization algorithm. The head node takes care of communication and scheduling tasks. Ray Tune can be integrated with other distributed DL frameworks, such as PyTorch-DDP or Horovod to not only parallelize the HPO loop but also the training of the single trials, see Fig. 3.4.
Other packages include Dragonfly [87] and SMAC [88], which both focus on scalable BO, including different surrogate models and acquisition functions. Optuna [89], which also mainly relies on BO for the optimization process features easy automatization and tracking of experiments.
From the HPC side, DeepHyper [90] puts a focus on asynchronous BO, which becomes necessary when running a large number of trials in parallel [91]. The computation of the BO surrogate model usually happens only on the head node of the distributed process, a

[16]Ray: `https://www.ray.io/`

bottleneck if the number of worker nodes is large. The adoption of an asynchronous communication approach and a local computation of the surrogate models on each worker node alleviates this bottleneck. DeepHyper has been applied to different scientific use cases on up to 1920 GPUs [92, 93, 91]. Under the hood, it relies on the MPI or Ray backend for communication. PENGUIN leverages a decoupled performance prediction method on HPC to avoid interfering with the search process for NAS workloads [94]. In [95], a pipeline for HPO which runs on HPC systems is introduced in the context of DL-based COVID-19 diagnostics. Propulate [96] is another HPC-ready HPO package running on MPI, with a focus on evolutionary optimization. However, none of the discussed frameworks put a special focus on integrating parallel HPO with distributed DL, a key necessity for handling the multi-TB-scale datasets addressed in this thesis.

## 3.3 Advancements Beyond Related Work

As systematic HPO involves the evaluation of multiple configurations in parallel, it is a problem that can easily be distributed to multiple workers. The majority of HPO algorithms in the literature are, however, only evaluated on a smaller number of CPU workers. This work presents one of the first systematic evaluations of the scalability of methods like HB on large scales of GPU workers in the context of **TO1**. Additionally, the performance of HPO methods is often only evaluated on standardized CV or smaller tabular benchmarking datasets, and more diverse scientific datasets are overlooked. This is addressed in the context of **TO4** in this thesis. As these scientific datasets are often large, even the evaluation of a single training epoch of the hyperparameter trials requires long runtimes. This work, in contrast to other methods, puts a focus on combining distributed DL with distributed HPO to accelerate the search for optimal hyperparameter combinations in the context of **TO2**. In that direction, one of the introduced algorithms, RASDA extends the classical ASHA method and treats the number of GPUs allocated to the data-parallel training loop of the single trials as a fidelity, which becomes necessary when applying HPO to problems that involve multi-TB-scale datasets and can exploit the inherent features of HPC systems. So far, also none of the existing algorithms put their main focus on leveraging novel accelerator capabilities for performing HPO. By evaluating the performance of successive halving and evolutionary-based HPO methods in combination with the AMP workflow and by presenting a performance prediction workflow for running HPO on a novel QA, this work aims to address these gaps in existing research in the context of **TO3**.

# 4 Summary of Publications

This chapter provides a summary of the publications underlying this doctoral thesis and puts them into the context of the thesis objectives.

In the following evaluations, the problem size is in most cases specified by fixing the total number of hyperparameter configurations to evaluate (Paper II, III, V, VI) or the total number of epochs to train for (Paper I, V). This allows for the computation of scaling metrics (see Sec. 2.1) across different numbers of workers and also makes comparisons between different HPC systems and types of hardware possible. This approach of defining problem sizes is also widely adopted in the HPC literature. This is different from evaluations in the HPO literature, where often the total wall-clock time is fixed and as many hyperparameter configurations as possible during that timespan are evaluated. From the user's perspective, both approaches of defining the problem size are valid since the end goal remains obtaining the best-performing model at the end of an HPO run. As the successive halving-based algorithms that are used in this thesis also exhibit strong anytime performance and focus on delivering favorable results in the short timespans [40], they are in general also applicable in settings where a maximum compute budget in terms of the wall-clock time is provided.

## 4.1 Paper I

**M. Aach**, E. Inanc, R. Sarma, M. Riedel, and A. Lintermann. "Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks." In: Journal of Big Data 10.1 (2023), p. 96, https://journalofbigdata.springeropen.com/articles/10.1186/s40537-023-00765-w

*This paper contributes to **TO1** in the sense that it evaluates the three main frameworks used for distributed DL, Horovod, PyTorch-DDP, and Deepspeed and two data loaders in an HPC setting on a CV benchmark and validates their performance at scale.*

The paper uses one of the main benchmarking cases in CV and DL, the training of ResNet models of different sizes on the ImageNet dataset. The performance and scalability of three different frameworks: Horovod, PyTorch-DDP, and DeepSpeed, and two data loaders (the native PyTorch data loader and the DALI data loader) are assessed on the JUWELS Booster machine at Jülich Supercomputing Centre on up to 1024 NVIDIA A100 GPUs.

The scalability results in terms of data throughput over the number of GPUs, are shown in Fig. 4.1. In essence, the DALI data loader demonstrates higher throughput in comparison to the native Pytorch data loader, regardless of the choice of DL framework. For DALI, there is not much difference in performance between PyTorch-DPP and Horovod. On the largest setup of 1024 GPUs, Horovod and PyTorch-DDP achieve a parallel efficiency of 0.76. For the native data loader, DeepSpeed shows the highest throughput on a small number of GPUs, but worse performance than Horovod and PyTorch-DDP on setups of more than 64 GPUs, with PyTorch-DDP showing the strongest performance on 512 GPUs. An in-depth analysis with the NSys profiler[17] reveals that, as expected, over time the communication in relation to the computation increases and that the data loading becomes an important factor, stressing the importance of using an efficient data loading tool.

Apart from the computational efficiency, also the impact of large batch training on the validation accuracy is examined. As the tuning of the learning rate is fundamental for the convergence of the training, three different learning rate schedulers are tested: a step-wise approach where the learning rate is decreased by an order of magnitude every 30 steps, which is in line with the original training recipe [1]. The cosine annealing schedule [97], in contrast, uses a smoother annealing that decreases the learning rate every epoch. The exponential annealing schedule starts out with a large learning rate and then rapidly decreases it in the beginning and more gradually afterwards.

All schedules use a similar warm-up of the learning rate during the first five epochs. On the ResNet50 training benchmark, the step-wise and cosine annealing schedules reach similar validation accuracy up to a batch size of 16k, see Fig. 4.2. On the largest batch size tested (64k = 1024 GPUs), however, the exponential annealing schedule seems the

[17] NSys profiler: `https://docs.nvidia.com/nsight-systems/index.html`

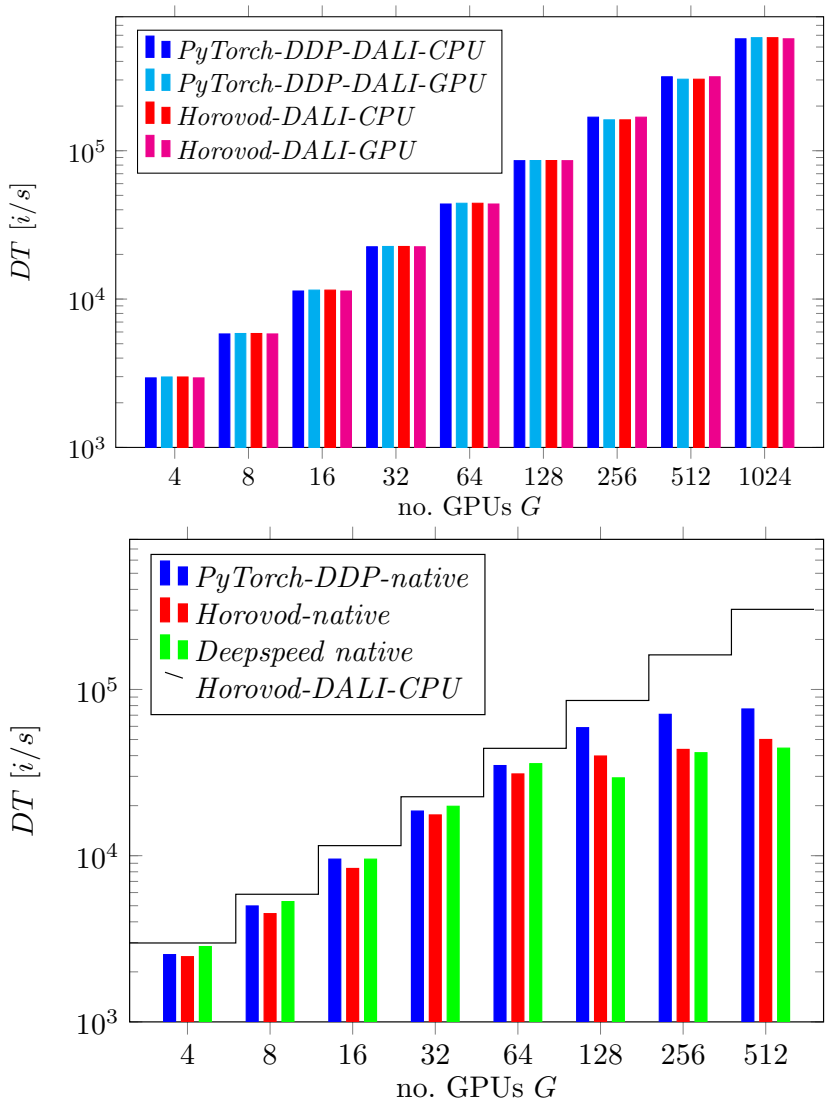



*Figure 4.1. Throughput DT in images per second $(i/s)$ of different frameworks and data loaders for the ResNet50 case, averaged over three experimental runs. Top: Throughput of Horovod and PyTorch with the DALI data loader. Bottom: Throughput of Horovod, PyTorch-DDP, and DeepSpeed with the native PyTorch data loader on raw ImageNet dataset, including comparison with Horovod-DALI-CPU throughput.*

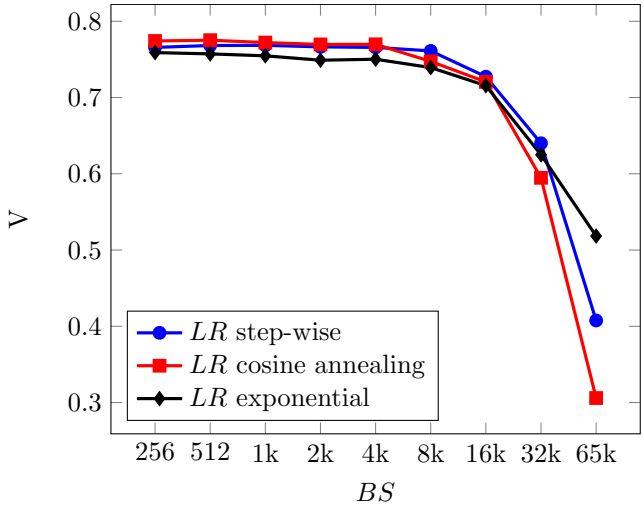



*Figure 4.2. Analysis of different learning rate schedulers for ResNet50 training, showing validation accuracy (V) over batch size BS.*

better choice, achieving a validation accuracy of $> 50\%$ while the others fall below that threshold. Still, the accuracy achieved in the smaller batch regime cannot be reached here.

## 4.2 Paper II

**M. Aach**, E. Inanc, R. Sarma, M. Riedel, and A. Lintermann. "Optimal Resource Allocation for Early Stopping-based Neural Architecture Search Methods." In: Proceedings of the Second International Conference on Automated Machine Learning (2023), https://proceedings.mlr.press/v224/aach23a.html

*This paper contributes mainly to **TO2**, as it investigates the empirical performance of different NAS algorithms on HPC systems and derives recommendations on how to optimally balance the parallelism between the distributed DL and distributed NAS/HPO loops. It also contributes to **TO1** in the sense that it uses a distributed combination of the Ray Tune and PyTorch-DDP libraries in the HPC environment.*

When using a fixed amount of parallel computing resources to perform NAS or HPO, there are two options on how to distribute these resources: On the one hand, state-of-the-art NAS algorithms are suitable for evaluating multiple configurations at the same time and this increases the likelihood of discovering well-performing ones. On the other hand, if more resources are allocated to the training of the single configurations (e.g., if more GPUs are used for the data-parallel training), results will be available faster. This paper empirically examines the total runtime of NAS workloads with the early stopping-based HB, ASHA, BOHB algorithms in general and with varying degrees of parallelism on NATS-Bench [98], a standardized evaluation framework for comparing different NAS algorithms. The search space for this benchmark is based on a CNN and features a total number of $8^5$ possible architectures, trained on the cifar-10, cifar-100, and imagenet-16 (a downscaled $16 \times 16$ pixel version of ImageNet) datasets.

The general speed-up of the algorithms when evaluating a fixed amount of 128 trials with one GPU per trial is examined in Fig. 4.3. As expected, the Random Search baseline performs best, with consistent parallel efficiencies $E_G > 0.9$. All other algorithms drop below 0.9 when using 32 GPUs in parallel and below 0.75 when using 64 GPUs in parallel. This is an indication that as this scale the available compute resources are not efficiently used anymore and it is advisable to not use more than 32 GPUs in parallel for a NAS run with the evaluated early stopping algorithms.

The results of using the early stopping-based NAS algorithms with varying degrees of parallelism are shown in Fig. 4.4. In total, 16 GPUs are available for a run, so if one GPU is used per trial that results in 16 trials running in parallel. If four GPUs are used (via data-parallel training) per trial that means only four trials can run at the same time. The plots show that when using **more GPUs per trial** this leads to a faster increase in validation score in the beginning, but also to a longer overall runtime for almost all cases (see green lines in Fig. 4.4). If just one GPU is used per trial, the consequence is a shorter overall runtime, but also a slower increase in the beginning (blue lines in Fig. 4.4). For the ASHA algorithm, a larger run on 64 GPUs is conducted, with up to 16 GPUs per trial used for data-parallel training. The results in Tab. 4.1 again confirm that using **fewer GPUs per trial** leads to much shorter runtimes (a factor of five, compared to using 16 GPUs per trial). A higher test set accuracy can also be observed.

This is likely because using more GPUs for data-parallel training also requires a larger global batch size, which can lead to lower validation performance. The results also hold true on different kinds of accelerators (e.g., on older NVIDIA or AMD hardware) and when using larger models and larger datasets (such as the complete ImageNet dataset).

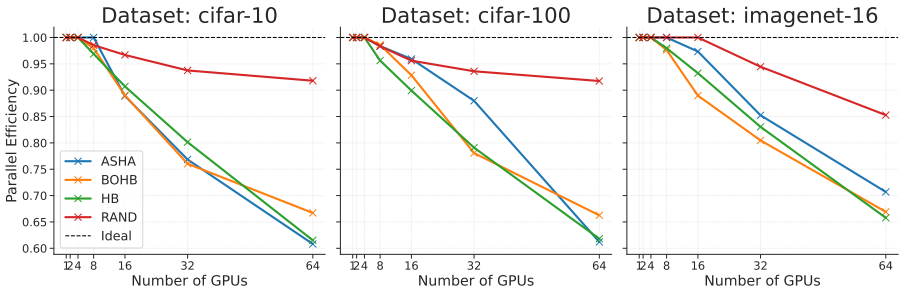



*Figure 4.3. Scalability comparison of Random Search, ASHA, BOHB, and HB. Parallel efficiency $E_G$ over the number of GPUs, based on the runtime until 128 samples are evaluated. $E_G = 1$ denotes the ideal case.*

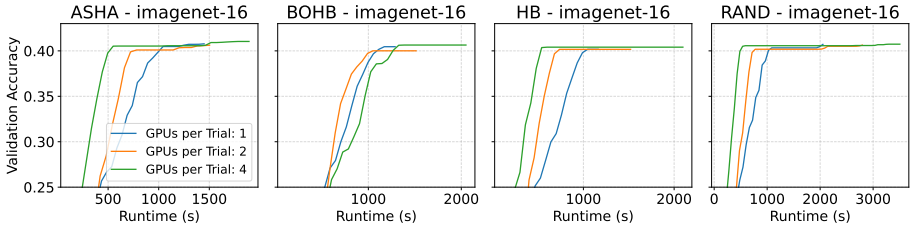



*Figure 4.4. Distributing trials across multiple GPUs, showing validation accuracy over runtime.*

*Table 4.1. Distributing trials across a total of* 64 *GPUs for ASHA on imagenet-16, measuring runtime until* 256 *samples are evaluated. Overview of runtime and test set accuracy achieved.*

| GPUs per trial | Runtime | Test set accuracy |
|---|---|---|
| 1 | 2,601 s | 43.4 % |
| 2 | 3,021 s | 42.7 % |
| 4 | 4,148 s | 42.3 % |
| 8 | 7,886 s | 42.9 % |
| 16 | 13,711 s | 41.4 % |

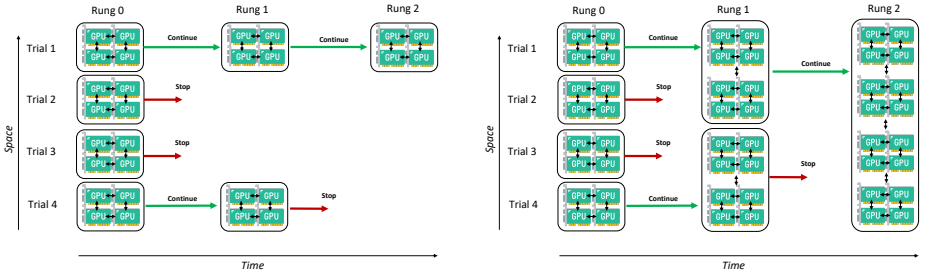
## 4.3 Paper III

**M. Aach**, R. Sarma, H. Neukirchen, M. Riedel, and A. Lintermann. "Resource-Adaptive Successive Halving for Hyperparameter Optimization on Large Datasets on High-Performance Computing Systems", submitted to Future Generation Computer Systems (2024), `https://arxiv.org/abs/2412.02729`

*This paper is the main contribution to **TO2**, as it presents the HPO scheduler facade suitable for optimizing DL on large datasets on HPC systems. It combines the concept of successive halving in time with successive doubling in space, allocating more GPUs to the data-parallel training of the more promising trials and thus accelerating their time to convergence. The approach is evaluated on several use cases on up to 1024 GPUs, also leveraging the insights from **TO1**. It is one of the first papers to apply systematic HPO to datasets on the multi-TB scale.*

Powerful HPC systems offer a natural setting for performing distributed HPO. As they usually feature a large number of accelerators (such as GPUs), many hyperparameter candidates can be evaluated in parallel at the same time. Apart from this *fast computation* feature, HPC systems are however also characterized by their network structure, which connects the different accelerators to facilitate *fast communication*. Most HPO methods only have modest communication requirements and neglect this inherent feature. To exploit this high-bandwidth communication to its fullest, two levels of parallelism are necessary. On the HPO level, multiple trials run in parallel, while on the level of the single trials, multiple GPUs are used at the same time to perform data-parallel training of the NNs. The new algorithm suggested in this paper, called RASDA combines the classical ASHA method for successive halving in time with a successive doubling routine in space, i.e. over time, it allocates more GPUs to the data-parallel training of the more promising trials, see Fig. 4.5. The data-parallel training involves the regular exchange of gradients of the models, which are usually large and thus require a fast and high-bandwidth network. As the calculation of when to perform a successive halving step is still based on the epoch numbers, and the data-parallel training accelerates only the wall-clock time of the trials, no interference with the successive halving procedure of ASHA is to be expected. The algorithm is applicable for problems that involve massive scientific datasets, where due to long training times, even with HPC resources just one round of trials is possible and users are interested in getting the best possible, fully-trained model in the shortest amount of time.

The algorithm is applied to NNs trained on three datasets from the CV, CFD, and Additive Manufacturing (AM) domain. All datasets are large in size, with the CFD dataset measuring a total of 8.3 TBs. A comparison with the plain ASHA algorithm shows that RASDA is able to accelerate the HPO process up to a factor of 1.9, reducing the runtime for HPO of an autoencoder for flow reconstruction trained on the CFD dataset from 325 to 170 minutes. Based on the insights of the GNS framework, the global batch size of the trials is gradually increased over time, in line with every doubling of compute resources, see Fig. 4.6. This gradual increase helps avoid the degradation of solution quality that is usually associated with large batch size training. Empirical




*(a) ASHA* *(b) RASDA*

*Figure 4.5. Comparison of (plain) ASHA, performing successive halving only in the time domain and RASDA, performing successive halving in the time and successive doubling in the space domain at the same time on a cluster. In the RASDA case, when a trial is terminated, its workers are allocated to the more promising trials to increase the parallelism of the data-parallel training. Black arrows indicate communication of gradients between GPUs.*

results even show that with the gradual batch size increase matching or even higher generalization performance on the test set can be observed, which is in line with findings in literature [64].

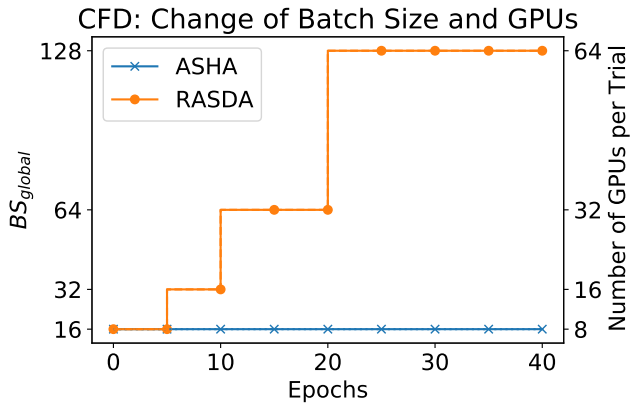



*Figure 4.6. Comparison of the global batch size and the number of GPUs per trial over for ASHA and RASDA on the CFD use case.*

As a last experiment, a large HPO run using RASDA method on 1024 GPUs is performed, evaluating 64 trials in parallel in three hours, with each trail starting with 16 GPUs. The run achieves a test set error of $4.88 \times 10^{-8}$ and a relative reconstruction error of just 0.016% and highlights the potential of large-scale HPO.

## 4.4 Paper IV

**M. Aach**, R. Sarma, E. Inanc, M. Riedel, and A. Lintermann. "Short Paper: Accelerating Hyperparameter Optimization Algorithms with Mixed Precision." In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23. New York, NY, USA: Association for Computing Machinery (2023), pp. 1776–1779., `https://doi.org/10.1145/3624062.3624259`

*The contribution of this paper lies in **TO3**, as the main outcome of this work is the exploitation and optimization of lower precision arithmetic on modern accelerators for the HPO process. Apart from that, the paper also applies the mixed precision computations to a scientific ML use case, see **TO4**.*

In recent years, it has been shown that it is not necessary to perform the computations related to NN training with double (`FP64`) or even single precision (`FP32`). Instead, on novel accelerators, it is possible to reduce the precision of most parameters to `FP16`. Mixed precision training has emerged as the most effective approach for implementing lower precision NN training. This method involves executing both the forward and backward passes predominantly in `FP16`, with only select operations remaining in higher precision. Performing only the forward pass in low precision and leaving the backward pass in single precision has shown to lead to mismatches [70]. Computing also the backward pass in lower precision is particularly advantageous from a performance perspective, as it is in general computationally more expensive than the forward pass, thus offering greater potential speed-ups. As leveraging the mixed precision workflow leads to faster training and less memory consumption, it has the potential to also reduce the runtime of the HPO process. However, most HPO and NAS algorithms make decisions on which trials to terminate and which to continue already based on observing a few epochs of training. Since the mixed precision workflow can change the training dynamics, it is not clear if these kinds of early stopping-based algorithms can safely be used for HPO. The methodology of mixed precision training is integrated into the PyTorch library with the PyTorch AMP package.

This work compares the performance of the ASHA, BOHB, and PBT algorithms with a Random Search baseline on two CV (ImageNet and cifar-10) datasets and CFD dataset. The process runs on 64 GPUs in parallel on the JURECA-DC-GPU machine. For the cifar-10 dataset, the NATS-Bench search space, based on an CNN architecture is used, while for the ImageNet dataset, a custom search space based on a ResNet with varying amounts of layers per stage is employed. The study on CFD utilizes a smaller version dataset of flow data from turbulent boundary layer simulations [49], training an autoencoder for flow reconstruction. This model comprises of an encoder and a decoder, each with four convolutional layers. The initial two layers in both the encoder and decoder are responsible for down- and up-sampling to achieve compression within the latent space. The remaining layers perform standard convolution. The model is taken from the AI4HPC repository [53] and while the architecture remains fixed, the hyperparameters learning rate, weight decay, (Nesterov) momentum, and number of

initial warm-up epochs are optimized. The mean squared error between the original and reconstructed flow fields measures the model's performance.

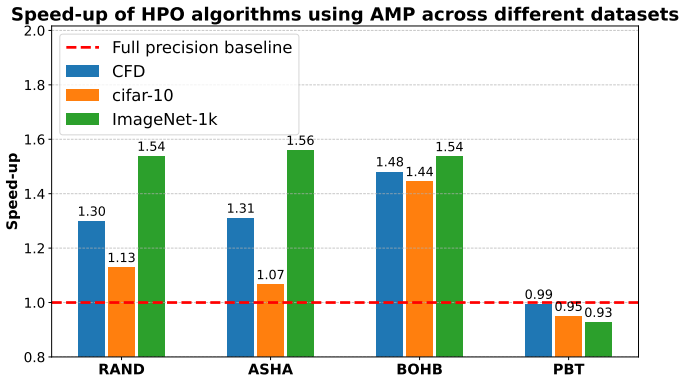



*Figure 4.7. Experimental results of running Random Search, ASHA, BOHB, and PBT with different datasets and search spaces.*

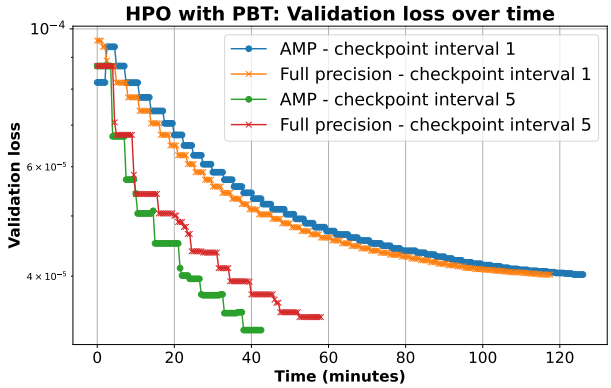



*Figure 4.8. Comparison of running PBT using different checkpointing intervals on the CFD dataset.*

The performance of coupling the AMP workflow with the different HPO algorithms is shown in Fig. 4.7. As expected, the largest speed-ups with a factor of up to 1.56 can be achieved on the ImageNet dataset. As it is the largest one in this study, this means also that the training is compute-intensive, which is where the AMP effect can be seen most drastically. Overall, ASHA gets the highest speed-up compared to the full precision baseline. Only for the PBT an actual reduction in performance can be observed. Upon examination, it was found that this is due to the frequent checkpointing that is required for evolutionary optimization. As the bottom fraction of trials (in terms of validation set performance) copies the state of the top fraction, this requires saving

| Dataset | Metric | ASHA | RAND |
|---|---|---|---|
| **CFD** | Runtime | 3046 s / 3992 s | 5253 s / 6825 s |
| | Test mse | $3.20 \times 10^{-3}$ / $3.31 \times 10^{-3}$ | $4.50 \times 10^{-3}$ / $4.50 \times 10^{-3}$ |
| **cifar-10** | Runtime | 3293 s / 3511 s | 3006 s / 3394 s |
| | Test acc. | 0.7798 / 0.7714 | 0.7677 / 0.7560 |
| **ImageNet** | Runtime | 10844 s / 16915 s | 6664 s / 10249 s |
| | Test acc. | 0.7006/0.6975 | 0.6758 / 0.6762 |

*Table 4.2. Comparison of running different HPO algorithms with (AMP/single precision) training, averaged over three random seeds. Better results are underlined.*

the model weights and transferring them to another accelerator to continue training with perturbations. This overhead offsets the performance gains of the AMP workflow. One way to increase performance for PBT is thus to increase the checkpoint frequency, see Fig. 4.8. When performing the checkpoint operation only every five epochs, a clear performance gain is visible. The performance of the best-found models for each dataset and algorithm in Tab. 4.2 on the test dataset also reveals that in some cases the AMP approach even reaches higher scores. This is due to the inherent regularization effect of mixed precision training and is an additional advantage of using AMP for HPO. It also showcases the importance of performing HPO with sophisticated algorithms, as the difference between test set error for e.g., ASHA and Random Search is more than 40% for the CFD use case.

## 4.5 Paper V

E. Wulff, J. P. Garcia Amboage, **M. Aach**, T. E. Gislason, TH. K. Ingolfsson, T. K. Ingolfsson, E. Pasetto, A. Delilbasic, M. Riedel, R. Sarma, M. Girone, and A. Lintermann. "Distributed Hybrid Quantum-Classical Performance Prediction for Hyperparameter Optimization", Quantum Machine Intelligence (2024), `https://link.springer.com/article/10.1007/s42484-024-00198-5`
(Note that the first three authors have equally contributed to Paper V)

*This paper contributes to **TO3** as it proposes a novel algorithm to couple a QA with a classical HPC system to perform efficient HPO at scale. The main idea of this paper is to highlight the feasibility of such a combination in the context of HPO workflows. The paper also adds onto **TO4** by evaluating the approach on different application domains.*

To improve the performance of the existing HPO method Hyperband (HB), this paper proposes a hybrid workflow that runs partial training of NNs with varying hyperparameters on multiple GPUs of a classical supercomputer, while predicting the hyperparameter configurations' performance with Support Vector Regression (SVR) on a QA. The workflow runs in a distributed fashion on 50 GPUs and completely automates the communication between the classical and the quantum systems.

Performance prediction can accelerate the HPO process by predicting the future performance of a model under a given set of hyperparameters based on the early results of the training, e.g., using a partial learning curve. In this case, the first few epochs of the training of the NN are used as training data for the surrogate model that is used as the performance predictor. Integration of quantum SVR as performance predictors into a hybrid HPO pipeline is possible by using the Swift-Hyperband [99] algorithm, which is based on the f-Hyperband algorithm [85] and follows the three-step workflow depicted in Fig. 4.9. Essentially, the QA is used for performing extrapolation of learning curves, based on the Quantum Trained Support Vector Regression (QT-SVR) method. This is motivated by the fact that in contrast to classical SVR, which uses a single deterministic prediction, the QA makes use of multiple, heuristically obtained predictions, which are weighted and combined into a single prediction. This ensembling of solutions has the effect of regularization and can therefore lead to more robust predictions [22].

The workflow is tested on the Extreme Scale Booster partition of the DEEP-EST supercomputer [100], utilizing 50 NVIDIA V100 GPU worker nodes in parallel. The quantum calculations take place on the JUPSI system. Among others, it is benchmarked on tabular datasets taken from the OpenML platform, focusing on tabular, high-quality datasets from different domains with 500 to 100,000 observations that are not trivially solvable by linear regression methods [101], such as the Naval Propulsion or Video Transcoding dataset. The search space for the models consists of two architectural parameters of NNs (depth and width) and three optimizer-related parameters (batch size, initial learning rate, and weight decay). The maximum amount of epochs a model can be trained for is fixed before the run.

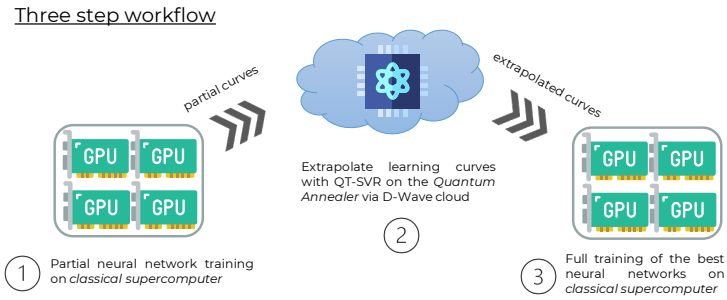
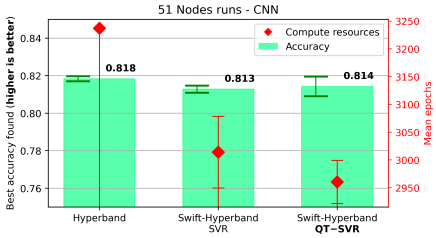
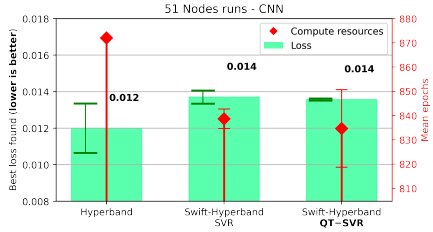



*Figure 4.9. Components of the hybrid quantum-classical HPO workflow.*

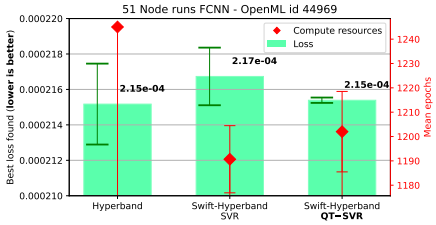
For performance evaluation, the distributed quantum-classical implementation of Swift-Hyperband is compared to the original (plain) HB algorithm and to Swift-Hyperband using classical SVR as performance predictors. In Fig. 4.10, the empirical results are shown on different NN architectures and datasets. The plots report the lowest loss (measured by mean squared error, where lower is better) or highest accuracy (where higher is better) achieved by each HPO algorithm in green along with the average resources consumed (in terms of the total number of training epochs) in red, using an average of three runs. It can be seen that Swift-Hyperband, both in the case of using SVRs and QT-SVRs, achieves similar target model loss as plain HB while consuming



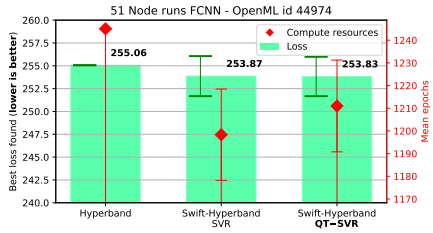

*(i) HPO on the cifar-10 dataset.*




*(ii) HPO on the TinyImagenet dataset.*




*(iii) HPO on the Naval Propulsion dataset.*




*(iv) HPO on the Video Transcoding dataset.*

*Figure 4.10. Comparison of different classical and quantum HPOs methods, showing mean epochs consumed in red, as well as best accuracy (i) and best loss (ii), (ii), (iv) found in green.*

$\sim$2-5% fewer computational resources. When comparing the different performance prediction methods used within Swift-Hyperband (SVR and QT-SVR), the quantum-based regression method is able to match the validation set loss (or accuracy) of the classical method in all cases, and in some cases even outperforms it (see Fig. 4.10). In general, these empirical results demonstrate the feasibility of hybrid quantum-classical HPO workflows.

### Remarks after Publication

This published work builds on the f-Hyperband algorithm [85] that combined HB with classical performance prediction. In the f-Hyperband paper, the authors also report isolated scores for the capabilities of several regression methods to predict the learning curves of several NN architectures. All regression methods are trained on 100 samples, with features consisting of hyperparameters, network architectures, and learning curves. As the QA, due to hardware constraints, is only able to handle a maximum of 20 samples for the training, the QT-SVR models were only trained with these 20 samples and features consisting of the learning curve. A comparison of the classical SVR and QT-SVR on these terms did not yield consistent results and the analysis is therefore not included in the paper. Still, the empirical evaluations of Swift-Hyperband show the QT-SVR-based performance predictors to match and occasionally outperform the classical SVR-based ones. This indicates that the QT-SVRs have a regularizing effect that prevents the erroneous termination of some promising trials [102].

## 4.6 Paper VI

**M. Aach**, R. Sedona, A. Lintermann, G. Cavallaro, H. Neukirchen, and M. Riedel. "Accelerating Hyperparameter Tuning of a Deep Learning Model for Remote Sensing Image Classification." In: IGARSS IEEE International Geoscience and Remote Sensing Symposium (2022), pp. 263–266, https://doi.org/10.1109/IGARSS46834.2022.9883257

*The main contributions of this paper are in **TO4**, as it presents an application of HPO to a scientific use case from the RS domain. Apart from that it also contributes to **TO1** and **TO2** as it uses a combination of the Horovod and Ray framework to efficiently perform HPO and distributed DL in an HPC setting.*

The focus of this work is using distributed DL and HPO for training an EfficientNet model [103] on the BigEarthNet dataset [48] with the goal of achieving an accurate land cover classification. BigEarthNet is a large dataset with more than 500,000 land cover patches. To accelerate the training of the models, data-parallel DL on 16 GPUs at the same time is employed. At the same time the hyperparameters: learning rate, momentum, weight decay, and Nesterov momentum are optimized, with six configurations training in parallel (resulting in total usage of 96 GPUs). To accelerate the training even further, the batch size is changed during the training. Starting with an initial global batch size of 512 samples (32 samples per GPU), this is increased to a global batch size of 16384 samples (1024 samples per GPU) after 20% of the training process. Through a modification of the training loop, this is achieved without having to save and reload the model weights, so without overhead. As shown in Fig. 4.11, the runtime per epoch is more than four times lower. The whole HPO run is three times as fast with the changing batch size approach, without notable loss in validation score for the best-performing trial (see Tab. 4.3). It is important to note that when training with a batch size of 16384 right from the beginning, the training diverges [104]. This stresses the importance of starting with a smaller batch size and increasing it after a certain number of epochs. In Fig. 4.12, different timings for performing the switch in batch size are explored empirically for the best-found hyperparameter combination. It can be seen that switching too early (e.g., after epoch 10 or 15) leads to a lower final validation score. Switching later than epoch 20 (e.g., in epoch 25) achieves the same validation score, but also results in a longer total runtime.

*Table 4.3. Runtime of the hyperparameter tuning and accuracy of the best-performing run for constant and changing batch size BS, showing accumulated and average trial runtime and validation F1 micro (macro) score.*

| $BS_{global}$ | total runtime | trial runtime | F1 scores |
|---|---|---|---|
| 512 | 27 hrs | 355 mins | 0.78 (0.72) |
| 512 →16,384 | 10 hrs | 136 mins | 0.78 (0.70) |

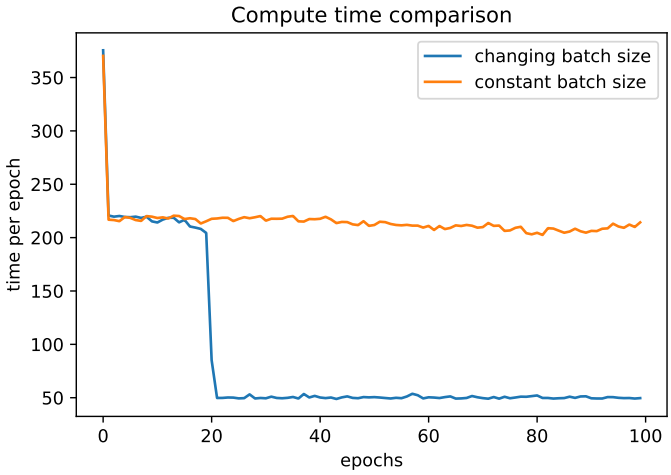



*Figure 4.11. Comparison of runtime per epoch over the number of training epochs. The changing batch size approach (after epoch 20) reduces the time per epoch.*

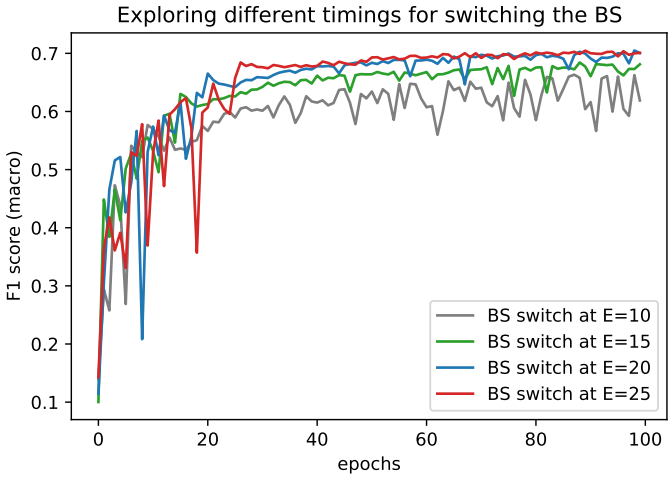



*Figure 4.12. Validation score curves throughout the training for different timings for the batch size switch.*

# 5 Conclusions

This chapter summarizes the findings of this thesis in Sec. 5.1 and provides an outlook for future work in Sec. 5.2.

## 5.1 Summary

As the performance of Neural Network (NN) is highly dependent on the choice of hyperparameters, systematic Hyperparameter Optimization (HPO) has become a crucial component of the Machine Learning (ML) pipeline. As lots of different hyperparameter combinations need to be evaluated to discover optimal ones, this process requires large amounts of computational resources. Especially researchers who work in domains that regularly deal with large scientific datasets and aim to train optimized Deep Learning (DL) models on them, are often faced with such computational challenges that make the application of systematic HPO methods infeasible. Modern High-Performance Computing (HPC) systems, comprising of a large number of accelerators and an optimized interconnect network structure, can however fulfill these requirements. While many HPO methods generally have put a focus on distributed performance and are often embarrassingly parallel by nature, their performance has so far not emphasized running in an HPC environment. This disconnect between HPO methods and HPC capabilities presents a significant opportunity for improvement in the field of ML. By developing HPO techniques that can fully harness the power of HPC systems, researchers could potentially explore much larger hyperparameter spaces, train more complex models, and work with datasets of unprecedented scale.

