Optimization of AI Methods on Distributed-Memory Computing Architectures

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr. Ing.)

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
Dissertation

von

Daniel Coquelin

Tag der mündlichen Prüfung: 31. Oktober 2024
Erster Gutachter: Prof. Dr. Achim Streit, Karlsruhe Institute of Technology
Zweiter Gutachter: Prof. Dr. Håkan Grahn, Blekinge Institute of Technology

Declaration

Erklärung zur Selbstständigen Anfertigung der Dissertationsschrift
Hiermit erkläre ich, dass ich die Dissertationsschrift mit dem Titel

Optimization of AI Methods on Distributed-Memory Computing Architectures

selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich oder inhaltlich übernommenen Stellen Stellen als solche kenntlich gemacht und die Regeln zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie (KIT) beachtet habe.

Ort und Datum Daniel Coquelin

Abstract

The increasing complexity of deep learning models necessitates innovative training techniques to overcome the bottlenecks inherent in large-scale distributed settings. This dissertation addresses these challenges by developing novel methods that minimize communication overhead and exploit inherent model redundancies without sacrificing accuracy.

First, I introduce Distributed, Asynchronous, and Selective Optimization (DASO), a hierarchical data-parallel training approach that strategically integrates asynchronous communication and stale gradients. DASO reduced communication overhead while maintaining convergence speed and accuracy, effectively mitigating the synchronization bottleneck in traditional synchronous training, resulting in an average training speedup of more than 25%.

Building upon DASO, I investigate the internal dynamics of neural network weights during training, revealing an early stabilization of the orthogonal bases within weight matrices. This key observation motivates the development of Orthogonality-Informed, Adaptive Low-Rank (OIALR) training. This novel method dynamically compresses models by identifying and discarding less informative components of weight matrices, resulting in significant reductions in model size and training time without compromising accuracy.

Further advancing efficient distributed training, I propose *AB* training, a hybrid approach that combines the low-rank representations of OIALR with a hierarchical training scheme inspired by DASO. By training independent worker subgroups on different components of decomposed weight matrices, *AB* training achieves a 70% reduction in communication volume across all experiments while meeting or exceeding the accuracy of standard synchronous training.

Finally, I explore the application of hyperparameter optimization (HPO) to automate the discovery of high-performing model configurations. Using the

`Propulate` framework, I demonstrate its efficacy in efficiently navigating large search spaces and finding optimal hyperparameters for image classification tasks on the challenging BigEarthNet dataset.

This dissertation presents a shift towards more efficient, asynchronous, and low-rank-aware distributed training. By embracing these innovative techniques, my research unlocks new possibilities for scaling deep learning to increasingly complex models and massive datasets, paving the way for advancements in a wide range of application domains.

Zusammenfassung

In dieser Dissertation wird eine umfassende Untersuchung der Herausforderungen und Chancen bei der Skalierung des Trainings neuronaler Netze in verteilten Umgebungen präsentiert. Der Schwerpunkt liegt auf Datenparallelität (DP), dem vorherrschenden Paradigma für das Training im großen Maßstab, und dem Kommunikationsengpass, mit dem es aufgrund der häufigen Synchronisation von Modellparametern konfrontiert ist. Um diesen Engpass zu beheben, stelle ich drei neue Methoden vor: Distributed Asynchronous and Selective Optimization (DASO), Orthogonality-Informed Adaptive Low-Rank (OIALR) Training und AB Training.

DASO nutzt hierarchische Kommunikation und asynchrone Updates, um den Kommunikationsaufwand erheblich zu reduzieren, das Training zu beschleunigen und gleichzeitig eine konkurrenzfähige Genauigkeit beizubehalten. Durch theoretische Analysen und empirische Bewertungen demonstrieren wir die Wirksamkeit von DASO sowohl bei der Bildklassifizierung als auch bei semantischen Segmentierungsaufgaben. Meine Untersuchung der Gewichtsdynamik während des Trainings zeigt die Stabilisierung orthogonaler Basen in frühen Stadien, was zur Entwicklung des Orthogonality-Informed Adaptive Low-Rank (OIALR) Trainings führt. Diese Methode nutzt die Gewichtsstabilisierung, um eine signifikante Modellkomprimierung zu erreichen, ohne die Leistung zu beeinträchtigen.

AB Training, eine Kulmination unserer Erkenntnisse, kombiniert niedrigdimensionale Darstellungen mit einem hierarchischen Trainingsschema, wodurch der Kommunikationsaufwand weiter reduziert und die Generalisierung in DP verbessert wird. Umfangreiche Experimente auf Standard-Benchmarks bestätigen die Effektivität von AB Training bei der Erzielung von Komprimierung und Genauigkeit, obwohl Herausforderungen bei extremen Skalierungen festgestellt werden.

Ich untersuche auch den Einsatz evolutionärer Hyperparameteroptimierung (HPO) und neuronaler Architektensuche (NAS) zur automatisierten Mo-

dellfindung und -optimierung. Ich demonstriere den Erfolg des Populate-Frameworks beim Auffinden leistungsstarker Architekturen und Hyperparameter für die Klassifizierung von Fernerkundungsbildern und betone das Potenzial automatisierter Methoden, die Modellentwicklung zu beschleunigen.

Meine Ergebnisse erweitern nicht nur das Verständnis der Trainingsdynamik neuronaler Netze, sondern präsentieren auch praktische Lösungen zur Minderung von Kommunikationsengpässen und zur Verbesserung der Modelleffizienz. Durch die Untersuchung asynchroner Updates, niedrigdimensionaler Darstellungen und hierarchischer Kommunikation eröffne ich neue Wege für ein skalierbares und effizientes verteiltes Training. Diese Arbeit hat das Potenzial, den Zugang zu hochleistungsfähigem Deep Learning zu erweitern, wissenschaftliche Entdeckungen zu beschleunigen und Innovationen in verschiedenen Bereichen voranzutreiben.

Obwohl meine Forschung wichtige Herausforderungen angeht, zeigt sie auch Bereiche für zukünftige Untersuchungen auf. Die Verfeinerung von Modellzusammenführungsstrategien, die Entwicklung maßgeschneiderter Hyperparameter-Zeitpläne und die Erforschung alternativer Update-Mechanismen sind entscheidend, um das Potenzial von niedrigdimensionalem, verteiltem Training bei extremen Skalierungen voll auszuschöpfen. Darüber hinaus ist ein tieferes theoretisches Verständnis des Zusammenspiels zwischen Batchgröße, niedrigdimensionalen Darstellungen und Generalisierung für die Weiterentwicklung des Feldes unerlässlich. Diese Dissertation trägt zu den laufenden Bemühungen bei, Deep Learning zugänglicher, effizienter und skalierbarer zu machen, und ermöglicht es Forschern und Praktikern letztendlich, immer komplexere und wirkungsvollere Probleme im Zeitalter von Big Data und groß angelegten Modellen anzugehen.

Acknowledgements

I hardly recognize the person who started this whole process. I know that I wouldn't be here without my supervisor Achim Streit. Thank you for your unwavering support, guidance, and encouragement throughout my journey. Your expertise, insightful feedback, and constructive criticism have been instrumental in shaping my research.

To Markus Götz, who encouraged me to start this path, and who helped to guide me along the way, I wish to express my deepest gratitude. Without your guidance, I would not be here.

I would like to extend my sincere thanks to all of my colleagues at KIT including: James Kahn, Julian Herold, Katharina Flgel, Jörg Mayer, Juan Pedro Gutiérrez Hermosillo Muriedas, Arvid Weyrauch, Nicholas Kiefer, and many more. You have made working here a genuine pleasure.

To Oskar Taubert, Jan Debus, Frank Schlegel, Markus Götz, and Charlie Debus, thank you for the years of laughter. I always look forward to Thursdays.

To Marie Weiel, thank you for all of the conversations when we definitely were not procrastinating. I dread that reality that soon I wont be able to drop by your office to chat. To Kalep Phipps, thank you for taking so much time to read over my drafts and for being a great friend.

I would like to express my heartfelt appreciation to my parents and brother for their love and support throughout my life. I would also like to thank them for the encouragement to follow my own path, even though it has led me far from them. You are always in my thoughts.

To my cats, Milka and Snickers, for sometimes being there when I needed comfort, if you felt like it. To my partner, Anne, thank you for your love and support. I am who I am today because of you.

All of you have touched my life in ways that I could never fully express. Because I knew you, I have been changed for good.

List of Publications

During my doctoral studies, I published the following papers as the first author, sorted in chronological order:

1. **D. Coquelin**, R. Sedona, M. Riedel, et al. "Evolutionary Optimization of Neural Architectures in Remote Sensing Classification Problems". In: 2021 IEEE International Geoscience and Remote Sensing Symposium IGARSS. 2021, pp. 1587–1590. DOI: 10.1109/IGARSS47720.2021.9554309.

2. **D. Coquelin**, C. Debus, M. Götz, et al. "Accelerating neural network training with distributed asynchronous and selective optimization (DASO)". en. In: Journal of Big Data 9.1 (Feb. 2022), p. 14. ISSN: 2196-1115. DOI: 10.1186/s40537-021-00556-1.

3. **D. Coquelin**, B. Rasti, M. Götz, et al. "Hyde: The First Open-Source, Python-Based, Gpu-Accelerated Hyperspectral Denoising Package". In: 2022 12th Workshop on Hyperspectral Imaging and Signal Processing: Evolution in Remote Sensing (WHISPERS). 2022, pp. 1–5. DOI: 10.1109/WHISPERS56178.2022.9955088.

4. **D. Coquelin**, K. Flügel, M. Weiel, et al. "Harnessing Orthogonality to Train Low-Rank Neural Networks". In: ECAI 2024. IOS Press, 2024, pp. 2106–2113. DOI: 10.3233/FAIA240729.

5. **D. Coquelin**, K. Flügel, M. Weiel, et al. "AB-Training: A Communication-Efficient Approach for Distributed Low-Rank Learning". In: (2024). URL: https://arxiv.org/abs/2405.01067. arXiv: 2405.01067 [cs.LG].

I have also assisted those around me and contributed to the following works during my studies:

- M. Götz, C. Debus, **D. Coquelin**, et al. "HeAT – a Distributed and GPU-accelerated Tensor Framework for Data Analytics". In: 2020 IEEE International Conference on Big Data (Big Data). 2020, pp. 276–287. DOI: 10.1109/BigData50022.2020.9378050.
- M. Weiel, M. Götz, A. Klein, **et al.** "Dynamic particle swarm optimization of biomolecular simulation parameters with flexible objective functions". In: Nature Machine Intelligence 3.8 (July 2021), pp. 727–734. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00366-3.
- C. Comito, **D. Coquelin**, M. Tarnawa, et al. helmholtz-analytics/heat: Scalable SVD, GSoC'22 contributions, Docker image, PyTorch 2 support, AMD GPUs acceleration (v1.3.0); 1.3.0. 2023. DOI: 10.5281/ZENODO.8060498.
- O. Taubert, M. Weiel, **D. Coquelin**, et al. "Massively Parallel Genetic Optimization Through Asynchronous Propagation of Populations". en. In: High Performance Computing. Ed. by A. Bhatele, J. Hammond, M. Baboulin, et al. Cham: Springer Nature Switzerland, 2023, pp. 106–124. isbn: 978-3-031-32041-5. DOI: 10.1007/978-3-031-32041-5_6.
- O. Taubert, F. von der Lehr, A. Bazarova, **et al.** "RNA contact prediction by data efficient deep learning". In: Communications Biology 6.1 (Sept. 2023). ISSN: 2399-3642. DOI: 10.1038/s42003-023-05244-9.
- K. Flügel, **D. Coquelin**, M. Weiel, et al. "Feed-Forward Optimization With Delayed Feedback for Neural Networks". In: (2023). DOI: 10.48550/arXiv.2304.13372.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ELU	exponential linear unit. 8
GELU	gaussian error linear unit. 8
IoU	intersection over union. 55, 57
MSE	mean square error. 15
ReLU	rectified linear unit. 8
AdamW	Adam with decoupled weight decay. 20, 87
ASGD	asynchronous stochastic gradient descent. 44
ASIC	application-specific integrated circuit. 21
BF16	Brain Floating Point. 37
BGD	Batch Gradient Descent. 17, 18
CNN	convolutional neural network. 10, 30, 36, 67
CP	Canonical Polyadic. 40
CPU	Central Processing Unit. 21, 53
CUDA	compute unified device architecture. 21
DASO	Distributed, Asynchronous, and Selective Optimization. iii, xvi, 43, 44, 46, 47, 49–52, 55–59, 61, 83, 117, 120
DDP	`DistributedDataParallel`. 27, 51, 88, 97
DNN	deep neural network. 6
DP	data parallel. xxiii, 25–27, 29, 30, 43, 45, 81, 82, 84–89, 96, 117, 118, 120
DPNN	data parallel neural network. 43–45
ECR	estimated communication reduction. xx, 98, 99
ETT	Electricity Transformer Temperature. 77
FP16	16-bit floating-point format. 37
FP32	32-bit floating-point format. 36, 37
GD	Gradient Descent. xv, 17, 18
GPU	Graphics Processing Unit. xvi, xviii, 21, 23, 24, 27, 37, 43, 46–48, 51–59, 88, 89, 91, 92, 112, 114, 148

Heat	Helmholtz Analytics Framework. 51
HP	hyperparameter. xix, xxi, 3, 4, 8, 10, 13, 20, 32–34, 53, 57, 69, 71, 73, 74, 77, 85, 88, 101, 103, 113, 119, 148, 151, 152
HPC	high-performance computing. 21, 22, 102, 104
HPO	hyperparameter optimization. iii, 4, 32–34, 71, 75, 101–104, 111–114, 119
iid	independent and identically distributed. 16, 46, 50, 117
LARS	Layer-wise Adaptive Rate Scaling. 29
LR	learning rate. 17, 19, 28, 29, 55, 56, 58, 76, 87, 148, 151, 152
ML	machine learning. 82
MLP	Multilayer Perceptron. 8, 9
MP	model parallel. 30
MPI	Message Passing Interface. 22, 23, 51, 52, 112
MPMD	Multiple Program Multiple Data. 22, 23
NAS	neural architecture search. xviii, 34, 102, 103, 105, 107–111
NCCL	NVIDIA collective communications library. 23
NLP	natural language processing. 11, 13
NN	neural network. 1, 4–6, 8, 10, 19, 24, 28, 32, 35, 113
ODE	Ordinary Differential Equation. 65
OIALR	Orthogonality-Informed, Adaptive Low-Rank. iii, xix, xxi, 63, 69–80, 98, 117–119, 148–151
RCCL	ROCm communication collectives library. 23
SGD	Stochastic Gradient Descent. 17–19, 25, 50, 51, 66, 84, 111
SPMD	Single Program Multiple Data. 22, 23
SVD	singular value decomposition. 39, 40, 62, 64, 66, 69, 74, 79, 81–83, 85, 90, 97, 118
TPU	Tensor Processing Unit. 21
ViT	Vision Transformer. xxi, 72, 73, 75, 76, 79, 88, 90, 92, 97, 149, 152

1. Introduction

1.1. Motivation

Neural networks (NNs) have emerged as a foundational tool within modern artificial intelligence, driving breakthroughs across diverse domains including natural language processing [3], computer vision [4], and scientific modeling [5]. Their success stems from their capacity to learn complex representations from vast amounts of data [6], enabling them to perform a wide array of tasks with exceptional effectiveness, often surpassing human capabilities in tasks like handwriting recognition, image classification, and language understanding [7].

The performance of NNs is intrinsically linked to the size of the model architecture and the training dataset [8]. As model size and data volume increase, so do the computational demands, necessitating the adoption of parallel, distributed-memory training techniques to efficiently utilize available resources. However, scaling the training process introduces significant challenges that remain largely unsolved, hindering the development and deployment of ever-larger and more powerful neural networks.

Different tasks exhibit varying degrees of inherent parallelism. Embarrassingly parallel tasks, such as 3D video rendering, can be divided into independent subtasks with minimal communication overhead, potentially leading to near-linear speedups with increasing resources. Conversely, serial tasks cannot be divided into independent parts and offer no opportunities for parallelization. Neural network training occupies a complex middle ground, presenting unique challenges to efficient parallelization.

While neural network training can be parallelized in various ways, two primary approaches dominate: model parallelism and data parallelism. Model parallelism distributes the operations within a network across multiple devices, requiring communication during both the forward and backward passes.

Although potentially effective for very large models, this approach often requires a more complex implementation and can result in load-balancing issues. Data parallelism, the more widely adopted approach, replicates identical model instances across devices, each processing a distinct subset of the data before aggregating gradients to maintain synchronized model parameters. While conceptually more straightforward, data parallelism faces challenges that limit its scalability. Communication overhead, especially the synchronization of large model representations, can become a significant bottleneck as model size increases. Furthermore, the large batch sizes stemming from large-scale data parallelism can detrimentally impact a network's ability to generalize to unseen data, a phenomenon known as large batch effects [9].

Despite considerable research effort, these challenges persist due to their inherent complexity. Optimizing communication patterns in distributed training involves navigating trade-offs between bandwidth usage, latency, and computational efficiency. Mitigating large batch effects requires a deeper understanding of the interplay between optimization algorithms, batch size, and generalization performance, an area where current theoretical understanding remains incomplete. Moreover, the rapid evolution of neural network architectures and training methodologies necessitates continuous adaptation and innovation in distributed training techniques.

Overcoming these computational and algorithmic barriers requires the development of innovative distributed training methodologies that prioritize communication efficiency, optimize resource utilization, and mitigate the adverse effects of large-scale training. This thesis delves into these challenges, exploring novel approaches to improve the scalability and efficiency of distributed neural network training. The key contributions and the structure of this thesis are visually summarized in Figure 1.1.

1.2. Research Questions

In this thesis I address the following central research questions:

Understanding and Addressing Large Batch Effects: Can novel optimization methods or training regimes be developed to counteract the detrimental effects of large batch training, leading to performance gains and improved generalization in large-scale settings?

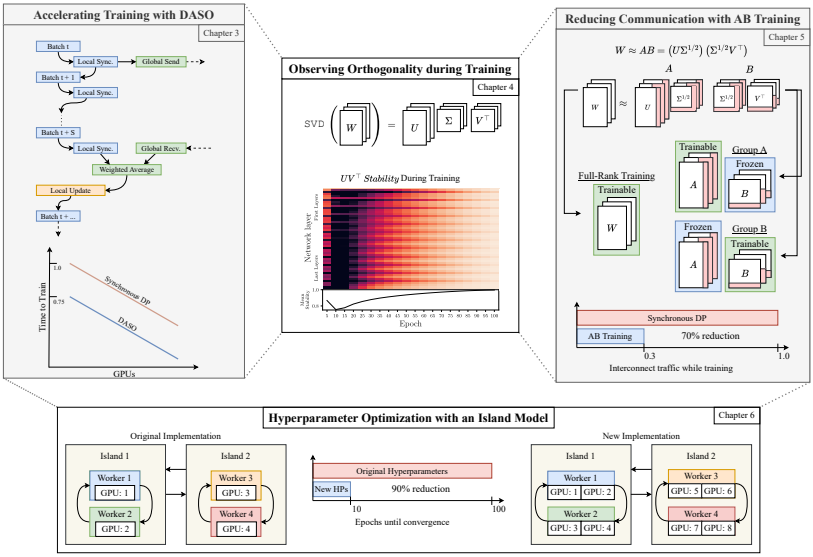


Figure 1.1.: The graphical abstract for this thesis.

Efficiency in Distributed-Memory Training: Can distributed training algorithms be designed to better balance computational and communication efficiency while scaling to accommodate the demands of ever-larger neural networks and datasets?

Exploiting Low-Rank Representations: To what extent can low-rank representations reduce the computational and communication requirements of distributed neural network training?

Distributed Hyperparameter Optimization: New methods come with different hyperparameters (HPs). How can the proper hyperparameters for new methods be found to maximize performance in a distributed setting?

1.3. Outline

The thesis is structured as follows:

Chapter 2: "Background" introduces the fundamental knowledge for this work. It includes information about how a neural network is trained with and without parallelism as well as an introduction into network compression methods and how hyperparameters are optimized.

Chapter 3: "Data Parallel Training Informed by Network Topology" presents a hierarchical training method which utilizes the computing cluster's network topology to accelerate training without loss of accuracy.

Chapter 4: "Orthogonality in Neural Networks" presents a finding surrounding the orthogonal bases of a NN's weights and how they evolve during training. After this I show how to use the finding to train low-rank neural networks which outperform traditional methods.

Chapter 5: "Using Low-Rank Representations in Data Parallel Training" uses the findings of Chapter 3 and Chapter 4 to train low-rank NNs in parallel to reduce network traffic without loss of accuracy. Furthermore, it studies large batch effects by analyzing how the low-rank representations of a network's weights change during scaling.

Chapter 6: "Tuning Training Methods by Choosing Better Hyperparameters" demonstrates how to find well-performing HPs for any method using evolutionary hyperparameter optimization (HPO) and how well the configurations found can perform.

2. Background

2.1. Notation

Throughout this thesis, I will use a standard notation in formulae. Multi-dimensional tensors will be in capital bold characters, i.e., $\boldsymbol{A}$, while vectors will be italicized bold characters, i.e., $\boldsymbol{a}$. For the neural network specific notation, I will use x as a single input and $\boldsymbol{X}$ as a set of inputs. A prediction made by a network, i.e., the output, will be referred to as $\hat{y}$, while the ground truth is y. The sets of predictions and ground truths are $\hat{\boldsymbol{Y}}$ and $\boldsymbol{Y}$, respectively. Variables representing individual numbers are italicized and will never be x or y to avoid confusion. A complete list of standard notations for this thesis is shown in Table 2.1.

2.2. Fundamentals of Artificial Neural Networks

NNs are a powerful and versatile class of machine learning algorithms loosely inspired by the structure of biological neural systems [10]. At their core, they excel at learning complex hierarchical representations [6] through a series of interconnected computational units called artificial neurons. These neurons are typically composed of a mathematical function, typically multiplication with a learned weight, and the optional addition of a learned bias term.

The first artificial neuron was the Perceptron [10]. The Perceptron was designed to classify an image by taking multiple inputs, weighing them linearly, and then applying a heavyside step-function to determine whether the input belonged to a class. The mathematical formulation of the Perceptron is

$$z = h\left(\boldsymbol{x} \cdot \boldsymbol{w}\right) \tag{2.1}$$

Table 2.1.: Common notations to be used through this thesis.

Symbol	Meaning
$\forall$	For all
$\in$	Is an element of
$\mathbb{R}$	Set of real numbers
a	A variable
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
x	A single data element input to an NN
$\boldsymbol{X}$	A set of data elements input to an NN stacked on the first dimension
$\hat{y}$	A single prediction made by an NN
$\hat{\boldsymbol{Y}}$	A set of predictions made by an NN
y	The ground truth, or label, of a data element NN
$\boldsymbol{Y}$	The set of ground truths for a given $\boldsymbol{X}$
w	An individual weight of a neural network
$\boldsymbol{W}$	A matrix of weights
b	A bias value
z	The activated output of an NN layer
$\mathcal{L}$	The loss function
η	The learning rate
$\boldsymbol{U}$	An orthonormal matrix of left singular vectors
$\sum_{i=1}^{n}$	Summation from $i = 1$ to n
Σ	A diagonal matrix of singular values sorted in descending order
$\boldsymbol{V}$	An orthonormal matrix of right singular values

where h is the heavyside function and $\boldsymbol{x} \cdot \boldsymbol{w}$ is the dot product of the input vector and the learned weights. While the Perceptron laid the groundwork for artificial neural networks, its limitations spurred researchers to explore more complex architectures.

To construct a neural network, individual neurons are organized into layers, which are then arranged sequentially. The output of each layer serves as the input to the subsequent layer, creating a chain-like flow of information. Figure 2.1 shows a basic NN. deep neural network (DNN)s consist of an input layer, multiple hidden layers, and an output layer, all computed consecutively. The depth of a DNN is determined by its number of hidden layers. Nonlinearities are often introduced to assist in modeling complex functions. The structure of DNNs promotes learning increasingly complex features of the

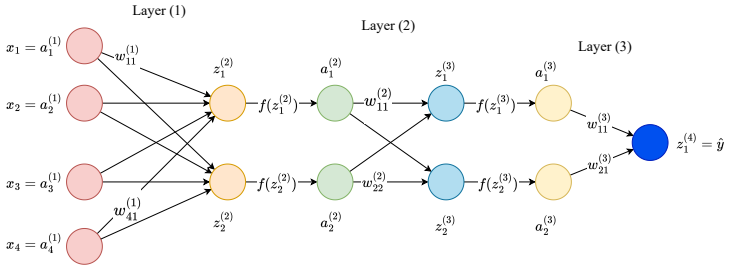
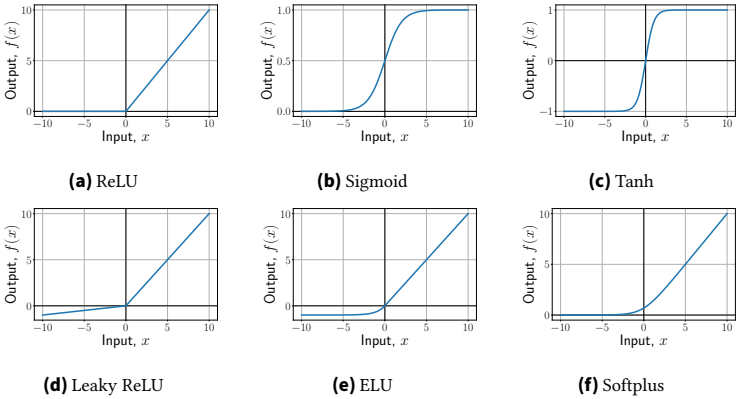


Figure 2.1.: A simple neural network composed of three linear layers without bias terms which takes an input of size four and outputs a single element.



(a) ReLU **(b)** Sigmoid **(c)** Tanh

(d) Leaky ReLU **(e)** ELU **(f)** Softplus

Figure 2.2.: Common Activation Functions. Note the changing scales of the y-axes.

input data elements. Deep networks are more effective than shallow networks (those with fewer layers) for specific problems [11].

Deep neural networks have many components. In the following subsections, I will discuss those most widely used.

2.2.1. Activation Functions

In neural networks, activation functions are applied element-wise to the output of individual neurons or entire layers. While the Perceptron initially

employed the Heaviside step function for binary classification, the role of activation functions has expanded significantly. By introducing non-linearity, they enable NNs to model complex relationships within data. Without activation functions, a network would be limited to a simple linear transformation, severely hindering its representational power.

The most common activation functions include rectified linear unit (ReLU), exponential linear unit (ELU), gaussian error linear unit (GELU), sigmoid, and hyperbolic tangent [12]. These functions all behave differently, and the choice of activation function can greatly effect the network's performance [13]. Figure 2.2 shows examples of common activation functions.

2.2.2. Multilayer Perceptrons

After the invention of the Perceptron, the field of artificial neural networks advanced to the Multilayer Perceptron (MLP), a structure composed of sequential linear layers (also known as fully connected layers) interwoven with nonlinear activation functions.

By establishing connections between every neuron in a layer to all neurons in the preceding layer, they are adept at discerning intricate, global relationships within input data. Each neuron in a linear layer receives input from all neurons in the prior layer, enabling the layer's parameters to concentrate on capturing global interactions and learning complex mappings across the entire input space.

The mathematical formulation of a linear layer with a nonlinear activation starts with the matrix multiplication of the inputs with the weight matrix. Then, optionally, a bias vector is added before the activation function is applied. Assuming a single input, $\boldsymbol{x}_n$, which has n features and an output, $\hat{\boldsymbol{y}}_p$, with p features, this layer takes the form of

$$\boldsymbol{z}_p = \boldsymbol{x}_n \boldsymbol{W}_{n \times p} + \boldsymbol{b}_p, \qquad \hat{\boldsymbol{y}} = f(\boldsymbol{z}) \tag{2.2}$$

where $\boldsymbol{W}_{n \times p}$ is the matrix of the weights, $\boldsymbol{b}_p$ is the vector of the bias values, and f is the nonlinear activation function. Figure 2.1 shows a simple diagram of a deep neural network without bias terms.

The most critical HPs for MLPs are:

Output Features: The number of features that will come out of the MLP

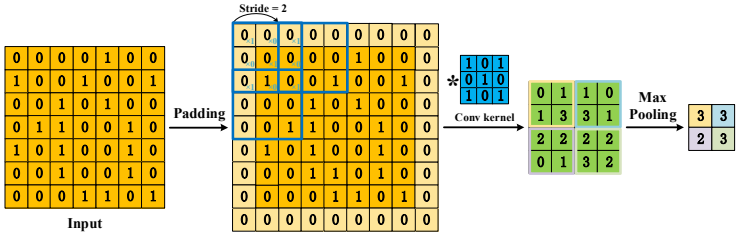


Figure 2.3.: Visualization of the operations of a typical convolution layer including the pooling operation at the end. Source: Li et al. [14].

Bias: Whether or not to add the learned bias terms to the output features

2.2.3. Convolution Blocks and Convolutional Neural Networks

Since the introduction of the MLP, the convolution (conv) layer has become a cornerstone of neural networks. Convolution layers are specialized for extracting spatially localized patterns from data possessing a grid-like topology, such as images [15]. They employ learnable filters, known as kernels, that 'slide' over the input, extracting intricate patterns within specific receptive fields. These filters can detect visually meaningful patterns such as edges, corners, shapes, or textures, providing a strong foundation for image classification or object detection tasks [16].

In the two-dimensional case, the output of a convolution layer at position (i, j) is

$$y(i,j) = (\boldsymbol{X} \star \boldsymbol{K})(i,j) = \sum_{h=0}^{q-1} \sum_{w=0}^{r-1} \boldsymbol{X}(i+h, j+w)\, \boldsymbol{K}(h,w) \tag{2.3}$$

for an input $\boldsymbol{X}$ and a convolution filter $\boldsymbol{K}_{q \times r}$. The operation involves element-wise multiplication between corresponding elements of the filter and the input, followed by summation over the filter's receptive field. This process is repeated for all spatial locations, effectively mapping the input to an output feature map.

Crucially, the same learned filter, and thus the same set of weights, is applied at every location in the input, a concept known as weight sharing. This

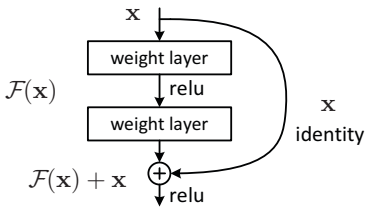


Figure 2.4.: A basic residual learning building block. Source: He et al. [4].

results in enhanced computational efficiency and robustness when performing the same task. Moreover, weight sharing instills a degree of translational invariance, enabling the network to recognize features accurately regardless of their position within the input – a vital property for computer vision applications.

A convolutional neural network (CNN) is built with multiple convolution blocks, commonly composed of convolution layers, normalization layers [17, 18], pooling layers [19], and a residual connection [4]. Pooling operations, e.g., max pooling or average pooling, downsample feature maps by aggregating information over spatial regions, further improving computational efficiency by reducing memory requirements while introducing robustness to minor input transformations and distortions. An example of a convolution layer followed by a max pooling operation is shown in Figure 2.3.

Normalization layers and residual connections, though applicable to various NN architectures, play a particularly crucial role in the design of modern CNNs. Normalization layers, like *BatchNorm* [17] and *LayerNorm* [18], normalize the inputs to the layer. *BatchNorm* normalizes across the batch dimension, while *LayerNorm* directly estimates the normalization statistics from the inputs and normalizes all features within each sample. Both of these methods have learnable affine transform parameters. Residual, or skip, connections, introduced to CNNs via ResNets [4], provide a path for data to reach the latter parts of the neural network by skipping layers. A basic network building block with a residual connection is shown in Figure 2.4.

Convolution layers have several important hyperparameters (HPs):

Kernel Size: The kernel size the filter's receptive field (e.g., a 3×3 filter to be convolved with the input). Smaller filters emphasize fine-grained, localized features, while larger filters capture wider-ranging patterns.

Stride: The stride parameter controls the step size by which the filter is shifted across the input. For example, with a stride of two (see Figure 2.3), the filter will move over by two elements instead of the default one. Larger strides decrease output feature map dimensionality and increase computational efficiency.

Padding: The padding parameter determine how many extra elements, usually of value zero, are concatenated around the input's borders impacts the output's spatial dimensions and prevents the shrinkage of dimensions with successive convolutions.

Output Channels: The number of output channels determines the depth of the output feature map. Each output channel results from applying a distinct filter across all input channels.

2.2.4. Self-Attention and Transformers

The self-attention mechanism [20] has revolutionized deep learning. Initially designed for natural language processing (NLP), it is now widely applied across diverse domains. Self-attention, also known as scaled dot-product attention, operates on input data sequences, where each element (e.g., a word in a sentence) is represented as a numerical vector, known as an embedding. Self-attention enables a model to weigh the importance of different elements in the sequence in relation to each other, facilitating the capture of complex relationships and dependencies across the entire input, particularly those spanning long distances.

At the core of self-attention is the computation of attention weights, which quantify the relevance of each input element to every other element in the sequence. These weights are derived through a series of transformations and comparisons.

First, the input embeddings are projected into the query ($\boldsymbol{q}$), key ($\boldsymbol{k}$), and value ($\boldsymbol{v}$) vector spaces through the learned transformations

$$\boldsymbol{q} = \boldsymbol{x}\boldsymbol{W}_q \qquad \boldsymbol{k} = \boldsymbol{x}\boldsymbol{W}_k \qquad \boldsymbol{v} = \boldsymbol{x}\boldsymbol{W}_v \tag{2.4}$$

where $\boldsymbol{W}_q$, $\boldsymbol{W}_k$, and $\boldsymbol{W}_v$ are the weight matrices for the query, key, and value transformations, respectively, and $\boldsymbol{x}$ is an input embedding.

After this, a similarity measure quantifies the relevance of each key to the query. One of the most common choices for this is scaled dot-product attention [20]:

$$\boldsymbol{S} = \frac{\boldsymbol{q}\boldsymbol{k}^\top}{\sqrt{d}} \tag{2.5}$$

where d is the number of elements in $\boldsymbol{k}$ and $\boldsymbol{S}$ is a matrix of un-normalized compatibility scores of the query $\boldsymbol{q}$ with the key $\boldsymbol{k}$.

Typically, scores are then normalized with softmax normalization to obtain alignment scores as

$$\boldsymbol{\alpha} = \frac{\exp\left(\boldsymbol{s}\right)}{\sum_j^d \exp\left(\boldsymbol{s}_j\right)} \tag{2.6}$$

Higher alignment scores, $\boldsymbol{\alpha}$, signify greater relevance of the corresponding value to the task. Value vectors are weighted according to their alignment scores and summed to generate the output: $\boldsymbol{z} = \sum_j^d \boldsymbol{\alpha}_j \boldsymbol{v}_j$. This aggregation step allows the model to incorporate information from relevant elements while decreasing the influence of less important ones. The output vector serves as the output representation for a specific position in the sequence, enriched with contextual information derived from the entire input [21].

The Transformer architecture [20], depicted in Figure 2.5, is a model structure that leverages self-attention layers. It has emerged as a leading design for state-of-the-art deep learning models. It follows an encoder-decoder structure, where the encoder processes the input sequence into a comprehensive latent representation, and the decoder generates the corresponding output sequence.

The Transformer has several vital features. First, it employs multi-head attention, which consists of multiple parallel self-attention mechanisms that operate independently. This enables each attention head to focus on different aspects of the input sequence and potentially learn different concepts, allowing the model to represent the relationships between elements more effectively. Second, the Transformer relies on positional encoding [22], a mechanism that injects information about the position of each element into the input embeddings. This is crucial because the Transformer processes all elements in parallel, and positional encoding allows it to maintain the sequential order of the input.

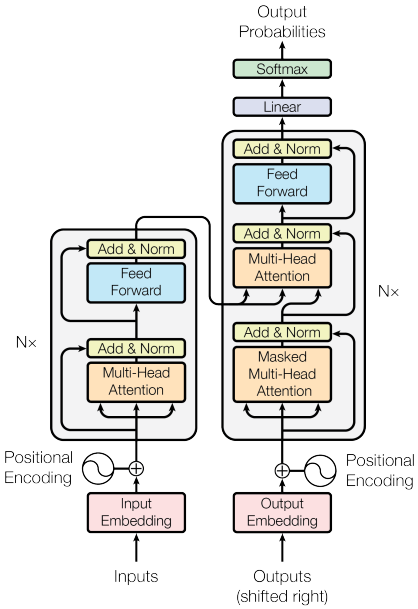


Figure 2.5.: The original Transformer model architecture. Source: Vaswani et al. [20].

