

Tools and Methods for Distributed and Large-Scale Training of Deep Neural Networks

DOCTORAL THESIS IN INFORMATION AND COMMUNICATION TECHNOLOGY

Specialization in Software and Computer Systems

Sina Sheikholeslami

Doctoral dissertation, which, with the permission of the KTH Royal Institute of Technology, is submitted for public defense for the Degree of Doctor of Philosophy on Thursday, the 27th of March 2025, at 09:00 in Sal-A, Electrum, Kistagången 16, Stockholm.

Principal Supervisor:

Professor Vladimir Vlassov, KTH Royal Institute of Technology

Co-supervisors:

Associate Professor Amir H. Payberah, KTH Royal Institute of Technology Dr. Jim Dowling, Hopsworks AB

Advance Reviewer and Substitute Grading Committee Member:

Associate Professor Hossein Azizpour, KTH Royal Institute of Technology

Opponent:

Associate Professor Salman Toor, Uppsala University.

Grading Committee:

Professor Keijo Heljanko, University of Helsinki Associate Professor Marina Papatriantafilou, Chalmers University of Technology Associate Professor Monowar Bhuyan, Umeå University

© Sina Sheikholeslami, 2025

TRITA-EECS-AVL-2025:28 ISBN 978-91-8106-214-4

This doctoral dissertation has been prepared using IATEX. Printed by Universitets service US-AB, Stockholm, Sweden 2025

Abstract

Deep Neural Networks (DNNs) have been at the forefront of recent breakthroughs in Machine Learning (ML) and Deep Learning (DL). DNNs are increasingly used in various tasks, from Earth observation and analysis of satellite images to medical diagnosis and smart chatbots. A major contributor to these advances has been the abundance of training data, computation resources, and frameworks that enable efficient training of ever-larger and more complex DNNs, within a paradigm referred to as distributed DL, and in particular, distributed training, which is the focus of this doctoral dissertation. In distributed training, the data and computation are distributed across several workers as opposed to single-host training in which both the data and computation reside and happen on a single worker. In this setting, distributed training can help overcome the limitations of single-host training, such as memory constraints, computational bottlenecks, and data availability.

However, distributed training comes with a number of challenges that need to be carefully addressed in order to have a system that efficiently makes use of it. These challenges include, but are not limited to, efficient distribution of computation and data across the workers, the presence of straggler workers in a cluster (workers that get significantly behind in their computation step compared to the other workers), especially in synchronous execution settings, and communication and synchronization among the workers. This implies that the system should provide scalability in both the computation and the data dimensions.

On the other hand, from a programming and usability point of view, using the distributed training paradigm typically requires knowledge of distributed computing principles and experience with distributed and dataintensive computing frameworks as well as applying major changes to the code used for single-host training. Furthermore, as training a DNN involves several steps and stages (e.g., data preparation, hyperparameter tuning, model training, etc.), it would be desirable to possibly reuse the computational results of different steps in each other (e.g., reusing weights learned during hyperparameter tuning trials, for weight initialization of the model training step) in order to improve training time. Finally, when developing larger and more complex DNNs, we also need to know about each design choice's contributions.

The contributions of this doctoral dissertation address the aforementioned challenges, and collectively optimize large-scale DNN training, making it more accessible, efficient, and computationally sustainable while reducing the redundancy in ML/DL workflows, and providing usable tools for conducting ablation studies.

Keywords

Distributed Deep Learning, Ablation Studies, Data-parallel Training, Deep Neural Networks, Systems for Machine Learning, Weight Initialization, Hyperparameter Optimization

Sammanfattning

Deepa neurala nätverk (DNNs) har varit i framkant av de senaste genombrotten inom maskininlärning (ML) och djupinlärning (DL). DNN används i allt större utsträckning inom en rad olika områden, från jordobservation och analys av satellitbilder till medicinsk diagnostik och smarta chattbotar. En stor bidragande faktor till dessa framsteg är tillgången på stora mängder träningsdata, kraftfulla beräkningsresurser och ramverk som möjliggör effektiv träning av allt större och mer komplexa DNNs inom ett paradigm som kallas distribuerad DL. Inom detta område är distribuerad träning fokus för denna doctoral dissertation. I distribuerad träning fördelas data och beräkningar över flera arbetarnoder, till skillnad från träning på en enskild värd där både data och beräkningar hanteras av en enda nod. I denna kontext kan distribuerad träning bidra till att övervinna begränsningar såsom minnesbegränsningar, beräkningsflaskhalsar och begränsad datatillgång.

Distribuerad träning innebär dock flera utmaningar som måste hanteras noggrant för att säkerställa effektiv resursanvändning. Dessa utmaningar inkluderar, men är inte begränsade till, effektiv fördelning av beräkningar och data mellan noder, förekomsten av stragglers (arbetarnoder som hamnar efter i sina beräkningar jämfört med andra), särskilt i synkrona exekveringsmiljöer, samt kommunikation och synkronisering mellan noderna. För att systemet ska vara skalbart behöver det kunna hantera både ökande beräkningsbehov och större datamängder.

Ur ett programmerings- och användbarhetsperspektiv kräver distribuerad träning ofta djupgående kunskap om distribuerad beräkning och erfarenhet av dataintensiva ramverk. Dessutom innebär det ofta omfattande anpassningar av kod som används för träning på en enskild värd. Eftersom träning av en DNN innefattar flera steg och faser (t.ex. datapreparering, hyperparametertuning, modellträning etc.), vore det önskvärt att återanvända beräkningsresultat från olika steg (t.ex. vikter inlärda under hyperparametertuning för att initialisera modellträningen) för att förbättra träningseffektiviteten. Slutligen, vid utveckling av större och mer komplexa DNNs, är det också viktigt att förstå varje designvals inverkan.

Denna doctoral dissertation adresserar de ovan nämnda utmaningarna och optimerar storskalig DNN-träning genom att göra den mer tillgänglig, effektiv och beräkningsmässigt hållbar, samtidigt som redundansen i ML/DL-arbetsflöden minskas och användbara verktyg för ablationsstudier tillhandahålls.

Nyckelord

Distribuerad djupinlärning, Ablationsstudier, Dataparallell träning, Djupa

neurala nätverk, System för maskininlärning, Viktinitialisering, Hyperparameteroptimering

Preface

I started my doctoral studies in September 2019. Back then, DNNs had already proved to be a game changer for many applications, but the past couple of years have seen a totally different interest in - and public awareness of - the potential that Artificial Intelligence (AI), and in particular ML and DL can unlock¹. This is largely thanks to the public release of ChatGPT in late 2022, which is only one of the many tools based on many different DNN architectures (but it is probably the best-known AI tool out there at the time of writing this doctoral dissertation). Nevertheless, the focus of this doctoral dissertation has been to investigate how we can come up with better tools, methods, and frameworks for training ever-larger DNNs (as one of the most widely used types of ML/DL models) in a way that is faster, more resource-efficient, and ideally leads to improved models.

Now, back in 2019, there were already a lot of challenges when it came to training DNNs at scale - which meant, training big DNNs on big datasets on big clusters of computers (a.k.a. workers, which typically have one or more Graphics Processing Units (GPUs) as their computation workhorses). Large-scale training naturally required using multiple computers, or clusters of computers, in a distributed manner. However, many practitioners and researchers were not that well-versed in distributed systems principles or distributed computing tools. Moreover, the model training step is just one of many in a typical model training pipeline. As a result, the codebase for an ML/DL training pipeline includes specific code for each step, further contributing to code redundancy and maintenance challenges. We looked at two particular steps, hyperparameter tuning and ablation study, since they are both embarrassingly parallel tasks and typically share a significant amount of code with each other and the model training step.

When we want to train a DNN from scratch (which means, we have a model that has not been trained on any datasets), we should select, or in most cases, search for, suitable hyperparameter values, in a step known as hyperparameter tuning/optimization (which is also the main application of MAGGY). Hyperparameters are configuration values that the model cannot learn on its own, and we need to set them to be able to start the training process. You can think of them as knobs and settings of the training process. Examples of hyperparameters are the learning rate (specifying how much the model is changed after each step of training), the number of layers in a DNN, and batch size, which refers to the number of dataset examples

 $^{^1{\}rm ML},$ DL, and in general AI had already been regarded as hype in the preceding years, but the release of ChatGPT made AI the talk of the town.

in each mini-batch of data that is used for training. Each hyperparameter tuning trial typically involves training the same (or a very similar model, if the model architecture is also a hyperparameter), on a subset of the data that will be used for the full model training. The configuration with the best performance (e.g., in terms of classification accuracy) will be selected and its hyperparameter values will be used for initializing the model training step. As you can imagine, the code for the hyperparameter tuning step will turn out to be very similar to the model training code, and many popular hyperparameter tuning techniques are embarrassingly parallel.

Once a model has been trained, we would naturally be interested to understand why it performs the way it does. Ablation studies are a simple yet powerful approach for answering such questions. The idea is simple: you systematically remove or modify certain components, re-train the model, and observe how it behaves. Ablation studies can be performed on any component of an ML system, e.g., layers of a DNN, specific features of the training dataset, or parts of the training pipeline configuration. Despite their simplicity and usefulness, ablation studies can be tedious and resource-intensive. Running them often requires multiple versions of the code (which, are also very similar to the model training code), re-training models with different configurations, and analyzing the results, all of which can introduce redundancy and maintenance overheads. Fortunately, ablation studies are also embarrassingly parallel, meaning multiple experiments can be run independently.

Observing these challenges led us to come up with our first contribution, distribution oblivious training functions [1]. This basically meant that the practitioners and researchers could refactor the code that they were used to write daily (usually in Jupyter Notebooks), and make it ready to be run on a cluster of computers - i.e., in a distributed manner. To showcase the potential of this approach, we developed and released an opensource Python framework for asynchronous and parallel hyperparameter tuning, called MAGGY [2], and then the first dedicated framework for automated parallel ablation studies in DL, called AUTOABLATION, built on top of MAGGY. We published the AUTOABLATION paper in EuroMLSys 2021 [3], in which we also proposed a formal definition of ablation studies in the context of ML and DL.

After that, in 2022, we identified a possible source of gains in one of the standard practices of distributed training of DNNs, known as Data-parallel Training (DPT). By then, it was already well-known that the different examples in a training dataset (think different images in a dataset of images)

are not equal in terms of their difficulty or importance for the model that would be trained on them, e.g., to learn how to correctly classify unseen images after being trained on a training dataset of images. The model usually finds some of the training images (examples) harder to learn, and some of them easier. We saw an opportunity to use *example importance* to replace the standard way the dataset examples were being partitioned across the different workers, which was at random: we came up with a couple of dataset partitioning heuristics and showed that maintaining the example importance across the workers, instead of randomly assigning the examples to the workers during training, can lead to better models and faster accuracy milestones - and on top of that, it can substantially reduce the Input/Output (I/O) and network overhead of the cluster used for model training. This resulted in our importance-aware DPT paper [4], which also won the Best Artifact Award at DAIS 2023.

Later, in 2023, I personally became interested in the idea of *reusing* the computations and results of different DL pipelines in each other. While working on an idea to exploit computation reuse for the generation and scheduling of ablation study experiments, we came up with a simple yet effective weight initialization approach for model training, which reuses what the model has learned during the hyperparameter tuning step that happens prior to the model training step, in order to initialize the model weights for the training step. Let me elaborate on this.

So, typically, during hyperparameter tuning, in each trial, we partially train a model on a subset of the training dataset. Then, once we find suitable hyperparameter values (and our winning hyperparameter trial), in the model training step we fully train the same model from the winning trial, this time on the full training dataset. Why not reuse what the model has already learned during the winning hyperparameter tuning trial? after all, we have to initialize the model weights once again to start the full training process. Would this head-start make the model more vulnerable to pitfalls such as falling into local minima²? or could it lead to a reduction in training time, or maybe an even better-trained model? to answer these questions, we performed an experimental evaluation on a number of benchmark models and datasets used for image classification. Our results show that for many of these combinations, we can expect training speedup while maintaining or even improving the model performance [5].

 $^{^{2}}$ In learning based on Stochastic Gradient Descent, the model is basically trying to navigate its way around a loss landscape that can be considered as mountainous terrain, and the goal is to descend to the lowest possible valley (the global minimum of the loss function). However, along the way, the model may get stuck in smaller dips (local minima) that are not the absolute lowest point.

It was now the summer of 2024, and Large Language Models (LLMs) were all the rage, so I also started to think about possible applications of LLMs in my research work. It then occurred to me that back then, when we were developing AUTOABLATION, I used to wish there would be an easier and more generalized way to automate model and dataset manipulation (we had to manually add support for each new component and each underlying DL framework that we wanted to use with AUTOABLATION). Thankfully, LLMs, by this point had become increasingly good and much more reliable at generation and modification of ML/DL code, and ablation studies could be a very fit use-case for them. So, we developed a tool called ABLATION-MAGE, which generates the corresponding ablation study artifacts using natural language *hints* provided by the users in their code [6].

Now, this was a quick overview of my research work and the journey in which my colleagues and I made a bunch of contributions to the field. If you are interested in a more formal and detailed discussion of my research during my doctoral studies, I invite you to read the rest of this doctoral dissertation.

Acknowledgements

In yours truly's humble opinion, acknowledgements are by far the best part of any doctoral dissertation. I may have never read a doctoral thesis in full (except my own!), but I can assure you that I have read hundreds and hundreds of acknowledgment sections. It's the only part of the thesis that is solely about the student and those around them, and I'm always up for some good stories!

The thesis you are about to read is the culmination of the last five and a half years of my life in terms of research. I always tried to make my best effort to maintain what I consider academic and scholarly integrity and reproducible science, and to avoid making any conclusions that are not supported by evidence. That's why you'll see lots of box plots and repeated experiments in my papers, as well as all the publicly available code and experiment results.

But I could never have done this on my own. Doing a Ph.D. is not easy unless you are a genius, and it takes its toll also on the people around you: your supervisors, your colleagues, your friends and support groups, and your family. I tried not to forget this, but I'm sure I've let out more steam than I should have on the people around me - and yet you stuck with me to the very end.

To my main supervisor, Vlad: thank you for the undoubted trust you put in me even before I formally started my Ph.D. under your supervision, for teaching me how to do proper academic research, for your patience with my missed deadlines, and for your words of encouragement when I needed them most. Throughout the years, as is customary between any student and their supervisors, we've had many arguments and disagreements, but never once did I doubt for a single moment that you are a kind, caring academic father. And, of course, I will never forget October 2022, when I was on the verge of a nervous breakdown (something I never imagined could happen to me in a million years) from all the horrible news coming from Iran. You helped me get myself back together as a person. I want you to know that you were among a very rare group of supervisors who did such a thing for their students, and for this and so many other things, I am grateful to you.

To Amir: back in 2014, I was a last-year bachelor student, fascinated with everything computers but lacking the ability to focus on a single subject, who could not decide what he'd like to work on. One of the luckiest things that has happened to me is that you decided to come back to Tehran Polytechnic, and I was fortunate to be your student and TA and do my undergrad thesis under your supervision. Eleven years later, you have been not only my co-supervisor and mentor but also my brother with a heart of gold. All these years, I never once doubted that I could rely on you. Amir, I hope I have finally been a good student (not missing this final deadline!). Nevertheless, I'm proud to have been your student, and your friend. You told us about what Christian said to you: "Always be kind to the students." I don't think anyone can argue against the fact that you have always been kind to us. Thank you!

To Jim: you put your trust in me in 2018 and gave me the opportunity to work with you at Hopsworks on my M.Sc. thesis. And then I became your Ph.D. student. Throughout these years, any 15-30 minute meeting with you would set the course of my work for the subsequent months. Many of the ideas in my papers were initiated in one of our one-on-one discussions. Thank you for everything.

I would also like to express my gratitude to all those involved in the process of my Ph.D. defense. I am especially thankful to Associate Professor Salman Toor for accepting the role of my thesis opponent, and I look forward to our discussion. I am also deeply grateful to the members of my grading committee: Professor Keijo Heljanko, Associate Professor Marina Papatriantafilou, and Associate Professor Monowar Bhuyan, for their time and effort in evaluating my work. Special thanks go to Associate Professor Hossein Azizpour for his advance review of my thesis and his detailed feedback. Lastly, I'm grateful to Professor Sarunas Girdzijauskas for taking the time to serve as the Chair of my defense session.

Now, to those who have had the greatest impact on my life over these years.

Zahra, thank you for your unconditional love and support. Hearing words of encouragement from a "proper" doctor has always been helpful to me. Thank you for your pure kindness, for effortlessly bringing smiles to my face, for putting up with my messed-up sleep schedule, and for being my bestest friend.

Sara, thank you for being the kind and amazing person that you are. You have always been there for me since I moved to Stockholm in 2018 and have made everything so much easier than it could have been. I look forward to many more gossip sessions, many more trips and get-togethers, and who knows, maybe a research collaboration.

Avenia, thank you for always believing in me, always making sure that I'm doing okay with my life, putting up with my jokes, and being "Avenia kind". I look forward to more weekend brunches, and deep discussions about the meaning of life, the universe, and everything (together with Sara).

Samie, thank you for being a brother, best friend, and partner-in-crime for me and, again, for putting up with my jokes. I'm looking forward to getting certified in yet another means of transportation alongside you. Parmiss, among other things, thank you for always looking out for me, for being a beacon of reliability in our adventures, and for always organizing amazing trips.

Tianze, what a journey it has been! We became classmates back in Eindhoven in 2017, and we ended up working in the same office as Ph.D. students. I have rarely met someone who is as smart and knowledgeable, and yet as humble as you. Thank you for all the daily support, and for your invaluable contributions to the work included in this thesis.

Hamid, thank you for our many many walks around Lappis Beach, our daily commutes to and back from Electrum, for all the fruitful discussions (I'm pretty sure there have been quite a few times in which I've gone "Wow this is what they call a fruitful discussion!"), and for being a great friend.

Mohammad, thank you for being the role model of a scientist that you are, but most importantly for being an amazing friend. It's a privilege to have a friend as thoughtful and caring as you.

Pedram, I have always enjoyed discussing life, the economy, science, general news, our childhood, and all the gossip, for that matter, with you. Thank you for all this and for so much more, and for making the effort and visiting us in Stockholm *twice* (so far) while we still have not been to Toulouse.

Shoutout to all my friends, fellow students, and colleagues at SCS and in Electrum, and in particular Tianze, Susanna, Hamid, Sana, Zainab, David, Fabian, Francisco, Desta, Amirhossein, Shirin, Sara, Klas, Vangjush, Javier, Sameen, Vahid, Massimo, Mohammad, Edward, Amir, Paris, Theofilos, Kim, and Amir Mahmood. Thanks to Moritz Meister for being an amazing colleague and collaborator, Seif Haridi for his encouragement and kind advice, and Fredrik Lundevall for all the stories over lunch. Special thanks to Ahmad Al-Shishtawy, for being the reviewer of my half-time and 80% seminars.

Thanks to Thomas Sjöland for always supporting us at SCS and all the lessons and tales on the history of Sweden and KTH. I would also like to thank our HR officers, Madeleine Printzsköld, Karl Henning, and Fanny Sjöström, and our doctoral education support staff, Susy Mathew, Emanuele Borg, and Emmy Axén. I'm also thankful to Joanna Leksell and Göran Olofsson of EIT Digital for their kindness and help over the years.

Susanna and Saumey, serving with you at the EECS Ph.D. Council has been the most fun (and perhaps the most impactful) part of my doctoral studies. It has been an honor. I'll never forget the gossip, the "Live-WhatsApping" of meetings in which one of us was absent, and I'm proud of what we, together with our selfless and amazing Ph.D. representatives, were able to achieve.

Daniel, the same appreciation goes to you; it has been a privilege serving with you on the Chapter Board. Susanna, Saumey, Daniel, Mattias, and all the past and present members of DrInK, thank you for all the amazing events at the Chapter Hall, the crowded Pubs, the unforgettable Karaoke Nights, the Gasques, and for everything you have done for the Ph.D. students at KTH.

I want to thank my friends in Stockholm (and around) for all the happy memories we made together and for their constant support: Ara, Rezvaneh, Dimi, Parastu, Albin, Amir, Shahab, Romina, Mehdi, Farhad, Shima, Saba, Niloofar, Aida, Oskar, Andrea, Babak, Neshat, Razieh, Aref, Shamim, and Louis.

Now, to my friends across the oceans, across the seas... you know how much you mean to me. Thank you for keeping up with me and my occasional nagging: Shaghayegh, Sasan, Hamid, Nima, Navid, Sepand, Kurosh, Pegah, Reyhaneh, Motina, Mohammad, Keyvan, Kiana, Ata, Mahdi, Armin, Farzad, Aida, Mohammad, Mahdi, Sumeet, Dipika, Davor, Hamidreza, Hoda, and Zahra.

I'm also grateful to my relatives outside of Iran for always checking on me, visiting me from time to time, and providing support and a place to stay whenever I needed to get away from Stockholm. Uncle Mostafa, Aunt Naeimeh, Dayi Siavash, Aunt Shina, Sahar, and Cyrus - thank you! I'm also grateful to my Aunt Zahra and Aunt Javaher and I cherish the memories of my beloved Aunt Pira and Bababoo, both of whom I'm sure would be very happy to know that I have finished my studies.

Now, to the people to whom I owe everything I have in my life and who have been there for me since day one. Words will surely fall short, so I'll keep it concise. Sidi, thank you for being the kindest person I have ever known and for being the coolest maman I could have wished for. Hadi, my personal hero and role model, apart from everything else, you know you are the reason I got interested in academia. Thank you for everything you and maman have done for me. You two are the reason for everything I have and everything I will ever have in my life. Being your son is enough to call myself the luckiest person ever. Dorsa, I could not have asked for a better sister than you. However stubborn I may seem, I have always looked up to you, and your love and kindness have always warmed my heart. Thank you for all you have done for me. Aryan, my "legal brother", thank you for your kindness, encouragement, and all the laughs. And to my cousin Arman, thank you for always being the kind and big brother that you are.

Wow! That was quite a lot of acknowledgements, right? I can't complain, though; I've been surrounded by so many amazing friends. In the end, I dedicate this doctoral dissertation to all the researchers who practice open science, reproducible research, and, in particular, my fellow Ph.D. students who publish their code and experimental data, even though no one asks them to, and even though it often goes unrecognized. You are awesome. I love you.

List of Papers

Paper P1

Meister, M., Sheikholeslami, S., Andersson, R., Ormenisan, A. and Dowling, J.

"Towards Distribution Transparency for Supervised ML with Oblivious Training Functions".

Workshop on MLOps Systems, co-located with the Third Conference on Machine Learning and Systems (MLSys). March 2020

Paper P2

Meister, M., Sheikholeslami, S., Payberah, A.H., Vlassov, V. and Dowling, J.

"Maggy: Scalable Asynchronous Parallel Hyperparameter Search".

Proceedings of the 1st Workshop on Distributed Machine Learning (DistributedML), co-located with The 16th International Conference on Emerging Networking Experiments and Technologies (CoNEXT) (pp. 28-33). December 2020

Paper P3

Sheikholeslami, S., Meister, M., Wang, T., Payberah, A.H., Vlassov, V. and Dowling, J.