In this thesis, parallel and scalable HPO methods and workflows have been introduced, which are tailored towards running on these modern HPC systems. By efficiently leveraging two-level parallelism: on the HPO level by running multiple trials in parallel and on the single trial level by performing distributed DL among multiple Graphics Processing Unit (GPU), it has now become feasible to apply advanced HPO methods to large scientific datasets.
The developed approaches not only make HPO more accessible to a broader range of scientific applications but also significantly reduce the time and computational resources required for optimizing complex NN architectures. By harnessing the power of HPC systems, these methods enable researchers to explore vast hyperparameter spaces more efficiently than ever before. One of the key contributions presented in this thesis is the development of the Resource-Adaptive Successive Doubling Algorithm (RASDA) algorithm, which combines successive halving in the time domain with successive

doubling in the resource (space) domain. This novel approach allocates more GPUs to the data-parallel training of promising trials over time, effectively accelerating their time to convergence. The algorithm has shown significant improvements in HPO runtime, reducing it by up to a factor of 1.9 compared to plain Asynchronous Successive Halving Algorithm (ASHA), while maintaining or even improving solution quality. The research also addressed challenges associated with large batch training, which is often necessary when utilizing many GPUs in parallel. The strategy of gradually increasing the batch size over time was implemented and empirically validated. The method helps to avoid the degradation of solution quality typically associated with large batch size training, and in some cases, even leads to improved generalization performance. The performance of the RASDA method on the 8.3 Terabyte (TB) dataset from the Computational Fluid Dynamics (CFD) domain on 1024 GPUs offers a first example of the potential of such systematic large-scale HPO.

This thesis also investigated the application of Automatic Mixed Precision (AMP) training within the context of HPO. By leveraging lower precision arithmetic on modern accelerators, the research demonstrated that HPO runtime could be further reduced without compromising model quality. In fact, for some datasets, the use of AMP even resulted in better generalization performance, likely due to the inherent regularization effect of this approach. The thesis also proposed a novel hybrid quantum-classical approach for HPO. This method couples a Quantum Annealer (QA) with a classical HPC system to perform efficient HPO at scale. The quantum-based regression method showed promising results, matching or even outperforming classical methods in predicting model performance while consuming fewer computational resources.

In summary, the research question “Can we leverage the features of HPC for HPO?” was positively answered, following the accomplishment of the four thesis objectives. **TO1** for the evaluation of DL and HPO frameworks on HPC systems, including performance analysis of distributed DL frameworks like Horovod, PyTorch-DDP, and DeepSpeed on up to 1024 GPUs. **TO2** focused on developing a scheduler for distributed HPO that leverages distributed DL, resulting in the RASDA algorithm. **TO3** explored the exploitation of novel computing and accelerator paradigms, including mixed precision training and the hybrid quantum-classical algorithm. All developed approaches were verified by improving the performance of DL models trained on large scientific datasets from different domains, including the Remote Sensing (RS), CFD or Computer Vision (CV) domain within **TO4**. This work not only advances the field of HPO but also bridges the gap between cutting-edge DL techniques and computational challenges in scientific research, potentially accelerating progress across multiple disciplines.

## 5.2 Future Directions

By combining the aspects of distributed HPO and DL with HPC, the findings of this thesis offer several promising lines of future work. These can be categorized into three main areas:

- Application to an even broader range of scientific domains
- Evaluation on even larger-scale computing systems, such as Exascale machines
- Extension to other types of Artificial Intelligence (AI), such as generative models

Addressing the first point, this thesis has expanded the use of HPO methods to different fields of science, but as the developed methods are agnostic to the underlying type of DL model, they can be applied to any type of DL problem. A key direction for future work is therefore the application of the developed workflows to even more industrial and scientific fields that regularly deal with large datasets and see if similar results can be achieved. Examples include the healthcare domain, where large medical image datasets or genomics sequence databases are available, the autonomous driving domain where datasets of different sensor modalities are collected as well as astronomical observation, climate modeling, or particle physics data. In comparison to current methods in the field of AutoML, the RASDA method can effectively deal with such large datasets. As more and more researchers will be getting access to modern HPC clusters in the future, its is also of great importance to have open-source methods, such as RASDA available already now to run on these systems.

Regarding the second point, the RASDA method proposed in this thesis has already demonstrated promising scalability on a large HPC system. With the advent of Exascale HPC machines, which offer an order of magnitude more computational power, new possibilities emerge. Machines like JUPITER[18] that is currently in the build up phase will feature more than 24,000 GPUs. Scaling RASDA to such advanced systems opens up opportunities for optimizing scientific ML models at an unprecedented scale. On the one hand, the hyperparameter search space could be extended to include even more architectural parameters and a broad search for completely novel types of NN could be launched. On the other hand, by using such an amount of accelerators, many more hyperparameter configurations can be evaluated in parallel. As the scaling for RASDA from 128 GPUs to 1024 GPU, observed on the CFD dataset produced a model with a significantly lower error rate, one line of future work could explore if similar improvements could be made at even larger scale. At the same time, RASDA, currently a pure scheduling method, could also be combined with a searching method for further improvements. Further advancements in hardware developments, such as even lower precision arithmetic and larger Quantum devices could also be more explored, given the promising results presented in this thesis.

Finally, addressing the third point, a special focus for future research lies in the optimization of generative AI models, and so-called “Foundation Models” in specific. As

[18] JUPITER: `https://www.fz-juelich.de/de/ias/jsc/jupiter`

these kinds of models often measure several billion to trillion parameters in size, finding optimal architectural and optimizer-related hyperparameters is even more difficult. Due to the large amount of computational resources required for just a single training run, HPC systems are again necessary to accomplish such workloads. The training of Foundation Models also not only requires the use of data-parallel training but also other parallelization schemes (such as model-parallelism). Research on how to integrate this within the RASDA method, which in its current form is explicitly focused only on data-parallel training could bring further advancements. Also the timings of when during the training run to increase the global batch size (by the addition of GPUs) could be further explored. Currently, this follows the geometric resource allocation schedule of ASHA, but other insights based on the Gradient Noise Scale (GNS) framework could accelerate and optimize the training progress even further. In general, as the algorithms and workflows introduced in this thesis are already optimized for usage in an HPC setting, they are suitable candidates for application in the field of generative AI as well.

# Paper I

**Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks**

M. Aach, E. Inanc, R. Sarma, M. Riedel, and A. Lintermann

Journal of Big Data 10.1 (2023), p. 96

This article is an open-access article distributed under the terms and conditions of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/).

Marcel Aach wrote the code and the paper, ran all experiments on the HPC machines and performed the analysis.

Aach *et al. Journal of Big Data* (2023) 10:96
https://doi.org/10.1186/s40537-023-00765-w

Journal of Big Data

RESEARCH Open Access

# Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks

Marcel Aach[1,2*], Eray Inanc[1], Rakesh Sarma[1], Morris Riedel[1,2] and Andreas Lintermann[1]

*Correspondence:
m.aach@fz-juelich.de

[1] Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Wilhelm-Johnen-Straße, 52428 Jülich, Germany
[2] School of Engineering and Natural Sciences, University of Iceland, Reikjavik, Iceland

**Abstract**

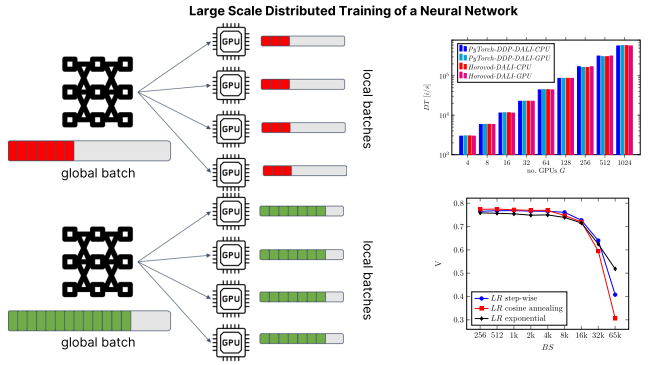
Continuously increasing data volumes from multiple sources, such as simulation and experimental measurements, demand efficient algorithms for an analysis within a realistic timeframe. Deep learning models have proven to be capable of understanding and analyzing large quantities of data with high accuracy. However, training them on massive datasets remains a challenge and requires distributed learning exploiting High-Performance Computing systems. This study presents a comprehensive analysis and comparison of three well-established distributed deep learning frameworks—*Horovod*, *DeepSpeed*, and *Distributed Data Parallel* by *PyTorch*—with a focus on their runtime performance and scalability. Additionally, the performance of two data loaders, the native *PyTorch* data loader and the *DALI* data loader by NVIDIA, is investigated. To evaluate these frameworks and data loaders, three standard ResNet architectures with 50, 101, and 152 layers are tested using the ImageNet dataset. The impact of different learning rate schedulers on validation accuracy is also assessed. The novel contribution lies in the detailed analysis and comparison of these frameworks and data loaders on the state-of-the-art Jülich Wizard for European Leadership Science (JUWELS) Booster system at the Jülich Supercomputing Centre, using up to 1024 A100 NVIDIA GPUs in parallel. Findings show that the *DALI* data loader significantly reduces the overall runtime of ResNet50 from more than 12 h on 4 GPUs to less than 200 s on 1024 GPUs. The outcomes of this work highlight the potential impact of distributed deep learning using efficient tools on accelerating scientific discoveries and data-driven applications.

**Keywords:** High-Performance Computing, Distributed deep learning, Performance analysis, Convolutional neural network, ImageNet

Springer Open

© The Author(s) 2023. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

**Graphical Abstract**




## Introduction

In the past few years, deep neural networks have become powerful tools to solve problems from different scientific disciplines. Especially in the field of image recognition, significant advancements have been made using Convolutional Neural Networks (CNNs) and Transformers [1, 2]. As the size of the datasets used for training and the model sizes continuously increase, their training becomes more and more computationally expensive. Therefore, it is of utmost importance to find suitable and efficient methods to reduce the training runtime by as much as possible.

Using High-Performance Computing (HPC) systems, it is possible to accelerate the training and model generation processes, i.e., by intelligently subdividing the problem and using massively parallel hardware for efficiently distributing the computational load. The two main strategies for distributing the training of neural networks to different workers, where a worker is usually a Graphics Processing Unit (GPU), are model and data parallelism [3]. The former method splits the neural network and distributes it across the workers. In contrast, the latter splits the input data and the network is trained with different batches per worker. At the end of an epoch, all gradients are merged to apply the same update to the network's weights on every worker. The Message Passing Interface (MPI) is frequently used to communicate the parameters between the workers in either a synchronous or asynchronous way. In the synchronous case, an `AllReduce` operation is executed for gradient reduction, while in the asynchronous case, a single central parameter server receives all gradient updates from the workers and performs the optimization step. For the asynchronous case, the overall performance is limited by the network bandwidth of the parameter server and depends on the amount of data to transmit per worker. The usage of alternative strategies such as asynchronous ring communications or employing MPI can alleviate this bottleneck. In general, with an increasing number of computational

nodes, the communication to share the gradients becomes the main bottleneck. To minimize this communication overhead, the number of gradient reductions has to be reduced.

The focus of this study is on data parallelism, as it offers benefits for neural networks of any size when trained on large datasets, whereas model parallelism is primarily advantageous for network architectures that do not fit onto a single GPU. Consequently, data parallelism holds an advantage in its ability to cater to a wider range of neural network architectures, whereas model parallelism is better suited for handling larger neural networks that face memory limitations. The standard approach to scale Deep Learning (DL) trainings on large HPC systems is to increase the global batch size *BS*, which may, however, lead to insufficient validation accuracies [4]. To retain a sufficient accuracy, modifications to the training process are frequently necessary, which may also affect the parallel performance. It is furthermore challenging to apply the right framework for a specific learning task that involves large input data or models, and at the same time, benefit from the computational power of HPC systems.

The motivation of this study is to provide guidance in this direction. To this end, the performance in terms of accuracy and scalability of the parallel frameworks—*Horovod* [5], *PyTorch-Distributed Data Parallel* (*PyTorch-DDP*) [6] and *DeepSpeed* [7] are evaluated on the European HPC system Jülich Wizard for European Leadership Science (JUWELS) [8] Booster module (place 12 in Top500 [9] as of 08/2022). These three frameworks are easily accessible as they are open-source and have gained high popularity in recent years. Their analysis supports the research community to decide on a framework. The benchmarks are performed using the popular ImageNet dataset [10] and three residual neural networks (ResNets) with 50, 101, and 152 layers [1] to enable generic comparison with established benchmark studies.

The main contributions of this study are as follows:

- A comprehensive scalability analysis and comparison of training ResNet50, ResNet101, and ResNet152 on ImageNet with *Horovod*, *PyTorch-DDP*, and *DeepSpeed*, using the *DALI* and native *PyTorch* data loaders ranging on up to 1024 GPUs is performed. The results demonstrate that in combination with the native *PyTorch* data loader, *DeepSpeed* shows the best performance on a small number of GPUs, while *Horovod* and *PyTorch-DDP* outperform it when using a larger number of GPUs. However, in comparison, the use of *DALI* leads to higher throughput and improves scalability for all ResNet architectures.
- An assessment of the influence of step-wise, cosine, and exponential learning rate annealing on the validation accuracy for different batch sizes ranging from $BS = 256$ to 65,536 is performed. These findings reveal that cosine annealing delivers superior performance on small and medium batch sizes, while exponential annealing achieves the highest accuracy for the largest batch size.

The paper is structured as follows. The established distributed DL frameworks as well as the challenges that arise when scaling the training to large HPC systems are discussed in "Related work" section. In "Overview of the benchmark setup" section, an

overview of the experimental setup is given. Subsequently, "Benchmark results" section presents the main benchmark results from a computational perspective. Finally, "Summary, conclusion, and outlook" section summarizes the findings, draws conclusions, and gives an outlook to future work.

## Related work

The ImageNet dataset, originally introduced for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), features 1,281,167 training images and 50,000 validation images divided into 1000 object classes [10]. This dataset is commonly used in the literature as an important benchmark to test algorithms, where their performance is measured through validation accuracy $V$ [11], which is computed from correctly classified validation images.

AlexNet [12] (a CNN) was one of the first machine learning models to achieve high accuracy on the ImageNet dataset in the ILSVRC 2012. Since then, multiple improvements have been made to the original network structure [13], yielding the current ResNet architecture [1] with a varying amount of convolutional layers. Although Transformer architectures [14] have recently been shown to achieve even better results in image classification tasks, CNNs are still widely used due to lower training duration, low energy usage, and their good scalability [15, 16].

Many distributed DL frameworks exist for scaling especially neural network models to multiple workers in an HPC environment. A literature review is given in [17] and an in-depth performance analysis is presented in [3]. One of the first frameworks to scale to a large number of computational nodes equipped with Central Processing Units (CPUs) was *DistBelief* by Google [18] using an asynchronous Stochastic Gradient Descent (SGD) method. Others, such as *Petuum* [19] or *Project Adam* [20] have improved this idea, e.g., by introducing dynamic scheduling. With *FireCaffe* [21], the focus shifted towards using GPU clusters and synchronous gradient reduction methods.

The most common DL libraries such as *TensorFlow* [22] from Google, *PyTorch* from Facebook, and *MXNet* [23] from Apache, have their unique distributed training strategies. Other frameworks such as *HeAT* [24] take a more general approach to distributed machine learning by providing a scalable *NumPy*-like [25] Application Programming Interface (API) to enable all kinds of data analytics, not being limited only to neural networks. While all of these frameworks mainly focus on enabling data parallelism, more recently Microsoft introduced the *ZeRO* [26] and *DeepSpeed* [7] frameworks that can train models with billions of parameters through model parallelism. A small scale study (up to four GPUs) of the performance of *TensorFlow* and *FireCaffe* on different HPC systems is available in [27]. An overview of the frameworks and their parallelism and communication strategies are shown in Table 1.

The present work solely focuses on synchronous communication and data-parallel functionality of the commonly used frameworks. *Horovod, PyTorch-DDP*, and *DeepSpeed* are all compatible with the HPC system's job scheduler SLURM [28] and the InfiniBand communication pattern by default, while the distributed versions of *MXNet* and *TensorFlow* are not.

In the literature, various tests using a ResNet50 on the ImageNet dataset exist. In the original ResNet paper [1], it takes 29 h to train the network for 90 epochs on eight NVIDIA Tesla P100 GPUs with a batch size of $BS = 256$. Subsequently, the training time is reduced to 1 h on 256 NVIDIA Tesla P100 GPUs [4]. Finally, in [29], a training time of only 74 s on 2048 NVIDIA Tesla V100 GPUs is achieved. The current benchmark record is 28.8 s as of 10/2022 [30], where TPUs have been used instead of GPUs. To train a ResNet50 in such a short time, the authors increase the batch size to $BS = 8192$ and $BS = 81{,}920$. A large $BS$ value, however, usually leads to a reduction in generalization performance. To prevent this, several hyperparameter modifications are applied in [29] to reach a $V$ comparable to that of [1]. Therefore, the learning rate $LR$ is scaled linearly with respect to the number of workers. This is motivated by the fact that with fewer weight updates, the learning rate has to be increased to achieve similar gradient adjustments compared to using a small learning rate with more frequent gradient updates. However, this linear scaling rule does not apply to all cases, i.e., during the start of the training, where a lot of parameters are subject to changes, a large learning rate may prohibit the optimizer from converging. Therefore, a warm-up technique is used that slowly increases the learning rate during the first five epochs [31]. Another technique reported to improve the accuracy when training with a large $BS$ is label-smoothing [32], which is a regularization method adapted to classification models. Using larger ResNet architectures, such as the ResNet101 or ResNet152, leads to slightly improved $V$ values [1].

**Code Snippet 1** Integration of *Horovod*

```
import torch
import horovod.torch as hvd

# Initialize Horovod
hvd.init()

# Define model and optimizer
model = models.ResNet50()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Add Horovod distributed optimizer
optimizer = hvd.DistributedOptimizer(optimizer,
            named_parameters=model.named_parameters())

# Broadcast model to processes
hvd.broadcast_parameters(model.state_dict(), root_rank=0)

# Start training loop
for epoch in range(100):
...

# Clean-up the parallel process group
hvd.shutdown()
```

In this work, the learning rate warm-up, learning rate scaling, and the label-smoothing techniques are used to stabilize the training with the default SGD optimizer with

**Table 1** Overview of distributed DL frameworks, adapted from [6, 17]

| Framework | Parallelism | Communication |
|---|---|---|
| *DistBelief* [18] | Model + Data | Asynchronous |
| *FireCaffe* [21] | Data | Synchronous |
| *Horovod* [5] | Model + Data | Synchronous |
| *MXNet* [23] | Model + Data | Bounded Asynchronous |
| *Petuum* [19] | Model + Data | Bounded Asynchronous |
| *TensorFlow* [22] | model + Data | Bounded Asynchronous |
| *PyTorch-DDP* [6] | Model + Data | Synchronous |
| *DeepSpeed* [7] | Model + Data | Synchronous |

Bounded asynchronous is a hybrid of synchronous and asynchronous communication

relatively large *BS* values. Additionally, three different learning rate schedules are explored and their performance in terms of *V* is analyzed.

## Overview of the benchmark setup

This section gives an overview of the main frameworks used in this study, i.e., *Horovod* is introduced in "Horovod" section, *PyTorch* in "PyTorch-distributed data parallel" section, *DeepSpeed* in "DeepSpeed" section, and the data loaders in "Data loaders and dataset compression" section. Furthermore, their communication operations are presented and examples of how to include them into actual Python code are provided. General issues that arise when scaling to a large amount of GPUs are addressed in "GPU scaling issues" section and the different ResNet architectures are introduced in "Residual neural networks" section. Three different learning rate scheduling methods with the potential of increasing the accuracy of the training are introduced in "Learning rate scheduling" section. The hardware and software configuration of the supercomputer used for the benchmark tests is presented in "JUWELS HPC system and software stack" section.

### Horovod

*Horovod* is an open-source distributed DL library originally developed by Uber for *TensorFlow* [5]. It is also supported as a backend library in the most common DL frameworks such as *PyTorch* and Apache *MXNet*. Minimal code changes are required to integrate *Horovod* into these DL frameworks. Code snippet 1 gives an example of how to integrate *Horovod* with *PyTorch*.

The work by Pumma et al. [33] provides an overview and an analysis of the communication patterns in *Horovod*. It is one of the first libraries to use a decentralized `Ring AllReduce` approach [34] to compute the gradient reduction instead of a single parameter server receiving all the updates, cf. "Introduction" section. It relies on low-level communication libraries such as *MPI*, the *NVIDIA Collective Communications Library (NCCL)* [35], or Facebook *Gloo* [36]. It is observed that the *NCCL* `AllReduce` yields superior performance on NVIDIA GPUs [6].

On a local worker level, the communication operations in *Horovod* are asynchronously handled by a separate background thread. This thread repeatedly checks for communication requests and performs the corresponding data transfers. Since these transfers may be requested asynchronously, the order of execution per worker may also be different. However, *Horovod* uses collective communication directives from other libraries and hence has to execute a consensus protocol to ensure consistency (in terms of order) across all workers. The process is summarized here:

(i) One global background thread receives all the transfer requests from the local background threads.
(ii) The global background thread puts the requests in the correct order and sends the list back to the local instances.
(iii) Each local thread combines its local data and carries out the data exchange with the other workers via `AllReduce`.

This back and forth communication creates overhead that can limit the scalabilty of the framework.

In *Horovod*, the computations and communications are coupled with the ability to batch small `AllReduce` operations. This exploitation of batching communication operations is known as tensor fusion [5]. With this operation, the smaller data volumes are transferred across different workers by locally fusing the data that are ready to be reduced. Hence, fewer `AllReduce` operations are required. In large neural networks with large number of parameters, this operation is expected to yield huge parallel performance gains.

#### PyTorch-distributed data parallel

*PyTorch* is a machine learning framework primarily developed by Facebook AI Research. The *PyTorch-DDP* module features a built-in way to parallelize the training of neural networks across multiple workers, e.g, GPUs. Code snippet 2 shows an example of how *DDP* in *PyTorch* is used. Similar to *Horovod*, the *PyTorch-DDP* library uses an `All-Reduce` paradigm (with the communication libraries *NCCL*, *Gloo*, or *MPI*) for updating the gradients used in deep neural networks. To trigger the communication operation, a custom 'hook' is registered in the internal automatic differentiation engine that is integrated into the backward pass operation of deep neural networks [6]. A separate code for managing the communication is hence not required.

Analogous to *Horovod*'s tensor fusion operation, *PyTorch-DDP* features gradient bucketing, where instead of an immediate `AllReduce` operation the algorithm waits for a few processor cycles once a batch of gradients is complete, and buckets (or 'fuses' in the sense of *Horovod*) multiple gradient parameters into a single parallel operation. Hence, the computation and communication are overlapped, thus skipping frequent gradient synchronization. A drawback of this method is a possible mismatch in the `All-Reduce` operation if the reduction order is not the same across all processes—resulting in an incorrect reduction or data inconsistencies. This issue is addressed by bucketing

the gradients in the reversed order obtained during the forward pass operation. This is motivated by the fact that the last layers of a network are likely the first ones to finish computation during the backward pass. Another issue is the skipped bucketed gradients that never enter the `AllReduce` operation. *PyTorch-DDP* handles this issue by a participation algorithm, which checks the output tensors during the forward pass to find all non-participated parameters (i.e., based on gradients that have not been updated) in the current iteration to be included in the next iteration.

**Code Snippet 2** Integration of *PyTorch-DDP*

```
import torch.distributed as dist
import torch.nn.parallel as par

# Initialize the parallel process group
dist.init_process_group(...)

# Define model and optimizer
model = models.ResNet50()
opt = optim.SGD(model.parameters(), lr=0.01)

# Distribute model to processes
dist_model = par.DistributedDataParallel(model)

# Start training loop
for epoch in range(100):
...

# Clean-up the parallel process group
dist.destroy_process_group()
```

#### DeepSpeed

The focus of *DeepSpeed* developed by Microsoft Research is on training large language models. These models usually feature several billion parameters and are trained on datasets from the natural language domain, which are significantly larger than most computer vision datasets. The main issue with these large language models is their massive memory footprint, a problem that is addressed with the *ZeRO* optimizer as part of *DeepSpeed*. This parallel optimizer eliminates memory redundancies by not only distributing the training data across workers but also the optimizer, gradient, and (if required) model parameters across workers. In contrast to the default data-parallel approach, the model is, therefore not necessarily replicated on each worker. Still, after each training step, an `AllReduce` communication step is necessary to ensure consistency. This is performed in a two-step process: first, different parts of the data are distributed to different workers with a `ReduceScatter` command, then each worker gathers the different chunks of data with an `AllGather` operation [26]. Code snippet 3 shows the integration of *DeepSpeed* within *PyTorch*, which is currently the only supported DL backend.

**Code Snippet 3** Integration of *DeepSpeed*

```
import torch
import deepspeed as ds

# Initialize the parallel process group
ds.init_distributed(...)

# Define model and optimizer
model = models.ResNet50()
opt = optim.SGD(model.parameters(), lr=0.01)

# Distribute model and optimizer
distrib_model, distrib_optimizer = ds.initialize(
                                    model=model,
                                    optimizer=optimizer,
                                    model_parameters=model.parameters())
# Start training loop
for epoch in range(100):
...

# Clean-up the parallel process group
ds.sys.exit()
```

#### Data loaders and dataset compression

Two types of data loaders and corresponding dataset compressions are compared in this work. One of the data loaders is the native *PyTorch* data loader, which uses the raw ImageNet data in the `JPG` format. This data loader only supports raw image data and performs all pre-processing steps on the CPUs. The other data loader is the *NVIDIA Data Loading Library (DALI)* [37], where a compressed version (`TFRecord`) of the ImageNet dataset is used. *DALI* is an open-source framework to accelerate the data-loading process in DL applications by involving the GPU, following a pipeline-based approach. Usually, the GPU runs computations much faster compared to the data-loading speed of the CPU. The idea of DALI is to prevent the GPU from starving by moving the data-loading process to the GPU at an early stage. The GPU then performs the data pre-processing, such as image resizing, cropping, and normalization on the fly. By pipe-lining these operations and executing them directly on the GPU, *DALI* minimizes the amount of data that needs to be transferred between the CPU and GPU, which reduces the overhead associated with these operations. *DALI* supports multiple data formats and with its unified interface, it is easy to integrate into all common DL frameworks. With this seamless integration developers can exploit the full potential of their GPU-based systems without having to modify their existing workflows significantly or switch between different data loading libraries. While the main focus of *DALI* is the GPU-based approach, it also offers the possibility to use the CPU for all steps of the pipeline. In this case also the pre-processing is performed on the CPU. Initial benchmarks show a speed-up between 20-40% in throughput compared to the original *PyTorch* data loader [38].

It should be noted that in terms of actual disk space, the difference between the compressed `TFRecord` version of the ImageNet dataset (144 GB) and the raw `JPG`

data (154 GB) is marginal. However, the file structure of the `TFRecord` dataset is much better suited for data loading in comparison to the over one million single image files in the raw dataset.

#### GPU scaling issues

GPU scaling in deep learning presents several challenges, including communication overhead, load balancing, and memory limitations. Communication overhead arises due to the constant synchronization and information exchange required between multiple GPUs during training. This overhead can reduce efficiency and performance as it grows with the number of GPUs. Solutions include using high-bandwidth, low-latency interconnects, and implementing efficient communication algorithms such as ring-based `AllReduce` methods [39, 40]. Load balancing is crucial for ensuring an even distribution of computational workload across all GPUs, maximizing resource utilization. An uneven workload distribution can lead to idle GPUs, wasting resources and increasing runtime. Dynamic load balancing algorithms and data or model parallelism techniques can help distribute tasks and data efficiently across multiple GPUs. Memory limitations pose a challenge when large models (or datasets) exceed a single GPU memory capacity, causing out-of-memory errors or forcing smaller batch sizes, which can negatively impact performance and convergence.

#### Residual neural networks

A prevalent challenge when training deep neural networks is the vanishing gradient problem, which leads to accuracy degradation [41]. ResNet architectures address this issue by introducing "skip-connections" that enable the training of deep networks without compromising accuracy. For larger vision datasets, ResNet50, ResNet101, and ResNet152 are the most widely adopted models. All models consist of one input layer and one fully-connected output layer but vary in the number of intermediate convolutional layers (48 vs. 98 vs. 150), see Fig. 1 for a visualization of the architecture. As a result, $3.8 \times 10^9$ floating-point operations per forward pass are needed for a ResNet50, $7.6 \times 10^9$ for a ResNet101, and $11.3 \times 10^9$ for a ResNet152. Although having more layers typically allows for the representation of more complex phenomena, it is essential to

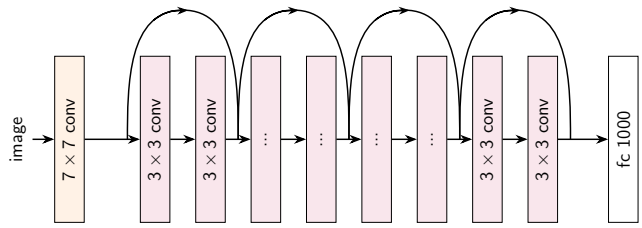



**Fig. 1** Standard ResNet architecture with one input layer (in orange), a varying number convolutional layers (in purple) and a fully-connected output layer (in white). Figure adapted from [1]

consider the trade-off between model complexity and training efficiency, as an increased number of floating-point operations leads to longer runtimes.

#### Learning rate scheduling

A known problem in large-scale distributed DL is the major drop in the validation accuracy $V$ when using a large $BS$ [4], regardless of the used framework, data loader, or optimizations (e.g., label-smoothing). A larger $BS$ yields fewer optimization steps, thus compromising the accuracy of the optimizer. This issue is one of the key challenges in distributed DL. To avoid divergence of the training process for $BS \geq 32k$ on ImageNet, further optimizations, such as $LR$ scheduling, are necessary. The most common approach for a $LR$ schedule on the ImageNet dataset is a stepwise annealing method [1]. This schedule reduces the $LR$ in regular intervals over time by an order of magnitude, i.e., in the ImageNet training, these intervals are set at epoch numbers 30, 60, and 80.

Since the $LR$ schedule plays an important role in the performance of a model, different approaches exist in the literature. The cosine-annealing schedule [42] is supported by *PyTorch* and uses the cosine function for smoother $LR$ annealing over time. In this case, the learning rate $LR_t$ at the current epoch $t$ is defined by:

$$LR_t = LR_{min} + \frac{1}{2}(LR_{max} - LR_{min})\left[1 + \cos\left(\frac{t}{t_{max}}\pi\right)\right], \tag{1}$$

where $LR_{min}$ and $LR_{max}$ are the minimum and maximum values of the learning rate, and $t_{max}$ is the total number of epochs used in the training. This decays $LR$ gradually at every epoch, compared to the sharp drops of the step-wise annealing. Another approach to $LR$ scheduling is the exponential decay schedule. Here, the $LR$ starts with a large value and is then decreased rapidly in the beginning and gradually afterwards in an exponential manner. The learning rate $LR_t$ at the current epoch $t$ is given by:

$$LR_t = LR_{max} * \gamma^t, \tag{2}$$

where the decay factor is usually set to $\gamma = 0.95$. The comparison in Fig. 8a shows that this scheduling method decreases the learning rate at similar orders of magnitude as the step-wise scheduler but in a smoother way.

#### JUWELS HPC system and software stack

The benchmarks presented in "Benchmark results" section are performed on the JUWELS HPC system. This system has a Modular Supercomputing Architecture (MSA) [43] and consists of a cluster and a booster module. The experiments use the GPU-based JUWELS Booster module, which consists of 936 compute nodes, each equipped with two AMD EPYC Rome 7402 CPU with 2x24 cores, clocked at 2.8 GHz, 512 GB Dynamic Random Access Memory (DRAM), and four NVIDIA A100 GPU with 40 GB High Bandwidth Memory (HBM). The GPUs communications in a compute node are performed via NVLINK [44]. The interconnect between compute nodes is a Mellanox InfiniBand HDR network with DragonFly+ topology. Each compute node has four HDR host channel

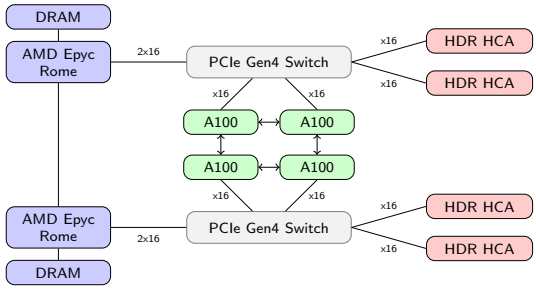



**Fig. 2** JUWELS Booster node schematic. Two AMD Epyc Rome CPUs are connected to four NVIDIA A100 GPU and the HDR HCAs via two PCIe Gen4 switches. The GPUs communicate via NVLINK

adapters. A Peripheral Component Inter-Connect Express (PCIe) Gen4 bus connects the components. In total, the JUWELS Booster is equipped with 3,744 GPUs and has 73 PFlops peak performance. Figure 2 shows the schematic of a single node.

The compressed and uncompressed ImageNet datasets are both stored on the *SCRATCH* partition of the JUWELS General Parallel File System [45]. This partition is optimized for the storage of large data and features a high Input/Output (I/O) bandwidth.

For running the experiments, the following software versions are deployed, which are available through JUWELS' EasyBuild [46] software system:

- `GCC 11.2.0`
- `OpenMPI 4.1.2`
- `Python 3.9.6`
- `CUDA 11.5`
- `PyTorch 1.11.0`
- `Horovod 0.24.3`
- `Deepspeed 0.6.3`
- `DALI 1.12.0` (virtual environment)

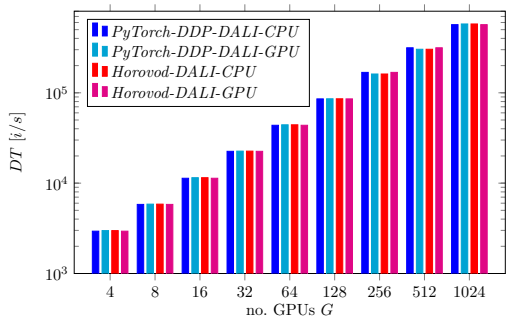
The number of CPU threads per data loader instance is set to 8.

### Benchmark results

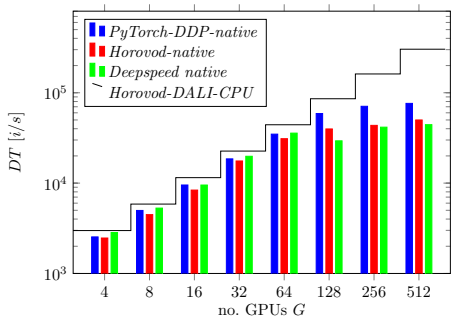
This section evaluates the performance of the three frameworks *Horovod*, *PyTorch-DDP*, and *DeepSpeed* with the native *PyTorch* and the *DALI* data loaders on the JUWELS Booster. The runtime $T$ of training a ResNet50, a ResNet101, and a ResNet152 on the ImageNet dataset for 90 epochs with a batch size of $BS = 64$ per GPU is measured and analyzed in "Efficiency" section. Additionally, the effect of three different learning rate schedulers on $V$ is explored in "Accuracy" section.

#### Efficiency

The results for the ResNet50 training in terms of data throughput $DT$, measured in images $i$ per second over the number of GPUs, are shown in Fig. 3. Overall, the *DALI*




(a) Throughput of *Horovod* and *PyTorch* with the ***DALI* data loader** CPU and GPU version on the **compressed** ImageNet dataset.




(b) Throughput of *Horovod*, *PyTorch-DDP*, and *DeepSpeed* with the **native** *PyTorch* data loader on **raw** ImageNet dataset, including comparison with *Horovod-DALI-CPU* throughput. The largest configuration only features 512 GPUs in this case as no significant additional speed-up is expected on larger configurations.

**Fig. 3** Throughput of different frameworks and *DALI* (**a**) and native (**b**) data loader for the **ResNet50** case, averaged over three experimental runs. The variance between runs is small (in general < 5%) and therefore not shown

data loader (Fig. 3a) achieves a higher throughput of images compared to the native *PyTorch* data loader (Fig. 3b), and this is observed to be independent of the distributed DL framework. For the native data loader, the three frameworks show similar performances up to 64 GPUs. For a smaller number of GPUs, *DeepSpeed* shows the highest *DT*, but performance drops for a larger number of GPUs, where *PyTorch-DDP* performs the best. In summary, for all three frameworks, it is evident that the native data loader cannot match the performance of the *DALI* data loader.

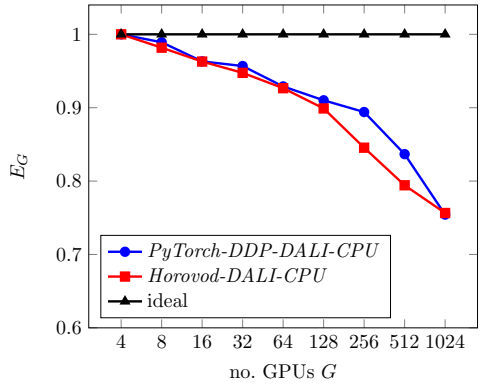
For a consistent scalability comparison, the parallel efficiency metric $E_G$ is calculated as

$$E_G = \frac{S_G}{G}, \tag{3}$$

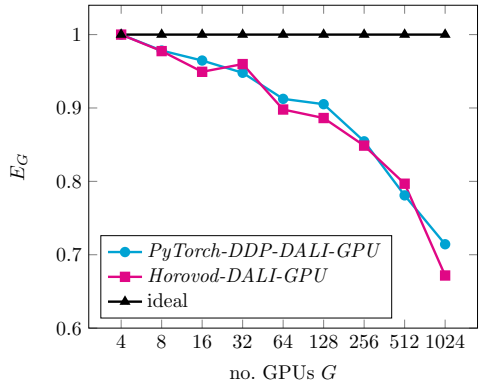
where $S$ is the speed-up and $G$ is the number of workers. The speed-up $S_G$ is computed as

$$S_G = \frac{T_4}{T_G}, \tag{4}$$

with reference runtime $T_4$, i.e., using four GPUs (one node on the JUWELS Booster). Note that $E_G$ of (or close to) unity is the ideal scenario with perfect scaling. The quantity $E_G$ is plotted in Fig. 4 for the *DALI* data loader over the number of GPUs. Note that




(a) Data loading and image pre-processing handled by the CPUs




(b) Data loading and image pre-processing handled by the GPUs

**Fig. 4** Parallel efficiency of *Horovod* and *PyTorch-DDP* on up to 1024 GPUs with the *DALI* data loader for CPU- (**a**) and GPU-based (**b**) pre-processing with compressed ImageNet dataset for the **ResNet50** case, averaged over three runs. Black line denotes the ideal case. The variance between runs is small (in general $< 5\%$) and therefore not shown

the *DeepSpeed* framework cannot use the *DALI* data loader. It is clear that the tested frameworks show similar scaling performances. Independent of the hardware acceleration (pre-processing on CPU in Fig. 4a or GPU in Fig. 4b) and the framework, the $E_G$ value remains above 0.65 up to 1024 GPUs. The training with *PyTorch-DDP* using the CPU for data input performs slightly better than its *Horovod* counterpart on 256 and 512 GPUs. With 1024 GPUs, the trainings using CPU for data input achieve a higher $E_G$ value of 0.76 compared to the ones using GPU, which is at $E_G = 0.68$. These findings show data loading with CPUs to be favorable for large-scale trainings. An analysis of the average CPU usages shows an occupancy of less than 40% across all configurations. It is assumed that the computationally strong host CPUs make up for any performance gains achieved by transferring the image pre-processing onto the GPUs. For hardware setups with less powerful host CPUs, using the GPU-based *DALI* version could still improve performance.

Figure 5 presents the parallel efficiency results for the native *PyTorch* data loader. When compared to the scalability of the *DALI* data loader (see Fig. 4), the scaling performance of the tested frameworks is considerably worse. The $E_G$ values of the training with *PyTorch-DDP* using the native data loader (denoted as *PyTorch-DDP-native* in Fig. 5) drops below 0.44 already with 256 GPUs, whereas *Horovod* and *DeepSpeed* have $E_G$ values of 0.50 and 0.33, respectively on 128 GPUs. With 512 GPUs, all of the frameworks achieve an $E_G$ value of less than 0.24, indicating the limitations of the data loader on parallelization. The superior performance of the *PyTorch-DDP-native* data loader could be due to its better compatibility with the *PyTorch-DDP* framework.

Fig. 6 shows the scaling performance of three ResNet architectures using the *DALI* data loader and *PyTorch-DDP* framework. The ResNet50, 101, and 152 model show similar $E_G$ values up to 16 GPUs. On larger configurations with more GPUs, the ResNet152 achieves the highest $E_G$ values, reaching 0.81 on 1024 GPUs. When more than 512 GPUs are utilized, the ResNet50 achieves the lowest $E_G$ values. This behavior is expected, as

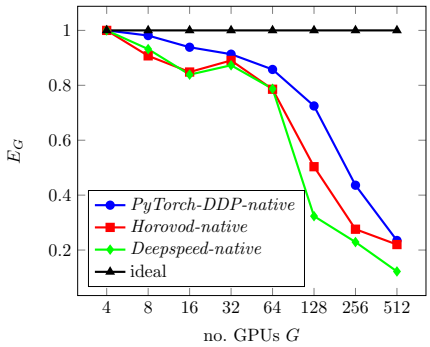



**Fig. 5** Parallel efficiency of *Horovod*, *PyTorch-DDP* and *DeepSpeed* on up to 512 GPUs with the native *PyTorch* data loader and raw ImageNet dataset for the **ResNet50** case, averaged over three runs. Black line denotes the ideal case. The variance between runs is small (in general < 5%) and therefore not shown

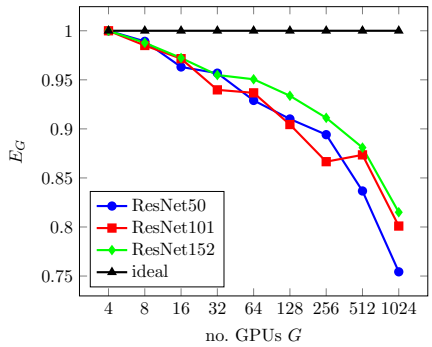



**Fig. 6** Parallel efficiency comparison of *PyTorch-DDP* on up to 1024 GPUs for **different ResNets** with *DALI* data loader (CPU-based) and compressed ImageNet dataset, averaged over three runs. The black line denotes the ideal case. The variance between runs is small (in general < 5%) and therefore not shown

a larger neural network means that more computation is necessary, which improves the computation to communication ratio and therefore also the scaling behavior. Nevertheless, the superiority of the *DALI* data loader over the *native* data loader can also be observed for the training with ResNet101 (see Fig. 7). On configurations with more than 32 GPUs, the *DALI* data loader clearly outperforms the *native* one in terms of scaling performance. Moreover, it is evident that *PyTorch-DDP* shows slightly better scaling performance than the *Horovod* framework (compare blue with green lines in Fig. 7).