Overall, the Transformer's combination of multi-head attention, positional encoding, and self-attention allows it to dynamically weigh the relevance of different sequence elements, granting it remarkable success in various tasks, including NLP [23], image recognition [24], and speech processing [25].

The most essential HPs for self-attention and the Transformer are:

Number of Attention Heads: This defines the number of parallel attention mechanisms, allowing the model to simultaneously focus on different aspects of the input.

Size of the Embedding Dimension: This defines the dimensionality of the input embeddings and the internal representations used in the attention mechanism.

2.3. Training a Neural Network

Neural networks are initialized with random weights and are typically trained using iterative, gradient-based optimization techniques. These gradients quantify the impact of each weight on the network's output or prediction. To compute these gradients, one typically leverages backpropagation [26], an application of the chain rule which propagates error information backward from the output layer to the input layer.

To illustrate the mechanics of backpropagation, let us consider a simple network comprising L linear layers, akin to the structure depicted in Figure 2.1, with a continuous output. For this example, I will use the following notation:

- $w_{jk}^{(l)}$: The weight connecting the j^{th} neuron in layer $l-1$ to the k^{th} neuron in layer l.
- $b_k^{(l)}$: The bias of the k^{th} neuron in layer l.
- $a_k^{(l)}$: The activated output of the k^{th} neuron in layer l.
- $z_k^{(l)}$: The weighted sum of the inputs to the k^{th} neuron in layer l before applying the activation function.
- $\mathcal{L}$: The loss function that quantifies the difference between the predicted output $\hat{y}$ and the ground truth y.

In the forward pass, an input data element x is propagated through the network. For every neuron k, each layer l performs the computation

$$z_k^{(l)} = \sum_j w_{jk}^{(l)} a_j^{(l-1)} + b_k^{(l)} \qquad \text{where} \qquad a_k^{(l)} = f\left(z_k^{(l)}\right) \tag{2.7}$$

where f is the activation function. For the first layer, $a_j^{(l-1)}$ is the input x_j. In this example, I will assume that the last layer has no activation function. This process continues until the last layer yields the predicted output $\hat{y} = z_k^{(L)}$. The predicted output is then compared to the ground truth by the loss function $\mathcal{L}$.

To optimize a network using a gradient-based method to improve the prediction quality, one must know how sensitive a prediction is to changes in each

weight. Mathematically, this is represented as the partial derivative of the loss function with respect to the individual weight, $\frac{\partial \mathcal{L}}{\partial w_{jk}}$.

For the output layer, this is relatively straightforward. Using the chain rule, we can find $\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}}$ as

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{jk}^{(L)}} \tag{2.8}$$

The derivative of the loss function defines the rate of change of the loss with respect to the output. For example, with the mean square error (MSE) loss, formulated as $\frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i)^2$ where n is the number of predictions, $\boldsymbol{y}$ is a vector of the ground truths, and $\hat{\boldsymbol{y}}$ is a vector of predictions, this is

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \delta_k^{(L)} = (\hat{y} - y) \cdot f'(z_k^{(L)}) \tag{2.9}$$

The rate of change of the output with respect to the weights, the second term in Equation (2.8), is the output of the previous layer or

$$\frac{\partial \hat{y}}{\partial w_{jk}^{(L)}} = a_j^{(L-1)} \tag{2.10}$$

Putting this together, the gradients for the output layer using the MSE loss function are

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}} = (\hat{y} - y) \cdot f'(\hat{y}) \cdot a_j^{(L-1)} \tag{2.11}$$

For the other layers of the network, the calculation changes slightly. Equation (2.8) becomes

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial w_{jk}^{(l)}} \tag{2.12}$$

where $\frac{\partial \mathcal{L}}{\partial z_k^{(l)}}$ is how much a small change in the output of a neuron k in layer l would change the loss value and $\frac{\partial z_k^{(l)}}{\partial w_{jk}^{(l)}}$ captures how changes to the weight $w_{jk}^{(l)}$ influence the output of the neuron k.

Using the chain rule and some substitutions, we can redefine Equation (2.12) as

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}} = \frac{\partial \mathcal{L}}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial z_k^{(l+1)}} \cdot a_k^l = \frac{\partial \mathcal{L}}{\partial a_k^{(l+1)}} f'\left(z_k^{(l+1)}\right) \cdot a_k^l \tag{2.13}$$

Using this, the gradients for the weights of layer l can be found once the gradients for the layer $l+1$ are found. Therefore, to calculate all the gradients, one must move backward through the network and propagate the gradients until the first layer, hence the name backpropagation. Although the gradients represent how much a weight affects the loss function and in which direction the weight should be adjusted to change the loss, it does not indicate how much the weight should be changed to improve network performance.

2.3.1. Data Partitioning, Generalization, and Overfitting

A fundamental assumption in machine learning is the independent and identically distributed (iid) assumption. It posits that each data point in a dataset is drawn independently from the same underlying probability distribution. In practice, it is often assumed that a finite dataset provides a reasonably good approximation of this true distribution, allowing us to treat the samples as approximately iid.

Dataset partitioning is crucial to promoting generalization. Under the iid assumption, the data elements which compose the training set can be considered representative samples of the underlying data distribution. This set is directly used for model optimization. A validation set, disjoint from the training set, is employed during training to periodically assess the model's performance on unseen data, quantifying how well the model has learned generalizable patterns. Finally, a held-out test set provides an unbiased evaluation of the fully trained model, offering a more realistic estimate of its expected performance in real-world deployment.

During model training, gradients guide the model towards a minima in the loss landscape, ideally corresponding to high predictive accuracy. However, given the substantial capacity of modern neural networks, models can easily overfit, memorizing the training data instead of learning generalizable patterns. Overfitting occurs when a model becomes excessively specialized to the training data, leading to poor generalization. A growing disparity

between training and validation performance during training is a standard indicator of overfitting. Achieving strong generalization is paramount for models to have real-world impacts.

2.3.2. Optimization

Optimization algorithms aim to efficiently traverse an objective function's landscape to determine a configuration that minimizes the objective function. In neural networks, the objective function is typically referred to as the loss function. This is primarily due to their robustness to high-dimensional problems, something with which other methods struggle [27].

Gradient Descent (GD) is a widely used iterative optimization algorithm that aims to minimize a function by moving in the direction of the steepest descent, shown in Figure 2.6, as indicated by the negative gradient. In the context of neural networks, Gradient Descent (GD) is employed to update the model's trainable parameters in order to reduce the loss function. There are three primary methods of GD: Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD), and Mini-batch SGD.

In BGD, gradients are computed using the *entire* dataset. While this guarantees convergence to the global minimum for convex functions, the computational cost of calculating gradients over the entire training set before each weight update can be prohibitive. Additionally, BGD's reliance on the full training dataset makes it susceptible to overfitting, where the model achieves high accuracy on training data but performs poorly on unseen validation and test data.

SGD [29] updates model weights using gradients computed from a single, randomly selected data point. While SGD often necessitates more iterations due to the inherent noise in these updates, it offers superior computational efficiency compared to BGD by eliminating the need to process the entire dataset before each weight update.

Formally, SGD updates a weight w_t at a time in training t as

$$w_{t+1} = w_t - \eta \nabla_w \mathcal{L}\left(\hat{y}, y\right) \tag{2.14}$$

where η is the learning rate (LR), a hyperparameter governing the magnitude of update steps, $\nabla_{w_t} \mathcal{L}\left(\hat{y}, y\right)$ is the gradient of the loss function $\mathcal{L}$ with respect

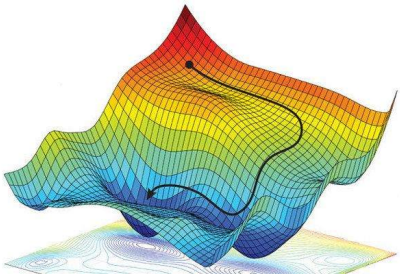


Figure 2.6.: An example loss landscape and a possible path which GD may take to optimize the problem. Source: Hutson [28].

to w, $\hat{y}$ and y are the network's prediction and the ground truth respectively. The magnitude of a weight's gradient indicates the sensitivity of the loss function to changes in that parameter. A positive gradient indicates that increasing the parameter's value should increase the loss, and vice versa for a negative gradient [29].

Mini-batch Stochastic Gradient Descent (SGD) is a foundational optimization algorithm widely used for training neural networks. It strikes a balance between BGD and SGD. In mini-batch SGD, gradients are computed using a randomly selected subset of the training data, referred to as a mini-batch or simply batch. This approach offers improved stability compared to traditional SGD and enhanced computational efficiency compared to BGD. Mini-batch SGD updates a given weight as

$$w_{t+1} = w_t - \frac{\eta}{B} \sum_{i=1}^{B} \nabla_{w_t} \mathcal{L}_i \left(\hat{y}_t, y_t \right), \tag{2.15}$$

where B is the number of data elements in the batch.

Mini-batch SGD offers several advantages over single-element SGD. Modern hardware and deep learning frameworks are highly optimized for parallelized computations, and mini-batch SGD effectively leverages this, as the network performs the same operations for each data element independently for significant speed gains. Additionally, while gradient estimates over batches exhibit less noise than single examples, a moderate level of noise can actually help escape shallow local minima and improve convergence [30]. The batch size thus acts as a hyperparameter influencing both the optimization process and the

model's generalization behavior, providing a tool for balancing convergence speed and regularization.

In this thesis, and in the NN community at large, mini-batch SGD is frequently referred to as simply SGD.

The learning rate (LR) is a critical hyperparameter in the optimization process. As the relative importance of individual parameters can vary depending on the input data, an excessively high LR can lead to divergence, while an overly small LR may result in slow convergence or premature stagnation in local minima. Learning rate schedulers introduce dynamism to the LR throughout the training process. A common strategy involves initializing the LR with a relatively high value to facilitate rapid initial progress, followed by a gradual reduction over time [31]. This allows the network to explore more broadly early in training while later focusing on fine-tuning solutions within the discovered region.

Momentum [32] is a technique commonly utilized with SGD to accelerate convergence and overcome local minima. It modifies the update rule as follows:

$$v_{t+1} = \beta v_t + (1 - \beta) \nabla_{w_t} \mathcal{L}(\hat{y}_t, y_t) \tag{2.16}$$

$$w_{t+1} = w_t - \eta v_{t+1} \tag{2.17}$$

where v_t represents the accumulated momentum at iteration t, and β is a hyperparameter. The gradients of previous iterations thus have an exponentially decaying effect on the most current update. This mechanism introduces a degree of inertia, damping oscillations and potentially enabling faster progress.

Weight decay [33] is a regularization strategy to prevent overfitting and improve generalization. It introduces a penalty term into the update rule which punishes larger weights:

$$w_{t+1} = (1 - \lambda) w_t - \eta \nabla_{w_t} \mathcal{L}(\hat{y}_t, y_t) \tag{2.18}$$

where λ is a hyperparameter controlling the strength of the regularization. Intuitively, weight decay favors simpler solutions by promoting weight magnitudes close to zero unless strong gradients from the data indicate otherwise.

SGD exhibits limitations stemming from its sensitivity to the LR and its uniform update policy. Finding an optimal learning rate schedule can demand

extensive experimentation. Additionally, a global learning rate may be suboptimal in scenarios with a high degree of feature frequency imbalance or when the loss landscape exhibits significant anisotropies.

Adaptive optimizers have emerged to address these issues. The most famous of these is the Adam [34] optimizer. Shortly after its introduction, Adam with decoupled weight decay (AdamW) [35] was released. AdamW maintains distinct learning rates for each weight, dynamically adjusting them based on the history of gradients. Much like momentum, AdamW incorporates exponential averaging of past gradients, potentially speeding up convergence [36]. It also leverages information approximating the second moment of gradients, enabling adaptive scaling of updates to handle varying parameter magnitudes better. The update rule for AdamW is

$$
\begin{aligned}
g_{t+1} &= \nabla_{w_t} \mathcal{L}\left(\hat{y}_t, y_t\right) \\
m_{t+1} = \beta_1 m_t + \left(1-\beta_1\right) g_{t+1} \qquad & v_{t+1} = \beta_2 v_t + \left(1-\beta_2\right) g_{t+1}^2 \\
\hat{m}_{t+1} = m_{t+1} / \left(1-\beta_1^{t+1}\right) \qquad & \hat{v}_{t+1} = v_{t+1} / \left(1-\beta_2^{t+1}\right) \\
w_{t+1} &= w_t - \frac{\eta_t \hat{m}_{t+1}}{\left(\sqrt{\hat{v}_{t+1}} + \epsilon\right)} + \gamma w_t
\end{aligned}
$$

where η_t is the learning rate at a given iteration t of training, γ is the weight decay HP, and β_1 and β_2 are HPs of the optimization algorithm determining the decay rates of the moving averages m and v. To optimize a network effectively, AdamW maintains an optimizer state, which tracks the m and v throughout training. Multiple applications within this thesis use the AdamW optimizer.

Despite the advancements offered by adaptive optimizers, there is no universally optimal optimizer. Factors influencing the best choice include the dataset's size, feature distributions, and inherent noise or anisotropy. The network's architecture (depth, width, and choice of nonlinearities) also plays a role. Considerations about computational resources are essential, as some optimizers have higher memory overhead than others, which becomes a consideration with limited hardware and larger networks.

2.4. Computing for Neural Networks

Even at modest scales, neural networks comprise millions of parameters. Each forward and backward pass involves numerous tensor operations, pri-

marily matrix multiplications and vector-matrix products. These operations exhibit inherent parallelism, with many element-wise computations being performed independently. This fine-grained parallelism is well-suited for modern hardware accelerators like Graphics Processing Unit (GPU)s and Tensor Processing Unit (TPU)s.

Originally developed for graphics rendering, Graphics Processing Unit (GPU)s possess thousands of cores optimized for the vector and matrix operations prevalent in deep learning [37]. Their massively parallel architecture and high memory bandwidth enable significant performance gains compared to traditional Central Processing Unit (CPU)s. In 2017, Google introduced the TPU, an accelerator with custom application-specific integrated circuit (ASIC)s tailored explicitly for tensor computations [38]. Custom ASICs designed for the specific tensor operations in neural networks are now commonplace in accelerators. These accelerators are often optimized for fused multiply-add (FMA) operations, a fundamental building block in neural network computations, further enhancing performance gains through vectorization.

Modern data center GPUs incorporate specialized cores for tensor operations, but efficient task scheduling is crucial to fully harness their computational power. GPU task schedulers queue operations and optimize execution to maximize throughput and minimize idle time, ensuring efficient resource utilization. The compute unified device architecture (CUDA) framework [39] is the most widely used platform for utilizing GPUs in deep learning, offering libraries and tools optimized for NVIDIA GPUs. For AMD GPUs, the open-source ROCm platform [40] provides a similar feature set.

While accelerators can significantly speed up computation on a single machine, many real-world applications demand computational resources that surpass the capabilities of a single device. These applications necessitate the use of parallel distributed-memory systems, where multiple interconnected computational nodes work together to tackle large tasks.

Each node within a distributed-memory system can be viewed as a standalone computer with its own processors, memory, and potentially multiple GPUs. A high-speed interconnect network connects the nodes, facilitating the exchange of information. Figure 2.7 shows a basic diagram of this. With sufficient nodes and computer power, the computing cluster becomes an high-performance computing (HPC) cluster.

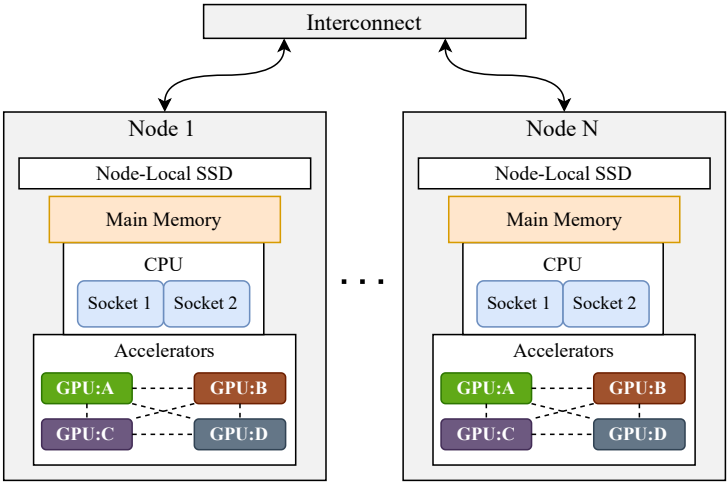
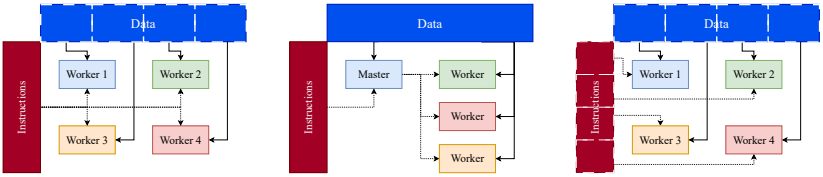


Figure 2.7.: Typical distributed-memory cluster computing setup. Storage not shown, all nodes connect to it independently.

Specialized programming models are crucial to fully exploit the potential of distributed-memory architectures. The Single Program Multiple Data (SPMD) paradigm, illustrated in Figure 2.8a, is widely adopted. In Single Program Multiple Data (SPMD), each processing unit executes the same program on a distinct subset of the data, simplifying code design and synchronization. In contrast, the Multiple Program Multiple Data (MPMD) paradigm, depicted in Figure 2.8c, offers greater flexibility by enabling different workers to specialize in specific tasks. However, this flexibility may come at the cost of increased development complexity due to the need for explicit inter-process communication and coordination [41]. A common implementation of Multiple Program Multiple Data (MPMD) is the master-worker paradigm, Figure 2.8b, where a designated master process orchestrates the distribution of tasks to the worker processes.

The Message Passing Interface (MPI) [42] standard, a staple in the HPC community since its introduction, is often used to facilitate the efficient usage of computing clusters. In this standard, the number of parallel workers is known as the world size and each worker gets a numerical identifier known as its rank. MPI implementations and bindings exist in several languages, and



(a) The SPMD model, where all workers receive all instructions and a subset of the data.

(b) The master-worker MPMD model, where one worker distributes the workload to the others.

(c) A general MPMD model, where each worker receives a subset of the instructions and a subset of the data.

Figure 2.8.: Comparison of Different Parallel Programming Models. The communication channels between workers are not shown.

the nomenclature has permeated nearly all communication frameworks. In addition to MPI, specialized communication libraries like NVIDIA collective communications library (NCCL) [43] and ROCm communication collectives library (RCCL) [44] are optimized for efficient communication between GPUs, further accelerating distributed training workflows.

Most of the experiments presented in this thesis utilized the distributed-memory, parallel hybrid supercomputer HoreKa at the Karlsruhe Institute of Technology Scientific Computing Center [45]. Each node on this system is equipped with two 38-core Intel Xeon Platinum 8,368 processors at 2.4 GHz base and 3.4 GHz maximum turbo frequency, 512 GB local memory, a local 960 GB NVMe SSD disk, two network adapters, and four NVIDIA A100-40 GPUs with 40 GB memory connected via NVLink [46] with 600 GB/s. Inter-node communication uses a low-latency, non-blocking NVIDIA Mellanox InfiniBand 4X HDR interconnect with 200 Gbit/s per port [47].

2.4.1. Frameworks

To facilitate the development and deployment of neural networks, several powerful software frameworks have emerged. These frameworks offer a high-level API for GPU computations, which includes a rich ecosystem of tools, libraries, and abstractions that streamline model definition, training, evaluation, and deployment. Most importantly, these frameworks provide automatic differentiation tools to calculate derivatives efficiently. Automatic differentiation constructs a computational graph representing the flow of data through the operations in a model [48]. It then applies the chain rule to

efficiently compute the gradient of the output with respect to each input, see Section 2.3.

Popular NN frameworks include TensorFlow [49], PyTorch [50], and Keras [51]. The choice of framework typically depends on individual preferences and specific project requirements.

2.5. Training Neural Networks on Distributed-Memory Architectures

While the accelerator-based approach works well for small networks and datasets, many applications, such as large-scale natural language processing [3], high-resolution image recognition [24], and complex scientific simulations [52], necessitate larger models and significantly more training data, rendering single-GPU training infeasible. This challenge underscores the critical need for algorithmic parallelism, where the training process is strategically designed to distribute computational work and data across multiple processing units, whether within a single machine or across a network of interconnected devices.

Two fundamental principles guide the understanding of how parallelism can enhance a program's performance: Amdahl's Law and Gustafson's Law. Amdahl's Law posits that the potential speedup achievable by optimizing a specific part of a system is inherently constrained by the proportion of time that part is actively utilized [53]. In essence, the parts of the system that necessitate serial execution impose a ceiling on the overall performance gains attainable through parallelization. This concept is often illustrated empirically through strong scaling measurements, where the problem size remains constant while computational resources are increased.

Gustafson's Law builds upon Amdahl's Law by considering scenarios where the execution workload expands in tandem with the availability of computational resources [54]. This principle underpins the concept of weak scaling measurements, where the workload per resource is held constant while both the number of resources and the problem size are increased proportionally.

This section delves into the various forms of algorithmic parallelism commonly employed in neural network training, with a particular focus on data

parallelism, given its widespread adoption and central relevance to the research conducted within this work.

2.5.1. Data Parallelism

Due to its simplicity, data parallel (DP) training has emerged as the dominant strategy for scaling neural network training [55]. Figure 2.9 shows a diagram of a standard data parallel training workflow. The model architecture is replicated across the available workers in the data parallel (DP) paradigm. Each worker is responsible for processing a disjoint subset of the training data for each forward-backward pass. However, the model representations must be the same on all workers when each batch is processed to maintain the standard mini-batch SGD update formula. Therefore, the gradients are either aggregated before the optimization step or the model parameters after the optimization step to maintain a synchronized model. As the synchronization is typically an average operation, this has the effect of increasing the mini-batch size by a factor of the number of workers.

The widespread adoption of DP training is due to several factors. First, its implementation often requires minimal modifications to existing sequential training code, rendering it a readily accessible solution for practitioners [56]. Second, DP exhibits favorable scalability with respect to dataset size, as increasing the number of workers directly translates to increased data processing capacity, to a certain threshold [55]. Lastly, DP training's conceptual simplicity allows it to be applied across a wide spectrum of model architectures, learning tasks, and hardware architectures, without the need for intricate task partitioning or specialized communication patterns [4, 24, 57].

Despite its strengths, synchronous DP training suffers a significant drawback: the synchronization bottleneck. The amount of communication required to synchronize a model scales with the model's size and, if not adequately addressed, the number of workers. This synchronization can cause a communication bottleneck, becoming a critical limiting factor in achieving optimal efficiency during training. [58].

Approaches like the Ring Allreduce algorithm [59], implemented in libraries such as Horovod [60], mitigate this issue by reducing the number of concurrent communications. In this scheme, each worker communicates with two neighbors at each step, unlike the all-to-all approach, where every worker

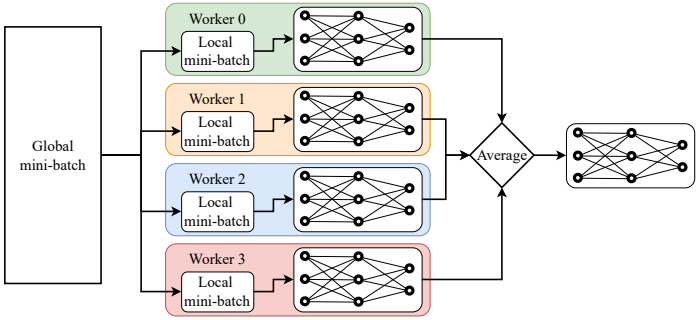


Figure 2.9.: Data parallel training of a neural network on a single mini-batch where the aggregation method is an average. Aggregation of the neural networks can represent either the aggregation of the gradients or the model parameters.

communicates with every other. However, this reduction in per-step communication comes at the cost of introducing additional communication steps. This tradeoff is more optimal for bandwidth utilization than the all-to-all communication strategy. However, the overall impact on training efficiency depends on the specific hardware and network characteristics.

Alleviating this bottleneck is the goal of asynchronous DP methods. These include centralized and decentralized algorithms [61]. A parameter server aggregates gradients or model states in centralized methods to optimize the model. In this architecture, workers operate independently, computing forward-backward passes on their assigned data subsets. They then send the resulting gradients to the parameter server, which is responsible for aggregating these updates and updating the global model parameters accordingly. Before a worker's next forward-backward pass, it will receive the most recent model parameters. A parameter server is a common approach for asynchronous DP training setups [55]. The decoupling of computation and communication in asynchronous parameter server methods allows for greater parallelization and significantly reduces waiting times, improving hardware utilization [62].

However, training with a parameter server is not without its challenges [63]. Since workers operate independently, they may send gradients based on older, stale versions of the model parameters. The staleness of gradients and

model parameters can negatively impact convergence, potentially leading to instability or even divergence in some cases [64].

In decentralized asynchronous methods, workers communicate directly with other workers. This can lead to improved fault tolerance and resilience to communication bottlenecks, as well as potentially faster convergence due to reduced communication overhead [65]. However, ensuring consistent model updates and managing gradient staleness without a central authority is a challenge [66]. Various algorithms, such as gossip-based communication [67] and decentralized optimization [68] techniques, have been proposed to address these challenges, and ongoing research continues to explore the potential of decentralized asynchronous DP training for efficient and scalable large-scale model training.

Despite these challenges, ongoing research into asynchronous methods, show promise in addressing the communication bottlenecks inherent in large-scale distributed training. By carefully managing gradient staleness [62] and leveraging advanced optimization techniques [63], asynchronous DP training and its variants can further improve training efficiency and scalability.

There are several prominent frameworks and libraries used for data-parallel training. Horovod [60], an early open-source solution, gained recognition for its efficient data parallelism implementation and user-friendly interface, leveraging the Ring Allreduce algorithm for inter-GPU communication. PyTorch's recommended data parallelism method is the `DistributedDataParallel` (DDP) module [69]. This module provides a convenient and efficient way to parallelize training across multiple workers. DDP automatically handles gradient synchronization and model parameter updates, simplifying the implementation of synchronous data parallelism. TensorFlow offers the `tf.distribute.Strategy` module, which provides a unified interface for various distributed training strategies, including synchronous and asynchronous data parallelism [70]. With minimal code changes, researchers can scale their TensorFlow models across multiple workers.

2.5.1.1. Large Batch Effects

DP training excels at scalability by distributing data across multiple workers. Therefore, the effective batch size in typical synchronous DP training scales with the number of workers. While larger batches can lead to faster

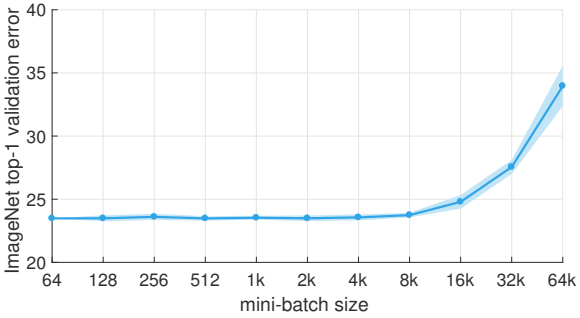


Figure 2.10.: Top-1 validation error for a ResNet-50 model on ImageNet vs the global mini-batch size. Error range is two standard deviations. Maximum learning rate set with a linearly function dependent on the global batch size. A learning rate warmup of five epochs was used. Source: Goyal et al. [31]

per-iteration processing, batches which are too large result in poor generalization [31]. A model trained on very large batches may achieve a low training error but struggle to generalize effectively to unseen data. Figure 2.10 shows how the validation error increases with the batch size. Significantly, where this effect begins depends on the data, model, and optimizer.

This degradation is rooted in the tendency for models trained with large batches to converge towards sharp minima in the loss landscape [71, 72]. These sharp minima often represent solutions that have overfit to the training data, prioritizing the memorization of specific examples over the extraction of broader, generalizable patterns [71, 73]. Conversely, smaller batch sizes inject a degree of stochasticity into the optimization process, encouraging convergence towards flatter minima. Empirically, flatter minima are associated with superior generalization [72, 74].

The adverse impact of large batch sizes has motivated the development of several mitigation strategies:

Learning Rate Warmup: The most common learning rate schedule starts with a large learning rate, which then shrinks during training. In an attempt to avoid sharp minima early in training, a learning rate schedule that starts with a small LR and gradually increases it was proposed [31]. This approach showed moderate success and was able to increase the global batch size of training significantly. The learning rate warmup has since become commonly used when training NNs at all scales.

Layer-wise Adaptive Rate Scaling (LARS): The LARS algorithm [57] adaptively adjusts each layer's LR based on the ratio of weight norms to gradient norms. This adaptive scaling mitigates the impact of large gradients, often prevalent during the initial training phases. While learning rate warmup techniques can also address this issue, layer-wise adaptive learning rates have successfully improved convergence and stability, particularly in smaller-scale settings [75].

The LAMB Optimizer: The LAMB optimizer builds upon LARS by incorporating layer-wise adaptive moments. These moments enhance the optimizer's stability and convergence properties, particularly in large-batch training scenarios [3].

Despite the effectiveness of these mitigation strategies, large batch effects remain an issue. Further theoretical and experimental investigation is needed to show the intricate relationship between batch size, gradient averaging, optimization algorithms, and generalization behavior in DP training at scale.

2.5.1.2. Specialized Data-Parallel Methods

While traditional data parallel training offers straightforward scalability for many scenarios, certain situations require specialized approaches to achieve optimal performance. When individual data samples become too large to fit within the memory of a single accelerator, as can occur in hyperspectral imaging [76], video processing [77], or medical imaging [78], traditional data-parallel methods become infeasible.

Domain-wise data splitting, or domain parallelism, offers an alternative distribution scheme for such scenarios. By partitioning data along its intrinsic dimensions (e.g., separating an RGB image into color channels), this approach enables parallel processing across multiple accelerators. While domain parallelism has shown promise in certain domains [79, 80, 81], it incurs additional computational and communication overhead during backpropagation. Careful optimization is essential to mitigate these overheads and ensure overall training efficiency.

Contemporary optimizers, notably those in the Adam family, maintain per-parameter state tensors that can significantly increase the memory requirements of large-scale training. To mitigate this issue and leverage the parallelism inherent in DP, ZeRO [82] distributes the optimization of model layers

across the workers. Before each forward computation, each worker collects the current model layers from the other workers. The forward and backward computations are performed as is typical for DP training. Then, after the gradients are calculated, each worker calculates the optimization updates for only the subset of layers for which it is responsible. While effective for large-scale training, this approach may be less efficient for smaller-scale deployments due to the increased communication-to-computation ratio [83]. A similar strategy is implemented in PyTorch's `FullyShardedDataParallel` module [84].

2.5.2. Model Parallelism

Model performance scales with simultaneous increases in the number of data elements and model parameters [8]. Data parallelism offers scalability by distributing data across multiple workers. However, in its most basic form it is limited to training models that fit in a single device's memory. Model parallel (MP) training partitions a model across multiple computational resources allowing for the training of models with massive parameter counts. Model parallel training encompasses two primary paradigms: tensor parallelism and pipeline parallelism [55].

Tensor parallelism, illustrated in Figure 2.11, involves partitioning the tensors and operations within an individual layer across multiple computational devices. This approach is particularly advantageous for the large-scale matrix operations prevalent in transformer-based architectures and CNNs, as specific components of these operations, i.e., the attention heads [85] and convolutional channels or kernels [86], can be parallelized without incurring a sizeable inter-device communication overhead [87, 88]. Specialized libraries and communication protocols can further enhance the efficiency of tensor parallelism by optimizing the distribution and synchronization of tensor fragments.

As depicted in Figure 2.12, pipeline parallelism partitions a model into sequential stages, each executed by a separate worker. Input data flows sequentially, necessitating communication only of activations between stages. This allows for very large internal network states, which are beneficial for massive inputs or layers. However, varying computational demands across layers can lead to worker load imbalances. Additionally, forward passes create dependencies where downstream workers await upstream completion, while backward

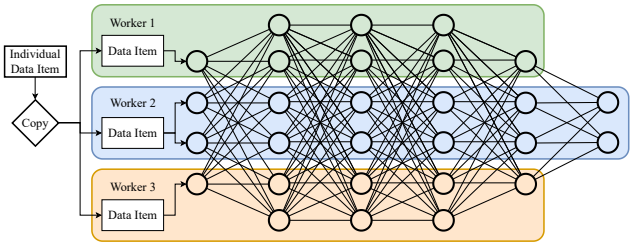


Figure 2.11.: Tensor Parallelism: Horizontal partitioning of a neural network across three devices. Each device processes a distinct portion of the tensor operations within each layer.

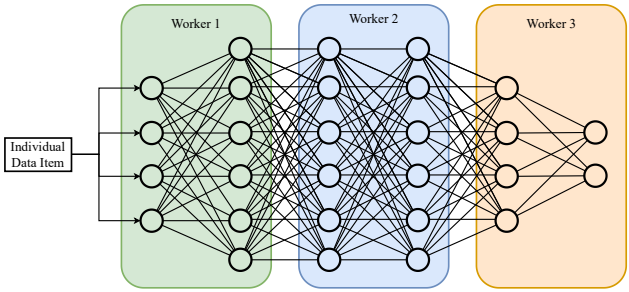


Figure 2.12.: Pipeline Parallelism: Vertical partitioning of a neural network across three devices. Each device processes a sequential segment of the model. Data flows from left to right, with intermediate activations and gradients exchanged between devices during forward and backward passes.

passes see upstream workers dependent on downstream stages. These dependencies result in "bubbles" of worker idle time, impacting efficiency. While pipeline parallelism enables the training of massive models, load imbalances and bubble overheads require careful management and optimization [89].

A significant challenge in model parallelism lies in determining an effective model partitioning strategy. The goal is to minimize communication overhead while ensuring a balanced computational load across workers. Striking this balance can be a non-trivial optimization problem, particularly for complex architectures with varying computational demands across layers. Furthermore, ensuring that the partitioned model retains its functionality and accuracy can require careful consideration and adaptation of the training algorithm.

Despite these complexities, various large-scale training scenarios have successfully deployed model parallelism [90, 89, 86, 88]. Pipelining implementations, in particular, have shown promising results by dividing the model into stages and allowing concurrent execution of different mini-batches across these stages, thus improving throughput and reducing training time [90]. However, unlike data parallel training, which is often amenable to automation, model parallelism typically requires tailored solutions based on the specific network architecture and underlying system configurations [91]. This lack of strong automation can present challenges in terms of development and deployment. Nonetheless, specialized libraries and frameworks such as Megatron-LM [87] and Mesh-TensorFlow [92] provide tools and abstractions to facilitate model parallel implementations, easing the burden of manually partitioning and coordinating complex model components. In practice, hybrid parallelism, combining aspects of both data and model parallelism, often provides the most effective solution for training extremely large neural networks [55].

2.6. Hyperparamter Optimization

Hyperparameters (HPs) are non-learnable configuration settings set before training begins. HPs encompass a wide variety of settings, including the learning rate, batch size, weight decay coefficients, choice of optimization algorithm, degree of data augmentation, and neural network's architecture. In contrast to the model's internal weights and biases, which are learned directly from the training data, hyperparameters must be set a priori. Identifying optimal HP configurations is critical, given their substantial impact on final model performance.

hyperparameter optimization (HPO) aims to discover the optimal configuration of HPs that minimize an objective function. In the context of NN tuning, this objective function is often the validation loss or other validation metrics such as the prediction accuracy. The standard HPO cycle, as depicted in Figure 2.13, involves the iterative training and evaluation of numerous networks across a diverse range of HP combinations to assess their efficacy. However, HPO remains computationally demanding and intricate, frequently necessitating substantial computational resources and sophisticated search strategies.