"AutoAblation: Automated Parallel Ablation Studies for Deep Learning". Proceedings of the 1st Workshop on Machine Learning and Systems (EuroMLSys), co-located with The 16th European Conference on Computer Systems (EuroSys) (pp. 55-61). April 2021

Paper P4

Sheikholeslami, S., Payberah, A.H., Wang, T., Dowling, J. and Vlassov, V. "The Impact of Importance-Aware Dataset Partitioning on Data-Parallel Training of Deep Neural Networks".

Proceedings of the 23rd IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS) (pp. 74-89).

P Best Artifact Award

June 2023

Paper P5

Sheikholeslami, Wang, T., A.H., Wang, T., Dowling, J. and Vlassov, V.
"Deep Neural Network Weight Initialization from Hyperparameter Tuning Trials".
Proceedings of the 31st International Conference on Neural Information Processing (ICONIP).
December 2024

Paper P6

Sheikholeslami, S., Ghasemirahni, H., Payberah, A.H., Wang, T., Dowling, J. and Vlassov, V. *"Utilizing Large Language Models for Ablation Studies in Machine Learning and Deep Learning"*.
Proceedings of the 5th Workshop on Machine Learning and Systems (EuroMLSys), co-located with The 20th European Conference on Computer Systems (EuroSys).

March 2025

Acronyms

Acronyms

- AI Artificial Intelligence
- ${\bf API}$ Application Programming Interface
- **DL** Deep Learning
- $\mathbf{DNN}\,$ Deep Neural Network
- ${\bf DPT}\,$ Data-parallel Training
- ${\bf GPU}$ Graphics Processing Unit
- $\mathbf{I/O} ~ \mathrm{Input/Output}$
- ${\bf LLM}\,$ Large Language Model
- $\mathbf{ML}\,$ Machine Learning

Contents

Ι	Thesis Overview	1				
1	Introduction					
	1.1 Research Objectives and Challenges	5				
	1.2 Thesis Contributions	8				
	1.3 Research Methodology	10				
	1.4 Sustainability and Social Aspects	11				
	1.5 Publications	12				
	1.6 Software	15				
	1.7 Dissertation Organization	16				
2	2 Background and Related Work 1					
	2.1 Machine Learning Systems	17				
	2.2 Machine Learning Pipelines	18				
	2.3 Hyperparameter Tuning and Optimization	20				
	2.4 Weight Initialization Techniques	21				
	2.5 Distributed Model Training	22				
	2.6 Ablation Studies	23				
	2.7 Machine Learning Tools and Frameworks	24				
3	Summary of Appended Papers 2'					
4	Conclusions and Future Work	35				
	4.1 Dissertation Summary	35				
	4.2 Broader Impact	38				
	4.3 Future Work	39				
Bibliography						
II Appended Papers 47						

Part I Thesis Overview

Chapter 1

Introduction

Deep Neural Networks (DNNs) have been at the forefront of recent breakthroughs in Machine Learning (ML) and Deep Learning (DL). DNNs are increasingly used in various tasks, from Earth observation and analysis of satellite images to medical diagnosis and smart chatbots. A major contributor to these advances has been the abundance of training data, computation resources, and frameworks that enable efficient training of ever-larger and more complex DNNs, within a paradigm referred to as distributed DL, and in particular, *distributed training* which is the focus of this doctoral dissertation. In distributed training, the data and computation are distributed across several workers as opposed to single-host training in which both the data and computation reside and happen on a single worker. In particular, in this doctoral dissertation we consider a worker as a computing entity that trains a DNN in collaboration with other similar entities. In a machine with one or multiple Graphics Processing Units (GPUs), or in a cluster of several machines with one or multiple GPUs, we typically consider the process running on each GPU as a separate worker. In this setting, distributed training can help overcome the limitations of single-host training, such as memory constraints, computational bottlenecks, and data availability [7]–[11].

However, distributed training comes with a number of challenges that need to be carefully addressed in order to have a system that efficiently makes use of it. These challenges include, but are not limited to, efficient distribution of computation and data across the workers, the presence of straggler workers in a cluster (workers that, compared to the other workers in a cluster, get significantly behind in their computation task), especially in synchronous execution settings, and communication and synchronization among the workers. Also, some ML/DL experiments such as hyperparameter tuning and ablation studies consist of trials that have different resource requirements. This implies that the system should provide *scalability* in both the computation and the data dimensions.

CHAPTER 1. INTRODUCTION

On the other hand, from a programming point of view, using the distributed training paradigm typically requires knowledge of distributed computing principles and experience with distributed and data-intensive computing frameworks as well as applying major changes to the code used for single-host training [12]. Furthermore, as training a DNN involves several steps and stages (e.g., data preparation, hyperparameter tuning, model training, etc.), it would be desirable to possibly reuse the computational results of different steps in each other (e.g., reusing weights learned during hyperparameter tuning trials, for weight initialization of the model training step) in order to improve training time. Finally, when developing larger and more complex DNNs, new training approaches, or in general, new ML systems, in order to reason about the different components of the system, we also need to know about each design choice's contributions to the performance of the system.

Therefore, we need tools and methods that can (i) efficiently and effectively distribute the training workload and its components (e.g., the dataset, the model, the computation) across different workers while maintaining the quality and convergence of the model compared to single-host training, and (ii) help us inspect the relative impact of different components of an ML system to its performance.

In this doctoral dissertation, I present my contributions, listed below, in developing tools and methods for distributed and large-scale ML and DL:

- C1. **Distribution-oblivious training functions** [1], a programming abstraction to unify single-host and distributed ML/DL training functions,
- C2. **MAGGY** [2], an open-source framework for asynchronous, parallel hyperparameter tuning that can run ML/DL code on top of Apache Spark using distribution-oblivious training functions,
- C3. **AUTOABLATION** [3], the first dedicated framework for automated design and parallel execution of ablation study experiments in ML and DL,
- C4. Importance-aware Data-parallel Training (DPT) [4], a novel dataset partitioning approach for DPT of DNNs that considers dataset example importance, which compared to random partitioning in vanilla DPT, can reduce model training time and network and Input/Output (I/O) overhead while maintaining model accuracy,
- C5. A novel weight initialization approach for training DNNs that reuses computation results from the hyperparameter tuning step and can reduce the training time while maintaining or improving model accuracy compared to random weight initialization [5], and

C6. **ABLATIONMAGE** [6], a tool based on Large Language Models (LLMs) that further simplifies the conduction of ablation studies.

Also, distribution-oblivious training functions (C1), MAGGY (C2) and AUTOABLATION (C3) have been integrated into Hopsworks [13], a platform for ML/DL, as presented in [14]. The Hopsworks platform has been used in many application domains, e.g., finance, healthcare, recommender systems, and in particular, for Earth observation [15] in the ExtremeEarth project [16].

This chapter serves both as an introduction and an overview of this doctoral dissertation. I will start by describing the main research objectives and challenges addressed in this doctoral dissertation, and then summarize our primary contributions. I will then discuss the research methodologies used for this work, and go over other important aspects of my doctoral research, in particular, the broader research and sustainability impacts of my work. I will then list the included publications and related software artifacts and finally describe the structure of the doctoral dissertation.

1.1 Research Objectives and Challenges

The primary focus of the research work included in this doctoral dissertation is to develop, enhance and improve tools and methods for training DNNs at scale, addressing challenges and obstacles related to efficient distributed training, scalability of ML/DL workflows and experiments, reusability of computation in ML/DL pipelines, and automation of ablation studies. To this end, we pursue the following research objectives.

01. Develop Tools and Frameworks for Scalable, Distributed DNN Training Experiments.

The training stage of DNNs includes several steps such as hyperparameter tuning, ablation studies, and the actual model training. A particular challenge here is the effect of straggler workers on the overall wall-clock time of experiments [17], especially when the underlying computation follows execution models similar to bulk synchronous parallel [18], [19]. Another challenge, from a usability perspective, is that the current distributed training techniques often require significant code refactoring, making it difficult for practitioners to transition from single-host training. A similar usability challenge also exists when it comes to moving between different steps of ML/DL workflows; e.g., moving from hyperparameter tuning to model training or ablation studies results in redundancy and duplication of the codebase. Hence, tools and frameworks should provide usable programming abstractions

CHAPTER 1. INTRODUCTION

that make it easier for developers to write distributed training code, transform their single-host code to distributed code, or move between different steps of an experiment with minimal effort.

Challenges to be Addressed. The key challenges in scalable distributed DNN training experiments addressed by this objective are: (i) Straggler Mitigation – minimizing the impact of slow or delayed workers to improve overall training efficiency in distributed settings, and (ii) Usability and Abstraction – designing intuitive programming abstractions that simplify the transition from single-host to distributed training, reducing developer effort and complexity.

Approach and Contributions. To address the above challenges, this objective refers to developing tools and frameworks that can enable experiments related to training large DNNs over large datasets efficiently in a distributed manner, i.e., can run large-scale ML/DL experiments across multiple workers, while improving the usability for developers and practitioners by reducing the redundancy and complexity of the codebase. As part of our contributions, we design and develop tools and frameworks that address these issues. These include distribution-oblivious training functions, a programming abstraction to unify single-host and distributed ML/DL training functions (C1), MAGGY, a framework for asynchronous parallel hyperparameter tuning (C2), and AUTOABLATION, a framework enabling automated parallel ablation study experiments for ML/DL (C3). These contributions (C1-C3) were also integrated into the Hopsworks platform [14].

Related publications included in the doctoral dissertation:

- P1. "Towards Distribution Transparency for Supervised ML with Oblivious Training Functions" [1] (2020)
- P2. "MAGGY: Scalable Asynchronous Parallel Hyperparameter Search" [2] (2020)
- P3. "AUTOABLATION: Automated Parallel Ablation Studies for Deep Learning" [3] (2021)

O2. Enhance the Efficiency of Existing DNN Training Techniques.

Distributed training is an already well-established paradigm and consists of several techniques and approaches used by researchers and practitioners on a daily basis [8], [9]. However, the standard techniques may not be optimized, particularly with respect to computation efficiency and reuse. Hence, improving these techniques in order to make them more efficient for large-scale workloads can have a significant impact on the amount of time and compute resources used for DL experiments. **Challenges to be Addressed.** We focus on two key challenges: (i) Inefficiencies in Distributed Training, and (ii) Limited Computation Reuse Across Pipeline Steps. The former challenge refers to the fact that standard distributed training techniques may not be fully optimized in terms of computation, network usage, and I/O efficiency, leading to unnecessary overhead and prolonged training times. The latter challenge states that existing training approaches do not effectively reuse computations from prior steps, such as hyperparameter tuning, which could otherwise accelerate training and improve model performance.

Approach and Contributions. To address the aforementioned challenges, the contributions of this doctoral dissertation (i) improve the dataset partitioning for DPT, a popular distributed training technique, in order to reduce the training time as well as network and I/O overhead (C4), and (ii) introduce a novel weight initialization for DNN training that reuses computations from the hyperparameter tuning step, which can lead to speed-ups in training, and potentially better performing DNNs (C5).

Related publications included in the doctoral dissertation:

- P4. "The Impact of Importance-Aware Dataset Partitioning on Data-Parallel Training of Deep Neural Networks" [4] (2023)
- P5. "Deep Neural Network Weight Initialization from Hyperparameter Tuning Trials" [5] (2024)

O3. Improve the state of Ablation Studies in ML and DL.

Ablation studies, in which different components of an ML system (e.g., dataset features or model layers) are removed in order to understand their individual effects on the overall performance of the system, are an arguably necessary but often overlooked part of ML/DL research practice [3], [20]–[22]. This status is largely due to the fact that the typical practices for performing ablation studies incur additional costs (both in time and computing) and manual effort, as they require performing extra experiments and maintaining multiple versions of the code needed for defining and training the different system configurations. So, it would be desired to both make it easier for practitioners and researchers to conduct ablation studies and, at the same time, reduce the costs associated with them.

Challenges to be Addressed. The key challenges in improving ablation studies in ML and DL are the following: (i) High Implementation Overhead – conducting ablation studies requires significant manual effort, including maintaining multiple code versions for different ablation

CHAPTER 1. INTRODUCTION

trials, and (ii) Computational and Time Constraints – ablation experiments are resource-intensive, demanding additional time and computational power, leading many researchers to forgo them in their research projects. At the same time, the specific challenges discussed in relation to O1 (i.e., Straggler Mitigation and Usability and Abstraction) apply to this area as well. Any parallelization of ablation trials should account for stragglers and uneven resource requirements between trials, since model training with different model, dataset, or system configurations requires different amounts of time and computation. Moreover, there is a shortage of dedicated frameworks for ablation studies in ML/DL research that can provide abstractions to avoid redundant code maintenance.

Approach and Contributions. To address the aforementioned challenges and fulfill this objective, we provide two related contributions: AUTOABLATION, the first dedicated framework for ablation studies in ML/DL, which enables automated parallel ablation study experiments (C3), and ABLATIONMAGE, a tool that utilizes LLMs for automation of ablation study experiments (C6).

Related publications included in the doctoral dissertation:

- P3. "AUTOABLATION: Automated Parallel Ablation Studies for Deep Learning" [3] (2021)
- P6. "Utilizing Large Language Models for Ablation Studies in Machine Learning and Deep Learning" [6] (2025)

1.2 Thesis Contributions

Overall, the contributions of this thesis advance the field of distributed DL with a number of integrated tools and methods designed to improve the efficiency, scalability, and usability of distributed ML/DL workflows.

Our first contribution, distribution-oblivious training functions (C1) [1], is an abstraction for ML/DL training code that enables reusing the same code in both single-host and distributed execution settings. We then use this abstraction as the foundation for the programming model of **MAGGY** (C2) [2] and **AUTOABLATION** (C3) [3]. The former is an opensource framework for asynchronous, parallel hyperparameter tuning that can run arbitrary Python code on top of Apache Spark and increases the resource utilization of GPU clusters since its asynchronous execution provides straggler-tolerance, and the latter extends MAGGY with support for automated parallel ablation studies. Distribution-oblivious training functions



Figure 1.1: Categorization of thesis contributions as tools or methods.

(C1), MAGGY (C2) and AUTOABLATION (C3) have also been integrated in the Hopsworks platform for ML/DL [14].

Our fourth contribution, **importance-aware data-parallel training** (C4) [4], is a novel approach for partitioning dataset examples across the workers in a DPT setting, which considers the importance of dataset examples rather than simple random partitioning. This approach can lead to both speed-up and better best test accuracy of the trained models, as well as a significant reduction in the network and I/O overhead of the cluster.

We then investigate the idea of *reusing* the computation results of one DL stage or step into another one. In particular, we propose a simple yet effective **weight initialization method** for training DNNs that reuses the weights learned during the hyperparameter tuning trials (C5) [5]. Through a number of experiments on benchmark image classification models and datasets, we show that this weight initialization method can lead to speed up in the training stage, while maintaining or even improving the best test accuracy of the trained models, compared to when random initialization is used.

Finally, witnessing the rise in the capability of LLMs, particularly in generating ML/DL code, we utilize them in creating another tool for automation of ablation studies called **AblationMage** (C6) [6].

To summarize, the contributions of this doctoral dissertation collectively optimize large-scale DNN training, making it more accessible, efficient, and computationally sustainable while reducing the redundancy in ML/DL workflows, and providing usable tools for conducting ablation studies. Figure 1.1 and Table 1.1 provide the summary and categorization of these contributions. As a reminder, the research objectives pursued in this doctoral dissertation are O1: develop tools and frameworks for scalable, distributed DNN training experiments, O2: enhance the efficiency of existing DNN training techniques, and O3: improve the state of ablation studies in ML and DL.

Contribution	Type	Objective	Main Idea
Distribution-	Method	01	A programming abstraction to
oblivious Training			unify single-host and distributed
Functions $(C1)$ [1]			ML/DL training functions
$M_{ACCV}(C2)[2]$	Tool	01	A framework for asynchronous,
[MAGG1 (02) [2]]			parallel hyperparameter tuning
ΔυτοΔειάτιον		01, 03	A framework for automated
(C3) [3]	Tool		parallel ablation study
			experiments
Importance-aware	Method	O2	Replacing random dataset
Data-parallel			partitioning with example
Training $(C4)$ [4]			importance-aware heuristics
Weight	Method	O2	Bonlacing random initialization
Initialization from			of DNNg with weights loorned
Hyperparameter			during hyperparameter tuning
Tuning Trials			triala
(C5) [5]			tilais
AblationMage	Tool	O3	Utilizing LLMs for conducting
(C6) [6]			ablation studies

Table 1.1: Summary and categorization of the contributions

1.3 Research Methodology

The research carried out for this doctoral dissertation spans both (i) designing and implementing new tools and frameworks for ML workflows, as well as (ii) improving existing tools, methods, and algorithms. Hence, we employ principles from design science research [23] paired with experimental validation. For each contribution of this doctoral dissertation, we started by identifying specific gaps in existing workflows, tools, and methods (e.g., the need for distribution transparency in ML frameworks, or exploring the potential benefit of considering example importance in dataset partitioning in DPT). When developing the tools and frameworks, we followed best practices from the leading ML frameworks such as Keras [24] (e.g., providing high-level abstraction and modularity). We would then evaluate each framework, tool, or method, with several experiments to verify their usability and proposed impact. Throughout all the papers, we attempted to use well-known benchmark datasets (such as CIFAR-10/100) and models (such as the ResNet [25] or Inception [26] family of models) as well as applicationspecific datasets (e.g., the TenGeoPSAR dataset of satellite imagery [27]) and followed the standard evaluation practices in ML and DL research.

It is worth mentioning that all developed tools and frameworks, raw experimental results of the publications, and scripts for reproducing those experiments have been publicly released following the publication of each paper, and the artifact from one of the publications (Importance-aware DPT [4]) won the best artifact award of DAIS 2023.

1.4 Sustainability and Social Aspects

It is estimated that the energy consumption and CO_2 emissions associated with training state-of-the-art Artificial Intelligence (AI) models have been doubling every four to six months in recent years [28]. Since this doctoral dissertation focuses on tools and methods related to training DNNs at scale, it contributes to sustainable computing primarily by reducing the time, energy, and resource costs associated with large-scale ML/DL experiments.

Distribution transparency (C1) enables parallelization of ML experiments that otherwise would require significant effort from developers. Asynchronous parallel execution of hyperparameter tuning and ablation study trials (C2 & C3) paired with support for early stopping provided by MAGGY greatly increase the resource utilization of power-hungry GPU clusters, while reducing the wall-clock time of experiments. When it comes to importance-aware DPT (C4), refraining from repartitioning and redistributing the dataset across the workers at each epoch can significantly reduce the network and I/O overhead. Reusing the computation results (model weights) from the hyperparameter tuning trials (C5) can reduce the wall-clock time of the model training stage, hence reducing associated costs and energy consumption. Furthermore, our work on ablation study experiments in ML/DL (C3 & C6) contribute to the scientific experiment lifecycle in ML and DL research, by providing easy-to-use and efficient tools and frameworks to significantly reduce the manual labor associated with such experiments.

However, we should also consider that the research included in this doctoral dissertation introduces several potential risks despite its contributions to improving the efficiency and scalability of distributed DL. A general risk is that unintended energy consumption growth may occur due to improved training efficiency incentivizing larger-scale experiments, potentially offsetting sustainability benefits [29]. In energy economics, this is referred to as the rebound effect [30], [31].

Bias amplification is a concern in importance-aware DPT, where prioritizing informative examples might reinforce existing biases and lead to unfair model outcomes. Security risks in model reuse emerge from reusing weights from hyperparameter tuning trials, as adversarial vulnerabilities could persist in the final model. Lastly, automation risks with LLMs highlight the potential for errors or hallucinations in ML workflows if LLM-generated modifications to ML code are not thoroughly validated.

To mitigate these risks, future work can incorporate privacy-preserving techniques (e.g., differential privacy and federated learning), fairness assessments in dataset partitioning, robust verification mechanisms for LLMgenerated ablation studies, and responsible AI practices to promote sustainable AI development.

1.5 Publications

The following six peer-reviewed publications (P1-P6), listed below in chronological order of publication, are included in this doctoral dissertation.

P1. Meister, M., Sheikholeslami, S., Andersson, R., Ormenisan, A.A. and Dowling, J., 2020, March. Towards Distribution Transparency for Supervised ML with Oblivious Training Functions. Workshop on MLOps Systems, co-located with the Third Conference on Machine Learning and Systems (MLSys). [1]

Author Contribution The author of this doctoral dissertation contributed to the design of the abstraction and preparing the manuscript text.

P2. Meister, M., Sheikholeslami, S., Payberah, A.H., Vlassov, V. and Dowling, J., 2020, December. MAGGY: Scalable Asynchronous Parallel Hyperparameter Search. In Proceedings of the 1st Workshop on Distributed Machine Learning (DistributedML), co-located with The 16th International Conference on Emerging Networking Experiments and Technologies (CoNEXT) (pp. 28-33). [2]

Author Contribution The author of this doctoral dissertation contributed to the design and implementation of MAGGY, analysis of the results, and preparing the manuscript text. P3. Sheikholeslami, S., Meister, M., Wang, T., Payberah, A.H., Vlassov, V. and Dowling, J., 2021, April. AUTOABLATION: Automated Parallel Ablation Studies for Deep Learning. In Proceedings of the 1st Workshop on Machine Learning and Systems (EuroMLSys), co-located with The 16th European Conference on Computer Systems (EuroSys) (pp. 55-61). [3]

Author Contribution The author of this doctoral dissertation implemented AUTOABLATION as part of MAGGY, designed and performed the experiments, analyzed the results in collaboration with the coauthors, and wrote the majority of the manuscript.

P4. Sheikholeslami, S., Payberah, A.H., Wang, T., Dowling, J. and Vlassov, V., 2023, June. The Impact of Importance-Aware Dataset Partitioning on Data-Parallel Training of Deep Neural Networks. In Proceedings of the 23rd IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS) (pp. 74-89). [4]

Author Contribution The author of this doctoral dissertation discussed the idea of exploiting example importance for training speed up with the co-authors, proposed its usage for DPT, designed and performed the experiments, analyzed the results in collaboration with the co-authors, and wrote the majority of the manuscript.

P5. Sheikholeslami, S., Wang, T., Payberah, A.H., Dowling, J. and Vlassov, V., 2024, December. *Deep Neural Network Weight Initialization from Hyperparameter Tuning Trials*. In Proceedings of the 31st International Conference on Neural Information Processing (ICONIP). [5]

Author Contribution The author of this doctoral dissertation discussed the idea of reusing computation results from different DL steps in each other with the co-authors, proposed the weight initialization approach, designed and performed the experiments, analyzed the results in collaboration with the co-authors, and wrote the majority of the manuscript.

P6. Sheikholeslami, S., Ghasemirahni, H., Payberah, A.H., Wang, T., Dowling, J. and Vlassov, V., 2025, March. Utilizing Large Language Models for Ablation Studies in Machine Learning and Deep Learning. In Proceedings of the 5th Workshop on Machine Learning and Systems (EuroMLSys), co-located with The 20th European Conference on Computer Systems (EuroSys). [6]

CHAPTER 1. INTRODUCTION

Author Contribution The author of this doctoral dissertation discussed the idea utilizing LLMs for ablations studies with the co-authors, implemented ABLATIONMAGE, designed and performed the experiments, and analyzed the results in collaboration with the co-authors, and wrote the majority of the manuscript.

1.5.1 Other Publications

In addition to the six included papers, I have contributed to the following peer-reviewed publications, which, though not included in this doctoral dissertation, have nevertheless had a significant impact on my research experiences during my doctoral studies.