For further verification of the results and a comparison of the *DALI* and native data loader, the NSys profiling tool [47] is used to analyze the amount of communication,

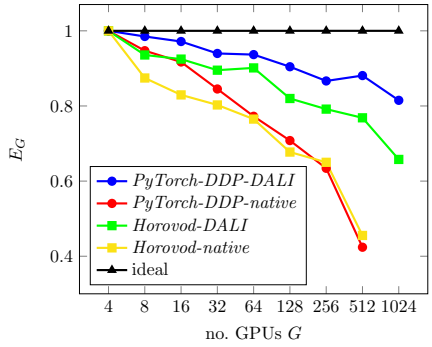



**Fig. 7** Parallel efficiency of *Horovod* and *PyTorch-DDP* on up to 1024 GPUs training a **ResNet101** with the *DALI* data loader and compressed ImageNet dataset and native *PyTorch* data loader and uncompressed ImageNet dataset, averaged over three runs. Black line denotes the ideal case. The variance between runs is small (in general < 5%) and therefore not shown

computation, and data loading that takes place during the training runs of all three ResNet architectures, where the results are shown in Table 2 in terms of three quantities. These are: (i) the runtime share by the Compute Unified Device Architecture (CUDA) kernels to perform 'communication'-related NCCL tasks (named communication), (ii) the runtime share to perform 'computation'-related tasks with the *cuDNN* library [48] (named computation), such as the time for the calculation of the convolutional layers, and (iii) the runtime share to execute 'input/output'-related tasks, such as data loading (named I/O).

For both types of data loaders, the time percentage spent on communication increases with the number of workers, while the efforts for the computation and data loading processes reduce. This behavior is expected when scaling up tasks that require frequent communication on a cluster while keeping the size of the dataset

**Table 2** Profiling CUDA kernel time in percent spent on communication operations via `AllReduce`, computations with the *cuDNN* library, and data loading functions

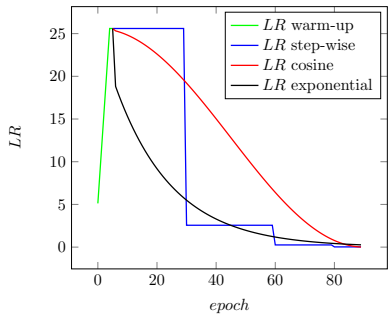
| No. GPUs | *PyTorch-DDP DALI* | | | *PyTorch-DDP native* | | |
|---|---|---|---|---|---|---|
| | **AllReduce [%] (Communication)** | **data[%] (I/O)** | ***cuDNN*[%] (Computation)** | **AllReduce [%] (Communication)** | **data[%] (I/O)** | **cuDNN[%] (Computation)** |
| (a) Training of ResNet50 on ImageNet | | | | | | |
| 4 | 15.40 | 22.00 | 32.50 | 22.40 | 21.00 | 30.80 |
| 8 | 19.00 | 21.40 | 31.75 | 23.95 | 20.05 | 29.20 |
| 16 | 21.00 | 20.95 | 30.70 | 27.15 | 18.83 | 27.35 |
| 32 | 27.09 | 18.98 | 28.14 | 31.30 | 17.26 | 25.11 |
| 64 | 30.87 | 17.76 | 26.35 | 32.75 | 16.30 | 23.55 |
| 128 | 33.61 | 17.03 | 24.99 | 49.48 | 11.77 | 17.33 |
| 256 | 37.08 | 15.78 | 23.26 | 76.77 | 5.06 | 7.14 |
| 512 | 43.48 | 13.57 | 20.02 | 82.61 | 3.66 | 5.52 |
| 1024 | 46.18 | 11.56 | 17.31 | – | – | – |
| (b) Training of ResNet101 on ImageNet | | | | | | |
| 4 | 13.30 | 23.00 | 46.00 | 28.65 | 22.50 | 38.12 |
| 8 | 20.55 | 21.25 | 41.45 | 30.15 | 18.28 | 35.52 |
| 16 | 24.08 | 20.37 | 39.65 | 35.67 | 16.76 | 32.71 |
| 32 | 25.36 | 18.71 | 36.99 | 35.46 | 14.59 | 28.43 |
| 64 | 37.17 | 16.69 | 33.39 | 37.69 | 15.31 | 29.88 |
| 128 | 36.29 | 16.74 | 34.02 | 42.32 | 13.39 | 26.38 |
| 256 | 39.31 | 15.54 | 31.56 | 56.43 | 11.38 | 22.83 |
| 512 | 37.73 | 15.40 | 31.59 | 59.18 | 11.87 | 24.45 |
| 1204 | 49.18 | 11.87 | 24.45 | – | – | – |
| (c) Training of ResNet152 on ImageNet | | | | | | |
| 4 | 16.20 | 22.40 | 44.60 | 18.41 | 21.97 | 44.17 |
| 8 | 20.55 | 21.75 | 42.35 | 20.65 | 21.95 | 40.75 |
| 16 | 25.90 | 20.05 | 39.07 | 24.77 | 20.70 | 38.62 |
| 32 | 29.16 | 18.72 | 37.15 | 30.31 | 18.77 | 35.32 |
| 64 | 33.56 | 16.90 | 33.82 | 38.34 | 16.42 | 30.80 |
| 128 | 36.16 | 16.66 | 33.73 | 45.75 | 14.02 | 26.60 |
| 256 | 38.33 | 15.51 | 31.60 | 49.39 | 15.05 | 28.46 |
| 512 | 40.36 | 14.41 | 29.43 | 51.76 | 11.16 | 25.36 |
| 1024 | 43.21 | 13.08 | 26.99 | – | – | – |

The first 10 epochs of the training process are profiled with the NSys Profiler (first five epochs for four GPUs due to time limits of the profiler)

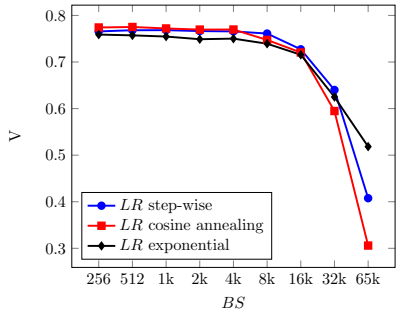
constant. For a smaller number of GPUs, the difference in time spent executing the `AllReduce` commands is similar for the *DALI* and native data loader methods. However, this difference increases rapidly with more workers. For example, in the ResNet50 case with 512 GPUs, the native data loader spends 82.61% of its time on communication, while the *DALI* data loader spends only 43.48%. A similar trend can be observed for the trainings with ResNet101 and ResNet152, where the native data loader spends 59.18 and 51.76% of its time on communication, compared to 37.73% and 40.36 for the *DALI* data loader, respectively. This substantial discrepancy could explain the poor scaling behavior of the native data loader. Regarding the computation time with the *cuDNN* library, it decreases for both data loaders as the number of GPUs increases, which is expected as the overall computational workload is distributed across a larger number of GPUs. For all three ResNet cases, the *DALI* data loader consistently exhibits higher computation percentages than the native data loader, suggesting that it effectively utilizes GPU resources. As for data loading, the time spent decreases as the number of GPUs increases for both data loaders. Although the relative data loading time is comparable between the two data loaders, it is important to emphasize that the *DALI* data loader is much faster in absolute timing. For example, in the ResNet152 case on 64 GPUs, the *DALI* data loader is responsible for 16.9% of the total runtime which amounts to $\approx 25s$ in absolute timing. For the native data loader case, the relative value is roughly the same with 16.42% of the total runtime, which, however, amounts to $\approx 47s$ in absolute timing. As expected, when comparing the three ResNet models it is evident that the communication overhead slightly reduces for smaller ResNet architectures, while the computation time increases as the size of the ResNet grows. Due to the low scaling performance of the native data loader, no evaluations on 1024 GPUs were performed for this case.

#### Accuracy

To investigate the issue of lower accuracy with a larger $BS$ value, the effect of different learning rate schedules on the learning rate $LR$ itself and $V$ are explored in Fig. 8 for the training with ResNet50. The three methods deployed in this case are thestep-wise, the cosine, and the exponential annealing methods, as described in "Learning rate scheduling" section. Fig. 8b depicts the evaluation of $V$ using the different learning rate schedules over growing batch sizes. All three scheduling methods have similar performances up to $BS = 4k$ (corresponds to 64 GPUs) with the exponential scheduling method being slightly worse than the others. At $BS = 8k$ (128 GPUs), the first significant drop of $V$ from $\approx 77\%$ to 74% is observed. From $BS = 16k$ (256 GPUs), the quantity $V$ drops consistently to lower values. For all three scheduling methods, there is a sharp drop of $V$ for $BS = 32k$ and $BS = 65k$ (512 and 1024 GPUs). The difference in $V$ is large for $BS = 65k$, where $V \approx 52\%$, $V \approx 41\%$, and $V \approx 31\%$ for the exponential annealing, step-wise, and cosine annealing scheduling methods, respectively. It is interesting to observe that the exponential scheduling method outperforms the cosine annealing for $BS = 32k$ and also the step-wise scheduling method for $BS = 65k$, even though exponential scheduling starts with the worst $V$ value even for $BS = 256$. It is




(a) Learning rate $LR$ variation over epochs using different learning rate scheduling methods including warm-up in first five epochs (on 1,024 GPUs).




(b) Validation accuracy $V$ for different learning rate schedulers with increasing batch size $BS$. Average over 3 runs.

**Fig. 8** Analysis of different learning rate schedulers for **ResNet50** training, showing learning rate over epochs (**a**) and validation accuracy over batch size (**b**). Note the original learning rate *LR* of 0.025 is scaled with the number of GPUs

evident that none of the scheduling methods can avoid the drop in $V$ for trainings with large $BS$ values, however, a training with the exponential learning rate schedule is the most favorable for large $BS$. Figure 9 depicts the validation accuracy curves over the number of epochs for an exemplary training of a ResNet50 with $BS = 65k$ with the three different learning rate schedules. While the learning curves show similar behaviour for the first 20 epochs, the exponential schedule outperforms the other two in the following 70 epochs by large margin.

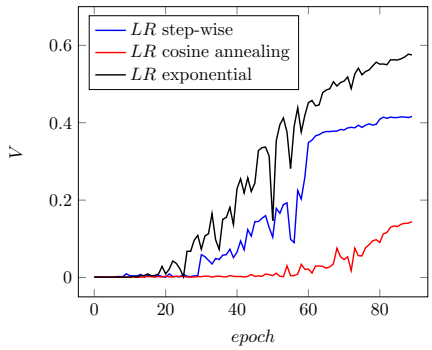



**Fig. 9** Validation accuracy $V$ over the number of epochs for a ResNet50 training with $BS = 65k$ on 1024 GPUs with different learning rate schedulers

## Summary, conclusion, and outlook

In this study, three distributed DL frameworks, i.e., *PyTorch-DDP*, *Horovod*, and *DeepSpeed* were analyzed in combination with the *DALI* and native *PyTorch* data loader on the JUWELS Booster module on up to 1024 GPUs in terms of data throughput, the runtime, and the scaling performance. For this analysis, the ResNet50, ResNet101, and ResNet152 architectures were trained on the ImageNet dataset. Furthermore, the impact of the batch size on the validation accuracy and the effect of different learning rate scheduling methods were investigated for training a ResNet50.

The superiority of the *DALI* data loader over the native framework-based data loader in terms of scaling performance was evident. A parallel efficiency of over 0.85 on up to 256 GPUs and over 0.75 on 1024 GPUs for training ResNet50 was achieved. This value was over 0.80 on 1024 GPUs for training ResNet101 and ResNet152. It can be concluded that *DALI* is well suited to be used in large-scale distributed machine learning setups, regardless of the underlying framework or size of the neural network. Comparatively, the native *PyTorch* data loader could only achieve an efficiency of 0.45 for a training with 512 GPUs, hence, an even lower number of GPUs.

As the global batch size has to be increased with the number of GPUs, the good scaling performance can only be reached with a large global batch size, leading to a reduction in validation accuracy of the training. Even though no solution exists to address this problem, this work has shown that some mitigation was possible through choosing the right learning scheduling methods. An exponential learning rate scheduling method showed the best performance in terms of validation accuracy for a large batch size of $65k$ on 1024 GPUs for training ResNet50, whereas for smaller batch sizes, the cosine or step-wise annealing scheduling methods achieved better accuracies.

Overall, the total training time was reduced from $\approx$ 13 h on 4 GPUs to $\approx$ 200 s on 1024 GPUs for training ResNet50 (234 times faster) and $\approx$ 17 h to $\approx$ 300 s for the Resnet152 case (204 times faster), respectively. This good scaling behavior proves the combined power of distributed DL and HPC when using the right tools and methods. Such short training times enable the developers to focus more on code and

hyperparameter tuning, leading in the end to better models. In summary, researchers should scale their training with the usage of *Horovod* or *PyTorch-DDP* to as much GPUs as possible, until the degradation of the accuracy sets in.

There exist other distributed DL frameworks that could be analyzed in terms of performance on large HPC systems. Furthermore, on the hardware side, different processor architectures that are tailored for machine learning, e.g., TPUs or Graphocore IPUs [49], are emerging and will undoubtedly play a key role in distributed DL. In the future, these developments will be further investigated.

The issue with the accuracy drop for large batch sizes also requires further attention. Other promising techniques apart from learning rate scheduling methods include novel optimizers, e.g., NVLAMB [50]. A comprehensive hyperparameter tuning routine, which includes other optimizer-related parameters, such as weight-decay rate or momentum can also impact the performance.

While this work evaluated the data loaders and frameworks already at large scale, further scaling tests are needed for even bigger systems, e.g., Exascale machines. Therefore, larger datasets and more complex models will be required. Other model architectures such as Autoencoders or Transformers have shown great success on various tasks and hence might be a good choice.

Future directions of general big data research include developing more efficient and adaptive distributed DL algorithms to handle heterogeneous data sources, reduce storage and communication overheads in HPC systems, and perform energy efficiency training of DL models on massive datasets.

**Abbreviations**

| | |
|---|---|
| API | Application Programming Interface |
| BS | Batch Size |
| CNN | Convolutional Neural Network |
| CPU | Central Processing Unit |
| CUDA | Compute Unified Device Architecture |
| DALI | NVIDIA Data Loading Library |
| DL | Deep Learning |
| DRAM | Dynamic Random Access Memory |
| GPU | Graphics Processing Unit |
| HPC | High-Performance Computing |
| ILSVRC | ImageNet Large Scale Visual Recognition Challenge |
| I/O | Input/Output |
| IPU | Intelligent Processing Unit |
| JUWELS | Jülich Wizard for European Leadership Science (JUWELS) |
| LR | Learning Rate |
| MPI | Message Passing Interface |
| MSA | Modular Supercomputing Architecture |
| NCCL | NVIDIA Collective Communications Library |
| PCIe | Peripheral Component Inter-Connect Express |
| ResNet | Residual Neural Network |
| SGD | Stochastic Gradient Descent |
| TPU | Tensor Processing Unit |
| V | Validation Accuracy |

**Acknowledgements**
Not applicable.

**Author contributions**
MA: writing code and paper, running experiments and analysis, EI: contributions to code and analysis, review, RS: contributions to code and analysis, review, MR: supervision, contribution to analysis, review, AL: supervision, funding, contribution to analysis, review. All authors read and approved the final manuscript.

**Funding**
Open Access funding enabled and organized by Projekt DEAL. The research leading to these results has been conducted in the CoE RAISE project, which receives funding from the European Union's Horizon 2020-Research and Innovation Framework Programme H2020-INFRAEDI-2019-1 under Grant Agreement No. 951733. The authors gratefully acknowledge the Partnership for Advanced Computing in Europe (PRACE) for funding this project by providing computing time on the Supercomputer JUWELS [8] at Jülich Supercomputing Centre (JSC). Open Access publication funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 491111487.

**Availability of data and materials**
The code to reproduce the experiments is available to the public via https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/resnet-benchmarks
The datasets generated and/or analysed during the current study are available in the ImageNet repository, https://www.image-net.org/download.php.

## Declarations

**Ethics approval and consent to participate**
Not applicable.

**Consent for publication**
Not applicable.

**Competing interests**
The authors declare that they have no competing interests.

Received: 3 November 2022 Accepted: 8 May 2023
Published online: 08 June 2023

**References**

1. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2016;pp. 770–778. https://doi.org/10.1109/CVPR.2016.90
2. Dosovitskiy A, Beyer L, Kolesnikov A, Weissenborn D, Zhai X, Unterthiner T, Dehghani M, Minderer M, Heigold G, Gelly S, Uszkoreit J, Houlsby N. An image is worth 16x16 words: Transformers for image recognition at scale. In: International Conference on Learning Representations (2021). arxiv:2010.11929
3. Ben-Nun T, Hoefler T. Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. ACM Comput Surv. 2019. https://doi.org/10.1145/3320060.
4. Goyal P, Dollár P, Girshick R, Noordhuis P, Wesolowski L, Kyrola A, Tulloch A, Jia Y, He K. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour 2018. arXiv:1706.02677
5. Sergeev A, Balso M.D. Horovod: fast and easy distributed deep learning in TensorFlow 2018. arXiv:1802.05799
6. Li S, Zhao Y, Varma R, Salpekar O, Noordhuis P, Li T, Paszke A, Smith J, Vaughan B, Damania P, Chintala S. PyTorch distributed: experiences on accelerating data parallel training. Proc VLDB Endow. 2020;13(12):3005–18. https://doi.org/10.14778/3415478.3415530.
7. Rasley J, Rajbhandari S, Ruwase O, He Y. DeepSpeed: system optimizations enable training deep learning models with over 100 billion Parameters, pp. 3505–3506. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3394486.3406703
8. Jülich Supercomputing Centre. JUWELS: Modular Tier-0/1 Supercomputer at Jülich Supercomputing Centre. J Large Scale Res facil JLSRF. 2019;5:135. https://doi.org/10.17815/jlsrf-5-171
9. Top500. https://top500.org/lists/top500/list/2022/06/. Accessed: 2022-09-20
10. Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, Huang Z, Karpathy A, Khosla A, Bernstein M, Berg AC, Fei-Fei L. ImageNet large scale visual recognition challenge. Int J Comput Vision (IJCV). 2015;115(3):211–52. https://doi.org/10.1007/s11263-015-0816-y.
11. Mattson P, Cheng C, Diamos G, Coleman C, Micikevicius P, Patterson D, Tang H, Wei G-Y, Bailis P, Bittorf V, Brooks D, Chen D, Dutta D, Gupta U, Hazelwood K, Hock A, Huang X, Kang D, Kanter D, Kumar N, Liao J, Narayanan D, Oguntebi T, Pekhimenko G, Pentecost L, Janapa Reddi V, Robie T, St John T, Wu C-J, Xu L, Young C. Zaharia, M. Mlperf training benchmark. In: Dhillon, I., Papailiopoulos, D., Sze, V. (eds.) Proceedings of Machine Learning and Systems. 2020;vol 2:336–49.
12. Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. Commun ACM. 2017;60(6):84–90. https://doi.org/10.1145/3065386.
13. Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Bach F, Blei D. (eds.) Proceedings of the 32nd International Conference on Machine Learning. Proceedings of Machine Learning Research 2015: vol. 37, pp. 448–456. PMLR, Lille, France.
14. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Lu, Polosukhin I. Attention is all you need. Advances in Neural Information Processing Systems. 2017;30.
15. Li D, Chen X, Becchi M, Zong Z. Evaluating the energy efficiency of deep convolutional neural networks on cpus and gpus. In: 2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom), pp. 477–484 2016. https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.76.

16. Strubell E, Ganesh A, McCallum A. Energy and policy considerations for deep learning in NLP. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pp. 3645–3650. Association for Computational Linguistics, Florence, Italy 2019. https://doi.org/10.18653/v1/P19-1355
17. Langer M, He Z, Rahayu W, Xue Y. Distributed training of deep learning models: a taxonomic perspective. IEEE Trans Parallel Distributed Syst. 2020;31(12):2802–18. https://doi.org/10.1109/tpds.2020.3003307.
18. Dean J, Corrado G, Monga R, Chen K, Devin M, Mao M, Ranzato, MA, Senior A, Tucker P, Yang K, Le Q, Ng A. Large scale distributed deep networks. Advances in Neural Information Processing Systems. 2012;25.
19. Xing EP, Ho Q, Dai W, Kim JK, Wei J, Lee S, Zheng X, Xie P, Kumar A, Yu Y. Petuum: a new platform for distributed machine learning on big data. IEEE Trans Big Data. 2015;1(2):49–67. https://doi.org/10.1109/TBDATA.2015.2472014.
20. Chilimbi T, Suzue Y, Apacible J, Kalyanaraman K. Project Adam: building an efficient and scalable deep learning training system. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. OSDI'14, 2014: pp. 571–582. USENIX Association, USA.
21. Iandola FN, Moskewicz M, Ashraf K, Keutzer K. FireCaffe: near-linear acceleration of deep neural network training on compute clusters. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2592–2600 (2016). arxiv:1511.00175
22. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, et al. MD. TensorFlow: large-scale machine learning on heterogeneous systems (2015). arxiv:1603.04467
23. Chen T, Li M, Li Y, Lin M, Wang N, Wang M, Xiao T, Xu B, Zhang C, Zhang Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems 2015. arxiv:1512.01274
24. Götz M, Debus C, Coquelin D, Krajsek K, Comito C, Knechtges P, Hagemeier B, Tarnawa M, Hanselmann S, Siggel M, et al. Heat—a distributed and gpu-accelerated tensor framework for data analytics. 2020 IEEE International Conference on Big Data (Big Data) 2020. https://doi.org/10.1109/bigdata50022.2020.9378050.
25. Numpy. https://numpy.org/. Accessed 20 Sep 2022.
26. Rajbhandari S, Rasley J, Ruwase O, He Y. Zero: Memory optimizations toward training trillion parameter models. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–16 (2020). https://doi.org/10.1109/SC41405.2020.00024
27. Shams S, Platania R, Lee K, Park S-J. Evaluation of deep learning frameworks over different hpc architectures. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 1389–1396 (2017). https://doi.org/10.1109/ICDCS.2017.259
28. SLURM. https://slurm.schedmd.com/. Accessed 20 Sep 2022.
29. Yamazaki M, Kasagi A, Tabuchi A, Honda T, Miwa M, Fukumoto N, Tabaru T, Ike A, Nakashima K. Yet another accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds 2019. arxiv:1903.12650.
30. Kumar S, Bradbury J, Young C, Wang YE, Levskaya A, Hechtman B, Chen D, Lee H, Deveci M, Kumar N, Kanwar P, Wang S, Wanderman-Milne S, Lacy S, Wang T, Oguntebi T, Zu Y, Xu Y, Swing A. Exploring the limits of concurrency in ML training on Google TPUs. 2021. arxiv:2011.03641
31. Krizhevsky A. One weird trick for parallelizing convolutional neural networks. 2014. arxiv:1404.5997
32. Szegedy C, Vanhoucke V, Ioffe S, Shlens J, Wojna Z.D Rethinking the inception architecture for computer vision. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2818–2826 (2016). https://doi.org/10.1109/CVPR.2016.308.
33. Pumma S, Buono D, Checconi F, Que X, Feng W-C. Alleviating load imbalance in data processing for large-scale deep learning. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), pp. 262–271 (2020). https://doi.org/10.1109/CCGrid49817.2020.00-67.
34. Gibiansky A. Bringing HPC techniques to deep learning 2017. https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/. Accessed 31 Aug 2021.
35. NCCL. https://developer.nvidia.com/nccl. Accessed 20 Sep 2022.
36. Gloo. https://github.com/facebookincubator/gloo. Accessed 20 Sep 2022.
37. DALI. https://developer.nvidia.com/dali. Accessed 20 Sep 2022.
38. Zolnouri M, Li X, Nia V.P. Importance of data loading pipeline in training deep neural networks 2020. arxiv:2005.02130.
39. Wang G, Lei Y, Zhang Z, Peng C. A communication efficient ADMM-based distributed algorithm using two-dimensional torus grouping AllReduce. Data Sci Eng. 2023;1–12.
40. Zhou Q, Kousha P, Anthony Q, Shafie Khorassani K, Shafi A, Subramoni H, Panda DK. Accelerating MPI all-to-all communication with online compression on modern GPU clusters. In: High Performance Computing: 37th International Conference. ISC High Performance 2022. Hamburg, Germany: Springer; 2022. p. 3–25.
41. Bengio Y, Simard P, Frasconi P. Learning long-term dependencies with gradient descent is difficult. IEEE Trans Neural Netw. 1994;5(2):157–66. https://doi.org/10.1109/72.279181.
42. Loshchilov I, Hutter F. SGDR: Stochastic gradient descent with warm restarts. International Conference on Learning Representations (2017).
43. Suarez E, Eicker N, Lippert T. Modular supercomputing architecture: from idea to production. Contemporary high performance computing. 2019;23–55. https://doi.org/10.1201/9781351036863-9.
44. NVLINK. https://www.nvidia.com/en-us/data-center/nvlink/. Accessed 20 Sep 2022.
45. GPFS. https://apps.fz-juelich.de/jsc/hps/juwels/filesystems.html. Accessed 17 Apr 2023.
46. EasyBuild. https://github.com/easybuilders/easybuild. Accessed 20 Sep 2022.
47. NSys. https://docs.nvidia.com/nsight-systems/index.html. Accessed 20 Sep 2022.
48. cuDNN. https://developer.nvidia.com/cudnn. Accessed 20 Sep 2022.
49. Graphcore. https://www.graphcore.ai/products/ipu. Accessed 20 Sep 2022.
50. NVLAMB. https://github.com/NVIDIA/DeepLearningExamples/blob/master/PyTorch/LanguageModeling/BERT/README.md. Accessed 20 Sep 2022.

## Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

# Paper II

**Optimal Resource Allocation for Early Stopping-based Neural Architecture Search Methods**

M. Aach, E. Inanc, R. Sarma, M. Riedel, and A. Lintermann

Proceedings of the Second International Conference on Automated Machine Learning (2023)

This article is an open-access article distributed under the terms and conditions of the Creative Commons Attribution License (`http://creativecommons.org/licenses/by/4.0/`).

Marcel Aach wrote the code and the paper, ran all experiments on the HPC machines and performed the analysis.

# Optimal Resource Allocation for Early Stopping-based Neural Architecture Search Methods

**Marcel Aach**[1,2] **Eray Inanc**[1] **Rakesh Sarma**[1] **Morris Riedel**[1,2] **Andreas Lintermann**[1]

[1]Jülich Supercomputing Centre, Forschungszentrum Jülich, Germany
[2]School of Engineering and Natural Sciences, University of Iceland, Iceland

**Abstract** The field of Neural Architecture Search (NAS) has been significantly benefiting from the increased availability of parallel compute resources, as optimization algorithms typically require sampling and evaluating hundreds of model configurations. Consequently, to make use of these resources, the most commonly used early stopping-based NAS methods are suitable for running multiple trials in parallel. At the same time, also the training time of single model configurations can be reduced, e.g., by employing data-parallel training using multiple Graphics Processing Units (GPUs). This paper investigates the optimal allocation of a fixed amount of parallel workers for conducting NAS. In practice, users have to decide if the computational resources are primarily used to assign more workers to the training of individual trials or to increase the number of trials executed in parallel. The first option accelerates the speed of the individual trials (exploitation) but reduces the parallelism of the NAS loop, whereas for the second option, the runtime of the trials is longer but a larger number of simultaneously processed trials in the NAS loop is achieved (exploration). Our study encompasses both large- and small-scale scenarios, including tuning models in parallel on a single GPU, with data-parallel training on up to 16 GPUs, and measuring the scalability of NAS on up to 64 GPUs. Our empirical results using the HyperBand, Asynchronous Successive Halving, and Bayesian Optimization HyperBand methods offer valuable insights for users seeking to run NAS on both small and large computational budgets. By selecting the appropriate number of parallel evaluations, the NAS process can be accelerated by factors of $\approx 2-5$ while preserving the test set accuracy compared to non-optimal resource allocations.

## 1 Introduction

Neural Architecture Search (NAS) describes the process of automatically designing neural networks for various applications (Hutter et al., 2019). As NAS requires searching through a vast space of possible neural network architectures to find the one that best suits the given problem, this process is computationally expensive. Therefore, there has been a growing interest in exploiting parallel computing to speed up this process. By using parallelism, multiple candidate architectures can be evaluated concurrently, which significantly reduces the total runtime to find an architecture candidate that performs well. In recent years, the available NAS and Hyperparameter Optimization (HPO) frameworks have made an effort to run in parallel (Li et al., 2020a; Kandasamy et al., 2020; Liaw et al., 2018; Balaprakash et al., 2018). However, investigations into how these methods scale up are sparse and the optimal degree of parallelism to use in the NAS process – when only limited resources are available – is still an open question.

This paper aims to investigate the impact of the parallelism of the NAS loop on the runtime and quality of the outcome of three of the most commonly used early stopping-based NAS methods: Hyperband (HB) by (Li et al., 2017), Asynchronous Successive Halving Algorithm) (ASHA) by (Li et al., 2020a), and Bayesian Optimization and HyperBand (BOHB) by (Falkner et al., 2018). All of these algorithms have 'meta-hyperparameters' themselves, that have a large influence on the

AutoML 2023 © 2023 the authors, released under CC BY 4.0

performance. One example is the reduction factor, which defines the percentage of trials that are terminated by early stopping. Investigations have shown that a reduction factor of three or four is a robust choice (Li et al., 2020a). However, the optimal number of parallel evaluations that should be used has not been studied extensively.

We study the effect of parallelism on the NAS methods using both, small and large computational budgets on the standardized NATS-Benchmark (Dong et al., 2021), training up to 256 Convolutional Neural Networks (CNNs) on three image classification datasets. In the first step, the optimal number of parallel evaluations to run when using a **single** Graphics Processing Unit (GPU) – also refereed to as a worker or a device in the following – is evaluated. As a second step we increase the number of GPUs to 16 and vary the number of parallel evaluations by running data-parallel training of the architectures on one to four GPUs per trial. Our main objective is to explore the balance between accelerating single trials by allocating more resources per trial and speeding up the entire NAS loop by running more trials in parallel, considering a fixed amount of available hardware resources. The original motivation of HB and its successors ASHA and BOHB was to develop an adaptive, multi-fidelity approach for resource allocation, however, mainly from a temporal perspective. That is, 'resources' or fidelity are defined as a fixed amount of training epochs a NAS run can use and that then is shared between trials.

In this work, 'resources' and fidelity are defined as fixed amount of physical workers to address this issue from a hardware perspective. To investigate the **efficient utilization** of these workers, we are looking to answer the following two research questions:

- **RQ-1**: How many resources should be allocated in total for a NAS?
- **RQ-2**: How many resources should be allocated for each NAS trial?

Our main contribution is an investigation of these research questions on small- and large-scale computational budgets (up to 64 GPUs), featuring different algorithms, types and sizes of datasets, and neural networks and types of GPUs. Our findings suggest to increase the parallelism of the NAS loop at the cost of the parallelism of the single trials, independent of the factors such as algorithm, dataset, or hardware type.

In Sec. 2, we introduce the methods used for evaluation and embed this study in related scalability and resource allocation work. Section 3 then describes the benchmark and technical frameworks used, while in Sec. 4 and Sec. 5 the results of the experiments on various scales are reported and discussed. We finish with a conclusion in Sec.6.

## 2 Related Work

### 2.1 Hyperband (HB)

In Machine Learning (ML), the performance of a certain model with respect to a certain metric, i.e., the validation error, can be described by the function $f : \mathcal{X} \rightarrow \mathbb{R}$ where $\mathcal{X}$ is the space of possible model architectures. The main goal of NAS is to minimize the objective function $f$ by finding an architecture configuration $x^* \in \mathcal{X}$ such that $x^* \in \arg\min_{x \in \mathcal{X}} f(x)$. Evaluations of the objective functions are expensive as the model configurations usually require full training. HB (Li et al., 2017) seeks to approximate this objective function by evaluating it on a smaller budget, e.g., by running a trial with fewer epochs. Worse performing trials are then cut off early and their resources are transferred to the best performing trials. This early stopping procedure is known as Successive Halving (Jamieson and Talwalkar, 2016). To assess the most promising trials, HB waits for *all* trials of a bracket to reach a certain threshold in time before applying Successive Halving. This leads to idling workers as some will be faster than others (which can be circumvented by spawning a new bracket if enough workers are available). HB is a pure scheduling algorithm as new configurations to evaluate are randomly sampled.

### 2.2 Bayesian Optimization Hyperband (BOHB)

Another option to accelerate the NAS process is the use of optimization methods to find the minimum of $f$, where Bayesian optimization (BO) has become the main method of choice. The idea behind BO is to use a probabilistic model of $f$ based on data points observed in the past. In the case of NAS, this means finding promising new model architectures based on the performance of past trials.

BO is computationally costly and BOHB (Falkner et al., 2018) solves this issue by combining the optimization process with HB for scheduling. In BOHB, HB chooses the number of hyperparameter configurations and their assigned budget. BO chooses the hyperparameters by deploying a tree parzen estimator (Bergstra et al., 2011). Combining both approaches has several advantages: good results are achievable on smaller budgets while on larger budgets the performance is better than other methods, such as plain HB or random search (Bergstra and Bengio, 2012).

### 2.3 Asynchronous Successive Halving Algorithm

ASHA (Li et al., 2020a) addresses the problem of large scale NAS by improving the scalability of HB. To avoid idling workers ASHA, in contrast to HB decides on a rolling basis which trials are promising. When two trials reach the time barrier, the trial with the better performance is continued while the other is paused until the performance of the next completed trial can be juxtaposed. This asynchronous comparison leads to speed-ups. ASHA focuses purely on the scheduling aspects of NAS and new architecture candidates are randomly sampled, contrary to BOHB.

### 2.4 Scalability and Resource Allocation Investigations

In the original works on ASHA and BOHB, initial investigations on their scalability with respect to the total number of workers used in the NAS process are already performed. In the BOHB case, the algorithm is scaled up to 32 Central Processing Unit (CPU) workers to perform HPO on a small benchmarking dataset from OpenML (Frey and Slate, 1991; Vanschoren et al., 2014), attaining a speed-up of $15x$ in comparison to using a single worker. However, no scalability results on larger datasets or GPUs are reported. In a proposed extension to the BOHB framework, substantial speed-ups are measured, still just on small multi-layer perceptron models on tabular datasets (Klein et al., 2020).

While in the ASHA paper, a study on a total number of 500 GPUs is reported, the authors do not juxtapose these results with results achieved on a smaller number of workers for the same problem to compute a speed-up. The paper already addresses the trade-off of using resources to train a model faster vs. evaluating more models in parallel. This is, however, done on just a single model and dataset, and by simulating the training times using an analytical performance model (Qi et al., 2017). In contrast, our study performs evaluations on different types of GPUs, encompasses larger models and datasets, and takes measurements by actually performing the training runs (instead of simulating them).

### 2.5 Data-Parallel Deep Learning

Data-parallel training is especially suited for training deep neural networks on large datasets on multiple GPUs. This method greatly accelerates the training process.

To perform data-parallel training, the training dataset is partitioned across multiple workers, and each worker trains a copy of the model on a subset of the data. During each iteration, the gradients of the loss function with respect to the model parameters are computed on each device, and then averaged across all devices to obtain a global gradient update. The model parameters are then updated using this global gradient (Li et al., 2020b). The global gradient update can be computed as

$$\Delta w = \frac{1}{N} \sum_{i=1}^{N} \Delta w_i, \tag{1}$$

where $\Delta w_i$ is the gradient computed on device $i$ and $N$ is the total number of devices. With a growing number of devices, this averaged gradient changes, which impacts the generalization performance (Keskar et al., 2017). It is possible to circumvent this degradation of performance by scaling the learning rate with the number of devices (Goyal et al., 2017). This works, however, only if the number of devices that is trained on in parallel is small. Another way to avoid this degradation is to optimize the learning rate using BO (Égelé et al., 2021).

## 3 Methodology

### 3.1 Ray Tune

For our experiments we use the open-source Ray Tune (Liaw et al., 2018) library. It offers to run multiple NAS optimizations across multiple GPUs via a unified interface. We use the implementations of ASHA, BOHB, and HB from within Ray Tune to eliminate implementation-specific disturbance factors in our analysis. All communication and scheduling is therefore handled via Ray. One of the most important features of the Ray library is the possibility to easily modify the number of workers to use per evaluation, even allowing for floating point values. This way, it is possible to run multiple trials in parallel on a single GPU, which share its compute power via 'context swichting'. In our experiments, we use Ray Tune for the NAS loop and PyTorch-DDP (Li et al., 2020b) for the training of the single models.

### 3.2 NATS Benchmark

Benchmarks play an important role in NAS research by providing a standardized evaluation framework for comparing different NAS algorithms on multiple datasets. For this study, we run our main experiments on the size search space $\mathcal{S}_s$ from the NATS-Bench by (Dong et al., 2021). For this size search space, a general CNN model with fixed topology is created, where the number of channels for each stage of convolutional layers is sampled from $k \cdot 8, k \in \{1, \ldots, 8\}$. As there are five layer stages in the model, the total number of possible configurations is $8^5$. We train all models with the hyperparameters mentioned in the original NATS-Bench paper. This includes the usage of the Stochastic Gradient Descent optimizer with learning rate of $lr = 0.1$, momentum of $mom = 0.9$, weight decay of $wd = 0.0005$, batch size of $bs = 256$, and nesterov momentum for 90 epochs. The quantity $lr$ is annealed from 0.1 to 0 over the course of training with the cosine annealing schedule (Loshchilov and Hutter, 2017), also in accordance with the NATS-Bench training protocol.

### 3.3 Datasets

We train each model configuration on the most popular image classification datasets cifar-10, cifar-100 (Krizhevsky, 2009), and imagenet (Russakovsky et al., 2015). To reduce the computational costs of our analysis, we use a down-sampled variant of imagenet for most of our experiments, where each image is scaled to $16 \times 16$ pixels and the number of classes is reduced to 120 (also referred to as imagenet16-120). As these datasets do not feature a validation dataset, we set aside 20% of the training dataset for validation to evaluate the obtained model configurations during the NAS optimization process[1]. The testset remains untouched. For all datasets, we reuse the pre-processing and augmentation techniques from NATS-Bench.

## 4 Experimental Results

In this section, the experimental results are presented, evaluating the scalability of the different NAS methods on up to 64 GPU and the general performance of NAS methods when sharing a single or

[1]This is different from the dataset splits reported in NATS-Bench, which uses a 50/50 split. However, to measure the effect of resource allocation, we need our training dataset to be sufficiently large, which is why we chose the 80/20 split.

multiple GPUs per trial. The experiments from Sec. 4.1, 4.2, 4.3 and 4.4 are performed on JURECA-DC-GPU (Jülich Supercomputing Centre, 2021), a High-Performance Computing (HPC) machine that features NVIDIA A100[2] GPUs, while the experiments from Sec. 4.5 are performed on the MareNosturm4 CTE-AMD machine, equipped with AMD MI50 [3] GPUs and the DEEP-EST (Suarez et al., 2021) machine, with NVIDIA V100 [4] GPUs. Code to reproduce the results is available via a GitLab repository [5].

### 4.1 General Scalability

To answer the first research question (**RQ-1**), this section investigates the general scalability of the three early-stopping based NAS methods, plain HB, BOHB, and ASHA. As this work focuses on efficient resource allocation, the question at hand is to find out at which point adding more workers to the NAS task is not profitable anymore to guarantee efficient utilization. A common metric for tracking this is to perform strong scaling tests and compare the parallel efficiencies $E_G$ of different NAS methods. In strong scaling analyses, the problem size remains constant while the number of workers is increased. In this case, the problem size is defined as the total number of samples the NAS methods need to evaluate, and is set to 128. The speedup can be calculated by

$$S_G = \frac{T_1}{T_G}, \tag{2}$$

where $G$ is the number of GPUs, $T_1$ is the runtime of the single GPU baseline case, and $T_G$ is the runtime on $G$ GPUs. Then, $E_G$ can be expressed by

$$E_G = \frac{S_G}{G}, \quad E_G \in [0, 1] \tag{3}$$

In case $E_G$ equals (or is close to) unity, the best possible scaling performance is achieved (only in exceptional cases values larger than unity are possible). Analyzing $E_G$ allows for a standardized comparison of the different methods. Figure 1 shows $E_G$ values for setups between 1 and 64 GPUs on all three datasets from NATS-Bench, running one trial per GPU. The random search (RAND) plots serve as a best case scenario as all trials are fully trained independently from each other and there is no early stopping. Allocation of more GPU resources reduces the runtime almost proportionally. Hence, $E_G$ values for this case are always above 0.85. Among other methods, the ASHA case achieves the best parallel performance on up to 8 GPUs with $E_G = 1$ across all three datasets. This result is expected, since the trials in this method run in asynchronous manner. Overall, all methods perform reasonably well up to 32 GPUs with $E_G$ values above 0.75. However, it can be seen that $E_G$ drops below 0.75 on 64 GPUs across all datasets and algorithms. The drop in performance in comparison to random search becomes apparent for larger number of GPUs.

### 4.2 Resource Allocation on a Single GPU

To answer the second research question (**RQ-2**), we investigate how the amount of resources allocated to a single trial impacts the total runtime of the NAS process.