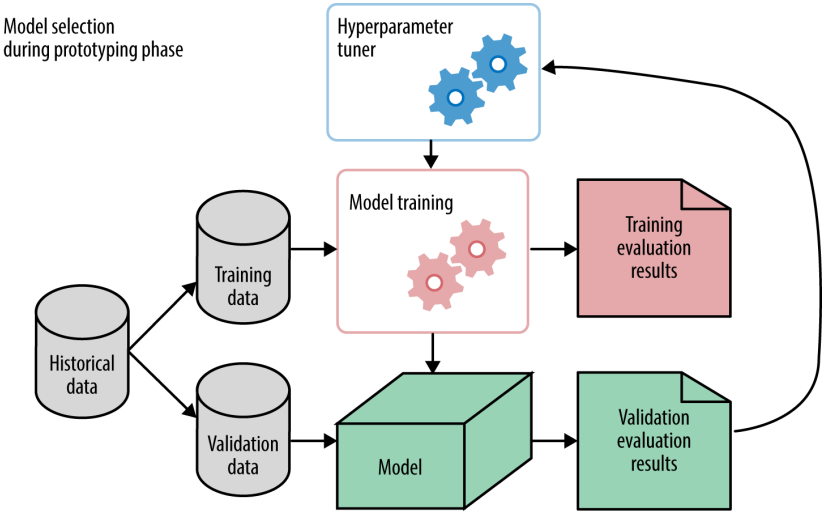


Figure 2.13.: Example workflow for tuning hyperparameters. Source: Avhale [93]

The challenge of HPO stems from several factors. First, hyperparameter spaces are often high-dimensional and non-convex, meaning there can be nonlinear dependencies between the parameters, and local minima may not represent the best possible performance [94]. Second, although each HP combination can be trained independently, training and evaluating each combination is computationally expensive [95]. Finally, theoretical guidance for selecting optimal hyperparameters is limited, as their effects can vary significantly depending on the specific network architecture, dataset, and optimization method employed [96].

A diverse array of methodologies has emerged to address the challenges inherent in hyperparameter optimization. Grid search, a technique that systematically explores the possible combinations of HP values within a user-defined grid, offers transparency and straightforward implementation. However, its computational cost scales exponentially with the dimensionality of the hyperparameter space, limiting its applicability to high-dimensional problems [97]. Random search tests random HP combinations and provides a more computationally efficient alternative, often delivering surprisingly good performance across various scenarios. [98].

Bayesian optimization, a more sophisticated technique, constructs a probabilistic model of the objective function, most typically model performance, to guide the selection of promising hyperparameter configurations [99]. By leveraging prior results and incorporating uncertainty, Bayesian optimization can be more sample-efficient than random search, particularly when evaluations are expensive [97].

Evolutionary optimization [100], particularly genetic algorithms, represents another promising approach for hyperparameter optimization. Rooted in the principles of natural selection, genetic algorithms simulate a process wherein a "population" of candidate hyperparameter configurations is iteratively refined. This refinement is achieved through a cycle of selection (favoring high-performing configurations), crossover (combining aspects of different configurations), and mutation (introducing random variations). The inherent diversity maintained by genetic algorithms through mutation can be advantageous in practical scenarios, and they have outperformed Bayesian optimization methods [101].

Practitioners often combine these automated search methods with manual tuning based on domain knowledge and experience. This hybrid approach leverages the strengths of algorithmic exploration and human intuition to navigate the complex landscape of hyperparameter optimization.

2.6.1. Neural Architecture Search

HPO typically focuses on optimizing the hyperparameters of a fixed model architecture to simplify the already complex search space. neural architecture search (NAS), a subfield of HPO, expands this optimization process to include the model architecture. This expansion significantly increases the search space's complexity due to the vast number of possible architectures and the intricate interactions between the architectural choices and the training hyperparameters.

Due to the dependencies of optimization HPs on network architectures, NAS algorithms must navigate this complex landscape efficiently, often employing specialized techniques to explore and evaluate different architectures. In addition, since new architectures often require tailored hyperparameters, effective NAS is an integral part of the development process to realize the full potential of novel architectures.

2.7. Neural Network Compression

The increased computation and memory requirements of modern NNs has spurred the development of numerous compression techniques, broadly categorized as:

Pruning: Methods that focus on removing weights or neurons within a model.

Quantization: Methods that aim to reduce the model size or the other tensors used during training by representing network weights and/or other tensors with lower bit precision.

Knowledge Distillation: Approaches involving training a smaller "student" model to mimic a larger, pre-trained "teacher" model, thereby transferring knowledge while reducing complexity.

Efficient Architecture Design/Learning: Methods that either design specific model architectures for specific tasks or methods that dynamically learn a compact and efficient model architecture during training.

Low-Rank Approximation: Techniques that involve replacing the full-rank weight tensors with lower-rank approximations.

Model compression techniques frequently come with a trade-off in predictive performance [102]. However, when preserving full model accuracy is critical, retraining the compressed model on the original dataset can often mitigate or eliminate any performance degradation arising from the compression process.

2.7.1. Pruning

Pruning is a widely adopted technique for reducing the size of neural networks by selectively removing weights. It capitalizes on the over-parameterization of large networks, where certain weights or neurons contribute minimally to the overall performance. Pruning can be broadly classified into structured and unstructured methods [102].

Structured pruning removes specific structures within the network, such as entire convolutional filters, channels, or even layers. In the context of

CNNs, studies have revealed that many filters or channels exhibit low activation magnitudes or minimal impact on the final output [103]. Removing these redundant structures reduces the model size and directly translates to fewer computations during inference, leading to tangible speed improvements [104].

Unstructured pruning, in contrast, removes individual weights throughout the network, regardless of their structural organization. This approach often results in sparse weight matrices, where many elements are to zero. The Lottery Ticket Hypothesis, which posits that dense, randomly initialized networks contain sparse subnetworks that can achieve comparable performance, has fueled interest in unstructured pruning [105].

However, unstructured pruning introduces challenges in achieving computational efficiency. While reducing the number of active parameters, the resulting sparsity patterns are often irregular, making it challenging to leverage optimized sparse matrix operations [106]. When sparse matrix operations cannot be utilized due to the insufficient implementation of sparse functions or incompatible sparsity, the inactive parameters are represented with zeros in the model, and traditional dense operations are used.

Several other pruning methods have been proposed, each with different strategies for identifying and removing redundant components. Movement pruning eliminates weights that have exhibited minimal change during training, suggesting they are less relevant for learning [107]. Sensitivity-based pruning analyzes the sensitivity of the loss function to weight perturbations and removes weights with the lowest impact [108].

The effectiveness of pruning often depends on the pruning schedule and the specific criterion used for selecting elements to remove. Iterative pruning, where weights are removed iteratively during training, can lead to better results than one-shot pruning [102]. Additionally, retraining the complete model architecture can help recover any lost accuracy by allowing the remaining weights to compensate for the removed connections.

2.7.2. Quantization

Quantization relies on the observation that neural networks often exhibit resilience to reduced numerical precision [109]. By representing weights and activations with lower bit widths than the standard 32-bit floating-point

format (FP32), quantization can significantly reduce a network's memory footprint and computational requirements without sacrificing predictive performance [110].

Traditionally, neural networks have been trained using FP32 (single precision) or 16-bit floating-point format (FP16) (half-precision) arithmetic. However, recent advances have pushed the boundaries further. Brain Floating Point (BF16) is a popular 16-bit format for training neural networks that allocates more bits to the mantissa instead of the exponent (as is done for FP16), as network weights are not expected to be high values. This bit allocation grants it a wider dynamic range than traditional FP16. Moreover, mixed precision training uses both FP32 and lower precision formats during training [111]. This maximizes computational throughput on hardware like GPUs, which frequently have specialized cores for calculations using reduced bit-lengths while maintaining model accuracy.

Intriguingly, research has demonstrated that networks can be quantized to even lower precisions, such as 8-bit integers, or even extreme cases like binary (1-bit) or ternary (3-bit) representations, with surprisingly little degradation in performance on specific tasks [112]. These low-bit networks offer substantial reductions in memory and computation.

Research has also explored quantizing gradients during training to reduce the network traffic during distributed training. However, the discretization of the gradients can adversely affect convergence [113]. Sophisticated techniques like error feedback and gradient scaling have been proposed to mitigate this noise and maintain training stability [114, 115].

2.7.3. Knowledge Distillation

Knowledge distillation [116] is a model compression technique where a smaller "student" model is trained to replicate the behavior of a larger, pre-trained "teacher" model. The student model typically has significantly fewer parameters than the teacher model, making it more computationally efficient and more accessible to deploy. The teacher model's knowledge is transferred to the student using the teacher's output probabilities or intermediate representations as soft targets during training. This allows the student to learn from the teacher's expertise while employing a more compact architecture. The student model can sometimes surpass the teacher's performance due to

improved generalization capabilities [117]. Knowledge distillation has found widespread application in various domains, particularly in compressing large language models for faster and more efficient deployment [118].

2.7.4. Efficient Architecture Design or Learning

Efficient architecture design and learning represent multifaceted approaches to neural network compression. They focus on either crafting a more efficient model before training, or discovering model structures that are inherently more efficient in terms of computation and parameter count during training. These strategies are driven by the recognition that model architecture is pivotal in determining a model's computational demands and overall efficiency [119].

Manual design of efficient architectures often involves carefully selecting layer types, utilizing specialized components or incorporating sparsity-inducing techniques. For instance, MobileNet models [120] leverage depth-wise separable convolutions to drastically reduce parameter count and computational complexity while maintaining competitive accuracy on image classification tasks.

Sparse training aims to train efficient neural networks in a sparse paradigm. This inherent sparsity can substantially reduce memory usage and computational overhead, especially during inference. However, effectively training sparse networks requires specialized techniques to ensure convergence and avoid performance degradation [121].

Rigging the Lottery [119] is a prominent sparse training algorithm that employs a dynamic pruning and growth strategy. It begins with a randomly initialized sparse network and iteratively removes a fraction of the weakest connections while simultaneously growing new connections based on their gradient magnitudes. This allows the network to explore the space of sparse architectures during training, potentially leading to highly efficient and performant solutions. However, the training of these models take much longer than traditional training methods.

Efficient architecture design and learning represent a promising frontier for neural network compression. By tailoring model structures to specific tasks and resource constraints or dynamically adapting architectures during

training, the computational costs and memory footprints can substantially reduce without sacrificing model performance [121].

2.7.5. Low-Rank Approximation

Low-rank approximations play a pivotal role throughout this thesis. To ensure a solid foundation for subsequent analyses, this section will delve deeper into their properties and applications. I will consider both tensors, representing multidimensional data, and matrices, their two-dimensional counterparts.

A tensor is considered low-rank if it can be accurately approximated by a sum of a small number of rank-one tensors (tensors formed by the outer product of vectors). Intuitively, low-rank approximation techniques seek to identify and isolate the most informative components within a tensor, effectively compressing its essential information content into a more compact representation. Mathematically, the low-rank approximation for a real matrix $\boldsymbol{A}$ of shape $m \times n$ seeks to minimize the Frobenius norm of the difference between the original matrix and its low-rank approximation:

$$\min ||\boldsymbol{A} - \boldsymbol{BC}||_F^2 \tag{2.19}$$

where $\boldsymbol{B}$ and $\boldsymbol{C}$ are real matrices of shapes $m \times k$ and $k \times n$, respectively, and $k < \min(m, n)$.

Tensor decomposition methods provide a principled framework for determining such low-rank approximations. Among these, the matrix decomposition method singular value decomposition (SVD) is a fundamental technique with widespread applications. For a matrix $\boldsymbol{A}$, SVD yields a factorization:

$$\boldsymbol{A}_{m \times n} = \boldsymbol{U}_{m \times k} \boldsymbol{\Sigma}_{k \times k} \boldsymbol{V}_{k \times n}^{\top} \tag{2.20}$$

where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthonormal matrices containing the left and right singular vectors of $\boldsymbol{A}$, respectively, and $\boldsymbol{\Sigma}$ is a diagonal matrix containing the singular values. The singular values reflect the importance of the corresponding singular vectors when reconstructing the original matrix. The combination of the left singular vectors and the singular values are sometimes referred to as the principle components.

A low-rank approximation of $\boldsymbol{A}$ is obtained by retaining only the top-k singular values and associated singular vectors. This approximation captures the

most significant information while reducing the number of parameters. However, it is essential to note that if $k > m \cdot n/(1+m+n)$, the SVD representation requires more storage space than the original full-rank matrix.

Low-rank approximation methods are gaining prominence in neural network compression because they can significantly reduce the model size and computational overhead. The central idea is to replace full-rank weight matrices within a neural network with their lower-rank approximations, thus reducing the parameter count without altering the network's architecture. This not only leads to more compact models but can also accelerate computations [122]. Additionally, low-rank approximations have been observed to enhance generalization performance by acting as a form of regularization [123]. However, these benefits often come at the cost of increased computational complexity during training, potential accuracy degradation, or both [102].

While SVD is well-suited for matrices, deep learning models often employ higher-order tensors, such as weight tensors in convolutional layers. Many methods simply collapse the weight tensors to form a two dimensional representation. To extend low-rank approximations to higher-order tensors, techniques like Canonical Polyadic (CP) decomposition (expressing a tensor as a sum of rank-one tensors) [124] and Tucker decomposition (representing a tensor as a core tensor multiplied by factor matrices) [125] can be employed [126, 125]. Each of these decompositions, as well as collapsing dimensions, presents distinct trade-offs in expressiveness, computational complexity, and interpretability, necessitating careful selection based on the specific application.

The naive approach of directly replacing weight matrices with low-rank counterparts in pre-trained networks often leads to significant accuracy degradation [127]. Therefore, low-rank neural networks must be trained directly in their compressed form to maintain performance. However, this poses challenges due to the optimization difficulties and potential for performance degradation arising from the constraints imposed by low-rank representations [128].

Ongoing research seeks to address these challenges through various strategies. These include methods for selecting appropriate inner ranks for network layers during training, ranging from empirical heuristics to adaptive approaches that adjust ranks based on validation performance or other criteria [102]. Additionally, specialized optimization algorithms have been developed to handle the specific optimization constraints of low-rank representations, ensuring

stable and effective training [129]. Careful initialization of low-rank factors also plays a crucial role in achieving good convergence and performance. Despite these advancements, current low-rank training methods continue to grapple with the trade-offs between compression, computational efficiency, and model accuracy.

3. Data Parallel Training Informed by Network Topology

The content of this chapter is based on:

D. Coquelin, C. Debus, M. Götz, et al. "Accelerating neural network training with distributed asynchronous and selective optimization (DASO)". en. In: Journal of Big Data 9.1 (Feb. 2022), p. 14. ISSN: 2196-1115. DOI: 10.1186/s40537-021-00556-1.

Modern neural networks have consistently demonstrated superior performance as their complexity and size increase [4, 20, 8]. Data parallel (DP) training has emerged as a dominant strategy to effectively train such large models on massive datasets, distributing data across multiple GPUs to accelerate computation. However, this scaling approach inherently relies on large batch sizes, which can hinder generalization performance [31, 71], as discussed in Section 2.5.1.1.

The communication overhead associated with the frequent synchronization steps required in synchronous DP training further compounds this challenge. As discussed in Section 2.5, this overhead can become a significant bottleneck, particularly when scaling to large numbers of GPUs or for clusters without a high-bandwith, high-speed interconnect.

In this chapter, I introduce Distributed, Asynchronous, and Selective Optimization (DASO), a novel algorithm designed to mitigate this communication bottleneck and accelerate data parallel neural network (DPNN) training. DASO leverages the hierarchical structure of modern computer clusters, where multiple GPUs are grouped within compute nodes, and employs asynchronous gradient updates to reduce communication overhead.

The key contributions of this chapter are:

- A detailed presentation of the DASO algorithm, highlighting its hierarchical communication strategy, asynchronous updates with stale gradients, and the flexibility to synchronize parameters after multiple batches.
- A thorough parameter study of DASO, providing insights into the optimal configuration of its key hyperparameters.
- A theoretical analysis of DASO proving convergence.
- An extensive performance evaluation of DASO compared to Horovod [60] and a traditional synchronous DPNN approach on image classification and semantic segmentation tasks.

DASO aims to unlock significant speedup in distributed training scenarios by reducing the time-consuming global synchronization step, utilizing stale gradients, and adapting a communication strategy inspired by the underlying cluster topology. DASO represents a promising direction towards more efficient and scalable training of large-scale neural networks.

3.1. Related Work

When I conducted this research, state-of-the-art approaches included optimizing the communication patterns for a specific architecture [55]. One notable example achieved training times of only 74.7 seconds on the ImageNet dataset [130]. However, such methods were often highly architecture-specific and did not generalize well to different network structures [55].

Asynchronous stochastic gradient descent (ASGD) [131, 132] offered an alternative approach where individual workers updated parameters without waiting for global synchronization, potentially improving hardware utilization. However, this introduced the challenge of using stale gradients (gradients for old model states), which could hinder convergence. Several variants of ASGD attempted to leverage stale gradients effectively [133] or accelerate training with delayed parameter server updates [134].

Hierarchical algorithms, which leverage the structure of compute clusters, were also explored to maximize resource utilization. Methods like Local SGD, post-local SGD, and hierarchical SGD [135] introduced local and global update steps but still relied on periodic global synchronization.

Several approaches focused on optimizing the global parameter synchronization operation within MPI-based frameworks, exploring different network topologies [136, 137]. However, these methods remained constrained by the need for global synchronization after each forward-backward pass. Horovod [60] emerged as a leading MPI-enabled DPNN framework. It employed techniques such as tensor fusion and data compression to reduce communication overhead. While effective, its communication strategies were primarily designed for synchronous updates.

Since this research was conducted, the field of distributed training has continued to advance rapidly. Modern approaches have explored techniques like gradient compression [114] and bucketed communication to further mitigate communication bottlenecks and improve scalability.

Current state-of-the-art DP training methods predominantly rely on the high-speed interconnects available in high-end computing clusters and employ a strategy of bucketing gradients, where communication of the gradients begins eagerly during the backward pass as soon as a bucket is full. This strategy has proven effective when high-bandwidth, low-latency communication channels are available but falters in environments with limited network capacity. Furthermore, this approach may be unsuitable when gradient information could potentially expose sensitive data, raising privacy concerns [138].

In scenarios where network constraints or privacy requirements are paramount, recent developments have focused on utilizing stale gradients or model states to train performant models while minimizing communication overhead [139]. These methods leverage the inherent robustness of neural networks to imperfect or delayed updates [140], demonstrating that significant progress can still be made even with less frequent or less precise communication.

3.2. Distributed Asynchronous and Selective Optimization (DASO)

The standard synchronous approach to DPNN training involves performing a forward-backward pass on each worker with its assigned portion of the distributed batch, followed by a global averaging operation to synchronize model gradients (see Section 2.5). This averaging step approximates the true gradient computed over the entire dataset, assuming that each portion of the

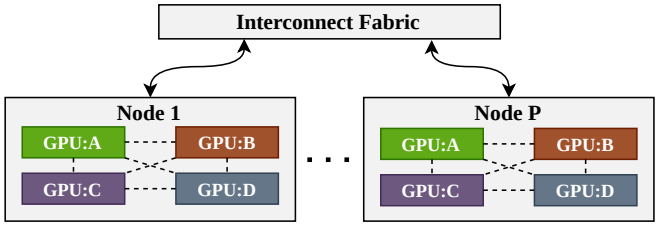


Figure 3.1.: An overview of a common node-based computer cluster with P nodes and four GPUs per node. GPU colors represent communication *group* membership. The dashed lines indicate GPU-to-GPU communication channels.

distributed batch is iid [141]. The traditional update function for distributed SGD is given by:

$$w_{t+1} = w_t - \frac{\eta}{P} \sum_{i=1}^{P} w_i \tag{3.1}$$

where w_{t+1} is a weight for batch $t + 1$, η is the learning rate, and P is the number of processes.

Under iid, I make a further approximation: the average gradients for a subset of the global minibatch do not significantly differ from those of the complete minibatch, given that both sets are not dominated by the gradients for a single data element [142]. This approximation, coupled with the observation that modern compute clusters often have heterogeneous inter-node and intra-node communication capabilities, provides an opportunity to reduce communication overhead and alleviate the inherent bottleneck of synchronous data-parallel training.

Based on these principles, I propose the Distributed, Asynchronous, and Selective Optimization (DASO) method. DASO departs from the traditional uniform communication model and employs a hierarchical network model with two distinct levels: node-local and global networks.

The global network spans all GPUs across all nodes, while the node-local networks comprise the GPUs within each individual node. The dotted lines show the node-local communication paths in Figure 3.1. The global network is further divided into multiple communication groups, each containing a single GPU from every node, i.e., GPU 'A' on node 1 is in a group with the GPU 'A' on all other nodes.

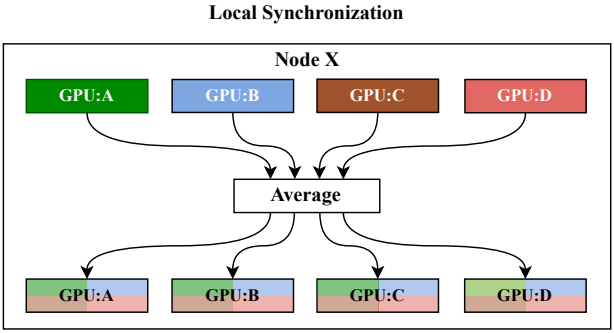


Figure 3.2.: Schematic of the local synchronization step for a single node with four GPUs. The gradients from each GPU are averaged and each GPU's gradients are set to the result.

DASO utilizes a multi-step synchronization process. Local synchronization, shown in Figure 3.2, occurs after each backward pass, averaging the gradients within the node-local group. Global synchronization, shown in Figure 3.3, occurs after one or more local synchronizations and involves averaging network parameters among members of a single global group, i.e., all GPU 'A's. This reduces the inter-node communication traffic by a factor equal to the minimum number of GPUs per node. Following this global step, a local update, shown in Figure 3.4, broadcasts the averaged parameters from the group member used for global synchronization to all other local group members.

Global synchronization can be performed in either a blocking or non-blocking fashion. In the former, training is paused during the communication steps, whereas the latter allows training and communication to occur concurrently. To mitigate the communication overhead associated with global synchronization, parameters are quantized to a 16-bit floating-point format prior to transmission, a technique demonstrated to maintain model accuracy while significantly reducing communication volume [115].

The asynchronous nature of non-blocking communication introduces stale gradients (updates computed using older parameter values). DASO compensates for this by employing a weighted average of the stale global parameters (w_t) and current local parameters after $S-1$ steps (w_{t+S-1}):

$$w_{t+S} = \frac{2Sw^l_{t+S-1} + \sum_{i=1}^{P} w^i_t}{2S+P} \tag{3.2}$$

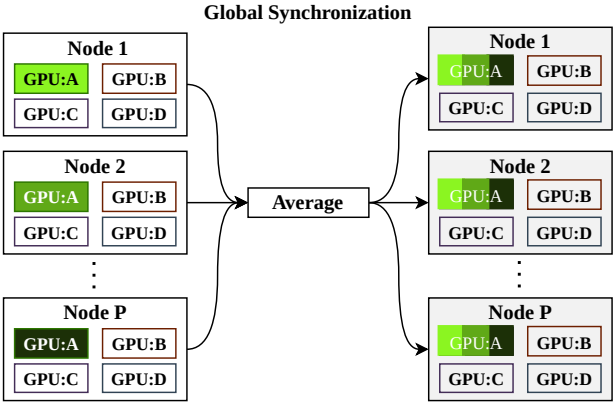


Figure 3.3.: Schematic of the global synchronization step performed by the global communication *group* consisting of GPU:A on each node. The network parameters are averaged by each GPU in the *group*, and the network parameters of each *group* member are set to the result.

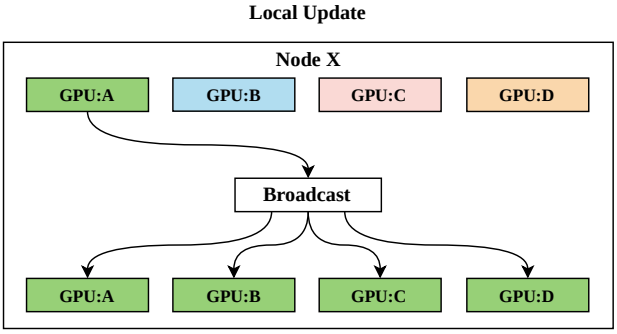


Figure 3.4.: Schematic of the local update step to be performed after the global synchronization step shown in Figure 3.3. The group member responsible for the global communication, in this case GPU:A, sends its network parameters to all other node-local GPUs, which replace the old parameters on those GPUs.

where S is the number of training steps between the sending and receiving of the global network state, t is a total number of training steps, w^l_{t+S} is a weight on GPU l after step $t + S$, and P is the number of GPUs in the global network. A detailed analysis of this update rule is provided in the following section.

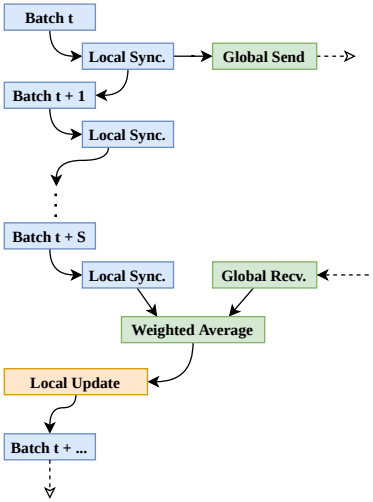


Figure 3.5.: Cycling Flow Process flow diagram of the synchronization steps during the cycling phase where t is the batch number and S is the batches to wait before global synchronization. The weighted average is calculated as shown in Equation (3.2)

DASO training comprises three phases: warmup, cycling, and cooldown. The warmup and cooldown phases employ blocking global synchronizations to ensure initial stability and final convergence, respectively. The cycling phase, shown in Figure 3.5, utilizes non-blocking synchronization, with the number of training steps between global synchronizations (B) and the number of training steps to wait for global updates (S) dynamically adjusted to optimize training progress.

3.2.1. Theoretical Analysis of DASO

The following proof of DASO's global synchronization method is heavily based on the convergence analysis shown by Bottou, Curtis, and Nocedal [143] and will demonstrate that the gradients determined with DASO are bounded.

Proof. Let $X \subset \mathbb{R}^n$ be a known set, and $f : X \to \mathbb{R}$ a differentiable, convex, L-smooth, and unknown function. The stochastic gradient estimator of $f(x)$

is a function $\tilde{g}(x)$ for inputs x determined by a random variable ζ, such that $\mathbb{E}[\tilde{g}(x;\zeta)] = \nabla f(x:\zeta)$. For brevity, I will omit the explicit dependence on ζ moving forward.

In its most basic form, SGD updates a model's state at step $t+1$ as $w_{t+1} = w_t - \eta\tilde{g}(x_t)$, where η is the learning rate and w is the set of network parameters being optimized.

In practice, minibatch SGD is often used for computational efficiency reasons. In minibatch SGD, the true stochastic gradient is approximated by averaging across m input items x_i, i.e. $\tilde{G}(x_t) = \frac{1}{m}\sum_{i=1}^{m}\tilde{g}(x_{t,i})$. The set of network parameters being optimized w_{t+1} for minibatch SGD is then

$$w_{t+1} = w_t - \eta\tilde{G}(x_t) \tag{3.3}$$

where $\tilde{G}(x_t)$ is the estimator of $\nabla f(x_t)$.

Let us now consider, that S subsequent training steps are performed between an update to the model's weights. With this, a weight can be expressed as:

$$w_{t+S} = w_t - \eta\sum_{i=0}^{S-1}\tilde{G}(x_{t+i}) \tag{3.4}$$

One of the primary assumptions in SGD is that the objective function has Lipschitz-continuous gradients, leading to:

$$f(x_{t+1}) - f(x_t) \leq -\eta\nabla f(x_t)^T\,\mathbb{E}[\tilde{g}(x_t)] + \frac{1}{2}\eta^2 L\mathbb{E}\left[\|\tilde{g}(x_t)\|_2^2\right] \tag{3.5}$$

where the Lipschitz constant, L, is greater than zero. Equation (3.5) implies that the expected decrease in the objective function, $f(x)$, is bounded above, irrespective of how the stochastic gradients arrived at x_t [143].

In DASO, the local synchronization step is bound via the same assumptions as minibatch SGD, so long as the iid assumption is upheld. However, the non-standard global synchronization step in DASO, which utilizes both local and global model states, requires further analysis.

DASO's global synchronization is defined as:

$$w_{t+S}^{\text{DASO}} = \frac{2Sw_{l:t+S-1} + \sum_{i=1}^{P} w_{p:t}^{i}}{2S+P} \tag{3.6}$$

where the l and p subscripts represent the node-local and global model states, S is the number of local update steps before global synchronization, and P is the number of processes.

Similar to Equation (3.3), this can also be represented via the locally and globally calculated gradients, $\tilde{G}_l\left(x_{l:t}\right)$ and $\tilde{G}_p\left(x_{p:t}\right)$ respectively. The global synchronization function in the gradient representation is as follows:

$$w_{t+S}^{\text{DASO}} = w_t - \alpha\left(2S\sum_{k=0}^{S-1}\tilde{G}_l\left(x_{l:t+k}\right) + \sum_{i=1}^{P}\tilde{G}_p\left(x_{p:t}^i\right)\right) \tag{3.7}$$

where $\alpha = \eta/(2S+P)$. By utilizing this expression, the standard SGD update rule, and the fact that the updates between t and S are local synchronizations which take the form of repeated minibatch updates, we can derive the globally calculated gradients to be:

$$\tilde{G}^{\text{DASO}}\left(x_{t+S-1}\right) = P\sum_{\beta=0}^{S-1}\tilde{G}_l\left(x_{l:t+S-\beta}\right) - 2S\tilde{G}_l\left(x_{l:t+S-1}\right) - \sum_{i=1}^{P}\tilde{G}_p\left(x_{p:t}^i\right) \tag{3.8}$$

As all gradient elements in this expression are bound under the Lipschitz continuity assumption, it follows that $\tilde{G}^{\text{DASO}}\left(x_{t+S-1}\right)$ is similarly bounded. □

3.2.2. Implementation

A proof-of-concept implementation of the DASO algorithm has been developed within the versatile Helmholtz Analytics Framework (Heat) framework [144]. Heat, an open-source Python library, is designed for distributed and GPU-accelerated data analytics. Heat provides both low-level array operations and a suite of higher-level machine learning algorithms, making it a suitable foundation for prototyping and evaluating the DASO method.

My implementation leverages the strengths of both Heat and PyTorch [50]. For local, intra-node communication, I utilize PyTorch's DDP module and Pytorch's distributed package, which provide efficient mechanisms for synchronizing model parameters and gradients within a single machine. Heat's MPI backend is employed for inter-node communication. This backend automatically handles the communication of PyTorch tensors, simplifying the

Listing 3.1: Simplified training script demonstrating the usage of DASO in HeAT for a PyTorch neural network (`net`) and PyTorch optimizer (`optimizer`).

```
import heat as ht
import torch
...
# create PyTorch distributed group
rank, world_size = ht.MPI_WORLD.rank, ht.MPI_WORLD.size
local_rank = rank % num_local_gpus
torch.distributed.init_process_group(
    backend="nccl",
    rank=local_rank,
    world_size=world_size
)
...
# the DASO optimizer is created
daso_optimizer = ht.optim.DASO(
    local_optimizer=optimizer,
    total_epochs=num_epochs
)
...
# the hierarchical network is created
ht_model = ht.nn.DataParallelMultiGPU(net, daso_optimizer)
```

exchange of model parameters and gradients between nodes in a distributed cluster.

To accommodate DASO's hierarchical communication structure, MPI groups are created to represent the local and global communication groups. As mentioned, the global groups consist of one GPU from every node in the cluster, facilitating efficient inter-node communication (Figure 3.3).

The implementation was designed to be minimally intrusive to existing PyTorch codebases. The data loaders must be adapted to distribute the dataset across all available GPUs. This typically involves specifying the number of GPUs and the global rank of each GPU. Four additional function calls are required to initialize and manage the DASO training process (as illustrated in Listing 3.1). These calls create the node-local groups, instantiate the DASO object with a PyTorch optimizer, and specify the number of training epochs. This straightforward integration allows researchers and practitioners to readily experiment with DASO and assess its effectiveness in accelerating the training of their existing PyTorch models in distributed environments.

3.3. Experimental Evaluation and Discussion

All experiments were conducted on the JUWELS Booster supercomputer at the Jülich Supercomputing Centre [145]. This high-performance cluster comprises 936 GPU nodes, each equipped with two AMD EPYC Rome CPUs and four NVIDIA A100 GPUs [146], interconnected via a high-speed NVIDIA Mellanox HDR InfiniBand fabric. The software stack utilized for these experiments included CUDA 11.0, Python 3.8.5, ParaStationMPI 5.4.7-1-mt, Horovod 0.21.1, PyTorch 1.7.1+cu110, and NCCL 2.8.3-1. ParaStationMPI 5.4.7-1-mt has a CUDA-aware MPI implementation, enabling direct GPU-to-GPU communication for efficient distributed training.

3.3.1. DASO Hyperparameter Study

To investigate the effects of delayed global synchronization and the resulting use of stale gradients on both accuracy and training time, I conducted a comprehensive hyperparameter (HP) study. Specifically, I focused on parameters B, the number of forward-backward passes between global synchronizations, and S, the number of batches between the sending and receiving of global parameters. For this study, I employed the ImageNet-2012 dataset [147] to train a ResNet-50 neural network [4] across two different scales: 32 GPUs (8 nodes) and 128 GPUs (32 nodes). It is important to note that for this study, B and S were held constant throughout each experimental run.

The ImageNet-2012 dataset comprises 1.2 million labeled images. I evaluated classification quality using the accuracy with which the model correctly predicts the image labels in a single attempt, commonly known as the top-1 accuracy. I utilized the NVIDIA Data Loading Library (DALI)[148] for data loading and preprocessing. The specific training HPs are adapted from the work of Goyal et al. [31] for ResNet-50 on ImageNet. The results of the HP study are shown in Table 3.1.

Examining the measurements with $S = 0$ (no staleness in global parameters), a clear anti-correlation between top-1 accuracy and the number of training steps between global synchronizations (B) is evident for both node configurations. This effect is more pronounced for the larger-scale experiments with 128 GPUs. Considering that the expected accuracy of ResNet-50 on ImageNet is around 76%, the results demonstrate a negligible loss of accuracy when

Table 3.1.: Parameter study results. B is the number of forward-backward passes between global synchronizations and S is the number of batches to wait for the global synchronization data.

		32 GPUs (8 nodes)		**128 GPUs** (32 nodes)	
B	S	Runtime, h	Val. Top-1, %	Runtime, h	Val. Top-1, %
1	0	4.56	76.77	1.21	76.54
1	1	4.25	76.09	1.16	74.92
2	0	4.04	76.88	1.08	76.30
2	1	3.89	75.81	1.04	74.89
2	2	3.89	75.92	1.05	75.09
4	0	3.70	76.43	0.98	74.35
4	1	3.66	75.83	1.01	73.30
4	2	3.72	75.50	0.98	71.96
4	4	3.71	75.71	0.98	73.86
8	0	3.49	75.26	0.91	69.27
8	4	3.53	74.61	0.92	65.47
8	8	3.58	75.26	0.93	69.67
16	0	3.32	73.13	0.86	58.54
16	4	3.34	73.18	0.86	56.89
16	8	3.39	73.21	0.87	54.53
16	16	3.48	74.21	0.89	62.37
32	0	3.22	70.75	0.82	43.69
32	4	3.23	70.28	0.82	44.06
32	16	3.30	69.58	0.84	41.25
32	32	3.41	72.55	0.87	50.95

$B \leq 4$ while offering substantial reductions in training time. As B increases beyond this point, the training time decreases while the classification accuracy declines.

The effect of stale gradients is observed when fixing B and varying S. As S increases (introducing more staleness), the accuracy initially decreases, aligning with the expectation that stale gradients can negatively impact convergence [149]. However, an intriguing observation emerges: when B equals S, the accuracy tends to rebound, sometimes surpassing the accuracy

achieved with $S = 0$. This suggests a potential regularization effect of stale gradients, a phenomenon that warrants further investigation.

3.3.2. Performance Evaluation

To assess the computational performance of DASO, I conducted experiments on two challenging and data-intensive deep learning tasks: image classification and semantic segmentation. For image classification, I chose the established benchmark of training a ResNet-50 architecture [4] on the ImageNet-2012 dataset [147]. This task is valuable due to the widespread use of pre-trained ResNet-50 models as backbones in numerous computer vision pipelines [150]. For semantic segmentation, I trained a state-of-the-art hierarchical multi-scale attention network [151] on the Cityscapes dataset [152], representing a challenging real-world scenario.

I compared DASO against two established baselines: Horovod [60] and a "classic" synchronous data-parallel implementation without compression or tensor fusion. This classic approach was also implemented within the Heat framework to ensure a fair comparison, enabling direct comparison with DASO. While Horovod shares a similar basic strategy of overlapping communication with computation, it also employs compression techniques, tensor fusions, and other optimizations to further enhance training speed. As the most popular choice for DP training on computer clusters when this work was completed, Horovod is a vital reference point for evaluating DASO's efficacy.