- Angelovska, M., Sheikholeslami, S., Dunn, B. and Payberah, A.H., 2021. Siamese Neural Networks for Detecting Complementary Products. In Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop (pp. 65-70). [32]
- Hagos, D.H., Kakantousis, T., Vlassov, V., Sheikholeslami, S., Wang, T., Dowling, J., Fleming, A., Cziferszky, A., Muerth, M., Appel, F. and others, 2021. *The ExtremeEarth Software Architecture for Copernicus Earth Observation Data*. In Proceedings of the 2021 Conference on Big Data from Space (BiDS'21) (pp. 181-185). [15]
- Koubarakis, M., Stamoulis, G., Bilidas, D., Ioannidis, T., Mandilaras, G., Pantazi, D.-A., Papadakis, G., Vlassov, V., Payberah, A.H., Wang, T., Sheikholeslami, S. and others, 2021. Artificial Intelligence and Big Data Technologies for Copernicus Data: The ExtremeEarth Project. In Proceedings of the 2021 Conference on Big Data from Space (BiDS'21) (pp. 9-12). [16]
- Hagos, D.H., Kakantousis, T., Vlassov, V., Sheikholeslami, S., Wang, T., Dowling, J., Paris, C., Marinelli, D., Weikmann, G., Bruzzone, L. and others, 2021. *ExtremeEarth Meets Satellite Data from Space*. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 14, pp. 9038-9063. [33]
- Asratyan, A., Sheikholeslami, S. and Vlassov, V., 2021. A Parallel Chain Mail Approach for Scalable Spatial Data Interpolation. In 2021 IEEE International Conference on Big Data (BigData) (pp. 306-314). [34]
- Hagos, D.H., Kakantousis, T., Sheikholeslami, S., Wang, T., Vlassov, V., Payberah, A.H., Meister, M., Andersson, R. and Dowling, J.,

2022. Scalable Artificial Intelligence for Earth Observation Data Using Hopsworks. Remote Sensing, 14(8), p. 1889. [14]

- Chikafa, G., Sheikholeslami, S., Niazi, S., Dowling, J. and Vlassov, V., 2022. Cloud-Native RStudio on Kubernetes for Hopsworks. Advances in Parallel & Distributed Processing, and Applications. Springer, Cham. [35]
- Johannesson, T., Rubensson, I., Sheikholeslami, S., Al-Shishtawy, A. and Vlassov, V., 2024. DUGET: Leveraging Machine Learning for Dynamic User Grouping and Evolution Tracking in Public Transit Systems. In 2024 IEEE International Conference on Big Data (Big-Data) (pp. 1785-1794). [36]
- Xu, Z., Nordström, P., Sheikholeslami, S., Al-Shishtawy, A. and Vlassov, V., 2024. A Semi-Supervised Model for Non-Cellular Elements Segmentation in Microscopy Images of Wood. In 2024 IEEE International Conference on Big Data (BigData) (pp. 2049-2056). [37]

1.6 Software

In alignment with the principles of open science, all contributions of this doctoral dissertation are accompanied by publicly available, open-source implementations and reusable experimental artifacts.

- MAGGY implements distribution-oblivious training functions as its programming model and is available as open-source software on GitHub 1 .
- AUTOABLATION extends MAGGY to provide support for parallel ablation studies. The instructions and artifacts to reproduce the experiments from the AUTOABLATION paper [3] are also available on GitHub².
- We released a proof-of-concept extendable PyTorch implementation of importance-aware DPT in a GitHub repository ³. All the raw experimental results, the code to reproduce the experiments and the plots from the corresponding paper [4] are also available in the repository in form of CSV files, Python scripts, and Jupyter Notebooks. The artifact of this paper won the **Best Artifact Award of DAIS 2023**.

¹https://github.com/logicalclocks/maggy

²https://github.com/ssheikholeslami/ablation-paper-experiments

³https://github.com/ssheikholeslami/importance-aware-data-parallel-train ing

CHAPTER 1. INTRODUCTION

- We released a proof-of-concept extendable PyTorch implementation of weight initialization from hyperparameter tuning trials in a GitHub repository⁴. All the raw experimental results, the code to reproduce the experiments and the plots from the corresponding paper [5] are also available in the repository in form of CSV files, Python scripts, and Jupyter Notebooks.
- Finally, ABLATIONMAGE utilizes LLMs for conducting ablation studies in ML/DL. Our prototype implementation, as well instructions to reproduce the experiments from the corresponding paper [6] are available on GitHub⁵.

1.7**Dissertation Organization**

This doctoral dissertation is organized in the form of a compilation thesis and includes six of my peer-reviewed, published research papers (P1-P6) that form its second part (appended papers⁶). The first part, commonly known in Sweden as a *comprehensive summary* or *kappa*, shall serve as a brief overview of the research field and outline the contributions of the doctoral dissertation.

The remainder of the first part of this doctoral dissertation is organized as follows. The necessary background information for following along with the appended papers is provided in Chapter 2, and Chapter 3 presents summaries of the appended papers. Finally, Chapter 4 concludes the first part of the doctoral dissertation, where I also discuss potential directions for future research in related areas.

⁴https://github.com/ssheikholeslami/dnn-weight-initialization-from-hp-tun ing ⁵https://github.com/ssheikholeslami/utilizing-llms-for-ablation-studies

⁶The papers are reproduced verbatim from their published versions, except for layout and template adjustments. For P6, its preprint version accepted at EuroMLSvs is included.
Chapter 2

Background and Related Work

In this chapter, we provide the necessary background and related work for understanding the main contributions, of this doctoral dissertation. We assume the reader to have a general understanding of Machine Learning (ML) and Deep Learning (DL), in particular the workflow for training a Deep Neural Network (DNN). As more detailed background and related work are presented in dedicated sections of each of the included publications, here we focus on giving a holistic overview of the concepts related to training of DNNs that are the foci of the contributions of this doctoral dissertation.

We start by explaining the concept of ML Systems, and then define our understanding of ML pipelines. In particular, we discuss the *training pipelines*, which this doctoral dissertation largely focuses on, in more detail. Then, we will briefly introduce concepts and techniques such as hyperparameter tuning and optimization, weight initialization for training, distributed model training, and ablation studies in the context of ML and DL. Finally, we provide a quick overview of the major tools and frameworks we build on, or use, to implement our methods, with their design and features having significant implications for our design choices. These include Apache Spark, TensorFlow, and PyTorch.

2.1 Machine Learning Systems

An *ML system* can be defined as a computer system that contains models, algorithms, data, and computing capabilities to train, maintain, or serve ML models [38], [39]. Consequently, the term is used to refer to both ML-based products and services (e.g., a face recognition system), and the frameworks that are used for training, maintaining, and serving ML and DL models. However, in the context of ablation studies, any similar definition can be used as long as it describes a system that deals with ML models and con-

sists of several modular components that can be independently changed or removed.

That said, from an operational point of view, an ML system can contain a number of *pipelines*, which we will discuss next.

2.2 Machine Learning Pipelines

In the ML and DL community, the term *pipeline* is perhaps used in varied contexts to refer to different collections of entities, but we define an ML pipeline as a process that has one or several inputs (e.g., raw data, processed data, a pre-trained model, hyperparameters, etc.), and one or several outputs (e.g., preprocessed data, a trained model, a set of hyperparameter values, etc.) typically in form of *artifacts* that can either be used independently or as inputs to other pipelines¹ [40], [41]. Using pipelines in ML systems enables easier management and implementation of ML projects, and pipelines are usually designed and implemented in a way that makes them largely reusable in other ML-based projects and systems [42]. Each ML pipeline, in turn, can be broken down into several steps. Again, due to the difference in terminology, sometimes some of these steps are also referred to as distinct "stages", e.g., the "hyperparameter tuning stage".

That being said, while there are different definitions of ML pipelines, in this chapter we use a more recent definition that assumes three distinct grouping of ML pipelines: feature pipelines, training pipelines, and inference pipelines [43], [44]. Within this framework, our contributions are primarily to the training pipelines. We will now discuss each of the three pipelines in more detail.

2.2.1 Feature Pipelines

Simply put, feature pipelines have raw data as their input and features (and labels) as their main output [40], [44]. As these features and labels are then used as the input to training pipelines, feature pipelines typically include several steps to prepare raw data (that can come from multiple sources, in different formats, modalities, and intervals) for model training. These steps include, but are not limited to, data ingestion, data cleaning, data preprocessing, data transformation, data normalization, tokenization, augmentation, data validation, and versioning. Efficient feature pipelines not

¹One can say that a discussion topic within the ML systems community is on how these collections of entities, stages, and steps are defined, and what level of granularity should be used to refer to standalone pipelines. As an example, we can consider data ingestion and data cleaning as having their own separate pipelines or define them as individual steps within a feature pipeline.

only lead to better trained models, but also reduce computational overhead by eliminating redundant or irrelevant information. By automating these steps, feature pipelines ensure consistency across training and inference, enabling reproducibility and seamless integration into large-scale ML and DL systems, and across several teams within an enterprise, if needed.

2.2.2 Training Pipelines

In ML and DL experiments, the (model) training pipeline is the core of the workflow, responsible for training ML models and DNNs. For DNNs, this essentially means optimizing the weights of the model based on a number of examples in a training dataset so that the model performs well on a downstream task on unseen data as evaluated by its performance on a test set. This pipeline includes several steps, namely model definition and selection, hyperparameter tuning, weight initialization, and optimization techniques such as gradient-based learning. After a model is sufficiently trained, we might want to understand what contributes to its performance, and here, techniques such as ablation studies [3], [21] or explainability or interpretability methods are used [45].

Training modern DNNs is computationally intensive, often requiring strategies like distributed training [4], [46], mixed-precision arithmetic, and adaptive learning rates to balance efficiency and accuracy. Additionally, mechanisms such as early stopping, checkpointing, and automated logging help streamline the training process, ensuring that models converge effectively while minimizing unnecessary computation.

A well-structured training pipeline also integrates automated hyperparameter tuning to optimize parameters like learning rate, batch size, and regularization methods. Techniques such as Bayesian optimization, grid search, and asynchronous tuning frameworks — often combined with distributed computing — help speed up this search process [2]. Furthermore, techniques such as leveraging prior computation through reusing weights from successful hyperparameter tuning trials for weight initialization before model training, can accelerate convergence and improve model performance [5]. By implementing robust model training pipelines, researchers and practitioners can efficiently train deep learning models while ensuring scalability, reproducibility, and adaptability across different datasets and hardware environments.

2.2.3 Inference Pipelines

When the model is sufficiently trained and evaluated, it is time to put it to use, i.e., have it respond to *inference* requests. Thus, an inference pipeline is responsible for efficiently deploying trained DNNs to, e.g., make predictions on new data or generate new data instances based on the input data. Unlike training pipelines, which focus on efficient, fast, and accurate optimization and learning, the inference pipeline prioritizes speed, scalability, and resource efficiency to handle real-world applications in production settings. This pipeline includes model serialization, loading, and execution, often incorporating techniques such as model quantization, pruning, and hardware acceleration. For large-scale or latency-sensitive applications, inference pipelines may leverage techniques like batch processing, caching, and distributed serving frameworks to improve performance. Ensuring consistency between training and inference is crucial, requiring careful design and implementation of feature and training pipelines. By designing an optimized inference pipeline, DL models can be seamlessly deployed into production, delivering fast and accurate predictions while minimizing computational overhead.

2.3 Hyperparameter Tuning and Optimization

Hyperparameter tuning and optimization is a step (or stage) in a training pipeline that deals with finding the best set of hyperparameter values for the model or the training algorithm². Hyperparameters are configuration variables that influence the training process and the model's performance but are not directly learned from the data [47], [48]. Perhaps the most wellknown example of a hyperparameter is the *learning rate* of the optimization algorithm used for training; other examples include batch size, number of layers, and regularization parameters. The choice of hyperparameters can significantly impact the accuracy, convergence speed, and generalization ability of a model. However, this requires either domain expertise, or running several trials using a hyperparameter optimization algorithm. Hence, efficiently finding the best, or good enough sets of hyperparameter values is critical to an ML system [49].

Traditional methods for hyperparameter tuning, such as grid search and random search [50], can be computationally expensive and often inefficient, especially in the context of high-dimensional hyperparameter spaces. Grid search systematically explores predefined values, which can be exhaustive but lacks flexibility. Random search samples hyperparameter configurations randomly, providing better exploration in many cases but still being resource-intensive for large-scale problems. These methods can be considered "undirected search" methods, since they do not take into account the result of earlier trials for choosing the next set of values to evaluate. How-

²Or any other hyperparameters of the training pipeline.

ever, these methods can be enhanced in terms of efficiency using concepts such as early stopping of underperforming trials, or asynchronous parallel execution of trials. That being said, A major contribution of MAGGY [2] is that it enables asynchronous parallel execution of hyperparameter tuning trials, including trials using random search and grid search, with support for early stopping, on top of Apache Spark.

More recent approaches leverage past performance (i.e., the result of earlier trials) to suggest values for the next trials. Hence, these methods can be considered as "directed search" methods. A prominent example of these sets of methods is Bayesian Optimization and its different variants [51]. These methods can balance between "exploration" and "exploitation", making them more efficient for settings with limited computational budgets³.

In distributed and large-scale environments, hyperparameter optimization becomes even more challenging. Frameworks like Apache Spark [52] or PyTorch Lightning [53] enable parallel exploration of hyperparameter configurations, significantly reducing runtime. Asynchronous methods, such as ASHA [17], allow continuous evaluation of configurations, dynamically allocating resources to the most promising trials. These approaches reduce waste of time and compute resources, and improve scalability, enabling practitioners to tune hyperparameters effectively even for computationally expensive models or large datasets.

2.4 Weight Initialization Techniques

In a training pipeline, weight initialization is one of the last steps before the actual training of the model (on the full training dataset)⁴, and plays a crucial role in the training and performance of DNNs. It sets the starting point for the model optimization process, thereby significantly influencing the convergence speed and final accuracy of the model [54], [55]. Poorly chosen initial weights can lead to issues such as vanishing or exploding gradients, especially in deep architectures, hindering effective training.

Over the years, researchers have proposed various initialization strategies, such as random uniform and Gaussian initializations, to address these challenges. Some popular approaches, including Xavier [56] and Kaiming (He) initialization [57], tailor the weight distribution to the non-linearity of

³Consequently, setting "the knob" correctly (the trade-off between exploration and exploitation) can itself turn into a higher-level hyperparameter.

⁴As each hyperparameter tuning trial includes training of a model on a subset of the training set, we also have to initialize the weights. However, when talking about weight initialization in this doctoral dissertation, we mainly refer to the step happening before the model training step that happens after the hyperparameter tuning step, in which we train the model on the full training dataset using the hyperparameter values found in the previous step.

the activation functions, to ensure stable gradient propagation during the optimization process. A recent review of weight initialization techniques and approaches can be found in [58].

While these initialization techniques have proved effective compared to naive random initialization, they often ignore valuable prior information derived from related training tasks or steps. Most importantly, during hyperparameter tuning, we train several variants of a DNN on subsets of the training data with the goal of finding suitable hyperparameter values. However, these partially trained models are typically discarded after tuning, and only the hyperparameter values that trained the best model are used as the output of this step. Our work on weight initialization from hyperparameter tuning trials [5], proposes a new weight initialization method that reuses the weights learned during the hyperparameter tuning step to initialize the DNN weights for the actual training $step^5$, which can improve or maintain the best test accuracy of the trained model while reducing the overall time and compute resources required for training.

2.5 Distributed Model Training

Training DNNs is computationally intensive, often requiring vast amounts of data and compute resources. As model sizes and dataset complexities continue to grow, single-host training becomes impractical due to memory limitations and long training times [7], [11]. Distributed training addresses these challenges by leveraging multiple computing nodes, or workers, to distribute and/or parallelize the workload, reducing training time and enabling large-scale model training that would otherwise be infeasible.

There are two primary paradigms for distributed training of DNNs: data parallelism and model parallelism [8]. In Data-parallel Training (DPT), the model is replicated across multiple workers, each processing a different subset of the data. Gradients are computed independently and aggregated across workers before updating the shared model parameters. This approach is widely used due to its scalability, particularly when combined with efficient communication strategies such as asynchronous updates or gradient compression. In model-parallel training, different parts of the model are distributed across multiple workers, making it useful for extremely large architectures that cannot fit into a single machine's GPU memory.

Our work introduces contributions to improve the techniques used for distributed DNN training. In particular, distribution-oblivious training functions [1] enable seamless execution across single-host and distributed

 $^{^{5}}$ Without the loss of generality, we focus on hyperparameters that do not change the architecture of the DNN, so that all hyperparameter tuning trials train the exact same model architecture.

settings, abstracting away infrastructure-specific details. We also propose importance-aware DPT [4], which takes into account the individual importance of dataset examples rather than relying on conventional random partitioning, leading to faster convergence and possibly better model performance. These advancements reduce computational overhead, optimize resource allocation, and make distributed DNN training more accessible and efficient.

2.6 Ablation Studies

Generally speaking, an ablation study in the context of ML and DL refers to a systematic examination of an ML system by removing its different components one at a time, in order to understand the individual contributions of those components to the overall performance of the system [3], [20]–[22]. As an example, when doing ML research that revolves around developing novel DNN architectures or training algorithms, researchers can use ablation studies to gain deeper insights into what drives the success of their models or algorithms. For example, one can find out that they really did not need that specific convolutional layer in their DNN after all.

Despite their importance, conducting ablation studies can be time-consuming and computationally expensive, especially for large-scale models with numerous components. Traditional approaches often involve manually tweaking configurations and rerunning experiments, making the process errorprone and resource-intensive. In the absence of dedicated frameworks, this also means that the practitioners should also maintain multiple mostlysimilar copies of their artifacts, each corresponding to a different ablation trial. This has motivated the need for automated solutions that can handle the complexity and scale of modern ML systems.

Our work on AUTOABLATION [3], introduces an automated framework for conducting parallel ablation studies, enabling efficient exploration of multiple configurations simultaneously. Our final contribution, ABLATION-MAGE [6], leverages the capabilities of Large Language Models (LLMs) to facilitate the design and execution of ablation studies further. By combining automation with intelligent code generation, ABLATIONMAGE significantly reduces the overhead of manual experimentation. Together, these contributions demonstrate that ablation studies can be transformed from a labor-intensive process into an accessible, efficient, and integral part of the ML research and development cycles.

2.7 Machine Learning Tools and Frameworks

Modern DL training and experimentation workflows rely on frameworks that enable scalable training, efficient computation, and seamless integration with other distributed systems. Among these, Apache Spark, Tensor-Flow, and PyTorch stand out as key technologies that support large-scale data processing and DL model development.

Apache Spark [19], [59] is a distributed computing framework designed for large-scale data processing and analytics. It provides a high-level Application Programming Interface (API) for parallel execution across clusters, making it a popular choice for handling big data workloads efficiently. Unlike traditional MapReduce-based systems, Spark employs an in-memory computing model that significantly accelerates iterative tasks, such as those found in ML and DL workflows. Its integration with libraries like MLlib for scalable ML and support for distributed data processing makes it well-suited for tasks such as hyperparameter tuning and distributed deep learning. In our work, we leverage Spark's scalability to run parallelized jobs and tasks and extend it to provide asynchronous parallel execution of hyperparameter optimization and ablation study trials, enabling efficient resource utilization in large-scale DL pipelines.

TensorFlow [60], [61] is an open-source DL framework developed by Google, designed and optimized for high-performance numerical computation and large-scale model training. It provides flexible APIs for defining and training DNNs, with built-in support for hardware acceleration on Graphics Processing Units (GPUs). TensorFlow's dataflow-based execution model enables efficient distributed training, making it a leading tool both in academia and the industry for DL research and production deployment. We provide TensorFlow support in MAGGY and AUTOABLATION as the standard DL framework, and AUTOABLATION utilizes Keras on TensorFlow backend to automate the generation of ablation trials.

Besides TensorFlow, PyTorch [62] is another widely used DL framework, known for its dynamic computation graph and high adoption by researchers. PyTorch's Pythonic approach to DL model development makes it an ideal framework for fast prototyping and conducting research experiments. Developed by Facebook, PyTorch provides fast tensor operations, automatic differentiation, and GPU acceleration. Its support for distributed training through Torch Distributed (torch.distributed) allows efficient scaling across multiple devices, whether on a single machine or several workers in a cluster [63]. In our work, we leverage PyTorch's flexibility to implement and evaluate DNN training methods. In particular, we use it to implement importance-aware DPT [4] and weight initialization from hyperparameter tuning trials [5].

CHAPTER 2. BACKGROUND AND RELATED WORK

In the next chapter, we will provide a summary of the included papers in this doctoral dissertation, and relate the contributions of each paper to our main research objectives.

Chapter 3

Summary of Appended Papers

In this chapter, we provide the summaries of each of the included papers in this doctoral dissertation, and map their contributions to the main research objectives introduced in chapter 1.

Paper 1

Towards Distribution Transparency for Supervised ML With Oblivious Training Functions

Moritz Meister, Sina Sheikholeslami, Robin Andersson, Alexandru A. Ormenisan, and Jim Dowling

In Workshop on MLOps Systems, co-located with the Third Conference on Machine Learning and Systems (MLSys), 2020

Abstract: Building and productionizing Machine Learning (ML) models is a process of interdependent steps of iterative code updates, including exploratory model design, hyperparameter tuning, ablation experiments, and model training. Industrial-strength ML involves doing this at scale, using many compute resources, and this requires rewriting the training code to account for distribution. The result is that moving from a single host program to a cluster hinders iterative development of the software, as iterative development would require multiple versions of the software to be maintained and kept consistent. In this paper, we introduce the distribution oblivious training function as an abstraction for ML development in Python, whereby developers can reuse the same training function when running a notebook on a laptop or performing scale-out hyperparameter search and distributed training on clusters. Programs written in our framework look

CHAPTER 3. SUMMARY OF APPENDED PAPERS

like industry-standard ML programs as we factor out dependencies using best-practice programming idioms (such as functions to generate models and data batches). We believe that our approach takes a step towards unifying single-host and distributed ML development.

Contributions: This paper introduces distribution-oblivious training functions, a programming abstraction for ML development that allows for seamless execution of code across single-host setups or multiple workers. The work presented in this paper contributes to the main research objective O1: *Develop Tools and Frameworks for Scalable, Distributed DNN Training Experiments* as the programming model of two subsequent frameworks, MAGGY and AUTOABLATION.

Maggy: Scalable Asynchronous Parallel Hyperparameter Search

Moritz Meister, Sina Sheikholeslami, Amir H. Payberah, Vladimir Vlassov, and Jim Dowling

In the 1st Workshop on Distributed Machine Learning (DistributedML), co-located with the 16th International Conference on Emerging Networking Experiments and Technologies (CoNEXT), 2020

Abstract: Running extensive experiments is essential for building Machine Learning (ML) models. Such experiments usually require iterative execution of many trials with varying run times. In recent years, Apache Spark has become the de-facto standard for parallel data processing in the industry, in which iterative processes are implemented within the bulk-synchronous parallel (BSP) execution model. The BSP approach is also being used to parallelize ML trials in Spark. However, the BSP task synchronization barriers prevent asynchronous execution of trials, which leads to a reduced number of trials that can be run on a given computational budget. In this paper, we introduce MAGGY, an open-source framework based on Spark, to execute ML trials asynchronously in parallel, with the ability to early stop poorly performing trials. In the experiments, we compare MAGGY with the BSP execution of parallel trials in Spark and show that on random hyperparameter search on a convolutional neural network for the Fashion-MNIST dataset MAGGY reduces the required time to execute a fixed number of trials by 33% to 58%, without any loss in the final model accuracy.