For the experiments depicted in Fig. 2, the different early stopping-based NAS algorithms run until 64 different architecture candidate samples have been evaluated. The plots depict the validation accuracy of the best architecture currently discovered by the NAS over the time for running up to eight trials in parallel on a single GPU. The plots are partly truncated on the x- and y-axis, and standard deviation is omitted for readability reasons, see App. A for full plots including

[2] https://www.nvidia.com/en-us/data-center/a100/
[3] https://www.amd.com/en/products/professional-graphics/instinct-mi50
[4] https://www.nvidia.com/en-us/data-center/v100/
[5] https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/nas-allocation

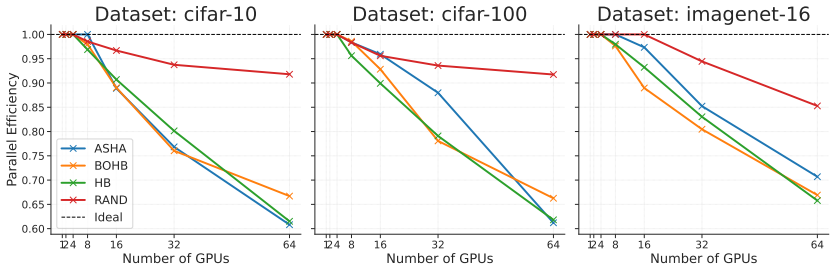



Figure 1: Scalability comparison: Parallel efficiency $E_G$ over the number of GPUs, based on the runtime until 128 samples are evaluated. $E_G = 1$ denotes the ideal case.

error bars. Comparing the ASHA, BOHB, and HB to the random search (RAND) runtimes in Fig. 2 proves that early stopping methods greatly benefit the computation. The differences between the final validation accuracies that the NAS methods converge to are with less than 1% only marginal across all datasets. BOHB is the only exception, but also here differences are only slightly above 1%. Therefore, it can be stated that the validation accuracy is not impacted by the number of parallel trials.

It is, however, evident that the runtime is largely influenced by the parallelism of the NAS algorithm, i.e., the NAS methods take up to twice as long to finish when running just a single trial at a time (blue lines in Fig. 2). The main reason for this is that early stopping-based methods can only make their decisions on which trials to stop and which to continue based on having multiple, intermediate results available. If just one trial is running (sequentially) at a time, the algorithm needs to constantly checkpoint the current trial, load a new architecture, and train it for a specified amount of time to have a comparison point. This checkpointing and re-loading creates overhead that ultimately leads to longer runtimes. When running multiple trials in parallel, these kinds of decisions can be made on the fly. While the amount of checkpoint operations is similar, less re-loading is necessary in this case. The differences between the runtimes for 2, 4, and 8 trials per GPU are small (less than 2,000s). For both, the ASHA and BOHB case running 4 or 8 trials in parallel take roughly the same time to finish. What is interesting to observe is the fact that the *fewer* resources are used per trial, i.e., 1, 2, or 4 trials per GPU, the *faster* the NAS method discovers architectures with high validation accuracies, at least at the beginning. This can be observed by the steep increase of the blue, orange, and green lines in Fig. 3 for small runtimes. This results in enhanced anytime performance for the NAS loop, i.e., stopping the algorithm at a random point in time and evaluating the performance at that point. This means that halting the process at a random moment would likely produce higher validation accuracy than when running the maximum of 8 trials per GPU.

In addition to the frequent checkpointing and re-loading, one possible reason for the longer runtime of the sequential trial case could be the idling of the GPU due to not receiving sufficient data for calculations fast enough. When running 8 trials in parallel, the GPU processes $8 \cdot 256 = 2{,}048$ data samples at a time, while in the single trial case just 256 samples (=one batch) are processed at a time. The latter is not enough to efficiently utilize a GPU.

To account for this effect, a reference experiment is conducted, where the quantity *bs* is scaled inversely with the number of trials running on the GPU. Figure 3 shows the corresponding results. Usually, a larger *bs* value leads to a faster training time, as the GPU can streamline the computations of the gradients better and does not have to interrupt as often to perform an optimizer update. For

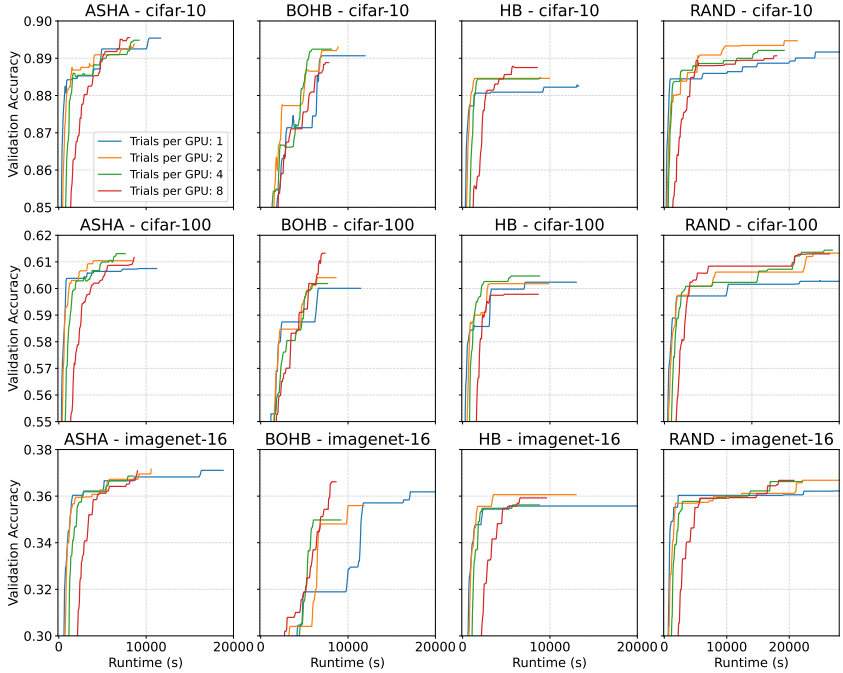



Figure 2: Multiple trials on a single GPU, showing validation accuracy over runtime. Results are averaged over three different seeds. The plots are partly truncated for readability. In all cases, running 1 trial per GPU (blue line) clearly has the longest runtime while running 4 or 8 trials (green and orange lines) per GPU finish faster.

this test, the batch size is set to $bs = 256$ as the baseline for the '8 Trials per GPU', then $bs = 2 \cdot 256$ is selected for the '4 Trials per GPU' case, and so on. By doing this, we align the number of data samples processed between the different levels of parallelism. To obtain comparable performance in terms of validation accuracy, the quantity $lr$ is scaled with the batch size. We only run this test for the imagenet-16-120 dataset as the linear batch size and $lr$ scaling already degrades the validation accuracy when training on the cifar-10 and cifar-100 datasets.

The results show that the time difference between the '1 Trial per GPU' scenario and the other cases reduces significantly. Table 1 compares the outcomes and GPU utilizations of the NAS process with and without the batch scaling for the ASHA case. While the factor in average runtime between non-optimal ('1 Trial per GPU') and optimized ('4 Trial per GPU') resource allocation with no batch scaling is at $18{,}817s/8{,}524s \approx 2.2$, it reduces to $10{,}749s/6{,}989s \approx 1.54$ for the batch scaling ('1 vs. 2 Trials per GPU'). The batch scaling also improves the GPU utilization of almost all resource allocation strategies, however, for the sequential scenario it only increases from 38% to 49%. This finding confirms that the main reason hindering efficient resource usage is the switching and reloading processes between trials and not the amount of simultaneously processed data. Overall it can again be seen that the resulting test set accuracy is not impacted by the number of parallel trials.

Table 1: Averaged results of using ASHA on the imagenet-16 dataset on a **single** GPU with standard deviation as uncertainty measurement. The runtime is the wallclock time of the whole NAS loop. The test set accuracy is computed by using the architecture with the best performance on the validation set. GPU utilization is measured using an internal tool of the HPC machine.

| Trials per GPU | Without (inverse) batch scaling | | | With (inverse) batch scaling | | |
|---|---|---|---|---|---|---|
| | Runtime | Test set accuracy | GPU utilization | Runtime | Test set accuracy | GPU utilization |
| 1 | 18,817 ±414 s | 37.6 ±0.77% | 38 ±17 % | 10,749 ±257 s | 36.7 ±0.24 % | 49 ±21 % |
| 2 | 10,583 ±212 s | 37.7 ±1.09% | 79 ±9 % | 6,989 ±248 s | 37.3 ±0.11 % | 82 ±5 % |
| 4 | 8,524 ±113 s | 37.7 ±0.91% | 83 ±5 % | 7,495 ±230 s | 37.1 ±0.49 % | 82 ±4 % |
| 8 | 8,996 ±152 s | 37.5 ±0.30% | 86 ±3 % | 9,252 ±237 s | 37.4 ±0.06 % | 86 ±4 % |

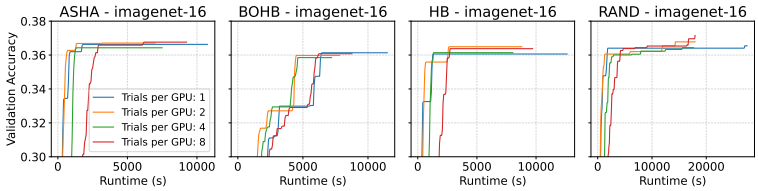



Figure 3: Multiple trials on a single GPU with inverse batch scaling according to the number of trials, showing validation accuracy over runtime.

### 4.3 Resource Allocation on Multiple GPUs with Data-Parallel Training

The effect of employing a higher number of workers per trial is investigated here to answer the second research question (**RQ-2**). Therefore, data-parallel training, see Sec. 2.5, is employed. To obtain the results depicted in Fig. 4, the runtime of the early stopping-based NAS algorithms on a **fixed** number of 16 GPUs is measured to evaluate 64 different architecture candidates. The base case with a batch size of $bs = 256$ and a learning rate of $lr = 0.1$ is scaled linearly with the number of GPUs.

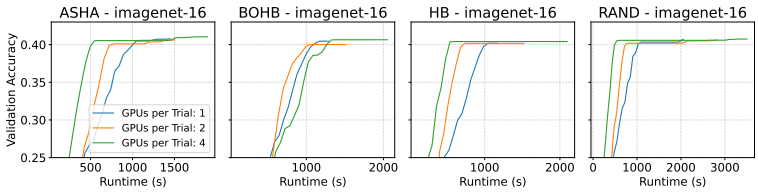
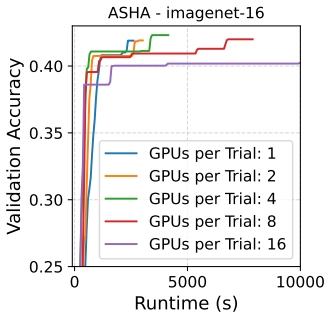



Figure 4: Distributing trials across **multiple** GPUs, showing validation accuracy over runtime. The results are averaged over three different seeds.

The results are in line with the findings from the single GPU case. When using four GPUs per trial, the NAS methods take the longest time to finish. For the ASHA and BOHB cases, this implies that the runtimes of the 'GPUs per trial: 1' configuration are $\approx 1.3$ and $\approx 1.59$ faster than the 'GPUs per trial: 4' configuration. However, with the exception of BOHB, the observations regarding the anytime performance hold true here. That is, using more GPUs per trial accelerates the discovery of architectures with better validation accuracies in the beginning of the NAS run.

To investigate resource allocations on a larger computational budget, a NAS study for the ASHA algorithm is performed with a fixed number of 64 GPUs as resources, measuring the runtime until 256 model architectures are evaluated. The number of workers to use for the data-parallel training is varied from $1-16$. The results are depicted in Fig. 5. Again, it is observed that increasing the parallelism of the NAS loop and decreasing the parallelism of the data-parallel training ('GPUs per trial: 1 or 2' case) results in an optimal resource allocation. In this case, the factor between optimized and sub-optimal allocations becomes even larger at $13{,}711s/2{,}601s \approx 5.2$ with no impact on the test set accuracy.




| GPUs per trial | Runtime | Test set accuracy |
|---|---|---|
| 1 | 2,601 s | 43.4 % |
| 2 | 3,021 s | 42.7 % |
| 4 | 4,148 s | 42.3 % |
| 8 | 7,886 s | 42.9 % |
| 16 | 13,711 s | 41.4 % |

Figure 5: Distributing trials across a total of 64 GPUs for ASHA on imagenet-16. Left (plot): The validation accuracy over runtime until 256 samples are evaluated. The plot is partly truncated for readability. Right (table): Overview of runtime and test set accuracy achieved.

### 4.4 Evaluation with Large Models

To test how well the findings transfer to larger neural network models and datasets, we perform an additional evaluation with a custom ResNet50 (He et al., 2016) search space, where the number of convolutional filters at each stage of the ResNet is sampled from the set $\mathcal{S} = \{64, 128, 256, 512\}$. As there are four layer stages in the model, the total number of possible configurations is $4^4 = 256$. The sampled models are roughly two orders of magnitude larger in terms of the number of parameters than the models from the NATS-Bench search space (e.g., 30M vs. 0.3M parameters). The training is performed on the complete imagenet-1k dataset ($\approx$ 300 GB in size), utilizing the same training protocol as for the NATS-Bench experiments and using a fixed budget of 64 GPUs. The results in Fig. 6 (left) show that also in this case increasing the parallelism of the NAS loop at the cost of the parallelism of the single trials reduces the total runtime significantly (blue vs. green line).

### 4.5 Evaluation on Different GPU Types

To asses to what extend the results are influenced by the type of GPU utilized for the NAS runs, we perform additional experiments on different HPC machines, featuring less powerful GPUs than the NVIDIA A100s from the previous sections. For the single GPU case (Sec. 4.2), one AMD MI50 is used while for the multi-GPU case (Sec. 4.3) 16 NVIDIA V100s are used for a comparison. The results are depicted in Fig. 6 (middle and right). For both experiments, it can be observed that running more trials in parallel leads to a reduced runtime (red line in the middle plot and blue line in the right plot).

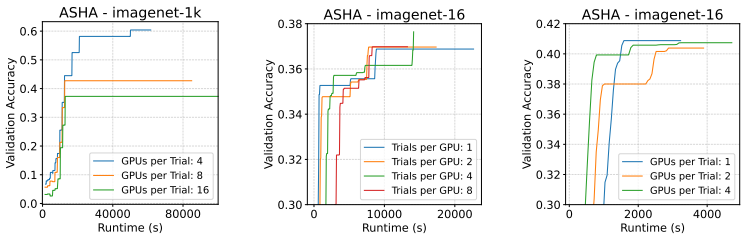



Figure 6: Left: Running trials with ResNet search space on imagenet-1k. Middle: Running trials on a single AMD MI50 GPU. Right: Running trials across 16 NVIDIA V100 GPUs.

## 5 Discussion and Insights

From the results, several things become evident. First, to answer **RQ1**, the general scalabilty of early stopping-based NAS methods show good performance on up to 32 GPU workers in parallel. This is not only true for the smaller cifar-10 and cifar-100 datasets, but also for the larger imagenet 16-120 dataset. A good scalability indicates that adding more workers to a problem proportionally reduces the total runtime, leading to faster computation of the NAS loop and optimized resource usage. The benefits of adding more GPUs to a problem outweighs the drawbacks of more communication and scheduling overhead in this case. Efficiency drops for all methods (excluding random search) when using 64 GPUs in parallel. This indicates that at this point drawbacks of adding more workers to the problem grow, resulting in sub-optimal resource usage. For datasets and models comparable in size to the ones from NATS-Bench, it is therefore recommended to use a maximum of 32 workers for the NAS loop to keep parallel efficiency above $\approx 0.75$, i.e., ensuring better resource exploitation. If practitioners are comfortable with parallel efficiency of only $\approx 0.6$, even up to 64 GPUs can be used, but it should be noted that in this case a large portion of compute resources is spent on communication and scheduling tasks. In response to **RQ2**, the resource allocation investigations on a single GPU show that it is in general advisable to increase the parallelism of the NAS loop to achieve a decreased total runtime. In addition to the wasted resources by idling the GPU when running only a single trial, the overhead created by frequent checkpointing and re-loading of configurations increases the runtime by a large factor. This holds true even when adjusting for GPU utilization by adjustment of the batch size (see. Fig. 3), though the effect becomes notably smaller. The same conclusions can be made when running data-parallel training on multiple GPUs – at least some degree of parallelism of the NAS loop should be ensured. If it is the aim to get a strong anytime performance, it is advisable to increase the resources available to the single training loops. However, the main takeaway message of this study is that also small levels of NAS parallelism (just $2-4$ concurrently running trials) deliver a good trade-off between fast overall runtime and strong anytime performance. This takeaways message is consistent both on small and the large resource budgets, on small and large datasets and models and on various types of GPUs.

## 6 Conclusion and Future Work

In this work, the optimized and sub-optimal resource allocation for the three early-stopping based NAS methods ASHA, BOHB, and HB has been explored. The scalability on three image classification datasets with CNN models from NATS-Bench has been obtained, and recommendations on how many resources to use in total, and how to balance them between single trials and overall NAS loop has been provided. Our analysis contributes to the field of AutoML by providing guidance on how to allocate their devices when running a study with NAS algorithms with the ultimate goal of achieving fast runtimes. For future work, it is of interest to develop strategies that perform adaptive re-allocation of hardware resources, in similar fashion as HB does from a temporal perspective.

## 7 Broader Impact Statement

This work presents a study on optimal resource allocations for NAS methods. While our evaluations are initially computationally expensive, the findings in this paper help researchers to run their NAS algorithms more efficiently, resulting in less wasteful use of resources and lower energy consumption.

## 8 Submission Checklist

1. For all authors...

   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] We introduce the problem and summarize the content and results of our study in Sec.1.

   (b) Did you describe the limitations of your work? [Yes] We mention this work is performed on the NATS-Bench datasets and therefore only applicable to comparable datasets and models in Sec. 5

   (c) Did you discuss any potential negative societal impacts of your work? [Yes] We included a Broader Impact statement on the environmental impacts.

   (d) Have you read the ethics author's and review guidelines and ensured that your paper conforms to them? `https://automl.cc/ethics-accessibility/` [Yes]

2. If you are including theoretical results...

   (a) Did you state the full set of assumptions of all theoretical results? [N/A] no theoretical results

   (b) Did you include complete proofs of all theoretical results? [N/A] no theoretical results

3. If you ran experiments...

   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results, including all requirements (e.g., `requirements.txt` with explicit version), an instructive `README` with installation, and execution commands (either in the supplemental material or as a URL)? [Yes] [We include the Python code and the batch scripts to submit the scripts as jobs on an HPC machine, however in anonymized form. This means that scripts will not run out of the box and would need to be adjusted to different HPC machines.]

   (b) Did you include the raw results of running the given instructions on the given code and data? [No] [The results are processed using a text-parser. Unfortunately, the raw log files cannot be shared anonymized form.]

   (c) Did you include scripts and commands that can be used to generate the figures and tables in your paper based on the raw results of the code, data, and instructions given? [No] [As mentioned above, the raw log files cannot be shared anonymized form.]

   (d) Did you ensure sufficient code quality such that your code can be safely executed and the code is properly documented? [Yes] We include a README file with explanations for the code as well.

   (e) Did you specify all the training details (e.g., data splits, pre-processing, search spaces, fixed hyperparameter settings, and how they were chosen)? [Yes] All training details are taken from NATS-Bench and specified in Sec.2.

   (f) Did you ensure that you compared different methods (including your own) exactly on the same benchmarks, including the same datasets, search space, code for training and hyperparameters for that code? [Yes] The same benchmark is used for almost all experiments (with the exception of the experiments in Sec. 4.4).

   (g) Did you run ablation studies to assess the impact of different components of your approach? [Yes] [We accounted for the effect of resource usage via the inverse batch scaling.]

(h) Did you use the same evaluation protocol for the methods being compared? [Yes] [The same evaluation protocol was used for all methods.]

(i) Did you compare performance over time? [Yes] [All plots in Sec. 4 are over time.]

(j) Did you perform multiple runs of your experiments and report random seeds? [Yes] [Results for the single GPU and some of the multi GPU cases are averaged over 3 different seeds.]

(k) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] [Error bars are omitted for readability in the main paper but available in the appendix.]

(l) Did you use tabular or surrogate benchmarks for in-depth evaluations? [No] [To assess the impact of parallelism, we believe it is important to run the actual calculations.]

(m) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] [For each experiment, the number of GPUs used and the runtime is specified in the plots.]

(n) Did you report how you tuned hyperparameters, and what time and resources this required (if they were not automatically tuned by your AutoML method, e.g. in a NAS approach; and also hyperparameters of your own method)? [Yes] [We tuned the same parameters as mentioned in the NATS Benchmark.]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

(a) If your work uses existing assets, did you cite the creators? [Yes] [The dataset creators are cited.]

(b) Did you mention the license of the assets? [No] [We are not aware of license issues with the image classification datasets.]

(c) Did you include any new assets either in the supplemental material or as a URL? [N/A] [No new assets used.]

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] [We are not aware of any issues in this regard with the image classification datasets.]

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] [We are not aware of any issues in this regard with the image classification datasets.]

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] [Not applicable.]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] [Not applicable.]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] [Not applicable.]

**Acknowledgements**. The research leading to these results has been conducted in the CoE RAISE project, which receives funding from the European Union's Horizon 2020 – Research and Innovation Framework Programme H2020-INFRAEDI-2019-1 under grant agreement no. 951733. The authors gratefully acknowledge the computing time granted by the JARA Vergabegremium and provided on the JARA Partition part of the supercomputer JURECA at Forschungszentrum Jülich.

### References

Balaprakash, P., Salim, M., Uram, T. D., Vishwanath, V., and Wild, S. M. (2018). Deephyper: Asynchronous hyperparameter search for deep neural networks. In *2018 IEEE 25th International Conference on High Performance Computing*, pages 42–51.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In Shawe-Taylor, J., Zemel, R., Bartlett, P., Pereira, F., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305.

Dong, X., Liu, L., Musial, K., and Gabrys, B. (2021). NATS-bench: Benchmarking NAS algorithms for architecture topology and size. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1.

Égelé, R., Balaprakash, P., Guyon, I., Vishwanath, V., Xia, F., Stevens, R., and Liu, Z. (2021). Agebo-tabular: Joint neural architecture and hyperparameter search with autotuned data-parallel training for tabular data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA. Association for Computing Machinery.

Falkner, S., Klein, A., and Hutter, F. (2018). BOHB: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the 35th International Conference on Machine Learning*, pages 1436–1445.

Frey, P. W. and Slate, D. J. (1991). Letter recognition using holland-style adaptive classifiers. *Machine Learning*, 6(2):161–182.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2017). Accurate, large minibatch sgd: Training imagenet in 1 hour. Available at `https://arxiv.org/abs/1706.02677`.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

Hutter, F., Kotthoff, L., and Vanschoren, J. (2019). *Automated Machine Learning: Methods, Systems, Challenges*. Springer Publishing Company, Incorporated, 1st edition.

Jamieson, K. and Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. In Gretton, A. and Robert, C. C., editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 240–248, Cadiz, Spain. PMLR.

Jülich Supercomputing Centre (2021). Jureca: Data centric and booster modules implementing the modular supercomputing architecture at jülich supercomputing centre. *Journal of large-scale research facilities*, 7:A182.

Kandasamy, K., Vysyaraju, K. R., Neiswanger, W., Paria, B., Collins, C. R., Schneider, J., Poczos, B., and Xing, E. P. (2020). Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly. *Journal of Machine Learning Research*, 21(81):1–27.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2017). On large-batch training for deep learning: Generalization gap and sharp minima. Available at `https://arxiv.org/abs/1609.04836`.

Klein, A., Tiao, L. C., Lienart, T., Archambeau, C., and Seeger, M. (2020). Model-based asynchronous hyperparameter and neural architecture search. Available at https://arxiv.org/abs/2003.10865.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Available at https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2017). Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(1):6765–6816.

Li, L., Jamieson, K., Rostamizadeh, A., Gonina, E., Ben-tzur, J., Hardt, M., Recht, B., and Talwalkar, A. (2020a). A system for massively parallel hyperparameter tuning. In Dhillon, I., Papailiopoulos, D., and Sze, V., editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 230–246.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. (2020b). Pytorch distributed: Experiences on accelerating data parallel training. *Proceedings of Very Large Data Base Endowment Inc.*, 13(12):3005–3018.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., and Stoica, I. (2018). Tune: A research platform for distributed model selection and training. Available at https://arxiv.org/abs/1807.05118.

Loshchilov, I. and Hutter, F. (2017). SGDR: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations*.

Qi, H., Sparks, E. R., and Talwalkar, A. (2017). Paleo: A performance model for deep neural networks. In *Proceedings of the International Conference on Learning Representations*.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252.

Suarez, E., Kreuzer, A., Eicker, N., and Lippert, T. (2021). *The DEEP-EST project*, volume 48 of *Schriften des Forschungszentrums Jülich IAS Series*, pages 9–25. Forschungszentrum Jülich GmbH Zentralbibliothek, Verlag, Jülich.

Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. (2014). OpenML. *ACM Special Interest Group on Knowledge Discovery and Data Mining Explorations Newsletter*, 15(2):49–60.

## A Untruncated Plots Including Uncertainty

For reasons of readability, the plots in the main part of the paper were truncated and omitted uncertainty measurements. In the following, these plots are shown without truncation and including standard deviation as uncertainty measurement. Figure 7 is the untruncated version of Fig. 2, Fig. 8 of Fig. 4, and Fig. 9 of Fig. 4.

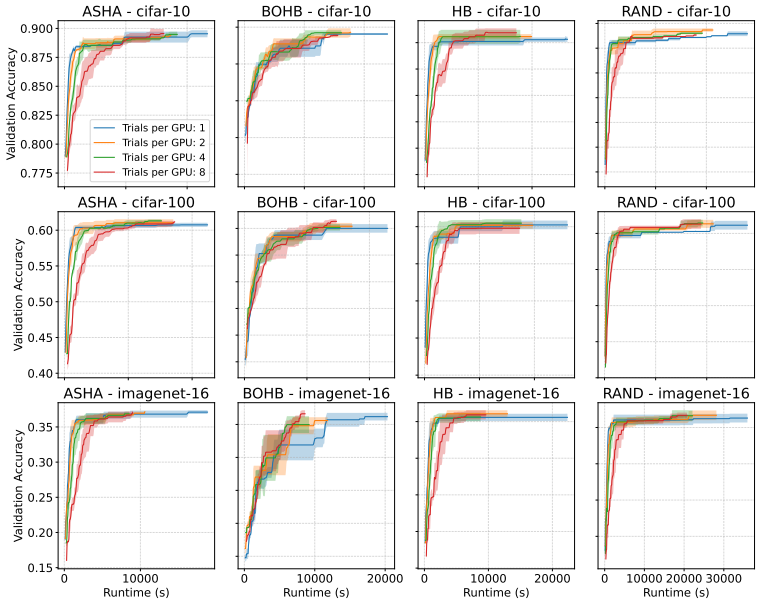



Figure 7: Multiple trials on a single GPU, showing validation accuracy over runtime. Results are averaged over three different seeds.

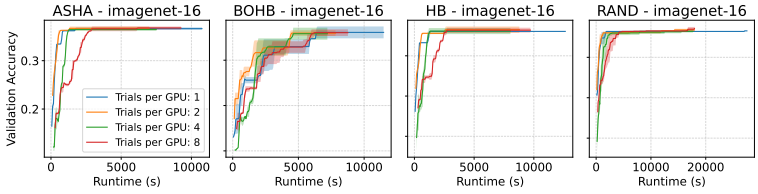



Figure 8: Multiple trials on a single GPU with inverse batch scaling according to the number of trials, showing validation accuracy over runtime. The results are averaged over three different seeds.

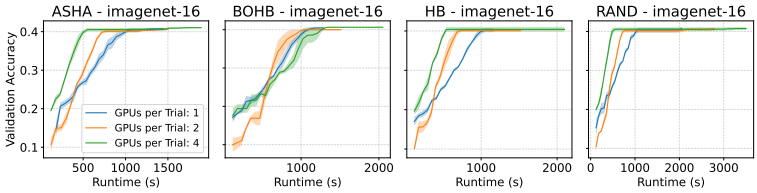



Figure 9: Distributing trials across **multiple** GPUs, showing validation accuracy over runtime. The results are averaged over three different seeds.

## B Scaling Plots

This section presents the scaling plots from Sec. 4.1 and Fig. 1 in more extensive way, i.e. the plots are split to show the parallel effiency on the small scale (1-8 GPUs) and on the large scale (16-64 GPUs), see Fig. 10.

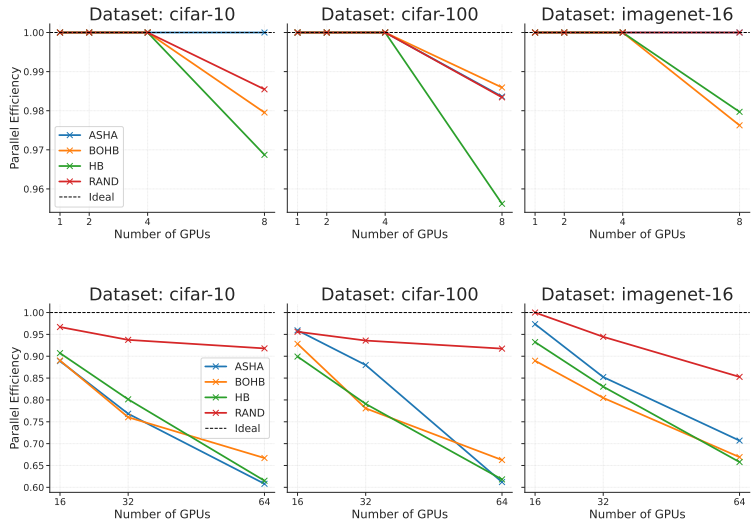



Figure 10: Scalability comparison: Parallel efficiency $E_G$ over the number of GPUs (Top: 1-8, Bottom: 16-64), based on the runtime until 128 samples are evaluated. $E_G = 1$ denotes the ideal case.

# Paper III

**Resource-Adaptive Successive Doubling for Hyperparameter Optimization on Large Datasets on High-Performance Computing Systems**

M. Aach , R. Sarma, H. Neukirchen, M. Riedel, and A. Lintermann

Submitted to Future Generation Computer Systems (2024)

Copyright to be determined based on future publisher.

Marcel Aach wrote the main part of the code and the paper, ran all experiments on the HPC machines and performed the analysis. The application of the developed method to the CFD use case was performed in collaboration with R. Sarma.

# Resource-Adaptive Successive Doubling for Hyperparameter Optimization with Large Datasets on High-Performance Computing Systems

Marcel Aach[a,b], Rakesh Sarma[a], Helmut Neukirchen[b], Morris Riedel[a,b], Andreas Lintermann[a]

[a] *Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Germany*
[b] *School of Engineering and Natural Sciences, University of Iceland*

**Abstract**

The accuracy of Machine Learning (ML) models is highly dependent on the hyperparameters that have to be chosen by the user before the training. However, finding the optimal set of hyperparameters is a complex process, as many different parameter combinations need to be evaluated, and obtaining the accuracy of each combination usually requires a full training run. It is therefore of great interest to reduce the computational runtime of this process. On High-Performance Computing (HPC) systems, several configurations can be evaluated in parallel to speed up this Hyperparameter Optimization (HPO). State-of-the-art HPO methods follow a bandit-based approach and build on top of successive halving, where the final performance of a combination is estimated based on a lower than fully trained fidelity performance metric and more promising combinations are assigned more resources over time. Frequently, the number of epochs is treated as a resource, letting more promising combinations train longer. Another option is to use the number of workers as a resource and directly allocate more workers to more promising configurations via data-parallel training. This article proposes a novel Resource-Adaptive Successive Doubling Algorithm (RASDA), which combines a resource-adaptive successive doubling scheme with the plain Asynchronous Successive Halving Algorithm (ASHA). Scalability of this approach is shown on up to 1,024 Graphics Processing Units (GPUs) on modern HPC systems. It is applied to different types of Neural Networks (NNs) and trained on large datasets from the Computer Vision (CV), Computational Fluid Dynamics (CFD), and Additive Manufacturing (AM) domains, where performing more than one full training run is usually infeasible. Empirical results show that RASDA outperforms ASHA by a factor of up to 1.9 with respect to the runtime. At the same time, the solution quality of final ASHA models is maintained or even surpassed by the implicit batch size scheduling of RASDA. With RASDA, systematic HPO is applied to terabyte-scale scientific dataset for the first time in the literature, enabling efficient optimization of complex models on massive scientific data.

*Keywords:* hyperparameter optimization, high-performance computing, distributed deep learning, machine learning

## 1. Introduction

In recent years, the amount of openly available data has drastically increased. This includes datasets from different scientific fields, such as Computer Vision (CV) [43], Earth Observation (EO) [45], High-Energy Physics (HEP) [41], Additive Manufacturing (AM) [9], or Computational Fluid Dynamics (CFD) [4]. To analyze these data efficiently and gain novel insights based on hidden correlations, the use of Deep Learning (DL) techniques and Neural Networks (NNs) has become essential due to their ability to automatically extract complex patterns. As the prediction quality of these NN models is highly dependent on the so-called hyperparameters, which are frequently related to, e.g., the NN architecture or the optimizer, systematic Hyperparameter Optimization (HPO) has become a crucial ingredient of Machine Learning (ML) workflows [12]. However, this search for optimal combinations of hyperparameters is challenging due to often high-dimensional search spaces. Furthermore, the performance of a sample from the search space can only be evaluated with a high degree of confidence after a full model training run. In the case of deep NNs trained on large datasets, this can become a major hurdle, even with extensive computing resources. Additionally, the search space is often diverse in nature. For example, the search space could be comprised of the learning rate, an optimizer-related parameter represented as floating point number, and the number of layers, an architectural parameter represented as an integer number $l > 0$. Categorical values, such as "type of optimizer" or "type of layer" are also possible. This makes the application of classical, gradient-based optimization methods infeasible. Hyperparameters also change under different models and datasets, making the generalization difficult to assess. One of the state-of-the-art HPO methods is the Asynchronous Successive Halving Algorithm (ASHA) [30]. It randomly samples multiple combinations, evaluates their performance with a lower training budget and then – after comparing their performance – terminates under-performing trials early on. To reduce the time to solution, ASHA is frequently executed in parallel, where multiple NN configurations (trials) are evaluated at the same time.

Modern High-Performance Computing (HPC) systems offer a natural setting for running this kind of workload. They feature accelerators, such as Graphics Processing Units (GPUs), that are ideally suited for efficient NN trainings (*fast computation*). Furthermore, these accelerators are connected by an

optimized communication network that enables *fast inter-node communication*. While current distributed HPO methods, such as ASHA, leverage the fast computation capabilities to train different hyperparameter candidates, the communication requirements are usually modest and limited to the exchange of the value of a certain metric, e.g., the current loss on the validation set for the comparison of the performance between trials.

This work introduces a novel method, the Resource-Adaptive Successive Doubling Algorithm (RASDA), that leverages *both* HPC features to perform HPO efficiently at scale. It combines two levels of parallelism: (i) on the HPO level, different trials are run in parallel and (ii) on the level of each trial run, the NN training is accelerated with data-parallel training. The latter splits the datasets onto multiple GPUs and performs gradient synchronization after each training step. As these gradients are typically large, they require high-bandwidth communication. RASDA then leverages the successive doubling principle, which progressively allocates more resources to more promising hyperparameter combinations, treating the amount of GPUs that are used for data-parallel training as resources (performing a doubling in space). In contrast, other successive halving techniques, such as the plain ASHA, treat the number of epochs during training of a model as resources and thus perform only halving in time, see Fig. 1.

The developed method is suitable for problems that involve large scientific datasets, where due to long training times, even with HPC resources it is not feasible to train more than the initially sampled hyperparameter configurations and users are interested in getting the best possible, fully-trained model in the shortest amount of time. Therefore, this study performs an extensive evaluation of RASDA on different datasets from the CV, CFD, and AM domains, which are up to 8.3 Terrabyte (TB) in size, to prove its capability to deal with large datasets. These datasets are used to tune the hyperparameters of different types of NNs, namely a Convolutional Neural Network (CNN), an autoencoder and a transformer. RASDA is also benchmarked against the current state-of-the-art successive halving HPO method ASHA. The new RASDA code is openly available on GitHub[1] for the community.

This article is structured as follows. Section 2 summarizes the related work and highlights the differences to this work. The main details of RASDA are presented in Sec. 3. The application cases are explained in Sec. 4, followed by a presentation and discussion of the empirical results of the algorithm in Sec. 5. Finally, a summary and outlook are provided in Sec. 6.

## 2. Related Work

In ML, the performance of a certain model measured by a specific metric, such as the validation error, can be represented by the function $f : \mathcal{X} \to \mathbb{R}$ where $\mathcal{X}$ denotes the space of possible hyperparameter combinations. The primary goal of HPO is to minimize the objective function $f$ by identifying a hyperparameter configuration $x^* \in \mathcal{X}$ such that $x^* \in \arg\min_{x \in \mathcal{X}} f(x)$.

[1] RASDA source: `https://github.com/olympiquemarcel/rasda`

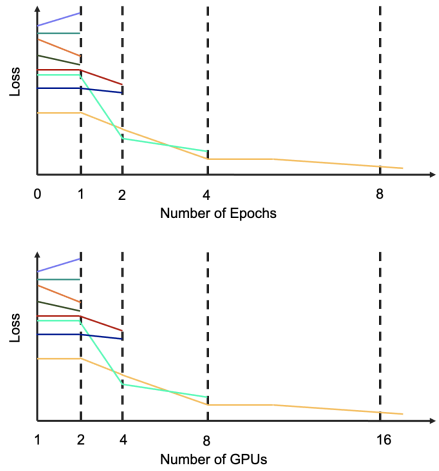



Figure 1: Comparison of successive halving in time (top) and halving in time combined with doubling in space (bottom). Each line corresponds to the learning curve of a single HPO combination.

Evaluations of the objective function are costly because they typically involve fully training the model for each configuration. To optimize this workflow, several approaches exist. These are either based on approximating $f(x)$ by a lower fidelity estimate, e.g., by the performance after a few training epochs or a model trained on a fraction of the data, or on choosing better hyperparameter configurations to evaluate, e.g., using Bayesian optimization (BO)). This section summarizes these approaches, i.e. Sec. 2.1 describes the successive halving method, Sec. 2.2 and Sec. 2.3 summarize resource-adaptive as well as other HPO algorithms and Sec. 2.4 introduces the concept of data-parallel training.

### 2.1. Successive Halving

Successive halving is a variant of Random Search [8], which uses the fact that most ML algorithms are iterative in nature. Intermediate performance results are thus accessible long before the algorithm is fully trained. The problem of finding optimal hyperparameters in a vast search space can then be framed in the context of a multi-armed bandit problem, where each arm represents a hyperparameter combination, and pulling an arm corresponds to training the combination for some iterations [21]. The goal is to identify the arm that yields the highest reward with the lowest budget possible. To do so in an efficient way, successive halving uniformly allocates an initial budget $B$ to $n_{arms}$ arms and evaluates their performance after a few iterations at a milestone with budget $B/n_{arms}$. It then eliminates the worst-performing half of the arms and promotes the most promising-performing arms by continuing to pull them. Each

of these successive halving steps is referred to as a rung. When following this procedure for a few steps from rung to rung, only one arm, i.e., the one with the best performance, remains at the end. Hyperband (HB) [29] extends this concept by iterating over different numbers of initial arms $n_{arms}$ (also referred to as brackets) to evaluate.

However, when performing HPO at a larger scale, these methods are sensitive to so-called stragglers. To determine the combinations belonging to the under- and top-performing half, the performance measurement for all combinations needs to be available, which means that faster trials need to wait for the slower ones. ASHA addresses this scalability problem by deciding on a rolling basis which trials are worth continuing. When two trials have finished their initial number of iterations, the trial with the better performance is promoted. At the same time, the other trial is paused until the performance of the next completed trial can be juxtaposed. In contrast to HB, ASHA is mostly performed with only a single bracket, and was evaluated on up to 500 GPUs in [30].

Another possibility of finding a minimum of the objective function $f$ is to use black-box optimization methods such as BO. The idea behind BO is to use a probabilistic model of $f$ that is based on data points observed in the past. In the case of HPO, this corresponds to finding new promising hyperparameter combinations based on the performance of past combinations. The Bayesian Optimization and HyperBand (BOHB) algorithm [11] combines the BO process with HB for scheduling. To this aim, HB is used to choose the number of hyperparameter configurations and their assigned budget, while BO is used to choose the hyperparameters by deploying a tree parzen estimator [7].

The mentioned methods have in common that they focus only on identifying the most promising arm and delivering that hyperparameter combination as a result at the end of a run. In contrast, RASDA also ensures the full training of the best combination to yield a complete model.

### 2.2. Resource-Adaptive Schedulers

Most of the existing successive halving-based HPO schedulers treat the number of epochs or training time as resources (also known as fidelity in the literature). It is, however, also possible to treat the spatial amount of computational resources, e.g., the number of GPU, used for training a model as a fidelity. A low-fidelity measurement then corresponds to the performance of a NN trained with a small number of devices. The most relevant existing HPO schedulers that focus on this computational resource-adaptive scheduling are presented in the following.

HyperSched [32] introduces a scheduler to dynamically allocate resources in time and space to the best-performing hyperparameter trials. It thereby not only identifies the most promising model but also trains it – ideally fully – by a fixed deadline. The main novelty of the algorithm is its deadline awareness, which means that it schedules fewer new trials as it approaches the deadline. This way, the exploration of new configurations is stopped in favor of deeper exploitation of the running trials. HyperSched is evaluated in [32] on different CV benchmarking datasets on up to 32 GPUs on Cloud computing instances.

Rubberband [40] extends HyperSched by leveraging the elasticity of the Cloud for the task of scheduling HPO workloads. It takes into account not only the performance of a combination but also the financial costs of a GPU hour, with the goal of minimizing the costs of an HPO job. Based on the idea of diminishing returns when scaling the training of a single model, the algorithm de-allocates resources (and thus saves costs) from less-promising trials, once a promising trial has been identified. It also creates a resource allocation plan a priori the run to optimize the performance of the single trials that are trained via distributed DL. The resource allocation plan is initialized with an initial burn-in period during which training latencies and scaling performance of trials are measured.