The primary goal was to assess each approach's strong scaling behavior. Furthermore, I sought to evaluate how training time and the task-specific target metric (top-1 accuracy for classification, intersection over union (IoU) for segmentation) evolve as the number of GPUs increases.

The network hyperparameters were held constant across all experiments, and a learning rate scheduler that reduces the LR upon training loss plateauing was employed. The local optimizer settings were also identical for each use case. Horovod was configured to use 16-bit floating-point compression for message packaging, while DASO employed Brain Floating Point 16 (BF16) compression. The classic method used 32-bit floating-point during communication. The training batch size per GPU was fixed across experiments, so the total distributed batch size scaled linearly with the number of GPUs.

Table 3.2.: Hyperparameters used to train ResNet-50 using the ImageNet-2012 dataset.

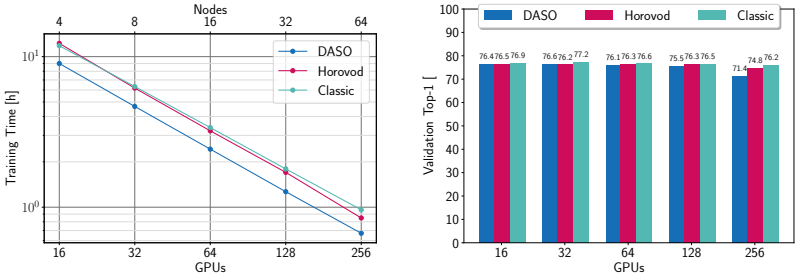
Data Loader	DALI [148]	
Local Optimizer	SGD	
Local Optimizer Parameters	Momentum: 0.9	Weight Decay: 0.0001
Epochs	90	
LR Decay	Reduce on Stable	
LR Parameters	Patience: 5	Decay Factor: 0.5
LR Warmup Phase	5 epochs, see Goyal et al. [31]	
Maximum LR	Scaled by number of GPUs [31]	
Loss Function	Cross Entropy	

Based on the findings of the hyperparameter study in Section 3.3.1, I set DASO's maximum number of batches between global synchronizations (B) to four and the number of batches between sending and receiving global parameters (S) to one. These values were chosen to balance computational speed and model accuracy as the number of training devices was scaled up.

3.3.2.1. Image Classification on ImageNet

This experiment used the ResNet-50 architecture on the ImageNet-2012 dataset and utilized the hyperparameters shown in Table 3.2. The training was conducted on 4, 8, 16, 32, and 64 nodes (16, 32, 64, 128, and 256 GPUs, respectively). The results in Figure 3.6a demonstrate desirable strong scaling behavior for DASO and Horovod up to 128 GPUs, where a doubling of GPUs roughly halves training time. However, the classic method's scaling slightly deteriorates with increasing GPUs. DASO's hierarchical communication and reduced synchronizations result in up to 25% faster training than Horovod.

Regarding accuracy Figure 3.6b, DASO and Horovod achieve similar results up to 128 GPUs, while the classic method slightly outperforms both. Beyond 128 GPUs, DASO and Horovod have at most 75% top-1 accuracy, a phenomenon partially explained by the decreased accuracy typically observed with larger batch sizes in traditional networks [31]. Since the per-GPU batch size is fixed, a larger GPU count implies a larger overall batch size, leading to accuracy degradation. This effect is amplified for DASO due to the additional impact



(a) Training Time

(b) Validation Top-1

Figure 3.6.: ImageNet ResNet-50 training times and top-1 accuracy results on the ImageNet dataset when trained with DASO, Horovod, and the classic algorithm for increasing node counts. Each node has four GPUs.

of stale gradients. The classic method, which does not compress gradients, maintains consistent accuracy across all node counts, suggesting a benefit of full-precision communication.

3.3.2.2. Semantic Segmentation – Cityscapes

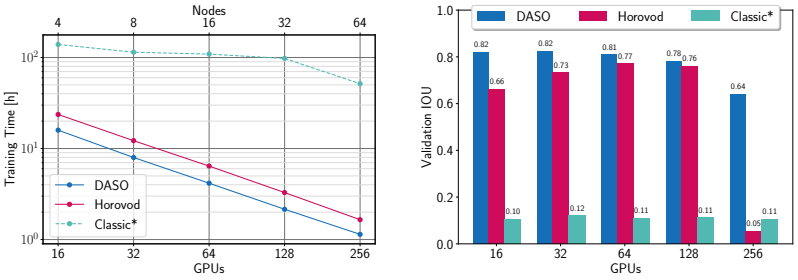
To further evaluate DASO's performance, I trained a hierarchical multi-scale attention network [151] for semantic segmentation on the Cityscapes dataset [152]. This dataset consists of finely and coarsely annotated images of streets from 50 cities. The network's architecture includes an HRNet-OCR backbone, a dedicated fully convolutional head, an attention head, and an auxiliary semantic head.

I assessed segmentation quality using the mean IoU score [153]. I established a baseline mIoU of 0.8258 by training the original network with four GPUs on a single node using PyTorch's `DistributedDataParallel` package. The HPs used for these experiments are shown in Table 3.3.

Training times for varying node counts are shown in Figure 3.7a. Notably, the classic algorithm was prohibitively slow and unable to complete within a reasonable timeframe, highlighting the necessity of optimized methods like DASO and Horovod for large-scale training. DASO outperforms Horovod by approximately 35% for up to 128 GPUs, demonstrating the advantages of its

Table 3.3.: Hyperparameters used to train the hierarchical multi-scale attention network using the Cityscapes dataset.

Data Loader	PyTorch
Local Optimizer	SGD
Local Optimizer Parameters	Momentum: 0.9 Weight Decay: 0.0001
Epochs	175
Learning Rate (LR) Decay	Reduce on Stable
LR Parameters	Patience: 5 Decay Factor: 0.75
LR Warmup Phase	5 epochs, see Goyal et al. [31]
Maximum LR	0.4
Loss Function	Region Mutual Information [154]



(a) Training Time As the classical network hit the time limit, the values are estimated.

(b) Maximum IOU Classic network accuracy values are the best results when training was stopped.

Figure 3.7.: Cityscapes Benchmarking results for the selected hierarchical split level attention network [151] on the Cityscapes dataset with DASO, Horovod, and the classic DPNN method for increasing node counts, each with four GPUs.

hierarchical communication and asynchronous updates. The time savings diminish slightly at higher GPU counts because there are fewer batches per epoch, reducing the impact of skipped global synchronizations.

Quality measurements (Figure 3.7b) indicate that while Horovod and DASO underperform compared to the original implementation, DASO consistently achieves higher mIoU scores than Horovod. This discrepancy is attributed to the use of a naive learning rate scheduler, and I hypothesize that a more refined

scheduler could close the gap. At 256 GPUs, Horovod fails to yield meaningful results, possibly due to the lack of synchronized batch normalization and the extremely large mini-batch size.

3.4. Conclusion and Outlook

In this chapter, I introduced the Distributed Asynchronous and Selective Optimization (DASO) method, a novel approach designed to alleviate the communication bottleneck that often impedes large-scale data-parallel neural network training. By adopting a hierarchical communication scheme and leveraging asynchronous updates with stale gradients, DASO reduced training time by more than 25% while maintaining comparable accuracy to established synchronous methods.

My parameter study revealed a nuanced interplay between the frequency of global synchronization, gradient staleness, and model performance. While increasing the number of batches between global synchronizations can accelerate training, it can also degrade accuracy if not carefully managed. Interestingly, the study also suggested a potential regularization effect of stale gradients under certain conditions, highlighting the need for further investigation into the complex dynamics of asynchronous distributed training.

The empirical evaluation of DASO on image classification and semantic segmentation tasks provided compelling evidence of its effectiveness. DASO consistently outperformed Horovod, a leading distributed training framework at the time of development, in terms of training time while achieving competitive accuracy. These results underscore the potential of DASO to unlock significant speedups in large-scale distributed training scenarios.

While DASO demonstrates considerable promise, several avenues for future research remain open. The observed regularization effect of stale gradients warrants a deeper theoretical and empirical analysis, potentially leading to novel insights into the optimization landscape of neural networks. Additionally, extending DASO to a broader range of model architectures and datasets will further test its robustness and generalizability.

My experimental observations on the behavior of neural network weights with stale updates prompted me to investigate the internal dynamics of these models during training. In the next chapter, I delve into the phenomenon

of weight orthogonality stabilization, which emerges as a critical factor in understanding the resilience of neural networks to stale gradients and the potential for further optimization. This exploration will shed light on the underlying mechanisms that enable efficient distributed training and pave the way for future algorithmic innovations.

4. Orthogonality in Neural Networks

The content of this chapter is based on:

D. Coquelin, K. Flügel, M. Weiel, et al. "Harnessing Orthogonality to Train Low-Rank Neural Networks". In: ECAI 2024. IOS Press, 2024, pp. 2106–2113. DOI: 10.3233/FAIA240729.

The previous chapter's exploration of data, culminating in the DASO method, sought to address the communication bottlenecks inherent to large-scale neural network training. A key observation emerged: neural networks are surprisingly robust to stale gradients during training. This robustness hints at a fundamental structural property within neural network weights that allows for efficient learning even under imperfect synchronization.

In this chapter, I shift my focus from the external challenges of distributed training to the internal mechanisms governing how networks learn. My central thesis of this chapter is that the iterative optimization process, typically guided by gradient descent, imposes a specific structure on the network's weights. Specifically, I will provide evidence that the weight matrices within a network tend to evolve towards low-rank representations, with the orthogonal basis of each matrix stabilizing during training.

As state-of-the-art neural networks often operate at the limits of computational feasibility, there has been a persistent drive toward model compression and sparsification techniques. Low-rank approximations, a class of methods that factorize matrices into smaller components, have proven particularly effective for compressing neural networks [127]. Such approaches can significantly reduce the model's memory footprint and potentially accelerate computations [129, 155].

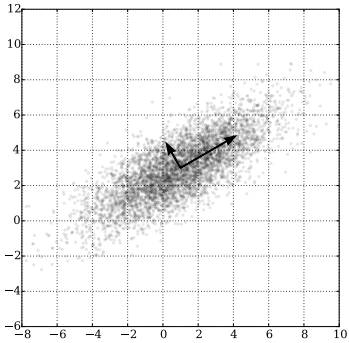


Figure 4.1.: A multivariate Gaussian distribution centered at (1,3) with a standard deviation of 3 in roughly the (0.866, 0.5) direction and of 1 in the orthogonal direction. The arrows represent the principal axes of this distribution, scaled by their respective spreads, and originating from the center point. Source: Nicoguaro [156]

SVD is a well-established tool for obtaining low-rank matrix approximations. As a refresher, it decomposes a matrix $\boldsymbol{A}$ of shape $m \times n$ as

$$\boldsymbol{A}_{m \times n} = \boldsymbol{U}_{m \times k} \Sigma_{k \times k} \boldsymbol{V}^{\top}_{k \times n} \tag{4.1}$$

where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthonormal matrices containing the left and right singular vectors of $\boldsymbol{A}$, respectively, k is the inner rank, and Σ is the diagonal matrix of singular values, which represent the importance of the corresponding singular vectors in the reconstruction of $\boldsymbol{A}$. The left singular vectors $\boldsymbol{U}$ and the singular values Σ form the principal components of $\boldsymbol{A}$. SVD enables the compression of a matrix by discarding smaller singular values and their corresponding basis vectors. This compression, however, comes at the cost of reduced reconstruction accuracy of the original matrix.

Principal components can be interpreted as the most dominant trends within a dataset. As an illustrative example, Figure 4.1 depicts a multivariate Gaussian distribution where the principal axes, scaled by their respective standard deviations, align with the directions of greatest variance. Notably, a faithful approximation of this dataset can be achieved using only a few singular vectors and their corresponding values.

I hypothesize that the orthogonal components of a network's weight matrices are learned early in training, opening the door for effective compression during later stages. This chapter presents a comprehensive investigation into this phenomenon and introduces a novel training method that leverages this basis stabilization.

Specifically, my contributions are as follows:

- I provide evidence that the orthogonal basis $\boldsymbol{U}\boldsymbol{V}^\top$ of each of a network's weights stabilizes during training.
- I propose Orthogonality-Informed, Adaptive Low-Rank (OIALR) neural network training, a novel method harnessing this finding to achieve model compression and computational efficiency.
- I demonstrate the seamless integration of OIALR into existing training workflows with minimal accuracy loss, benchmarking its performance across diverse datasets, modalities, architectures, and tasks.
- I showcase OIALR's ability to outperform conventional full-rank and other low-rank training methods, demonstrating its potential for improving efficiency and effectiveness in deep learning.

For clarity and brevity, I refer to the set containing the orthogonal basis $\boldsymbol{U}\boldsymbol{V}^\top$ of each of a network's multidimensional weights as the network's "orthogonal bases" throughout this chapter. I provide open-source implementations of the most common layer types, a method to wrap arbitrary model architectures for any learning task, and the OIALR training method [1].

4.1. Related Work

Various techniques exist to reduce the size and complexity of neural networks. These include pruning, which removes unimportant weights; quantization, which reduces the bit-precision of weights; sparse training, which encourages sparsity during model initialization and training; and low-rank compression, which reduces the memory required for storing the network itself [123].

[1] https://github.com/Helmholtz-AI-Energy/oialr

These low-rank compression methods factorize a full-rank matrix M into the product of two or more smaller matrices, reducing the number of parameters required to represent the original matrix. Formally, such a factorization is defined as $M_{m\times n} = A_{m\times r}B_{r\times n}$, where $r < \min(m, n)$, and r is referred to as the inner rank.

The application of low-rank approximations to neural networks dates back to early work on SVD-NET [157], which demonstrated the feasibility of decomposing weight matrices into smaller components, specifically the low-rank SVD of the weights and training these components directly. Subsequent research has shown that low-rank representations can be beneficial for reducing computational complexity during both training and inference [129, 155]. Additionally, low-rank methods have been shown to enhance generalization performance by mitigating overfitting [158, 159, 160].

The prevalence of large pre-trained models has spurred interest in applying low-rank approximations after training for model compression and deployment. While direct application of low-rank decompositions to pre-trained models can lead to performance degradation [161], fine-tuning low-rank models for specific tasks has shown promise. For example, LoRA [162] demonstrates that large language models can effectively adapt to specialized tasks by incorporating additional low-rank weight components during fine-tuning.

Training low-rank models from scratch while maintaining high accuracy presents unique challenges. Initial experiments with training all network weights directly in their low-rank form often had a negative impact on performance [163, 164]. In response, sophisticated methods have been developed that transition into low-rank representations gradually during training or adaptively adjust the rank throughout the process [127].

Many low-rank training methods make use of orthogonality-based low-rank representations like SVD. A matrix's orthogonal basis can be viewed as the coordinate system in which the matrix operates. The orthogonal bases of the weights can provide valuable insights into the internal workings of an otherwise opaque neural network. Intuitively, orthogonal networks can be more interpretable, as each dimension of the basis represents an independent feature. This concept has been leveraged in works like ExNN [165] and the Bort optimizer [166], which employ orthogonality constraints or regularizations to enhance model explainability.

While many low-rank training methods utilize orthogonality, maintaining this property during training is typically not a priority. However, recent research suggests that explicity enforcing or encouraging orthogonality during low-rank training can improve network performance [167, 129]. For instance, the Deep Learning Low-Rank Training (DLRT) method [129] trains each component of a singular-value decomposed weight matrix in separate forward-backward passes while explicitly preserving orthogonality. These findings underscore the potential value of integrating orthogonality constraints into low-rank training methodologies, a direction that will be explored in depth in the subsequent sections.

4.2. Orthogonality in Neural Network Training

The work of Schotthöfer et al. [129] formulates the training of a neural network as a continuous-time gradient flow. Employing specialized low-rank numerical integrators for matrix Ordinary Differential Equations (ODEs), they create a training process with three forward and backward passes for each batch to train low-rank neural networks. This training approach offers insights into the impact of the factorization on gradients, although it introduces additional computational complexity. Building upon this foundation, I aim to simplify the low-rank training process by exploiting an observed phenomenon: the orthogonal components of a network's parameters stabilize during training.

Consider a single weight matrix $\boldsymbol{W}_k(t)$ of network layer, k, of shape $m \times n$ at training step t, belonging to the manifold of matrices of rank r_k denoted as $\mathcal{M}_{r_k}$. The other networks weights are assumed to be 'fixed in time' and treated as constants for gradient calculations. From Schotthöfer et al. [129], the training can be formulated as the continuous process:

$$\min\{\|\dot{\boldsymbol{W}}_k\left(t\right) + \nabla_{\boldsymbol{W}_k}\mathcal{L}\left(\boldsymbol{W}_k\left(t\right)\right)\| : \dot{\boldsymbol{W}}_k\left(t\right) \in \mathcal{T}_{\boldsymbol{W}_k(t)}\mathcal{M}_{r_k}\} \tag{4.2}$$

where $\mathcal{L}$ is the loss function, $\mathcal{T}_{\boldsymbol{W}_k(t)}\mathcal{M}_{r_k}$ is the tangent space of $\mathcal{M}_{r_k}$ at position $\boldsymbol{W}_k\left(t\right)$, and $\dot{\boldsymbol{W}}_k\left(t\right)$ denotes the temporal derivative of the weight matrix. From a numerical analysis perspective, this optimization problem is a Galerkin condition on the tangent space [168]

$$\langle \dot{\boldsymbol{W}}_k(t) + \nabla_{\boldsymbol{W}_k}\mathcal{L}(\boldsymbol{W}_k(t)), \delta\boldsymbol{W}_k\rangle = 0 \quad \forall \delta\boldsymbol{W}_k \in \mathcal{T}_{\boldsymbol{W}_k(t)}\mathcal{M}_{r_k} \tag{4.3}$$

where $\delta \boldsymbol{W}_k$ is an element of the tangent space $\mathcal{T}_{\boldsymbol{W}_k(t)} \mathcal{M}_{r_k}$.

Using the SVD decomposition, $\boldsymbol{W}_k = \boldsymbol{U}_k \boldsymbol{S}_k \boldsymbol{V}_k^\top$, this element can be described as:

$$\delta \boldsymbol{W}_k = \delta \boldsymbol{U}_k \boldsymbol{S}_k \boldsymbol{V}_k^\top + \boldsymbol{U}_k \delta \boldsymbol{S}_k \boldsymbol{V}_k^\top + \boldsymbol{U}_k \boldsymbol{S}_k \delta \boldsymbol{V}_k^\top \tag{4.4}$$

where $\delta \boldsymbol{U}_k$ and $\delta \boldsymbol{V}_k$ are elements of the tangent space of the Stiefel manifold (the set of all orthonormal k-frames) with r_k orthonormal columns at the points $\boldsymbol{U}_k$ and $\boldsymbol{V}_k$, and $\boldsymbol{S}_k$ is a matrix of shape $r_k \times r_k$.

Now, consider the QR decomposition of $\boldsymbol{W}_k$ as

$$\boldsymbol{W}_k = \boldsymbol{Q}_k \boldsymbol{R}_k \tag{4.5}$$

where $\boldsymbol{Q}_k$ is an $m \times m$ orthonormal matrix and $\boldsymbol{R}_k$ is an $m \times n$ upper triangular matrix. Modifications to $\boldsymbol{W}_k$ can induce changes in the basis $\boldsymbol{Q}_k$, the mixing matrix $\boldsymbol{R}_k$, or both. Given that SGD operates through incremental weight adjustments, I proposed that either the basis or the mixing matrix is predominantly learned initially, followed by subsequent refinement of the other component. Specifically, I hypothesize that the orthogonal component of $\boldsymbol{W}_k$ stabilizes during the early stages of training.

To empirically validate this hypothesis, a uniquely-determined, memory-efficient orthogonal basis must be tracked through training. The 2D weight representations are chosen to be either square or with the leading dimension as the largest, i.e., $m \geq n$.

For these weights, the full $\boldsymbol{Q}$ matrix is of size $m \times m$, making it memory inefficient to track $\boldsymbol{Q}$ over multiple training steps, specifically when $m \gg n$. Therefore, the semi-orthogonal bases $\boldsymbol{U}_k \boldsymbol{V}_k^\top$ as determined by the compact SVD of $\boldsymbol{W}_k$ is used. This semi-orthogonal matrix is more memory efficient, size $m \times m$, and uniquely determined as it is the unitary factor in the polar decomposition of $\boldsymbol{W}_k$.

To find the similarity between two vectors, one can use the cosine similarity (also referred to as the cosine distance). The average of the cosine distance between each of the bases' vectors can show how much the bases differ. However, if two orthogonal bases are similar, most of the cosine distances will be near zero with only a single value close to one. To determine the

degree to which the orthogonal component of $\boldsymbol{W}_k$ changes between two timesteps i and j, I define its *Stability* as

$$S_{k,ij} = \frac{\operatorname{tr}\left(\left(\boldsymbol{U}_k\boldsymbol{V}_k^\top\right)_i \left(\boldsymbol{V}_k\boldsymbol{U}_k^\top\right)_j\right)}{m} \tag{4.6}$$

where tr is the trace of a matrix, i.e., the sum of its diagonal, and the product $\left(\boldsymbol{U}_k\boldsymbol{V}_k^\top\right)_i$ is the orthogonal component of $\boldsymbol{W}_k$ at time step i. This metric ranges from zero to one, where a *Stability* of zero indicates that the two bases share no similarities and a *Stability* of one indicates two identical bases.

The mixing matrix, $\boldsymbol{R}_k$, is tracked using a form of Euclidean similarity:

$$D_{k,ij} = 1 - \sqrt{\frac{\left(\boldsymbol{R}_{k,i} - \boldsymbol{R}_{k,j}\right)^2}{mn}} \tag{4.7}$$

where $\boldsymbol{R}_{k,i}$ and $\boldsymbol{R}_{k,i}$ are the $\boldsymbol{R}$ matrices for $\boldsymbol{W}_k$ at two different timsteps i and j. This similarity metric is chosen to maintain the same scale and behavior as *Stability*.

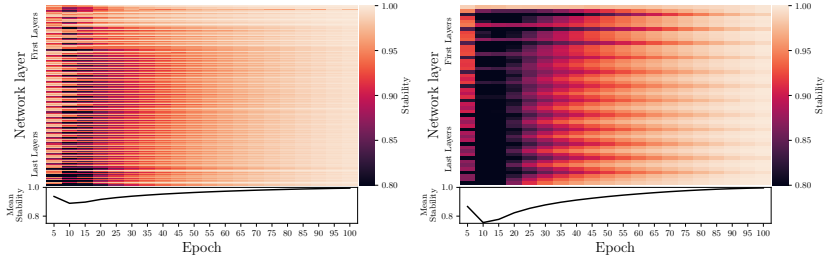
Figure 4.2 shows how the mixing matrix and the orthogonal component of the weights change during the training of two networks of different architectures: the ResNet-RS 101 [169] CNN and the VisionTransformer (ViT) B/16 [24], on ImageNet-2012 [170]. In both cases, the *Stability* (Figures 4.2a and 4.2b) decreases in the first ten epochs, i.e. the parameters' basis vectors are changing, as the networks move away from their random initialization. Then, the *Stability* converges towards one, indicating that the basis is not changing greatly between timesteps. This is in sharp contrast to the euclidean similarity plots of the linear mixing matrix $\boldsymbol{R}$, (Figures 4.2c and 4.2d), where the largest changes occur towards the middle of training, while at the beginning and end of training the changes are smaller.

These findings suggest that in later stages of training $\delta\boldsymbol{U}_k$ and $\delta\boldsymbol{V}_k$ tend towards zero. This allows us to simplify the tangent space element (Equation (4.4)) as:

$$\delta\boldsymbol{W}_k = \cancel{\delta\boldsymbol{U}_k\boldsymbol{S}_k\boldsymbol{V}_k^\top} + \boldsymbol{U}_k\delta\boldsymbol{S}_k\boldsymbol{V}_k^\top + \cancel{\boldsymbol{U}_k\boldsymbol{S}_k\delta\boldsymbol{V}_k^\top} \tag{4.8}$$

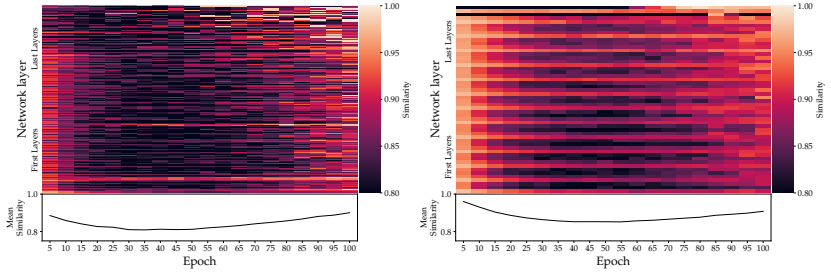
and by using the chain rule and these assumptions, the time derivative of $\boldsymbol{W}_k$ becomes

$$\dot{\boldsymbol{W}}_k = \frac{d}{dt}\left\{\boldsymbol{U}_k\boldsymbol{S}\boldsymbol{V}_k^\top\right\} = \cancel{\dot{\boldsymbol{U}}_k\boldsymbol{S}_k\boldsymbol{V}_k^\top} + \boldsymbol{U}_k\dot{\boldsymbol{S}}_k\boldsymbol{V}_k^\top + \cancel{\boldsymbol{U}_k\boldsymbol{S}_k\dot{\boldsymbol{V}}_k^\top} = \boldsymbol{U}_k\dot{\boldsymbol{S}}_k\boldsymbol{V}_k^\top. \tag{4.9}$$



(a) ResNet-RS 101, *Stability*, Equation (4.6)

(b) Vision Transformer B/16, *Stability*, Equation (4.6)



(c) ResNet-RS 101, Euclidean similarity, Equation (4.7)

(d) Vision Transformer B/16, Euclidean similarity, Equation (4.7)

Figure 4.2.: Analysis of the orthogonal basis *Stability* (Equation (4.6)) and linear mixing Euclidean similarity (Equation (4.7)) for ResNet and ViT models during ImageNet-2012 training. Both metrics compare the network's current parameters with those of five epochs prior. The x-axis denotes the training epoch, and the y-axis denotes the network layer (layers nearest the input at the top). Mean *Stability* and similarity are shown below each heatmap, showing that most of the changes to the bases happen early in training, while the linear mixing experiences the greatest changes towards the middle or training.

Thus, under these assumptions, the Galerkin condition as shown in Equation (4.3), is now

$$\langle \boldsymbol{U}_k \dot{\boldsymbol{S}}_k \boldsymbol{V}_k^\top + \nabla_{\boldsymbol{W}_k} \mathcal{L}(\boldsymbol{W}_k(t)), \delta \boldsymbol{W}_k \rangle = 0 \quad \forall \delta \boldsymbol{W}_k \in \mathcal{T}_{\boldsymbol{W}_k(t)} \mathcal{M}_{r_k} \tag{4.10}$$

and $\boldsymbol{U}_k$ and $\boldsymbol{V}_k$ can be treated as constants and with $\boldsymbol{U}_k^\top \boldsymbol{U}_k = 0$ and $\boldsymbol{V}_k^\top \boldsymbol{V}_k = 0$,

$$\langle \dot{\boldsymbol{S}}_k + \boldsymbol{U}_k^\top \nabla_{\boldsymbol{W}_k} \mathcal{L}(\boldsymbol{W}_k(t)) \boldsymbol{V}_k, \delta \boldsymbol{S}_k \rangle = 0 \tag{4.11}$$

finally, we arrive at

$$\dot{\boldsymbol{S}}_k = -\boldsymbol{U}_k^\top \nabla_{\boldsymbol{W}_k} \mathcal{L}(\boldsymbol{W}_k(t)) \boldsymbol{V}_k. \tag{4.12}$$

This simplification enables a significant reduction in the complexity of low-rank training, as one can now focus solely on updating the low-rank matrix S_k.

4.3. Orthogonality-Informed Adaptive Low-Rank (OIALR) Training

I propose a novel algorithm, Orthogonality-Informed, Adaptive Low-Rank (OIALR) training, to harness the observed stabilization of orthogonal bases during neural network training. Unlike previous methods that often prioritize either accuracy or training time, OIALR aims to maintain both by reducing the number of trainable parameters while preserving model performance.

OIALR training begins in a traditional full-rank scheme to allow the network's orthogonal bases to stabilize(see Figure 4.2). After a predetermined number of iterations d (a hyperparameter of the algorithm), the network's weight matrices are converted to their $U\Sigma V^\top$ representation via SVD. Empirically, setting d to one-third of the total number of training iterations yields favorable results. I freeze the orthogonal matrices U_k and $V_k^\top$ at this transition point, training only the square matrix Σ_k.

With a user-specified frequency ν, the bases U_k and V_k are updated by recomputing the SVD of the trained Σ_k matrix as

$$\begin{gathered} U_k', \Sigma_k', V_k'^\top \leftarrow \texttt{SVD}(\Sigma_k) \\ U_k \leftarrow U_k U_k' \qquad V_k^T \leftarrow V_k'^T V_k^T \qquad \Sigma_k \leftarrow \Sigma_k' \end{gathered} \tag{4.13}$$

By discarding singular values whose absolute magnitude falls below a threshold β times the largest singular value in Σ_k, the inner rank of the $U\Sigma V^\top$ is reduced, where β is a HP defaulting to 0.1.

As can be observed in Figure 4.2, the layers closest to the network input tend to require more training steps before they stabilize. Therefore, the update of U_k and $V_k^\top$ is initially limited to the last $\ell = L \cdot \alpha \cdot u$ weights of the network, where L is the total number of network weight tensors, α is a hyperparameter defaulting to 0.1, and u is proportional to the number of updates performed via Equation (4.13). As training progresses, more layers are

Algorithm 1: The OIALR training method

Inputs : Model M, training steps $t_{\max}$, delay steps d, low-rank update frequency ν, singular value cutoff fraction β, percentage of layers in each low-rank update step size α

Parameter: $L \leftarrow$ Number of possible low-rank weight tensors in M

Parameter: $\ell \leftarrow L \cdot \alpha$

for $t \leftarrow 1$ **to** $t_{\max}$ **do**
  if $t < d$ **then**
    Train full-rank network.
  else if $t = d$ **then**
    Convert network weights to $\boldsymbol{U}\Sigma\boldsymbol{V}^T$ representations.
    Train Σ of the $\boldsymbol{U}\Sigma\boldsymbol{V}^T$ representations ($\boldsymbol{U}$ and $\boldsymbol{V}$ are frozen).
  else if $t \bmod \nu = 0$ **then**
    for $i \leftarrow L - \ell$ **to** L **do**
      Update $\boldsymbol{U}_i\Sigma_i\boldsymbol{V}_i^T$ with Equation (4.13).
      Remove singular values $< \beta \cdot \max(\Sigma_i)$
      Reshape $\boldsymbol{U}_i$, $\boldsymbol{V}_i$, Σ_i, and optimizer states.
    $\ell \leftarrow \ell + L \cdot \alpha$
    Train Σ of the $\boldsymbol{U}\Sigma\boldsymbol{V}^T$ representations ($\boldsymbol{U}$ and $\boldsymbol{V}$ are frozen).
  else
    Train Σ of the $\boldsymbol{U}\Sigma\boldsymbol{V}^T$ representations ($\boldsymbol{U}$ and $\boldsymbol{V}$ are frozen).

gradually incorporated into the low-rank update scheme. A comprehensive overview of the OIALR training procedure is provided in Algorithm 1.

Empirically, I found that training a network's input and output layers in low rank can negatively affect predictive performance. This was most often tied to the shape of the layer weight. For example, the output layer of model trained on CIFAR10, may only have ten output channels, but thousands of inputs. Converting this matrix to low rank does not greatly benefit compression or computation. Therefore, the first, last, and any other layers with weight tensors of disadvantageous shapes can be excluded from low-rank training if desired.

4.4. Experimental Evaluation and Discussion

To evaluate the effectiveness of the OIALR training approach, I conducted a series of experiments across diverse neural network architectures and datasets. The primary objective was to demonstrate that OIALR can substantially reduce the number of trainable parameters while maintaining or even enhancing network performance and training time, unlike many existing methods that prioritize one of these aspects at the expense of the other.

The experimental evaluation was performed in three stages. First, I aimed to simulate the experience of a typical researcher by applying OIALR directly to a well-established neural network configuration for a computer vision task (Section 4.4.2). Second, I compared OIALR to other popular low-rank and sparse training methods to assess its relative effectiveness in achieving model compression and maintaining performance (Section 4.4.3). Finally, to gauge OIALR's efficacy in a more practical setting, I investigated its application on a time-series forecasting application using the Autoformer model, an architecture deployed at the 2022 Winter Olympics for weather prediction [171] (Section 4.4.5).

I employed identical HPs for baseline full-rank and OIALR training in the initial experiments. However, recognizing that OIALR dynamically modifies the network structure during training, I hypothesized that distinct HP configurations might vary between applications. Therefore, in the final two experiments (Sections 4.4.4 and 4.4.5), I employed `Propulate` [101], an asynchronous evolutionary optimization package proven effective for hyperparameter optimization (HPO) [101], to identify hyperparameters specifically tailored to OIALR.

In this section, I report the number of trainable parameters as a percentage relative to the conventional model. Non-trainable parameters in traditional models, e. g., running averages in batch normalization layers, typically make up a minuscule percentage of the total network parameters. For OIALR-trained models, the set of non-trainable parameters is dominated by the orthogonal bases $\boldsymbol{U}$ and $\boldsymbol{V}$.

To ensure that the experiments reflect real-world practices, I incorporated state-of-the-art techniques and training protocols, including strong image transformations [172], dropout [173], learning rate warm-up [174], and cosine learning rate decay [175], following the implementations detailed in the `timm`

library [1]. All networks were trained using the AdamW optimizer [176]. The complete hyperparameter configurations for each experiment are provided in Appendix A.1. The reported results represent the average of three independent runs with different random seeds.

4.4.1. Computational environment

I ran all experiments on the distributed-memory, parallel hybrid supercomputer HoreKa at KIT. Each compute node is equipped with two 38-core Intel Xeon Platinum 8368 processors at 2.4 GHz base and 3.4 GHz maximum turbo frequency, 512 GB local memory, a local 960 GB NVMe SSD disk, two network adapters, and four NVIDIA A100-40 GPUs with 40 GB memory connected via NVLink. Inter-node communication uses a low-latency, non-blocking NVIDIA Mellanox InfiniBand 4X HDR interconnect with 200 Gbit/s per port. All experiments used Python 3.10.6 with `CUDA`-enabled `PyTorch` 2.0.0 [50]. The source code for the implementation is publicly available[2].

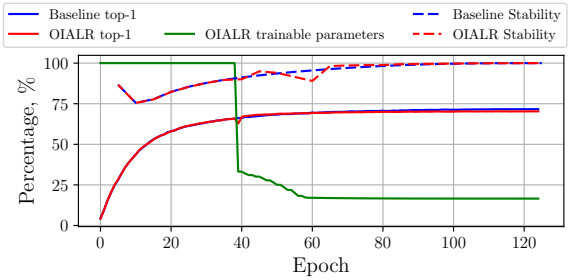
4.4.2. Naive Testing: Transformers and ResNets

Table 4.1.: Training ViT-B/16 and ResNet-RS 101 on ImageNet-2012 for 125 epochs with a batch size of 1024 with and without OIALR. Hyperparameters are identical in both cases. The final percentage of trainable parameters relative to the baseline model is reported in the last row.

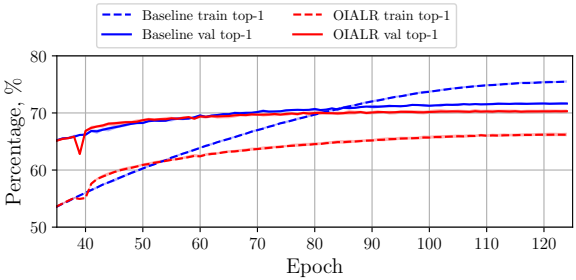
Model	ViT-B/16		ResNet-RS 101	
Training method	Baseline	OIALR	Baseline	OIALR
Loss	**2.16**	2.20	**1.78**	1.81
Top 1 Accuracy	**71.64 %**	70.30 %	**78.75 %**	77.95 %
Top 5 Accuracy	**89.18 %**	88.73 %	**94.21 %**	93.95 %
Runtime	3.29 h	**3.26 h**	**5.55 h**	5.92 h
Compression	1 : 1	**1 : 1.011**	**1 : 1**	1 : 0.955
Trainable Parameters	—	16.56 %	—	15.66 %

To assess OIALR's performance in typical computer vision scenarios, I trained the Vision Transformer (ViT)-B/16 [24] and ResNet-RS 101 [169] models on

[2] https://github.com/Helmholtz-AI-Energy/oialr



(a) Top-1 validation accuracy, percentage of trainable parameters as compared to the traditional network, and average *Stability* measured with a five-epoch frequency.