Contributions: This paper introduces MAGGY, an open-source framework for asynchronous, parallel hyperparameter tuning that can run arbitrary Python code on top of Apache Spark using distribution-oblivious training functions. Through experimental evaluation, we show that MAGGY can significantly reduce the execution time of hyperparameter tuning trials. The work presented in this paper contributes to the main research objective O1: *Develop Tools and Frameworks for Scalable, Distributed DNN Training Experiments* as a tool and framework.

AutoAblation: Automated Parallel Ablation Studies for Deep Learning

Sina Sheikholeslami, Moritz Meister, Tianze Wang, Amir H. Payberah, Vladimir Vlassov, and Jim Dowling

In the 1st Workshop on Machine Learning and Systems (EuroMLSys), co-located with the 16th European Conference on Computer Systems (EuroSys), 2021

Abstract: Ablation studies provide insights into the relative contribution of different architectural and regularization components to machine learning models' performance. In this paper, we introduce AUTOABLATION, a new framework for the design and parallel execution of ablation experiments. AUTOABLATION provides a declarative approach to defining ablation experiments on model architectures and training datasets, and enables the parallel execution of ablation trials. This reduces the execution time and allows more comprehensive experiments by exploiting larger amounts of computational resources. We show that AUTOABLATION can provide nearlinear scalability by performing an ablation study on the modules of the Inception-v3 network trained on the TenGeoPSAR dataset.

Contributions: This paper introduces AUTOABLATION, the first dedicated framework for automated design and parallel execution of ablation study experiments in ML and DL, implemented on top of MAGGY. We also provide a formal definition of ablation studies in the context of ML and DL. The work presented in this paper contributes to the research objectives O1: Develop Tools and Frameworks for Scalable, Distributed DNN Training Experiments and O3: Improve the state of Ablation Studies in ML and DL as a tool and framework.

The Impact of Importance-aware Dataset Partitioning on Data-parallel Training of Deep Neural Networks

Sina Sheikholeslami, Amir H. Payberah, Tianze Wang, Jim Dowling, and Vladimir Vlassov

In the 23rd IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), 2023

Abstract: Deep neural networks used for computer vision tasks are typically trained on datasets consisting of thousands of images, called examples. Recent studies have shown that examples in a dataset are not of equal importance for model training and can be categorized based on quantifiable measures reflecting a notion of "hardness" or "importance". In this work, we conduct an empirical study of the impact of importance-aware partitioning of the dataset examples across workers on the performance of dataparallel training of deep neural networks. Our experiments with CIFAR-10 and CIFAR-100 image datasets show that data-parallel training with importance-aware partitioning can perform better than vanilla data-parallel training, which is oblivious to the importance of examples. More specifically, the proper choice of the importance measure, partitioning heuristic, and the number of intervals for dataset repartitioning can improve the best accuracy of the model trained for a fixed number of epochs. We conclude that the parameters related to importance-aware data-parallel training, including the importance measure, number of warmup training epochs, and others defined in the paper, may be considered as hyperparameters of dataparallel model training.

Contributions: This paper introduces Importance-aware DPT, a novel dataset partitioning approach for DPT of DNNs that considers dataset example importance instead of random partitioning. Through experimental evaluation, we show that compared to vanilla DPT, which uses random partitioning, importance-aware DPT can reduce model training time while maintaining model accuracy. The work presented in this paper contributes to the main research objective O2: *Enhance the Efficiency of Existing DNN Training Techniques* as a method.

Deep Neural Network Weight Initialization from Hyperparameter Tuning Trials

Sina Sheikholeslami, Tianze Wang, Amir H. Payberah, Jim Dowling, and Vladimir Vlassov

In the 31st International Conference on Neural Information Processing (ICONIP), 2024

Abstract: Training of deep neural networks from scratch requires initialization of the neural network weights as a first step. Over the years, many policies and techniques for weight initialization have been proposed and widely used, including Kaiming initialization and different variants of random initialization. On the other hand, another requirement for starting the training stage is to choose and set suitable hyperparameter values, which are usually obtained by performing several hyperparameter tuning trials. In this paper, we study the suitability of weight initialization using weights obtained from different epochs of hyperparameter tuning trials and compare it to Kaiming uniform (random) weight initialization for image classification tasks. Based on an experimental evaluation using ResNet-18, ResNet-152, and InceptionV3 models, and CIFAR-10, CIFAR-100, Tiny ImageNet, and Food-101 datasets, we show that weight initialization from hyperparameter tuning trials can speed up the training of deep neural networks by up to 2x while maintaining or improving the best test accuracy of the trained models, when compared to random initialization.

Contributions: This paper introduces A novel weight initialization approach for training DNNs that reuses computations from the hyperparameter tuning stage. Through experimental evaluation, we show that compared to random weight initialization, this approach can reduce the training time while maintaining or improving model accuracy. The work presented in this paper contributes to the main research objective O2: *Enhance the Efficiency of Existing DNN Training Techniques* as a method.

Utilizing Large Language Models for Ablation Studies in Machine Learning and Deep Learning

Sina Sheikholeslami, Hamid Ghasemirahni, Amir H. Payberah, Tianze Wang, Jim Dowling, and Vladimir Vlassov

In the 5th Workshop on Machine Learning and Systems (EuroMLSys), co-located with the 20th European Conference on Computer Systems (EuroSys), 2025

Abstract: In Machine Learning (ML) and Deep Learning (DL) research, ablation studies are typically performed to provide insights into the individual contribution of different building blocks and components of an ML/DL system (e.g., a deep neural network), as well as to justify that certain additions or modifications to an existing ML/DL system can result in the proposed improved performance. Although dedicated frameworks for performing ablation studies have been introduced in recent years, conducting such experiments is still associated with requiring tedious, redundant work, typically involving maintaining redundant and mostly-identical versions of code that correspond to different ablation trials. Inspired by the recent promising performance of Large Language Models (LLMs) in the generation and analysis of ML/DL code, in this paper we discuss the potential of LLMs as facilitators of ablation study experiments for scientific research projects that involve or deal with ML and DL models. We first discuss the different ways in which LLMs can be utilized for ablation studies and then present the prototype of a tool called ABLATIONMAGE, that leverages LLMs to semi-automate the overall process of conducting ablation study experiments. We showcase the usability of ABLATIONMAGE as a tool through three experiments, including one in which we reproduce the ablation studies from a recently published applied DL paper.

Contributions: This paper introduces ABLATIONMAGE, a tool based on LLMs that further simplifies the conduction of ablation studies. The work presented in this paper contributes to the main research objective O3: *Improve the state of Ablation Studies in ML and DL* as a tool.

Chapter 4

Conclusions and Future Work

We now conclude the first part of this doctoral dissertation by providing a brief summary of our contributions and discussing a number of possible avenues for further research and study.

First, we describe how we have pursued the main research objectives and addressed the challenges introduced in section 1.1, and our contributions. We highlight how each contribution enhances the existing landscape of scalable Machine Learning (ML)/Deep Learning (DL) training and automated experimentation, providing practical improvements while opening new research directions in these areas. Finally, we discuss a number of possible avenues for further research and study.

The second part of the dissertation, which immediately follows this chapter, contains verbatim copies of the peer-reviewed manuscripts I co-authored that contribute to this doctoral dissertation.

4.1 Dissertation Summary

This doctoral dissertation introduces tools and methods that improve the efficiency, scalability, and usability of distributed ML and DL workflows, with those related to training of Deep Neural Networks (DNNs) as a particular focus. In section 1.1, we identified a number of challenges in large-scale and distributed training of DNNs, and defined three research objectives to be pursued in order to address the said challenges.

For Objective O1 (Develop Tools and Frameworks for Scalable, Distributed DNN Training Experiments), we introduce tools and methods that simplify and improve distributed deep learning workflows. A major challenge in this area is the complexity of transitioning from single-host to distributed training, as well as mitigating straggler effects in distributed environments. To pursue this objective and address these challenges, we developed distribution-oblivious training functions (C1) [1] to unify single-host and distributed ML/DL training functions. We further extended this work by developing **MAGGY** (C2) [2], a framework for scalable, asynchronous parallel hyperparameter tuning, and AUTOABLATION (C3) [3], a framework enabling automated parallel ablation study experiments. These tools provide abstractions and automation that enhance usability while maintaining efficiency in distributed training environments. The proposed distribution-oblivious training functions reduce the programming complexity typically associated with distributed ML/DL workflows, lowering the entry barrier for practitioners and researchers. MAGGY advances hyperparameter tuning frameworks by incorporating distributionoblivious training fuctions with asynchronous execution and early stopping, making large-scale hyperparameter tuning significantly more resourceefficient. Additionally, AUTOABLATION introduces a novel automated approach to ablation studies in ML/DL, enabling systematic model evaluation while leveraging parallel execution for increased efficiency and distributionoblivious training functions for usability.

For Objective O2 (Enhance the Efficiency of Existing DNN **Training Techniques**), we focused on improving the computational efficiency of existing widely-used distributed training techniques and approaches by addressing inefficiencies in dataset partitioning step in Data-parallel Training (DPT), and exploiting computation reuse for the weight initialization step that is a pre-requisite for model training. The standard DPT approach does not consider the varying importance of dataset examples, ignoring the opportunity for exploiting example importance to improve the efficiency of model training. To improve this, we introduced importanceaware DPT (C4) [4], which partitions datasets based on example importance rather than random assignment, reducing network and Input/Output (I/O) overhead while maintaining or improving model accuracy. Furthermore, we explored computation reuse across different steps in the training pipeline. We proposed a novel weight initialization method (C5) [5] that reuses model weights from hyperparameter tuning trials, leading to faster convergence and potential improvements in final model accuracy. The proposed importance-aware DPT method advances existing data-parallel training techniques by incorporating importance-based dataset partitioning, an approach that significantly reduces resource overhead while preserving model performance. This challenges the long-standing assumption of random data partitioning in distributed training and provides a new perspective on optimizing DPT. Meanwhile, the novel weight initialization strategy from hyperparameter tuning trials introduces a practical form of computation reuse in DL workflows, reducing the computational cost of model training while maintaining or even improving convergence speed and final

model accuracy.

Finally, for Objective O3 (Improve the State of Ablation Studies in ML and DL), we tackled the high computational and manual effort required for conducting ablation studies, which are crucial for understanding the contributing factors to the performance of a DNNs. To automate and parallelize ablation experiments, we introduced **AUTOABLATION** (C3) [3], the first dedicated framework for automated ablation studies in ML/DL. Later, leveraging advancements in Large Language Models (LLMs), we developed **ABLATIONMAGE** (C6) [6], a tool that utilizes LLMs to automate the design and execution of ablation studies, significantly reducing human effort and redundancy in conducting these experiments. AUTOABLATION represents a step forward in systematic model evaluation, making it easier and more efficient to perform large-scale ablation studies in ML/DL while minimizing the need for manual intervention. ABLATIONMAGE further pushes the boundaries of automation in ML/DL research by leveraging LLMs to generate and modify ablation study configurations, showcasing how generative artificial intelligence can facilitate the scientific research process. These contributions not only reduce the effort required to conduct ablation studies but also establish a foundation for more automated and scalable model evaluation frameworks.

To summarize, the key contributions presented in this dissertation are:

- C1. Distribution-oblivious Training Functions: A novel programming abstraction that enables a seamless transition between single-host and distributed ML/DL training without requiring significant code modifications or familiarity with distributed computing frameworks [1].
- C2. **MAGGY**: An open-source framework for scalable, asynchronous, and parallel hyperparameter tuning that improves resource utilization and reduces training time, while leveraging distribution-oblivious training functions for increased usability [2].
- C3. AUTOABLATION: The first dedicated framework for automated, parallel ablation study experiments in ML/DL, enabling a systematic approach to the evaluation of different components (e.g., dataset features or model layers) in an ML system [3].
- C4. **Importance-aware DPT**: A dataset partitioning strategy that considers data example importance for partitioning the dataset across data-parallel workers, leading to improved training efficiency and model performance [4].
- C5. Weight Initialization from Hyperparameter Tuning Trials: A novel weight initialization strategy that reuses model weights learned

during hyperparameter tuning, accelerating model convergence while maintaining or improving model accuracy [5].

C6. **ABLATIONMAGE**: A tool leveraging LLMs to automate and streamline the execution of ablation studies in ML/DL [6].

Furthermore, distribution-oblivious training functions (C1), MAGGY (C2) and AUTOABLATION (C3) were integrated into Hopsworks [13], a platform for ML/DL, as presented in [14].

While the experimental evaluation of our introduced tools and methods mainly involves benchmark image classification tasks, we should mention that all of our contributions are domain-independent, and can be extended to or used in any ML/DL training experiment. Ablation studies are part of the scientific research process in ML/DL and thus are domain-independent by definition. In particular, MAGGY can essentially work with any Python function that returns a value (or metric); validation or test accuracies are such metrics that are used in hyperparameter tuning and ablation study trials. Regarding importance-aware DPT, the dataset partitioning heuristics can be applied to any DNN training task that can benefit from dataparallelism, and our released proof-of-concept allows for effortless implementation of different heuristics and importance metrics and measures. The same argument can be made for our weight initialization approach and its released proof-of-concept.

4.2 Broader Impact

The rapid adoption of Artificial Intelligence (AI), and in particular ML and DL applications, has led to a massive increase in computational demands, emphasizing the need for scalable and efficient training methodologies. This doctoral dissertation contributes to computational sustainability by reducing the resource requirements of ML/DL workflows, particularly in large-scale parallel and distributed training settings. By improving hyperparameter tuning, dataset partitioning, and weight initialization steps in the model training pipeline, this work reduces redundant computations, improves parallelization efficiency, and minimizes energy consumption, thus aligning with sustainable AI practices.

Additionally, tools such as MAGGY, AUTOABLATION, and ABLATION-MAGE democratize access to usable and scalable ML experimentation by abstracting away the complexities of distributed computing or eliminating the tedious work required for preparing redundant experimentation artifacts. However, we should also mention that these contributions might also pose potential risks. In particular, lowering the entry barrier for practitioners and reducing the costs associated with the training of DNNs, can lead to an overall higher global energy consumption or other possible negative societal impacts of AI [29].

4.3 Future Work

In general, for future work, we would like to increase the scale and dimensions of our experimental validation setups. Although all the contributions of this doctoral dissertation have been evaluated through experiments on benchmark tasks and datasets in ML/DL, there is room for further improvement. In particular, most of our downstream tasks revolve around image classification (a very popular task in DL research). Hence, we would be interested in evaluating our tools and methods on training DNNs for other downstream tasks (e.g., language modeling) on more families of DNNs, and other data modalities (e.g., text and tabular data). There is also a possibility of studying the proposed contributions from a theoretical perspective.

We would also like to scale up the size and complexity of the datasets used for the experiments, especially since some of the benefits of our contributions (e.g., the reduction of network and I/O overhead in importanceaware data-parallel training) will become more clear in larger scales. For importance-aware DPT, we are also interested in investigating the use of other importance measures and metrics (e.g., example forgetting [64]) for dataset partitioning.

Finally, for our weight initialization method, we would like to study what leads to some dataset/model combinations working better with this approach than other combinations.

Bibliography

- M. Meister, S. Sheikholeslami, R. Andersson, A. A. Ormenisan, and J. Dowling, "Towards distribution transparency for supervised ml with oblivious training functions," in *Workshop on MLOps Systems*, 2020.
- M. Meister et al., "Maggy: Scalable asynchronous parallel hyperparameter search," in Workshop on Distributed Machine Learning, 2020, pp. 28–33.
- [3] S. Sheikholeslami, M. Meister, T. Wang, A. H. Payberah, V. Vlassov, and J. Dowling, "Autoablation: Automated parallel ablation studies for deep learning," in *Proceedings of the 1st Workshop on Machine Learning and Systems*, 2021, pp. 55–61.
- [4] S. Sheikholeslami, A. H. Payberah, T. Wang, J. Dowling, and V. Vlassov, "The impact of importance-aware dataset partitioning on data-parallel training of deep neural networks," in *IFIP International Conference on Distributed Applications and Interoperable Systems*, Springer, 2023, pp. 74–89.
- [5] S. Sheikholeslami, T. Wang, A. H. Payberah, J. Dowling, and V. Vlassov, "Deep neural network weight initialization from hyperparameter tuning trials," in *International Conference on Neural Information Processing (ICONIP)*, Springer, 2025, pp. 74–89.
- [6] S. Sheikholeslami, H. Ghasemirahni, A. H. Payberah, T. Wang, J. Dowling, and V. Vlassov, "Utilizing large language models for ablation studies in machine learning and deep learning," in *Proceedings of the 5th Workshop on Machine Learning and Systems*, 2025.
- G. Wang, Distributed Machine Learning with Python: Accelerating model training and serving with distributed systems. Packt Publishing Ltd, 2022.
- [8] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li, "Communicationefficient distributed deep learning: A comprehensive survey," arXiv preprint arXiv:2003.06307, 2020.

BIBLIOGRAPHY

- [9] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," ACM Computing Surveys (CSUR), vol. 52, no. 4, pp. 1–43, 2019.
- [10] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *Acm computing surveys (csur)*, vol. 53, no. 2, pp. 1–33, 2020.
- [11] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools," ACM Computing Surveys (CSUR), vol. 53, no. 1, pp. 1–37, 2020.
- [12] X. Liu, D. Gu, Z. Chen, et al., "Rise of distributed deep learning training in the big model era: From a software engineering perspective," ACM Transactions on Software Engineering and Methodology, vol. 32, no. 6, pp. 1–26, 2023.
- [13] M. Ismail et al., "Hopsworks: Improving user experience and development on hadoop with scalable, strongly consistent metadata," in 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2017, pp. 2525–2528.
- [14] D. H. Hagos, T. Kakantousis, S. Sheikholeslami, et al., "Scalable artificial intelligence for earth observation data using hopsworks," *Remote Sensing*, vol. 14, no. 8, p. 1889, 2022.
- [15] D. H. Hagos, T. Kakantousis, V. Vlassov, et al., "The extremeearth software architecture for copernicus earth observation data," in Proceedings of the 2021 Conference on Big Data from Space (BiDS'21), 2021, pp. 181–185.
- [16] M. Koubarakis, G. Stamoulis, D. Bilidas, et al., "Artificial intelligence and big data technologies for copernicus data: The extremeearth project," in *Proceedings of the 2021 Conference on Big Data from* Space (BiDS'21), 2021, pp. 9–12.
- [17] L. Li, K. Jamieson, A. Rostamizadeh, et al., "A system for massively parallel hyperparameter tuning," *Proceedings of Machine Learning* and Systems, vol. 2, pp. 230–246, 2020.
- [18] L. G. Valiant, "A bridging model for parallel computation," Communications of the ACM, vol. 33, no. 8, pp. 103–111, 1990.
- [19] M. Zaharia et al., "Spark: Cluster computing with working sets," *Hot-Cloud*, vol. 10, no. 10-10, p. 95, 2010.
- [20] I. Fostiropoulos and L. Itti, "Ablator: Robust horizontal-scaling of machine learning ablation experiments," in *International Conference* on Automated Machine Learning, PMLR, 2023, pp. 19–1.

- [21] R. Meyes, M. Lu, C. W. de Puiseau, and T. Meisen, "Ablation studies in artificial neural networks," arXiv preprint arXiv:1901.08644, 2019.
- [22] I. Hameed, S. Sharpe, D. Barcklow, et al., "Based-xai: Breaking ablation studies down for explainable artificial intelligence," arXiv preprint arXiv:2207.05566, 2022.
- [23] A. Hevner, S. Chatterjee, A. Hevner, and S. Chatterjee, "Design science research in information systems," *Design research in information* systems: theory and practice, pp. 9–22, 2010.
- [24] F. Chollet *et al.*, *Keras*, https://keras.io, 2015.
- [25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer* vision and pattern recognition, 2016, pp. 770–778.
- [26] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of* the *IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.
- [27] C. Wang et al., "A labelled ocean sar imagery dataset of ten geophysical phenomena from sentinel-1 wave mode," *Geoscience Data Journal*, vol. 6, no. 2, pp. 105–115, 2019.
- [28] C. E. Tripp, J. Perr-Sauer, J. Gafur, et al., "Measuring the energy consumption and efficiency of deep neural networks: An empirical analysis and design recommendations," arXiv preprint arXiv:2403.08151, 2024.
- [29] R. Vinuesa, H. Azizpour, I. Leite, et al., "The role of artificial intelligence in achieving the sustainable development goals," *Nature com*munications, vol. 11, no. 1, p. 233, 2020.
- [30] C. Von Utfall Danielsson, "The rebound effect: Theory, evidence and implications for energy policy," 2010.
- [31] A. Druckman, M. Chitnis, S. Sorrell, and T. Jackson, "Missing carbon reductions? exploring rebound and backfire effects in uk households," *Energy policy*, vol. 39, no. 6, pp. 3572–3581, 2011.
- [32] M. Angelovska, S. Sheikholeslami, B. Dunn, and A. H. Payberah, "Siamese neural networks for detecting complementary products," in Proceedings of the 16th conference of the European chapter of the association for computational linguistics: student research workshop, 2021, pp. 65–70.
- [33] D. H. Hagos, T. Kakantousis, V. Vlassov, et al., "Extremeearth meets satellite data from space," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 14, pp. 9038–9063, 2021.

BIBLIOGRAPHY

- [34] A. Asratyan, S. Sheikholeslami, and V. Vlassov, "A parallel chain mail approach for scalable spatial data interpolation," in 2021 IEEE International Conference on Big Data (Big Data), 2021, pp. 306–314.
- [35] G. Chikafa, S. Sheikholeslami, S. Niazi, J. Dowling, and V. Vlassov, "Cloud-native rstudio on kubernetes for hopsworks," Advances in Parallel & Distributed Processing, and Applications, 2022.
- [36] T. Johannesson, I. Rubensson, S. Sheikholeslami, A. Al-Shishtawy, and V. Vlassov, "Duget: Leveraging machine learning for dynamic user grouping and evolution tracking in public transit systems," in 2024 IEEE International Conference on Big Data (BigData), IEEE, 2024, pp. 1785–1794.
- [37] Z. Xu, P. Nordström, S. Sheikholeslami, A. Al-Shishtawy, and V. Vlassov, "A semi-supervised model for non-cellular elements segmentation in microscopy images of wood," in 2024 IEEE International Conference on Big Data (BigData), IEEE, 2024, pp. 2049–2056.
- [38] What are ml systems? [Online]. Available: https://www.hopsworks. ai/dictionary/ml-systems.
- [39] C. Huyen, Designing machine learning systems. O'Reilly Media, Inc., 2022.
- [40] What is a machine learning pipeline? [Online]. Available: https://www.hopsworks.ai/dictionary/ml-pipeline.
- [41] [Online]. Available: https://scikit-learn.org/1.5/modules/ generated/sklearn.pipeline.Pipeline.html.
- [42] H. Hapke and C. Nelson, Building machine learning pipelines. O'Reilly Media, 2020.
- [43] J. Dowling, Building Machine Learning Systems with a Feature Store. O'Reilly Media, 2025.
- [44] J. de la Rúa Martínez, F. Buso, A. Kouzoupis, et al., "The hopsworks feature store for machine learning," in Companion of the 2024 International Conference on Management of Data, 2024, pp. 135–147.
- [45] C. Molnar, Interpretable machine learning. Lulu.com, 2019.
- [46] Z. Tang, S. Shi, W. Wang, B. Li, and X. Chu, "Communicationefficient distributed deep learning: A comprehensive survey," arXiv preprint arXiv:2003.06307, 2020.
- [47] T. Yu and H. Zhu, "Hyper-parameter optimization: A review of algorithms and applications," arXiv preprint arXiv:2003.05689, 2020.
- [48] M. Feurer and F. Hutter, "Hyperparameter optimization," Automated machine learning: Methods, systems, challenges, pp. 3–33, 2019.