Sequential Elimination with Elastic Resources (SEER) [10] further takes advantage of the elasticity in the cloud by adaptively allocating and de-allocating compute resources during the HPO run. At the same time, it focuses on maximizing the accuracy of trials, in combination with minimizing the total financial cost. Therefore, it limits the amount of workers allocated to the top trials once sub-linear scaling performance sets in.

Both Rubberband and SEER rely heavily on the adaptive allocation and de-allocation of GPU instances, which is possible in an elastic cloud setting but not on HPC systems, where the amount of GPUs allocated to the overall HPO job is usually static. HyperSched, meanwhile, focuses on maximizing the performance by the deadline. In contrast, the proposed RASDA method aims to deliver the best-possible result in the shortest amount of time.

### 2.3. Other HPO Algorithms and Libraries

Many other algorithms and libraries for performing HPO exist. These include BO-based libraries such as Dragonfly [25] and SMAC [34], allowing the user to select different surrogate models and acquisition functions. Optuna [3] also relies on BO and provides automated tracking and visualization of trials. Since parallel computing resources have become increasingly available in recent years, several algorithms have emphasized large-scale, distributed HPO: DeepHyper [6] focuses on performing asynchronous BO on HPC systems and has been applied to several scientific use-cases [5, 22, 35]. Distributed evolutionary optimization can be performed with Propulate [46] and Population Based Training (PBT) [20].

While most of these libraries support multi-fidelity HPO, none of them so far supports performing resource-adaptive scheduling of trials, which is, however, supported by RASDA.

### 2.4. Data-Parallel Deep Learning

Data-parallel training is a technique to reduce the runtime of the training of DL models on large datasets by using multiple devices, such as GPUs. In data-parallel training, the training dataset $\mathcal{D}$ is divided among the number of workers $N$, where each worker is assigned an identical copy of the model to train on a distinct subset of the data $\mathcal{D}_1 \cup \mathcal{D}_2 \cup \ldots \cup \mathcal{D}_N$. Specifically,

each worker $i = 1 \dots N$ runs one model forward and backward pass with a predefined number of samples, the local batch size $BS_{local}$, of its subset of data to compute its local gradients $\Delta w_i$ with respect to the model parameters $w$. After the backward pass, these local gradients are aggregated and averaged across all workers by

$$\Delta w = \frac{1}{N} \sum_{i=1}^{N} \Delta w_i, \tag{1}$$

The averaged global gradient is then used to update the model parameters on all workers every $BS_{global} = BS_{local} \cdot N$ samples [31]. To remain computationally efficient, each worker needs a sufficient amount of data to run the training, thus $BS_{local}$ needs to be large. At the same time, $BS_{global}$ increases linearly with $N$. When $BS_{global}$ becomes too large, it can impact the generalization performance for two reasons. First, the number of optimizer updates per epoch decreases, as an update is performed every $BS_{global}$ samples. This can be addressed to some extent by scaling the learning rate with the number of devices [14]. This approach is, however, infeasible for an extremely large $BS_{global}$, since in such a case also the learning rate becomes too large. Second, Stochastic Gradient Descent (SGD) with large batch sizes tends to converge to *sharp minima* [17] which does not generalize well, see [26] for more details.

## 3. Resource-Adaptive Successive Doubling Algorithm

This section presents details on RASDA in Sec. 3.1 and provides an explanation on how issues with large batch size training, cf. Sec. 2.4, are addressed in Sec. 3.2.

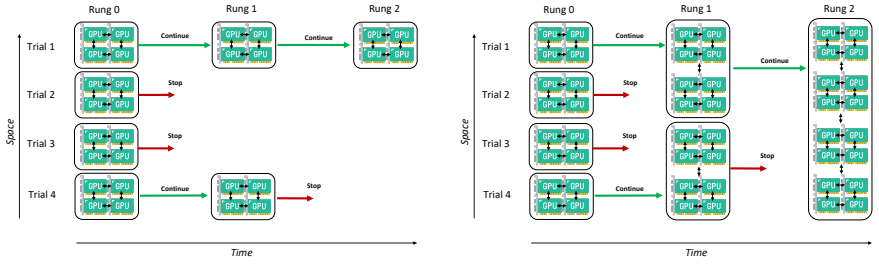
### *3.1. Algorithm Design and Implementation*

The main idea of RASDA is to combine a successive halving step in the time domain, i.e., train more promising configurations for longer, and a successive doubling step in the spatial domain, i.e., allocate more workers to more promising configurations. This way, when reaching a rung milestone, the worst-performing trials are terminated (halving in time) and the free workers are allocated to the top-performing trials (doubling in space), see Fig. 2. The additional workers are then used to increase the parallelism of the data-parallel training of the configuration, which leads to faster training times.

For the re-allocation of workers, a second successive doubling routine in addition to the successive halving routine of ASHA is used (the resource allocation part is described in Alg. 1): All trials start out with an initial number of workers (`base_resources`). When a trial reports a new `trial_result`, it is first checked if the current `training_iteration`, e.g., the current epoch, corresponds to one of the rung `milestones`. At every rung `milestone`, the (plain) ASHA scheduler then reduces the number of running trials by the reduction factor `rf`. The resources for all trials that are allowed to continue are then increased with the scaling factor `sf` by the RASDA scheduler, yielding the `new_resources` for the trial (following Alg. 1). If the reduction and scaling factors are equal, i.e. `rf` = `sf`, all workers are continuously allocated to a trial. In practice, however, some trials do run faster than others. The advantage of the ASHA and RASDA scheduler is that they both perform asynchronous halving and doubling, i.e., top-performing trials are promoted to the next rung even if not all trials in the current rung have reached their milestones. This reduces idling times between halving steps. It should be noted that due to this asynchronous execution, the percentage of trials terminated at each milestone can be smaller than `rf`. As the total number of workers in the system is a constant, the trials that are allowed to continue might need to wait until their new resource requirements are met.

At these rung `milestones`, two processes occur: the (plain) ASHA scheduler reduces the number of running trials by the reduction factor `rf`, while Alg.1 handles the reallocation of GPU resources among the remaining trials.

The total number of rungs and their corresponding mile-




(a) ASHA (b) RASDA

Figure 2: Comparison of (plain) ASHA, performing successive halving only in the time domain, and RASDA, performing successive halving in the time and successive doubling in the space domain at the same time on an GPU cluster. In the RASDA case, when a trial is terminated, its workers are allocated to the more promising trials to increase the parallelism of the data-parallel training. Black arrows indicate communication of gradients between GPUs.

stones in the RASDA scheduler are calculated based on the minimum and maximum iterations min_$t$ and max_$t$, along with the scaling factor `sf`, as

$$\texttt{num\_rungs} = \left\lfloor \frac{\log\left(\frac{\max\_t}{\min\_t}\right)}{\log(\texttt{sf})} \right\rfloor, \tag{2}$$

$$\texttt{rung\_milestones} = \min\_t \cdot \texttt{sf}^k, \tag{3}$$

with $k = 0, \ldots,$ `num_rungs`. This ensures a geometric progression of the milestones, as described for ASHA by Li et al. [30].

The algorithm is implemented with Ray Tune [33], an open-source library for performing distributed HPO. Ray Tune orchestrates the optimization process by launching a single head node and several worker nodes on an HPC cluster. The head node then connects to the worker nodes and starts the trials. During training, the worker nodes report their current status including performance metrics to the head node that makes scheduling decisions, such as termination or continuation of new trials.

Ray Tune already features implementations of several successive halving methods. The implementation of RASDA therefore relies on the implementation of ASHA that exists already inside of Ray Tune for performing the time-wise successive halving. For the spatial successive doubling, RASDA makes use of the *ResourceChangingScheduler* interface[2], enabling the modification of resource requirements for trials. At each milestone, the trial is saved, including the current weights of the model. If the decision is made to continue the training, the trial is relaunched with the new resource requirements. It should be noted, that the ASHA implementation of Ray Tune has some minor differences to the original algorithm in [30]. However, empirical evidence shows that these differences do not impact performance.

The data-parallel training part is handled by the PyTorch-DDP library [31], which uses the NVIDIA Collective Communications Library (NCCL) backend[3] for communication and gradient synchronization.

### *3.2. Large Batch Training*

Recall from Sec. 2.4 that scaling the data-parallel training to a large number of devices and increasing $BS_{global}$ can impact the generalization performance of models. The following provides an intuitive explanation of how this issue is addressed by the RASDA scheduler.

McCandlish et al. [39] empirically studied large batch training for various models: they introduce the Gradient Noise Scale (GNS) metric, which serves as a noise-to-signal measure of the training progress. In theory, if the true gradient $G_{true}$ from performing full-batch Gradient Descent without the stochastic component would be available, it would be possible to compute the a simple version of the GNS by

$$\text{GNS}_{simple} = \frac{\text{tr}(\Sigma)}{|G_{true}|^2}, \tag{4}$$

where $\Sigma$ is the per-data-sample covariance matrix of $G_{true}$. Essentially, the nominator measures the noise of the gradient, while the denominator measures its magnitude. As the DL model converges, the gradient decreases in size, which results in an increase of the GNS over training time. McCandlish et al. use an approximation to compute the GNS based on the estimated stochastic gradient $G_{est}$ and confirm that the GNS indeed increases over time.

Based on the GNS, Qiao et al. [42] introduce the concept of "statistical efficiency" of the DL training, measuring the amount of training progress made per data sample processed in a batch. The key insight is that when the GNS is low, there is no benefit for the learning progress in adding more data samples to the batch (thus increasing $BS_{global}$), as the stochastic gradient $G_{est}$ is a precise approximation of $G_{true}$ already. However, when the GNS is high, adding more data samples to the batch reduces the noise and leads to a better gradient approximation. As the GNS starts out small and increases over time, this justifies the usage of larger batch sizes during the later part of training. This approach has also been used successfully for HPO and scheduling tasks in the past [42, 2].

Additionally, Smith et al. [44] find that increasing $BS_{global}$ over time has a similar effect as decaying in the learning rate, which is common practice in DL nowadays [36]. Based on these findings, the following two insights can be derived:

- Training with a small $BS_{global}$ generally helps generalization and is computationally efficient at the beginning of training, in terms of the training progress per processed data sample.
- Increasing $BS_{global}$ over time and using a large $BS_{global}$ as the model is converging is computationally efficient as well.

[2] ResourceChangingScheduler (version 2.8.0): `https://docs.ray.io/en/latest/tune/api/doc/ray.tune.schedulers.ResourceChangingScheduler.html`

[3] NCCL backend: `https://github.com/NVIDIA/nccl`

**Algorithm 1** Resource Adaptive Successive Doubling Method

**Input:** `trial_result, base_resources, sf, milestones`

```
if trial_result["training_iteration"] ∈ milestones then
    current_rung ← milestones.index(trial_result["training_iteration"])
    new_resources ← base_resources × sf^current_rung
    return new_resources
else
    return None
end if
```

This aligns well with the scheduling of the RASDA algorithm. In the beginning, the trials train with a small $BS_{global}$, i.e., the number of workers allocated for the data-parallel training is small. As time progresses, $BS_{global}$ increases with each resource doubling step, as more and more workers are allocated to the data-parallel training. The evaluation in Sec. 5 shows that by leveraging this approach, the generalization capabilities of the final models match or exceed those of models that are continuously trained with a small $BS_{global}$.

Another crucial point is the correct scaling of the learning rate with the batch size. In the evaluation in Sec. 5, the learning rate is scaled linearly with the number of workers, i.e., up to a factor of 8×, when using SGD [28]. Furthermore, it follows a square-root scaling rule when using Adaptive Moment Estimation (ADAM) [37]. In the case of re-scaling, the learning rate is not immediately scaled to a larger value. Instead, there is a warm-up over one or two epochs. This re-scaling parameter is included as a hyperparameter in the search space, see Tab. 1. Thereby, the HPO run automatically optimizes towards learning stability.

### *3.3. Performance Optimization*

To ensure efficient performance, several additional optimizations are made to the trials in the HPO loop. This includes selecting the $BS_{local}$ sufficiently large such that it fills the GPU memory in addition to the model for each of the applications. As the training datasets have to be loaded by each trial in parallel when performing HPO, they are loaded into shared memory when they fit in size. Training datasets that do not fit into shared memory are stored on a partition of the file system with high bandwidth to avoid bottlenecks. For data loading, the native PyTorch data loader as well as the NVIDIA DALI library[4] are used.

A preliminary study determined that saving the model weights into a checkpoint too often can lead to bottlenecks [1]. Therefore, the checkpoint frequency is reduced to every five epochs and the rung milestones of the ASHA and RASDA scheduler are adjusted accordingly. Ray Tune needs an initial start-up time to launch the head node and all connected worker nodes. As this is the same for ASHA and RASDA, these timings are excluded from the measurements.

## 4. Application Cases

To assess the proposed RASDA scheduler, its performance is evaluated across a range of different tasks from the CV, CFD, and AM domain. These cases feature various models with different hyperparameters to optimize as well as training datasets of different sizes. The application domains and the set-up of these tasks is described in the following.

[4]DALI: `https://developer.nvidia.com/dali`

Table 1: Search space for the experiments, comprised of several optimizer-related and architectural parameters. Superscripts indicate which hyperparameters are used as search space for which applications: CV, CFD, AM. The "re-scaling warm-up´´ parameter handles the gradual increase of the learning rate when the number of devices and with it $BS_{global}$ is increased.

| **Hyperparameter** | **Type** | **Range** |
|---|---|---|
| Learning rate$^{\text{CV, CFD, AM}}$ | float | log[1e-5, 1] |
| Weight decay$^{\text{CV, CFD, AM}}$ | float | log(0, $1e-1$] |
| Initial warm-up$^{\text{CV, CFD, AM}}$ | int | [1,2,3,4,5] |
| Optimizer$^{\text{CV, CFD, AM}}$ | cat | ["sgd", "adam"] |
| Layer initialization$^{\text{CV, CFD}}$ | cat | ["kaiming" [[15]], "xavier"[13]] |
| Activation function$^{\text{CV, CFD}}$ | cat | ["ReLU", "LeakyReLU", "SELU", "Tanh", "Sigmoid"] |
| Convolution kernel size$^{\text{CV, CFD}}$ | int | [5,7,9] |
| Re-scaling warm-up$^{\text{CFD, AM}}$ | int | [1,2] |
| Patch size$^{\text{AM}}$ | int | [2, 4] |
| Depth$^{\text{AM}}$ | int | [1, 2, 4] |
| Number of attention heads$^{\text{AM}}$ | int | [3, 6, 12, 24] |
| MLP ratio$^{\text{AM}}$ | float | [1., 2., 3., 4.] |

### *4.1. Computer Vision*

For the CV domain, the hyperparameters of a ResNet50 [16] trained on the ImageNet dataset [43] are optimized, as this is still one of the most important reference benchmarks [38]. The ImageNet dataset contains 1,281,167 training images and 50,000 validation images divided into 1,000 object classes. In TFRecord file format, the dataset is approximately 146 Gigabyte (GB) in size.

The ResNet follows a basic CNN architecture with multiple residual connections between layers. The HPO search space for the ResNet includes several architectural hyperparameters, e.g., the type of activation functions or size of the input convolution kernel, as well as optimizer-related parameters, such as the learning rate or weight decay, see Tab. 1 for an exhaustive list.

All models are trained for min $\_t = 5$ to max $\_t = 40$ epochs and a reduction and scaling factor $\texttt{sf} = \texttt{rf} = 2$ is chosen for the schedulers. Following Eq. (3), this results in rung milestones at epochs 5, 10, 20, and 40.

As classification accuracy score, the percentages of the correctly classified training, validation, and test images are computed.

### *4.2. Additive Manufacturing*

The AM dataset is taken from the RAISE-LPBF benchmarking dataset [9], which includes a selection of high-speed video recordings at 20,000 frames per second of a laser powder bed fusion processes for stainless steel. The laser power and speed parameters are systematically varied. The goal is to reconstruct the power and speed of the laser from this video input. By comparing the predicted laser parameters with the pre-set parameters of the machine producing the laser, anomalies in the printing process can be detected faster, leading to

more efficient quality control. The base ML model used for this task is a SwinTransformer [47], with the HPO search space consisting of multiple, Transformer-specific architectural and optimizer-related parameters, such as the number of attention heads, see Tab. 1. The model is trained on the C027 cylinder with a 80/20 split for training and validation and is approximately 60 GB in size. It is evaluated on the C028 cylinder for testing purposes. The Mean-Squared Error (MSE) between predicted and actual laser power and speed is computed to assess the accuracy of the SwinTransformer. All models are trained for min $\_t = 5$ to max $\_t = 20$ epochs and a reduction and scaling factor $\texttt{sf} = \texttt{rf} = 2$ is chosen for the schedulers. Following Eq. (3), this results in rung milestones at epochs 5, 10, and 20.

### *4.3. Computational Fluid Dynamics*

The CFD dataset contains actuated turbulent boundary layer flow data, generated from a simulation [4]. The CFD dataset is stored in HDF5 file format and comprises several widths. In this study, widths of 1,000, 1,200, and 1,600 are used as training dataset (approximately 4.8 TB in size). Width of 1,800 and 3,000 are used as validation and test datasets (approximately 3.5 TB in size) to assess extrapolation performance. Altogether, the dataset is approximately 8.3 TB in size.

A convolutional autoencoder, selected from the AI4HPC repository [18, 19], is employed for flow reconstruction. The autoencoder comprises an encoder, a decoder, and a latent space representing a compressed, lower-dimensional version of the input. Both the encoder and decoder include four convolutional layers. In the encoder, the initial two layers perform downsampling to compress the data, while in the decoder, they perform up-sampling to decompress the data in the latent space. The remaining layers perform regular convolution. The HPO search space consists of the type of activation function as an architectural parameter and several optimizer-related ones, see Tab. 1.

The autoencoders are trained for min $\_t = 5$ to max $\_t = 40$ epochs and a reduction and scaling factor $\texttt{sf} = \texttt{rf} = 2$ is chosen for the schedulers. Following Eq. (3), this results in rung milestones at epochs 5, 10, 20, and 40.

The MSE between the input and the reconstructed output flow field is computed and used to assess the accuracy of the autoencoders. As a further measure of solution quality, also the relative reconstruction error is computed on the test set.

## 5. Results

This section presents the experimental results of running the proposed algorithm on two supercomputer systems, which are introduced in Sec. 5.1. Section 5.2 focuses on the scaling performance of the RASDA algorithm on up to 1,024 GPUs, while Sec. 5.3 compares the RASDA against the plain ASHA scheduler without any resource adaptation.

### *5.1. Supercomputers*

The two supercomputer modules used for the experiments in this study are both located at the Jülich Supercomputing Centre.

The first system is the JURECA-DC-GPU module [24] consisting of a total of 192 accelerated compute nodes. Each node is equipped with two AMD EPYC 7742 CPUs with 128 cores clocked at 2.25 GHz and four NVIDIA A100 GPUs, each with 40 GB high-bandwidth memory. The second HPC system is the JUWELS BOOSTER module [27] consisting of a total of 936 compute nodes. Each node is equipped with two AMD EPYC Rome 7402 CPUs with 48 cores clocked at 2.8 GHz, and four NVIDIA A100 GPU with 40 GB high-bandwidth memory. The main difference between the two systems is the number of InfiniBand interconnects: the JURECA-DC-GPU system features only two per node, while the JUWELS BOOSTER has four per node and therefore a higher network transmission bandwidth.

As of June 2024, both supercomputers are among the top 10% most energy-efficient supercomputers in the world, according to the GREEN500 list[5].

### *5.2. Scaling Performance*

To evaluate the scalability of the RASDA algorithm, two weak scaling experiments, where the number of HPO configurations to evaluate is increased with the number of GPUs, are conducted with a lower number of training epochs. For this purpose, the CV application case as a representative benchmark for DL workloads is selected. It should be noted that while the asynchronous nature of the plain ASHA algorithm naturally leads to good scalability [30], the goal of this study is to demonstrate that the additional resource allocation mechanism in RASDA maintains this favorable scaling behavior.

The first weak scaling experiment considers a smaller scale of 8 to 64 GPUs. The runtime and accuracy of the RASDA algorithm is compared to the plain ASHA algorithm for training a ResNet50 on the ImageNet dataset for 20 epochs, see Fig. 3. It can be seen that on all scales (from 8 to 64 GPUs), the RASDA algorithm achieves consistently lower runtimes up to a factor of 1.45 faster than its ASHA counterpart while matching the final test set accuracy in almost all cases.

The second scaling experiment considers a large scale of 128 to 1,024 GPUs, see Fig. 4. The weak scalability of the RASDA algorithm is evaluated by training a ResNet for six epochs. The results show that the algorithm maintains a high parallel efficiency of $> 0.84$ on up to 1,024 GPUs.

It should be noted that strong scaling experiments that keep the number of hyperparameter configurations consistent across all scales are generally infeasible for this type of HPO workload, as evaluating a large number of configurations on a small number of GPUs would take too long.

### *5.3. Speed-Ups and Accuracy*

To evaluate the performance of the RASDA algorithm in terms of speed-up and accuracy and to juxtapose it to the plain ASHA algorithm considering the application cases, the number of training epochs is increased within the min $\_t$ and max $\_t$ range specified in Sec. 4. The general results for the three application case, averaged over three different runs for all application

[5] GREEN500: https://top500.org/lists/green500/list/2024/06/

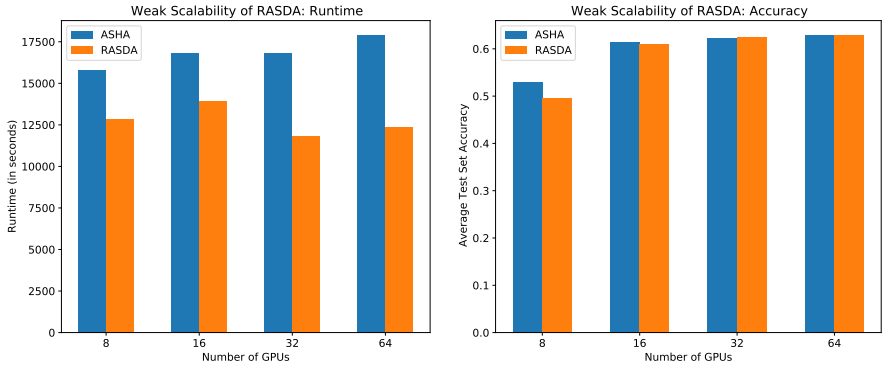



Figure 3: Comparison of ASHA and RASDA for training a ResNet50 model on ImageNet for 20 epochs on different scales on the JURECA-DC-GPU system.

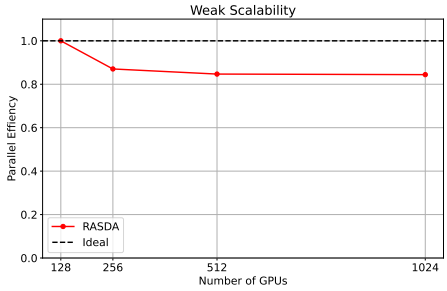



Figure 4: Weak scalability of the RASDA algorithm on up to 1,024 GPUs on the JUWELS BOOSTER system, including ideal scalability for comparison.

cases, are presented in Tab. 2, Tab. 3, and Tab. 4. The solution quality over time is presented in Fig. 5, an in-depth performance analysis of the runtimes per epoch is given in Fig. 6, and the change of batch size and number of GPUs per trial is depicted in Fig. 7. The results correspond to an exemplary best-performing trial from one of the three runs. The following paragraphs provide a more detailed discussion of these tables and figures.

For the CV application case, a total of 32 hyperparameter combinations are evaluated simultaneously on 64 GPUs on the JURECA-DC-GPU system, with each parallel trial starting with two GPUs. Compared to the plain ASHA approach, the RASDA algorithm reduces the overall average runtime of the HPO process by a factor of $\approx$ 1.71 from 527 to 308 minutes, see Tab. 2. The average solution quality, i.e., the training, validation, and test set accuracy of the best trial discovered during the process, slightly outperforms the ones of the plain ASHA. This indicates that scaling the batch size and the learning rate during the training process does not impact the learning process in this case. A closer look at one of the best-performing trials in Fig. 6 reveals that indeed the average runtime decreases in the RASDA case once the resource adaptation in space sets in after the first five epochs. As can be seen in Fig. 7, $BS_{global}$ increases from 256 to 2,048 during the training and the number of GPUs from 2 to 16 per trial for the RASDA case, while both stay constant in the plain ASHA case. The plot of the validation accuracy over the number of epochs in Fig. 5 confirms that RASDA slightly outperforms the ASHA approach in terms of solution quality.

For the AM application case, the HPO process evaluates 16 configurations, using a total of 128 GPUs on the JURECA-DC-GPU system. The trials start out with 8 GPUs each, which increases to 32 GPUs for the top-performing trials, at the same time increasing $BS_{global}$ from 64 to 256. As the models are only trained for a total amount of 20 epochs (due to the long training times of transformer models), only two resource-doubling steps, i.e., at epoch 5 and epoch 10, take place, see Fig. 7. Table 3 provides an overview of the results in terms of runtime and solution quality. In comparison with the plain ASHA algorithm, a speed-up by a factor of 1.52 is achieved, reducing the required HPO runtime of the models from 96 to 63 minutes. On both the validation and test dataset, the best configuration found by RASDA again outperforms the one found with the plain ASHA after 20 epochs, as can be seen in Fig. 5.

The CFD application case features the largest dataset used in this study. The whole HPO process evaluates 16 configurations on 128 GPUs simultaneously on the JUWELS BOOSTER module. Each trial starts with 8 GPUs, which is increased over time to 64 GPU by the RASDA algorithm. As can be seen from Tab. 4, the most significant speed-up with a factor of $\approx$ 1.9 is achieved in this case, with RASDA reducing the runtime of the HPO process from 325 to 170 minutes. In this case, also the average MSE decreases by a factor of $\approx$ 1.88. This is likely due

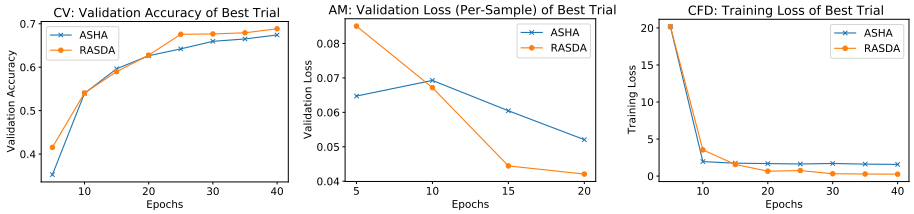



Figure 5: Exemplary comparison of the performance (in terms of validation accuracy, training loss, and validation loss) of the best configuration found by ASHA and RASDA for the different application cases.

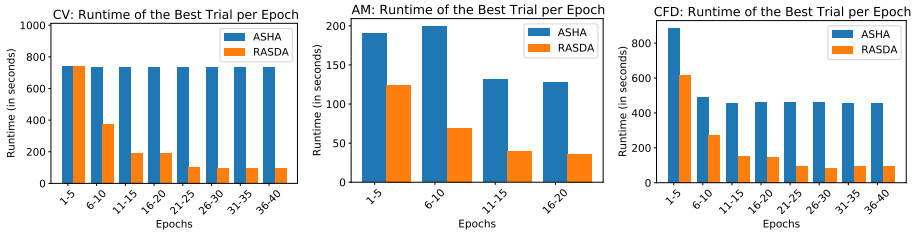



Figure 6: Exemplary comparison of the runtime per epoch of the best configuration found by ASHA and RASDA for the different application cases. Note that for the CFD and AM case, the architectural parameters chosen by the respective HPO method also influence the model size, which is why here differences in runtime can be observed already during the first five epochs.

to the even better generalization capabilities caused by increasing the batch size over time (following the insights explained in Sec. 3.2). Obviously, this outperforms just annealing of the learning rate. This observation is in line with the findings of Smith et al. [44]. RASDA also achieves a low relative reconstruction error of just 1.15% on the test set.

In general, the most substantial speed-up is established on the largest dataset from the CFD domain. This is expected, as with a larger dataset, the benefit of adding more GPUs to the data-parallel training loop also increases. It is additionally interesting to observe that the speed-ups can be attained on both the JURECA-DC-GPU and JUWELS BOOSTER systems, although the latter features twice the network bandwidth. While RASDA already yields substantial benefits on JURECA-DC-GPU with its moderate network infrastructure, the doubled network bandwidth of JUWELS BOOSTER further amplifies these speed-ups, highlighting how the approach particularly profits from fast interconnects.

### *5.4. Performance at 1,024 GPUs Scale*

While the superiority of RASDA over plain ASHA has been confirmed in the previous experiments using 64 and 128 GPUs, a final RASDA experiment on a 1,024 GPU scale is conducted on the JUWELS BOOSTER system. Again, using the CFD application case, the number of configurations to be evaluated is increased to 64, with each trial starting with 16 GPUs. The models are trained for $\min\_t = 5$ and $\max\_t = 20$ epochs.

Table 2: HPO for the CV application case, trained for 40 epochs on 64 GPUs on the JURECA-DC-GPU system. Results are averaged over three random seeds. Better results (↑ or ↓ depending on the metric) are underlined.

| **Metric** | **ASHA** | **RASDA** | **Diff.** |
|---|---|---|---|
| Train Accuracy ↑ | 0.6976 | <u>0.7310</u> | 1.05× |
| Val Accuracy ↑ | 0.6728 | <u>0.6813</u> | 1.01× |
| Test Accuracy ↑ | 0.6688 | <u>0.6766</u> | 1.01× |
| Runtime (in seconds) ↓ | 31637 | <u>18502</u> | 1.71× |

Table 3: HPO for the AM application case, trained for 20 epochs on 128 GPUs on the JURECA-DC-GPU system. Results are averaged over three random seeds. Better results (↑ or ↓ depending on the metric) are underlined. For better comparison the metrics were recomputed on a per-sample basis after the run.

| **Metric** | **ASHA** | **RASDA** | **Diff.** |
|---|---|---|---|
| Val MSE ↓ | 0.0455 | <u>0.0404</u> | 1.12× |
| Test MSE ↓ | 0.0554 | <u>0.0516</u> | 1.07× |
| Runtime (in seconds) ↓ | 5784 | <u>3803</u> | 1.52× |

The HPO run took three hours and resulted in an improved model with a validation MSE of $\approx 3.63 \times 10^{-7}$, a test MSE of $\approx 4.88 \times 10^{-8}$ and a relative test error of $\approx 0.0016$. Depending on the metric, this is a 7 to 49 times increase in solution quality, compared to the results of the HPO run on 128 GPUs (see Tab. 5), which highlights the potential of large-scale HPO for scientific ML.

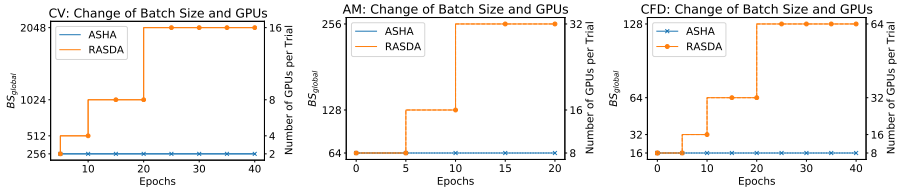



Figure 7: Comparison of the global batch size and the number of GPUs per trial for ASHA and RASDA for the different application cases.

Table 4: HPO for the CFD application case, trained for 40 epochs on 128 GPUs on the JUWELS BOOSTER module. Results are averaged over three random seeds. Better results (↑ or ↓ depending on the metric) are underlined.

| Metric | ASHA | RASDA | Diff. |
|---|---|---|---|
| Val MSE ↓ | $5.28 \times 10^{-6}$ | $\underline{2.81 \times 10^{-6}}$ | 1.88× |
| Test MSE ↓ | $4.42 \times 10^{-6}$ | $\underline{2.40 \times 10^{-6}}$ | 1.84× |
| Test Relative Error ↓ | 0.0185 | $\underline{0.0115}$ | 1.61× |
| Runtime (in seconds) ↓ | 19487 | $\underline{10242}$ | 1.90× |

Table 5: Large-scale HPO for the CFD application case, evaluating 64 configurations, trained for a maximum of 20 epochs on 1,024 GPUs on the JUWELS BOOSTER module, including relative improvement to the HPO run on 128 GPUs.

| Metric | RASDA - 1,024 GPUs | vs. 128 GPUs |
|---|---|---|
| Val MSE | $3.63 \times 10^{-7}$ | 7.74× |
| Test MSE | $4.88 \times 10^{-8}$ | 49.22× |
| Test Rel. Error | 0.0016 | 7.17× |

## 6. Summary and Outlook

RASDA, a novel resource-adaptive successive doubling algorithm for HPO, suitable for running on HPC systems, was introduced. The key idea is to not only perform successive halving in time and let promising configurations train for longer (as is already the case in plain ASHA), but to combine it with successive doubling in space and allocate more computational resources to the data-parallel training of promising configurations.

The RASDA method was evaluated extensively on a standard benchmarking task in the CV domain as well as on two large datasets (up to 8.3 TB in size) from the CFD and AM domains. The results confirm that RASDA leads in these cases to speed-ups up to a factor of $\approx 1.9$ in comparison to the ASHA algorithm.

Another property of RASDA is that it progressively scales up the global batch size of the trials as it adds more GPUs to their training loops. This helps them to avoid the degradation in solution quality, which is usually associated with large batch training. Remarkably, the approach did enhance the solution quality, aligning with literature findings suggesting that increasing the batch size can match or surpass the effects of learning rate annealing.

In addition, this study represents the first application of systematic HPO to a scientific dataset at the TB scale. A comparison of the application of RASDA on 128 and 1024 GPUs revealed a significant improvement in model performance. Specifically, the larger-scale application identifies a model that is significantly superior in solution quality. These results demonstrate the scalability and efficiency of the RASDA method, thus paving the way for the application of HPO methods on current and future Exascale supercomputers. For future work, the optimal timing for scaling the batch size (through the addition of more GPUs to the data-parallel training loop) should be investigated more thoroughly. Furthermore, the impact of scaling the batch size on various hyperparameters beyond the learning rate (such as the weight decay values) warrants a deeper exploration.

## Acknowledgements

This research has been performed in the CoE RAISE project, which received funding from the European Union's *Horizon 2020 – Research and Innovation Framework Programme* H2020-INFRAEDI-2019-1 under grant agreement no. 951733.

The authors gratefully acknowledge computing time on the supercomputer JURECA [24] at Forschungszentrum Jülich under grant no. raise-ctp2. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS Supercomputer JUWELS [23] at Jülich Supercomputing Centre (JSC).

## References

[1] Aach, M., Sarma, R., Inanc, E., Riedel, M., Lintermann, A., 2023. Short paper: Accelerating hyperparameter optimization algorithms with mixed precision, in: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, ACM. p. 1776–1779. doi:10.1145/3624062.3624259.

[2] Aach, M., Sedona, R., Lintermann, A., Cavallaro, G., Neukirchen, H., Riedel, M., 2022. Accelerating hyperparameter tuning of a deep learning model for remote sensing image classification, in: IGARSS 2022 - 2022 IEEE International Geoscience and Remote Sensing Symposium, IEEE. pp. 263–266. doi:10.1109/IGARSS46834.2022.9883257.

[3] Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M., 2019. Optuna: A next-generation hyperparameter optimization framework, in: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, ACM. p. 2623–2631. URL: https://doi.org/10.1145/3292500.3330701, doi:10.1145/3292500.3330701.

[4] Albers, M., Meysonnat, P.S., Fernex, D., Semaan, R., Noack, B.R., Schröder, W., Lintermann, A., 2023. Actuated Turbulent Boundary Layer Flows Dataset. URL: https://juser.fz-juelich.de/record/996125, doi:10.34730/5dbc8e35f21241d0889906136cf28d26. the authors of this dataset are Albers, M., Meysonnat, P. S., Fernex, D., Semaan, R., Noack, B. R., and Schröder, W. The author Lintermann, A. manages the distribution of this dataset.

[5] Balaprakash, P., Egele, R., Salim, M., Wild, S., Vishwanath, V., Xia, F., Brettin, T., Stevens, R., 2019. Scalable reinforcement-learning-based neural architecture search for cancer deep learning research, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM. URL: https://doi.org/10.1145/3295500.3356202, doi:10.1145/3295500.3356202.

[6] Balaprakash, P., Salim, M., Uram, T.D., Vishwanath, V., Wild, S.M., 2018. Deephyper: Asynchronous hyperparameter search for deep neural networks, in: 2018 IEEE 25th International Conference on High Performance Computing, IEEE. pp. 42–51. doi:10.1109/HiPC.2018.00014.

[7] Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B., 2011. Algorithms for hyper-parameter optimization, in: Shawe-Taylor, J., Zemel, R., Bartlett, P., Pereira, F., Weinberger, K.Q. (Eds.), Proceedings of the 24th International Conference on Neural Information Processing Systems, Curran Associates, Inc.

[8] Bergstra, J., Bengio, Y., 2012. Random search for hyper-parameter optimization. Journal of Machine Learning Research 13, 281–305. URL: http://jmlr.org/papers/v13/bergstra12a.html.

[9] Blanc, C., Ahar, A., De Grave, K., 2023. Reference dataset and benchmark for reconstructing laser parameters from on-axis video in powder bed fusion of bulk stainless steel. Additive Manufacturing Letters 7, 100161. doi:https://doi.org/10.1016/j.addlet.2023.100161.

[10] Dunlap, L., Kandasamy, K., Misra, U., Liaw, R., Jordan, M., Stoica, I., Gonzalez, J.E., 2021. Elastic hyperparameter tuning on the cloud, in: Proceedings of the ACM Symposium on Cloud Computing, ACM. p. 33–46. URL: https://doi.org/10.1145/3472883.3486989, doi:10.1145/3472883.3486989.

[11] Falkner, S., Klein, A., Hutter, F., 2018. BOHB: Robust and efficient hyperparameter optimization at scale, in: Proceedings of the 35th International Conference on Machine Learning, PMLR. pp. 1436–1445. URL: https://proceedings.mlr.press/v80/falkner18a.html.

[12] Feurer, M., Hutter, F., 2019. Hyperparameter Optimization. Springer International Publishing, Cham. pp. 3–33. URL: https://doi.org/10.1007/978-3-030-05318-5_1, doi:10.1007/978-3-030-05318-5_1.

[13] Glorot, X., Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks, in: Teh, Y.W., Titterington, M. (Eds.), Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR. pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[14] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., He, K., 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. doi:10.48550/ARXIV.1706.02677, arXiv:1706.02677.

[15] He, K., Zhang, X., Ren, S., Sun, J., 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, in: Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), IEEE. p. 1026–1034. doi:10.1109/ICCV.2015.123.

[16] He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778. doi:10.1109/CVPR.2016.90.

[17] Hochreiter, S., Schmidhuber, J., 1997. Flat minima. Neural Comput. 9, 1–42. doi:10.1162/neco.1997.9.1.1.

[18] Inanc, E., Sarma, R., Aach, M., Lintermann, A., 2023a. AI4HPC. URL: https://zenodo.org/record/7705421, doi:10.5281/zenodo.7705417.

[19] Inanc, E., Sarma, R., Aach, M., Sedona, R., Lintermann, A., 2023b. AI4HPC: Library to train AI models on HPC systems using CFD datasets, in: Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@NeurIPS 2023). URL: https://openreview.net/pdf?id=zQTa2XdPnP.

[20] Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W.M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., Kavukcuoglu, K., 2017. Population based training of neural networks. doi:10.48550/arXiv.1711.09846, arXiv:1711.09846.

[21] Jamieson, K., Talwalkar, A., 2016. Non-stochastic best arm identification and hyperparameter optimization, in: Gretton, A., Robert, C.C. (Eds.), Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, PMLR. pp. 240–248. URL: https://proceedings.mlr.press/v51/jamieson16.html.

[22] Jiang, S., Balaprakash, P., 2020. Graph neural network architecture search for molecular property prediction, in: 2020 IEEE International Conference on Big Data (Big Data), IEEE. pp. 1346–1353. doi:10.1109/BigData50022.2020.9378060.

[23] Jülich Supercomputing Centre, 2021. JUWELS Cluster and Booster: Exascale Pathfinder with Modular Supercomputing Architecture at Juelich Supercomputing Centre. Journal of large-scale research facilities JLSRF 7. doi:10.17815/jlsrf-7-183.

[24] Jülich Supercomputing Centre, 2021. JURECA: Data centric and booster modules implementing the modular supercomputing architecture at Jülich Supercomputing Centre. Journal of large-scale research facilities JLSRF 7. doi:10.17815/jlsrf-7-182.

[25] Kandasamy, K., Vysyaraju, K.R., Neiswanger, W., Paria, B., Collins, C.R., Schneider, J., Poczos, B., Xing, E.P., 2020. Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly. Journal of Machine Learning Research 21, 1–27. URL: http://jmlr.org/papers/v21/18-223.html.

[26] Keskar, N.S., Mudigere, D., Nocedal, J., Smelyanskiy, M., Tang, P.T.P., 2017. On large-batch training for deep learning: Generalization gap and sharp minima. doi:10.48550/arXiv.1609.04836, arXiv:1609.04836.

[27] Kesselheim, S., Herten, A., Krajsek, K., Ebert, J., Jitsev, J., Cherti, M., Langguth, M., Gong, B., Stadtler, S., Mozaffari, A., Cavallaro, G., Sedona, R., Schug, A., Strube, A., Kamath, R., Schultz, M.G., Riedel, M., Lippert, T., 2021. JUWELS booster – a supercomputer for large-scale AI research, in: Jagode, H., Anzt, H., Ltaief, H., Luszczek, P. (Eds.), High Performance Computing, Springer. pp. 453–468. doi:10.1007/978-3-030-90539-2_31.