(b) Top-1 accuracies for training and validation with baseline and OIALR training methods.

Figure 4.3.: Training of a ViT-B/16 network on ImageNet-2012 over 125 epochs.

the ImageNet-2012 dataset, utilizing the ReaL validation labels [177] for a more robust evaluation. The ViT-B/16 model, with 86 million trainable parameters, provided a rigorous test for the OIALR training method. I maintained identical HPs for full-rank and OIALR training for this initial experiment to test the 'drop-in replacement' performance. To reduce computational cost and environmental impact, I limited training to 125 epochs, as validation accuracy had largely stabilized by this point (see Figure 4.3a). Additionally, I employed a reduced image resolution of 160×160 pixels to further reduce energy consumption. Table 4.1 shows the results of these experiments.

Figure 4.3 illustrates the evolution of top-1 accuracy, the percentage of trainable parameters relative to the full-rank model, and the average network *Sta-*

bility throughout training. Notably, the baseline model's *Stability* increases smoothly while OIALR's *Stability* fluctuates due to the periodic reduction in weight ranks. This fluctuation stems from the fact that the orthogonal bases from earlier epochs contain more vectors than the current, reduced-rank bases. For example, $\boldsymbol{U}$ is of shape $m \times k_1$ in an earlier epoch, while later in training, it is of shape $m \times k_2$ where k_1 is larger than k_2. As this is the inner dimension for both $\boldsymbol{U}$ and $\boldsymbol{V}^\top$, more vectors are involved in creating $\boldsymbol{U}_1\boldsymbol{V}_1^\top$ than in $\boldsymbol{U}_2\boldsymbol{V}_2^\top$.

As shown in Figure 4.3b, there is a temporary dip in accuracy when transitioning from full-rank to the $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ representation, likely due to residual momentum states in the optimizer. However, the network quickly recovers and performs comparably to the full-rank model. In this untuned scenario, OIALR training required only 1% more time while maintaining accuracy within 1.34 % of the baseline while drastically reducing the number of trainable parameters to 16.56 % of the full-rank model. Crucially, Figure 4.3b also shows that the full-rank model enters an overfitting regime, whereas the OIALR-trained model does not, suggesting potential generalization benefits.

4.4.3. Comparison with related low-rank and sparse training methods

To show where OIALR fits into the landscape of full-to-low-rank, low-rank, and sparse training methods, I performed a comparative analysis shown in Table 4.2. In these experiments, the baseline and compression methods use the same HPs.

OIALR and DLRT [129] are SVD-based low-rank factorization methods. LRNN [178] uses a traditional two-matrix representation ($\boldsymbol{W} \approx \boldsymbol{A}\boldsymbol{B}$). CP [179], SFP [180], PP-1 [155], and ThiNet [103] are structured pruning methods using either channel or filter pruning for convolution layers. GAL [181] and RNP [182] are unstructured pruning methods. RigL [119] is a sparse training method.

OIALR demonstrated competitive performance across multiple architectures and datasets. Although achieving subpar results on ResNet-50, it marginally improved accuracy over the baseline for VGG16 [183] on CIFAR-10. This suggests that OIALR may be particularly effective for over-parameterized

networks, where eliminating less useful basis vectors allows those remaining to contribute more meaningfully to the overall performance.

Table 4.2.: Comparison of OIALR with various compression methods. 'Diff. to baseline' refers to the difference in top-1 validation (ImageNet-2012) or test (CIFAR-10) accuracy between the baseline and the listed methods. Positive values indicate that the listed method outperforms the traditionally trained network. Absence of data indicated by '—'. For non-OIALR results see [129, 119].

	Training method	Diff. to baseline	Compression	Trainable parameters
ResNet-50 ImageNet-2012	OIALR [184]	-1.72 %	1 : 1.21	15.15 %
	DLRT [129]	-0.56 %	1 : 1.85	**14.20 %**
	PP-1 [155]	**-0.20 %**	1 : 2.26	—
	CP [179]	-1.40 %	1 : 2.00	—
	SFP [180]	**-0.20 %**	1 : 2.39	—
	ThiNet [103]	-1.50 %	1 : 2.71	—
	RigL [119]	-2.20 %	**1 : 5.00**	—
VGG16 ImageNet-2012	OIALR [184]	-1.53 %	**1 : 2.74**	**4.77 %**
	DLRT [129]	-2.19 %	1 : 1.16	78.40 %
	PP-1 [155]	**-0.19 %**	1 : 1.25	—
	CP [179]	-1.80 %	1 : 1.25	—
	ThiNet [103]	-0.47 %	1 : 1.45	—
	RNP(3X) [182]	-2.43 %	1 : 1.50	—
VGG16 CIFAR-10	OIALR [184]	**0.10 %**	**1 : 3.70**	**13.88 %**
	DLRT [129]	-1.89 %	1 : 1.79	77.50 %
	GAL [181]	-1.87 %	1 : 1.30	—
	LRNN [178]	-1.90 %	1 : 1.67	—

4.4.4. Ablation study on a mini ViT on CIFAR-10

To assess the impact of HPO on OIALR's performance, I trained a reduced-size ViT model, termed a mini ViT, on the CIFAR-10 dataset both with and without hyperparameter optimization. I used the same hyperparameters as the baseline full-rank training for the untuned runs. In contrast, I employed the `Propulate` framework [101] to search for hyperparameters tailored explicitly to the OIALR training process for the tuned runs.

Table 4.3.: A mini ViT trained on CIFAR-10. 'OIALR, tuned' training runs used tuned HPs, while 'OIALR' used the same HPs as the baseline. Accuracies and loss values are determined on the test dataset.

Training method	Baseline	OIALR	OIALR, tuned
Loss	0.88	0.91	**0.85**
Top 1 Accuracy	85.17 %	83.05 %	**86.33 %**
Top 5 Accuracy	98.34 %	98.38 %	**98.53 %**
Time to Train	12.14 min	11.99 min	**11.19 min**
Compression	1: 1	1 : 0.68	**1 : 1.82**
Trainable Parameters	—	30.98 %	**9.97 %**

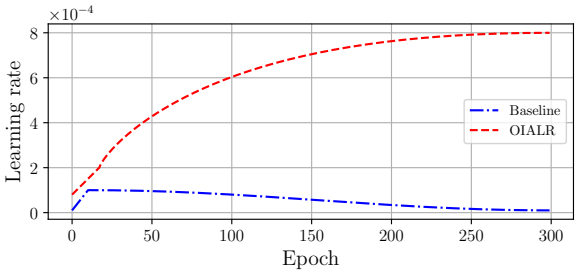


Figure 4.4.: Learning rate schedules for baseline and OIALR training for a mini ViT on CIFAR-10. OIALR training learning rate schedule determined by HP search.

Given the demonstrated efficacy of reduced-size ViT models in achieving strong performance with lower computational costs [185], I opted for a ViT-B/16 variant with a patch size of eight, six layers, and six attention heads; the original ViT-B/16 configuration uses a patch size of 16, 12 layers, and 12 attention heads.

The results of this experiment, presented in Table 4.3 and Figure 4.4, reveal several interesting insights. Notably, the optimal learning rate schedule discovered by `Propulate` for OIALR training involves increasing the LR as the number of parameters decreases. This aligns with the intuition that with fewer trainable parameters, the effective learning rate for the remaining parameters can be increased without causing model degradation.

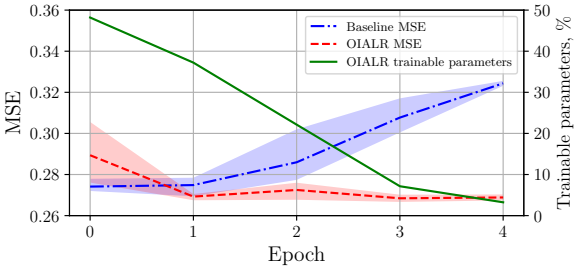
In the untuned case, while OIALR reduced trainable parameters by 69.02 %, the top-1 test accuracy dropped by over 2%. In contrast, the tuned OIALR model achieved a remarkable reduction of 90.03 % in trainable parameters while simultaneously improving predictive performance from 85.17 % to 86.33 %. Moreover, the training time was reduced by 8.52 % compared to the baseline.

4.4.5. Ablation study on Autoformer on ETTm2

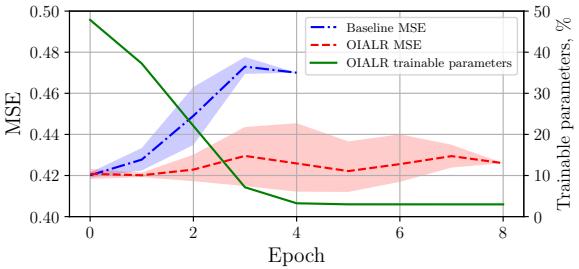
Table 4.4.: Training of the Autoformer model on the ETTm2 dataset. Baseline and untuned OIALR HPs were the default parameters from [171]. Tuned OIALR HPs were found via `Propulate`. Prediction lengths (PL) in the leftmost column are in 15 min time steps. The optimal value for mean squared error (MSE) and mean absolute error (MAE) is zero.

PL	Training method	MSE	MAE	Compression	Trainable parameters
96	Baseline	0.2145	0.2994	1 : 1.00	—
	OIALR	0.2140	0.2974	1 : 0.55	46.16 %
	OIALR, tuned	**0.2112**	**0.2942**	**1 : 2.09**	**12.19** %
192	Baseline	0.2737	0.3356	**1 : 1.00**	—
	OIALR	0.2773	0.3336	1 : 0.62	105.31 %
	OIALR, tuned	**0.2686**	**0.3305**	1 : 0.95	**27.15 %**
336	Baseline	0.3277	0.3640	1 : 1.00	—
	OIALR	0.3253	0.3863	1 : 0.56	45.67 %
	OIALR, tuned	**0.3212**	**0.3591**	**1 : 3.66**	**7.14 %**
720	Baseline	0.4194	0.4157	1 : 1.00	—
	OIALR	0.4213	0.4186	1 : 0.52	51.33 %
	OIALR, tuned	**0.4120**	**0.4147**	**1 : 7.38**	**4.46 %**

The training of a time-series forecasting transformer serves as a critical test for the OIALR method. The Electricity Transformer Temperature (ETT) dataset [186] provides a valuable benchmark for time-series forecasting. It comprises 70,000 measurements at varying time granularities, each with seven features, recording load and oil temperature data from electrical transformers. I focus on the ETTm2 dataset, which offers a 15-minute resolution. Common prediction horizons for this dataset include 96, 192, 336, and 720 time steps.



(a) Prediction length 2 days, 192 15 min time steps



(b) Prediction length 7.5 days, 720 15 min time steps

Figure 4.5.: MSE and the percentage of trainable parameters relative to the full-rank model for the Autoformer trained on the ETTm2 dataset using two different prediction lengths in 15 min time steps.

For my experiments, I employed the Autoformer model [171], a well-known Transformer-based architecture available in the HuggingFace [187] repository. The Autoformer distinguishes itself from other Transformer models by incorporating auto-correlation layers and one-dimensional convolutions. This design contributed to its successful deployment for weather forecasting at the 2022 Winter Olympics.

As seen in Figure 4.5, the Autoformer is susceptible to rapid overfitting of this dataset. To combat this, OIALR training was done without a full-rank warm-up. While some overfitting was still observed in the OIALR results, it was notably less severe than in the baseline. As shown in Table 4.4, the tuned

OIALR models consistently outperformed the baseline across all prediction lengths while dramatically reducing the number of parameters.

The untuned OIALR models also demonstrated competitive performance, surpassing the baseline in some instances and achieving an reducing percentage of trainable parameters to 45.72% on average. However, due to the specific structure of the model's $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ representation and the inability of the model to adequately reduce the inner rank, these models required more total parameters than the baseline. The tuned OIALR models, in contrast, generally achieved a much higher compression ratio. Interestingly, the tuned OIALR models required more parameters for shorter prediction horizons, highlighting the nuanced interaction between model complexity, task requirements, and the effectiveness of low-rank representations.

In contrast to the previous experiment with the ViT model, the optimal learning rate schedule identified for this use case closely resembled a traditional schedule with a warm-up phase followed by gradual decay. This observation may be related to the rapid overfitting exhibited by both low-rank and full-rank models on this dataset (Figure 4.5), suggesting that aggressive learning rate reduction might be beneficial in such scenarios.

4.5. Conclusion

In this chapter, I exposed aspects of the underlying mechanisms of neural network learning by examining the evolution of the network's weights throughout training. By employing SVD to analyze weight matrices, I observed a striking phenomenon: the orthogonal component of a network weight tends to stabilize early in the training process. This insight deepens our understanding of the dynamics of neural network optimization and provides a foundation for novel training methodologies.

Building upon this observation, I introduced Orthogonality-Informed, Adaptive Low-Rank (OIALR) training, a method designed to exploit this stabilization for model compression and computational efficiency. My comprehensive evaluation across various neural network architectures and datasets demonstrates OIALR's effectiveness in significantly reducing the number of trainable parameters while preserving, and in some cases enhancing, model performance and training speed. While OIALR may not universally surpass

traditional full-rank training in all scenarios, my results show its potential to outperform these methods with appropriate hyperparameter tuning.

The substantial reduction in trainable parameters achieved by OIALR holds significant implications for practical deep-learning applications. Smaller models facilitate fine-tuning, transfer learning, and deployment on resource-constrained devices, making deep learning more accessible and adaptable. Moreover, this reduction in model size directly translates to decreased communication overhead during distributed training, potentially narrowing the performance gap between high-end computing clusters and more affordable alternatives.

However, OIALR represents a single step towards harnessing the power of orthogonality in deep learning. In the following chapter, I broaden my scope to investigate the integration of orthogonality-informed training methods into a large-scale distributed data-parallel workflow. I aim to demonstrate that the principles underlying OIALR can be effectively leveraged to address the communication bottlenecks inherent in these settings, thus paving the way for a new generation of efficient and scalable distributed training algorithms.

5. Using Low-Rank Representations in Data Parallel Training

The content of this chapter is based on:

D. Coquelin, K. Flügel, M. Weiel, et al. "AB-Training: A Communication-Efficient Approach for Distributed Low-Rank Learning". In: (2024). URL: https://arxiv.org/abs/2405.01067. arXiv: 2405.01067 [cs.LG].

The pursuit of improved predictive accuracy in machine learning has driven the development of increasingly complex and large-scale neural networks. Training these models requires vast datasets and substantial computational resources, pushing the boundaries of current hardware capabilities. Data parallelism (DP) has emerged as the dominant paradigm for distributed training, enabling the distribution of the computational workload across multiple devices. However, this approach introduces a significant communication bottleneck, as maintaining a synchronized model across all workers necessitates frequent exchange of large volumes of data. Furthermore, training with large batch sizes, often employed in data parallel settings, can negatively impact model generalization.

This chapter explores the potential of low-rank representations to address these challenges. Building upon the previous chapter, which demonstrated the stabilization of orthogonal bases in the singular value decomposition (SVD) of weight matrices during training, and Chapter 3, I propose a novel approach that leverages low-rank approximations and hierarchical independently trained groups to reduce communication overhead and improve generalization in DP training. This approach seeks to capitalize on two key advantages of low-rank representations: their inherent ability to compress model information, leading to reduced communication volume, and their

regularization properties, which can mitigate the detrimental effects of large batch sizes.

My core contributions presented in this chapter are as follows:

- I introduce *AB* training, a novel low-rank training method based on SVD and a hierarchical training scheme designed to reduce communication overhead in data-parallel training.
- I empirically demonstrate *AB* training's ability to substantially reduce network traffic (by an average of 70.31%) compared to traditional synchronous DP training, enabling efficient training on systems with limited bandwidth.
- I provide evidence that *AB* training improves regularization at smaller scales, particularly for Vision Transformers, and enhances generalization performance.
- I demonstrate *AB* training's potential to achieve significant compression ratios (up to 44.14 : 1 in ideal scenarios) while maintaining competitive accuracy. Compression ratios ranging from 1.19 : 1 to 2.54 : 1 are achieved in more realistic settings while surpassing the accuracy of traditional DP training.

This chapter highlights the promise of low-rank training for large-scale distributed learning, offering potential benefits for both distributed-memory computing and the broader machine learning (ML) community. An open-source implementation of my method is available at `https://github.com/Helmholtz-AI-Energy/AB-Training` and the configurations to reproduce the experiments is shown in the Appendix A.2.

5.1. Related Work

5.1.1. Distributed Training of Neural Networks

Data parallel (DP) training, the prevalent paradigm for distributed training, replicates a model across multiple devices, each processing a disjoint subset (the local batch) of the training data (Section 2.5.1). Gradients are typically aggregated across all replicas after each forward-backward pass, ensuring synchronous updates. However, as model size and compute requirements

increase, the communication overhead associated with gradient synchronization becomes a significant bottleneck, particularly in bandwidth-constrained environments.

Various techniques have been proposed to mitigate this bottleneck, including gradient accumulation, topology-aware communication patterns, asynchronous methods, and gradient compression [114, 188]. Hierarchical methods like H-SGD [135] and DASO [140] leverage localized synchronization within smaller groups before global updates, exploiting network topology to reduce communication costs. Asynchronous approaches, often employing a parameter server, can reduce waiting times but require careful hyperparameter tuning and may face convergence challenges [64]. Despite these advancements, scalability in data parallelism remains constrained by communication bottlenecks and the adverse impact of large batch sizes on generalization performance.

5.1.2. Low-Rank Neural Network Training

Low-rank training offers a promising avenue for reducing neural networks' computational and memory footprint. The singular value decomposition (SVD) plays a central role in this domain, enabling the factorization of weight matrices into smaller components. By retaining only the k largest singular values and corresponding vectors, a low-rank approximation of the original matrix can be obtained, significantly reducing the number of parameters for a sufficiently large k.

As detailed in Section 4.1, various methods have leveraged SVD for compressing neural networks during training. Some directly train the $\boldsymbol{U}$, Σ, and $\boldsymbol{V}$ matrices resulting from the SVD of a network's weights [129, 184], while others construct alternative representations based on SVD and train on those [189, 190]. Both approaches have demonstrated potential regularization effects, improving generalization in specific scenarios [158, 160, 159].

While these techniques address various computational challenges in neural network training, their explicit integration with distributed environments to optimize communication remains an active area of research. More precisely, existing approaches can train low-rank models in parallel, but they do not attempt to use the distributed system to their advantage [184, 191, 192].

5.2. AB Training

Data-parallel neural network training often faces two significant challenges at scale: the communication bottleneck incurred by synchronizing large model representations across compute nodes and the potential for degraded generalization performance associated with large batch sizes. To address these, I propose a novel training method, termed *AB training*, which integrates low-rank weight representations within a hierarchical data-parallel framework. *AB* training mitigates the limitations previously mentioned by leveraging the reduced communication requirements of low-rank representations and the improved generalization potential of training worker subgroups independently.

While often necessary for efficient data-parallel training, large batch sizes can negatively impact model generalization. This stems from the reduced gradient noise inherent in large-batch SGD. The standard SGD update rule is:

$$w_t = w_{t-1} - \frac{\eta}{B} \sum_{i=1}^{B} \nabla_w \mathcal{L}_{t-1,i} \left(\hat{y}_{t-1,i}, y_{t-1,i} \right), \tag{5.1}$$

where w_t represents a parameter at time step t, η is the learning rate, B is the batch size, and $\mathcal{L}_{t-1,i} \left(\hat{y}_{t-1,i}, y_{t-1,i} \right)$ is the loss for the i'th data element in the mini-batch evaluated during time step $t-1$. Consequently, the parameter value after k iterations is

$$w_k = w_0 - \frac{\eta}{B} \sum_{s=1}^{k-1} \sum_{i=1}^{B} \nabla_w \mathcal{L}_{s,i} \left(\hat{y}_{s,i}, y_{s,i} \right). \tag{5.2}$$

Larger batch sizes reduce the magnitude of the summed gradients and the associated noise, potentially resulting in the convergence toward sharp minima in the loss landscape, which result in overfitting and poor generalization [72, 30].

Motivated by ensemble methods' success and noise's beneficial role in generalization [30], *AB* training partitions the DP model replicas into smaller, independently trained subgroups. Each subgroup trains independently on a distinct data subset, effectively creating an ensemble of models. The increased gradient noise within these smaller local batches encourages broader exploration of the loss landscape and convergence towards diverse local minima, enhancing the overall model's generalization capabilities. Furthermore,

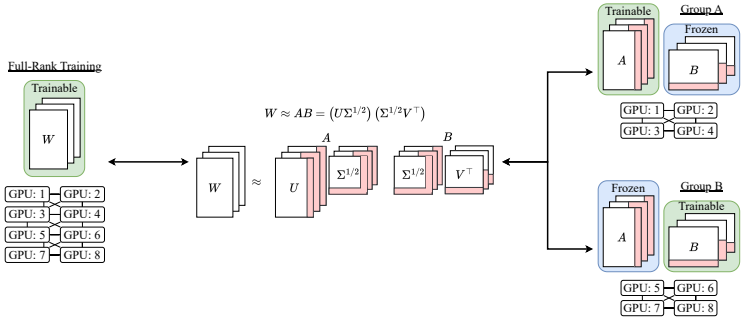


Figure 5.1.: Visualization of communication groups and trainable matrices during *AB* training phases. Red shaded regions represent removed matrix elements.

training independent groups reduces network traffic compared to traditional DP synchronization.

To minimize communication overhead *AB* training employs a low-rank representation of the weight matrices derived from their SVD. For a weight matrix $W_{m\times n}$ in a given layer, the decomposition is

$$\boldsymbol{W}_{m\times n} \approx \boldsymbol{A}_{m\times k}\boldsymbol{B}_{k\times n} = \left(\boldsymbol{U}_{m\times k}\boldsymbol{\Sigma}^{1/2}_{k\times k}\right)\left(\boldsymbol{\Sigma}^{1/2}_{k\times k}\boldsymbol{V}^{\top}_{k\times n}\right), \tag{5.3}$$

where $\boldsymbol{U}$, $\boldsymbol{\Sigma}$, and $\boldsymbol{V}$ as determined by the SVD of $\boldsymbol{W}$, and k represents the inner rank, determined by:

$$k = \max\left(\{i : \Sigma_{ii} > \Sigma_{11} \cdot \texttt{sigmaCutoff}\}\right), \tag{5.4}$$

where `sigmaCutoff` is a user-defined HP less than one, and Σ_{ii} is the i'th diagonal element of Σ.

Weight tensors with more than two dimensions are flattened along their trailing dimensions for decomposition. If the resulting matrix shape is computationally unfavorable for SVD (i.e., $n > m$), it is transposed before decomposition.

Instead of training directly on the Σ matrix, which would lead to zero gradients for off-diagonal elements, *AB* training operates on the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices as defined in Equation (5.3). This ensures that the orthogonal components of the weight matrix are updated indirectly, maintaining a dense representation and avoiding "dead" connections during backpropagation.

To expedite initial convergence, the *AB* training process commences with a synchronous full-rank DP warmup phase. Subsequently, the workers are split into independently trained groups. During this independent phase, half of the subgroups (*A* groups) train the $\boldsymbol{A}$ matrix, while the remaining half (*B* groups) train the $\boldsymbol{B}$ matrix (as illustrated in Figure 5.1).

The changes made to the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices on worker i can be represented as

$$\boldsymbol{A}_i = \boldsymbol{A}_0\boldsymbol{C}_{A,i} \qquad \boldsymbol{B}_i = \boldsymbol{C}_{B,i}\boldsymbol{B}_0 \tag{5.5}$$

where $\boldsymbol{C}_{A,i}$ and $\boldsymbol{C}_{B,i}$ are real matrices and $\boldsymbol{A}_0$ and $\boldsymbol{B}_0$ are the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices from the last *AB* decomposition of $\boldsymbol{W}$ as shown in Equation (5.3). Therefore, the changes to a weight matrix $\boldsymbol{W}$ on worker i can be approximated as:

$$\boldsymbol{W}_i \approx \boldsymbol{A}_i\boldsymbol{B}_i = \boldsymbol{A}_0\boldsymbol{C}_{A,i}\boldsymbol{C}_{B,i}\boldsymbol{B}_0. \tag{5.6}$$

One of the $\boldsymbol{C}$ matrices on each worker is the identity matrix, reflecting that either $\boldsymbol{A}$ or $\boldsymbol{B}$ is fixed during the independent training phase.

For an example system with four workers, each of which represents an entire group, the traditional averaging of the full-rank weight representation is

$$\overline{\boldsymbol{W}} \approx \overline{\boldsymbol{A}\boldsymbol{B}} = \frac{1}{4}\left(\boldsymbol{A}_1\boldsymbol{B}_0 + \boldsymbol{A}_2\boldsymbol{B}_0 + \boldsymbol{A}_0\boldsymbol{B}_3 + \boldsymbol{A}_0\boldsymbol{B}_4\right) \tag{5.7}$$

where $\boldsymbol{A}_i$ and $\boldsymbol{B}_i$ are the learned $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices on worker i.

Unlike the traditional systems for averaging independently trained models, *AB* training averages the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices across all workers before reconstructing the full-rank weight matrix. This results in the merging operation

$$\overline{\boldsymbol{W}} \approx \overline{\boldsymbol{A}}\,\overline{\boldsymbol{B}} = \frac{1}{16}\begin{pmatrix} 4\boldsymbol{A}_0\boldsymbol{B}_0 + 2\boldsymbol{A}_1\boldsymbol{B}_0 + 2\boldsymbol{A}_2\boldsymbol{B}_0 + 2\boldsymbol{A}_0\boldsymbol{B}_3 + 2\boldsymbol{A}_0\boldsymbol{B}_4 + \\ \boldsymbol{A}_1\boldsymbol{B}_3 + \boldsymbol{A}_1\boldsymbol{B}_4 + \boldsymbol{A}_2\boldsymbol{B}_3 + \boldsymbol{A}_2\boldsymbol{B}_4 \end{pmatrix}. \tag{5.8}$$

By incorporating both the stale model states ($\boldsymbol{A}_0\boldsymbol{B}_0$), the independently learned states ($\boldsymbol{A}_i\boldsymbol{B}_0$ and $\boldsymbol{A}_0\boldsymbol{B}_i$), and the mixed states ($\boldsymbol{A}_i\boldsymbol{B}_j$ where $i \neq j$). This aggregation method allows for a richer mixing of the information learned by the independent subgroups.

The *AB* training procedure consists of the following phases:

Full-Rank DP Warmup: Initial training with traditional data parallelism allows the weights to move quickly away from their random initialization. This phase helps to avoid early divergence for a given number of iterations (`warmupSteps`).

Independent *AB* Decomposition: Each model instance independently computes an *AB* decomposition of its weight matrices as per Equation (5.3). The rank of the approximation is determined by a user-provided hyperparameter (`sigmaCutoff`) and Equation (5.4).

Group Training: For `numABSteps` iterations, half of the independent groups train the $\boldsymbol{A}$ matrix (the A groups) while the other groups train $\boldsymbol{B}$ (the B groups). Untrained matrices are frozen.

Synchronization and Update: The $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices are averaged across all workers, and the full-rank weight matrices are reconstructed.

Full-Rank DP Rebound: The reconstructed full-rank network is trained with traditional data parallelism for `fullRankReboundSteps` to promote convergence and mitigate potential accuracy degradation.

A formalized algorithm is shown in Algorithm 2, and a method diagram is shown in Figure 5.2.

Many neural network optimizers, such as AdamW [176], rely on second-order derivative approximations. Due to dimensionality changes, switching between low-rank and full-rank representations can invalidate these approximations. Therefore, *AB* training will reset the optimizer states which were invalidated. To repopulate the states with meaningful information, *AB* training utilizes a *learning rate rebound* strategy. The learning rate rebound involves reducing the learning rate (LR) to near zero and gradually increasing it back to its scheduled value over a user-specified number of steps, allowing the optimizer to adapt to the new parameter representations. This rebound is applied whenever the network's parameters change shape.

5.3. Experimental Evaluation

A series of experiments were conducted on established neural network architectures to assess the effectiveness of *AB* training. These experiments focus on image classification tasks using the ImageNet-2012 [193] and CIFAR10 [194] datasets. The primary goals were to demonstrate significant reductions in communication overhead, potential regularization benefits, scalability, and achievable compression during training. Specifically, the ResNet-50 [4] and

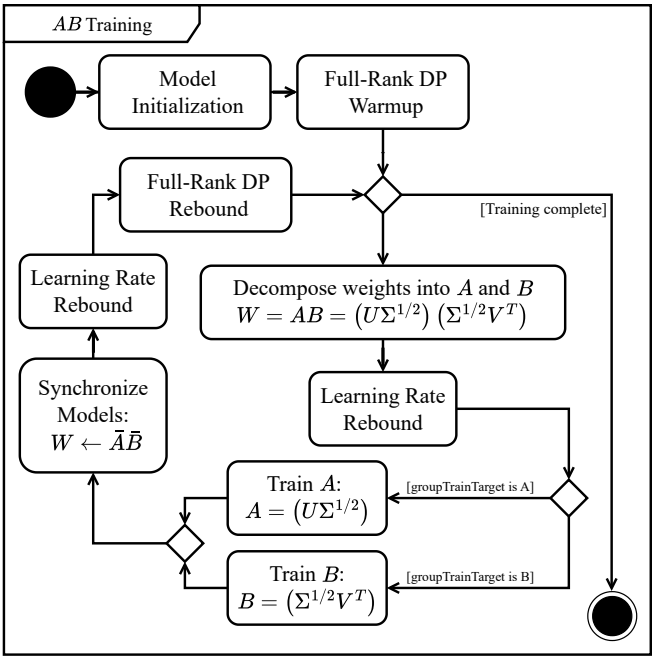


Figure 5.2.: A UML diagram of the AB training procedure.

Vision Transformer (ViT) B/16 [24] models were trained on ImageNet-2012, and the VGG16 [195] model was trained on CIFAR10.

For comparison, I implemented a traditional DP training baseline using PyTorch's `DistributedDataParallel` (DDP) class, referred to as "Traditional DDP" or "Trad. DDP". Additionally, I included an "*AB* Training (No Group)" variant where the B matrix is trained using traditional DP instead of the full-rank weights, allowing us to isolate the impact of the group training strategy. The ViT and ResNet-50 experiments utilized HPs from Dosovitskiy et al. [24], while VGG16 HPs were taken from Coquelin et al. [184]. All methods employs the same hyperparameters.

I investigated two scaling strategies: constant local batch size scaling and constant global batch size scaling. In the former, the global batch size increases proportionally with the number of GPUs, simulating common practice in large-scale training. In the latter, the global batch size is held constant while

Algorithm 2: The *AB* training method. $\boldsymbol{W}$ is a parameter of the input model M and *worldSize* is the number of workers used for traditional DP training, each with an individual ID *procId*.

Input : Model M , training data, $numTrainingSteps$, hyperparameters $warmupSteps$, $numABSteps$, and $fullRankReboundSteps$

```
if workerId ≤ worldSize / 2 then
    groupTrainTarget ← A
else
    groupTrainTarget ← B
for i ← 1 to warmupSteps do
    traditionalFullRankDPTraining()
repeat
    foreach W in M do
        A, B ← abDecomposition(W)                    ▷ Eq. 5.3
        removeSmallSingularValues()
    startLearningRateRebound()
    groupTrainTarget.setTrainable(True)
    for i ← 1 to numABSteps do
        independentSubgroupTraining()
    A ← allReduce(A/worldSize)
    B ← allReduce(B/worldSize)
    foreach W in M do
        W ← AB
    startLearningRateRebound()
    for i ← 1 to fullRankReboundSteps do
        traditionalFullRankDPTraining()
until numTrainingSteps == completed steps
```

the local batch size decreases proportionally with the number of GPUs, highlighting the communication reduction potential of my method. All reported measurements represent the average of three runs with different random seeds, and models were initialized using orthogonal initialization [196].

5.3.1. Computational Environment

All experiments were conducted on HoreKa, a distributed-memory, parallel hybrid supercomputer. Each compute node is equipped with two 38-core Intel Xeon Platinum 8368 processors, 512 GB of local memory, a 960 GB NVMe SSD, two network adapters, and four NVIDIA A100-40 GPUs interconnected via NVLink. Inter-node communication utilizes a high-speed, low-latency NVIDIA Mellanox InfiniBand 4X HDR interconnect. The software environment consisted of Python 3.10.6 and CUDA-enabled PyTorch 2.0.0 [50].

5.3.2. Datasets and Models

I employed the ImageNet-2012 dataset for the scaling experiments, comprising 1.2 million images. Basic image augmentation techniques were applied, including normalization, random resizing and cropping, and random horizontal flipping. I trained the ResNet-50 and ViT-B/16 models on this dataset, chosen for their widespread use and distinct architectural characteristics. Additionally, I trained VGG16 on the CIFAR10 dataset, containing 50,000 images, with similar image augmentation applied. All models were trained using the AdamW optimizer [176]. Comprehensive training hyperparameters are detailed in Appendix A.2.

5.3.3. Hyperparameter Considerations

Careful hyperparameter tuning is essential to balance the communication reduction achieved by *AB* training with potential impacts on training stability and accuracy. Key hyperparameters include warmup and full-rank rebound durations, the number of *AB* training iterations, the rank-reduction parameter for *AB* decomposition, and the frequency of SVD and synchronization steps.

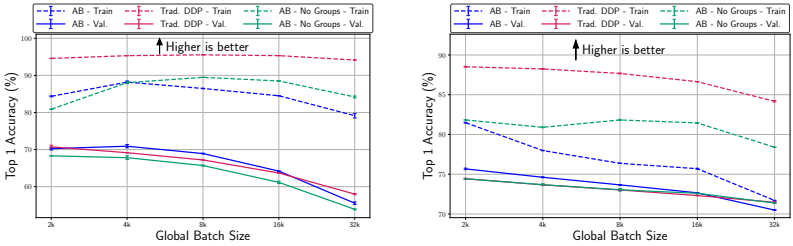
My hyperparameter search on CIFAR100 with a smaller ViT variant using `Populate` [101] yielded the following guidelines, which were used for all experiments:

- `warmupSteps`: 20% of the total training steps
- `numABSteps`: 3.33% of the total training steps
- `fullRankReboundSteps`: 0.25 · `numABSteps`

- Learning rate rebound steps: $0.5 \cdot$ `numABSteps`

These guidelines offer a reasonable starting point for hyperparameter tuning. However, optimal configurations may vary depending on the model, dataset, and computational environment.

5.3.4. Constant Local Batch Size Scaling



(a) Vision Transformer B/16 **(b)** ResNet-50

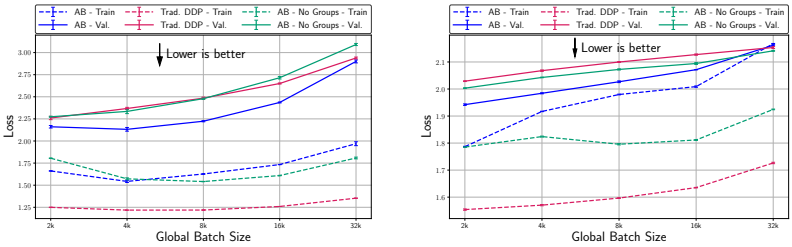
Figure 5.3.: Highest top-1 accuracy for each training run on ImageNet-2012 for two network architectures with a constant *local* batch size of 256. Global batch sizes range from 2,048 to 32,768 in powers of 2. Error bars are plotted, though not always visible.

In this experiment, I maintain a fixed local batch size while increasing the number of GPUs, effectively scaling up the global batch size while decreasing the number of training iterations per epoch. For instance, if a two-node (8-GPU) run employs a global batch size of 2,048 and undergoes 600 iterations per epoch, a corresponding four-node (16-GPU) run would have a global batch size of 4,096 but only 300 iterations per epoch. This strategy enables me to investigate the communication efficiency gains of *AB* training as computational resources are increased while the potential negative impacts of larger batch sizes become more prominent. This scenario is representative of how users typically scale their training in practice.