- [49] F. Hutter, L. Kotthoff, and J. Vanschoren, Automated Machine Learning: Methods, Systems, Challenges. Springer Nature, 2019.
- [50] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization.," *Journal of machine learning research*, vol. 13, no. 2, 2012.
- [51] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.
- [52] X. Meng, J. Bradley, B. Yavuz, et al., "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.
- [53] W. A. Falcon et al., "Pytorch Lightning," 2019.
- [54] N. Weymaere and J.-P. Martens, "On the initialization and optimization of multilayer perceptrons," *IEEE Transactions on Neural Net*works, vol. 5, no. 5, pp. 738–751, 1994.
- [55] D. Arpit, V. Campos, and Y. Bengio, "How to initialize your network? robust initialization for weightnorm & resnets," Advances in Neural Information Processing Systems, vol. 32, 2019.
- [56] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.
- [57] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.
- [58] M. V. Narkhede, P. P. Bartakke, and M. S. Sutaone, "A review on weight initialization strategies for neural networks," *Artificial intelli*gence review, vol. 55, no. 1, pp. 291–322, 2022.
- [59] M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th* USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2012, pp. 2–2.
- [60] M. Abadi et al., "Tensorflow: A system for large-scale machine learning," in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 265–283.
- [61] Martín Abadi, Ashish Agarwal, Paul Barham, et al., TensorFlow: Large-scale machine learning on heterogeneous systems, Software available from tensorflow.org, 2015. [Online]. Available: https://www. tensorflow.org/.

BIBLIOGRAPHY

- [62] A. Paszke et al., "Pytorch: An imperative style, high-performance deep learning library," Advances in Neural Information Processing Systems, vol. 32, pp. 8026–8037, 2019.
- [63] S. Li, Y. Zhao, R. Varma, et al., "Pytorch distributed: Experiences on accelerating data parallel training," Proceedings of the VLDB Endowment, vol. 13, no. 12, 2020.
- [64] M. Toneva, A. Sordoni, R. T. d. Combes, A. Trischler, Y. Bengio, and G. J. Gordon, "An empirical study of example forgetting during deep neural network learning," in *ICLR*, 2019.

Part II Appended Papers

Towards Distribution Transparency for Supervised ML With Oblivious Training Functions

Workshop on MLOPS Systems, co-located with the Third Conference on Machine Learning and Systems (MLSys), 2020

Towards Distribution Transparency for Supervised ML With Oblivious Training Functions

Moritz Meister¹, Sina Sheikholeslami², Robin Andersson¹, Alexandru A. Ormenisan^{1,2}, and Jim Dowling^{1,2}

 1 Logical Clocks AB, Stockholm, Sweden 2 KTH Royal Institute of Technology, Stockholm, Sweden

Abstract

Building and productionizing Machine Learning (ML) models is a process of interdependent steps of iterative code updates, including exploratory model design, hyperparameter tuning, ablation experiments, and model training. Industrial-strength ML involves doing this at scale, using many compute resources, and this requires rewriting the training code to account for distribution. The result is that moving from a single host program to a cluster hinders iterative development of the software, as iterative development would require multiple versions of the software to be maintained and kept consistent. In this paper, we introduce the distribution oblivious training function as an abstraction for ML development in Python, whereby developers can reuse the same training function when running a notebook on a laptop or performing scale-out hyperparameter search and distributed training on clusters. Programs written in our framework look like industry-standard ML programs as we factor out dependencies using best-practice programming idioms (such as functions to generate models and data batches). We believe that our approach takes a step towards unifying single-host and distributed ML development.

1 Introduction

Machine learning (ML) is a complex subject, and the process of learning to program (train) ML applications usually involves starting with the simplest possible program, avoiding complexities such as feature engineering and scalability (distributed programming), and slowly adding complexity over time. In particular, moving from single-host applications to distributed applications is challenging, especially for supervised ML as it requires rewriting entire applications. This keeps many developers, who are used to single host debugging and testing and have limited knowledge about distributed environments, from discovering the benefits of distributed ML: faster hyperparameter sweeps and reduced training times.

The contribution of this paper is the design and implementation a framework that unifies single-host and distributed training functions based on an abstraction we call the *distribution oblivious training function*. We make training functions reusable by following the dependency inversion principle [1] to factor out those aspects of training functions that are subject to change between single-host and distributed applications. We demonstrate our framework for Keras/TensorFlow (TF) programs, but the approach generalizes to other frameworks that support distribution, such as PyTorch.

2 Distribution Transparency in ML



Figure 1: Inner and outer loops for the training function

Transparency in distributed systems [2] refers to hiding distributionspecific aspects of an application from the developer - for example, a developer invoking a function may not know (or need to know) if the function she is calling is local to her application or on a remote server. Distribution transparency enables developers to write code that is reusable between single-host and distributed instantiations of a program.

In supervised ML, the core logic that is common across all programs is the training function - a series of steps including defining a model architecture, then ingesting labelled training data and feeding it to the model and iterating until some termination (or convergence) criteria are met. The output of the training function is a model that can be used to make predictions on new data, drawn from the same distribution as the training data.

Training functions, however, can be used in many different contexts when we distribute supervised ML programs: single host notebooks, distributed hyperparamter search, parallel ablation studies, and distributed training
Process Step	Distributed aspects of the Training Function	
HParam Search	Model Architecture, Hyperparameters, Early Stopping Optimizer Regularization	
Ablation Study	Model Architecture, Features (Input Data), Reg-	
Distributed Training	Features (Input Data), Optimizer, Accelerator- awareness	

Table 1: Distributed aspects of the training function that need to be re-written for different distributed contexts

are common examples. However, existing frameworks for supervised ML, such as Keras/TensorFlow and PyTorch, require training functions to be rewritten to account for the distribution strategy, what accelerators the computations are scheduled on, and whether the optimizer needs to share its results with other hosts (for distributed training). In figure 1, we illustrate how training functions can be used - as part of (1) the inner loop when the same training function is either run on a single host or on many hosts in parallel (as part of data-parallel distributed training) using (distributed) stochastic gradient descent, or (2) the outer loop when the training function is run on different hosts for example with different hyperparameters for each host, and a global optimizer that collects the results of hyperparameters.

Table 1 lists the aspects of training functions in Keras/TensorFlow that need to be rewritten for a distribution context, compared to the singlehost Python program. For the inner loop, the hyperparameters and model architecture can vary, and code needs to be rewritten to account for how many hardware accelerators are being used. For the outer loop, the variable aspects related to model configuration are controlled by a global optimizer, such as a Bayesian optimizer proposing different configurations (trials), or an ablator, generating trials by leaving one or more components out at a time. The trials can be run in parallel on a cluster and the results collected by a global optimizer or ablator.

3 Distribution Oblivious Training Function

In figure 2, we can see how developers structure their applications to write *distribution oblivious training functions* and include them in the different distribution contexts. Firstly, developers write the common training function, and as is now considered good ML engineering practice, developers



Figure 2: ML model development process

also write separate functions for model architecture generation and data batch reading. The training function becomes a parameterizable higher order function with generator functions and hyperparameter configurations as input. For hyperparameter search, a search space needs to be defined from which a global optimizer (can be user-defined) draws the hyperparameters from, but for other cases, the hyperparameters are fixed. For example, the final distributed training step should make use of the best configuration found in the previous search experiments. The distribution context and environment can be initialized outside the training function (it is oblivious to it) to make appropriate use of the resources such as accelerators. Other means to achieve transparency of the two loops include the use of pluggable hooks, such as the Keras/TF callbacks.

4 Unified Execution Framework with Jupyter Notebooks on Hopsworks

With Hopsworks [3] and the Maggy framework [4][5], we provide a unified development and execution framework for distribution transparent Jupyter notebooks [6]. That is, the developer writes a Jupyter notebook that can be run/debugged using a single host Python kernel, and the same notebook can also be run on a cluster using many hosts and hardware accelerators as a PySpark application. The developer only needs to set a distribution context parameter that controls which cells to run in the notebook - the oblivious training function is a single cell used by all the different distribution contexts. The notebooks can also be parameterized and run by an external workflow manager (Airflow) in production ML pipelines, similar to Papermill by Neflix [7].

5 Related Work

Previous work on this topic can be categorized in three dimensions: Pipeline orchestration, ML lifecycle management and automated ML (AutoML). Pipeline orchestration covers the aspect of taking an entire ML pipeline into production, which includes data preparation and engineering, modeling, training, serving inference and managing the deployments. TensorFlow Extended (TFX) [8] is a TF based platform with the goal of minimizing glue code between these pipeline steps. Compared to the previous category, ML lifecycle management is concerned with the iterative nature of the ML development process. By tracking artifacts, logs and experiments, results can be easily reproduced, making the process more transparent with respect to the trained models themselves. MLFlow [9] achieves this by allowing the user to make explicit calls to log this meta-data. AutoML aims to automate every aspect of the pipeline. However, due to the high computational requirements, recent work was focusing on the automation of the separate steps first. Because many parts of a ML model behave like a black-box and can be encoded in hyperparameters, one can fall back on search for optimization of such parameters [10].

6 Summary

In this short paper, we introduced the distribution oblivious training function for supervised ML and showed how it can be used to write distribution transparent ML programs. In the Hopsworks platform, this provides developers with a unified framework and codebase where Jupyter notebooks can first be developed as single-host Python programs, then extended to distributed contexts, and iterative development across single-host and distributed versions is not just possible, but encouraged. The distribution oblivious training function can have several benefits for ML systems. It can (1) enable reductions in technical debt in pipeline orchestration, (2) enable iterative development between laptops and clusters, and (3) improve model training lifecycle management by factoring out explicit logging calls from user code.

Acknowledgement

This work is supported by the ExtremeEarth¹ project funded by European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 825258.

References

- R. C. Martin, Agile software development: principles, patterns, and practices. Prentice Hall, 2002.
- [2] A. Tanenbaum et al., Distributed systems: principles and paradigms. Prentice-Hall, 2007.
- [3] M. Ismail, E. Gebremeskel, T. Kakantousis, G. Berthou, and J. Dowling, "Hopsworks: Improving user experience and development on hadoop with scalable, strongly consistent metadata," in 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2017, pp. 2525–2528.
- [4] L. C. AB. "Maggy: Asynchronous distributed hyperparameter optimization based on apache spark," Logical Clocks AB. (Jan. 2020), [Online]. Available: https://github.com/logicalclocks/maggy.
- [5] M. J. Meister, "Maggy: Open-source asynchronous distributed hyperparameter optimization based on apache spark," M.S. thesis, 2019.
- [6] T. Kluyver, B. Ragan-Kelley, F. Pérez, et al., "Jupyter notebooksa publishing format for reproducible computational workflows.," in *ELPUB*, 2016, pp. 87–90.
- [7] A. Y. Wang, A. Mittal, C. Brooks, and S. Oney, "How data scientists use computational notebooks for real-time collaboration," *Proceed*ings of the ACM on Human-Computer Interaction, vol. 3, no. CSCW, pp. 1–30, 2019.
- [8] D. Baylor, E. Breck, H.-T. Cheng, et al., "Tfx: A tensorflow-based production-scale machine learning platform," in Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2017, pp. 1387–1395.
- [9] M. Zaharia, A. Chen, A. Davidson, et al., "Accelerating the machine learning lifecycle with mlflow.," *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 39–45, 2018.
- [10] F. Hutter, L. Kotthoff, and J. Vanschoren, Automated Machine Learning: Methods, Systems, Challenges. Springer Nature, 2019.

¹Project website: http://earthanalytics.eu.

Paper 2

Maggy: Scalable Asynchronous Parallel Hyperparameter Search

The 1st Workshop on Distributed Machine Learning (DistributedML), co-located with the 16th International Conference on Emerging Networking Experiments and Technologies (CoNEXT), 2020

MAGGY: SCALABLE ASYNCHRONOUS PARALLEL HYPERPARAMETER SEARCH

Moritz Meister¹, Sina Sheikholeslami², Amir H. Payberah², Vladimir Vlassov², and Jim Dowling^{1,2}

 1 Logical Clocks AB, Stockholm, Sweden 2 KTH Royal Institute of Technology, Stockholm, Sweden

Abstract

Running extensive experiments is essential for building Machine Learning (ML) models. Such experiments usually require iterative execution of many trials with varying run times. In recent vears, Apache Spark has become the de-facto standard for parallel data processing in the industry, in which iterative processes are implemented within the bulk-synchronous parallel (BSP) execution model. The BSP approach is also being used to parallelize ML trials in Spark. However, the BSP task synchronization barriers prevent asynchronous execution of trials, which leads to a reduced number of trials that can be run on a given computational budget. In this paper, we introduce MAGGY, an open-source framework based on Spark, to execute ML trials asynchronously in parallel, with the ability to early stop poorly performing trials. In the experiments, we compare MAGGY with the BSP execution of parallel trials in Spark and show that on random hyperparameter search on a convolutional neural network for the Fashion-MNIST dataset MAGGY reduces the required time to execute a fixed number of trials by 33% to 58%, without any loss in the final model accuracy.

1 Introduction

Traditionally, building Machine Learning (ML) models used to be an expensive and time-consuming process. However recently, Automated ML (AutoML) approaches have enabled data scientists to automate many aspects of this process at the cost of increased computational resources. Nevertheless, many parts of building ML models behave as black-boxes without gradient information of the loss available, thus AutoML has to fall back on less efficient search algorithms to optimize them. These search algorithms are executed in *experiments*, where a model is trained with different *con-figurations* (such as different learning rates or convolution filter sizes) to produce a performance metric (such as any loss or accuracy metric), which are then used by the search algorithm to propose new, potentially better configurations. Training such a model configuration is referred to as a *trial*.

In Deep Learning (DL), models are ever-growing in architecture size and complexity to beat the previous state-of-the-art. However, training large models with massive amounts of data not only increases the training time, but also causes a state explosion in the search space, as the performance of these models becomes more sensitive to a growing number of *hyperparameters*. Hyperparameters are parameters of an ML model (such as learning rate or choices about the model's architecture, regularization, and optimization) that cannot be optimized by the learning algorithm itself.

These characteristics render search extremely costly, as exploding search spaces require the evaluation of exponentially more trials. Moreover, to make DL models more robust and explainable, a new best practice, called *ablation studies* [1], [2], has evolved that is in nature similar to hyperparameter search experiments. Ablation studies require many trials to evaluate the relative contribution of different architectural and regularization components to models' performance. Therefore, they also suffer from the same curse of dimensionality with increasing model size.

Current state-of-the-art solutions for hyperparameter optimization (HPO) mainly schedule trials and update the search model asynchronously [3], [4]. Given that Apache Spark [5], [6] has become a popular data-parallel processing framework, the industry is increasingly building tools to accommodate the advanced algorithms for HPO on Spark [7], [8]. Spark implements iterative processes, such as HPO, within the *bulk-synchronous parallel (BSP)* execution model. However, the task synchronization barriers in BSP prevent asynchronous execution of trials, which leads to a reduced number of trials that can be run on a given computational budget. On the other hand, actor-based systems with their inherent asynchrony, like Ray [9], have shown to be a good fit for parallelized ML experiments. Nevertheless, history has shown that general-purpose programming frameworks (such as Spark), when equipped with specialized functionalities, tend to dominate specialized frameworks, in the long run.

In this paper, we introduce MAGGY, a framework for parallel ML experiments that extends Spark with support for asynchronous trials, early stopping, and global trial optimization. MAGGY introduces both a programming model for these experiments and a new driver-to-executor communication protocol that allows for globally managed asynchronous trials within the bulk-synchronous execution model (Figure 1). This protocol uses driver-worker heartbeats to add early-stopping and asynchronous schedul-

PAPER 2. MAGGY



Figure 1: MAGGY enables driver-to-executor communication that allows for globally managed asynchronous trials within the bulk-synchronous model.

ing functionality within Spark tasks, as shown in Figure 1. MAGGY tackles the following challenges in modern ML model development:

- 1. programming support for defining, optimizing, and running parallel ML experiments;
- 2. efficient use of parallel compute resources through asynchronous trials;
- 3. support for *global* directed search in high-dimensional hyperparameter search spaces. By global optimization, we mean that the optimizer has complete and up-to-date knowledge of all trials' learning curves and can make decisions on early stopping of poorly performing trials.

The experimental evaluation of MAGGY shows that it can reduce the run time of experiments with a fixed number of trials, requiring between 33% and 57% of the time that of a BSP Spark implementation. This reduction in time is achieved despite the added overhead of asynchronous communication, scheduling and performance sampling. The strength of MAGGY becomes apparent with early stopping, which introduces additional variation in trial run times and therefore more asynchrony.

2 Preliminary and Related Work

Although Apache Spark [5], [6] was initially developed for data-parallel processing, nowadays it provides a unified analytics engine, including ML applications. With its high-level libraries for SQL queries on semi-structured data, streaming data, ML, and graph processing, it became a generalpurpose framework. The fundamental data structure in Spark is resilient

PAPER 2. MAGGY



Figure 2: The bulk synchronous execution of iterations in Apache Spark only allows new trials to be executed as tasks at the beginning of a stage. This synchronization barrier results in wasted computation (when a trial is stopped early, or due to straggling trials) and delayed updates to the search model.

distributed dataset (RDD), which is a distributed collection of items [6]. The RDD provides the core abstraction in Spark, enabling data-parallel processing and fault tolerance. However, the success and ease of use of Spark come from the high-level APIs building transparently around this abstraction.

A Spark *job* is expressed as a directed acyclic graph (DAG), capturing the interdependencies between stages of independent *tasks*. Within this computational model, we can parallelize ML experiments by mapping trials to tasks. However, this approach has some limitations due to the synchronization barrier at the end of a stage that results in the inability to early stop a poorly performing trial during a stage and reuse the executor of the task's available computation resources for other trials during the rest of the stage. The early stopping can be added to Spark by enabling the driver to collect statistics on the performance of trials at executors periodically, and send messages to workers to stop poorly performing trials. However, such implementation of early stopping in Spark still wastes compute resources by not enabling new trials to be run until the end of the stage (Figure 2).

In contrary to the data-parallelism of Spark, Ray [9] is based on an actor concurrency model. It provides a flexible and asynchronous computational model expressed in stateful actors and stateless tasks, which are executed dynamically, allowing for one task or actor to spawn new actors/tasks. The Ray asynchronous computation model makes it more suitable for iterative workloads. Ray ships with a Python library for scalable hyperparameter tuning, called Tune [10]. It integrates with many ML frameworks (such as Keras, PyTorch, and XGBoost), and comes with its own implementations of popular optimization algorithms and provides support for a variety of thirdparty optimization libraries and services like HyperOpt [7], [8], Bayesian Optimization (BO) [11], and Google Vizier [12]. Due to its asynchrony, Tune can support early stopping, as well as multi-fidelity methods, such as HyperBand [13], BOHB [3], and ASHA [4].

HyperOpt [7], [8] is a Python library for distributed asynchronous HPO that has similar goals to MAGGY and was recently extended by a backend supporting distribution via Apache Spark. To overcome the inefficiencies of synchronous stages in Spark, Hyperopt maps ML trials to jobs with only one task. These jobs can then be executed asynchronously. However, this approach requires maintaining a thread for each scheduled job in the Spark driver, even if the job is not running yet, to retrieve the job's results. The Spark driver typically runs on few computational resources and can therefore become a bottleneck. Moreover, this design adds the overhead of starting a new job for each trial. This architecture also does not support global early stopping decisions. The optimizer is unaware of the current performance of the trials being trained. Therefore, it cannot make decisions on early stopping taking into account the knowledge about all learning curves.

Keras is a popular high-level API for TensorFlow [14] and comes with a Python library to tune models, KerasTuner [15]. KerasTuner integrates seamlessly with the Keras APIs and enables distributed experiments by starting the experiment script on different machines or processes. In the vision of Keras, KerasTuner integrates with the Google Cloud APIs to automate the process of starting worker nodes in the Google Cloud account of a user [16]. KerasTuner provides implementations of a variation of Hyper-Band [13] and Bayesian Optimization [11], but no explicit support for early stopping. A unique feature of KerasTuner is the possibility of intra-trial distribution to scale the training of single trials.

3 Hyperparameter Optimization (HPO)

In this section, we briefly recall some basic concepts from *hyperparameter* optimization (HPO). While AutoML aims to automate all aspects of the ML development process, a basic subproblem to solve is finding hyperparameters to maximize the performance of a model. Hutter et al. [17] provide a rich survey of AutoML methods, systems, and challenges, and classify HPO methods along two dimensions: *black-box* HPO and *multi-fidelity op- timization*. However, we believe that considering the underlying execution systems, a third dimension should be added, which is the *execution strategy*. The characteristics of methods in both the previous classes might be altered when executed in parallel or asynchronously, and the execution strategy dimension introduces more opportunities for new methods. This section serves

PAPER 2. MAGGY

as an overview for state-of-the-art HPO extended by considerations for the execution strategy and argues for the need for an asynchronous system to support these.

3.1 Black-box Hyperparameter Optimization

Blackbox optimization methods are split into two subsets, model-free (undirected search) and model-based (directed search) optimization. The former method, such as grid or random search, can be run in parallel without further coordination, as trials can be generated ahead of time. In particular, random search is a popular baseline, since it can find configurations with performance arbitrarily close to the optimum if it has enough computational resources [17].

On the other hand, model-based methods, like BO [11], are inherently sequential and require coordination to collect metrics and update the optimization model. BO samples the next trial to be evaluated based on previous iterations' results by using Bayesian posterior updates to a surrogate model, and encoding the prior belief over the objective function. The surrogate model's predictive distribution enables acquisition functions to determine the utility of different candidate points at low cost, trading off exploration and exploitation of the search space.

In the parallel setting, several points should be sampled based on the same information. However, if we apply deterministic strategies, each worker would evaluate the same configuration. A straightforward approach to deploy BO in an asynchronous parallel execution strategy is to impute the result of pending trials [18] with a constant (constant liar approach) or a Gaussian Process (GP) [18] predictive mean (Kriging Believer).

Other approaches, such as Thompson Sampling (TS) [19] or Tree Parzen Estimators (TPE) [7], use penalization around the locations of pending trials to encourage diversity (PLAyBOOK algorithm) [20] or sampling through a stochastic process, purposefully not to optimize the acquisition function fully to incorporate diversity. These asynchrony-enabling methods have shown to outperform their synchronous counterparts [7], [19], [20].