[28] Krizhevsky, A., 2014. One weird trick for parallelizing convolutional neural networks. doi:10.48550/arXiv.1404.5997, arXiv:1404.5997.

[29] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A., 2018. Hyperband: A novel bandit-based approach to hyperparameter optimization. Journal of Machine Learning Research 18, 1–52. URL: https://jmlr.org/papers/v18/16-558.html.

[30] Li, L., Jamieson, K., Rostamizadeh, A., Gonina, E., Ben-tzur, J., Hardt, M., Recht, B., Talwalkar, A., 2020a. A system for massively parallel hyperparameter tuning, in: Dhillon, I., Papailiopoulos, D., Sze, V. (Eds.), Proceedings of Machine Learning and Systems 2 (MLSys 2020), pp. 230–246. URL: https://proceedings.mlsys.org/paper_files/paper/2020/hash/a06f20b349c6cf09a6b171c71b88bbfc-Abstract.html.

[31] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., Chintala, S., 2020b. PyTorch distributed: Experiences on accelerating data parallel training. Proceedings of Very Large Data Base Endowment 13, 3005–3018. doi:10.14778/3415478.3415530.

[32] Liaw, R., Bhardwaj, R., Dunlap, L., Zou, Y., Gonzalez, J.E., Stoica, I., Tumanov, A., 2019. HyperSched: Dynamic resource reallocation for model development on a deadline, in: Proceedings of the ACM Symposium on Cloud Computing, ACM. p. 61–73. URL: https://doi.org/10.1145/3357223.3362719, doi:10.1145/3357223.3362719.

[33] Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J.E., Stoica, I., 2018. Tune: A research platform for distributed model selection and training. doi:10.48550/ARXIV.1807.05118, arXiv:1807.05118.

[34] Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., Hutter, F., 2022. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. Journal of Machine Learning Research 23, 1–9. URL: http://jmlr.org/papers/v23/21-0888.html.

[35] Liu, X., Rüttgers, M., Quercia, A., Egele, R., Pfaehler, E., Shende, R., Aach, M., Schröder, W., Balaprakash, P., Lintermann, A., 2024. Refining computer tomography data with super-resolution networks to increase the accuracy of respiratory flow simulations. Future Generation Computer Systems 159, 474–488. doi:https://doi.org/10.1016/j.future

.2024.05.020.

[36] Loshchilov, I., Hutter, F., 2017. SGDR: Stochastic gradient descent with warm restarts, in: International Conference on Learning Representations. URL: https://openreview.net/pdf?id=Skq89Scxx.

[37] Malladi, S., Lyu, K., Panigrahi, A., Arora, S., 2022. On the SDEs and scaling rules for adaptive gradient algorithms, in: Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A. (Eds.), Proceedings of the 36th International Conference on Neural Information Processing Systems, Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/32ac710102f0620d0f28d5d05a44fe08-Paper-Conference.pdf.

[38] Mattson, P., Cheng, C., Diamos, G., Coleman, C., Micikevicius, P., Patterson, D., Tang, H., Wei, G.Y., Bailis, P., Bittorf, V., Brooks, D., Chen, D., Dutta, D., Gupta, U., Hazelwood, K., Hock, A., Huang, X., Kang, D., Kanter, D., Kumar, N., Liao, J., Narayanan, D., Oguntebi, T., Pekhimenko, G., Pentecost, L., Janapa Reddi, V., Robie, T., St John, T., Wu, C.J., Xu, L., Young, C., Zaharia, M., 2020. MLPerf training benchmark, in: Dhillon, I., Papailiopoulos, D., Sze, V. (Eds.), Proceedings of Machine Learning and Systems, pp. 336–349. URL: https://proceedings.mlsys.org/paper_files/paper/2020/file/411e39b117e885341f25efb8912945f7-Paper.pdf.

[39] McCandlish, S., Kaplan, J., Amodei, D., Team, O.D., 2018. An empirical model of large-batch training. doi:arXiv.1812.06162, arXiv:1812.06162.

[40] Misra, U., Liaw, R., Dunlap, L., Bhardwaj, R., Kandasamy, K., Gonzalez, J.E., Stoica, I., Tumanov, A., 2021. Rubberband: cloud-based hyperparameter tuning, in: Proceedings of the Sixteenth European Conference on Computer Systems, ACM. p. 327–342. doi:10.1145/3447786.3456245.

[41] Pata, J., Wulff, E., Mokhtar, F., Southwick, D., Zhang, M., Girone, M., Duarte, J., 2024. Improved particle-flow event reconstruction with scalable neural networks for current and future particle detectors. Communications Physics 7, 124. doi:10.1038/s42005-024-01599-5.

[42] Qiao, A., Choe, S.K., Subramanya, S.J., Neiswanger, W., Ho, Q., Zhang, H., Ganger, G.R., Xing, E.P., 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning, in: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), USENIX Association. pp. 1–18. URL: https://www.usenix.org/conference/osdi21/presentation/qiao.

[43] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L., 2015. Imagenet large scale visual recognition challenge. International Journal of Computer Vision 115, 211–252. doi:10.1007/s11263-015-0816-y.

[44] Smith, S.L., Kindermans, P.J., Le, Q.V., 2018. Don't decay the learning rate, increase the batch size, in: International Conference on Learning Representations. URL: https://openreview.net/pdf?id=B1Yy1BxCZ.

[45] Sumbul, G., Charfuelan, M., Demir, B., Markl, V., 2019. Bigearthnet: A large-scale benchmark archive for remote sensing image understanding, in: IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium, IEEE. pp. 5901–5904. doi:10.1109/IGARSS.2019.8900532.

[46] Taubert, O., Weiel, M., Coquelin, D., Farshian, A., Debus, C., Schug, A., Streit, A., Götz, M., 2023. Massively parallel genetic optimization through asynchronous propagation of populations, in: Bhatele, A., Hammond, J., Baboulin, M., Kruse, C. (Eds.), High Performance Computing. ISC High Performance 2023, Springer. pp. 106–124. doi:10.1007/978-3-031-32041-5_6.

[47] Yang, Y.Q., Guo, Y.X., Xiong, J.Y., Liu, Y., Pan, H., Wang, P.S., Tong, X., Guo, B., 2023. Swin3d: A pretrained transformer backbone for 3d indoor scene understanding. URL: https://arxiv.org/abs/2304.06906, arXiv:2304.06906.

# Paper IV

**Short Paper: Accelerating Hyperparameter Optimization Algorithms with Mixed Precision**

M. Aach, R. Sarma, E. Inanc, M. Riedel, and A. Lintermann

Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23. New York, NY, USA: Association for Computing Machinery (2023), pp. 1776–1779

Copyright © 2023 ACM. Reprint for this dissertation permitted.

Marcel Aach wrote the main part of the code and the paper, ran all experiments on the HPC machines and performed the analysis. The evaluation on the CFD use case was performed in collaboration with E. Inanc and R. Sarma.

# Short Paper: Accelerating Hyperparameter Optimization Algorithms with Mixed Precision

Marcel Aach
Jülich Supercomputing Centre
Jülich, Germany
University of Iceland
Reykjavík, Iceland

Rakesh Sarma
Jülich Supercomputing Centre
Jülich, Germany

Eray Inanc
Jülich Supercomputing Centre
Jülich, Germany

Morris Riedel
University of Iceland
Reykjavík, Iceland
Jülich Supercomputing Centre
Jülich, Germany

Andreas Lintermann
Jülich Supercomputing Centre
Jülich, Germany

## ABSTRACT

Hyperparameter Optimization (HPO) of Neural Networks (NNs) is a computationally expensive procedure. On accelerators, such as NVIDIA Graphics Processing Units (GPUs) equipped with Tensor Cores, it is possible to speed-up the NN training by reducing the precision of some of the NN parameters, also referred to as mixed precision training. This paper investigates the performance of three popular HPO algorithms in terms of the achieved speed-up and model accuracy, utilizing early stopping, Bayesian, and genetic optimization approaches, in combination with mixed precision functionalities. The benchmarks are performed on 64 GPUs in parallel on three datasets: two from the vision and one from the Computational Fluid Dynamics domain. The results show that larger speed-ups can be achieved for mixed compared to full precision HPO if the checkpoint frequency is kept low. In addition to the reduced runtime, small gains in generalization performance on the test set are observed.

## CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**.

## KEYWORDS

Hyperparameter Optimization, Mixed Precision, High-Performance Computing

## 1 INTRODUCTION

The performance of Machine Learning (ML) models is highly dependent on the choice of hyperparameters. The hyperparameter space is not only spanned by features of the ML architecture, such as the number of neurons or layers, but also by parameters of the optimizer, e.g., the learning rate, batch size, or regularization like weight decay. Hyperparameter Optimization (HPO) describes the systematic search process for well-performing combinations of these parameters. Since the different configurations (or "trials") are independent, the process is ideally suited to be exploited in parallel on High-Performance Computing (HPC) systems. This way, several models across different scientific domains have been improved [2, 19]. However, HPO is still computationally challenging, as the ML models and the datasets trained on are continuously growing in size. There is hence a great interest in reducing the computational complexity of HPO. From a software perspective, one method to reduce this complexity is using early stopping techniques, which terminate trials with bad performance before they are fully trained. Another approach is to apply techniques that more intelligently sample the hyperparameter space, e.g., to use Bayesian Optimization (BO) [7] or evolutionary methods [10]. From a hardware perspective, using advanced accelerators, such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs), can drastically speed up the training process. Employing mixed precision arithmetics in training has intensively been investigated [16] and is frequently used in practice for deep Neural Networks (NNs) at large scale. The literature is lacking an evaluation of this GPU training feature for HPO tasks.

Therefore, this work aims to investigate Automatic Mixed Precision (AMP) as one of the improvements for HPO using modern NVIDIA GPUs, combined with three different HPO algorithms and a random search baseline. Half and full precision computations are mixed adaptively to run NN training more efficiently. The performance of the AMP approach is evaluated by comparing the corresponding results to those obtained with full precision arithmetics. Three datasets are used as basis for comparison: two vision datasets, i.e., cifar-10 [13] and ImageNet [17], and a Computational Fluid Dynamics (CFD) dataset [1]. The search space is defined by architectural parameters, such as the number of filters in a Convolutional Neural Network (CNN), and by optimizer-related parameters, e.g., the learning rate or momentum. The evaluations are performed on the GPU partition of the JURECA-DC machine at the Jülich Supercomputing Centre [12], using 64 NVIDIA A100 GPUs per HPO run in parallel. Code for reproducing this work is available on Gitlab[1].

The work is structured as follows. In Sec. 2 the background on HPO and the AMP package is described. Section 3 details the experimental setup, which is used for generating the results presented in Sec. 4. Finally, Sec. 5 provides the conclusion and future work.

[1] https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/mixed-precision-hpo

## 2 BACKGROUND

### 2.1 Hyperparameter Optimization Methods

In mathematical notation, the performance of a NN with respect to a metric, e.g., the mean-squared error on a validation dataset, is described by a function $f : \mathcal{X} \rightarrow \mathbb{R}$, where $\mathcal{X}$ is the space of hyperparameters. The HPO process performs minimization of the objective function $f$ by discovering combinations of hyperparameters $x^* \in \mathcal{X}$ such that $x^* \in \arg\min_{x \in \mathcal{X}} f(x)$. Evaluating the objective function is costly, because it requires the complete training of the NN, which is the main reason for HPO being compute intensive.

The acceleration of HPO frequently makes use of two approaches: (1) the evaluation of $f$ on a smaller compute budget using "successive halving" [11] and (2) using better-informed hyperparameter combinations. In the first approach, all trials are run until a certain point in time (usually a few epochs), at which their performance is assessed. A fraction of the under-performing trials are terminated (early stopping), while continuing with the rest. HyperBand (HB) [14] was the first algorithm to implement this concept. Its successor, the Asynchronous Successive Halving Algorithm (ASHA) [15], is nowadays more frequently used. BO belongs to category (2) and approximates $f$ with a probabilistic model, where new hyperparameter configurations are chosen based on the performance of earlier configurations. The Bayesian Optimization and HyperBand (BOHB) [7] algorithm combines both BO and early stopping. Furthermore, there exist genetic methods mimicking the process of evolution for finding optimal hyperparameters. First, an initial population of different ML models with randomly sampled hyperparameters is trained for a few epochs (a generation). Then, the performance is measured, and the models are ranked according to their results. Subsequently, different genetic operations, e.g., mutations, are applied. In the case of mutation, the worst performing trials copy the state and hyperparameters of the best performing models and apply small perturbations to these parameters. One of the most commonly used evolutionary HPO algorithm is Population Based Training (PBT) [10]. The iterative nature of the optimization process allows to find a series of hyperparameters (e.g., schedules).

### 2.2 Automatic Mixed Precision

Typically, computations and storage of deep NN parameters, e.g., in the forward and backward pass, use single (32 bit floating point or `FP32`) precision. However, with increasing model and dataset sizes, it is found that not all parameters require such high precision [16], leading to a potential reduction in computation time and disk space. One option is to apply half (16 bit floating point or `FP16`) precision, supported by GPUs, such as the NVIDIA A100 or AMD MI250. However, since values smaller than $2^{-24}$ cannot be represented in `FP16` and also just a few non-representable values (Not a Numbers – `NaN`s and Infinities – `Inf`s) can break a training, a master copy of the weights in `FP32` is kept in memory. To detect non-representable values and to avoid the propagation and deterioration of the model accuracy, the AMP training workflow with gradient scaling exists (integrated naively into PyTorch), see Algo. 1.

Note that occasionally skipping the optimizer step does not impair the convergence rate [16]. Due to regularization effects from the lower precision, some models can even reach higher accuracy [4]. Also, according to NVIDIA, leveraging lower precision computations results in large speed-ups. They compare the peak performance of the A100 GPU with Tensor Cores in half precision and achieve 312 TFLOPS while it is at 156 TFLOPS in full precision[2], i.e., a factor of two should be expected.

**Algorithm 1** PyTorch AMP Workflow Pseudocode

(1) Compute network forward pass in `FP16`
(2) Compute loss in `FP32`; scale by a large factor to ensure representability in `FP16`
(3) Compute backward pass in `FP16` and check for `NaN`/`Inf`
(4) **If** result contains no `NaN`/`Inf` **then**
  (a) Unscale gradients and update optimizer
**Else**
  (a) Skip optimizer update

In the context of HPO, mixed precision computations have so far been mainly leveraged solely to find suitable network architectures for low energy inference on certain hardware [3]. They have not been used to accelerate the HPO process itself, which is what the present work investigates.

## 3 EXPERIMENTAL SETUP

### 3.1 High-Performance Computing System

The experiments are performed on the GPU partition of JURECA-DC featuring two AMD EPYC 7742 Central Processing Units (CPUs) with 128 cores @2.25 GHz and four NVIDIA A100 GPUs with each 40 GB HBM2e memory. For the experiments, `CUDA/11.7`, `PyTorch/2.0.1`, and `Ray Tune/2.6.2` is used. With the distributed computing library Ray Tune[3], hyperparameter trials can be distributed across different nodes and the NN training across multiple accelerators using data-parallel methods. For the vision and CFD datasets, four GPUs and two GPUs are used for each trial. In total, 64 GPUs are allocated for the complete HPO, resulting in 16/32 concurrent HPO evaluations. The number of trials for ASHA and BOHB is fixed to 128, for PBT and Random Search (RAND) to 16 and 32.

### 3.2 Datasets and Models

The two vision datasets for benchmarking are cifar-10 and ImageNet-1k. For both, a 80/20 split of the training set is used to generate a new validation set for selecting the best performing HPO trial (i.e., final model selection). The original validation sets are used as test sets to evaluate the final performance of the best model to rule out overfitting. As the cifar-10 dataset is too small in terms of image dimension to measure substantial speed-ups from `FP16` training, the images are upscaled to 225x255 pixels.

For cifar-10, the hyperparameter search space consists of a fixed CNN topology, where the number of convolutional filters are varied for each model stage, adapted from NATS-Bench [6]. Apart from the architectural parameters, the learning rate, weight decay, and the number of warm-up epochs are optimized, resulting in an eight-dimensional search space. In the PBT case, the learning rate and

[2] https://www.nvidia.com/en-us/data-center/a100/
[3] https://www.ray.io/

weight decay are adapted after each epoch, yielding a schedule of multiple learning rates and weight decay values.

ImageNet-1k uses a custom ResNet architecture [8], where the number of channels vary throughout the model. As a performance metric, the classification accuracy is used.

The CFD dataset holds turbulent boundary layer flow data from a simulation [1]. The ML model used is a Convolutional Autoencoder for flow reconstruction. It consists of an encoder, a latent space (i.e., a lower dimensional representation of the input), and a decoder. The encoder and decoder both feature four convolutional layers, where the first two layers perform down- and up-sampling to achieve the compression in the latent space, while the other two perform regular convolution. The model is taken from the AI4HPC repository [9], which offers a selection of ML application codes optimized for HPC. The model remains fixed and only optimizer-related parameters (learning rate, weight decay, momentum, Nesterov momentum, and warm-up epochs) define the search space. The difference between the input and the (reconstructed) output flow field is quantified using the mean squared error metric. Training, validation and test set are generated by splitting the original dataset time-wise.

## 4 RESULTS

The results of combining RAND, ASHA, BOHB, and PBT with AMP are reported in Tab. 1, averaged over three different splits of the training and validation set. Additionally, the speed-ups achieved by utilizing AMP over the full precision training are depicted in Fig. 1. It should be noted that in the CFD use case the model size remains constant, i.e., it is expected that each training epoch takes roughly the same amount of time. In the cifar-10 and ImageNet-1k cases the model size changes, as architectural parameters are part of the search space, i.e., the runtime per epoch might change from trial to trial. From Fig. 1 (blue bars) it is obvious that for the CFD model, using AMP with RAND and ASHA, yields with a factor of $\approx 1.3$ in relative speed-up over full precision training. This result is expected as the model size remains constant and ASHA is essentially random search with early stopping. Benchmarking a single CFD model training run (w/o HPO) with full and mixed precision yields a factor of $^{2561s}/_{1766s} \approx 1.45$, which acts as an approximate upper speed-up limit. For the vision datasets, the attained speed-ups for ImageNet-1k are even higher with up to 1.56 for ASHA and RAND. However, they are much lower for cifar-10, which is at $1.07 - 1.13$. Larger and smaller models benefit differently from mixed precision training. In general, the ImageNet-1k models are one order of magnitude larger than the cifar-10 models. The benefits of AMP acceleration hence become notably larger and smaller, depending on the case.

A large speed-up on all datasets is observed for BOHB. Analyzing the HPO run output showed that BOHB performs early stopping much less aggressive than ASHA. Even though the same reduction factor for both algorithms is used (only the top 25% trials are allowed to continue), BOHB trains each configuration for at least three epochs while ASHA terminates many already after the first epoch. Starting a new trial is associated with an overhead that becomes (relatively) smaller when the trial is trained for longer time. For this reason a large speed-up is observed in the BOHB case of AMP compared to the full precision training. As can be seen in Tab. 1,

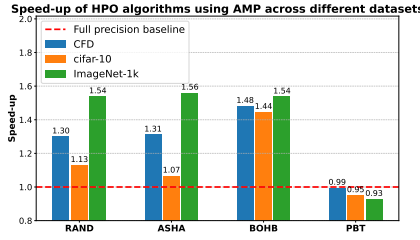



**Figure 1: Experimental results of running RAND, ASHA, BOHB, and PBT with different datasets and search spaces.**

this leads to much longer total runtimes, but also to sometimes better test set results.

No speed-up is achieved for the PBT algorithm – AMP even increases its runtime. The main reason for this is the way the PBT optimization works. The trials running concurrently are compared not only in terms of their performance on a metric, but also bad-performing trials copy the state of the good performing ones. Therefore, the whole state of the model (including the weights) and the optimizer is saved in a checkpoint and transferred to another accelerator, where perturbations to the original hyperparameters are performed and the training is continued. While the training benefits from the AMP acceleration, the weights are still stored in full precision, as mentioned in Sec. 2. That is, too frequent checkpointing and weight transfers create an overhead that cancels out the performance gains. One solution is to increase the checkpointing interval. As shown in Fig. 2, saving the model weights in a checkpoint only every five instead of every epoch can reduce the overall runtime of the HPO process (compare red/green to orange/blue lines). This leads to a noticeable speed-up of the AMP training over the full precision training with a relative factor of $\approx 1.2$.

The AMP heuristic of skipping the optimizer update if non-representable numbers are detected is already powerful. More targeted strategies, e.g., including some of the AMP parameters in the HPO search space did not improve performance. In comparison to one another, the ASHA algorithms seems to be the most favorable choice, achieving low runtimes with high test set performance.

Overall, the HPO runs that use AMP achieve slightly better performance in terms of accuracy and loss on the test set across almost all algorithms and datasets. This indicates that random noise introduced by the lower precision computations actually strengthens the generalization performance of models.

## 5 CONCLUSION AND FUTURE WORK

In this work, the speed-ups achieved by using different HPO algorithms (software) in combination with mixed precision capabilities of accelerators featuring Tensor Cores (hardware) have been investigated. It was shown that, depending on the checkpoint frequency or the model size, the computation can be accelerated by a factor of up to 1.56 on NVIDIA A100 GPUs. State-of-the-art accelerators, such as NVIDIA H100 GPUs, can run computations in even lower

| Dataset | Metric | ASHA | BOHB | PBT | RAND |
|---|---|---|---|---|---|
| **CFD** | Runtime (AMP/full) | 3046 s / 3992 s | 8994 s / 13319 s | 7428 s / 7383 s | 5253 s / 6825 s |
| | Test mse (AMP/full) | $3.20 \times 10^{-3}$ / $3.31 \times 10^{-3}$ | $2.63 \times 10^{-3}$ / $2.77 \times 10^{-3}$ | $3.40 \times 10^{-3}$ / $3.40 \times 10^{-3}$ | $4.50 \times 10^{-3}$ / $4.50 \times 10^{-3}$ |
| **cifar-10** | Runtime (AMP/full) | 3293 s / 3511 s | 9591 s / 13858 s | 2720 s / 2583 s | 3006 s / 3394 s |
| | Test acc. (AMP/full) | 0.7798 / 0.7714 | 0.7768 / 0.7830 | 0.6721 / 0.6093 | 0.7677 / 0.7560 |
| **ImageNet-1k** | Runtime (AMP/full) | 10844 s / 16915 s | 26354 s / 40495 s | 5642 s / 5233 s | 6664 s / 10249 s |
| | Test acc. (AMP/full) | 0.7006/0.6975 | 0.7055 / 0.7042 | 0.6021 / 0.5868 | 0.6758 / 0.6762 |

**Table 1: Comparison of running different HPO algorithms with full precision and AMP training, averaged over three random seeds. Better results are underlined.**

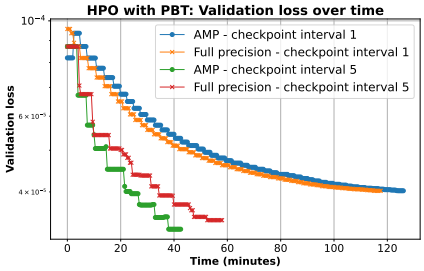



**Figure 2: Comparison of running PBT using different checkpointing intervals on CFD dataset.**

precision (primarily for Transformer models) with a potential to achieve even large speed-ups in the future. Additionally, other HPO frameworks that run on Message Passing Interface (MPI), e.g., DeepHyper [5] or Populate [18], could be explored in addition to Ray Tune.

## ACKNOWLEDGMENTS

The research leading to these results has been conducted in the CoE RAISE project, which receives funding from the European Union's Horizon 2020 – Research and Innovation Framework Programme H2020-INFRAEDI-2019-1 under grant agreement no. 951733. The authors gratefully acknowledge the computing time granted by the JARA Vergabegremium and provided on the JARA Partition part of the supercomputer JURECA at Forschungszentrum Jülich.

## REFERENCES

[1] Marian Albers, Pascal S. Meysonnat, Daniel Fernex, Richard Semaan, Bernd R. Noack, Wolfgang Schröder, and Andreas Lintermann. 2023. Actuated Turbulent Boundary Layer Flows Dataset. https://doi.org/10.34730/5dbc8e35f21241d0889906136cf28d26

[2] Prasanna Balaprakash, Romain Egele, Misha Salim, Stefan Wild, Venkatram Vishwanath, Fangfang Xia, Tom Brettin, and Rick Stevens. 2019. Scalable Reinforcement-Learning-Based Neural Architecture Search for Cancer Deep Learning Research *(SC '19)*. ACM, New York, NY, USA. https://doi.org/10.1145/3295500.3356202

[3] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. 2021. A Comprehensive Survey on Hardware-Aware Neural Architecture Search. arXiv:2101.09336 [cs.LG]

[4] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. 2020. Learning Sparse Low-Precision Neural Networks With Learnable Regularization. *IEEE Access* (2020). https://doi.org/10.1109/access.2020.2996936

[5] DeepHyper Development Team. 2018. "DeepHyper: A Python Package for Scalable Neural Architecture and Hyperparameter Search". https://github.com/deephyper/deephyper

[6] Xuanyi Dong, Lu Liu, Katarzyna Musial, and Bogdan Gabrys. 2021. NATS-Bench: Benchmarking NAS Algorithms for Architecture Topology and Size. *TPAMI* (2021), 1–1. https://doi.org/10.1109/tpami.2021.3054824

[7] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *ICML*. 1436–1445.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.

[9] Eray Inanc, Rakesh Sarma, Marcel Aach, and Andreas Lintermann. 2023. AI4HPC. https://doi.org/10.5281/ZENODO.7705421

[10] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Population Based Training of Neural Networks. arXiv:1711.09846 [cs.LG] arXiv: 1711.09846.

[11] Kevin Jamieson and Ameet Talwalkar. 2016. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *AISTATS*. 240–248.

[12] Jülich Supercomputing Centre. 2021. JURECA: Data Centric and Booster Modules implementing the Modular Supercomputing Architecture at Jülich Supercomputing Centre. *JLSRF* 7 (2021), A182. http://dx.doi.org/10.17815/jlsrf-7-182

[13] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. (2009).

[14] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *JMLR* 18, 1 (2017), 6765–6816.

[15] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A System for Massively Parallel Hyperparameter Tuning. In *MLSys*, Vol. 2. 230–246.

[16] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *ICLR*. https://openreview.net/forum?id=r1gs9JgRZ

[17] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *IJCV* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[18] Oskar Taubert, Marie Weiel, Daniel Coquelin, Anis Farshian, Charlotte Debus, Alexander Schug, Achim Streit, and Markus Götz. 2023. Massively Parallel Genetic Optimization Through Asynchronous Propagation of Populations. In *LNCS*. 106–124. https://doi.org/10.1007/978-3-031-32041-5_6

[19] Eric Wulff, Maria Girone, and Joosep Pata. 2023. Hyperparameter optimization of data-driven AI models on HPC systems. *JPCS* 2438, 1 (2023), 012092. https://doi.org/10.1088/1742-6596/2438/1/012092

# Paper V

**Distributed Hybrid Quantum-Classical Performance Prediction for Hyperparameter Optimization**

E. Wulff*, J. P. Garcia Amboage*, M. Aach* (equal contribution), T. E. Gislason, TH. K. Ingolfsson, TO. K. Ingolfsson, E. Pasetto, A. Delilbasic, M. Riedel, R. Sarma, M. Girone, and A. Lintermann

Quantum Machine Intelligence (2024)

This article is an open-access article distributed under the terms and conditions of the Creative Commons Attribution License (`http://creativecommons.org/licenses/by/4.0/`).

Marcel Aach contributed to the conceptualization, ran the experiments on the CV and OpenML datasets, and contributed to distributed hybrid workflow code on the HPC side and the communication protocol between quantum and HPC system.

# Distributed Hybrid Quantum-Classical Performance Prediction for Hyperparameter Optimization

Eric Wulff[1*†], Juan Pablo Garcia Amboage[1†], Marcel Aach[2,3†], Thorsteinn Eli Gislason[3], Thorsteinn Kristinn Ingolfsson[3], Tomas Kristinn Ingolfsson[3], Edoardo Pasetto[2,4], Amer Delilbasic[2,3], Morris Riedel[2,3], Rakesh Sarma[2], Maria Girone[1], Andreas Lintermann[2]

[1]CERN, Espl. de Particules 1, Meyrin, 1211, Geneva, Switzerland.
[2]Jülich Supercomputing Centre, Forschungszentrum Jülich, Wilhelm-Johnen-Straße, Jülich, 52428, Germany.
[3]University of Iceland, School of Engineering and Natural Sciences, Reykjavík, 107, Iceland.
[4] RWTH Aachen University, Aachen, 52056, Germany.

*Corresponding author(s). E-mail(s): eric.wulff@cern.ch;
†These authors contributed equally to this work.

**Abstract**

Hyperparameter Optimization (HPO) of neural networks is a computationally expensive procedure, which requires a large number of different model configurations to be trained. To reduce such costs, this work presents a distributed, hybrid workflow, that runs the training of the neural networks on multiple Graphics Processing Units (GPUs) on a classical supercomputer, while predicting the configurations' performance with Quantum Trained Support Vector Regression (QT-SVR) on a Quantum Annealer (QA). The workflow is shown to run on up to 50 GPUs and a QA at the same time, completely automating the communication between the classical and the quantum systems. The approach is evaluated extensively on several benchmarking datasets from the Computer Vision (CV), High-Energy Physics (HEP), and Natural Language Processing (NLP) domains. Empirical results show that resource costs for performing HPO can be reduced by up to 9% when using the hybrid workflow with performance prediction, compared to using a plain HPO algorithm without performance prediction. Additionally, the workflow obtains similar, and in some cases even better accuracy of the final hyperparameter configuration, when combining multiple heuristically obtained predictions from the QA, compared to using just a single classically obtained prediction. The results highlight the potential of hybrid quantum-classical machine learning algorithms. The workflow code is made available open-source to foster adoption in the community.

**Keywords:** Hyperparameter Optimization, Quantum Annealing, Hyperband, Distributed Computing

# 1 Introduction

The performance of neural networks with respect to their accuracy is highly sensitive to the choice of Hyperparameters (HPs). To optimize HPs efficiently, current popular Hyperparameter Optimization (HPO) algorithms, such as Hyperband (Li et al, 2017), the Asynchronous Successive Halving Algorithm (ASHA) (Li et al, 2018), and Bayesian Optimization Hyperband (BOHB) (Falkner et al, 2018), rely on early termination. In this method, under-performing trials are automatically terminated to free up compute resources for more promising trials. Choosing, e.g., the validation accuracy or the validation loss as a metric to relatively rank the trials, may lead to a suboptimal selection of trials to terminate due to the non-linearity of the training process. That is, the ranking of trials is unpredictably dynamic over the number of epochs. In practice, this problem is often mitigated by training such a large number of total model configurations that the impact of wrongfully early stopping a few promising configurations is minimized.

A potential extension of the early termination approach is to use a non-linear stopping criterion, e.g., using a model performance predictor that predicts future model performance improvements from a partially trained model. Such predictions can be used to either rank configurations in a more informed manner or to make the evaluation process more aggressive, i.e., by replacing actual evaluations of some configurations that have been trained up to a certain epoch with predictions of their performance. This was first suggested by Baker et al (2017), where Support Vector Regression (SVR) was used as a performance predictor. In conclusion, the training of the most promising configurations can be prioritized based on the predicted performance. This way, it is avoided to fully train configurations predicted to perform poorly. Consequently, this approach holds great potential for reducing the time and computational resources required for HPO.

In this work, a novel HPO algorithm called Swift-Hyperband (Amboage et al, 2023) (an extension of the original Hyperband method) is incorporated into a hybrid High Performance Computing (HPC) environment where it distributes the training of the target models onto multiple Graphics Processing Units (GPUs) and trains the performance predictors on a Quantum Annealer (QA) in autonomous fashion. Results show that Swift-Hyperband accelerates the HPO process of a High-Energy Physics (HEP)-based algorithm, Machine-Learned Particle Flow (MLPF) (Pata et al, 2021a), and other machine learning models that run on HPC systems. The goal of this paper is to show empirically how such a way of integrating a quantum system in a Machine Learning (ML) workflow can lead to resource savings in comparison to the plain Hyperband algorithm without any performance prediction and how leveraging a QA for the performance prediction can match and somtimes outperform classical performance prediction. In specific, the workflow can be applied to ML models of any size, as only a small part of the computation needs to be done on the QA, and the main training of the models is performed on classical GPUs. This makes it different from pure quantum ML workflows, which can currently only handle small problems. It should be noted, that currently there is not a clear, theoretical advantage of integrating QAs.

The presented hybrid workflow serves as a proof of concept for the integration of HPC and quantum computing technologies in a large-scale distributed fashion. By demonstrating the feasibility of such an integration, the way for future endeavors in harnessing the combined power of these computing paradigms, such as applying classic HPO techniques to quantum models among other future use cases, is paved.

The paper is structured as follows: The technical background and related work are explained in Section 2. Section 3 presents the experimental setup as well as the distributed Swift-Hyperband algorithm. The empirical results are detailed in Section 4, while Section 5 provides a conclusion and directions for future work.

## 2 Related Work and Theoretical Background

In this section, the relevant literature regarding HPO in general, the process of performance prediction to speed it up, and the foundations of classical and quantum SVR methods are summarized.

### 2.1 Quantum Annealing and QUBO Problems

Quantum annealing (Apolloni et al, 1989; Kadowaki and Nishimori, 1998) is a heuristic for optimization based on quantum computation that is often used to solve Quadratic Unconstrained Binary Optimization (QUBO) problems, i.e., discrete unconstrained optimization problems in which the problem variables can take values over a binary set (Date et al, 2021). The general cost function $E(v_1, \ldots, v_M)$ of a QUBO problem with M binary variables $v_i,\ i = 1, \ldots M$ is given by:

$$E(v_1, \ldots, v_M) := \sum_{i \leq j} Q_{ij} v_i v_j, \tag{1}$$

where $Q$ is the QUBO weight matrix that stores the coefficients of the problem. In quantum annealing, the quantum system is set to the ground state of an initial Hamiltonian $H_i$, whose ground state is known and easy to prepare. The system is then slowly evolved for a total annealing time $T_a$ by adding the contribution of a target Hamiltonian $H_p$, whose ground state encodes the solution of the optimization problem to be solved, and by reducing the contribution of the initial Hamiltonian $H_i$. The resulting Hamiltonian is then given by:

$$H(t) = A(t)H_i + B(t)H_p \tag{2}$$

where A(t) is a monotonically decreasing function such that $A(t = 1) = 1$ and $A(t = T_a) = 0$, and $B(t)$ is a monotonically increasing function such that $B(t = 0) = 0$ and $B(t = T_a) = 1$ (McGeoch, 2014). QAs from the company D-Wave implement a specific default annealing schedule, for which the corresponding values of the functions $A(t)$ and $B(t)$ can be found in the documentation [1]. On D-Wave systems, a default annealing time of $20\mu s$ is set. Both the annealing schedule and annealing time are set to their default values in this study.

### 2.2 Hyperparameter Optimization Algorithms

Early termination algorithms are nowadays one of the main tools for HPO of deep neural networks. They have shown to save considerable amounts of resources in the HPO process without losing the ability to find good configurations for the target model (Li et al, 2018; Falkner et al, 2018; Yu and Zhu, 2020). The Successive Halving algorithm (Jamieson and Talwalkar, 2016) is the foundation of most of the relevant early termination algorithms. Successive Halving trains a set of randomly generated configurations for a certain number of epochs, discards the worst-performing half and repeats this process until only one configuration remains. The Hyperband algorithm (Li et al, 2017) runs multiple instances of Successive Halving sequentially with different numbers of initial configurations and different locations of the decision points. BOHB (Falkner et al, 2018) is an algorithm that extends Hyperband by following a Bayesian optimization approach for selecting the configurations instead of generating them randomly. Finally, ASHA (Li et al, 2018) stands for Asynchronous Successive Halving and extends Successive Halving to efficiently benefit from large-scale distributed resources.

[1] D-Wave annealing schedule:https://docs.dwavesys.com/docs/latest/doc_physical_properties.html

## 2.3 Performance Prediction

Performance prediction aims at predicting the future performance of a model under a given set of hyperparameters based on the early results of the training, e.g., using a partial learning curve, see Fig. 1. The problem can formally be defined as predicting the validation loss or validation accuracy $l(\lambda)_R$ at training epoch $R \in \mathbb{N}$ that will be obtained by a machine learning model with the hyperparameter configuration $\lambda \in \Lambda \subset \mathbb{R}^h$, where $h$ is the number of hyperparameters of the network, see (Baker et al, 2017) and (Liu et al, 2022). The performance of the target model at previous training epochs can be used for this prediction. If the auxiliary model chosen to be used as a performance predictor is fast to train relative to the target model, performance prediction can be used to accelerate the HPO process. In this regard, it has been shown that computationally cheap regression methods, such as SVR, are good choices to be used as performance predictors (Baker et al, 2017).

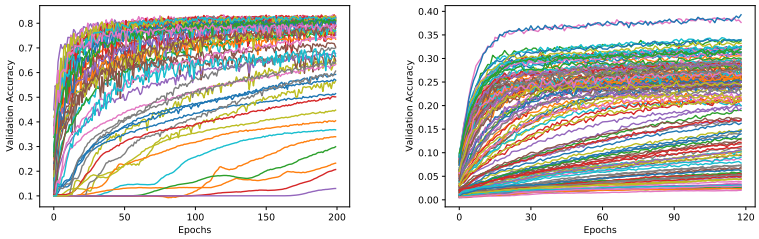



**Fig. 1**: Example learning curves of different Convolutional Neural Networks (CNNs) on the cifar-10 and TinyImageNet dataset.

A simple strategy to integrate performance prediction in an HPO process can be described as follows. First, a series of random HP configurations from a given search space is generated, a few of them are trained for $R$ epochs, and the generated learning curves are used to train a performance predictor model such as an SVR. Next, the remaining configurations are initially trained for $\tau < R$ epochs (e.g. $\tau = \frac{R}{2}$), whereafter their future performances are predicted. Finally, only those configurations that are considered promising, according to the predicted performance, are allowed to continue training until epoch $R$. Note that the initial full and partial trainings can be done in parallel.

## 2.4 Support Vector Regression Methods

Support Vector Machines (Boser et al, 1992; Drucker et al, 1996) are popular supervised learning algorithms that can be applied to classification and regression tasks. One of the reasons for their popularity is found in the fact that the determination of the model parameters amounts to a convex optimization problem, therefore any local solution corresponds to a global one (Bishop, 2006). However, it is important to point out that the global minimum of the training cost function may not be optimal in terms of generalization to the test set (Willsch et al, 2020). Moreover, they can approximate non-linear functions by applying the so-called *kernel trick* (Burges, 1998) thus increasing the algorithm's expressive potential.

The mathematical formulation of the SVR is subsequently briefly outlined. Given a training dataset $\{(\mathbf{x}_n, y_n),\ n = 1, \ldots, N\}$, where $\mathbf{x}_n \in \mathbb{R}^z$ is the input vector, $N$ is the number of training samples, and $y_n$ is its corresponding target value, the objective is to approximate

a regression function

$$g(\mathbf{x}) = \sum_{n=1}^{N} (\alpha_n - \hat{\alpha}_n)\kappa(\mathbf{x}_n, \mathbf{x}) + b, \tag{3}$$

which maps from $\mathbb{R}^z$ to $\mathbb{R}$. The parameters $\alpha_n, \hat{\alpha}_n$ are determined in the optimization process. The term $\kappa(\mathbf{x}_n, \mathbf{x})$ denotes the kernel function, which is in this study the Radial Basis Function (RBF) kernel with formula $e^{(-\gamma||\mathbf{x}_n - \mathbf{x}_m||^2)}$. More generally, a RBF kernel is a kernel whose value depends only on the distance of the input vectors, i.e., $\kappa(\mathbf{x}_m, \mathbf{x}_n) = \kappa(||\mathbf{x}_m - \mathbf{x}_n||)$. RBF kernels are one of the most popular choices for SVR kernels along with polynomial and sigmoid kernels (Bishop, 2006). It can be shown that the training phase amounts to solving the following constrained optimization problem (also referred to as the cost function):

$$\begin{aligned} L(\boldsymbol{\alpha}, \hat{\boldsymbol{\alpha}}) = & \frac{1}{2} \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} (\alpha_n - \hat{\alpha}_n)(\alpha_m - \hat{\alpha}_m)\kappa(\mathbf{x}_n, \mathbf{x}_m) + \\ & - \epsilon \sum_{n=0}^{N-1} (\alpha_n + \hat{\alpha}_n) + \sum_{n=0}^{N-1} (\alpha_n - \hat{\alpha}_n) y_n, \end{aligned} \tag{4}$$

satisfying the constraints:

$$\sum_{n=0}^{N-1} (\alpha_n - \hat{\alpha}_n) = 0, \tag{5a}$$

$$0 \leq \alpha_n \leq C, \tag{5b}$$

$$0 \leq \hat{\alpha}_n \leq C, \tag{5c}$$

where the terms $C$ and $\epsilon$ are hyperparameters that control the overfitting and the error sensitivity. The vectors $\boldsymbol{\alpha}$ and $\hat{\boldsymbol{\alpha}}$ are defined as $\boldsymbol{\alpha} = \{\alpha_1, \ldots, \alpha_N\}$ and $\hat{\boldsymbol{\alpha}} = \{\hat{\alpha}_1, \ldots, \hat{\alpha}_N\}$, respectively. The value of $b$ can be obtained from any point for which $0 < \alpha_n < C$ by

$$b = y_n - \epsilon - \sum_{m=1}^{N} (\alpha_m - \hat{\alpha}_m)\kappa(\mathbf{x}_n, \mathbf{x}_m). \tag{6}$$

Due to the constraints it must satisfy $\epsilon + g(x_n) - y_n = 0$. It is, however, preferable to average over different estimates of $b$ to yield a stable solution, i.e.,

$$b = \frac{1}{|\mathcal{S}|} \sum_{n \in \mathcal{S}} \left( y_n - \epsilon - \sum_{m=1}^{N} (\alpha_m - \hat{\alpha}_m)\kappa(\mathbf{x}_n, \mathbf{x}_m) \right). \tag{7}$$

In this equation, $\mathcal{S}$ corresponds to the set of *support vectors*, i.e., those vectors that contribute to the prediction of the target value, c.f. Eq. (3) (Bishop, 2006).