The experiment aims to evaluate *AB* training's effectiveness in reducing communication requirements and assess whether independent training groups can effectively learn diverse representations. The averaged model should maintain accuracy while exhibiting increased compression if the groups learn similar updates. In contrast, significant divergence between the groups should lead to reduced accuracy and compression, as individual updates conflict. In

such a case, *AB* training without independent groups (the "No Group" variant) would likely outperform *AB* training with groups.

Figure 5.3 presents the highest top-1 accuracy achieved during training for ResNet-50 and ViT-B/16 models under constant local batch size scaling. Figure 5.4 illustrates the corresponding lowest binary cross-entropy loss values across all epochs. Figure 5.5 depicts compression ratios achieved by *AB* training compared to the baseline.



(a) Vision Transformer B/16 **(b)** ResNet-50

Figure 5.4.: Lowest binary cross-entropy (the loss function used during training) during each training run on ImageNet-2012 for two network architectures with a constant *local* batch size of 256. Global batch sizes range from 2,048 to 32,768 in powers of 2. Error bars are plotted, though not always visible.

5.3.5. Constant Global Batch Size Scaling

In this experiment, I maintain a fixed global batch size of 4,096 while scaling the number of GPUs. Deviating from the common practice of maintaining constant local batch sizes, this approach accentuates the communication overhead reductions achieved by *AB* training. By reducing the computational load on each GPU, the communication bottleneck becomes more apparent. Additionally, it provides insights into the ability of independent training groups to learn meaningful representations when their local batch sizes are much smaller than the global batch. The results for these runs are shown in Figures 5.6 and 5.7.

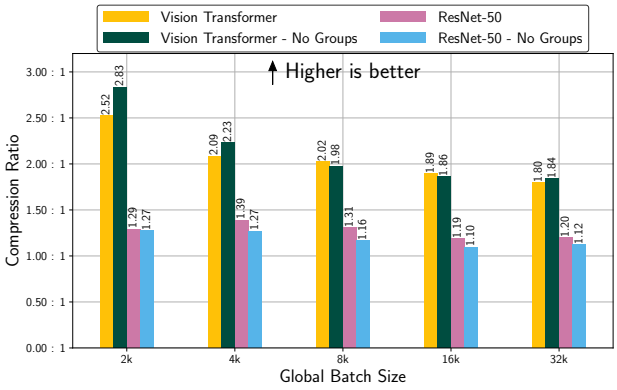
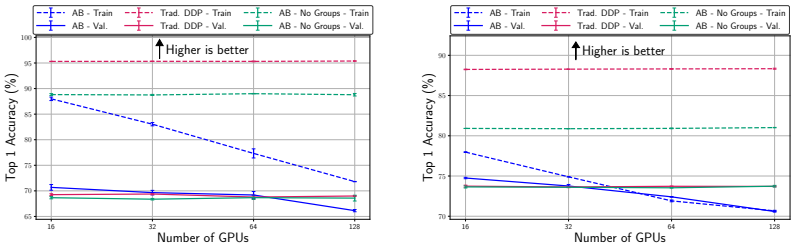


Figure 5.5.: The average compression ratios for *AB* and baseline models trained on ImageNet-2012 for two network architectures with a constant *local* batch size of 256.



(a) Vision Transformer B/16

(b) ResNet-50

Figure 5.6.: The highest top-1 accuracy for each training run on ImageNet-2012 for two network architectures with a constant *global* batch size of 4,096. Error bars are plotted, though not always visible.

5.4. Discussion

To evaluate the impact of *AB* training on network traffic, I define the scaled network traffic (T) as

$$T = \frac{p \cdot s}{t_b} \tag{5.9}$$

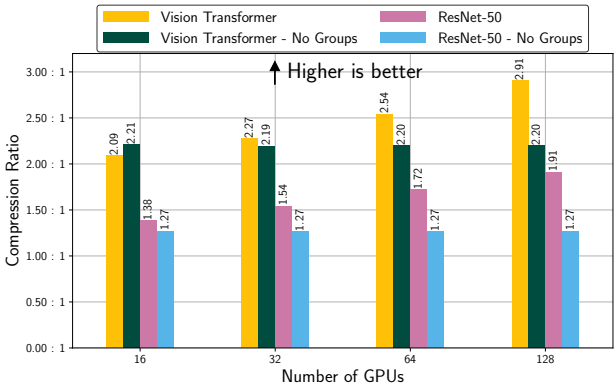
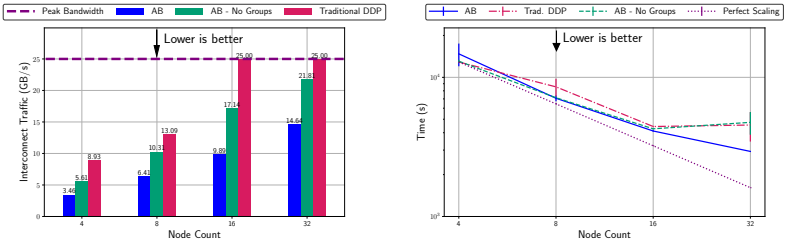


Figure 5.7.: The compression ratios for *AB* and baseline trained models on ImageNet-2012 for two network architectures with a constant *global* batch size of 4,096.



(a) Scaled interconnect traffic (Equation (5.9)). The 'Peak Bandwidth' is that of each port of each node's Infiniband card.

(b) Training wall-clock time.

Figure 5.8.: Scaled interconnect traffic (Equation (5.9)) and job wall-clock time for the ViT B/16 trained on ImageNet-2012.

where p is the memory required to store the trainable parameters during training in GB, s is the number of synchronizations per epoch, and t_b is the proportion of training spent in the backward pass (measured to be 0.54). The results consistently demonstrate a significant reduction in scaled network traffic, averaging 70.31% across all scaling tests (Tables 5.1 and 5.2), highlighting the effectiveness of AB training in mitigating the communication

Table 5.1.: Results from the constant *local* batch size scaling experiments. Scaled traffic reports the scaled interconnect traffic, as shown in Equation (5.9). The scaled interconnect traffic is limited to the bandwidth available on each node, 25 GB/s. Compression results show the final model compression ratio to the full-rank model. Time to train shows job wall-clock time. Bold values indicate the most favorable results between *AB*, *AB* - No Groups, and traditional DDP training.

ViT B/16				
Batch Size		AB	AB - No Groups	Traditional DDP
2k	Scaled Traffic, GB/s	**2.81 ± 0.08**	5.14 ± 0.19	8.87 ± 0.00
	Compression	2.52 : 1	**2.83 : 1**	1 : 1
	Time to Train, min	**435.80 ± 7.36**	442.94 ± 8.52	436.38 ± 0.35
	Validation Top-1, %	70.24 ± 0.36	68.31 ± 0.07	**70.73 ± 0.44**
4k	Scaled Traffic, GB/s	**2.83 ± 0.06**	5.72 ± 0.08	8.87 ± 0.10
	Compression	2.09 : 1	**2.22 : 1**	1 : 1
	Time to Train, min	**220.85 ± 0.88**	224.35 ± 1.22	249.03 ± 31.01
	Validation Top-1, %	**70.90 ± 0.44**	67.81 ± 0.49	69.15 ± 0.00
8k	Scaled Traffic, GB/s	**2.65 ± 0.04**	5.93 ± 0.13	8.51 ± 0.00
	Compression	**2.02 : 1**	1.98 : 1	1 : 1
	Time to Train, min	**115.65 ± 1.37**	121.95 ± 1.95	121.78 ± 1.02
	Validation Top-1, %	**68.92 ± 0.10**	65.70 ± 0.14	67.20 ± 0.07
16k	Scaled Traffic, GB/s	**2.54 ± 0.03**	4.91 ± 0.67	8.25 ± 0.18
	Compression	**1.86 : 1**	**1.86 : 1**	1 : 1
	Time to Train, min	64.83 ± 1.96	79.15 ± 9.31	**63.53 ± 1.10**
	Validation Top-1, %	**64.20 ± 0.01**	61.17 ± 0.33	63.68 ± 0.09
32k	Scaled Traffic, GB/s	**2.27 ± 0.41**	4.63 ± 0.51	7.52 ± 0.36
	Compression	1.80 : 1	**1.84 : 1**	1 : 1
	Time to Train, min	39.53 ± 5.29	43.36 ± 3.90	**36.56 ± 0.19**
	Validation Top-1, %	55.54 ± 0.34	53.89 ± 0.15	**58.00 ± 0.17**
ResNet-50				
2k	Scaled Traffic, GB/s	**1.11 ± 0.00**	2.87 ± 0.01	3.83 ± 0.01
	Compression	**1.29 : 1**	1.27 : 1	1 : 1
	Time to Train, min	368.15 ± 0.80	**364.91 ± 0.53**	367.36 ± 0.36
	Validation Top-1, %	**75.67 ± 0.10**	74.42 ± 0.04	74.43 ± 0.06
4k	Scaled Traffic, GB/s	**1.04 ± 0.05**	2.67 ± 0.04	3.85 ± 0.01
	Compression	**1.39 : 1**	1.27 : 1	1 : 1
	Time to Train, min	200.69 ± 10.30	195.46 ± 1.30	**195.19 ± 0.59**
	Validation Top-1, %	**74.61 ± 0.06**	73.68 ± 0.12	73.67 ± 0.05
8k	Scaled Traffic, GB/s	**0.91 ± 0.03**	2.71 ± 0.02	3.71 ± 0.09
	Compression	**1.31 : 1**	1.16 : 1	1 : 1
	Time to Train, min	108.19 ± 4.94	**104.70 ± 0.52**	105.77 ± 1.69
	Validation Top-1, %	**73.66 ± 0.03**	73.04 ± 0.15	73.02 ± 0.10
16k	Scaled Traffic, GB/s	**0.87 ± 0.01**	2.56 ± 0.04	3.63 ± 0.23
	Compression	**1.19 : 1**	1.10 : 1	1 : 1
	Time to Train, min	**58.61 ± 0.07**	59.54 ± 0.68	59.27 ± 0.10
	Validation Top-1, %	**72.66 ± 0.04**	72.60 ± 0.10	72.32 ± 0.03
32k	Scaled Traffic, GB/s	**0.78 ± 0.01**	2.17 ± 0.04	3.00 ± 0.06
	Compression	**1.20 : 1**	1.12 : 1	1 : 1
	Time to Train, min	**36.91 ± 0.07**	37.48 ± 0.15	37.91 ± 0.73
	Validation Top-1, %	70.49 ± 0.05	71.38 ± 0.07	**71.47 ± 0.08**

bottleneck inherent in data-parallel training. These findings underscore the substantial bandwidth demands of distributed training, even with moderately sized models, a challenge further amplified with larger architectures.

Table 5.2.: Results from the constant *global* batch size scaling experiments. Scaled traffic reports the scaled interconnect traffic, as shown in Equation (5.9). The scaled interconnect traffic is limited to the bandwidth available on each node, 25 GB/s. Compression results show the final model compression ratio to the full-rank model. Time to train shows job wall-clock time. Bold values indicate the most favorable results between *AB*, *AB* - No Groups, and traditional DDP training.

Nodes		AB	AB - No Groups	Traditional DDP
		ViT B/16		
4	Scaled Traffic, GB/s	**2.49 ± 0.49**	5.61 ± 0.16	8.93 ± 0.09
	Compression	2.09 : 1	**2.21 : 1**	1 : 1
	Time to Train, min	259.44 ± 49.68	227.99 ± 5.29	**224.97 ± 1.79**
	Validation Top-1, %	**70.68 ± 0.56**	68.66 ± 0.22	69.26 ± 0.18
8	Scaled Traffic, GB/s	**4.89 ± 0.44**	10.31 ± 0.32	13.09 ± 1.92
	Compression	**2.27 : 1**	2.19 : 1	1 : 1
	Time to Train, min	**123.35 ± 5.81**	124.85 ± 1.92	153.03 ± 17.77
	Validation Top-1, %	**69.66 ± 0.44**	68.37 ± 0.15	69.38 ± 0.14
16	Scaled Traffic, GB/s	**8.17 ± 0.17**	17.14 ± 0.28	25.00 ± 0.31
	Compression	**2.54 : 1**	2.20 : 1	1 : 1
	Time to Train, min	77.68 ± 9.18	**74.26 ± 0.76**	76.95 ± 0.87
	Validation Top-1, %	**69.19 ± 0.68**	68.67 ± 0.34	68.79 ± 0.40
32	Scaled Traffic, GB/s	**9.84 ± 0.07**	21.81 ± 2.66	25.00 ± 2.55
	Compression	**2.91 : 1**	2.20 : 1	1 : 1
	Time to Train, min	**51.10 ± 0.29**	81.41 ± 19.36	77.99 ± 18.29
	Validation Top-1, %	66.13 ± 0.22	68.58 ± 0.57	**68.99 ± 0.10**
		ResNet-50		
4	Scaled Traffic, GB/s	**1.07 ± 0.01**	2.69 ± 0.03	3.83 ± 0.01
	Compression	**1.38 : 1**	1.27 : 1	1 : 1
	Time to Train, min	**194.64 ± 0.91**	195.76 ± 3.49	195.03 ± 0.74
	Validation Top-1, %	**74.75 ± 0.09**	73.65 ± 0.13	73.74 ± 0.13
8	Scaled Traffic, GB/s	**1.46 ± 0.18**	3.98 ± 0.41	4.96 ± 0.09
	Compression	**1.54 : 1**	1.27 : 1	1 : 1
	Time to Train, min	**135.98 ± 12.43**	131.76 ± 12.52	139.82 ± 2.00
	Validation Top-1, %	**73.76 ± 0.19**	73.60 ± 0.02	73.64 ± 0.09
16	Scaled Traffic, GB/s	**1.91 ± 0.27**	5.57 ± 0.09	6.09 ± 0.46
	Compression	**1.72 : 1**	1.27 : 1	1 : 1
	Time to Train, min	111.46 ± 12.41	**92.52 ± 1.78**	113.00 ± 7.23
	Validation Top-1, %	72.41 ± 0.04	73.54 ± 0.04	**73.72 ± 0.03**
32	Scaled Traffic, GB/s	**1.84 ± 0.42**	5.70 ± 0.03	6.49 ± 1.85
	Compression	**1.91 : 1**	1.27 : 1	1 : 1
	Time to Train, min	110.09 ± 28.61	**92.37 ± 3.35**	121.55 ± 31.04
	Validation Top-1, %	70.59 ± 0.08	**73.73 ± 0.10**	73.71 ± 0.04

In 13 out of 18 experiments, *AB* training achieved validation accuracy competitive with traditional DP training with identical hyperparameters and similar training durations (Figures 5.3 and 5.6). Removing independent training groups resulted in a noticeable decrease in validation accuracy (Figure 5.3a) despite comparable compression levels (Figure 5.5). This suggests that the performance benefits of AB training arise from the synergistic combination of low-rank representations, independent subgroup training, and the merging of the low-rank representations as $\overline{\boldsymbol{W}} \approx \overline{\boldsymbol{A}}\,\overline{\boldsymbol{B}}$.

Large-scale experiments revealed a complex relationship between communication efficiency and model accuracy. In the ViT experiments with 32 nodes and a constant global batch size (Figure 5.8), the classic communication bottleneck was observed for traditional data-parallel training, where training time did not decrease despite increased computational resources. However, *AB* training continued to reduce training time due to decreased communication demands, even with the added overhead of computing the SVD of all network weights. Notably, *AB* training either maintained or reduced training time compared to PyTorch's DDP across all scaling measurements (Table 5.1 and Table 5.2).

AB training consistently achieved favorable compression ratios (Figures 5.5 and 5.7, and Tables 5.1 and 5.2). For runs matching or exceeding baseline accuracy, compression ratios for the Vision Transformer ranged from 1.89:1 to 2.54:1. At the same time, ResNet-50 demonstrated more modest but still noteworthy ratios of 1.19:1 to 1.72:1. This variation suggests a potential interaction between model architecture and the effectiveness of *AB* training, warranting further investigation.

Table 5.3 compares *AB* training's performance to other low-rank and pruning methods on ResNet-50 (ImageNet-2012) and VGG16 (CIFAR10). On CIFAR10, *AB* training achieved a 44.14:1 compression ratio with negligible accuracy loss, surpassing the compression of ICP [197] and ABCPrune [198]. Notably, *AB* training was the only method to outperform the baseline in the ResNet-50 benchmark.

The information to calculate the scaled network traffic with Equation (5.9) is not available for other methods. To compare the communication efficiency of each training method, I estimate the maximum communication savings achievable.

As all training methods shown iteratively reduce the network size, I assume they begin with the traditional network before removing parameters with a linear decrease in parameter count during training. In my experiments, I found that once the network started to compress, it quickly removed large portions of parameters and then removed fewer parameters during the remaining training steps. However, as the parameter removal rate is unknown, I assume that models train close to full rank for 25% of training and close to their

final model state for the remaining 75%. With this assumption, I define the estimated communication reduction (ECR) metric as

$$\text{ECR} = 100\% - F - Lc, \tag{5.10}$$

where F and L are the percentages of training spent at the full-rank and most-compressed network state, respectively, and c is the final compression ratio as a fraction. For *AB* training, the percentage of training spent near the final compression state, L, is reduced by the percentage time spent in the independent group training phase.

Table 5.3.: Comparison of low-rank and pruning methods for ResNet-50 on ImageNet-2012 and VGG16 on CIFAR10. 'Difference to Baseline' indicates validation top-1 performance relative to the original full-rank model in each study, with positive values denoting improved predictive performance over the baseline. *AB* training used a global batch size of 4,096 for ImageNet and 1,024 for CIFAR10, achieving maximum top-1 accuracies of 75.67% and 91.87%, respectively. The estimated communication reduction (estimated communication reduction (ECR)) is defined by Equation (5.10). *AB* training's ECR assumes independent groups do not utilize the compute system's interconnect. OIALR's and DLRT's ECR use the compression of the trainable parameters as they report.

	Method	Difference to Baseline	Compression Ratio	ECR
ResNet-50 ImageNet-2012	*AB*	**+1.55 %**	1.39 : 1	**73.29 %**
	AB (no groups)	-0.02 %	1.27 : 1	15.94 %
	OIALR [184]	-1.72 %	1.21 : 1	63.64 %
	DLRT [129]	-0.56 %	1.85 : 1	64.35 %
	PP-1 [155]	-0.20 %	2.26 : 1	41.81 %
	CP [179]	-1.40 %	2.00 : 1	37.5 %
	SFP [180]	-0.20 %	2.39 : 1	43.62 %
	ThiNet [103]	-1.50 %	**2.71 : 1**	47.32 %
	ABCPrune [198]	-2.15 %	1.84 : 1	34.24 %
VGG16 CIFAR10	*AB*	-0.23 %	**44.14 : 1**	65.60 %
	AB (no groups)	-0.89 %	36.63 : 1	**72.95 %**
	OIALR [184]	**+0.10 %**	3.70 : 1	64.59 %
	DLRT [129]	-1.89 %	1.79 : 1	16.88 %
	ABCPrune [198]	+0.06 %	8.83 : 1	66.51 %
	ICP [197]	-0.31 %	27.25 : 1	72.25 %

Regarding the ECR, *AB* training demonstrates superior performance to all methods except OIALR and DLRT, which achieve compression by directly

training the singular value matrix (Σ). Notably, the most significant communication reduction is observed for the ResNet-50 benchmark. However, for VGG16 trained on CIFAR10, both ICP [197] and the "*AB* Training (No Group)" variant exhibit higher estimated communication reductions than standard *AB* training. This suggests that while *AB* training offers excellent communication efficiency, its advantage diminishes at extremely high compression levels. These findings are corroborated by measured network traffic reductions, which averaged 70.13% across all tested scenarios, aligning closely with the estimated ECR.

The results reveal a complex relationship between low-rank representations, large batch sizes, and generalization. Significantly improved generalization was observed in all experiments, likely due to the regularization effect of low-rank representations, as evidenced by its presence in the "No Groups" measurements shown in Figure 5.6. However, accuracy degrades at larger scales, particularly with a constant local batch size (Figure 5.3). This degradation, coupled with decreasing compression ratios as the global batch size increases (Figure 5.5), suggests challenges in maintaining the effectiveness of averaging independently trained subgroups at extreme scales.

Specifically, I hypothesize that the generalizable patterns represented by the singular values and their corresponding vectors in the center of the singular value distribution are not consistently learned across all worker groups. Consequently, when the independently trained models are merged, their presence in the full-rank representation diminishes, leading to lower accuracy and compression ratios at large scales. Similar trends in the fixed global batch size scenario (Figures 5.6 and 5.7) further support this hypothesis.

The increasing compression and decreasing accuracy as the local batch size decreases suggest an incompatibility between the large global batch size during full-rank training and the smaller batches used during independent training. This likely leads to divergences in smaller singular values across groups, which are diminished during model aggregation.

While *AB* training demonstrates significant potential, these challenges highlight the need for further research into tailored model merging strategies and adaptive learning rate schedules. Investigating more alternative update mechanisms, such as non-average or loss-weighted averaging schemes, could mitigate the negative impacts of large batch sizes in this context.

5.5. Conclusion and Outlook

My experimental results highlight the significant potential of *AB* training to reduce communication overheads in distributed deep learning by leveraging low-rank representations and independent training of worker subgroups. The consistent 70% reduction in network traffic achieved across various models and datasets opens up new possibilities for training in environments with limited network bandwidth. This efficiency gain could prove invaluable for the research and development of larger, more complex neural networks.

Furthermore, the pronounced regularization effects observed with *AB* training, particularly at smaller scales, offer a promising avenue for improving generalization and model performance. The reduced overfitting demonstrated in my experiments underscores the potential benefits of integrating low-rank methods into standard training pipelines.

However, the challenges encountered at extreme scales, particularly the degradation in accuracy and compression efficiency, emphasize the complex interplay between low-rank representations, large batch effects, and hyperparameter optimization. These findings underscore the need for further research into tailored hyperparameter strategies that can effectively navigate the unique optimization landscape of low-rank, distributed training. The development of novel update mechanisms, such as non-average or loss-weighted averaging schemes, could also prove fruitful in mitigating the negative impacts of large batch sizes in this context.

While this work represents a significant step towards communication-efficient distributed training, it also reveals the importance of understanding the intricate relationship between model architecture, hyperparameters, and training dynamics. In the following chapter, I will delve deeper into the challenges and opportunities of hyperparameter optimization and neural architecture search, aiming to unlock its potential and pave the way for even more efficient and scalable deep learning workflows.

6. Tuning Training Methods by Choosing Better Hyperparameters

The content of this chapter is based on:

D. Coquelin, R. Sedona, M. Riedel, et al. "Evolutionary Optimization of Neural Architectures in Remote Sensing Classification Problems". In: 2021 IEEE International Geoscience and Remote Sensing Symposium IGARSS. 2021, pp. 1587–1590. DOI: 10.1109/IGARSS47720.2021.9554309

The previous chapters have delved into various methods for training and optimizing neural networks, including distributed training, low-rank approximations, and hybrid approaches. Each of these introduced its own set of non-learnable hyperparameters (HPs), which, combined with those governing the network architecture, optimizer, and learning rate scheduler, create a vast and complex configuration space to navigate.

The hyperparameters, established prior to the onset of training, exert a profound influence on both the training process and the predictive capabilities of the resultant model. The evaluation of individual configurations lends itself naturally to an embarrassingly parallel paradigm, highly amenable to exploitation within distributed-memory environments. This inherent parallelism empowers hyperparameter optimization (HPO) packages to effectively harness distributed-memory architectures, contingent upon the efficient generation of new hyperparameter combinations.

The task of HPO involves tuning hyperparameters to maximize a model's performance on a specified metric, such as validation accuracy. This is often a formidable challenge, as the interplay between parameters can be complex and highly dependent on the specific task, dataset, and model architecture.

For instance, while the learning rate is typically considered a critical hyperparameter, its ideal value has been shown to depend on the global batch size [199]. This exemplifies the broader challenge: hyperparameters are highly interdependent, making discovering optimal model configurations difficult.

This challenge lies at the heart of Neural Architecture Search (NAS), a subfield of HPO that seeks to automate the discovery of well-performing model architectures. While manual tuning based on expertise and intuition remains valuable, modern neural networks' sheer scale and complexity necessitate systematic, automated, and efficient approaches.

The search space for HPO is often vast, high-dimensional, and non-convex, characterized by hidden correlations and interactions among parameters that are difficult to discern. Evolutionary algorithms, which breed and mutate sets of hyperparameters, have shown promise in effectively navigating this complex landscape and discovering superior configurations [101].

In this chapter, I delve into the workings of `Propulate` [101], a robust asynchronous evolutionary optimization framework designed to tackle HPO in distributed settings. I will situate `Propulate` within the broader landscape of HPO method, showcase its application for image classification on the BigEarthNet [200] dataset, and discuss its extension to multi-rank worker scenarios, something particularly important for HPC environments.

6.1. Background and Related Work

Hyperparameter optimization is a long-standing challenge in machine learning with roots in the broader field of optimization. Simple approaches often involve manual trial and error, or systematic but computationally expensive grid search. Grid search's inherent limitations, particularly its inability to efficiently explore high-dimensional hyperparameter spaces [201], led to developing more sophisticated methods.

A simple yet surprisingly effective method is random search. This approach randomly samples hyperparameter configurations from a predefined distribution, offering a computational advantage over grid search. Given sufficient random sampling, random search can also find more optimal configurations than grid search [98].

As evaluating each hyperparameter set requires at least a partial model training, state-free methods like grid and random search can exceed available compute resources. To efficiently explore high-dimensional hyperparameter spaces, state-dependent strategies, like Bayesian and evolutionary optimization, use information from previous evaluations to guide the search and find the best possible combination with fewer evaluations.

Bayesian optimization, which has emerged as a prominent methodology in HPO [97], constructs a probabilistic model (typically a Gaussian process) of the objective function, which maps hyperparameter configurations to their corresponding model performance. The model is updated iteratively based on the results of past evaluations, guiding the algorithm to select new configurations that balance the exploration of the search space with the exploitation of promising regions. This approach can be particularly advantageous when evaluations are computationally expensive, as it aims to minimize the number of function evaluations required to find high-performing solutions. The Optuna framework [202] is a notable implementation of Bayesian optimization, offering flexibility and ease of use. One of the significant benefits of Bayesian optimization is the inherently existing importance metrics. Theoretically, these denote how much a given hyperparameter influences the objective function's output. However, this approach struggles when its core assumptions break down, e.g., when some HPs significantly affect others non-linearly [101]. This can occur in neural architecture search (NAS), where the network structure is part of the search space.

Inspired by natural evolution, evolutionary optimization algorithms provide a powerful alternative to Bayesian optimization. These algorithms maintain a population of candidate solutions, each representing a specific hyperparameter configuration. The population is iteratively improved through processes analogous to biological selection, crossover, and mutation. Candidate solutions are known as individuals. Each individual's fitness is evaluated using a predefined metric, such as validation accuracy. The most successful individuals are selected to generate offspring by combining the hyperparameters from two parents (crossover) and randomly perturbing some hyperparameters (mutation). This process repeats, gradually guiding the population towards more optimal solutions.

Evolutionary algorithms are metaheuristics, meaning they lack theoretical convergence or optimality guarantees. Their stochastic nature can lead to considerable variability in results, requiring multiple runs to ensure robustness.

However, they also offer several advantages. Firstly, evolutionary algorithms are well-suited for navigating complex, non-convex, high-dimensional search spaces, a common characteristic of HPO problems [203]. They can also dynamically adapt to the search space's characteristics and the feedback from evaluations, adjusting their search strategies as needed [204]. Furthermore, evolutionary algorithms can readily parallelize the evaluation of individuals, making them well-suited for large-scale, distributed HPO [205].

`Propulate` [101] is a state-of-the-art evolutionary optimization framework designed for massively parallel HPO at scale. Evolutionary algorithms have outperformed Bayesian optimization methods in various scenarios, achieving faster convergence and superior predictive performance [101]. Furthermore, `Propulate`'s asynchronous nature makes it well-suited for distributed environments, enabling efficient parallelization of the evaluation process.

One of the challenges in distributed HPO lies in efficient information sharing. While each worker can independently evaluate hyperparameter configurations, they must all share and utilize the results of these evaluations to inform the next generation of candidate solutions. A common approach uses a centralized database to store and aggregate proposed hyperparameters and their corresponding results [202]. However, this can lead to a file-locking bottleneck when many workers attempt to access the database simultaneously, a situation often encountered in HPC environments. Mitigating this bottleneck requires careful design of the communication and coordination mechanisms within the HPO framework, ensuring that information exchange does not become a limiting factor in the search process.

6.1.1. Propulate's Evolutionary HPO

`Propulate` is an evolutionary optimization framework, particularly designed for hyperparameter optimization. It leverages the principles of natural selection, crossover, and mutation to guide the search process toward optimal hyperparameter configurations.

Typically, the process begins with a population of randomly initialized hyperparameter sets, each representing an individual. These individuals are evaluated by training neural networks with the corresponding hyperparameter configurations and assessing their performance on a validation set. The most successful individuals are then selected to form a mating population.

New offspring are generated through crossover, combining hyperparameters from two parents, and mutation, randomly perturbing some hyperparameters. These new offspring are evaluated, and the process repeats for multiple generations or until a desired performance level is reached.

`Propulate` employed a simplified version of this process in its initial versions, with a single dedicated process managing the population and generating offspring. However, recent developments have seen the adoption of an island-based model with an improved communication strategy, offering enhanced exploration and scalability.

The island model divides the population into multiple subpopulations, each residing on a separate "island." Each island independently evolves its subpopulation, with workers evaluating individuals and generating offspring. Periodically, the islands exchange their best individuals. The goal of migration is to introduce genetic diversity and prevent premature convergence. This decentralized approach promotes a balance between exploration and exploitation, as each island explores a different search space region while benefiting from the occasional influx of potentially superior solutions from other islands.

`Propulate` uses asynchronous propagation of continuous populations. It softens the typically strict separation of generations to reduce the computational overhead caused by synchronous evaluations. In each iteration, each worker breeds and evaluates a new individual and informs all other workers on the same island about its result. It then receives individuals evaluated by others for a mutual update. Afterward, asynchronous migration happens between islands with a certain probability. In the next generation, the worker breeds a new individual from the continuous population of all evaluated individuals from any generation on its island. The workers thus proceed asynchronously without idle times despite the individuals' varying evaluation times.

6.1.2. BigEarthNet

For my NAS experiments, I used the BigEarthNet [206] dataset. BigEarthNet is a large-scale remote-sensing dataset containing patches extracted from 125 Sentinel-2 tiles (Level2A) acquired from June 2017 to May 2018 [206]. The archive comprises 590,326 patches, each is assigned one or more of the 19 available labels. The label nomenclature is an adaptation of the CORINE

Land Cover [207] consisting of labels from ten European countries updated in 2018 [200]. Each patch has twelve spectral bands at various resolutions:

- three RGB bands and band eight at 10 m resolution (120 by 120 pixels)
- bands 5, 6, 7, 8a, 11, and 12 at 20 m resolution (60 by 60 pixels)
- bands 1 and 9 at 60 m resolution (20 by 20 pixels)

I omitted the cirrus-sensitive band ten and patches covered with snow or clouds [208]. Figure 6.1 shows example patches.

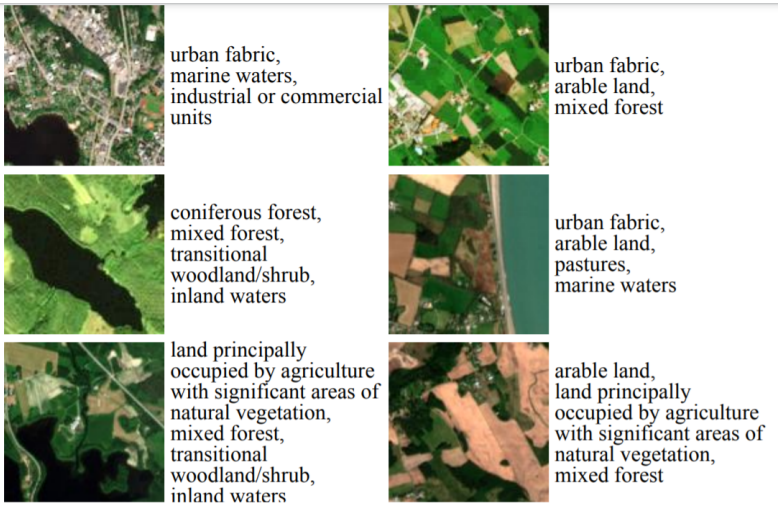


Figure 6.1.: Example patches and labels for Sentinel-2 tiles [200].

In Sumbul et al. [200], researchers trained multiple network types for multi-class classification of the BigEarthNet dataset, achieving varying degrees of success. Those experiments excluded bands 1 and 9, utilized the Adam optimizer, the sigmoid cross-entropy loss, a learning rate of 0.001, and were trained for 100 epochs.

The models were evaluated with the F_1 score, a metric that measures the accuracy of a model by considering both precision (the ability to avoid false positives) and recall (the ability to find all relevant instances). In a multi-class classification problem, the micro-F_1 score calculates the F_1 score globally by counting the total true positives, false negatives, and false positives across

all classes, treating all classes equally [209]. Conversely, the macro F_1 score calculates the average F_1 score of each class independently and then takes the mean, giving equal weight to all classes. In the evaluation of the ResNet-50 on BigEarthNet, the micro-F_1 score of 77.11 indicates good overall accuracy across all classes. In contrast, the macro-F_1 score of 67.33 suggests potential variability in performance across different classes, with possible lower accuracy on poorly represented classes.

6.2. Experiments

To test `Propulate`'s single-island version and its NAS performance, I tasked it with optimizing a ResNet-50-type model for multi-label classification of the BigEarthNet dataset.

I systematically divided the hyperparameter search space into six categories: optimizers, learning rate schedulers, activation functions, loss functions, the number of filters per convolutional block, and the activation order. Table 6.1 details the specific options explored within each category, excluding the number of filters. For this network, the number of filters in each convolutional block is determined by a fixed ratio relative to the number of filters in the first block. The search space also included the learning rate scheduler's and optimizer's parameters. The order of activation, batch normalization (BN), and convolution layers within residual building blocks can affect the accuracy of a network [210]; therefore, I included these options in the search space as well. I will use the test configurations in Figure 6.2 when referring to the activation order.

The search space for the number of filters refers to the number of filters in the first convolutional block and ranges from 2 to 256. The activation functions considered were Exponential Linear Unit (ELU), exponential, hard sigmoid, linear, Rectified Linear Unit (ReLU), Scaled Exponential Linear Unit (SELU), sigmoid, Softmax, Softplus, Softsign, Swish, and hyperbolic tangent. The loss functions included binary cross-entropy, categorical cross-entropy, categorical hinge, hinge, Kullback-Leibler divergence, and squared hinge. The selection of these functions was based on their prevalence within the machine learning community [12].

Table 6.1.: Neural architecture and hyperparameter search space. The Activation Function column shows the activation functions. Figure 6.2 shows detailed activation orders. ELU is the exponential linear unit, ReLU is the rectified linear unit, SELU is the scaled exponential linear unit, K-L divergence is the Kullback-Leibler divergence, and `tanh` is the hyperbolic tangent.

Optimizer [211, 212]		
Adadelta	AMSGrad	Nadam
Adagrad	Adamax	RMSprop
Adam	Ftrl	SGD
Activation functions [12]		
ELU	ReLU	Softplus
Exponential	SELU	Softsign
Hard sigmoid	Sigmoid	Swish
Linear	Softmax	tanh
Activation order [210]	**Loss [213, 214, 215]**	**LR scheduler [216]**
Original	Binary cross-entropy	Exponential decay
BN after addition	Categorical cross-entropy	Inverse time decay
Activation before addition	Categorical hinge	Polynomial decay
Activation-only pre-activation	Hinge	
Full pre-activation	K-L divergence	
	Squared hinge	

Data preparation followed the methodology outlined in Sumbul et al. [200], but without image augmentation. The network was implemented in TensorFlow [49]. Experiments were conducted on varying numbers of NVIDIA V100 GPUs on the ForHLR 2 cluster at KIT. Early stopping was employed to terminate training if the validation loss did not improve for ten consecutive epochs.

During the initial NAS runs, I observed that hyperparameter combinations involving the Adam, Adamax, Nadam, and RMSprop optimizers frequently resulted in training instability, leading to not-a-number (NaN) values. This instability is likely due to the interaction between these optimizers' adaptive algorithms and specific loss functions or hyperparameter values. To address this, I conducted separate NAS runs for each of these optimizers, isolating them from the broader search space to prevent their premature exclusion from the population due to instability.