3.2 Multi-fidelity Optimization and Early Stopping

Multi-fidelity optimization methods rely on evaluating many trials on small computational budgets (low fidelities) and allocating more budget to a few promising trials. Here, for example, the budget can be the number of epochs for training a neural network and the amount of data used for training. Successive Halving (SHA) [21] and its successors HyperBand [13] and Asynchronous Successive Halving (ASHA) [4] are three examples of multi-fidelity

optimization. Both SHA and HyperBand or ASHA rely on random sampling to generate new hyperparameter configurations. In contrast, Falkner et al. [3] introduce BOHB that uses TPE [7] together with HyperBand [13] and achieve the performance above state-of-the-art results on several ML benchmark problems. While fidelity optimization makes the budget allocation decision before starting a trial, other approaches make early stopping decisions at runtime. Such methods are performance curve prediction [22] or simple heuristics like median stopping rules, as used by Google Vizier [12]. Again, these methods benefit from a central source of truth with knowledge of all trials' learning curves to make optimal early stopping decisions.

4 Maggy

In this section, we introduce MAGGY, a system for asynchronous parallel HPO. Below, we first describe MAGGY's programming model and then explain its implementation details. MAGGY is open-source and available at the following link¹.

4.1 Programming Models

Parallel computing support for model training and HPO offers many benefits, such as the ability to reduce training time and hyperparameter experiments by adding more compute resources . However, parallel execution introduces additional obtrusive code artifacts and modifications, depending on the frameworks used, which leads to infrastructure code mixed with model training code. The programming model of MAGGY can help avoid the problem of mixing infrastructure and training logic by enabling writeonce and transparently distributed training functions. The same code, then, can be reused in Python program on a laptop or a cluster-scale PySpark program. The programs, written in MAGGY framework, are *oblivious training functions* [23] as we factor out distribution-related dependencies using best-practice programming idioms (such as functions to generate models and data batches).

In MAGGY, users define the training logic in a (higher-order) function that returns the models performance metric (e.g., any loss or accuracy metric), which is to be optimized. The function is parametrized with hyperparameters and generator functions for the model and data (Listing 1). This function, then, is launched with a user-specified search space and optimizer through the $lagom^2$ API (Listing 2).

¹https://github.com/logicalclocks/maggy

²Lagom is a Swedish word meaning "just the right amount".

```
Listing 1 Example of an oblivious training function.
```

```
def train_fn(hparam1, hparam2, ..., model_fn, dataset_fn):
    model = model_fn(hparam1, hparam2)
    model.compile(hparam3)
    train, test, val = dataset_fn()
    model.fit(train, ...)
    metric = model.evaluate(test, ...)
    return metric["metric_to_be_optimized"]
```

Listing 2 Example of launching an experiment with lagom.

```
from maggy import experiment, Searchspace
searchspace = Searchspace(hparam1=('DOUBLE', [0,1]), ...)
experiment.lagom(train_fn, controller="BOHB", searchspace)
```

This way, MAGGY instantiates the training function with different sets of parameters and launches them as trials on Spark executors, without requiring users to write code managing the distribution and execution of the training logic on the workers. In return, the users will get the metrics to be optimized from the training function, or a collection of items to be tracked along with the experiment and specify which returned metric is to be optimized. Note that the produced code is still pure Python code, and it can be run on a cluster of machines as on a single host environment by fixing the parameters and inputs.

MAGGY currently ships with implementations of random search and BO (TPE [7] and GP [18]) as optimizers and HyperBand, ASHA, and a median stopping rule for early stopping. However, MAGGY provides base classes for both these entities as part of a developer API to make it extensible. Users can implement their own optimizers or early stopping rules.

4.2 Design and Implementation

MAGGY is built on top of Spark and provides an easy to use and scalable system for ML experiments, with support for GPUs from version 3.0. In principle, MAGGY uses Spark as a resource manager with enhanced fault tolerance support. MAGGY executes experiments as launching Spark applications, where the requested number of executors (degree of parallelism) are each blocked with one long-running task, executing trials in a loop until the experiment finishes.

MAGGY provides the aforementioned functionality through a non-blocking RPC framework built within the Spark *driver* and *executors* (Figure 3). On the driver-side, MAGGY runs a controller thread responsible for the ex-

PAPER 2. MAGGY



Figure 3: MAGGY is setup as a RPC framework within the Spark Driver and Executors. The figure shows the entities and the flow of information for the communication protocol and runtime behaviour.

periment's global control, such as trial generation and early stopping. It communicates with a RPC server thread by modifying controls in a shared data layer and a message queue. The RPC server then responds to the clients' requests by performing lookups on the shared data or forwarding the message to the controller. The shared data layer is required for the server not to block until the controller executed the remote procedure, such as sampling a new trial. To avoid the driver becoming a single point of failure, a distributed file system or cloud storage can be leveraged to persist controller state.

On the other side, each executor runs a RPC client that requests and starts new trials, sends heartbeats with the current training metric during training, and can early stop a trial when it receives a stop signal in response to a heartbeat. A client polls for new trials and receives early stopping decisions as a response to the heartbeats sent with the current training metric. The client is stateless, hence, in case of failure, Spark can easily restart the task and start a new client polling for trials. In scenarios of experiments with runs for long periods of time, this results in the loss of single trials, which are transparently rescheduled by the controller. A worker that repeatably fails to execute trials is blacklisted from receiving future trials.

A crucial point for collecting the current training metric for early stopping is the connection between the user code and the RPC client. In order to hook into the user code, users have two options (Listing 3), either (i) make use of a reporter API to broadcast the metric with a heartbeat at the end of an iteration manually, or (ii) if a high-level framework like Keras is used, MAGGY provides callbacks to be added to the training logic, doing the same thing automatically. Approach (i) is especially useful for cases

PAPER 2. MAGGY

when the iteration loop is programmed by the user itself, as it is the case in PyTorch, for example.

Listing 3 The reporter API is used to broadcast a specified metric in the heartbeats to the controller, or via the Keras Callback interface.

```
(i)
from maggy import reporter
for current_epoch in range(epochs):
    ...
    reporter.broadcast(metric=accuracy, step=current_epoch)
(ii)
from maggy.callbacks import KerasEpochEnd
callback = KerasEpochEnd(reporter, 'val_acc')
...
keras.fit(..., callbacks=[callback], ...)
```

5 Experiments

We evaluated MAGGY by comparing its performance with synchronous parallel trials on Spark (equivalent to existing parallel hyperparameter tuning frameworks on Spark, such as Databricks' HyperOpt [24]). We trained a three-layer convolutional neural network with a fully connected layer on the Fashion-MNIST [25] dataset. Compared to MNIST, Fashion-MNIST requires more time to train and is more difficult to get high accuracy on, enabling us to measure the effect of early stopping. We apply the median early-stopping rule [26] in MAGGY to stop trials performing worse than the median after the first four trials have completed at the same point in time during training (in terms of stochastic gradient descent optimization steps). In experiments on MAGGY and Spark (synchronous parallel trials), we run a fixed number of trials (N=100) with random search for hyperparameters. We vary the number of workers from 4, to 8, to 16, to 32. The space of hyperparameters explored using random search in both MAGGY and Spark is as shown in Listing 4.

Listing 4 Hyperparameter space for Fashion-MNIST.

```
sp = Searchspace(kernel=('INTEGER', [2, 8]),
    pool=('INTEGER', [2, 8]), dropout=('DOUBLE', [0.01, 0.99]),
    learning_rate=('DOUBLE', [0.000001, 0.99]))
```

The performance of hyperparameter tuning experiments is closely linked to the sensitivity of the model being tuned to small changes in hyperparame-

Table 1: Relative speedup of MAGGY over the general Spark implementation, total experiment runtime in seconds and number of early stopped trials by MAGGY.

Workers	MAGGY/Spark	MAGGY (s)	Spark (s)	Early-Stop
4	0.41	16284	40051	54
8	0.33	9828	29511	52
16	0.47	6486	13745	47
32	0.58	3804	6474	44

Table 2: Final accuracy after 100 trials.

Workers	MAGGY Accuracy	Spark Accuracy
4	0.915	0.905
8	0.909	0.912
16	0.909	0.913
32	0.913	0.909

ters and the relative number of points in hyperparameter space that contains poorly performing hyperparameter combinations. The Fashion-MNIST hyperparameter space used in these experiments is relatively homogeneous, and we can see that all experiments converged to very similar accuracy. Other networks, such as Generative Adversarial Networks are notoriously difficult to produce reproducible experiments.

As we can see in Figure 4 and Figure 5, MAGGY reduces the wallclock time for random search hyperparameter trials by roughly half when using the median early-stopping rule, without any loss in accuracy. In Table 1, we can see that the median stopping rule stops, on average, half of the trials, reducing total execution time by approximately half. In Table 2, we can see that both MAGGY and Spark converge to similar accuracy, even though half of MAGGY's under-performing trials were stopped early.

PAPER 2. MAGGY



Figure 4: Asynchronous trials and the median stopping rule in MAGGY enables N=100 trials to be executed in lower wallclock time compared to Spark without any loss in accuracy (denoted on top of the bars). Adding more workers linearly reduces the total time required to execute all hyperparameter trials, both for Maggy and Spark. MAGGY's reduced execution time holds for varying number of workers (W=4, 8, 16, 32).



Figure 5: MAGGY finds better configurations faster due to asynchronous trials and the median stopping rule in MAGGY. Due to shorter trials, MAGGY concludes experiments with the same number of trials in shorter wallclock time. In Spark, trials are executed to completion (no early stopping), yielding similar accuracy as expected, but resulting in higher wallclock time to execute N=100 trials compared to MAGGY.

6 Conclusion

Spark is now a popular general purpose programming framework that is used at all stages in machine learning pipelines, from feature engineering to parallel hyperparameter tuning to distributed model training. However, actor-based frameworks have shown better performance for asynchronous ML trials, leading many developers to switch part of their pipelines to such frameworks. In this paper, we introduced MAGGY as an extension to Spark's synchronous processing model to allow it to run asynchronous ML trials, enabling end-to-end state-of-the-art ML pipelines to be run fully on Spark. MAGGY provides programming support for defining, optimizing, and running parallel ML trials. Users can define their own global optimizer for directed search in a high-dimensional hyperparameter search space, and the MAGGY runtime will manage the performance monitoring, scheduling, and early-stopping of asynchronous trials within Spark's synchronous execution model.

Acknowledgement

This work is supported by the ExtremeEarth³ project funded by European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 825258.

References

- R. Meyes, M. Lu, C. W. de Puiseau, and T. Meisen, "Ablation studies in artificial neural networks," arXiv preprint arXiv:1901.08644, 2019.
- [2] M. Pezeshki, L. Fan, P. Brakel, A. Courville, and Y. Bengio, "Deconstructing the ladder network architecture," in *International Conference on Machine Learning*, 2016, pp. 2368–2376.
- [3] S. Falkner, A. Klein, and F. Hutter, "Bohb: Robust and efficient hyperparameter optimization at scale," arXiv preprint arXiv:1807.01774, 2018.
- [4] L. Li, K. Jamieson, A. Rostamizadeh, et al., "Massively parallel hyperparameter tuning," arXiv preprint arXiv:1810.05934, 2018.
- [5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, et al., "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

 $^{^3\}mathrm{ExtremeEarth}$ project website: http://earthanalytics.eu

- [6] M. Zaharia, M. Chowdhury, T. Das, et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12), 2012, pp. 15– 28.
- [7] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International Conference on Machine Learn*ing, 2013, pp. 115–123.
- [8] J. Bergstra, D. Yamins, and D. Cox, Hyperopt: Distributed asynchronous hyper-parameter optimization, 2012. [Online]. Available: http: //hyperopt.github.io/hyperopt.
- [9] P. Moritz et al., "Ray: A distributed framework for emerging AI applications," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 561–577.
- [10] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," arXiv preprint arXiv:1807.05118, 2018.
- [11] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.
- [12] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1487–1495.
- [13] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6765–6816, 2017.
- [14] M. Abadi, P. Barham, J. Chen, et al., "Tensorflow: A system for largescale machine learning," in 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), 2016, pp. 265–283.
- [15] T. O'Malley et al., Keras Tuner, https://github.com/kerasteam/keras-tuner, 2019.
- [16] F. Chollet, Keras: The next five years, Feb. 2020. [Online]. Available: https://www.youtube.com/watch?v=HBqCpWldPII.
- [17] F. Hutter, L. Kotthoff, and J. Vanschoren, Automated Machine Learning: Methods, Systems, Challenges. Springer Nature, 2019.

- [18] D. Ginsbourger, J. Janusevskis, and R. Le Riche, "Dealing with asynchronicity in parallel gaussian process based global optimization," 2011.
- [19] K. Kandasamy, A. Krishnamurthy, J. Schneider, and B. Póczos, "Parallelised bayesian optimisation via thompson sampling," in *International Conference on Artificial Intelligence and Statistics*, 2018, pp. 133– 142.
- [20] A. S. Alvi, B. Ru, J. Calliess, S. J. Roberts, and M. A. Osborne, "Asynchronous batch bayesian optimisation with improved local penalisation," arXiv preprint arXiv:1901.10452, 2019.
- [21] K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *Artificial Intelligence and Statistics*, 2016, pp. 240–248.
- [22] B. Baker, O. Gupta, R. Raskar, and N. Naik, "Practical neural network performance prediction for early stopping," arXiv preprint arXiv:1705.10823, vol. 2, no. 3, p. 6, 2017.
- [23] M. Meister, S. Sheikholeslami, R. Andersson, A. A. Ormenisan, and J. Dowling, "Towards distribution transparency for supervised ml with oblivious training functions," in *Workshop on MLOps Systems*, 2020.
- [24] Databricks, Scaling hyperopt to tune machine learning models in python, Oct. 2019. [Online]. Available: https://databricks.com/blog/ 2019/10/29/scaling-hyperopt-to-tune-machine-learningmodels-in-python.html.
- [25] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms," arXiv preprint arXiv:1708.07747, 2017.
- [26] L. Prechelt, "Early stopping-but when?" In Neural Networks: Tricks of the trade, Springer, 1998, pp. 55–69.

Paper 3

AutoAblation: Automated Parallel Ablation Studies for Deep Learning

The 1st Workshop on Machine Learning and Systems (EuroMLSys), co-located with the 16th European Conference on Computer Systems (EuroSys), 2021

AUTOABLATION: AUTOMATED PARALLEL ABLATION STUDIES FOR DEEP LEARNING

Sina Sheikholeslami¹, Moritz Meister², Tianze Wang¹, Amir H. Payberah¹, Vladimir Vlassov¹, and Jim Dowling^{1,2}

¹ KTH Royal Institute of Technology, Stockholm, Sweden ² Logical Clocks AB, Stockholm, Sweden

Abstract

Ablation studies provide insights into the relative contribution of different architectural and regularization components to machine learning models' performance. In this paper, we introduce AU-TOABLATION, a new framework for the design and parallel execution of ablation experiments. AUTOABLATION provides a declarative approach to defining ablation experiments on model architectures and training datasets, and enables the parallel execution of ablation trials. This reduces the execution time and allows more comprehensive experiments by exploiting larger amounts of computational resources. We show that AUTOABLATION can provide near-linear scalability by performing an ablation study on the modules of the Inception-v3 network trained on the TenGeoPSAR dataset.

1 Introduction

Inspired by how the mammalian brain works, Deep Neural Networks (DNNs) have been at the forefront of recent breakthroughs in Artificial Intelligence. Since the early 19th century, a surgical procedure called an *ablation study* has been developed to understand the role of different components of the brain [1]. An ablation study involves removing a specific part of the brain of a mammal, and observing any resulting changes in its behavior. Given our limited understanding of brain function, this black-box approach has helped identify regions in the neocortex that are specialized for controlling specific behaviours and the relative contribution of brain regions to global function.

Similar to brains, we lack models to understand the function of Deep Learning (DL) systems at both the macro and micro levels. As such, blackbox experiments that modify model architectures while observing system performance offer an approach to help improve our understanding of a DL

PAPER 3. AUTOABLATION



Figure 1: Example trial configurations for model ablation (up) and feature ablation (bottom). Yellow highlighting indicates a "component" that is removed for the trial.

system. An ablation study in DL involves measuring the performance of a network after removing one or more of its components to help understand the relative contribution of the ablated components to overall performance [2]. Dataset features and model components (e.g., layers) are notable examples of ablatable components, but any design choice or module of the system can be considered in an ablation study.

In this setting, we consider the execution of an ablation study on a given DL model or dataset as a single *experiment* consisting of several *trials*, where each trial involves removing one or more model or dataset building blocks, e.g., layers or features. We distinguish two kinds of ablation trials, namely, *model ablation* trials and *feature ablation* trials, depending on what kind of block is ablated in the trial: model components (e.g., layers or modules), or dataset features, respectively. Each model ablation trial involves training and evaluating a model with one or more of its components (e.g., layers) removed (Figure 1, (up)). Similarly, a feature ablation trial involves training and evaluating the model using a different subset of features in the dataset (Figure 1, (bottom)).

Over the years, many Machine Learning (ML) papers have included ablation studies [3]–[7]. Moreover, following the recent trend towards explainable and interpretable ML systems, several recent works [8], [9], discuss how ablation studies can lead to more explainable ML models. However, it can be observed that a significant part of the ML research community still regards performing ablation studies to be unnecessary, and when researchers publish or propose new model architectures or training procedures, they may attribute the resulting gains only to the changes they have made to a base model architecture or training procedure, without performing any ablation studies that would allow to identify and quantify the actual impact of each of these proposed changes [8].

Based on our observations, two main reasons for this oversight are: (i)

performing ablation studies requires maintaining redundant copies of code that each correspond to a different configuration of the model or the dataset, and (ii) evaluating these different configurations requires extra time and compute resources. Looking closer at these challenges, however, reveals trivial yet interesting characteristics of ablation studies. When talking about the existence of redundant copies of code for ablation trials, we can observe that these copies are almost identical, except for the part related to the specific ablation trial. Moreover, executing a set of ablation trials is an embarrassingly parallel task.

In this work, we exploit the above characteristics to design and develop a framework to overcome the aforementioned challenges. Our framework, AUTOABLATION, is based on the concept of the *distribution oblivious training function* [10], in which we decouple *model creation* and *dataset creation* functions from the training function. This decoupling allows us to (i) eliminate the need for maintaining redundant copies of code for ablation trials, and (ii) provide distribution transparency for ML developers so the code that is developed for execution of ablation trials on a single host can easily be executed in parallel on a cluster of machines. This practice has recently enjoyed increased adoption in the community, as can be seen in the programming model of libraries such as PyTorch Lightning [11] and Keras Tuner [12].

With AUTOABLATION, we introduce a novel way to define and parallelize the execution of ablation studies for (i) DL model architectures, and (ii) training datasets. By decoupling model creation and dataset creation from the training function, we have come up with a simple and declarative Application Programming Interface (API) that eliminates the need for maintaining redundant copies of code for ablation studies. Furthermore, our framework enables parallel execution of ablation trials without requiring the developers to modify their code, which leads to shorter study times and better resource utilization. To the best of our knowledge, this is the first framework that provides support for the specification and parallel execution of ablation studies for DL. We demonstrate the usability and scalability of AUTOABLATION through three common scenarios in which ablation studies may be performed.

2 Preliminaries

In this section we provide a formal definition of ablation studies in DL, and describe the parallel execution of trials.

2.1 A Formal Definition of Ablation Study

Given a training dataset D and a model M, in training of the model, we aim to optimize its parameters with regards to an objective function (e.g., Mean Squared Error or Binary Cross-Entropy) using an iterative optimization algorithm (e.g., Stochastic Gradient Descent). In practice, developing a performant DL model requires many design decisions and trying out several *configurations* C. The goal of an ablation study is to investigate the relative contribution of each of these configurations to the performance of the model. A configuration can be a *dataset configuration* C_D or a *model configuration* C_M .

The dataset configuration describes what features of a given dataset we need to exclude for training. For a dataset X with n features, $C_D(X, \{x\})$ indicates that the features in $\{x\}$ should be excluded during training the models. For example, $C_D(X, \{x_1, x_3, x_4\})$ means to skip features 1, 3, and 4, and use the rest of the features to train the models. Similarly, the model configuration illustrates the architecture of a model. To be more precise, for a given model M with k components, $C_M(M, \{m\})$ means to exclude the listed components in $\{m\}$ during training M. A component can be a layer, a set of neurons, a filter, and so on. For example, if M is a Convolutional Neural Network (CNN) with two convolution layers (c_1 and c_2), one pooling layer in between (p_1), and one dense layer at the end (d_1), then $C_M(M, \{c_1, p_1\})$ means to remove the first convolution layer and the pooling layer from M and train the model with the rest of component, i.e., $\{c_2, d_1\}$.

We define a study S as a set of either dataset configurations, $S_D = \{C_D\}$, or model configurations, $S_M = \{C_M\}$. For example, $S_M = \{C_{M_1}, C_{M_2}, \dots, C_{M_z}\}$ means to train the model M with z different model configurations, such that in training with configuration C_{M_i} , we only consider the components of the model M, which are not listed in that configuration. We call the execution of an study as an *experiment*, which consists of several *trials*, where each trial corresponds to a configuration. Given the set of components that are to be ablated in an ablation experiment, an *ablation policy* specifies the trials that constitute the experiment. An *ablator*, in turn, is an implementation of an ablation policy that materializes the trials of the experiment. A simple policy could be to remove (exclude) one component per each trial. This is perhaps the most common form of performing ablation studies, and we refer to it as *Leave-One-Component-Out* (LOCO) ablation.

2.2 Parallel Execution of Trials

Over the last few years, several ML and DL frameworks and libraries have been introduced, such as TensorFlow [13], PyTorch [14], and Keras [15]. To make model ablation possible, the underlying DL framework should provide ways for exporting configuration representations of the models, and ways to distinguish different components. All of the above mentioned frameworks fulfill this requirement, e.g., Keras enables developers to set the name parameter for layers of a DNN, and export the configuration representation of a model in various formats, such as JSON.

The above mentioned frameworks, however, lack support for parallel and distributed execution of DL experiments consisting of independent trials. For example, TensorFlow and PyTorch provide distributed training capabilities for single models, but practitioners are often left building their own solutions to parallelize their experiments. Therefore, efforts have been made either to develop new frameworks for distributing DL workloads (e.g., Ray [16]), or to use existing distributed processing engines for DL workloads.

Among big data processing frameworks, Apache Spark [17] has been the target of several such efforts [18], as it has become an industry standard for data processing and engineering tasks. TensorFlowOnSpark [19] runs distributed training of a single model with TensorFlow within a Spark job, where each task within this job will serve as a worker process. However, mapping each trial in an experiment to a Spark task results in poor resource utilization, as running iterative jobs on Spark follows the Bulk Synchronous Parallel (BSP) execution model. Stages in Spark introduce task synchronization barriers, and for jobs to proceed to a new stage, all tasks (trials) from the previous stage have to be completed. In case of ablation studies, some trials may take significantly longer time to train due to their configurations, i.e., their specific model architecture, or the dataset subset. Hence, asynchronous execution of trials on an Apache Spark cluster would be highly desirable.

MAGGY [20] is a framework for asynchronous execution of trials on Apache Spark clusters. MAGGY launches a single Spark job for the whole experiment, and on each Spark executor, one long running task will be run to execute the trials. Once the evaluation of a trial is finished (or stopped) on an executor, the same task on the executor will be reinitialized with a new trial configuration. A Controller thread, running on the driver side, is responsible for generating new trial configurations. Depending on the nature of the experiment, the Controller can include an *optimizer* (for hyperparameter tuning experiments), or an *ablator* (for ablation studies). Currently, MAGGY supports asynchronous, parallel execution of hyperparameter tuning experiments [20], and with AUTOABLATION we extend it to also support ablation study experiments.