## 2.5 Quantum Support Vector Regression

To optimize the training phase of a Quantum Trained Support Vector Regression (QT-SVR) (Pasetto et al, 2022), it is necessary to reformulate the optimization problem as either an Ising or QUBO problem. In the currrent study, the problem is restructured as a QUBO problem by carrying out a 3-step problem conversion procedure, which consist of (i) encoding the

problem variables, (ii) adding penalty terms to encode the constraints, and (iii) defining the QUBO matrix. These steps are subsequently explained in more detail.

### 2.5.1 Problem Variable Encoding

The first step towards the construction of the QUBO problem consists of turning the problem variables into binary ones. Specifically, each of the original problem variables is encoded using $K$ qubits according to

$$\alpha_n = \sum_{k=0}^{K-1} B^{k-P} a_{Kn+k}, \tag{8}$$

$$\hat{\alpha}_n = \sum_{k=0}^{K-1} B^{k-P} a_{K(N+n)+k}, \tag{9}$$

where $B$ is an encoding basis and $a$ are the values of the qubits. The parameter $P$ is used to allow the usage of negative exponents in the encoding procedure.

This results in $2KN$ QUBO variables, where the first $KN$ variables are used to encode $\boldsymbol{\alpha}$, whereas the last $KN$ variables are used to represent $\hat{\boldsymbol{\alpha}}$. The $\boldsymbol{\alpha}$ and $\hat{\boldsymbol{\alpha}}$ are then substituted into Eq. (4).

### 2.5.2 Penalty Terms Addition

The QUBO problem must be unconstrained. It is, therefore, necessary to add penalty terms, whose influence is regulated by hyperparameters, to implicitly enforce the constraints. To enforce Eq. (5a), a square penalty regulated by the hyperparameter $\xi$ is added to the cost function in Eq. (4):

$$\xi \left( \sum_{n=1}^{N} (\alpha_n - \hat{\alpha})_n \right)^2. \tag{10}$$

The constraints defined by Eqs. (5b) and (5c), which are also referred to as box constraints, are implicitly satisfied by the encoding equations. As the qubit values collapse to either 0 or 1, the maximum value that each $\alpha_n$ and $\hat{\alpha}_n$ can take is

$$\sum_{i=0}^{K} B^{K-1-P}, \tag{11}$$

This is obtained if all qubit values $a_{Kn+k}$ or $a_{K(N+n)+k}$ collapse to 1 for $k = 0, \ldots, K-1$.

### 2.5.3 QUBO Matrix Definition

After the addition of the penalty terms (see Eq. (10)) to the cost function (see Eq. (4)) and the subsequent encoding of the problem variables (see Eq. (8)), the final QUBO cost function takes the form:

$$\sum_{n,m=0}^{N-1} \sum_{i,j=0}^{K-1} \sum_{s,t=0}^{1} a_{K(sN+n)+i} \tilde{Q}_{K(sN+n)+i,K(tN+m)+j} a_{K(tN+m)+j}, \tag{12}$$

where $\tilde{Q}$ is an $2KN \times 2KN$ matrix that encodes the problem and whose elements are given by

$$\tilde{Q}_{K(sN+n)+i,K(tN+m)+j} = (-1)^{(1-\delta_{st})} B^{i+j-2P} \left( \frac{1}{2}\kappa(\mathbf{x}_n, \mathbf{x}_m) + \xi \right) \tag{13}$$

$$+ \delta_{nm}\delta_{ij} B^{i-P} \delta_{st} \left( \epsilon + (-1)^{(1-s)(1-t)} y_n \right) \tag{14}$$

To obtain a problem formulation similar to Eq. (1), it is necessary to construct the upper-triangular $2KN \times 2KN$ QUBO matrix $Q$ from $\tilde{Q}$ by using

$$Q_{i,j} = \begin{cases} \tilde{Q}_{i,j} + \tilde{Q}_{j,i}, & \text{if } i < j; \\ \tilde{Q}_{i,j}, & \text{if } i = j; \\ 0, & \text{otherwise.} \end{cases} \tag{15}$$

The minimization problem can then be written as

$$\min_{\underline{\boldsymbol{a}} \in \{0,1\}^{2KN}} \underline{\boldsymbol{a}}^T \, Q \, \underline{\boldsymbol{a}}, \tag{16}$$

where $\underline{\boldsymbol{a}}$ is the $2KN$ vector obtained by the concatenation of $\boldsymbol{a}$ and $\hat{\boldsymbol{a}}$.

The final step to run a problem instance on the QA is creating a *minor embedding* (Choi, 2011). This arises from the fact that the graph structure of the Quantum Processing Unit (QPU) is not fully connected. It is, therefore, necessary to represent a logical qubit with a group of connected qubits that are constrained to have the same values. For a given QUBO problem it is possible to construct the corresponding problem graph $G(V, \mathcal{E})$ by considering as the set of nodes $V = \{a_i, \ldots, a_n\}$, where each node corresponds to a problem variable and the set of edges $\mathcal{E} = \{(a_i, a_j),\ \forall\ (i,j) \text{ such that } Q_{i,j} \neq 0\}$. The D-Wave Advantage system uses a *Pegasus* (Boothby et al, 2020) topology graph. For the experiments conducted in this study, the problem graph always had the same structure for each test run since it is a fully connected graph with a number of nodes equal to $2KN$. The values of $K$ and $N$ were always the same for each experimental run. Therefore, the embedding was calculated only once using the functions provided by the official D-Wave Ocean tools [2], which provide an interface to the QA machine.

### 2.5.4 Advantages of Q-SVMs over Classical Counterparts

So far, established work combining Support Vector Machines (SVMs) and general Quantum Computing (QC) has focused on performing the computation of the kernel on the quantum hardware (Rebentrost et al, 2014). This approach has a theoretical advantage in runtime, as kernel computation requires only logarithmic runtime in the quantum setting but at least quadratic runtime in the classical setting. However, computing not the kernel but the training process (as performed in this study) on a QA has two advantages:

- **Time complexity:** Usually, the whole training procedure of (classical) SVMs and SVRs in specific has **cubic** time complexity (Bottou et al, 2007; Abdiansah Abdiansah, 2015), which can lead to long runtimes for large datasets. To the contrary, a QUBO solved on QAs returns a set of low-energy solutions after a predefined amount of time, **independent of**

[2] D-Wave Ocean SDK version 6.9.0: https://www.dwavesys.com/solutions-and-products/ocean/

**the input problem size** (Date et al, 2021). This can lead to potential speed-ups for large training datasets in the future.

- **Solution combinations:** While classical SVMs return a single optimal solution, QAs provide multiple low-energy solutions to a given problem. These different solutions can be combined, analogous to ensembling. This has empirically been shown to improve generalization performance in classical learning algorithms (Dietterich, 2000), and a similar effect can be observed in quantum learning algorithms (Willsch et al, 2020; Cavallaro et al, 2020).

It should be noted, that the set of QA solutions is not guaranteed to be theoretically optimal (as in the case of classical SVMs) and that limitations in terms of the number of available qubits on current QAs also the limit the problem size that can be solved in a quantum setting. For D-Wave QAs, the number of available qubits has however grown steadily from $2,000$ to more than $5,000$ during the past years [3].

# 3 Experimental Setup

## 3.1 Machines

Two different types of machines are used for the hybrid setup of the workflow. The classical calculations are executed on the Extreme Scale Booster partition of the DEEP-EST supercomputer[4]. It features a total of 75 nodes, where each node is equipped with an Intel Xeon Central Processing Unit (CPU) with 8 cores and a NVIDIA V100 GPU. The quantum calculations take place on the D-Wave Advantage system JUPSI [5], as of 04/2024 the largest QA in Europe with 5614 qubits. Both machines are located at the Jülich Supercomputing Centre.

## 3.2 Performance Prediction Algorithm

The Fast-Hyperband (Baker et al, 2017) algorithm is a modified version of Hyperband that adds an additional, performance prediction-based decision point at the end of every epoch. This allows the algorithm to estimate which configurations are less likely to be promoted to the next round and terminate them earlier than the original Hyperband. Naturally, this saves computational resources compared to the classical Hyperband algorithm.

The Swift-Hyperband algorithm is similar to Fast-Hyperband but adds only one extra decision point based on performance prediction inside each Hyperband bracket round (instead of one extra decision point every epoch), see Fig. 2. Therefore, Swift-Hyperband requires training considerably fewer performance predictors than Fast-Hyperband. This enables Swift-Hyperband to run in a hybrid quantum-classical workflow, where the performance predictors are trained using a QA. Furthermore, each round of Swift-Hyperband can be parallelized as the trial trainings can be performed in parallel in contrast to Fast-Hyperband, which is a sequential approach.

Algorithm 1 and 2 in Appendix A present the pseudocode for Swift-Hyperband and its auxiliary routine `run_then_return_val_loss`, where the performance predictors are both trained and used to make predictions. For simplicity, only the pseudocode of the sequential version of Swift-Hyperband is presented. In the distributed implementation, the parallelization is performed at round level in the same way that is described for Hyperband in (Li et al, 2018). This is, the outer *for* loop in the routine `run_then_return_val_loss` is adapted to run its multiple iterations in parallel. The only disadvantage of this type of parallelization is the same as the one discussed for the classical version of Hyperband in (Li et al, 2018). That is, the number of iterations in the parallelized *for* loop is not constant through out

[3] D-Wave QAs: https://www.dwavesys.com/solutions-and-products/systems/
[4] DEEP-EST: https://www.fz-juelich.de/en/ias/jsc/systems/prototype-systems/deep_system
[5] JUPSI: https://www.fz-juelich.de/en/ias/jsc/systems/quantum-computing/juniq-facility/juniq/d-wave-advantagetm-system-jupsi

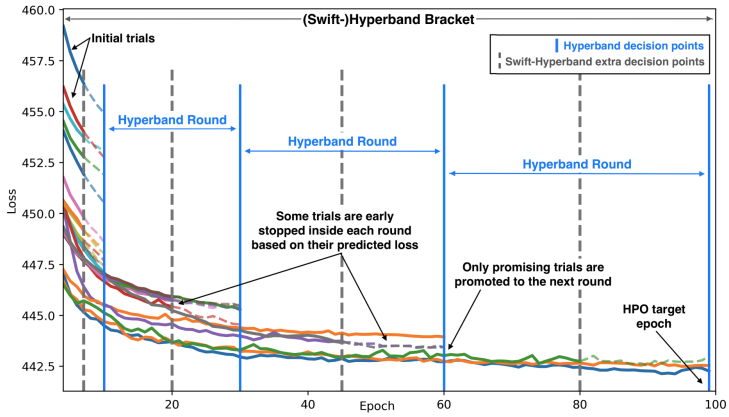



**Fig. 2**: Graphic representation of the Swift-Hyperband algorithm, taken from Amboage et al (2023).

the whole algorithm but instead decreases from the start to the end of each bracket. For this reason, using a high number of parallel computing nodes may result in some of them being idle towards the end of each bracket.

## 3.3 Hybrid Quantum-Classical Workflow

Swift-Hyperband is suited for hybrid quantum-classical workflows. A QT-SVR, trained on a QA can be used as the performance predictor, while the trainings of the target model can be performed in parallel across several nodes in an HPC center. However this does not hold for Fast-Hyperband. Note that, apart from being a sequential algorithm, Fast-Hyperband has the disadvantage that a high number of performance predictors is necessary. This makes it infeasible to run with QT-SVRs due to the time needed to communicate with the QA.

For this work, a hybrid quantum-classical implementation of Swift-Hyperband that uses the Message Passing Interface (MPI) standard to communicate across multiple nodes in an HPC cluster and relies on the D-Wave Ocean SDK to connect to a D-Wave Advantage System to train the QT-SVRs, has been developed. In this implementation, one node of the HPC cluster acts as a head node, communicating with the QA as well as coordinating up to 50 GPU-equipped worker nodes, where the different target model configurations are trained. This workflow is illustrated graphically in Fig. 3.

On current quantum hardware, the size of problems that can be computed is still limited. Considering the cubic time complexity associated with the training of classical SVR predictors and the capability of a quantum system to run calculations in a matter of milliseconds, a speed-up can, however, be expected for large datasets on future quantum hardware (Date et al, 2021).

### 3.3.1 QT-SVR Solution Combination

As mentioned in Sec. 2.5.4, the QA returns a set of low-energy solutions after a fixed amount of time, instead of just a single one. These multiple solutions can then be combined to create a more robust prediction, which has already been performed in earlier literature (Willsch et al, 2020; Pasetto et al, 2022). Given the true training target values $y^{train}$ and the predicted target

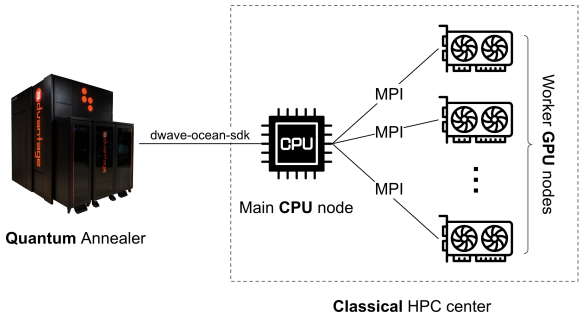



**Fig. 3**: Components along with their communication protocols used for the Swift-Hyperband distributed hybrid quantum-classical implementation.

values $\tilde{y}_i^{train}$, where $i = 1, \ldots, s$ in the range of the total number of solutions $s$, each of the $s$ candidate solutions (i.e. the different set of values for $\alpha_n$ and $\hat{\alpha}_n$) is assigned a weight $w_i,\ i = 1, \ldots, s$. To do so, for each set of $\alpha_n$ and $\hat{\alpha_n}$ the mean squared error loss $L_i$ is computed on the training dataset:

$$L_i = \frac{1}{N} \sum_{j=1}^{N} (y^{train(j)} - \tilde{y}^{train(j)})^2 \tag{17}$$

In order to give more credit to the solutions which achieved better performance (on the training data) and to reduce the contributions of the solutions that performed worse, for each $i = 1, \ldots, s$ a coefficient is defined $\hat{w}_i$ as $\hat{w}_i := \frac{1}{L_i}$. Finally, the solution combination coefficients are obtained as

$$w_i := \frac{\hat{w}_i}{\sum_{l=1}^{s} w_l} \tag{18}$$

These weights are then used to obtain the final solution $\boldsymbol{\alpha}^{final} := \{\alpha_1^{final}, \ldots, \alpha_N^{final}\}$ as a weighted average over the solutions returned by the QA:

$$\boldsymbol{\alpha}^{final} := \sum_{i=1}^{s} w_i \boldsymbol{\alpha}_i, \tag{19}$$

with $\boldsymbol{\alpha}_i := \{\alpha_i, \ldots, \alpha_s\}$. Essentially, this resembles a set of weak classifiers, that combined yield a strong classifier (Willsch et al, 2020; Cavallaro et al, 2020). In this study, the number of solutions combined is set to $s = 4$.

## 3.4 Datasets and Models

For this study, the capabilities of the hybrid workflow are evaluated on multiple datasets and neural network models. Fully-connected Neural Network (FCNN) are trained on small regression datasets from the OpenML project (Vanschoren et al, 2014), CNNs are trained on two well-known medium-sized datasets from the Computer Vision (CV) domain, a Long Short-Term Memory (LSTM) is trained on an Natural Language Processing (NLP) dataset, and the MLPF algorithm is trained on a HEP dataset. In the following, the datasets as well are explained. To enable reproducibility of the empirical results, also the HP search spaces that the HPs are sampled from a provided in detail.

### 3.4.1 OpenML Datasets

The OpenML platform[6] features a wide selection of easily accessible datasets, algorithms, and experiments. To make sure the results are reproducible, the method is tested on three datasets from the OpenML Curated Tabular Regression benchmark (Fischer et al, 2023). The focus of this benchmark is on tabular, high-quality datasets from different domains with 500 to 100,000 observations that are not trivially solvable by linear methods. For this study, the Grid Stability (id 44973), Video Transcoding (id 44974), and Naval Propulsion Plant (id 44969) datasets are selected.

However, to generate reproducible learning curves, not only the dataset and evaluation procedure is important, but also the hyperparameter search space that the different model configurations are sampled from has to be consistent. Therefore, the chosen OpenML datasets are trained on the hyperparameter search space detailed in the HPOBenchmark library (Eggensperger et al, 2021), a general benchmark for HPO. The search space consists of two architectural parameters of FCNNs, (depth and width) and three optimizer-related parameters (batch size, initial learning rate, and weight decay). The range the hyperparameters are sampled from is shown in Table 1. Each model is trained for a maximum of 50 epochs.

**Table 1**: Hyperparamters from the HPOBench search space used on the OpenML datasets. Depth and width determine the general shape of the network while batch size, initial learning rate, and weight decay (alpha) influence the Adam optimizer.)

| hyperparameter | type | range |
|---|---|---|
| alpha | float | log[1e-8, 1] |
| batch size | int | log[4,256] |
| depth | int | [1,3] |
| learning rate | float | log[1e-5, 1] |
| width | int | [16,1024] |

### 3.4.2 Computer Vision Datasets

The two datasets of choice from the CV domain are the cifar-10 (Krizhevsky, 2009) and TinyImageNet (Mnmoustafa, 2017) dataset. The cifar-10 dataset consists of 60,000 images in 10 different classes. The training is performed on 50,000 images while the remaining 10,000 images are used to compute the validation accuracy. TinyImageNet contains 100,000 images, split into 200 classes. For each class, there are 500 training images and 50 validation images. For both datasets, the image pre-processing steps involved a `RandomCrop`, `Resize`, `RandomHorizontalFlip`, and `Normalization` operation from the torchvision library[7]. Similarly to (Baker et al, 2017), small architectures with varying numbers of convolutional layers, convolutional filters, batch size, learning rate, and momentum are sampled from the hyperparameter search space, see Table 2, for the cifar-10 models, which are trained for a maximum of 100 epochs. For the TinyImageNet case, a fixed ResNet18 architecture (He et al, 2016) is trained for a maximum of 35 epochs and only optimizer-related parameters (batch size, learning rate, and momentum) are tuned. Performance prediction in both cases is performed on the validation set learning curves.

[6]OpenML: https://www.openml.org/
[7]Torchvision library: https://pytorch.org/vision/stable/index.html

**Table 2**: Hyperparameter search space used on cifar-10 and TinyImageNet. Layers and filters determine the general shape of the CNN while batch size, learning rate, and momentum influence the optimizer.

| hyperparameter | type | range |
|---|---|---|
| layers | int | [2,3,4] |
| filters | int | [16,32,48,64] |
| batch size | int | [64,128,256,512] |
| learning rate | float | log[1e-4, 1] |
| momentum | float | log[1e-4,0.9] |

### 3.4.3 HEP Model and Dataset

In response to the considerable surge in data generation anticipated in large HEP experiments in the forthcoming decades, there are ongoing efforts towards substituting conventional CPU-based algorithms with neural network-powered ones that can be efficiently and readily executed on GPUs, Field Programmable Gate Arrays (FGPAs) or other hardware accelerators. A prime illustration of such novel algorithms is the so-called MLPF (Pata et al, 2021a) algorithm, designed to perform particle-flow reconstruction (Sirunyan et al, 2017) through a data-driven methodology. Advantages of MLPF include extensibility, portability and scalability. Extensibility because the algorithm can easily be adapted to new detector geometries or conditions by retraining, portability, because the model can be executed on a wide variety of different hardware accelerators, and scalability, because runtime and memory consumption scale approximately linearly with the input collision event size.

The dataset used for HPO studies in this work is the open and publicly available DELPHES dataset (Pata et al, 2021b) first presented in (Pata et al, 2021a). HPO is performed on a 7-dimensional search space, see Table 3, and each trial is trained for a maximum of 100 epochs.

**Table 3**: Hyperparameter search space used on MLPF.

| hyperparameter | type | range |
|---|---|---|
| learning rate | float | log[1e-6, 3e-2] |
| dropout | float | [0,0.5] |
| weight decay | float | log[1e-6,1e-1] |
| num graph layers id | int | [0,4] |
| num graph layers reg | int | [0,4] |
| bin size | int | [8,16,32,64,128] |
| output dim | int | [8,16,32,64,128,256] |

### 3.4.4 NLP Model and Dataset

The datasets of choice from the NLP domain is the bAbI tasks dataset (Weston et al, 2015), in particular task 17. The bAbI tasks dataset consists of 20 elementary reasoning tasks, where task 17 is related to spatial and positional reasoning. Each particular instance of task 17 is conformed by a group of sentences related to the relative position of multiple colored blocks followed by a "yes or no" question about the location of one of the blocks. Therefore, the goal of the model is to infer the correct "yes/no" answer to the question given its precedent

sentences. The training is performed on 1,000 questions, and another 10,000 questions are used to compute the validation accuracy. A fixed LSTM architecture with varying training hyperparameters (see Table 4), where each model can be trained up to 300 epochs, is used [8].

**Table 4**: Hyperparameter search space used on the LSTM.

| hyperparameter | type | range |
|---|---|---|
| learning rate | float | log[1e-10, 1] |
| dropout | float | [0,1] |
| rho | float | [0,1] |
| weight decay | float | log[1e-5, 0.1] |

## 4 Results and Evaluation

For performance evaluation, the distributed quantum-classical implementation of Swift-Hyperband is compared to the original Hyperband algorithm and to Swift-Hyperband using classical SVRs. Several target models from different domains, i.e., from CV, NLP, and HEP, are optimized. For each target model, the best accuracy or loss achieved by each algorithm is reported along with the average resources consumed (in terms of the total number of training epochs), using an average of three runs, each one initialized with a different random seed (0,1 and 2). Detailed information on the target models and the HPO processes used to test the algorithms are shown in Table 5. The results, computed on 51 nodes of the DEEP-EST supercomputer in combination with the JUPSI QA are shown in Figs. 4 and 5.

**Table 5**: Summary of the benchmarking cases used to test the HPO algorithms.

| NN architecture | Dataset | Domain | Evaluation metric | # HPs for HPO | Target epoch for HPO | # GPU Nodes for HPO |
|---|---|---|---|---|---|---|
| CNN | CIFAR-10 (Krizhevsky, 2009) | CV | accuracy | 5 | 100 | 50 |
| CNN | Tiny ImageNet (Le and Yang, 2015) | CV | cross entropy loss | 3 | 35 | 50 |
| LSTM | bABI (Weston et al, 2015), task 17 | NLP | accuracy | 4 | 300 | 50 |
| MLPF (Pata et al, 2021a) | Delphes (Pata et al, 2021b) | HEP | Focal loss + Huber loss | 7 | 100 | simulated |
| FCNN | OpenML | Tabular | mean squared error loss | 5 | 50 | 50 |

From Figs. 4 and 5, it can be seen that Swift-Hyperband, both in the case of using SVRs and QT-SVRs, achieves results similar to classical Hyperband in terms of the target model performance while consuming less computational resources in all cases. The largest savings of $\sim 9.4\%$ are observed for the CNN training on cifar-10 (Fig. 4a), where Hyperband takes 3237 epochs, Swift-Hyperband SVR takes 3014 and Swift-Hyperband QT-SVR only 2960 epochs on average. For the Tiny ImageNet cases (Fig. 4b) Hyperband requires 872 training iterations, while Swift-Hyperband QT-SVR finishes in 834, resulting in a resource saving of $\sim 4.6\%$ with only a small difference in validation loss (0.012 vs. 0.014).

For all other cases, $\sim 2-5\%$ in savings can be seen. While plain Hyperband only bases the future performance of a trial based on the current validation loss or accuracy, Swift-Hyperband can make use of performance prediction methods and thus terminate some trials earlier, resulting in fewer total training epochs. When comparing the Swift-Hyperband SVR and QT-SVR versions, the quantum-based regression method is able to match the validation set performance of the classical method in almost all cases, and in the majority of cases even

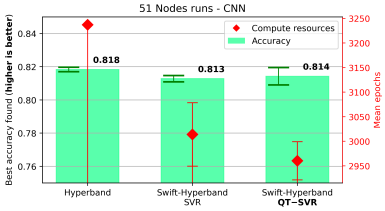
[8] LSTM Training: https://docs.ray.io/en/latest/tune/examples/includes/pbt_memnn_example.html

outperforms it. As explained in Section 2.5.4, the main difference between the quantum and classical SVR is, that the QT-SVR makes use of multiple, heuristically obtained predictions, which are weighted and combined into a single prediction. On the contrary, the classical SVR only uses a single deterministic prediction. The emperical results of these experiments prove that for these benchmarking cases the QT-SVR produces a more robust predictions of the future performance of trials when used inside a distributed version of Swift-Hyperband, which then leads to higher validation scores in the majority of cases.

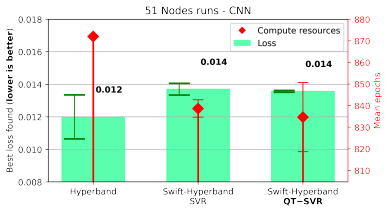
In terms of computing resources, the QT-SVR version requires fewer epochs than the SVR for the CNN cases (see Figs. 4a and 4b), but more for all other instances. This indicates that on the one hand, the QT-SVR version tends to over-estimate the performance of the target models, hence early-stopping fewer configurations in the prediction-based decision points (and use more compute resources) for the LSTM, MLPF and FCNN models. For the CNN models, on the other hand, it tends to under-estimate the performance, thus early-stopping more configurations and saving resources, but still achieves comparable target model performance. This is because the shapes of the learning curves (to which the regression method is applied) are highly dependent on the nature of the target model and the application case. It is interesting to observe, that the dimension of the hyperparameter search space (e.g. a seven-dimensional search space for Fig. 4d and only a three-dimensional one for Fig. 4b) does not influence the performance of the algorithms.

As an alternative to solving the QUBO problem purely on the QA, D-Wave offers the option to use a cloud-based hybrid solver[9]. This solver internally solves a part of the problem with state-of-the-art classical algorithms, while sending only those parts to the QPU that primarily benefit from it. In this case, also the combination of the solutions is performed internally and is not public. While this takes notably longer than pure QPU calculations (a few seconds vs. a few hundred milliseconds) it can also handle larger problems. A comparison on the Grid Stability dataset, see Fig. 5, shows the hybrid solver to outperform both SVR and QT-SVR-based Swift-Hyperband in terms of best-found model and number of epochs used. The plain Hyperband algorithm achieves the lowest loss overall, but at the cost of a much higher compute resource consumption. In total, the results still indicate that the hybrid solver is able to estimate the performance of the target model with high accuracy.

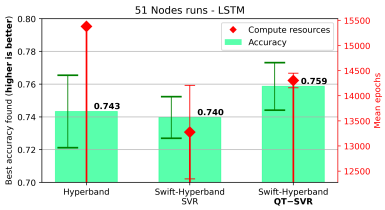
[9] D-Wave Hybrid Solver: https://docs.ocean.dwavesys.com/en/latest/overview/hybrid.html



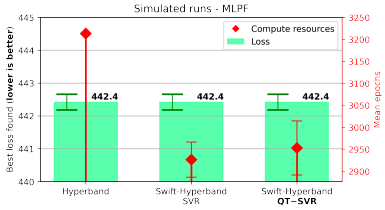

(a) CNN for cifar-10 dataset.



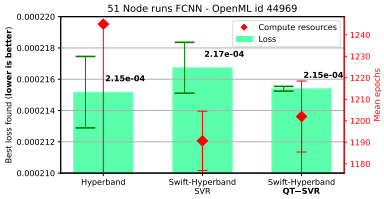

(b) CNN for Tiny ImageNet.



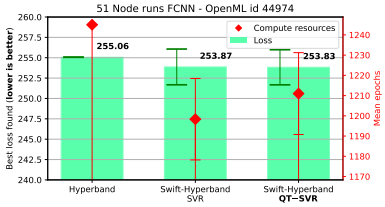

(c) LSTM for bAbI tasks dataset (task 17).




(d) MLPF for Delphes dataset.




(e) FCNN on the Naval Propulsion dataset.




(f) FCNN on the Video Transcoding dataset.

**Fig. 4**: Average resource consumption and performance of the best configuration found for each HPO algorithm applied to different target models. Results are averaged over three different random seeds and error bars are shown. Note that the number of epochs for Hyperband is deterministic and therefore no error bar is shown for the algorithm.

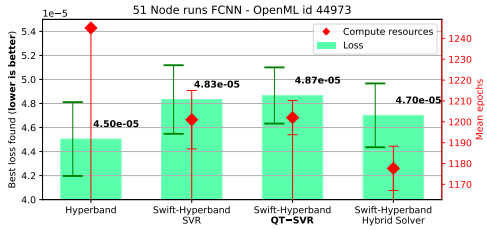



**Fig. 5**: Average resource consumption and performance of the best configuration found for each HPO algorithm on the Grid Stability dataset, including D-Waves internal hybrid solver. Results are averaged over three different random seeds and error bars are shown. Note that the number of epochs for plain Hyperband is deterministic and therefore no error bar is shown for the algorithm.

## 5 Summary, Conclusion and Outlook

This study presented a workflow for performing distributed, quantum-classical performance prediction for HPO. Compared to the established Hyperband algorithm, the proposed workflow saves resources with minimal sacrifices in terms of validation set performance of the best-found model. Especially in a distributed setting of $50$ GPUs running neural network training at the same time, resource savings of more than $9\%$ are substantial. It was also evident, that choosing a QT-SVR or a hybrid-solver method results empirically in better-performing models for a wide range of application cases compared to classical SVR, due to combining multiple heuristically obtained solutions. This stresses the potential of using quantum machine learning methods.

This work presents an important first step in the direction of automated integration of quantum devices in the supercomputing environment. This is of great importance, as current quantum machines are still too small to solve meaningful problems on their own. By combining them with a powerful supercomputer it becomes possible to tackle relevant, real-world problems from a diverse set of scientific domains. The proposed workflow is agnostic to the underlying machine learning model and can be applied to any problem. As the code of the workflow is open-source[10], the research community can benefit directly from this work.

The most promising direction of future work is the usage of larger and more advanced quantum hardware. For this study, a QA was chosen, as it is able to handle a much larger problem size than current, gate-based machines. With increases in the number of qubits of gate-based machines and advances in algorithm development, it might be possible to run other, more advanced quantum optimization algorithms that have the potential to not only save compute resources in a quantum-classical setting but also find more accurate solutions.

## 6 Statements and Declarations

E. Wulff, J.P. García Amboage, M. Aach, R. Sarma, M. Riedel, and A. Lintermann were supported by CoE RAISE. The CoE RAISE project has received funding from the European Union's Horizon 2020 – Research and Innovation Framework Programme H2020-INFRAEDI-2019-1 under grant agreement no. 951733. The authors gratefully acknowledge the computing time granted through JARA on the supercomputer JURECA (Jülich Supercomputing Centre, 2021) at Forschungszentrum Jülich. The authors gratefully acknowledge

[10] GitHub link: https://github.com/JP-Amboage/qtml-hybrid-workflow

the Jülich Supercomputing Centre for funding this project by providing computing time through the Jülich UNified Infrastructure for Quantum computing (JUNIQ) on the D-Wave Advantage™ System JUPSI.

The authors declare no competing interests.

## References

Abdiansah Abdiansah RW (2015) Time complexity analysis of support vector machines (svm) in libsvm. International Journal of Computer Applications 128(3):28–34. https://doi.org/10.5120/ijca2015906480, URL https://ijcaonline.org/archives/volume128/number3/22854-2015906480/

Amboage JG, Wulff E, Girone M, et al (2023) Optimizing AI-based HEP algorithms using HPC and Quantum Computing. URL https://indico.jlab.org/event/459/contributions/11847/attachments/9508/13784/CHEP2023___RAISE_Poster_FINAL.pdf

Apolloni B, Carvalho C, de Falco D (1989) Quantum stochastic optimization. Stochastic Processes and their Applications 33(2):233–244. https://doi.org/https://doi.org/10.1016/0304-4149(89)90040-9, URL https://www.sciencedirect.com/science/article/pii/0304414989900409

Baker B, Gupta O, Raskar R, et al (2017) Accelerating neural architecture search using performance prediction. https://doi.org/10.48550/ARXIV.1705.10823

Bishop CM (2006) Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, Berlin, Heidelberg

Boothby K, Bunyk P, Raymond J, et al (2020) Next-generation topology of d-wave quantum processors. https://doi.org/10.48550/ARXIV.2003.00133

Boser BE, Guyon IM, Vapnik VN (1992) A training algorithm for optimal margin classifiers. In: Proceedings of the Fifth Annual Workshop on Computational Learning Theory. Association for Computing Machinery, New York, NY, USA, COLT '92, p 144–152, https://doi.org/10.1145/130385.130401

Bottou L, Chapelle O, DeCoste D, et al (2007) Support Vector Machine Solvers, pp 1–27

Burges CJ (1998) Data Mining and Knowledge Discovery 2(2):121–167. https://doi.org/10.1023/a:1009715923555

Cavallaro G, Willsch D, Willsch M, et al (2020) Approaching remote sensing image classification with ensembles of support vector machines on the d-wave quantum annealer. In: IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium, pp 1973–1976, https://doi.org/10.1109/IGARSS39084.2020.9323544

Choi V (2011) Minor-embedding in adiabatic quantum computation: Ii. minor-universal graph design. Quantum Information Processing 10(3):343–353. https://doi.org/10.1007/s11128-010-0200-3

Date P, Arthur D, Pusey-Nazzaro L (2021) Qubo formulations for training machine learning models. Scientific Reports 11(1):10029. https://doi.org/10.1038/s41598-021-89461-4

Dietterich TG (2000) Ensemble methods in machine learning. In: Multiple Classifier Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 1–15

Drucker H, Burges CJC, Kaufman L, et al (1996) Support vector regression machines. In: Mozer M, Jordan M, Petsche T (eds) Advances in Neural Information Processing Systems, vol 9. MIT Press, URL https://proceedings.neurips.cc/paper_files/paper/1996/file/d38901788c533e8286cb6400b40b386d-Paper.pdf

Eggensperger K, Müller P, Mallik N, et al (2021) HPOBench: A collection of reproducible multi-fidelity benchmark problems for HPO. In: Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2), URL https://openreview.net/forum?id=1k4rJYEwda-

Falkner S, Klein A, Hutter F (2018) BOHB: Robust and efficient hyperparameter optimization at scale. In: Proceedings of the 35th International Conference on Machine Learning, pp 1436–1445

Fischer SF, Feurer M, Bischl B (2023) OpenML-CTR23 – a curated tabular regression benchmarking suite. In: AutoML Conference 2023 (Workshop), URL https://openreview.net/forum?id=HebAOoMm94

He K, Zhang X, Ren S, et al (2016) Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp 770–778, https://doi.org/10.1109/CVPR.2016.90

Jamieson K, Talwalkar A (2016) Non-stochastic best arm identification and hyperparameter optimization. In: Gretton A, Robert CC (eds) Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, Proceedings of Machine Learning Research, vol 51. PMLR, Cadiz, Spain, pp 240–248, URL https://proceedings.mlr.press/v51/jamieson16.html

Kadowaki T, Nishimori H (1998) Quantum annealing in the transverse ising model. Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics 58:5355–5363. https://doi.org/10.1103/PHYSREVE.58.5355

Krizhevsky A (2009) Learning multiple layers of features from tiny images

Le Y, Yang XS (2015) Tiny ImageNet Visual Recognition Challenge

Li L, Jamieson K, DeSalvo G, et al (2017) Hyperband: a novel bandit-based approach to hyperparameter optimization. J Mach Learn Res 18(1):6765–6816. URL https://dl.acm.org/doi/abs/10.5555/3122009.3242042

Li L, Jamieson KG, Rostamizadeh A, et al (2018) Massively parallel hyperparameter tuning. CoRR abs/1810.05934. 1810.05934

Liu S, Zhang H, Jin Y (2022) A survey on computationally efficient neural architecture search. Journal of Automation and Intelligence 1(1):100002. https://doi.org/https://doi.org/10.1016/j.jai.2022.100002, URL https://www.sciencedirect.com/science/article/pii/S2949855422000028

McGeoch CC (2014) Adiabatic Quantum Computation and Quantum Annealing: Theory and Practice, vol 5. Morgan & Claypool Publishers, https://doi.org/10.2200/

S00585ED1V01Y201407QMC008

Mnmoustafa MA (2017) Tiny imagenet. URL https://kaggle.com/competitions/tiny-imagenet

Pasetto E, Riedel M, Melgani F, et al (2022) Quantum SVR for Chlorophyll Concentration Estimation in Water With Remote Sensing. IEEE Geoscience and Remote Sensing Letters 19:1–5. https://doi.org/10.1109/LGRS.2022.3200325

Pata J, Duarte J, Vlimant J, et al (2021a) MLPF: efficient machine-learned particle-flow reconstruction using graph neural networks. The European Physical Journal C 81(5). https://doi.org/10.1140/epjc/s10052-021-09158-w

Pata J, et al (2021b) Simulated particle-level events of $t\bar{t}$ and QCD with PU200 using PYTHIA8+DELPHES3 for machine learned particle flow (MLPF). URL https://zenodo.org/record/4559324

Rebentrost P, Mohseni M, Lloyd S (2014) Quantum support vector machine for big data classification. Physical Review Letters 113(13). https://doi.org/10.1103/physrevlett.113.130503, URL http://dx.doi.org/10.1103/PhysRevLett.113.130503

Sirunyan AM, et al (2017) Particle-flow reconstruction and global event description with the CMS detector. J Instrum 12(10):P10003–P10003. https://doi.org/10.1088/1748-0221/12/10/p10003, 1706.04965

Vanschoren J, van Rijn JN, Bischl B, et al (2014) Openml: Networked science in machine learning. SIGKDD Explor Newsl 15(2):49–60. https://doi.org/10.1145/2641190.2641198

Weston J, Bordes A, Chopra S, et al (2015) Towards ai-complete question answering: A set of prerequisite toy tasks. 1502.05698

Willsch D, Willsch M, Raedt HD, et al (2020) Support vector machines on the d-wave quantum annealer. Computer Physics Communications 248:107006. https://doi.org/10.1016/j.cpc.2019.107006

Yu T, Zhu H (2020) Hyper-parameter optimization: A review of algorithms and applications. ArXiv abs/2003.05689. URL https://api.semanticscholar.org/CorpusID:212675087

# Appendix A Swfit-Hyperband Pseudocode

In this appendix, the pseudocode of the sequential version of Swift-Hypeband is presented. As it can be seen in Algorithm 1, Swift-Hyperband has five parameters: $R, \eta, d, \phi$ and *known_curve*. The parameters $\eta$ and $R$ have the same function as the parameters with the same respective name in the original Hyperband algorithm. That is, $\eta$ controls the trial discarding ratio at the end of every round. For $\eta = 2$ only the best-performing half of all configurations at the end of a given round are promoted to the next round, for $\eta = 3$ only the best third is promoted, etc. Therefore, increasing the value of $\eta$ makes the algorithm more aggressive. The least aggressive setting $\eta = 2$ is used by default in the experiments of this study. The quantity $R$ defines the target epoch for the HPO process, i.e., no configuration is trained for more than $R$ epochs. It is a problem-dependent parameter that is chosen by the user depending on the model or architecture to optimize. The remaining parameters $d$, $\phi$ and *known_curve* are specific for Swift-Hyperband. The quantity $d$ represents the minimum number of learning curves required to train each performance predictor, $\phi$ is the minimum fraction of trials that is trained until the end of each round independently of their predicted performance, and *known_curve* controls the position of the extra decision point. The natural choice *known_curve* $= 0.5$ places the new decision points exactly in the middle of each Hyperband round.

**Algorithm 1** Swift-Hyperband

**Input:** $R, \eta, d, \phi, known\_curve$
**Output:** Configuration with lowest loss seen during the algorithm

**Initialize:** $s_{max} = \lfloor \log_\eta(R) \rfloor$, $B = (s_{max} + 1)R$, $D = $ Dict(), $M = $ Dict()
**for** $s \in \{s_{max}, s_{max-1}, ..., 0\}$ **do**
    $n = \lceil \frac{B\eta}{R(s+1)} \rceil$, $r = R\eta^{-s}$
    `/* Begin Succesive Halving with n different configurations */`
    **for** $i \in \{0, ..., s\}$ **do**
        $n_i = \lfloor n\eta^{-i} \rfloor$
        $r_i = r\eta^i$
        **if** $r_i \notin D.keys$: $D[r_i] = $ List()
        **if** $r_i \notin M.keys$: $M[r_i] = $ Dict()
        $n_{next} = \lfloor \frac{n_i}{\eta} \rfloor$ **if** $i \neq s$, **else** 1
        `/* Performance prediction and parallelization (if applied) take part inside the following routine */`
        $L = $ run_then_return_val_loss$(T, r_{prev}, r_i, n_{next}, d, \phi, D, M, known_curve)$
        $T = $ top_k$(T, L, \lfloor n_i/\eta \rfloor)$
        $r_{prev} = r_i$
    **end for**
**end for**

**Algorithm 2** Routine run_then_return_val_loss for Swift-Hyperband

**Input:** $T, r_{prev}, r_i, n_{next}, D, M, d, \phi, known\_curve$
**Output:** $L$ (List with the final loss for each configuration in $T$)

**Initialize:**
$L =$ List(), $fully\_trained = 0, dp = r_{prev} + \lceil (r_i - r_{prev}) known\_curve \rceil, thres = 0$
**for** $t \in$ T **do**
  $l =$ List()
  **for** $i \in \{0, .., dp - 1\}$ **do**
    $l_i =$ loss_of_t_at_epoch_i($t, i$)
    $l$.append($l_i$)
  **end for**
  **if** $fully\_trained < \phi \cdot n_{next}$ **or** $D[r_i]$.length $< d$ **then**
    **for** $i \in \{dp, .., r_i\}$ **do**
      $l_i =$ loss_of_t_at_epoch_i($t, i$)
      $l$.append($l_i$)
    **end for**
    $L$.append($l_i$)
    $fully\_trained = fully\_trained + 1$
    $D[r_i]$.append($l$)
    $thres = L$.get_quantile(0.25)
  **else**
    **if** $dp \notin M[r_i].keys$ **then**
`/* model to predict performance at r_i using the learning curve until dp */`
      $M[r_i][dp] =$ PerformancePredictor()
      $M[r_i][dp]$.train($D[r_i][:, 0 : dp]$, $D[r_i][:, r_i]$)
    **end if**
    $pred = M[r_i][dp]$.predict($l$)
    **if** $pred < thres$ **then**
      **for** $i \in \{dp, .., r_i\}$ **do**
        $l_i =$ loss_of_t_at_epoch_i($t, i$)
        $l$.append($l_i$)
      **end for**
      $L$.append($l_i$)
      $fully\_trained = fully\_trained + 1$
      $D[r_i]$.append($l$)
    **else**
      $L$.append($\infty$)
    **end if**
  **end if**
**end for**

# Paper VI

**Accelerating Hyperparameter Tuning of a Deep Learning Model for Remote Sensing Image Classification.**

M. Aach, R. Sedona, A. Lintermann, G. Cavallaro, H. Neukirchen, and M. Riedel

IGARSS IEEE International Geoscience and Remote Sensing Symposium (2022), pp. 263–266

Copyright © 2024 IEEE. Reprint for this dissertation permitted.