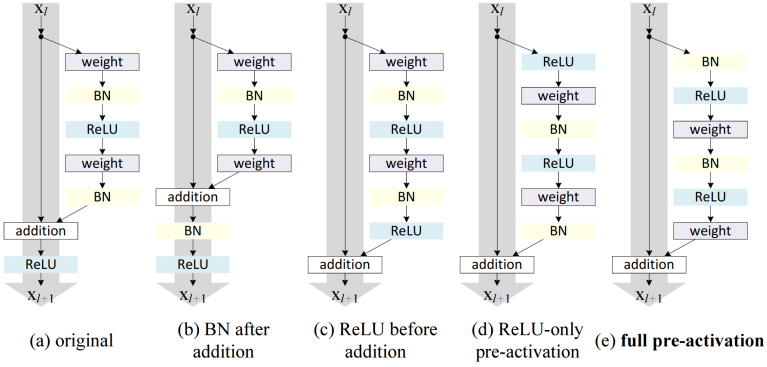


Figure 6.2.: The various orders of the activation, batch normalization, and convolution layers within residual building blocks used in a network. During the NAS, the activation function is defined by the hyperparameters [210].

6.3. Discussion

To gain insights into the effectiveness of different hyperparameters, I analyzed their relative selection frequency across all completed NAS runs. This frequency is a proxy for stability and accuracy, as poorly performing networks are eliminated from the population.

Regarding selection frequency, binary cross-entropy and categorical hinge were the most successful loss functions, while hinge, Kullback-Leibler divergence, and squared hinge were the least effective. For the learning rate schedulers, exponential decay, inverse time decay, and polynomial decay exhibited similar frequencies, suggesting that all three can be effective choices when appropriately parameterized.

The analysis of activation functions revealed a correlation between their effectiveness and the chosen optimizer. For instance, Adamax performed best with ELU, while Adadelta favored Softmax. Interestingly, the most frequently selected activation function was the linear function, likely due to its initial stability across various optimizers. During the initial optimization stages, the algorithm most often chose the linear function before gradually replacing it with more effective alternatives as the search progressed.

Table 6.2.: Class-level F_1 scores for the found network, ResNet-50, and the best results per class in Sumbul et al. [200]. Class names are abbreviated; for the full class name, see Sumbul et al. [200].

Class	Found	Original
Urban fabric	**75.86**	74.84
Industrial or...	45.79	**48.55**
Arable land	**84.82**	83.85
Permanent crops	**59.63**	51.91
Pastures	**74.54**	72.38
Complex cultivation...	**69.00**	66.03
Land principally...	**65.37**	60.94
Agro-forestry areas	**77.35**	70.49
Broad-leaved forest	**77.27**	74.05
Coniferous forest	**86.25**	85.41
Mixed forest	**82.15**	79.44
Natural grassland...	**47.77**	47.55
Moors, heathland...	**64.46**	59.41
Transitional woodland...	**64.72**	53.47
Beaches, dunes, sands	44.09	**61.46**
Inland wetlands	**61.99**	60.64
Coastal wetlands	**57.31**	47.71
Inland waters	**85.53**	83.69
Marine waters	**97.93**	97.53

Like the linear activation function, the initial NAS stages predominantly favored a small number of filters (four). As the search continued, the preferred number of filters shifted towards 32 or 64. Regarding activation order, the full pre-activation configuration (Figure 6.2) consistently emerged as the most effective.

Due to the separate searches, the frequency-based analysis was less informative for the Adam, Adamax, Nadam, and RMSprop optimizers. Instead, I assessed their effectiveness solely based on the F_1 scores achieved by the resulting networks.

Ftrl did not consistently yield competitive results and was gradually eliminated from the population in the general NAS run. Notably, the separate searches for Adamax, RMSprop, and Adam produced networks that were

competitive with the collective NAS results, suggesting that excluding these optimizers due to instability concerns was unwarranted from a predictive performance perspective.

Adadelta and SGD emerged as the most effective optimizers, each generating multiple configurations that met or exceeded the target micro-F_1 score of 77.11. The best-performing configuration achieved a micro-F_1 score of 77.25 and a macro-F_1 score of 69.57 in just ten epochs, surpassing the accuracy and significantly reducing the training time compared to the previously reported results [200].

A detailed analysis at the class level, Table 6.2, revealed that the discovered network notably outperformed the baseline models for certain classes, particularly coastal wetlands, moors, heathland and sclerophyllous vegetation, and agro-forestry areas. However, performance on the beaches, dunes, and sands class was significantly worse. These discrepancies are attributed to the inclusion of additional spectral bands in the training data, highlighting the potential impact of data representation on class-level accuracy.

My experiments on BigEarthNet showcase the efficacy of evolutionary NAS in automating the discovery of optimal model architectures and hyperparameters for remote sensing image classification. The superior performance of the discovered architecture compared to manually tuned baselines demonstrates the potential of this approach to improve classification accuracy while significantly reducing the time and effort required for hyperparameter tuning.

The observed variations in performance across different optimizers and activation functions highlight the complex and often unpredictable interplay between these components within the broader HPO landscape. The emergence of the linear activation function as a frequent choice in early NAS stages, followed by a gradual shift towards more specialized activations like ELU and Softmax, suggests a potential strategy for balancing exploration and exploitation during the search process.

However, excluding unstable optimizers early in the search process underscores the challenges inherent to HPO. The final results were influenced by the initial stability of the search space rather than solely by the optimizers' ultimate potential for achieving high performance.

The class-level analysis reveals the nuanced impact of data representation and feature selection on model performance. While the discovered architecture excelled in classifying certain land cover types, its performance on others was

notably worse than the baseline. This observation emphasizes the importance of carefully considering the specific characteristics of the dataset and task when designing and optimizing neural network architectures.

6.4. Multi-Rank Workers

A crucial step in HPO is distributing hyperparameter configurations to workers for evaluation. This process is straightforward when each worker uses a single MPI rank and GPU. However, when an individual's evaluation requires multiple ranks and GPUs, coordinating the distribution and evaluation of hyperparameters becomes more intricate. This functionality, absent in early versions of `Propulate`, is vital for effectively exploring hyperparameter spaces for multi-rank worker configurations, e.g. the data-parallel training of neural networks.

Initially, `Propulate` assumed a one-to-one correspondence between workers and MPI ranks (Figure 6.3a). This assumption simplified the initial implementation but limited its applicability for multi-rank scenarios. Therefore, the concept of a worker was broadened to encompass any number of sub-workers to accommodate these cases. This change results in so-called multi-rank workers (Figure 6.3b), and coordinated action is required among the ranks of each worker.

To accommodate these cases, the concept of a worker was broadened to encompass any number of sub-workers, resulting in so-called multi-rank workers (Figure 6.3b). This requires coordinated action among the ranks within each worker.

To achieve these modifications, I made the following changes to `Propulate`:

1. Workers were redefined as MPI groups, enabling efficient communication among ranks within a single worker.
2. I created additional MPI groups comprising each island's multi-rank workers' rank 0 processes. As illustrated in Figure 6.3b, rank 0 of worker 8, for instance, governs 'GPU:15'. This facilitates communication and coordination on an effective intra-island multi-rank worker level.

3. A separate MPI group, consisting of all multi-rank workers' rank 0 processes across the entire MPI world, was established. This group mirrors `Propulate`'s original communication structure and is used for migrating individuals between islands.
4. Only the designated rank 0 process of each worker actively participates in the propagation of individuals to guarantee uniformity in hyperparameter configurations across all ranks within a worker.

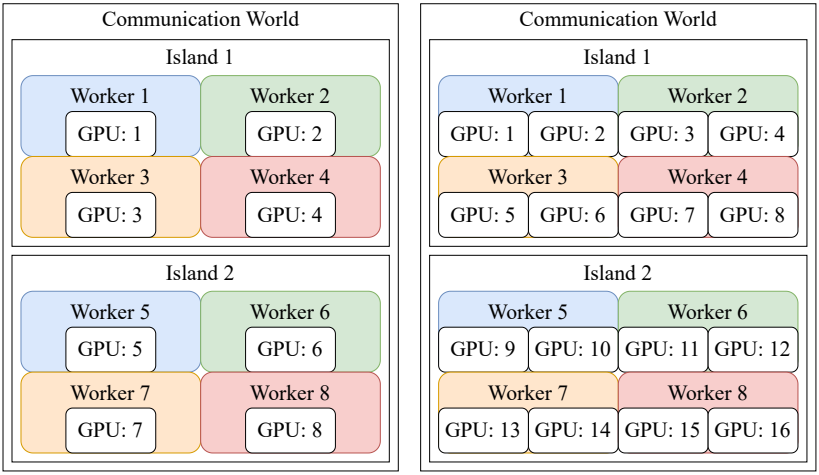
Within each multi-rank worker group, solely the rank 0 worker is responsible for sampling individuals from the population and disseminating them to the remaining ranks within the group. The other ranks contribute exclusively to the evaluation of these individuals, playing no direct role in the optimization of the HPs.

These modifications extend `Propulate`'s capabilities to effectively manage multi-rank workers. By enabling coordinated evaluation and communication within and across worker groups, the framework can explore hyperparameter spaces for a broader range of distributed training scenarios. This has been used successfully in the optimization of the *AB* training method presented in Chapter 5.

6.5. Conclusion

My work presented in this chapter demonstrates the potential of evolutionary HPO to automate and optimize NN architectures. When properly configured, evolutionary HPO can uncover hyperparameter combinations that lead to accelerated convergence rates and improved accuracy, as shown on the BigEarthNet dataset and in the previous two chapters. However, my findings also underscore the importance of carefully considering the stability and interactions of hyperparameters within the search space. As demonstrated by the need for separate HPO runs for specific optimizers, unstable configurations can unduly bias the search process.

Notably, the evolutionary HPO approach yielded an architecture that surpassed the accuracy of manually tuned models and converged significantly faster. The combination of Adadelta optimizer, polynomial learning rate decay, Softmax activation, and binary cross-entropy loss proved particularly



(a) Original worker design **(b)** Multi-rank worker design

Figure 6.3.: A simple overview of how `Propulate` modeled workers when first implemented, (a), and how the same number of workers and islands looks when workers each have two accelerators and two ranks (b). In this diagram, each GPU is controlled by a single, unique rank.

effective for the presented use-case of BigEarthNet classification. Furthermore, including additional spectral bands led to substantial accuracy gains for several land cover classes, showcasing the importance of data representation.

However, the inherent limitations of any HPO approach must be acknowledged. The defined search space bounds search effectiveness, and while encompassing commonly used functions, my current work did not explore cutting-edge or highly specialized techniques. Achieving substantial performance gains beyond those demonstrated here requires more advanced network architectures and loss functions tailored to the specific nuances of remote sensing data.

My findings illuminate the potential of automated methods for hyperparameter and architecture optimization in remote sensing applications. The discovered architecture and hyperparameter configurations offer a valuable starting point for further research and practical deployment of deep learning models in this domain.

In the broader context of my thesis, this chapter's results reinforce the theme that understanding and leveraging the intricacies of neural network training can lead to significant advancements in model performance and computational efficiency. While I have focused on hyperparameter tuning and architecture search here, my earlier explorations of low-rank representations and distributed training emphasize the multifaceted nature of optimizing deep learning workflows.

The final chapter will present the culmination of these investigations, where I synthesize my findings and discuss their implications for deep learning research and applications in general.

7. Conclusion

In this thesis, I investigated the challenges and opportunities inherent to training large-scale neural networks in distributed-memory environments. The fundamental necessity of distributed training, driven by the ever-increasing size and complexity of contemporary models and datasets, was firmly established. Subsequently, a deep dive into the core principles of data parallel (DP) training, the dominant paradigm for distributed training, was conducted, highlighting its advantages and limitations. Particular emphasis was placed on the communication bottleneck, a critical challenge that arises from the frequent synchronization of model parameters.

7.1. Key Findings and Contributions

The Distributed, Asynchronous, and Selective Optimization (DASO) method, presented in Chapter 3, showcased the potential for reducing training time by over 25% compared to state-of-the-art methods at the time. DASO's success stemmed from its topology-aware, hierarchical synchronization structure, which leveraged node-local communication and asynchronous updates with stale gradients. This demonstrated that global synchronization after every forward-backward pass is not strictly necessary for effective training under the assumption of iid data items. Furthermore, the results highlighted the potential of stale gradients in accelerating training and the effectiveness of topology-aware communication in reducing overhead.

To understand the underlying reasons for the surprising resilience of neural networks to stale gradients, I investigated the dynamics of weight matrices during training at small scales in Chapter 4. I observed a consistent stabilization of the weights' orthogonal bases early in the training process. This insight led to the development of Orthogonality-Informed, Adaptive Low-Rank (OIALR) training. This novel method freezes the bases $\boldsymbol{U}$ and $\boldsymbol{V}$,

as determined by the SVD of the weights, and trains only on the singular value matrix, Σ. While OIALR initially exhibited a slight decrease in accuracy compared to full-rank training, hyperparameter optimization revealed its potential to surpass full-rank performance in accuracy and training time. This highlighted the importance of tailored hyperparameter configurations for low-rank training methods as well as their potential for improved generalization.

To leverage the inherent robustness of neural networks to delayed network updates, topology-aware communication patterns, and the stabilization observed in the orthogonal components of network weights, I introduced the *AB* training method in Chapter 5. This novel approach integrated hierarchical synchronization and low-rank matrix approximations, enabling subgroups of workers to train independently on distinct components of decomposed weight matrices. Across a range of experiments and models, *AB* training consistently achieved over 70% reduction in scaled network communication, while matching or outperforming traditional DP training in terms of predictive accuracy without hyperparameter tuning. However, challenges in preserving performance at large scales were noted, particularly arising from the complex interactions between low-rank representations, large batch sizes, and the merging of independently trained models.

This thesis investigated neural network optimization, particularly focusing on distributed training and proposing novel methodologies. These methodologies, however, introduced new hyperparameters requiring tuning. For DASO, a grid search was employed due to its simplicity, but even with a limited search space, it proved computationally expensive, highlighting the potential for further optimization. For OIALR and AB training, the power of evolutionary algorithms, specifically leveraging the `Propulate` framework, was showcased for automated hyperparameter discovery. This approach successfully and efficiently identified robust hyperparameters, underscoring the critical role of automated hyperparameter optimization in unlocking the full potential of complex deep learning techniques. Such advancements enable researchers to push the boundaries of model performance and efficiency.

7.2. Revisiting Research Questions

Distributed Hyperparameter Optimization

New methods come with different HPs. How can the proper hyperparameters for new methods be found to maximize performance in a distributed setting?

My exploration of evolutionary HPO with `Propulate` in Chapter 6 highlights its effectiveness as a powerful tool for automating the discovery of high-performing neural network architectures and hyperparameters in distributed environments. The successful application of `Propulate` in the OIALR and *AB* training methods further validates its versatility and potential to advance the state of the art in deep learning. The ability to efficiently search vast, high-dimensional spaces and adapt to the unique characteristics of different datasets and tasks is crucial for continued progress in this field.

However, several open questions remain. Developing more efficient search algorithms, incorporating multi-objective optimization to balance accuracy with other factors like model size or inference speed, and exploring the transferability of learned architectures to diverse domains are all promising avenues for future investigation.

Exploiting Low-Rank Representations

To what extent can low-rank representations reduce the computational and communication requirements of distributed neural network training?

My work demonstrates the efficacy of low-rank representations in reducing the computational and communication demands of distributed neural network training. OIALR and *AB* training achieved noteworthy compression ratios, ranging from modest reductions to a remarkable $44.14 : 1$ compression in an idealized scenario. This highlights the potential for low-rank methods to alleviate the communication bottleneck and facilitate the training and deployment of larger models on resource-constrained devices.

Efficiency in Distributed-Memory Training

Can distributed training algorithms be designed to better balance computational and communication efficiency while scaling to accommodate the demands of ever-larger neural networks and datasets?

Hierarchical training methods, coupled with low-rank representations, proved highly effective in improving communication efficiency for data-parallel neural network training. *AB* training and DASO demonstrated substantial reductions in network traffic (more than 70% for *AB* training) and training time (more than 25% for DASO), validating the effectiveness of asynchronous updates and localized synchronization in mitigating communication bottlenecks.

However, achieving optimal efficiency in distributed training remains a complex challenge. My work highlighted the importance of carefully balancing communication and computation to avoid bottlenecks and maximize resource utilization. Future research could explore adaptive synchronization schemes, where synchronization frequency is dynamically adjusted based on factors like model convergence and network bandwidth, to further optimize training efficiency.

Understanding and Addressing Large Batch Effects

Can novel optimization methods or training regimes be developed to counteract the detrimental effects of large batch training, leading to performance gains and improved generalization in large-scale settings?

My research illuminates the intricate relationship between large batch sizes, low-rank representations, and generalization performance. While smaller batch sizes initially improved generalization in our independent group training in *AB* training, this trend reversed at extreme scales. This suggests that averaging independently trained models, although beneficial for exploring the loss landscape, can become detrimental when models diverge excessively.

The implications extend beyond optimization strategies. My findings suggest that traditional synchronous DP training transfers redundant information by synchronizing network gradients after every forward-backward pass. The rapid stabilization of the weight matrices' orthogonal components and the robustness of methods to stale model states hint at a potential for streamlined communication and computation in distributed training. While identifying the essential information remains challenging, these findings suggest that truly synchronous DP training is optional. Future research should investigate asynchronous training methods, sparse updates, and adaptive communication schemes to fully exploit these insights.

A deeper theoretical understanding of the interplay between batch size, low-rank representations, and optimization dynamics is crucial. This will enable researchers to develop robust, scalable training methods that leverage large batch sizes instead of simply dealing with them. This can open avenues to explore fundamentally different approaches to distributed training, potentially leading to significant gains in efficiency and performance.

7.3. Outlook

While this thesis presents significant advances in distributed training, several limitations and avenues for future research remain. The effectiveness of my methods could be further enhanced by developing more sophisticated model merging strategies and exploring adaptive learning rate schedules tailored to low-rank training. Additionally, investigating the theoretical underpinnings of the observed regularization effects in low-rank training could provide valuable insights into the generalization behavior of neural networks.

Implementing our methods in real-world production environments would require careful consideration of system-specific constraints and potential adaptations to diverse hardware configurations. Further investigation into the impact of network bandwidth limitations and the efficacy of our methods in heterogeneous computing environments would be crucial.

The challenge of mitigating large batch effects at extreme scales warrants continued research. Exploring alternative update mechanisms, such as non-average or loss-weighted averaging, or even developing new optimization algorithms tailored for low-rank distributed training, could offer promising solutions to this persistent problem. Overall, this thesis contributes to a growing body of knowledge aimed at making deep learning more efficient and scalable. The insights gained from this work lay a foundation for future advancements in distributed training.

Bibliography

[1] R. Wightman, N. Raw, A. Soare, et al. *rwightman/pytorch-image-models: v0.8.10dev0 Release*. Feb. 2023. DOI: 10.5281/ZENODO.4414861.

[2] T. Zhang and W. Li. *kDecay: Just adding k-decay items on Learning-Rate Schedule to improve Neural Networks*. arXiv:2004.05909 [cs]. Mar. 2022. DOI: 10.48550/arXiv.2004.05909.

[3] Y. You, J. Li, 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.

[4] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[5] J. Jumper, R. Evans, et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (Aug. 2021), pp. 583–589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2.

[6] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi. "Hierarchical Representation Learning in Graph Neural Networks With Node Decimation Pooling". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.5 (2022), pp. 2195–2207. DOI: 10.1109/TNNLS.2020.3044146.

[7] D. Kiela, M. Bartolo, et al. *Dynabench: Rethinking Benchmarking in NLP*. 2021. URL: https://arxiv.org/abs/2104.14337. arXiv: 2104.14337 [cs.CL].

[8] J. Kaplan, S. McCandlish, T. Henighan, et al. "Scaling Laws for Neural Language Models". In: (2020). URL: http://arxiv.org/pdf/2001.08361. arXiv: 2001.08361 [astro-ph.IM].

[9] E. Hoffer, I. Hubara, and D. Soudry. "Train longer, generalize better: closing the generalization gap in large batch training of neural networks". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 1729–1739. ISBN: 9781510860964.

[10] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519.

[11] D. Rolnick and M. Tegmark. "The power of deeper networks for expressing natural functions". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=SyProzZAW.

[12] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. "Activation Functions: Comparison of Trends in Practice and Research for Deep Learning". In: *arXiv preprint 1811.03378* (2018). URL: http://arxiv.org/abs/1811.03378.

[13] R. H. K. Emanuel, P. D. Docherty, H. Lunt, and K. Möller. "The effect of activation functions on accuracy, convergence speed, and misclassification confidence in CNN text classification: a comprehensive exploration". In: *The Journal of Supercomputing* 80.1 (June 2023), pp. 292–312. ISSN: 1573-0484. DOI: 10.1007/s11227-023-05441-7.

[14] Z. Li, F. Liu, et al. "A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.12 (2022), pp. 6999–7019. DOI: 10.1109/TNNLS.2021.3084827.

[15] M. Valueva, N. Nagornov, et al. "Application of the residue number system to reduce hardware costs of the convolutional neural network implementation". In: *Mathematics and Computers in Simulation* 177 (2020), pp. 232–243. ISSN: 0378-4754. DOI: https://doi.org/10.1016/j.matcom.2020.04.031.

[16] T. Kobayashi. "Analyzing Filters Toward Efficient ConvNet". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[17] S. Ioffe and C. Szegedy. "Batch normalization: accelerating deep network training by reducing internal covariate shift". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 448–456.

[18] J. L. Ba, J. R. Kiros, and G. E. Hinton. *Layer Normalization*. 2016. URL: https://arxiv.org/abs/1607.06450. arXiv: 1607.06450 [stat.ML].

[19] C.-Y. Lee, P. W. Gallagher, and Z. Tu. "Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree". In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*. Ed. by A. Gretton and C. C. Robert. Vol. 51. Proceedings of Machine Learning Research. Cadiz, Spain: PMLR, May 2016, pp. 464–472. URL: https://proceedings.mlr.press/v51/lee16a.html.

[20] A. Vaswani, N. Shazeer, et al. "Attention is all you need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.

[21] T. Luong, H. Pham, and C. D. Manning. "Effective Approaches to Attention-based Neural Machine Translation". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Ed. by L. Màrquez, C. Callison-Burch, and J. Su. Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 1412–1421. DOI: 10.18653/v1/D15-1166.

[22] F. Huang, K. Lu, et al. "Encoding Recurrence into Transformers". In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=7YfHla7IxBJ.

[23] A. Gillioz, J. Casas, E. Mugellini, and O. A. Khaled. "Overview of the Transformer-based Models for NLP Tasks". In: *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*. 2020, pp. 179–183. DOI: 10.15439/2020F20.

[24] A. Dosovitskiy, L. Beyer, et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. arXiv:2010.11929 [cs]. June 2021. DOI: 10.48550/arXiv.2010.11929.

[25] O. Chang, H. Liao, et al. "Conformer is All You Need for Visual Speech Recognition". In: *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2024, pp. 10136–10140. DOI: 10.1109/ICASSP48485.2024.10446532.

[26] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0.

[27] G. Morse and K. O. Stanley. "Simple Evolutionary Optimization Can Rival Stochastic Gradient Descent in Neural Networks". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. GECCO '16. Denver, Colorado, USA: Association for Computing Machinery, 2016, pp. 477–484. ISBN: 9781450342063. DOI: 10.1145/2908812.2908916.

[28] M. Hutson. "Has artificial intelligence become alchemy?" In: *Science* 360.6388 (2018), pp. 478–478. DOI: 10.1126/science.360.6388.478.

[29] H. Robbins and S. Monro. "A Stochastic Approximation Method". In: *The Annals of Mathematical Statistics* 22.3 (1951), pp. 400–407. DOI: 10.1214/aoms/1177729586.

[30] S. Smith, E. Elsen, and S. De. "On the Generalization Benefit of Noise in Stochastic Gradient Descent". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by H. D. III and A. Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 9058–9067. URL: https://proceedings.mlr.press/v119/smith20a.html.

[31] P. Goyal, P. Dollár, et al. *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour*. arXiv:1706.02677 [cs]. Apr. 2018. DOI: 10.48550/arXiv.1706.02677.

[32] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML'13. Atlanta, GA, USA: JMLR.org, 2013, III–1139–III–1147.

[33] S. Hanson and L. Pratt. "Comparing Biases for Minimal Network Construction with Back-Propagation". In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky. Vol. 1. Morgan-Kaufmann, 1988. URL: https://proceedings.neurips.cc/paper_files/paper/1988/file/1c9ac0159c94d8d0cbedc973445af2da-Paper.pdf.

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

[35] I. Loshchilov and F. Hutter. "Decoupled Weight Decay Regularization". In: (2017). URL: http://arxiv.org/pdf/1711.05101. arXiv: 1711.05101 [astro-ph.IM].

[36] Y. Pan and Y. Li. "Toward Understanding Why Adam Converges Faster Than SGD for Transformers". In: *OPT 2022: Optimization for Machine Learning (NeurIPS 2022 Workshop).* 2022. URL: https://openreview.net/forum?id=Sf1NlV2r6PO.

[37] E. Buber and B. Diri. "Performance Analysis and CPU vs GPU Comparison for Deep Learning". In: *2018 6th International Conference on Control Engineering and Information Technology (CEIT).* 2018, pp. 1–6. DOI: 10.1109/CEIT.2018.8751930.

[38] N. P. Jouppi, C. Young, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 1–12. ISSN: 0163-5964. DOI: 10.1145/3140659.3080246.

[39] D. Luebke. "CUDA: Scalable parallel programming for high-performance scientific computing". In: *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro.* 2008, pp. 836–838. DOI: 10.1109/ISBI.2008.4541126.

[40] *ROCm/ROCm.* original-date: 2016-03-18T00:24:29Z. July 2024. URL: https://github.com/ROCm/ROCm (visited on 07/17/2024).

[41] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[42] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1.* 2015. URL: https://fs.hlrs.de/projects/par/mpi//mpi31/.

[43] *NVIDIA/nccl.* original-date: 2015-11-14T00:12:04Z. July 2024. URL: https://github.com/NVIDIA/nccl (visited on 07/17/2024).

[44] *ROCm/rccl.* original-date: 2017-12-06T18:36:05Z. July 2024. URL: https://github.com/ROCm/rccl (visited on 07/17/2024).

[45] *Overview - NHR@KIT User Documentation.* URL: https://www.nhr.kit.edu/userdocs/horeka/.

[46] Y. Wei, Y. C. Huang, et al. “9.3 NVLink-C2C: A Coherent Off Package Chip-to-Chip Interconnect with 40Gbps/pin Single-ended Signaling”. In: *2023 IEEE International Solid-State Circuits Conference (ISSCC)*. 2023, pp. 160–162. DOI: 10.1109/ISSCC42615.2023.10067395.

[47] Mellanox. *Introducing 200G HDR InfiniBand Solutions*. 2019. URL: https://network.nvidia.com/files/doc-2020/wp-introducing-200g-hdr-infiniband-solutions.pdf.

[48] A. Paszke, S. Gross, et al. “Automatic differentiation in PyTorch”. en. In: (Oct. 2017). URL: https://openreview.net/forum?id=BJJsrmfCZ (visited on 04/11/2024).

[49] Martín Abadi, Ashish Agarwal, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[50] J. Ansel, E. Yang, et al. “PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation”. In: *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS ’24)*. ACM, Apr. 2024. DOI: 10.1145/3620665.3640366.

[51] F. Chollet et al. *Keras*. https://keras.io. 2015.

[52] M. Weiel, M. Götz, et al. “Dynamic particle swarm optimization of biomolecular simulation parameters with flexible objective functions”. In: *Nature Machine Intelligence* 3.8 (July 2021), pp. 727–734. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00366-3.

[53] G. M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS ’67 (Spring)* Not available (1967), Not available. ISSN: Not available. DOI: 10.1145/1465482.1465560.

[54] J. L. Gustafson. “Reevaluating Amdahl’s law”. In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415.

[55] T. Ben-Nun and T. Hoefler. “Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis”. In: *ACM Computing Surveys (CSUR)* 52.4 (2019), pp. 1–43. DOI: 10.1145/3320060.

[56] P. Liang, Y. Tang, et al. "A Survey on Auto-Parallelism of Large-Scale Deep Learning Training". In: *IEEE Transactions on Parallel and Distributed Systems* 34.8 (2023), pp. 2377–2390. DOI: 10.1109/TPDS.2023.3281931.

[57] Y. You, I. Gitman, and B. Ginsburg. "Large Batch Training of Convolutional Networks". In: (2017). URL: http://arxiv.org/pdf/1708.03888. arXiv: 1708.03888 [astro-ph.IM].

[58] J. Keuper and F.-J. Preundt. "Distributed Training of Deep Neural Networks: Theoretical and Practical Limits of Parallel Scalability". In: *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*. 2016, pp. 19–26. DOI: 10.1109/MLHPC.2016.006.

[59] A. Gibiansky. *Bringing HPC Techniques to Deep Learning - Andrew Gibiansky*. 2017. URL: https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/.

[60] A. Sergeev and M. D. Balso. *Horovod: fast and easy distributed deep learning in TensorFlow*. [accessed on 2021-08-06]. 2018. URL: https://arxiv.org/abs/1802.05799. arXiv: 1802.05799 [cs.LG].

[61] B. M. Assran, A. Aytekin, et al. "Advances in Asynchronous Parallel and Distributed Optimization". In: *Proceedings of the IEEE* 108.11 (2020), pp. 2013–2031. DOI: 10.1109/JPROC.2020.3026619.

[62] S. Zheng, Q. Meng, et al. "Asynchronous Stochastic Gradient Descent with Delay Compensation". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by D. Precup and Y. W. Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, Aug. 2017, pp. 4120–4129. URL: https://proceedings.mlr.press/v70/zheng17b.html.

[63] A. Koloskova, S. U. Stich, and M. Jaggi. "Sharper Convergence Guarantees for Asynchronous SGD for Distributed and Federated Learning". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo, S. Mohamed, et al. Vol. 35. Curran Associates, Inc., 2022, pp. 17202–17215. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/6db3ea527f53682657b3d6b02a841340-Paper-Conference.pdf.

[64] F. Niu, B. Recht, C. Re, et al. "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: (2011). URL: http://arxiv.org/pdf/1106.5730. arXiv: 1106.5730 [astro-ph.IM].

[65] A. Nedić and A. Olshevsky. "Stochastic Gradient-Push for Strongly Convex Functions on Time-Varying Directed Graphs". In: *IEEE Transactions on Automatic Control* 61.12 (2016), pp. 3936–3947. DOI: 10.1109/TAC.2016.2529285.

[66] M. S. Assran and M. G. Rabbat. "Asynchronous Gradient Push". In: *IEEE Transactions on Automatic Control* 66.1 (2021), pp. 168–183. DOI: 10.1109/TAC.2020.2981035.

[67] M. Assran, N. Loizou, N. Ballas, and M. Rabbat. "Stochastic Gradient Push for Distributed Deep Learning". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, June 2019, pp. 344–353. URL: https://proceedings.mlr.press/v97/assran19a.html.

[68] M. Assran and M. Rabbat. "An empirical comparison of multi-agent optimization algorithms". In: *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. 2017, pp. 573–577. DOI: 10.1109/GlobalSIP.2017.8309024.

[69] S. Li, Y. Zhao, R. Varma, et al. *PyTorch Distributed: Experiences on Accelerating Data Parallel Training*. [accessed on 2021-08-06]. 2020. URL: https://arxiv.org/abs/2006.15704. arXiv: 2006.15704 [cs.DC].

[70] N. Quang-Hung, H. Doan, and N. Thoai. "Performance Evaluation of Distributed Training in Tensorflow 2". In: *2020 International Conference on Advanced Computing and Applications (ACOMP)*. 2020, pp. 155–159. DOI: 10.1109/ACOMP50827.2020.00031.

[71] S. Hochreiter and J. Schmidhuber. "Flat Minima". In: *Neural Computation* 9.1 (Jan. 1997), pp. 1–42. ISSN: 1530-888X. DOI: 10.1162/neco.1997.9.1.1.

[72] P. Chaudhari, A. Choromanska, et al. "Entropy-SGD: biasing gradient descent into wide valleys". In: *Journal of Statistical Mechanics: Theory and Experiment* 2019.12 (Dec. 2019), p. 124018. DOI: 10.1088/1742-5468/ab39d9.

[73] J. Rissanen. "Modeling by shortest data description". In: *Automatica* 14.5 (Sept. 1978), pp. 465–471. ISSN: 0005-1098. DOI: 10.1016/0005-1098(78)90005-5.

[74] W. Wen, Y. Wang, F. Yan, et al. "SmoothOut: Smoothing Out Sharp Minima to Improve Generalization in Deep Learning". In: (2018). URL: http://arxiv.org/pdf/1805.07898. arXiv: 1805.07898 [astro-ph.IM].

[75] B. Singh, S. De, et al. "Layer-Specific Adaptive Learning Rates for Deep Networks". In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. 2015, pp. 364–368. DOI: 10.1109/ICMLA.2015.113.

[76] T. N. Mundhenk, G. Konjevod, W. A. Sakla, and K. Boakye. "A Large Contextual Dataset for Classification, Detection and Counting of Cars with Deep Learning". In: *Computer Vision – ECCV 2016*. Ed. by B. Leibe, J. Matas, N. Sebe, and M. Welling. Cham: Springer International Publishing, 2016, pp. 785–800. ISBN: 978-3-319-46487-9.

[77] S. Oh, A. Hoogs, et al. "A large-scale benchmark dataset for event recognition in surveillance video". In: *CVPR 2011*. 2011, pp. 3153–3160. DOI: 10.1109/CVPR.2011.5995586.

[78] G. Litjens, T. Kooi, et al. "A survey on deep learning in medical image analysis". In: *Medical Image Analysis* 42 (2017), pp. 60–88. ISSN: 1361-8415. DOI: https://doi.org/10.1016/j.media.2017.07.005.

[79] N. Dryden, N. Maruyama, et al. "Improving Strong-Scaling of CNN Training by Exploiting Finer-Grained Parallelism". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 210–220. DOI: 10.1109/IPDPS.2019.00031.

[80] X. Lyu. "The Impact of Input Image Data Size on The Training Speed of Convolutional Neural Networks". In: *2021 3rd International Conference on Machine Learning, Big Data and Business Intelligence (MLBDBI)*. 2021, pp. 654–657. DOI: 10.1109/MLBDBI54094.2021.00129.

[81] Y. Oyama, N. Maruyama, et al. "The Case for Strong Scaling in Deep Learning: Training Large 3D CNNs With Hybrid Parallelism". In: *IEEE Transactions on Parallel and Distributed Systems* 32.7 (2021), pp. 1641–1652. DOI: 10.1109/TPDS.2020.3047974.

[82] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. "ZeRO: memory optimizations toward training trillion parameter models". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, Nov. 2020, pp. 1–16. ISBN: 978-1-72819-998-6. (Visited on 04/11/2024).

[83] J. Wang, J. Ebert, O. Filatov, and S. Kesselheim. "Memory and Bandwidth are All Your Need for Fully Sharded Data Parallel". In: *2nd Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ICML 2024)*. 2024. URL: https://openreview.net/forum?id=qqVAsSh3Gc.

[84] Y. Zhao, A. Gu, et al. "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel". In: *Proc. VLDB Endow.* 16.12 (Aug. 2023), pp. 3848–3860. ISSN: 2150-8097. DOI: 10.14778/3611540.3611569.

[85] X. Lin, Y. Zhang, et al. *Efficient LLM Training and Serving with Heterogeneous Context Sharding among Attention Heads*. 2024. URL: https://arxiv.org/abs/2407.17678. arXiv: 2407.17678 [cs.CL].

[86] N. Dryden, N. Maruyama, et al. "Channel and filter parallelism for large-scale CNN training". 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: 10.1145/3295500.3356207.

[87] M. Shoeybi, M. Patwary, R. Puri, et al. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism". In: (2019). URL: http://arxiv.org/pdf/1909.08053. arXiv: 1909.08053 [astro-ph.IM].

[88] J. Du, X. Zhu, et al. "Model Parallelism Optimization for Distributed Inference Via Decoupled CNN Structure". In: *IEEE Transactions on Parallel and Distributed Systems* 32.7 (2021), pp. 1665–1676. DOI: 10.1109/TPDS.2020.3041474.

[89] D. Narayanan, A. Phanishayee, K. Shi, et al. "Memory-Efficient Pipeline-Parallel DNN Training". In: (2020). URL: http://arxiv.org/pdf/2006.09503. arXiv: 2006.09503 [astro-ph.IM].