Through a communication model based on non-blocking Remote Procedure Calls (RPC), once the job is launched (or an executor finishes an assigned trial), the executors can communicate with the driver and poll the controller for new trials independent of other executors. This removes the

PAPER 3. AUTOABLATION

barrier (synchronization step) imposed by the BSP execution model, leading to increased resource utilization as well as reduced total run-time of experiments.

3 AutoAblation

Currently, AUTOABLATION supports model ablation and feature ablation of DNNs. Model ablation is possible in form of individual layers, groups of layers, and modules (e.g., an Inception module), and feature ablation is possible in form of individual features or groups of features. To address the two challenges of (i) redundant code maintenance, and (ii) efficient parallel execution of ablation trials, we exploit the fact that the training logic remains largely unchanged between different trials. When we want to investigate the contribution of different components of a DL model to its overall performance, we have to construct different variants of the model architecture and apply the same training logic on these variants, using the same training data. Similarly, if we are interested in the importance of each feature of our training dataset, we use different combinations of the features to train the same model, using the same training logic.

Following the above observations, the programming model of AUTOAB-LATION is based on the decoupling of the model creation and dataset creation from the training logic. In other words, instead of having model creation, dataset creation, and training logic in a single block of code, the user wraps the training code in a function that is parameterized by dataset creation and model creation functions. This decoupling and parameterization allows the framework to automatically generate and replace parts of the logic that are specific to each trial. Performing an ablation experiment in AUTOABLATION consists of three steps¹: (i) defining the training components (including model creation and dataset creation), (ii) defining the ablation study, and (iii) executing the ablation trials in parallel. We will explain this workflow, as shown in Figure 2, in the following subsections.

3.1 Defining the Training Components

The first step is to define the actual training loop. This step is always part of the ML process, irrespective of whether an ablation study will be performed or not. The important thing here, however, is that the user has to decouple the model creation and the dataset creation from the training function. In most cases, this is equivalent to moving the code blocks responsible for the model creation and the dataset creation to their own functions, e.g.,

¹See https://maggy.readthedocs.io/ for the API documentation.

PAPER 3. AUTOABLATION



Figure 2: Workflow of an ablation experiment.

create_model() and create_dataset(), and passing them as arguments to the training function. A skeleton code for the first step is shown in Listing 1.

Model Creation. Here, the user has to wrap the model creation code in a Python function that we refer to as the *base model* function, which receives trial-specific parameters (e.g., layer identifiers) and returns a trainable model that can be used in the training function.

Dataset Creation. Similar to model creation, the process of creating the train/test/validation sets that will be used in the train/test/validation loops should be wrapped in its own function. The user can implement their own function for creating these sets or use default dataset creation functions shipped with AUTOABLATION.

Training Function. The training function is the actual pure Python code block that will be executed either on a single host or in parallel on a cluster of workers, and contains the code for training a DL model using a training dataset. In a typical implementation of a DL application, the whole process of preparing the train/test/validation sets, model architecture definition, and model training is implemented in a monolithic style; but in our programming model, as the user has already implemented the model creation and the dataset creation functions in the previous sub-steps, the model function and the dataset function are passed as arguments to the training function, and will instantiate the model and the dataset(s) once called.

3.2 Defining the Ablation Study

The next step is to define the ablation study by specifying the model configuration and dataset configuration (S_D and S_M , as defined in Section 2.1). To this end, the user has to create an AblationStudy instance and initialize it with the default model creation and dataset creation functions defined in

```
Listing 1 Defining the training components.
```

```
# define the model creation logic
def base_model(trial_params):
    # create the model ...
    return model
# define the dataset creation logic
def base_dataset(trial_params):
    # create the dataset ...
    return dataset
# define the training logic, parametrized by the model and dataset
def train(model_func, dataset_func):
    model = model_func()
    data = dataset_func()
    metric = model.fit(data)
    return metric
```

Listing 2 Defining the ablation study.

```
# define the ablation study
study = AblationStudy()
study.model.set_base_model_generator(base_model)
study.set_dataset_generator(base_dataset)
study.features.include('feature_name')
study.model.layers.include('layer_name')
study.model.add_module('module_name')
# launch the experiment
experiment.launch(train, study)
```

the previous step. After this, the user should specify which configurations they want to *include* in the study. Currently, AUTOABLATION API provides methods for defining configurations for dataset features, model layers, layer groups, modules, and custom models. Example usage of the API for defining an ablation study is shown in Listing 2.

3.3 Launching the Experiment

The final step is to invoke an API call that mainly receives the training function and the study specification, and initiates the execution of the trials through MAGGY, either sequentially on a single host or in parallel on a cluster of nodes.

4 Implementation

AUTOABLATION runs on top of MAGGY, an open-source Python-based framework for asynchronous execution of ML trials on top of Apache Spark. The experiment is launched as a Spark application that generates the trials of the experiment, and MAGGY distributes the trials on the set of available worker nodes (executors). Below, we explain how AUTOABLATION generates trials based on the ablation study specification defined in the second step of the workflow in Section 3.

4.1 Implementing the LOCO Ablator

As discussed in Section 2.1, an ablator is an implementation of an ablation policy. In AUTOABLATION, ablators are implemented as Python classes. The Controller thread in the Spark job creates an instance of an ablator class and uses it to generate Trial objects that contain the model creation and dataset creation functions specific to each trial. To execute a trial, an executor requests a new trial configuration from the Controller. If there is a trial to be evaluated, the executor will be sent a Trial object. The executor then de-serializes the object and unpacks its contents, and then passes them as arguments to the training function, and executes the training function.

The LOCO ablator uses the dataset creation and model creation functions to generate Trial objects that are then shipped to the executors as they request new trial configurations. Given an AblationStudy instance that contains different configurations, in order to create customized models and datasets for each trial, an ablator must modify the base model or dataset by removing these components. LOCO does this through parsing and modifying "configuration representations" of the components. For datasets, this is equivalent to the dataset schema (which can come in different formats). For models, many DL frameworks provide ways for saving or exporting model configurations, e.g., through JSON files (as in Keras) or serializable dictionaries (as in PyTorch). Hence, an ablator essentially implements the process of parsing and modifying these configuration representations and generating trials according to an ablation policy.

Given the schema of the base dataset, the LOCO ablator modifies the base schema to create a new schema for each feature ablation trial, and generates its corresponding create_dataset function. To generate create_model functions specific to each trial, the LOCO ablator uses the base model function (as described in Section 3.1) to export the configuration representation of the base model, and then parses it to find and select model components defined in the AblationStudy instance of the experiment. It then modifies the configuration representations and generates new create_model functions for each trial.

Input or output shape changes that may result from removal of components are either handled by the underlying framework (e.g., when removing layers of a model developed with Keras Sequential API), or require explicit handling in the implementation of the ablator (e.g., by using a randomly initialized tensor as the input of one forward pass of the modified model, to infer the correct shapes). However, if a trial cannot be automatically generated, the user still has the option to create a custom trial with their own model creation and dataset creation functions, and add it to the experiment. Finally, the LOCO ablator creates the corresponding **Trial** objects, and populates the buffer of trials that the executors can poll as the experiment is launched.

4.2 Parallel Execution of Trials

To execute different trials of an experiment, each Spark executor needs to have the training function that is parametrized by the create model and create_dataset function objects. The training function is supposed to remain the same through all trials, so it will be sent to the executors as the experiment is launched. The two parameters of the training function, will be provided through the **Trial** objects created by the LOCO ablator. The executors will then register with the MAGGY driver, and start polling the server for these objects. Depending on the ablation policy, a number of initial trials will be generated on the driver side; in the case of LOCO, since the number of trials can be determined from the components included in the ablation study, the ablator will generate all trials and put them in a buffer, which will be queried by the Controller every time an executor requests a new trial configuration. It should be noted that the Spark job is started and managed by MAGGY, and the start-up only takes a few seconds, which is negligible compared to the actual time it takes to train the model variants.

5 Evaluation

In this Section, we demonstrate three common scenarios, in which ablation studies can be performed, and show how we can define and execute such studies with AUTOABLATION². Below, we first evaluate the performance of AUTOABLATION in two different experiments: (i) feature ablation and (ii) model ablation, and then we show how it performs in various levels of parallelization.

EXP1: Feature Ablation of the Titanic Dataset. In this experiment, we perform feature ablation on a customized version of the Titanic dataset³.

 $^{^2{\}rm The}$ reproducible experiments: https://github.com/ssheikholeslami/ablation-paper-experiments

³https://www.kaggle.com/c/titanic/data

There are six features in the dataset in addition to the label, so we will have seven trials (including one base trial that contains all the features). The model we use is a simple Keras Sequential model with two hidden **Dense** layers. We keep 20% of the data as the test set and train on the rest for 10 epochs. Listing 3 shows the code required to define this experiment.

```
Listing 3 Defining the feature ablation experiment.

from maggy.ablation import AblationStudy
study = AblationStudy('titanic_train_dataset', label_name='survived')
list_of_features = ['pclass', 'fare', 'sibsp', 'sex', 'parch', 'age']
study.features.include(list_of_features)
```

After repeating the experiment five times, we can rank the features in terms of their average effect on the test accuracy, as shown in Table 1. For example, we observe that training the model with all the features (None) results in the worst test accuracy, while removing the fare feature from the training dataset leads to the best performance.

Table 1: Average accuracy on the test set resulting from excluding each feature from the training set.

Excluded Feature	Test Accuracy
None (base trial)	0.583
pclass	0.596
sex	0.609
sibsp	0.616
age	0.667
parch	0.672
fare	0.695

EXP2: Model Ablation of a Keras Sequential Model. In this experiment, we train a CNN to classify handwritten digits of the MNIST dataset [21]. The network has two Conv2D layers, followed by one MaxPooling2D layer, one Dropout layer, a Flatten layer, one Dense layer, another Dropout layer, and one Dense output layer. Our target is to investigate the relative contribution of the second Conv2D layer, the Dense layer, and the first and second Dropout layers to the performance of the model. The study can be defined using the code shown in Listing 4. After repeating the experiment five times, we can rank the selected layers in terms of their average effect on the test accuracy, as shown in Table 2. We can see that removing the second Conv2D layer has the worst effect on the test accuracy, while removing the Dropout layers results in a better performance than the performance of the base model.

EXP3: Model Ablation of Inception-v3. With this experiment, we

Table 2: Average accuracy on the test set resulting from excluding layers of interest from the base model.

Excluded Layer	Test Accuracy
second_conv	0.913
dense_layer	0.954
None (base trial)	0.969
second_dropout	0.982
first_dropout	0.988

demonstrate the near-linear scalability achieved by parallel execution of ablation trials with AUTOABLATION. We perform an ablation study on seven modules of the Inception-v3 network [22] in a transfer learning task on a subset of the TenGeoPSAR dataset [23]. This subset contains 5000 Synthetic Aperture Radar (SAR) images. We split the dataset into train (3200 images), validation (800 images), and test (1000 images) sets. The images are labeled with one of 10 classes, each representing a geophysical phenomena.

We load the network using Keras Applications API with pre-trained ImageNet [24] weights, and replace its output layer to suit our 10-class classification task. The Inception-v3 network consists of 11 blocks also known as "inception modules", and we are interested to know how each of the first seven modules affect the performance of the network (measured by the accuracy on the test set). Since this is a predefined network, we first compile it to find out about the names of the layers, and identify the entrance and end point of each module either by plotting the architecture or observing the model.summary() output information in Keras. Once we identify the layers, defining the ablation study can be done with the code shown in Listing 5.

```
Listing 5 Defining the Inception-v3 module ablation experiment.
from maggy.ablation import AblationStudy
study = AblationStudy("TenGeoPSARwv", 1, "type",)
study.model.add_module('max_pooling2d_1', 'mixed0')
study.model.add_module('mixed0', 'mixed1')
study.model.add_module('mixed1', 'mixed2')
...
study.model.add_module('mixed5', 'mixed6')
```

Each trial consists of 40 epochs of training, and we run the experiment in three settings: (i) a single executor (sequential, no parallelization), (ii)
PAPER 3. AUTOABLATION



Figure 3: AUTOABLATION provides near-linear scalability by parallelizing the execution of ablation trials.

two executors, and (iii) four executors. The total run-time for each of these settings is reported in Figure 3. We take the run-time of the sequential run as a baseline to approximate linear scalability; however, we should keep in mind that the ablation trials differ in their run-time since their model architectures are different from one another. We can conclude from Figure 3 that AUTOABLATION provides near-linear scalability by parallelizing the execution of ablation trials.

6 Related Work

Recently there have been many efforts to build frameworks, libraries, and tools to inquire insights regarding the performance of DL models or the effect of different dataset configurations in their performance. Many of such efforts address the problem of Interpretability and Explainability of ML/DL models⁴. Libraries such as LIME [25], SHAP [26], and TensorFlow's What-If Tool [27] provide extensive tools and visualizations for explaining the behaviour and outputs of ML/DL models through post-hoc analysis. DeepBase [28] is a system for deep neural inspection that provides a declarative API for defining hypothesis functions and then evaluates those hypotheses over a sequence of inputs. DeepBase is similar to AUTOABLATION as it shares a design requirement to reduce the amount of effort for performing model inspections, but with AUTOABLATION the same code can be used for hyperparameter tuning, distributed training, and other types of DL experiments [10]. LOFO-Importance [29] is a library that provides Leave-One-Feature-Out importance for datasets used to train models, by excluding one feature out of the training set at a time, and retraining the model on that subset. However, it does not provide support for model ablation experiments.

 $^{^4{\}rm A}$ list of related open-source projects can be found in: https://github.com/EthicalML/awesome-production-machine-learning

7 Conclusion and Future Work

In this paper, we introduced AUTOABLATION, a new framework for the design and parallel execution of ablation studies of deep learning models. We formulated an ablation study as an experiment that consists of several trials, where each trial represents a specific model architecture or dataset schema. We also presented a new programming model for designing an experiment that is based on the decoupling of model creation and dataset creation from the training function. We introduced the concept of the ablation policy that specifies what should be the trials that make up an ablation experiment, implemented in form of an ablator. Moreover, we showed how we leverage parallel execution of trials to speed up the total study time and increase resource utilization, through our Python-based execution framework called MAGGY. Through the experiments, we showed that AUTOABLA-TION provides near-linear scalability. Our next step would be to develop a generalized approach for handling shape mismatch issues, and to support automatic generation of more complex ablation trials and policies, e.g., cases in which removal of a layer requires other changes in the components of a model. As AUTOABLATION gets picked up by more users, we will use their feedback to provide support for more common ablation scenarios.

Acknowledgement

This work is supported by the ExtremeEarth⁵ project funded by European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 825258. The authors would like to thank Farzad Nozarian and Desta Hagos for their detailed feedback on the draft of this paper.

References

- [1] N. Carlson et al., *Psychology: the Science of Behavior*. Pearson, 2009.
- [2] S. Sheikholeslami, "Ablation programming for machine learning," M.S. thesis, 2019.
- [3] E. Horvitz et al., "Learning and reasoning about interruption," in Proceedings of the 5th International Conference on Multimodal Interfaces, ACM, 2003, pp. 20–27.
- [4] M. Richardson et al., "Beyond pagerank: Machine learning for static ranking," in *Proceedings of the 15th International Conference on World* Wide Web, ACM, 2006, pp. 707–715.

⁵ExtremeEarth project website: http://earthanalytics.eu .

- [5] R. Girshick et al., "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 580– 587.
- [6] M. Hessel et al., "Rainbow: Combining improvements in deep reinforcement learning," in 33 AAAI Conference on Artificial Intelligence, 2018.
- [7] D. Berthelot et al., "Mixmatch: A holistic approach to semi-supervised learning," arXiv preprint arXiv:1905.02249, 2019.
- [8] Z. C. Lipton and J. Steinhardt, "Troubling trends in machine learning scholarship," *arXiv preprint arXiv:1807.03341*, 2018.
- R. Meyes et al., "Ablation studies in artificial neural networks," arXiv preprint arXiv:1901.08644, 2019.
- [10] M. Meister et al., "Towards distribution transparency for supervised ml with oblivious training functions," in Workshop on MLOps Systems, 2020.
- W. A. Falcon et al., "Pytorch Lightning," GitHub. https://github.com/PyTorchLightning/pytorch-lightning, vol. 3, 2019.
- [12] T. O'Malley et al., Keras Tuner, https://github.com/kerasteam/keras-tuner, 2019.
- [13] M. Abadi et al., "Tensorflow: A system for large-scale machine learning," in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 265–283.
- [14] A. Paszke et al., "Pytorch: An imperative style, high-performance deep learning library," Advances in Neural Information Processing Systems, vol. 32, pp. 8026–8037, 2019.
- [15] F. Chollet et al., *Keras*, 2015.
- [16] P. Moritz et al., "Ray: A distributed framework for emerging AI applications," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 561–577.
- [17] M. Zaharia et al., "Spark: Cluster computing with working sets," *Hot-Cloud*, vol. 10, no. 10-10, p. 95, 2010.
- [18] B. Chambers and M. Zaharia, Spark: The Definitive Guide: Big Data Processing Made Simple. O'Reilly Media, Inc., 2018.
- [19] L. Yang et al., Open sourcing tensorflowonspark: Distributed deep learning on big-data clusters, 2017.

- [20] M. Meister et al., "Maggy: Scalable asynchronous parallel hyperparameter search," in Workshop on Distributed Machine Learning, 2020, pp. 28–33.
- Y. LeCun, The MNIST database of handwritten digits, http://yann.lecun.com/exdb/mnist/, 1998.
- [22] C. Szegedy et al., "Rethinking the inception architecture for computer vision," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.
- [23] C. Wang et al., "A labelled ocean sar imagery dataset of ten geophysical phenomena from sentinel-1 wave mode," *Geoscience Data Journal*, vol. 6, no. 2, pp. 105–115, 2019.
- [24] J. Deng et al., "Imagenet: A large-scale hierarchical image database," in 2009 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 2009, pp. 248–255.
- [25] M. T. Ribeiro et al., ""why should i trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 1135–1144.
- [26] S. M. Lundberg and S. Lee, "A unified approach to interpreting model predictions," *Advances in Neural Information Processing Systems*, vol. 30, pp. 4765–4774, 2017.
- [27] J. Wexler et al., "The what-if tool: Interactive probing of machine learning models," arXiv preprint arXiv:1907.04135, 2019.
- [28] T. Sellam et al., "Deepbase: Deep inspection of neural networks," in Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 1117–1134.
- [29] A. Erdem et al., Leave one feature out importance, https://github. com/aerdem4/lofo-importance, 2019.

Paper 4

The Impact of Importance-aware Dataset Partitioning on Data-parallel Training of Deep Neural Networks

The 23rd IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), 2023



THE IMPACT OF IMPORTANCE-AWARE DATASET PARTITIONING ON DATA-PARALLEL TRAINING OF DEEP NEURAL NETWORKS

Sina Sheikholeslami¹, Amir H. Payberah¹, Tianze Wang¹, Jim Dowling^{1,2}, and Vladimir Vlassov¹

¹ KTH Royal Institute of Technology, Stockholm, Sweden ² Hopsworks AB, Stockholm, Sweden

Abstract

Deep neural networks used for computer vision tasks are typically trained on datasets consisting of thousands of images, called examples. Recent studies have shown that examples in a dataset are not of equal importance for model training and can be categorized based on quantifiable measures reflecting a notion of "hardness" or "importance". In this work, we conduct an empirical study of the impact of importance-aware partitioning of the dataset examples across workers on the performance of dataparallel training of deep neural networks. Our experiments with CIFAR-10 and CIFAR-100 image datasets show that data-parallel training with importance-aware partitioning can perform better than vanilla data-parallel training, which is oblivious to the importance of examples. More specifically, the proper choice of the importance measure, partitioning heuristic, and the number of intervals for dataset repartitioning can improve the best accuracy of the model trained for a fixed number of epochs. We conclude that the parameters related to importance-aware data-parallel training, including the importance measure, number of warmup training epochs, and others defined in the paper, may be considered as hyperparameters of data-parallel model training.

1 Introduction

Data-parallel training (DPT) is the current best practice for training deep neural networks (DNNs) on large datasets over several computing nodes (a.k.a. workers) [1]. In DPT, the DNN (model) is replicated among the workers, and the training dataset is partitioned and distributed uniformly among them. DPT is an iterative process where in each iteration, each worker trains its model replica on its dataset partition for one epoch. After each iteration, the parameters or gradients of the worker models are aggregated and updated. Then, all workers continue the training using the same updated model replicas. This "vanilla" DPT scheme is shown in Figure 1.

The dataset partitions in vanilla DPT are constructed by random partitioning, i.e., randomly assigning training examples to each partition. However, it is known that not all examples within a training dataset are of equal importance for training DNNs [2]–[5] meaning that different examples contribute differently to the training process and the performance of the trained model (e.g., its prediction accuracy). Prior works have used example importance to improve DNN training schemes, mainly aiming at reducing the total training time or increasing the performance of the trained models. For example, in dataset subset search [3], the goal is to find subset(s) of a given training dataset that can be used to train equally good or more performant models compared to the models trained on the initial dataset. Example importance has also been used for developing more effective sampling algorithms for stochastic gradient descent (SGD) [5], or in active learning for choosing the best examples to label [4].

Contributions. All the above-mentioned solutions are mainly designed for non-distributed model training. In this paper, we study different heuristics to assign examples, based on their importance, to workers in a distributed environment and in DPT. In particular, the contributions of this work are as follows.

- We introduce *importance-aware* DPT, which replaces the random partitioning of the dataset across workers in vanilla DPT, with heuristics that partition the dataset based on some pre-determined notion of example importance, e.g., the average loss value of each example over a number of training epochs.
- We study the effects of the hyperparameters of importance-aware DPT, including different (i) example importance measures and metrics, (ii) partitioning heuristics, and (iii) partitioning intervals, on the quality of the training scheme. Our experiments for image classification tasks on CIFAR-10 and CIFAR-100 datasets demonstrate that importance-aware DPT can outperform vanilla DPT in terms of the



Figure 1: The vanilla DPT scheme with four workers and one parameter server. At each epoch, each worker gets a random partition of the dataset, and all the workers are assigned the same model replica. After one epoch of training, the workers send their local gradients or model parameters to the parameter server. The parameter server performs either gradient aggregation or model aggregation and sends back the new gradients or parameters to the workers.

best test accuracy achieved by models.

The remainder of this paper is structured as follows. In Section 2, we provide the necessary background, including an overview of DPT and a review of some related work. In Section 3, we present importance-aware DPT and discuss how it differs from vanilla DPT, which is importance-oblivious. In Section 4, we discuss our prototype implementation of importance-aware DPT in PyTorch. In Section 5, we present the results of our experimental evaluation of importance-aware DPT. Finally, in Section 6, we give our conclusions and discuss the current limitations of our importance-aware DPT prototype and further research directions.

2 Background and Related Work

Our work presented in this paper lies in the intersection of data-parallel DNN training and prior work that studies the difference of examples within a dataset in terms of their importance for model training. In this section, we give a brief overview of the DPT of DNNs and some related work on example importance.