Marcel Aach wrote the main part of the code and the paper, ran all experiments on the HPC machines, and performed the analysis. The application of the developed method to the RS use case was performed in collaboration with R. Sedona.

# ACCELERATING HYPERPARAMETER TUNING OF A DEEP LEARNING MODEL FOR REMOTE SENSING IMAGE CLASSIFICATION

*Marcel Aach*[1,2]*, Rocco Sedona*[1,2]*, Andreas Lintermann*[2]*, Gabriele Cavallaro*[2]*,*
*Helmut Neukirchen*[1]*, Morris Riedel*[1,2]

[1] School of Engineering and Natural Sciences, University of Iceland, Iceland
[2] Jülich Supercomputing Centre, Forschungszentrum Jülich, Germany

## ABSTRACT

Deep Learning models have proven necessary in dealing with the challenges posed by the continuous growth of data volume acquired from satellites and the increasing complexity of new Remote Sensing applications. To obtain the best performance from such models, it is necessary to fine-tune their hyperparameters. Since the models might have massive amounts of parameters that need to be tuned, this process requires many computational resources. In this work, a method to accelerate hyperparameter optimization on a High-Performance Computing system is proposed. The data batch size is increased during the training, leading to a more efficient execution on Graphics Processing Units. The experimental results confirm that this method reduces the runtime of the hyperparameter optimization step by a factor of 3 while achieving the same validation accuracy as a standard training procedure with a fixed batch size.

***Index Terms*—** Hyperparameter Tuning, Deep Learning, Batch Size, High-Performance Computing, Remote Sensing.

## 1. INTRODUCTION

The enormous investments that made freely available data acquired by modern Earth Observation (EO) programs have democratized access to timely satellite imagery of the entire planet. Missions such as the Copernicus Sentinel-2 can re-observe the same area every 5 days (under cloud-free conditions) by exploiting two polar orbiting satellites. Its free data represent an invaluable asset for tackling challenges as wide-ranging and important as quantifying the effects of climate change through land cover classification, vegetation mapping, environmental monitoring, etc. [1]. Nevertheless, the extraction of valuable information from raw satellite data is complex and requires large amounts of labelled training samples when using supervised learning with Deep Learning (DL) models. Furthermore, to achieve the best performance from a DL model, it is fundamental to optimize the values of its hyperparameters. This optimization step requires several processing steps that may consume a lot of computing power, i.e., leading to long processing times.

The present manuscript contributes to the Remote Sensing (RS) community by exploring a way to reduce these computational costs. While the group's previous work [2] focused on using evolutionary methods, this work aims at reducing hyperparameter tuning costs by training with a large batch size $BS$ without sacrificing validation accuracy. By running the hyperparameter tuning more efficiently, it becomes faster and cheaper for the community to find the best performing models. The experiments make use of the BigEarthNet-19 dataset [3]. It consists of 590,326 patches extracted from 125 Sentinel-2 tiles, each associated to one or more of the 19 labels of the simplified legend of the CORINE Land Cover [4], a thematic map from 10 European countries updated in 2018.

## 2. PROBLEM FORMULATION

The performance of deep neural networks depends on the hyperparameters set by the user before training. This usually involves a lot of manual tuning but may yield huge gains in performance. The main problem in finding the right set of hyperparameters is the expensive evaluation of different configurations. Each of them requires a full model training run. In principle, two main strategies for reducing the overall computational costs exist: (1) improving the choice of hyperparameters with optimization algorithms and (2) reducing the runtime of the training runs. This work focuses on the latter.

Large batch size parameter values $BS$ are necessary when a large dataset is used for training on multiple Graphics Processing Units (GPUs) on an High-Performance Computing (HPC) system. In this case, a large $BS$ value (that still fits into the GPU memory) leads to a higher GPU utilization, increasing the efficiency. $BS$ values of up to 80,000 samples have been reported for a Convolutional Neural Network (CNN) [5].

This work was performed in the Center of Excellence (CoE) Research on AI- and Simulation-Based Engineering at Exascale (RAISE) receiving funding from EU's Horizon 2020 Research and Innovation Framework Programme H2020-INFRAEDI-2019-1 under grant agreement no. 951733. The authors gratefully acknowledge the computing time granted by the JARA Vergabegremium and provided on the JARA Partition part of the supercomputer JURECA at Forschungszentrum Jülich.

However, training with large $BS$ values usually results in a lower validation performance, which is a general problem in distributed DL. Several techniques have been proposed to circumvent this problem, i.e., scheduling the learning rate $LR$ to slowly increase at the beginning and then decaying it over time [6], and using optimizers such as LARS [7] or LAMB [8] that introduce layerwise adaptive scaling mechanisms.

While these approaches focus on adjusting the parameter $LR$, this work adapts the parameter $BS$ itself to accelerate the process. Empirically, this has a similar effect as decaying the $LR$ over time [9] while using less parameter updates. Smith et al. [9] train a CNN on ImageNet starting with $BS = 8{,}000$ images, which is increased to $BS = 16{,}000$ after $E = 30$ epochs. Compared to the baseline of keeping the $BS$ constant at 8,000 throughout the whole training process, a $\approx 33\%$ faster convergence with no drop in validation accuracy ($76.1\%$) is reached. McCandlish et al. [10] introduce a metric called the Gradient Noise Scale (GNS) to predict the largest useful $BS$ value to be employed during each part of training. When using Stochastic Gradient Descent (SGD) with a small $BS$ value, the gradient update is a noisy approximation of the true gradient. A big batch resembles the true gradient much better. Following this intuition, the GNS measures the ratio of noise (variance) to signal (size) of the gradient. A large noise to signal ratio indicates that a bigger $BS$ value should be used and vice versa. Libraries such as Pollux [11] or KungFu [12] use the GNS and similar metrics to systematically optimize the throughput of DL models on HPC systems.

## 3. METHODOLOGY

### 3.1. Distributed Deep Learning

Training a neural network on large datasets can be time consuming as the the model needs to iterate though the whole input data once per epoch. One method to accelerate this process is to use data parallel training: the input data is split and distributed to different GPUs that all train separately on their own batches but perform a gradient synchronization at the end of each epoch. This way, the model on each GPU is the same but the data is different. Horovod [13] is an easy-to-use Python library that implements efficient data parallel training and was already used by us in the past to train on an RS dataset with up to 128 GPUs [14].

### 3.2. EfficientNet

The benefits of employing CNNs come at the cost of an increased computational budget. EfficientNet [15] is an architecture that, maintaining a fixed ratio between the width and the depth of the network, aims at decreasing the amount of parameters (i.e., weights and biases of the network) while maximizing the extraction of fine-grained and high-level features. It consequently curbs the usage of resources as compared to other benchmark models such as ResNet [16], reaching higher test accuracies while being of smaller size. Here, the EfficientNet-B0 [15] is used, a model achieving better test accuracies than ResNet-50 while having much less parameters ($5.3\,\mathrm{M}$ for EfficientNet-B0 vs. $26\,\mathrm{M}$ for ResNet-50).

### 3.3. Hyperparameter Optimization with Ray

Tuning the hyperparameters of a neural network involves training a lot of different sets of hyperparameters (*configurations*). A complete training run of said configuration is called a *trial*. Allocating resources and launching each trial manually is inefficient. Therefore, Ray[1] is used, which is an open-source library for distributed computing. Its subpackage Ray Tune can run distributed hyperparameter tuning at scale. It provides options to specify: the number of resources to use per trial, the hyperparameters, which range they are sampled from, and a scheduling or optimization algorithm. With this approach, a single Ray Tune job is started and Ray deals with all scheduling and communication tasks.

### 3.4. Changing the Batch Size

DL libraries like Tensorflow usually require the $BS$ to be set in the beginning of a training run and remain fixed throughout. To change the $BS$ using such libraries, it is necessary to train up to a certain epoch value $E$ with a fixed $BS$ value, check-point, and then continue the training with a different $BS$ value. This would introduce expensive memory access operations. Here, a different approach is followed. With the *GradientTape* mode of TensorFlow, an iterative way of calling the optimization steps has to be implemented, exposing the current batch in each iteration. This way, a big batch can be subdivided and the optimization step can be called on each of the smaller batches individually. Except for the epoch where the switch between $BS$ values occurs, tests have shown that no additional computational overhead is required when using this method, see Alg. 1 for a Python implementation.

## 4. EXPERIMENTAL RESULTS

### 4.1. Experimental Setup

The experiments are executed on the Jülich Research on Exascale Cluster Architectures (JURECA) system [17]. Its DC (data-centric) module features 192 accelerated compute nodes, each equipped with four NVIDIA A100 GPUs (with 40 GB high bandwidth memory each). The experiments use 24 nodes (96 GPUs in total) concurrently. The following Python libraries are employed: Horovod/0.23.0, TensorFlow/2.5.0, and Ray/1.8.0. The overall hyperparameter tuning run is launched with Ray Tune. Ray then allocates 4 nodes (16 GPUs) to each trial, within each trial data-parallel

[1] `https://www.ray.io/`

**Algorithm 1** Implementation of varying the batch size by batch subdivision.

```
# training iteration loop
for batch, (images, labels) in enumerate(dataset):
    split = 32 # factor of bigger to smaller batch
    # small batch case
    if (epoch < 20):
        # split up the original big batch into
            smaller batches
        images_split = np.array_split(images,
            split)
        labels_split = np.array_split(labels,
            split)
        # call the training step on each of the
            small batches
        for i in range(split):
            loss_value = training_step(
                images_split[i], labels_split[i])
    # big batch case
    else:
        loss_value = training_step(images, labels)
```

training is executed via Horovod. While on NVIDIA GPUs the preferred way of communication is through the NCCL[2] backend, Horovod in combination with Ray only supports the slower Gloo[3] backend, though.

The following hyperparameter ranges are evaluated: Learning rate $LR \in [10^{-3}, 1.0]$, momentum $M \in [0.0, 0.9]$, nesterov momentum $NM \in \{false, true\}$, and weight decay $WD \in [5{\cdot}10^{-5}, 10^{-1}]$. The selection of hyperparameters for a trial is performed with a random search. All models are trained for $E = 100$ epochs.

The training starts with a small batch size of $BS_{local} = 32$ per GPU ($BS_{global} = 16 \cdot 32 = 512$) and switches to $BS_{local} = 1{,}024$ per GPU ($BS_{global} = 16{,}384$) after $E = 20$. The choice of the $BS$ is motivated by our earlier results [14], where training on the BigEarthNet dataset was stable for $BS_{global} = 512$ but diverged for $BS_{global} = 16{,}384$. Switching $BS$ at $E = 20$ (at $20\%$ of the total training time) gives the optimizer sufficient time to equalize instabilities in early epochs and leads to a yet computationally efficient training with a larger $BS$ for $80\%$ of the time. The application of metrics like the GNS to achieve a more accurate guess of which $BS$ to use were unsuccessful. To evaluate the influence of this $BS$ value switching mechanism on the validation metrics and the computational resource consumption, the same 24 randomly sampled hyperparameter configurations are run once with and without varying $BS$. We provide the corresponding code in a GitLab repository[4].

[2] https://developer.nvidia.com/nccl
[3] https://github.com/facebookincubator/gloo
[4] https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/switching_bs

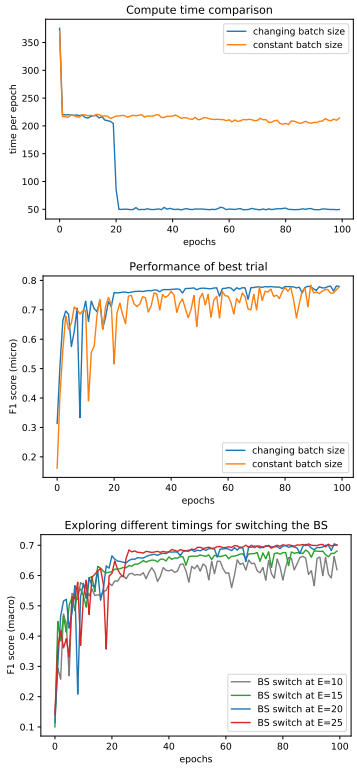



**Fig. 1**. **Top**: Mean time per epoch. **Center**: Best performing trial. **Bottom**: Comparison of timing for switch.

### 4.2. Evaluation

A comparison of the time per epoch with and without changing the $BS$ is shown in Fig. 1 (top). For the first $E = 20$ epochs, both methods take about the same time. Once the threshold is reached, the effect of the $BS$ value switch is visible as it is more than 4 times faster. The whole hyperparameter tuning run is about 3 times faster when varying the $BS$ as shown in Tab. 1. In terms of validation F1 scores (a weighted average of precision and recall), the best performing configuration for both approaches is $LR = 0.20735$, $M = 0.26415$, $WD = 5 \cdot 10^{-5}$, and $NM = false$. The scores achieved are in line with our earlier work [14, 2]. As the scores are almost similar for $BS = const.$ and the changing $BS$ method,

**Table 1**. Runtime of the hyperparameter tuning and accuracy of the best performing run for constant and changing batch size $BS$, showing accumulated and average trial runtime and validation F1 micro (macro) score.

| $BS_{global}$ | total runtime | trial runtime | F1 scores |
|---|---|---|---|
| 512 | 27 hrs | 355 mins | 0.78 (0.72) |
| 512 →16,384 | 10 hrs | 136 mins | 0.78 (0.70) |

the latter does not seem to suffer from the problem of a lower validation accuracy that large batch sizes usually come with.

The graphs in Fig. 1 (center) show the detailed training progress of the best performing trial. Overall, the training with a larger $BS$ value (changing $BS$ method) seems to be smoother than training with a smaller $BS$ (constant $BS$ method) for the whole duration. Furthermore, much of the training progress is already made in the first 20 epochs. Still, the last 80 epochs are necessary to achieve the final F1 scores. For the F1 micro score, the changing $BS$ approach even seems to perform slightly better, but this might be due to the smoothness of its training curve. In the end, both methods converge to a similar F1 score.

Figure 1 (bottom) evaluates the impact of the epoch switching on the training progress. Switching at $E = 10$ results in a lower final F1 macro score while with a switch at $E = 15$, the training performs similar to the original. Switching later in time leads to a better accuracy but also increases the runtime. However, as the graph for a switch at $E = 25$ shows, the gain is only marginal, so the original choice of $E = 20$ seems to be a good trade-off between accuracy and runtime. In all cases, the training never diverges, which indicates that the found hyperparameters seem to stabilize the optimizer even when training with larger batches earlier.

## 5. CONCLUSIONS

In this manuscript, a method to successfully accelerate the hyperparameter tuning process of a CNN trained with a RS dataset has been presented. Increasing the batch size during training from a smaller one in the beginning (to let the optimizer stabilize) to a larger one (to run data parallel training more efficient) seems to be a promising approach and does not reduce the validation accuracy. Compared to running the hyperparameter tuning with a fixed batch size, a speedup from 27 hours to 10 hours runtime on 96 GPUs has been achieved. While in this study, the focus has been on the BigEarthNet dataset, it would be interesting to see if the approach can also be transferred to other datasets and models.

## 6. REFERENCES


[1] J. Aschbacher, "ESA's earth observation strategy and Copernicus," in *Satellite Earth Observations and Their Impact on Society and Policy*. Springer, 2017.

[2] D. Coquelin et al., "Evolutionary optimization of neural architectures in remote sensing classification problems," in *IGARSS*. 2021, IEEE.

[3] G. Sumbul et al., "BigEarthNet dataset with a new class-nomenclature for remote sensing image understanding," 2021, arXiv: 2001.06372.

[4] M. Bossard, J. Feranec, and J. Otahel, "CORINE land cover technical guide – Addendum 2000," Tech. Rep. 40, European Environment Agency, Copenhagen, 2000.

[5] S. Kumar et al., "Exploring the limits of concurrency in ML training on Google TPUs," 2021, arXiv: 2011.03641.

[6] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," 2014, arXiv:1404.5997.

[7] Y. You et al., "Large batch training of convolutional networks," 2017, arXiv:1708.03888.

[8] Y. You et al., "Large batch optimization for deep learning: Training bert in 76 minutes," 2020, arXiv: 1904.00962.

[9] S. L Smith et al., "Don't decay the learning rate, increase the batch size," 2017, arXiv: 1711.00489.

[10] S. McCandlish et al., "An empirical model of large-batch training," 2018, arXiv: 1812.06162.

[11] A. Qiao et al., "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," 2021, arXiv: 2008.12260.

[12] L. Mai et al., "KungFu: Making training in distributed machine learning adaptive," in *OSDI 20*. Nov. 2020, pp. 937–954, USENIX Association.

[13] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," 2018, arXiv:1802.05799.

[14] R. Sedona et al., "Scaling up a Multispectral RESNET-50 to 128 GPUs," in *IGARSS*. 2020, IEEE.

[15] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," 2020, arXiv: 1905.11946.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015, arXiv: 1512.03385.

[17] P. Thörnig, "JURECA: Data centric and booster modules implementing the Modular Supercomputing Architecture at Jülich Supercomputing Centre," *Journal of large-scale research facilities*, vol. 7, 10 2021.

# References


[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[2] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale." In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=YicbFdNTTy.

[3] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, et al. "Highly accurate protein structure prediction with AlphaFold." In: *Nature* 596.7873 (2021), pp. 583–589. DOI: 10.1038/s41586-021-03819-2. URL: https://doi.org/10.1038/s41586-021-03819-2.

[4] Joosep Pata, Javier Duarte, Jean-Roch Vlimant, Maurizio Pierini, and Maria Spiropulu. "MLPF: efficient machine-learned particle-flow reconstruction using graph neural networks." In: *The European Physical Journal C* 81.5 (2021), p. 381. DOI: 10.1140/epjc/s10052-021-09158-w. URL: https://doi.org/10.1140/epjc/s10052-021-09158-w.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, et al. "Attention is All you Need." In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[6] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, et al. "PyTorch Distributed: Experiences on Accelerating Data Parallel Training." In: *Proceedings of Very Large Data Base Endowment Inc.* 13.12 (2020), pp. 3005–3018. ISSN: 2150-8097. DOI: 10.14778/3415478.3415530. URL: https://doi.org/10.14778/3415478.3415530.

[7] Tal Ben-Nun and Torsten Hoefler. "Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis." In: *ACM Comput. Surv.* 52.4 (2019). ISSN: 0360-0300. DOI: 10.1145/3320060. URL: https://doi.org/10.1145/3320060.

[8] Frank Hutter, Lars Kotthoff, and J. Vanschoren, eds. *Automatic machine learning: methods, systems, challenges*. English. Challenges in Machine Learning. Germany: Springer, 2019. ISBN: 978-3-030-05317-8. DOI: 10.1007/978-3-030-05318-5.

[9] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization." In: *Journal of Machine Learning Research* 18.1 (2017), pp. 6765–6816. ISSN: 1532-4435.

[10] Stefan Kesselheim, Andreas Herten, Kai Krajsek, Jan Ebert, Jenia Jitsev, et al. "JUWELS Booster – A Supercomputer for Large-Scale AI Research." In: *High Performance Computing*. Ed. by Heike Jagode, Hartwig Anzt, Hatem Ltaief, and Piotr Luszczek. Cham: Springer International Publishing, 2021, pp. 453–468. ISBN: 978-3-030-90539-2.

[11] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images." In: 2009.

[12] Gordon E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860.

[13] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[14] Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. USA, 1994.

[15] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management." In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4.

[16] Yannik Müller, Filipe Souza Mendes Guimarães, Carsten Karbach, and Wolfgang Frings. *LLview*. Version v2.3.1-base. July 2024. DOI: 10.5281/zenodo.12706843. URL: https://doi.org/10.5281/zenodo.12706843.

[17] Estela Suarez, Norbert Eicker, and Thomas Lippert. "Modular Supercomputing Architecture: from Idea to Production; 3rd." In: *Contemporary High Performance Computing: From Petascale toward Exascale, Volume 3*. Vol. 3. FL, USA: CRC Press, 2019, pp. 223–251. ISBN: 9781138487079. URL: https://juser.fz-juelich..

[18] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." In: *SIAM Journal on Computing* 26.5 (1997), pp. 1484–1509.

[19] John Preskill. "Quantum Computing in the NISQ era and beyond." In: *Quantum* 2 (Aug. 2018), p. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79. URL: https://doi.org/10.22331/q-2018-08-06-79.

[20] Tadashi Kadowaki and Hidetoshi Nishimori. “Quantum annealing in the transverse Ising model.” In: *Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics* 58 (5 1998), pp. 5355–5363. ISSN: 1063651X. DOI: 10.1103/PHYSREVE.58.5355.

[21] Gary Kochenberger, Jin-Kao Hao, Fred Glover, Mark Lewis, Zhipeng Lü, et al. “The unconstrained binary quadratic programming problem: a survey.” In: *Journal of Combinatorial Optimization* 28.1 (2014), pp. 58–81.

[22] Dennis Willsch, Madita Willsch, Hans De Raedt, and Kristel Michielsen. “Support vector machines on the D-Wave quantum annealer.” In: *Computer Physics Communications* 248 (Mar. 2020), p. 107006. DOI: 10.1016/j.cpc.2019.107006. URL: https://doi.org/10.1016%2Fj.cpc.2019.107006.

[23] Edoardo Pasetto, Morris Riedel, Farid Melgani, Kristel Michielsen, and Gabriele Cavallaro. “Quantum SVR for Chlorophyll Concentration Estimation in Water With Remote Sensing.” In: *IEEE Geoscience and Remote Sensing Letters* 19 (2022), pp. 1–5. DOI: 10.1109/LGRS.2022.3200325.

[24] Amer Delilbasic, Bertrand Le Saux, Morris Riedel, Kristel Michielsen, and Gabriele Cavallaro. “A Single-Step Multiclass SVM Based on Quantum Annealing for Remote Sensing Data Classification.” In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 17 (2024), pp. 1434–1445. DOI: 10.1109/JSTARS.2023.3336926.

[25] J. Kiefer and J. Wolfowitz. “Stochastic Estimation of the Maximum of a Regression Function.” In: *The Annals of Mathematical Statistics* 23.3 (1952), pp. 462–466.

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[27] Diederik Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization.” In: *International Conference on Learning Representations (ICLR)*. San Diego, CA, USA, 2015.

[28] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, et al. “GPipe: efficient training of giant neural networks using pipeline parallelism.” In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[29] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, et al. “Efficient large-scale language model training on GPU clusters using megatron-LM.” In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476209. URL: https://doi.org/10.1145/3458817.3476209.

[30] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. "Communication-Efficient Learning of Deep Networks from Decentralized Data." In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Ed. by Aarti Singh and Jerry Zhu. Vol. 54. Proceedings of Machine Learning Research. PMLR, 20–22 Apr 2017, pp. 1273–1282. URL: https://proceedings.mlr.press/v54/mcmahan17a.html.

[31] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Ananda Theertha Suresh, Dave Bacon, and Peter Richtárik. *Federated Learning: Strategies for Improving Communication Efficiency*. 2018. URL: https://openreview.net/forum?id=B1EPYJ-C-.

[32] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, et al. "Practical Secure Aggregation for Privacy-Preserving Machine Learning." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1175–1191. ISBN: 9781450349468. DOI: 10.1145/3133956.3133982. URL: https://doi.org/10.1145/3133956.3133982.

[33] Iacopo Colonnelli, Robert Birke, Giulio Malenza, Gianluca Mittone, Alberto Mulone, et al. "Cross-Facility Federated Learning." In: *Procedia Computer Science* 240 (2024). Proceedings of the First EuroHPC user day, pp. 3–12. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2024.07.003. URL: https://www.sciencedirect.com/science/article/pii/S1877050924016909.

[34] Matthias Feurer and Frank Hutter. "Hyperparameter Optimization." In: *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Cham: Springer International Publishing, 2019, pp. 3–33. ISBN: 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5_1. URL: https://doi.org/10.1007/978-3-030-05318-5_1.

[35] Ya Le and Xuan S. Yang. "Tiny ImageNet Visual Recognition Challenge." In: 2015.

[36] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, et al. "ImageNet Large Scale Visual Recognition Challenge." In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252.

[37] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, et al. "MLPerf Training Benchmark." In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 336–349.

[38] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "DARTS: Differentiable Architecture Search." In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=S1eYHoC5FX.

[39] Stefan Falkner, Aaron Klein, and Frank Hutter. "BOHB: Robust and Efficient Hyperparameter Optimization at Scale." In: *Proceedings of the 35th International Conference on Machine Learning*. 2018, pp. 1436–1445.

[40] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, et al. "A System for Massively Parallel Hyperparameter Tuning." In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 230–246.

[41] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. "OpenML: networked science in machine learning." In: *SIGKDD Explor. Newsl.* 15.2 (2014), pp. 49–60. ISSN: 1931-0145. DOI: 10.1145/2641190.2641198. URL: https://doi.org/10.1145/2641190.2641198.

[42] Matthias Feurer, Jan N. van Rijn, Arlind Kadra, Pieter Gijsbers, Neeratyoy Mallik, et al. "OpenML-Python: an extensible Python API for OpenML." In: *Journal of Machine Learning Research* 22.100 (2021), pp. 1–5. URL: http://jmlr.org/papers/v22/19-920.html.

[43] Giuseppe Casalicchio, Jakob Bossek, Michel Lang, Dominik Kirchhoff, Pascal Kerschke, et al. "OpenML: An R package to connect to the machine learning platform OpenML." In: *Computational Statistics* 34.3 (2019), pp. 977–991. DOI: 10.1007/s00180-017-0742-2. URL: https://doi.org/10.1007/s00180-017-0742-2.

[44] Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Pieter Gijsbers, Frank Hutter, et al. "OpenML Benchmarking Suites." In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. 2021. URL: https://openreview.net/forum?id=OCrD8ycKjG.

[45] Katharina Eggensperger, Philipp Müller, Neeratyoy Mallik, Matthias Feurer, Rene Sass, et al. "HPOBench: A Collection of Reproducible Multi-Fidelity Benchmark Problems for HPO." In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. 2021. URL: https://openreview.net/forum?id=1k4rJYEwda-.

[46] Gabriele Cavallaro, Morris Riedel, Matthias Richerzhagen, Jón Atli Benediktsson, and Antonio Plaza. "On Understanding Big Data Impacts in Remotely Sensed Image Classification Using Support Vector Machine Methods." In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 8.10 (2015), pp. 4634–4646. DOI: 10.1109/JSTARS.2015.2458855.

[47] Rocco Sedona, Gabriele Cavallaro, Jenia Jitsev, Alexandre Strube, Morris Riedel, and Jón Atli Benediktsson. "Remote Sensing Big Data Classification with High Performance Distributed Deep Learning." In: *Remote Sensing* 11.24 (2019). ISSN: 2072-4292. DOI: 10.3390/rs11243056. URL: https://www.mdpi.com/2072-4292/11/24/3056.

[48] Gencer Sumbul, Marcela Charfuelan, Begüm Demir, and Volker Markl. "Bigearthnet: A Large-Scale Benchmark Archive for Remote Sensing Image Understanding." In: *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*. 2019, pp. 5901–5904. DOI: 10.1109/IGARSS.2019.8900532.

[49] Marian Albers, Pascal S. Meysonnat, Daniel Fernex, Richard Semaan, Bernd R. Noack, et al. *Actuated Turbulent Boundary Layer Flows Dataset*. b2share, 2023. DOI: 10.34730/5dbc8e35f21241d0889906136cf28d26. URL: https://juser.fz-juelich.de/record/996125.

[50] Andreas Lintermann, Matthias Meinke, and Wolfgang Schröder. "Zonal Flow Solver (ZFS): a highly efficient multi-physics simulation framework." In: *Int. J. Comput. Fluid Dyn.* 34.7-8 (2020), pp. 458–485. ISSN: 1061-8562. DOI: 10.1080/10618562.2020.1742328.

[51] George Cybenko. "Approximation by superpositions of a sigmoidal function." In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[52] Mingxing Tan and Quoc Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 6105–6114. URL: https://proceedings.mlr.press/v97/tan19a.html.

[53] Eray Inanc, Rakesh Sarma, Marcel Aach, and Andreas Lintermann. *AI4HPC*. 2023. DOI: 10.5281/ZENODO.7705421. URL: https://zenodo.org/record/7705421.

[54] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. "Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction." In: *Artificial Neural Networks and Machine Learning – ICANN 2011*. Ed. by Timo Honkela, Włodzisław Duch, Mark Girolami, and Samuel Kaski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 52–59. ISBN: 978-3-642-21735-7.

[55] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, et al. "Asynchronous Stochastic Gradient Descent with delay compensation." In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 2017, pp. 4120–4129.

[56] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*. Available at https://arxiv.org/abs/1609.04836. 2017. arXiv: 1609.04836 [cs.LG].

[57] Alex Krizhevsky. *One weird trick for parallelizing convolutional neural networks*. arXiv:1404.5997. 2014. arXiv: 1404.5997 [cs.NE].

[58] Sadhika Malladi, Kaifeng Lyu, Abhishek Panigrahi, and Sanjeev Arora. "On the SDEs and Scaling Rules for Adaptive Gradient Algorithms." In: *Advances in Neural Information Processing Systems*. Ed. by Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho. 2022. URL: https://openreview.net/forum?id=F2mhzjHkQP.

[59] Yang You, Igor Gitman, and Boris Ginsburg. *Large Batch Training of Convolutional Networks*. 2017. arXiv: 1708.03888 [cs.CV]. URL: https://arxiv.org/abs/1708.03888.

[60] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, et al. "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes." In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=Syx4wnEtvH.

[61] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, et al. *Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds*. arXiv:1903.12650. 2019. arXiv: 1903.12650 [cs.LG].

[62] Sepp Hochreiter and Jürgen Schmidhuber. "Flat minima." In: *Neural Comput.* 9.1 (1997), pp. 1–42. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.1.1. URL: https://doi.org/10.1162/neco.1997.9.1.1.

[63] Chris Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-dickstein, Roy Frostig, and George Dahl. "Measuring the Effects of Data Parallelism on Neural Network Training." In: *Journal of Machine Learning Research (JMLR)* (2018). URL: https://arxiv.org/pdf/1811.03600.pdf.

[64] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. "Don't Decay the Learning Rate, Increase the Batch Size." In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=B1Yy1BxCZ.

[65] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. *An Empirical Model of Large-Batch Training*. 2018. arXiv: 1812.06162 [cs.LG]. URL: https://arxiv.org/abs/1812.06162.

[66] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, et al. "Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning." In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 1–18. ISBN: 978-1-939133-22-9. URL: https://www.usenix.org/conference/osdi21/presentation/qiao.

[67] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. "KungFu: Making Training in Distributed Machine Learning Adaptive." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 937–954. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/mai.

[68] Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Mądry. "FFCV: Accelerating Training by Removing Data Bottlenecks." In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2023, pp. 12011–12020.

[69] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, et al. "Mixed Precision Training." In: *ICLR*. 2018. URL: https://openreview.net/forum?id=r1gs9JgRZ.

[70] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. “Learning Sparse Low-Precision Neural Networks With Learnable Regularization.” In: *IEEE Access* 8 (2020), pp. 96963–96974. DOI: 10.1109/access.2020.2996936. URL: https://doi.org/10.1109%2Faccess.2020.2996936.

[71] Alexander Sergeev and Mike Del Balso. *Horovod: fast and easy distributed deep learning in TensorFlow*. arXiv:1802.05799. 2018. arXiv: 1802.05799 [cs.LG].

[72] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. “DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters.” In: New York, NY, USA: Association for Computing Machinery, 2020, pp. 3505–3506. ISBN: 9781450379984. DOI: 10.1145/3394486.3406703.

[73] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. “ZeRO: Memory optimizations Toward Training Trillion Parameter Models.” In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2020, pp. 1–16. DOI: 10.1109/SC41405.2020.00024.

[74] James Bergstra and Yoshua Bengio. “Random Search for Hyper-Parameter Optimization.” In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305. ISSN: 1532-4435.

[75] Kevin Jamieson and Ameet Talwalkar. “Non-stochastic Best Arm Identification and Hyperparameter Optimization.” In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*. Ed. by Arthur Gretton and Christian C. Robert. Vol. 51. Proceedings of Machine Learning Research. Cadiz, Spain: PMLR, 2016, pp. 240–248. URL: https://proceedings.mlr.press/v51/jamieson16.html.

[76] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E. Gonzalez, et al. “HyperSched: Dynamic Resource Reallocation for Model Development on a Deadline.” In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC ’19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 61–73. ISBN: 9781450369732. DOI: 10.1145/3357223.3362719. URL: https://doi.org/10.1145/3357223.3362719.

[77] Ujval Misra, Richard Liaw, Lisa Dunlap, Romil Bhardwaj, Kirthevasan Kandasamy, et al. “RubberBand: cloud-based hyperparameter tuning.” In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys ’21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 327–342. ISBN: 9781450383349. DOI: 10.1145/3447786.3456245. URL: https://doi.org/10.1145/3447786.3456245.

[78] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. “Algorithms for Hyper-Parameter Optimization.” In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger. Vol. 24. Curran Associates, Inc., 2011.

[79] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, et al. *Population Based Training of Neural Networks*. arXiv: 1711.09846. 2017. arXiv: 1711.09846 [cs.LG].

[80] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. “Neural Architecture Search: A Survey.” In: *J. Mach. Learn. Res.* 20.1 (2019), pp. 1997–2017. ISSN: 1532-4435.

[81] 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, 2015, pp. 3460–3468. ISBN: 9781577357384.

[82] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. *Freeze-Thaw Bayesian Optimization*. 2014. DOI: 10.48550/ARXIV.1406.3896. URL: https://arxiv.org/abs/1406.3896.

[83] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. “Learning Curve Prediction with Bayesian Neural Networks.” In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=S11KBYclx.

[84] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. “Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets.” In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Ed. by Aarti Singh and Jerry Zhu. Vol. 54. Proceedings of Machine Learning Research. PMLR, 2017, pp. 528–536. URL: https://proceedings.mlr.press/v54/klein17a.html.

[85] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. *Accelerating Neural Architecture Search using Performance Prediction*. 2017. DOI: 10.48550/ARXIV.1705.10823. URL: https://arxiv.org/abs/1705.10823.

[86] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. *Tune: A Research Platform for Distributed Model Selection and Training*. Available at https://arxiv.org/abs/1807.05118. 2018. DOI: 10.48550/ARXIV.1807.05118.

[87] Kirthevasan Kandasamy, Karun Raju Vysyaraju, Willie Neiswanger, Biswajit Paria, Christopher R. Collins, et al. “Tuning Hyperparameters without Grad Students: Scalable and Robust Bayesian Optimisation with Dragonfly.” In: *Journal of Machine Learning Research* 21.81 (2020), pp. 1–27. URL: http://jmlr.org/papers/v21/18-223.html.

[88] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, AndrÃ© Biedenkapp, Difan Deng, et al. “SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization.” In: *Journal of Machine Learning Research* 23.54 (2022), pp. 1–9. URL: http://jmlr.org/papers/v23/21-0888.html.

[89] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. “Optuna: A Next-generation Hyperparameter Optimization Framework.” In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD ’19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 2623–2631. ISBN: 9781450362016. DOI: 10.1145/3292500.3330701. URL: https://doi.org/10.1145/3292500.3330701.

[90] Prasanna Balaprakash, Michael Salim, Thomas D. Uram, Venkat Vishwanath, and Stefan M. Wild. “DeepHyper: Asynchronous Hyperparameter Search for Deep Neural Networks.” In: *2018 IEEE 25th International Conference on High Performance Computing*. 2018, pp. 42–51. DOI: 10.1109/HiPC.2018.00014.

[91] R. Egele, I. Guyon, V. Vishwanath, and P. Balaprakash. “Asynchronous Decentralized Bayesian Optimization for Large Scale Hyperparameter Optimization.” In: *2023 IEEE 19th International Conference on e-Science (e-Science)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023, pp. 1–10. DOI: 10.1109/e-Science58273.2023.10254839. URL: https://doi.ieeecomputersociety.org/10.1109/e-Science58273.2023.10254839.

[92] Prasanna Balaprakash, Romain Egele, Misha Salim, Stefan Wild, Venkatram Vishwanath, et al. “Scalable reinforcement-learning-based neural architecture search for cancer deep learning research.” In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356202. URL: https://doi.org/10.1145/3295500.3356202.

[93] Shengli Jiang and Prasanna Balaprakash. “Graph Neural Network Architecture Search for Molecular Property Prediction.” In: *2020 IEEE International Conference on Big Data (Big Data)*. Los Alamitos, CA, USA: IEEE Computer Society, 2020, pp. 1346–1353. DOI: 10.1109/BigData50022.2020.9378060. URL: https://doi.ieeecomputersociety.org/10.1109/BigData50022.2020.9378060.

[94] Ariel Keller Rorabaugh, Silvina Caíno-Lores, Travis Johnston, and Michela Taufer. “Building High-Throughput Neural Architecture Search Workflows via a Decoupled Fitness Prediction Engine.” In: *IEEE Transactions on Parallel and Distributed Systems* 33.11 (2022), pp. 2913–2926. DOI: 10.1109/TPDS.2022.3140681.

[95] Iacopo Colonnelli, Barbara Cantalupo, Concetto Spampinato, Matteo Pennisi, and Marco Aldinucci. “Bringing AI pipelines onto cloud-HPC: setting a baseline for accuracy of COVID-19 diagnosis.” In: *ENEA CRESCO in the fight against COVID-19* (Virtual). ENEA, Aug. 2021, pp. 66–73. DOI: 10.5281/zenodo.5151511. URL: https://doi.org/10.5281/zenodo.5151511.

[96] Oskar Taubert, Marie Weiel, Daniel Coquelin, Anis Farshian, Charlotte Debus, et al. “Massively Parallel Genetic Optimization Through Asynchronous Propagation of Populations.” In: *High Performance Computing*. Ed. by Abhinav Bhatele, Jeff Hammond, Marc Baboulin, and Carola Kruse. Cham: Springer Nature Switzerland, 2023, pp. 106–124. DOI: 10.1007/978-3-031-32041-5_6.

[97] Ilya Loshchilov and Frank Hutter. “SGDR: Stochastic Gradient Descent with Warm Restarts.” In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=Skq89Scxx.

[98] Xuanyi Dong, Lu Liu, Katarzyna Musial, and Bogdan Gabrys. “NATS-Bench: Benchmarking NAS Algorithms for Architecture Topology and Size.” In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), pp. 1–1. DOI: 10.1109/tpami.2021.3054824. URL: https://doi.org/10.1109%2Ftpami.2021.3054824.

[99] JP García Amboage, E Wulff, M Girone, and Tomás F Pena. *Optimizing AI-based HEP algorithms using HPC and Quantum Computing*. 2023. URL: https://indico.jlab.org/event/459/contributions/11847/attachments/9508/13784/CHEP2023___RAISE_Poster_FINAL.pdf.

[100] Estela Suarez, Anke Kreuzer, Norbert Eicker, and Thomas Lippert. “The DEEP-EST project.” In: *Porting applications to a Modular Supercomputer - Experiences from the DEEP-EST project*. Vol. 48. Schriften des Forschungszentrums Jülich IAS Series. Jülich: Forschungszentrum Jülich GmbH Zentralbibliothek, Verlag, 2021, pp. 9–25. URL: https://juser.fz-juelich.de/record/905812.

[101] Sebastian Felix Fischer, Matthias Feurer, and Bernd Bischl. “OpenML-CTR23 – A curated tabular regression benchmarking suite.” In: *AutoML Conference 2023 (Workshop)*. 2023. URL: https://openreview.net/forum?id=HebAOoMm94.

[102] Juan Pablo García Amboage, Eric Wulff, Maria Girone, and Tomás F. Pena. *Model Performance Prediction for Hyperparameter Optimization of Deep Learning Models Using High Performance Computing and Quantum Annealing*. 2023. arXiv: 2311.17508 [cs.LG]. URL: https://arxiv.org/abs/2311.17508.

[103] Mingxing Tan and Quoc Le. “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks.” In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 6105–6114. URL: https://proceedings.mlr.press/v97/tan19a.html.

[104] Rocco Sedona, Gabriele Cavallaro, Jenia Jitsev, Alexandre Strube, Morris Riedel, and Matthias Book. “Scaling Up a Multispectral Resnet-50 to 128 GPUs.” In: 2020 IEEE International Geoscience and Remote Sensing Symposium, Online event (Hawaii), 26 Sep 2020 - 2 Oct 2020. IEEE, 2020, pp. 1058–1061. DOI: 10.1109/IGARSS39084.2020.9324237. URL: https://juser.fz-juelich.de/record/890968.