[90] Y. Huang, Y. Cheng, A. Bapna, et al. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, et al. Vol. 32. 2019. URL: https://proceedings.neurips.cc/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.

[91] S. Moreno-Alvarez, J. M. Haut, M. E. Paoletti, and J. A. Rico-Gallego. "Heterogeneous model parallelism for deep neural networks". In: *Neurocomputing* 441 (2021), pp. 1–12. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2021.01.125.

[92] N. Shazeer, Y. Cheng, et al. "Mesh-TensorFlow: deep learning for supercomputers". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, pp. 10435–10444.

[93] K. Avhale. *Understanding of Optuna-A Machine Learning Hyperparameter Optimization Framework*. en. Aug. 2021. URL: https://medium.com/@kalyaniavhale7/understanding-of-optuna-a-machine-learning-hyperparameter-optimization-framework-ed31ebb335b9 (visited on 07/24/2024).

[94] F. Karl, T. Pielok, et al. "Multi-Objective Hyperparameter Optimization in Machine Learning—An Overview". In: *ACM Trans. Evol. Learn. Optim.* 3.4 (Dec. 2023). DOI: 10.1145/3610536.

[95] J. Dodge, G. Ilharco, et al. *Fine-Tuning Pretrained Language Models: Weight Initializations, Data Orders, and Early Stopping*. 2020. URL: https://arxiv.org/abs/2002.06305. arXiv: 2002.06305 [cs.CL].

[96] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. "Algorithms for Hyper-Parameter Optimization". In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor, R. Zemel, et al. Vol. 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf.

[97] A. Morales-Hernández, I. Van Nieuwenhuyse, and S. Rojas Gonzalez. "A survey on multi-objective hyperparameter optimization algorithms for machine learning". In: *Artificial Intelligence Review* 56.8 (Dec. 2022), pp. 8043–8093. ISSN: 1573-7462. DOI: 10.1007/s10462-022-10359-2.

[98] J. Bergstra and Y. Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305. URL: http://jmlr.org/papers/v13/bergstra12a.html.

[99] M. Feurer, J. Springenberg, and F. Hutter. "Initializing Bayesian Hyperparameter Optimization via Meta-Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 29.1 (Feb. 2015). DOI: 10.1609/aaai.v29i1.9354.

[100] D. Fogel. "An introduction to simulated evolutionary optimization". In: *IEEE Transactions on Neural Networks* 5.1 (1994), pp. 3–14. DOI: 10.1109/72.265956.

[101] O. Taubert, M. Weiel, et al. “Massively Parallel Genetic Optimization Through Asynchronous Propagation of Populations”. en. In: *High Performance Computing*. Ed. by A. Bhatele, J. Hammond, M. Baboulin, and C. Kruse. Cham: Springer Nature Switzerland, 2023, pp. 106–124. ISBN: 978-3-031-32041-5. DOI: 10.1007/978-3-031-32041-5_6.

[102] P. Wimmer, J. Mehnert, and A. P. Condurache. “Dimensionality reduced training by pruning and freezing parts of a deep neural network: a survey”. en. In: *Artificial Intelligence Review* (May 2023). ISSN: 1573-7462. DOI: 10.1007/s10462-023-10489-1.

[103] J.-H. Luo, J. Wu, and W. Lin. “ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression”. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 5068–5076. DOI: 10.1109/ICCV.2017.541.

[104] P. Molchanov, S. Tyree, et al. “Pruning Convolutional Neural Networks for Resource Efficient Inference”. In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=SJGCiw5gl.

[105] J. Frankle and M. Carbin. *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks*. 2019. arXiv: 1803.03635 [cs.LG].

[106] R. Ao, Z. Tao, et al. “Darb: A density-adaptive regular-block pruning for deep neural networks”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 5495–5502.

[107] V. Sanh, T. Wolf, and A. Rush. “Movement pruning: Adaptive sparsity by fine-tuning”. In: *Advances in neural information processing systems* 33 (2020), pp. 20378–20389.

[108] Y. Sakai, Y. Eto, and Y. Teranishi. “Structured pruning for deep neural networks with adaptive pruning rate derivation based on connection sensitivity and loss function”. In: *Journal of Advances in Information Technology* 1 (2022).

[109] J. Choi, S. Venkataramani, et al. “Accurate and Efficient 2-bit Quantized Neural Networks”. In: *Proceedings of Machine Learning and Systems*. Ed. by A. Talwalkar, V. Smith, and M. Zaharia. Vol. 1. 2019, pp. 348–359. URL: https://proceedings.mlsys.org/paper_files/paper/2019/file/c443e9d9fc984cda1c5cc447fe2c724d-Paper.pdf.

[110] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev. "Low-bit quantization of neural networks for efficient inference". In: *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE. 2019, pp. 3009–3018.

[111] P. Micikevicius, S. Narang, et al. "Mixed Precision Training". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=r1gs9JgRZ.

[112] M. Courbariaux, Y. Bengio, and J.-P. David. *BinaryConnect: Training Deep Neural Networks with binary weights during propagations*. 2016. arXiv: 1511.00363 [cs.LG].

[113] J. Guo, W. Liu, et al. "Accelerating Distributed Deep Learning By Adaptive Gradient Quantization". In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2020, pp. 1603–1607. DOI: 10.1109/ICASSP40776.2020.9054164.

[114] T. Vogels, S. P. Karimireddy, and M. Jaggi. "PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/d9fbed9da256e344c1fa46bb46c34c5f-Abstract.html (visited on 04/11/2024).

[115] D. Alistarh, D. Grubic, et al. *QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding*. [accessed on 2021-08-06]. 2017. URL: https://arxiv.org/abs/1610.02132. arXiv: 1610.02132 [cs.LG].

[116] G. Hinton, O. Vinyals, and J. Dean. "Distilling the Knowledge in a Neural Network". In: *NIPS Deep Learning and Representation Learning Workshop*. 2015. URL: http://arxiv.org/abs/1503.02531.

[117] G. Kaplun, E. Malach, P. Nakkiran, and S. Shalev-Shwartz. "Knowledge distillation: Bad models can be good role models". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 28683–28694.

[118] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 2020. arXiv: 1910.01108 [cs.CL].

[119] U. Evci, T. Gale, J. Menick, et al. "Rigging the Lottery: Making All Tickets Winners". In: *Proceedings of the 37th International Conference on Machine Learning*. ICML'20. JMLR.org, 2020.

[120] A. G. Howard, M. Zhu, et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[121] T. Hoefler, D. Alistarh, T. Ben-Nun, et al. "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks". In: *The Journal of Machine Learning Research* 22.1 (2021), pp. 10882–11005.

[122] Y. Idelbayev and M. A. Carreira-Perpinan. "Low-Rank Compression of Neural Nets: Learning the Rank of Each Layer". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020.

[123] J. Ren and D. Xia. "Deep Learning Model Optimization". In: *Autonomous driving algorithms and Its IC Design*. Singapore: Springer Nature Singapore, 2023, pp. 183–199. ISBN: 978-981-99-2897-2. DOI: 10.1007/978-981-99-2897-2_8.

[124] J. H. de M. Goulart, M. Boizard, et al. "Tensor CP Decomposition With Structured Factor Matrices: Algorithms and Performance". In: *IEEE Journal of Selected Topics in Signal Processing* 10.4 (2016), pp. 757–769. DOI: 10.1109/JSTSP.2015.2509907.

[125] H. Ding, K. Chen, et al. "A Compact CNN-DBLSTM Based Character Model for Offline Handwriting Recognition with Tucker Decomposition". In: *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*. Vol. 01. 2017, pp. 507–512. DOI: 10.1109/ICDAR.2017.89.

[126] V. Lebedev, Y. Ganin, et al. *Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition*. 2015. URL: https://arxiv.org/abs/1412.6553. arXiv: 1412.6553 [cs.CV].

[127] L. Deng, G. Li, et al. "Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey". In: *Proceedings of the IEEE* 108.4 (Apr. 2020). Conference Name: Proceedings of the IEEE, pp. 485–532. ISSN: 1558-2256. DOI: 10.1109/JPROC.2020.2976475.

[128] T. N. Sainath, B. Kingsbury, V. Sindhwani, et al. "Low-rank matrix factorization for Deep Neural Network training with high-dimensional output targets". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. ISSN: 2379-190X. May 2013, pp. 6655–6659. DOI: 10.1109/ICASSP.2013.6638949.

[129] S. Schotthöfer, E. Zangrando, et al. "Low-rank lottery tickets: finding efficient low-rank neural networks via matrix differential equations". en. In: *Advances in Neural Information Processing Systems* 35 (Dec. 2022), pp. 20051–20063. URL: https://papers.nips.cc/paper_files/paper/2022/hash/7e98b00eeafcdaeb0c5661fb9355be3a-Abstract-Conference.html.

[130] M. Yamazaki, A. Kasagi, A. Tabuchi, T. Honda, et al. *Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds.* [accessed on 2021-08-06]. 2019. URL: https://arxiv.org/abs/1903.12650. arXiv: 1903.12650 [cs.LG].

[131] X. Lian, W. Zhang, C. Zhang, and J. Liu. "Asynchronous Decentralized Parallel Stochastic Gradient Descent". In: *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 2018, pp. 3043–3052.

[132] S. Zhang, C. Zhang, Z. You, et al. "Asynchronous Stochastic Gradient Descent for DNN Training". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013, pp. 6660–6663. DOI: 10.1109/ICASSP.2013.6638950.

[133] S. Dutta, J. Wang, and G. Joshi. *Slow and Stale Gradients Can Win the Race.* [accessed on 2021-08-06]. 2020. URL: https://arxiv.org/abs/2003.10579. arXiv: 2003.10579 [stat.ML].

[134] N. Bogoychev, M. Junczys-Dowmunt, K. Heafield, and A. F. Aji. *Accelerating Asynchronous Stochastic Gradient Descent for Neural Machine Translation.* [accessed on 2021-08-06]. 2018. URL: https://arxiv.org/abs/1808.08859. arXiv: 1808.08859 [cs.CL].

[135] T. Lin, S. U. Stich, and M. Jaggi. "Don't Use Large Mini-Batches, Use Local SGD". In: *ArXiv* abs/1808.07217 (2020).

[136] Y. Ueno and R. Yokota. "Exhaustive Study of Hierarchical AllReduce Patterns for Large Messages Between GPUs". In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2019, pp. 430–439. DOI: 10.1109/CCGRID.2019.00057.

[137] H. Mikami, H. Suganuma, P. U.-Chupala, et al. *Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash.* [accessed on 2021-08-06]. 2018. URL: https://arxiv.org/abs/1811.05233. arXiv: 1811.05233 [cs.LG].

[138] D. C. Nguyen, M. Ding, et al. “Federated learning for internet of things: A comprehensive survey”. In: *IEEE Communications Surveys & Tutorials* 23.3 (2021), pp. 1622–1658.

[139] H. Wang, X. Liu, J. Niu, and S. Tang. “SVDFed: Enabling Communication-Efficient Federated Learning via Singular-Value-Decomposition”. In: *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*. 2023, pp. 1–10. DOI: 10.1109/INFOCOM53939.2023.10229042.

[140] D. Coquelin, C. Debus, M. Götz, et al. “Accelerating Neural Network Training with Distributed Asynchronous and Selective Optimization (DASO)”. en. In: *Journal of Big Data* 9.1 (Feb. 2022), p. 14. ISSN: 2196-1115. DOI: 10.1186/s40537-021-00556-1.

[141] A. Clauset. *A Brief Primer on Probability Distributions*. [accessed on 2021-08-06]. 2011. URL: http://tuvalu.santafe.edu/~aaronc/courses/7000/csci7000-001%5C_2011%5C_L0.pdf.

[142] J. Wang, S. Wang, R.-R. Chen, and M. Ji. “Demystifying Why Local Aggregation Helps: Convergence Analysis of Hierarchical SGD”. en. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 36.8 (June 2022). Number: 8, pp. 8548–8556. ISSN: 2374-3468. DOI: 10.1609/aaai.v36i8.20832.

[143] L. Bottou, F. E. Curtis, and J. Nocedal. “Optimization Methods for Large-Scale Machine Learning”. In: *ArXiv* (2018). [accessed on 2021-08-06]. URL: https://arxiv.org/abs/1606.04838.

[144] M. Götz, C. Debus, et al. “HeAT – a Distributed and GPU-accelerated Tensor Framework for Data Analytics”. In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, pp. 276–287. DOI: 10.1109/BigData50022.2020.9378050.

[145] D. Krause. “JUWELS: Modular Tier-0/1 Supercomputer at the Jülich Supercomputing Centre”. In: *Journal of Large-scale Research Facilities* 5 (2019), p. 135. ISSN: 2364-091X. DOI: 10.17815/jlsrf-5-171.

[146] *NVIDIA A100 TENSOR CORE GPU*. NVIDIA. 2021. URL: https://www.nvidia.com/en-us/data-center/a100/.

[147] J. Deng, W. Dong, R. Socher, L.-J. Li, et al. “ImageNet: A Large-scale Hierarchical Image Database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[148] NVIDIA Corporation. *NVIDIA Data Loading Library (DALI)*. [accessed on 2021-08-05]. 2021. URL: https://developer.nvidia.com/DALI.

[149] W. Zhang, S. Gupta, X. Lian, and J. Liu. *Staleness-aware Async-SGD for Distributed Deep Learning*. [accessed on 2021-08-06]. 2016. URL: https://arxiv.org/abs/1511.05950. arXiv: 1511.05950 [cs.LG].

[150] Y. Wu, A. Kirillov, F. Massa, et al. *Detectron2*. https://github.com/facebookresearch/detectron2. 2019.

[151] A. Tao, K. Sapra, and B. Catanzaro. *Hierarchical Multi-Scale Attention for Semantic Segmentation*. [accessed on 2021-08-06]. 2020. URL: https://arxiv.org/abs/2005.10821. arXiv: 2005.10821 [cs.CV].

[152] M. Cordts, M. Omran, S. Ramos, et al. "The Cityscapes Dataset for Semantic Urban Scene Understanding". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 3213–3223. DOI: 10.1109/CVPR.2016.350.

[153] H. Rezatofighi, N. Tsoi, J. Gwak, et al. *Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression*. [accessed on 2021-08-06]. 2019. URL: https://arxiv.org/abs/1902.09630. arXiv: 1902.09630 [cs.CV].

[154] S. Zhao, Y. Wang, Z. Yang, and D. Cai. *Region Mutual Information Loss for Semantic Segmentation*. [accessed on 2021-08-06]. 2019. URL: http://arxiv.org/abs/1910.12037. arXiv: 1910.12037 [cs.CV].

[155] P. Singh, V. K. Verma, P. Rai, and V. P. Namboodiri. "Play and Prune: Adaptive Filter Pruning for Deep Model Compression". In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. IJCAI'19. Macao, China: AAAI Press, 2019, pp. 3460–3466. ISBN: 9780999241141.

[156] Nicoguaro. *Principal component analysis*. Page Version ID: 1232418328. July 2024. URL: https://en.wikipedia.org/w/index.php?title=Principal_component_analysis&oldid=1232418328 (visited on 07/28/2024).

[157] D. Psichogios and L. Ungar. "SVD-NET: an algorithm that automatically selects network structure". In: *IEEE Transactions on Neural Networks* 5.3 (May 1994). Conference Name: IEEE Transactions on Neural Networks, pp. 513–515. ISSN: 1941-0093. DOI: 10.1109/72.286929.

[158] G. I. Winata, S. Cahyawijaya, et al. “Lightweight and Efficient End-To-End Speech Recognition Using Low-Rank Transformer”. In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. ISSN: 2379-190X. May 2020, pp. 6144–6148. DOI: 10.1109/ICASSP40776.2020.9053878.

[159] A.-H. Phan, K. Sobolev, K. Sozykin, et al. “Stable Low-Rank Tensor Decomposition for Compression of Convolutional Neural Network”. en. In: *Computer Vision – ECCV 2020*. Ed. by A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 522–539. ISBN: 978-3-030-58526-6. DOI: 10.1007/978-3-030-58526-6_31.

[160] S. Cahyawijaya, G. I. Winata, H. Lovenia, et al. *Greenformer: Factorization Toolkit for Efficient Deep Neural Networks*. arXiv:2109.06762 [cs]. Oct. 2021. DOI: 10.48550/arXiv.2109.06762.

[161] Y. Xu, Y. Li, et al. “Trained Rank Pruning for Efficient Deep Neural Networks”. In: *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*. 2019, pp. 14–17. DOI: 10.1109/EMC2-NIPS53020.2019.00011.

[162] E. J. Hu, Y. Shen, P. Wallis, et al. “LoRA: Low-Rank Adaptation of Large Language Models”. In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=nZeVKeeFYf9.

[163] R. Waleffe and T. Rekatsinas. “Principal Component Networks: Parameter Reduction Early in Training”. In: *CoRR* abs/2006.13347 (2020). URL: https://arxiv.org/abs/2006.13347. arXiv: 2006.13347.

[164] M. M. Bejani and M. Ghatee. *Adaptive Low-Rank Factorization to regularize shallow and deep neural networks*. arXiv:2005.01995 [cs, stat]. May 2020. DOI: 10.48550/arXiv.2005.01995.

[165] Z. Yang, A. Zhang, and A. Sudjianto. “Enhancing Explainability of Neural Networks Through Architecture Constraints”. In: *IEEE Transactions on Neural Networks and Learning Systems* 32.6 (2021), pp. 2610–2621. DOI: 10.1109/TNNLS.2020.3007259.

[166] B. Zhang, W. Zheng, J. Zhou, and J. Lu. “Bort: Towards Explainable Neural Networks with Bounded Orthogonal Constraint”. In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=My57qBufZWs.

[167] D. Povey, G. Cheng, Y. Wang, et al. "Semi-Orthogonal Low-Rank Matrix Factorization for Deep Neural Networks". en. In: *Interspeech 2018*. ISCA, Sept. 2018, pp. 3743–3747. DOI: 10.21437/Interspeech.2018-1417.

[168] O. Koch and C. Lubich. "Dynamical Low-Rank Approximation". In: *SIAM Journal on Matrix Analysis and Applications* 29.2 (2007), pp. 434–454. eprint: https://doi.org/10.1137/050639703. DOI: 10.1137/050639703.

[169] I. Bello, W. Fedus, X. Du, et al. *Revisiting ResNets: Improved Training and Scaling Strategies*. arXiv:2103.07579 [cs]. Mar. 2021. DOI: 10.48550/arXiv.2103.07579.

[170] O. Russakovsky, J. Deng, H. Su, et al. "ImageNet Large Scale Visual Recognition Challenge". en. In: *International Journal of Computer Vision* 115.3 (Dec. 2015), pp. 211–252. ISSN: 1573-1405. DOI: 10.1007/s11263-015-0816-y.

[171] H. Wu, J. Xu, J. Wang, and M. Long. "Autoformer: Decomposition Transformers with Auto-Correlation for Long-Term Series Forecasting". In: *Advances in Neural Information Processing Systems*. 2021.

[172] H. Touvron, M. Cord, M. Douze, et al. "Training data-efficient image transformers & distillation through attention". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by M. Meila and T. Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 10347–10357. URL: https://proceedings.mlr.press/v139/touvron21a.html.

[173] S. Nitish, G. Hinton, A. Krizhevsky, et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[174] A. Gotmare, N. S. Keskar, C. Xiong, and R. Socher. "A Closer Look at Deep Learning Heuristics: Learning rate restarts, Warmup and Distillation". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=r14EOsCqKX.

[175] I. Loshchilov and F. Hutter. *SGDR: Stochastic Gradient Descent with Warm Restarts*. arXiv:1608.03983 [cs, math]. May 2017. DOI: 10.48550/arXiv.1608.03983.

[176] I. L. and F. H. *Fixing Weight Decay Regularization in Adam*. 2018. URL: https://openreview.net/forum?id=rk6qdGgCZ.

[177] L. Beyer, O. J. Hénaff, A. Kolesnikov, et al. *Are we done with ImageNet?* arXiv:2006.07159 [cs]. June 2020. DOI: 10.48550/arXiv.2006.07159.

[178] Y. Idelbayev and M. A. Carreira-Perpinan. "Low-Rank Compression of Neural Nets: Learning the Rank of Each Layer". In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. ISSN: 2575-7075. June 2020, pp. 8046–8056. DOI: 10.1109/CVPR42600.2020.00807.

[179] Y. He, X. Zhang, and J. Sun. "Channel Pruning for Accelerating Very Deep Neural Networks". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 1398–1406. DOI: 10.1109/ICCV.2017.155.

[180] Y. He, G. Kang, X. Dong, et al. "Soft Filter Pruning for Accelerating Deep Convolutional Neural Networks". In: *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. IJCAI'18. Stockholm, Sweden: AAAI Press, 2018, pp. 2234–2240. ISBN: 9780999241127.

[181] S. Lin, R. Ji, C. Yan, et al. "Towards Optimal Structured CNN Pruning via Generative Adversarial Learning". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 2785–2794. DOI: 10.1109/CVPR.2019.00290.

[182] J. Lin, Y. Rao, J. Lu, and J. Zhou. "Runtime Neural Pruning". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, et al. Vol. 30. Curran Associates, Inc., 2017.

[183] S. Liu and W. Deng. "Very deep convolutional neural network based image classification using small training sample size". In: *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*. 2015, pp. 730–734. DOI: 10.1109/ACPR.2015.7486599.

[184] D. Coquelin, K. Flügel, et al. "Harnessing Orthogonality to Train Low-Rank Neural Networks". In: *ECAI 2024*. IOS Press, 2024, pp. 2106–2113. DOI: 10.3233/FAIA240729.

[185] A. Hassani, S. Walton, N. Shah, et al. *Escaping the Big Data Paradigm with Compact Transformers*. arXiv:2104.05704 [cs] version: 4. June 2022. DOI: 10.48550/arXiv.2104.05704.

[186] H. Zhou, S. Zhang, J. Peng, et al. *Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting*. arXiv:2012.07436 [cs]. Mar. 2021. DOI: 10.48550/arXiv.2012.07436.

[187] T. Wolf, L. Debut, et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing*. 2020. URL: https://arxiv.org/abs/1910.03771. arXiv: 1910.03771 [cs.CL].

[188] L. Abrahamyan, Y. Chen, G. Bekoulis, and N. Deligiannis. "Learned Gradient Compression for Distributed Deep Learning". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.12 (2022), pp. 7330–7344. DOI: 10.1109/TNNLS.2021.3084806.

[189] L. Liebenwein, A. Maalouf, D. Feldman, and D. Rus. "Compressing Neural Networks: Towards Determining the Optimal Layer-wise Decomposition". In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato, A. Beygelzimer, et al. Vol. 34. Curran Associates, Inc., 2021, pp. 5328–5344. URL: https://proceedings.neurips.cc/paper_files/paper/2021/file/2adcfc3929e7c03fac3100d3ad51da26-Paper.pdf.

[190] H. Yang, M. Tang, W. Wen, et al. "Learning Low-Rank Deep Neural Networks via Singular Vector Orthogonality Regularization and Singular Value Sparsification". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. June 2020.

[191] H. Wang, S. Agarwal, et al. "Cuttlefish: Low-Rank Model Training without All the Tuning". In: *ArXiv* abs/2305.02538 (2023). URL: https://api.semanticscholar.org/CorpusID:258480187.

[192] H. Wang, S. Agarwal, and D. Papailiopoulos. "Pufferfish: Communication-efficient Models At No Extra Cost". In: *ArXiv* abs/2103.03936 (2021). URL: https://api.semanticscholar.org/CorpusID:232148049.

[193] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 25. Curran Associates, Inc., 2012. URL: https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[194] A. Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". In: 2009. URL: https://api.semanticscholar.org/CorpusID:18268744.

[195] S. Liu and W. Deng. "Very deep convolutional neural network based image classification using small training sample size". In: *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*. 2015, pp. 730–734. DOI: 10.1109/ACPR.2015.7486599.

[196] W. Hu, L. Xiao, and J. Pennington. "Provable Benefit of Orthogonal Initialization in Optimizing Deep Linear Networks". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=rkgqN1SYvr.

[197] J. Chang, Y. Lu, et al. "Iterative clustering pruning for convolutional neural networks". In: *Knowledge-Based Systems* 265 (2023), p. 110386. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2023.110386.

[198] M. Lin, R. Ji, et al. "Channel Pruning via Automatic Structure Search". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*. Ed. by C. Bessiere. Main track. International Joint Conferences on Artificial Intelligence Organization, July 2020, pp. 673–679. DOI: 10.24963/ijcai.2020/94.

[199] D. Granziol, S. Zohren, and S. Roberts. "Learning Rates as a Function of Batch Size: A Random Matrix Theory Approach to Neural Network Training". In: *Journal of Machine Learning Research* 23.173 (2022), pp. 1–65. URL: http://jmlr.org/papers/v23/20-1258.html.

[200] G. Sumbul, J. Kang, et al. *BigEarthNet Dataset with A New Class-Nomenclature for Remote Sensing Image Understanding*. 2020. URL: http://arxiv.org/abs/2001.06372.

[201] B. Bischl, M. Binder, et al. "Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges". In: *WIREs Data Mining and Knowledge Discovery* 13.2 (2023), e1484. DOI: https://doi.org/10.1002/widm.1484.

[202] T. Akiba, S. Sano, et al. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[203] A. Chatzimparmpas, R. M. Martins, K. Kucher, and A. Kerren. "VisEvol: Visual analytics to support hyperparameter search through evolutionary optimization". In: *Computer Graphics Forum*. Vol. 40. 3. Wiley Online Library. 2021, pp. 201–214.

[204] A. M. Vincent and P. Jidesh. "An improved hyperparameter optimization framework for AutoML systems using evolutionary algorithms". In: *Scientific Reports* 13.1 (2023), p. 4737.

[205] P. C. Silva, P. d. O. e Lucas, H. J. Sadaei, and F. G. Guimaraes. "Distributed evolutionary hyperparameter optimization for fuzzy time series". In: *IEEE Transactions on Network and Service Management* 17.3 (2020), pp. 1309–1321.

[206] G. Sumbul, M. Charfuelan, B. Demir, and V. Markl. "BigEarthNet: A Large-Scale Benchmark Archive For Remote Sensing Image Understanding". In: *arXiv preprint 1902.06148* (2019). URL: http://arxiv.org/abs/1902.06148.

[207] M. Bossard, J. Feranec, and J. Otahel. *CORINE land cover technical guide – Addendum 2000.* Tech. rep. European Environment Agency, 2000.

[208] *Scripts to Remove Cloudy and Snowy Patches.* URL: https://gitlab.tubit.tu-berlin.de/rsim/bigearthnet-tools.

[209] A. Dudchenko and G. Kopanitsa. "Comparison of Word Embeddings for Extraction from Medical Records". en. In: *Int J Environ Res Public Health* 16.22 (Nov. 2019).

[210] K. He, X. Zhang, S. Ren, and J. Sun. "Identity Mappings in Deep Residual Networks". In: *arXiv preprint 1603.05027* (2016). URL: http://arxiv.org/abs/1603.05027.

[211] S. Sun, Z. Cao, H. Zhu, and J. Zhao. "A Survey of Optimization Methods from a Machine Learning Perspective". In: *arXiv preprint 1906.06821* (2019). URL: http://arxiv.org/abs/1906.06821.

[212] H. B. McMahan. "Analysis Techniques for Adaptive Online Learning". In: *arXiv preprint 1403.3465* (2014), pp. 76–89. DOI: 10.1109/MGRS.2020.2964708.

[213] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *Ann. Math. Statist.* 22.1 (Mar. 1951), pp. 79–86. DOI: 10.1214/aoms/1177729694.

[214] Q. Wang, Y. Ma, K. Zhao, and Y. Tian. "A Comprehensive Survey of Loss Functions in Machine Learning". In: *Annals of Data Science* (Apr. 2020). ISSN: 2198-5812. DOI: 10.1007/s40745-020-00253-5.

[215] S. Jadon. “A survey of loss functions for semantic segmentation”. In: *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)* (Oct. 2020). DOI: 10.1109/cibcb48159.2020.9277638.

[216] Y. Wu, L. Liu, J. Bae, et al. “Demystifying Learning Rate Polices for High Accuracy Training of Deep Neural Networks”. In: *arXiv preprint 1908.06477* (2019). URL: http://arxiv.org/abs/1908.06477.

A. Appendix

A.1. OIALR Experimental Hyperparameters

Parameters not listed use the default values in the respective implementations.

A.1.1. ImageNet-2012

The non-default hyperparameters for all experiments on the ImageNet-2012 dataset are shown in Table A.1. We utilized the ViT implementation from Torchvision [50] and the ResNet-RS 101 implementation from [1].

A.1.2. Mini-ViT on CIFAR-10

For training the mini-ViT we used most of the same parameters as listed in Table A.1 except a lower learning rate. The utilized ViT for these experiments was from [1]. The training parameters for these experiments are shown in Table A.2. The search space for `Populate` and the parameters for the search itself are shown in Table A.3

A.1.3. AutoFormer on ETTm2

The learning rate schedule used in the original AutoFormer [171] is a step-based schedule with fixed steps, it is denoted as 'type1.' The hyperparameters used in our experiments are listed in Table A.4. The parameters for the hyperparameter search are listed in Table A.5.

Table A.1.: Hyperparameters for training networks on ImageNet-2012 with OIALR. Dataset parameters are referring to the dataset transforms provided by [1]. LR k-decay is a parameter of the cosine learning rate decay [2]

General Training Hyperparameters			
Local batch size	128	Learning Rate Scheduler	
Global batch size	1024	Learning rate (LR)	0.001
Autocast to bfloat16	True	Minimum learning rate	0.00001
Epochs	125	Warmup LR	0.00001
Label smoothing	0.1	LR k-decay	1
Optimizer	AdamW	Warmup epochs	10
Sync batchnorm	True		
General dataset hyperparameters			
Interpolation	random	Auto augment	rand-m15-mstd0.5-inc1
Random erasing probability	0.25	crop pct	0.9
Random erasing mode	pixel	scale	(0.08, 1)
		Training crop size	160
OIALR hyperparameters			
Full rank first layer	False	Delay	25000
Stability frequency	1000	Full rank last layer	True
Sigma cutoff fraction	0.1		
ResNet-RS 101 hyperparameters			
Dropout	0.25	Validate crop size	224
ViT B/16 hyperparameters			
Dropout	0.1	Hidden dim	768
Mlp dim	3072	Num layers	12
Num heads	12	Patch size	16

A.2. AB Training Experiments

All the *AB* experiments used a common set of HPs, shown in Table A.6. The baseline experiments used the same HPs without *AB* training enabled.

For experiments using the ImageNet-2012 dataset, the crop size of the images was set to 224 for training and validation. Furthermore, the maximum LR for all of these experiments was 0.0035.

The constant global batch size scaling experiments all used a global batch size of 4096 data samples. This means that the local batch sizes for 16, 32, 64, and 128 GPUs are 256, 128, 64, and 32, respectively.

Table A.2.: Hyperparameters used for CIFAR10 training runs. General hyperparameters used for all runs, OIALR hyperparameters use for all OIALR runs. Dataset parameters refer to implementation options in `timm` [1]

General Hyperparamters			
Train crop size	32	Label smoothing	0.1
Local batch size	256	Optimizer	AdamW
Global batch size	1024	auto_augment	rand-m9-mstd0.5-inc1
Autocast to bfloat16	True	Crop percent	1
Random erasing probability	0.25	Image scale	(0.8, 1.0)
Random erasing mode	pixel	Interpolation	random
ViT depth	6	ViT num heads	6
ViT qkv_bias	False	ViT patch_size	8
ViT embed_dim	768	ViT drop_path_rate	0.2
ViT mlp_ratio	4		
Baseline		**Tuned**	
LR	0.0001	LR	0.0002
Minimum LR	0.00001	Minimum LR	0.0008
Warmup LR	0.00001	Warmup LR	0.00008
LR k-decay	1	LR k-decay	0.4
Warmup epochs	10	Warmup epochs	17
OIALR hyperparameters			
Delay	4000	Stability frequency	1000
Full rank last layer	True	Sigma cutoff fraction	0.2
Full rank first layer	False		

The scaling experiments on ResNet-50 used the hyperparameters shown in Table A.7. Furthermore, these experiments also used a dropout of 0.1 and a weight decay of 0.01. The scaling experiments on ViT-B/16 are shown in Table A.8. These experiments used a drop path rate of 0.1 and a weight decay of 0.1.

The experiments on CIFAR10 with VGG16 used the parameters shown in Table A.6 and Table A.9.

The learning rate schedulers utilized are from the `timm` library [1]. Unlisted parameters utilized their default values.

Table A.3.: Propulate search parameters for the mini ViT on CIFAR-10 for OIALR training.

Parameters to search over	Search space	Propulate parameter	Value
LR	(5e-5, 1e-3)	Crossover probability	0.7
Minimum LR	(5e-6, 1e-3)	Mutation probability	0.4
Warmup LR	(5e-6, 2e-4)	Random init probability	0.1
LR k-decay	(0.1, 2)	Number of islands	8
Warmup epochs	(1, 20)	Migration probabilty	0.9
OIALR sigma cutoff fraction	(0.01, 0.9)		
OIALR stability frequency	(200, 1000)		
OIALR delay	(10e2 10e3)		

Table A.4.: Hyperparameters used for training AutoFormer models on the ETTm2 dataset for OIALR training.

General Hyperparameters			
Dimension of linear layers	2048	Number encoder layers	2
Loss function	MSE	Early stopping patience	3
Decoder input size	7	Start token length	48
Use distilling	True	Activation function	gelu
Encoder input size	7	Batch size	32
Attention factor	1	Moving average window	25
Dimension of model	512	Maximum training epochs	20
Dropout	0.05	Output attention	False
Number of heads	8	Number decoder layers	1
Default LR schedule		**Tuned LR schedule**	
LR schedule	type1	LR schedule	cosine
Learning rate	0.0004	learning_rate	0.01
		lr_k_decay	0.85
		min_lr	0.0004
		warmup_lr	0.0001
		warmup_epochs	3
OIALR Hyperparameters			
Delay	600	Full rank first layer	True
Full rank last layer	True	Stability frequency	400
Full rank warmup	False	Sigma cutoff fraction	0.4

Table A.5.: The search space and settings for the hyperparameter search for OIALR using `Propulate`.

Parameter	Search Space	Propulate Parameter	Value
LR	(1e-5, 5e-3)	Crossover probability	0.7
Minimum LR	(5e-6, 1e-3)	Mutation probability	0.4
Warmup LR	(5e-6, 2e-4)	Random init probability	0.1
LR k-decay	(1e-3, 2)	Number of islands	4
Warmup epochs	(2, 10)	Migration probabilty	0.9
OIALR sigma cutoff fraction	(0.01, 0.9)		
OIALR stability frequency	(50, 2000)		
OIALR delay	(250, 2500)		
OIALR full rank first layer	(False, True)		
OIALR full rank last layer	(False, True)		

Table A.6.: The HPs used for all experiments using *AB* training. Baseline experiments use the same HPs.

Autocast	True
Training epochs	150
Loss function	Cross Entropy
Label smoothing	0.1
LR schedule	Cosine Decay
LR warmup epochs	10
Optimizer	AdamW
Maximum gradient norm	1.0
Initialization method	Orthogonal
AB full rank layers	Input and Output layers
AB low rank cutoff	0.15
AB group size	4

Table A.7.: The HPs used for the ResNet-50 scaling experiments in *AB* training. The constant global batch size experiments are marked with an asterisk.

Global batch size	2048	4096*	8192	16384	32768
AB `warmupSteps`	24000	9300*	6150	3750	1900
AB `numABSteps`	6000	3000*	1500	600	400
AB `fullRankReboundSteps`	600	300*	150	75	38
LR rebound steps	1500	750*	325	188	94
Iterations per epoch	600	300*	150	75	38

Table A.8.: The HPs used for the ViT-B/16 scaling experiments in *AB* training. The constant global batch size experiments are marked with an asterisk.

Global batch size	2048	4096*	8192	16384	32768
AB `warmupSteps`	24000	12300*	6150	3750	1520
AB `numABSteps`	4360	2180*	1500	750	270
AB `fullRankReboundSteps`	600	300*	150	75	38
LR rebound steps	1500	750*	325	188	94
Iterations per epoch	600	300*	150	75	38

Table A.9.: The HPs used for training VGG16 on CIFAR10 in *AB* training.

Training crop size	32
Epochs	150
LR	0.003
Weight decay	0.1
AB `warmupSteps`	4330
AB `numABSteps`	759
AB `fullRankReboundSteps`	60
AB `LR rebound steps`	267
AB group size	4