2.1 DNN Data-Parallel Training (DPT)

Given a training dataset D consisting of training examples $e \in D$, the aim of training the model M is to optimize model parameters with regards to a cost function, e.g., Mean Squared Error or Binary Cross-Entropy, using an iterative optimization algorithm, e.g., Stochastic Gradient Descent. A training dataset is typically made up of examples of a specific type, such as images, structured data, or sentences. During each *epoch* of training, batches of examples are passed through the model, and model parameters are optimized using the iterative optimization algorithm. To scale out the training process, one can use multiple processing nodes, a.k.a. *workers*, and partition the DNN (for model-parallelism) or the dataset (for DPT) and assign them to the workers to enable parallel training. For our purposes, we define a worker $w \in W$ as a process within a processing node that is allocated exactly one GPU, i.e., each worker corresponds to exactly one GPU in our cluster of processing nodes.

In a typical most common DPT scheme, which we refer to as *vanilla* DPT, the DNN is replicated across the workers. At the beginning of each epoch, the dataset is partitioned uniformly at random into disjoint subsets $p \in P$, such that worker w_i is allocated the partition p_i (dataset partitioning step). More formally, $P = \bigcup_{i=0}^{n-1} p_i$ such that $p_i \cap p_j = \emptyset$ for $i \neq j$; and $p_i \neq \emptyset$ for each *i*. For simplicity, we assume that the number of examples in the dataset, or |D|, is divisible by the number of workers, n = |W|; but the approach and results can easily be extended to cases where the assumption does not hold.

During an epoch, each worker independently trains its own replica of the DNN model (local training step) on its own partition p_i . At the end of an epoch, a model synchronization step occurs, e.g., using a parameter server, and the workers get a new identical replica of the model. This process is repeated for a specified budget (e.g., a pre-determined number of epochs) or until a model convergence criterion or performance metric is satisfied. We are interested to see if using a partitioning function, based on notions of example importance, may lead to better results compared to vanilla DPT's random partitioning in terms of the target performance metrics. We define the importance of an example, denoted by Imp, as a mapping of an example to a scalar value:

$$Imp: e \to \mathbb{R} \tag{1}$$

In practice, to implement Imp, a certain property of the example or the result of its interactions with the model (e.g., the loss generated by the example after a forward pass) is used in combination with an aggregation method (e.g., average, or variance of the losses over a number of epochs).

A partitioning function *PartitioningFunction* maps the examples to

workers to create the set of partitions P, where each worker w_i gets the partition p_i . We are interested in using the output of Imp to construct the *PartitioningFunction*. Example definitions for a *PartitioningFunction* are explained in Section 3.3.

2.2 Prior Work on Example Importance

The diversity of examples in training datasets has attracted increasing attention in recent years and has been exploited to improve the state-of-the-art in domains such as dataset subset search [2], [3], [6] and sampling for SGD [2], [4], [5], [7].

Chitta et al. [3] propose an ensemble active learning approach for dataset subset selection using ensemble uncertainty estimation. They also show that training classifiers on the subsets obtained in this way leads to more accurate models compared to training on the full dataset. Isola et al. [8] investigate the *memorability* of different examples based on the probability of each image being recognized (perceived as a repetition by the viewer) after a single view and train a predictor for image memorability based on image features. Memorability is also a familiar phenomenon to humans, as we can all think of images or visual memories that have stuck more in our minds compared to other images. Arpit et al. [9] define example difficulty as the average misclassification rate over a number of experiments.

Chang et al. [4] propose to prefer *uncertain* examples for SGD sampling, e.g., the examples that are neither consistently predicted correctly with high confidence nor incorrectly. They use two measures for "example uncertainty": (i) the variance of prediction probabilities and (ii) the estimated closeness between the prediction probabilities and the decision threshold. Yin et al. [7] observe that high similarity between concurrently processed gradients may lead to the speedup saturation and degradation of generalization performance for larger batch sizes and suggest that diversity-inducing training mechanisms can reduce training time and enable using larger batch sizes without the said side effects in distributed training.

Vodrahalli et al. [2] propose an importance measure for SGD sampling based on the gradient magnitude of the loss of each example at the end of training and use this measure to select a subset of the dataset for retraining. This measure can also be used to study the diversity of examples in datasets. Katharopoulos and Fleuret [5] propose an SGD sampling method that favors the more *informative* examples, which they describe as the examples that lead to the biggest changes in model parameters. Toneva et al. [6] propose *forgettability* as an importance measure for dataset examples. A forgettable example is an example that gets classified incorrectly at least once, after its first correct classification, over the course of training. They also suggest that the forgetting dynamics can be used to remove many examples from the base training dataset without hurting the generalization performance of the trained model.

Finally, in the domain of natural language processing, Swayamdipta et al. [10] have investigated the difference in example importance. They introduce *data maps* and calculate two measures for each example: the confidence of the model in the true class and the variability of the confidence across different epochs in a single training run. They then categorize the examples into three categories: *easy-to-learn*, *ambiguous*, and *hard-to-learn*.

3 Importance-aware DPT

Importance-aware DPT consists of three stages of model training, as shown in Figure 2. In the first stage, which we refer to as warmup training, we train the DNN using vanilla DPT for a number of "warmup" epochs (E_{warmup}) . Blocks (1) and (2) in Figure 2 show the first stage. In the second stage, we calculate the *importance* of each example according to a predefined importance measure, e.g., the average loss value of each example over E_{warmup} training epochs. In the third stage (blocks (3)-(5) in Figure 2), we continue training using *importance-aware* DPT in several *intervals*. Each interval consists of three steps: (i) dataset partitioning, i.e., assigning examples to partitions based on a *heuristic* and allocating one partition to each worker, (ii) model training, i.e., training the DNN using those fixed partitions for $E_{interval}$ epochs, and (iii) example importance calculation, in which we recalculate and update the importance value of each example for the next interval. In the rest of this section, we discuss importance-aware DPT in more detail.

3.1 Warmup Training

In the first stage, warmup training, the model is trained with vanilla DPT for E_{warmup} epochs, in which the dataset is randomly partitioned among the workers at the beginning of each epoch. We collect the value(s) needed for calculating the importance of examples during this stage. In this work, we use the loss value (the result of backpropagation forward pass) of each example in each epoch to calculate its importance value, which is the average loss over a number of epochs. It is worth noting that we will discard the loss values from the first E_{ignore} epochs in warmup training (e.g., the first three epochs), as the losses generated in the first few epochs are influenced by the random initialization of the neural network to a high degree.



Figure 2: An overview of Importance-aware data-parallel training. The model is first trained with Vanilla DPT for E_{warmup} epochs, after which the random dataset partitioning is replaced with heuristic-based dataset partitioning, and the dataset is partitioned at the beginning of each interval of training rather than at the beginning of each epoch.

2.4630	1.6089	 0.8972
0.9879	3.1874	 1.7276

Figure 3: example-epoch-loss matrix that is used to calculate the importance score of each example.

3.2 Importance Calculation

The second stage is a pause in model training, in which we calculate the importance of examples using values collected during warmup training. To demonstrate how this works, consider we calculate the importance of each example using "average loss across epochs". To do this, during warmup training, we collect the loss values (the result of the forward pass) of each example across E_{warmup} epochs. At the end of warmup training, we will have a matrix such as in Figure 3. In this matrix, each row corresponds to a single example, and each column corresponds to an epoch. Hence, an element $a_{i,j}$ in the matrix is the loss value of example i in epoch j. Calculating the importance of each example would then require a simple aggregation or computation over each row, e.g., a row-wise average. At the end of this stage, we have one or more scalar values attributed to each example, indicating its importance, which we use for sorting or categorizing the examples in the next stage (dataset partitioning).



Figure 4: Depiction of Stripes (left) and Blocks (right) partitioning heuristics for a setting with eight examples (indexed in order of importance) and four workers.

3.3 Dataset Partitioning Heuristics

Now that we have a mapping between examples and their importance values, we can use various heuristics to proceed with dataset partitioning for importance-aware DPT. Remind that in vanilla DPT, the examples are partitioned randomly across the parallel workers at the beginning of each epoch. We have defined two such heuristics, namely **Stripes** and **Blocks**, and compared them with random partitioning (i.e., vanilla DPT).

Stripes Heuristic.

The **Stripes** partitioning heuristic is a cyclic assignment of examples to workers. The intuition behind using this heuristic is to preserve the same distribution of examples with regard to their importance values, in each partition. To this end, we sort the examples of the dataset D by their importance value and create a list called *Sorted Examples* (*SE*). Then, the partition P_i that is allocated to worker w_i is determined as:

$$P_i = \{e \in D \mid sorted_index(e) \equiv i \pmod{n}\}$$

$$(2)$$

where $sorted_index(e)$ returns the index of example e in the sorted list SE, n is the number of workers, and i = 0, ..., n - 1. The **Stripes** heuristic is depicted on the left side of Figure 4.

Blocks Heuristic.

This partitioning heuristic assigns a continuous block of examples to each worker so that we will end up with different importance distributions across the workers. Assuming n workers, the **Blocks** heuristic allocates the first

 $\frac{|D|}{n}$ examples ranked in the SE list to the first worker, the second $\frac{|D|}{n}$ of SE to the second worker, and so on. Thus, the partition P_i that is allocated to worker w_i using the Blocks heuristic is determined as follows:

$$P_i = \{e \in D \mid i \times \frac{|D|}{n} \le sorted_index(e) < (i+1) \times \frac{|D|}{n}\}$$
(3)

where $sorted_index(e)$ returns the index of example e in the sorted list SE and i = 0, ..., n - 1. The Blocks heuristic is depicted on the right side of Figure 4.

3.4 Intervals of Model Training

After warmup training, calculating example importance, and partitioning the dataset based on the importance values, we continue model training using fixed partitions in intervals, each comprising of $E_{interval}$ epochs. At the beginning of each training interval, we repartition the dataset using the importance values calculated during the previous interval. This means that dataset repartitioning only occurs at the beginning of each interval rather than at the beginning of every epoch (as in vanilla DPT).

4 Implementation in PyTorch

This section presents the implementation details of importance-aware dataparallel training in PyTorch v1.10.1 [11], [12]. The implementation is mainly based on several classes and methods that (i) track and calculate the importance of examples as explained in Sections 3.1 and 3.2, (ii) partition the dataset across workers based on importance-aware heuristics defined in Section 3.3, and (iii) resume and continue the model training for fixed intervals of $E_{interval}$ epochs as described in Section 3.4.

4.1 Importance Calculation

Our proof-of-concept implementation of importance-aware DPT provides importance calculation for each example based on its average forward pass loss across a number of epochs. Loss function implementations in PyTorch, by default, do a batch-wise reduction on the losses and return a scalar aggregate value (e.g., the average loss of examples in the mini-batch when using CrossEntropyLoss¹). To get individual (per example) loss values, we construct an additional loss function of the same type and set its reduction

 $^{^1 \}rm As \ described \ in \ https://pytorch.org/docs/stable/generated/torch.nn.CrossEn tropyLoss.html$

parameter to None. This way, this loss function returns a tensor instead of a scalar.

Hence, each step of the training consists of two forward passes: the first one uses the customized loss function and writes values to a local worker copy of a loss-epochs matrix similar to the one depicted in Figure 3, and the second forward pass uses the default loss function implementation which is used with the backward pass. Each worker maintains its own copy of the loss-epochs matrix, and before each dataset partitioning step, the workers wait at a barrier (by calling torch.distributed.barrier()) for the main process to merge the local copies and aggregate, i.e., to compute the rowwise average which is the average loss of each example across the epochs. The output of this step is a sorted list of tuples (example, importance value) - the *Sorted Examples* list introduced in Section 3.3, that is used with the importance-aware partitioning heuristics.

4.2 Dataset Partitioning Heuristics

In PyTorch, the DistributedDataSampler class implements the logic for assigning examples to workers. By default, this class contains an implementation of random sampling, so we extend this class and add a sampler, called ConstantSampler, to arbitrarily assign the examples to workers. In this way, we decouple the implementation for assigning examples to workers, from the implementation of importance-aware partitioning heuristics. Hence, the same ConstantSampler can be used with different partitioning heuristics.

A dataset partitioning heuristics provides a mapping between examples and workers. We implement this mapping in PyTorch by creating a dictionary (dict) with worker indices as keys and a list of example indices as the value of each key. Depending on the heuristic, filling in this dictionary would then require iterating over the list of examples or workers. The result of this step, which is a dict that maps examples to workers, is used to construct a ConstantSampler instance that assigns the dataset examples across the workers. Once the ConstantSampler instance is constructed, the main process also reaches the barrier, so all the worker processes exit the barrier they had entered before merging their local matrices (as described in the previous section).

4.3 Modified Training Loop for Importance-aware Training

Model training in PyTorch typically consists of a few blocks of code for setting up the training (e.g., downloading the dataset, constructing the train/test/validation folds and data samplers, and creating the model), followed by a single loop for iterative training of the model. To implement different stages of importance-aware DPT, we first break down the default training loop into two separate loops: one for warmup training (Section 3.1) and the other for intervals of importance-aware training (Section 3.4). The first loop is similar to a typical PyTorch training loop but is extended with code to track and calculate the importance of examples. The second loop is nested: an outer loop maintains the intervals, while the inner loop contains the code for the actual dataset partitioning step, the example importance calculation step, and the model training step.

5 Evaluation

In this section, we describe our experimental setup and scenarios and discuss the results of the experiments. When talking about "model performance" we mainly refer to *best test accuracy* of a model trained for 100 epochs. Our hardware setup consists of a single machine with 4 GeForce RTX 2070 SUPER graphic cards, so we train on 4 workers.

5.1 Experimental Setup

To be able to empirically evaluate the effects of importance-aware dataset partitioning on the performance of DPT systems, we use two well-known DNN architectures for image classification: ResNet-18 and ResNet-34 [13] and train them on CIFAR-10 and CIFAR-100 datasets [14]. We use official PyTorch implementations of the models² and initialize them with random weights. In total, our experiments consist of 1830 training runs across 183 workloads (different combinations of datasets, models, partitioning heuristics, importance metrics, E_{warmup} , and $E_{interval}$). Three of these 183 workloads use vanilla DPT (ResNet-18 on CIFAR-10, ResNet-34 on CIFAR-10, and ResNet-34 on CIFAR-100), and we use them as baselines for comparison. For all runs that use importance-aware DPT, we set E_{ignore} to 5. We use the same hyperparameters for all runs of vanilla DPT and importanceaware DPT, i.e., SGD with a 0.9 Nesterov momentum and a learning rate starting at 0.1 and weight decay (L2 penalty) of 0.0005.

Considerations for Randomness: The training process of DNNs is a stochastic one and is affected by many factors, e.g., choice of hyperparameters, stochasticity in the optimization algorithms, and the stochastic behavior of the tools, frameworks, and hardware used for training [15]. To

²See https://pytorch.org/vision/main/models.html

Table 1: Average best test accuracies (over ten runs) and standard deviations for different combinations of E_{warmup} (W) and $E_{interval}$ (I), when training ResNet-18 on CIFAR-10 with **Stripes** policy and loss variance as the importance metric. The baseline (using vanilla DPT) is 82.983 ± 0.327 .

W	1	5	8	10	15	30
10	82.766 ± 0.185	82.848 ± 0.278	82.742 ± 0.152	82.862 ± 0.237	82.836 ± 0.387	82.988 ± 0.299
15	82.743 ± 0.373	82.752 ± 0.157	82.891 ± 0.302	82.888 ± 0.296	82.958 ± 0.247	82.873 ± 0.262
20	82.776 ± 0.243	82.832 ± 0.262	82.749 ± 0.309	82.722 ± 0.221	82.878 ± 0.283	83.044 ± 0.311
30	82.846 ± 0.202	82.858 ± 0.376	82.837 ± 0.263	82.946 ± 0.204	82.843 ± 0.307	82.773 ± 0.266
40	82.946 ± 0.246	82.773 ± 0.208	82.985 ± 0.238	82.869 ± 0.364	82.815 ± 0.296	82.827 ± 0.161
60	82.813 ± 0.283	82.898 ± 0.300	82.882 ± 0.152	82.764 ± 0.293	82.830 ± 0.249	82.705 ± 0.415

Table 2: Average best test accuracies (over ten runs) and standard deviations for different combinations of E_{warmup} (W) and $E_{interval}$ (I), when training ResNet-18 on CIFAR-10 with Stripes policy and average loss as the importance metric. The baseline (using vanilla DPT) is 82.983 ± 0.327 .

W	1	5	8	10	15	30
10	82.941 ± 0.262	$82.880 {\pm} 0.339$	82.859 ± 0.312	82.815 ± 0.290	82.836 ± 0.226	82.891±0.195
15	82.885 ± 0.231	$82.816 {\pm} 0.287$	82.841 ± 0.316	82.778 ± 0.259	82.866 ± 0.260	82.773±0.247
20	82.952 ± 0.314	82.913 ± 0.247	82.903 ± 0.240	82.889 ± 0.265	82.841 ± 0.278	82.919 ± 0.210
30	82.939 ± 0.294	82.854 ± 0.185	82.853 ± 0.236	82.889 ± 0.227	82.743 ± 0.335	82.929 ± 0.279
40	82.864 ± 0.138	82.903 ± 0.152	82.883 ± 0.225	82.766 ± 0.220	82.905 ± 0.244	82.851 ± 0.236
60	$82.908 {\pm} 0.337$	$82.931 {\pm} 0.339$	$82.818 {\pm} 0.245$	$82.956 {\pm} 0.228$	$82.806 {\pm} 0.195$	$82.758 {\pm} 0.237$

better control for this stochasticity, each of the 183 workloads is repeated ten times using ten pre-determined global random seeds. In Tables 1-5, we report the average best test accuracy and standard deviation of ten runs for each workload. Also, the box plot of the performance of the top five settings of each table, alongside the performance of the corresponding baseline (vanilla DPT), is shown in Figure 5.

5.2 Different Dataset Complexities

We consider workloads of (ResNet-34, Stripes, Variance) with each of the CIFAR-10 and CIFAR-100 datasets. The results of the runs can be seen in Tables 4-5, and in Figure 5 subfigures (4)-(5). CIFAR-10 and CIFAR-100 contain the same number of examples in train (50000 examples) and test (10000 examples) subsets, but they differ in the number of classes. CIFAR-10 has ten classes (5000 training examples per class), and CIFAR-100 has a higher complexity than CIFAR-10 in terms of classes.



(1) CIFAR-10, ResNet-18, Stripes, Variance



(3) CIFAR-10, ResNet-18, Blocks, Variance



(2) CIFAR-10, ResNet-18, Stripes, Average



(4) CIFAR-10, ResNet-34, Stripes, Variance



(5) CIFAR-100, ResNet-34, Stripes, Variance

Figure 5: Box plots comparing the performance of the top 5 settings of E_{warmup} (W) and $E_{interval}$ (INT) for different combinations of (Dataset, Model, Partitioning Heuristic, Importance Metric). The leftmost box plot in each subfigure is the performance of vanilla DPT (baseline), and the other five box plots are ordered in decreasing average best test accuracy. The white square on each box plot denotes the average best test accuracy for a setting. Each subfigure (1)-(5) corresponds to a table with the same number, which contains the average best test accuracies and standard deviations over ten runs for each of the combinations of W and INT.

Table 3: Average best test accuracies (over ten runs) and standard deviations for different combinations of E_{warmup} (W) and $E_{interval}$ (I), when training ResNet-18 on CIFAR-10 with Blocks policy and loss variance as the importance metric. The baseline (using vanilla DPT) is 82.983 ± 0.327 .

W	1	5	8	10	15	30
10	82.921 ± 0.352	83.067 ± 0.270	82.778 ± 0.426	82.743 ± 0.218	82.662 ± 0.240	82.706 ± 0.165
15	82.992 ± 0.321	82.899 ± 0.308	82.890 ± 0.253	82.805 ± 0.165	82.664 ± 0.178	82.109 ± 0.338
20	82.845 ± 0.292	82.939 ± 0.376	82.850 ± 0.429	82.716 ± 0.205	82.747 ± 0.289	82.523 ± 0.165
30	82.956 ± 0.189	82.942 ± 0.309	83.055 ± 0.153	82.954 ± 0.382	82.815 ± 0.247	82.583 ± 0.206
40	83.001 ± 0.270	82.861 ± 0.336	82.786 ± 0.247	82.925 ± 0.18	82.865 ± 0.177	82.894 ± 0.254
60	82.918 ± 0.348	82.873 ± 0.283	82.848 ± 0.271	82.886 ± 0.273	82.884 ± 0.228	82.462 ± 0.222

Table 4: Average best test accuracies (over ten runs) and standard deviations for different combinations of E_{warmup} (W) and $E_{interval}$ (I), when training ResNet-34 on CIFAR-10 with Stripes policy and loss variance as the importance metric. The baseline (using vanilla DPT) is 82.661 ± 0.478 .

W	1	5	8	10	15	30
10	82.650 ± 0.547	82.653 ± 0.399	82.590 ± 0.395	82.621 ± 0.243	82.751 ± 0.461	82.753±0.632
15	$82.537 {\pm} 0.332$	82.424 ± 0.510	82.745 ± 0.401	$82.799 {\pm} 0.481$	$82.832 {\pm} 0.239$	82.433 ± 1.020
20	82.845 ± 0.441	$82.659 {\pm} 0.637$	82.787 ± 0.407	82.606 ± 0.541	$82.890 {\pm} 0.321$	82.492 ± 0.300
30	82.671 ± 0.434	$82.539 {\pm} 0.307$	$82.719 {\pm} 0.509$	82.920 ± 0.287	$82.594 {\pm} 0.434$	82.720 ± 0.589
40	82.669 ± 0.426	82.773 ± 0.403	82.422 ± 0.728	$82.530 {\pm} 0.305$	82.649 ± 0.339	82.562 ± 0.353
60	$82.789 {\pm} 0.336$	82.615 ± 0.342	$82.683 {\pm} 0.397$	$82.768 {\pm} 0.525$	$82.678 {\pm} 0.451$	82.622 ± 0.661

The results show that there are several combinations of $(E_{warmup}, E_{interval})$ for training settings that can train better models than vanilla DPT. Thus, the gains of importance-aware DPT seem to hold across different datasets, given that we can find and select good hyperparameters for the training setting (e.g., E_{warmup} and $E_{interval}$).

5.3 Different Models

We consider workloads of (CIFAR-10, Stripes, Variance) with each of the ResNet-18 (18 layers, 8 residual blocks) and ResNet-34 (34 layers, 16 residual blocks) models [13]. The results of the runs can be seen in Tables 1 and 4, and in Figure 5 subfigures (1) and (4). There are combinations of $(E_{warmup}, E_{interval})$ corresponding to each model that train better models than their corresponding baselines, but ResNet-34 shows to gain more from importance-aware DPT than ResNet-18.

Table 5: Average best test accuracies (over ten runs) and standard deviations for different combinations of E_{warmup} (W) and $E_{interval}$ (I), when training ResNet-34 on CIFAR-100 with **Stripes** policy and loss variance as the importance metric. The baseline (using vanilla DPT) is 49.042 \pm 0.698.

W	I 1	5	8	10	15	30
10	49.169±0.33	5 49.064 ± 0.312	49.167 ± 0.432	48.758 ± 0.597	49.04 ± 0.503	49.033 ± 0.450
15	49.156 ± 0.33	$2 48.959 \pm 0.437$	49.264 ± 0.292	49.186 ± 0.498	49.073 ± 0.573	49.079 ± 0.351
20	48.978 ± 0.55	0 49.144 \pm 0.637	49.024 ± 0.365	49.149 ± 0.297	48.944 ± 0.436	48.977 ± 0.380
30	49.278 ± 0.39	9 48.906 ± 0.792	49.102 ± 0.393	48.897 ± 0.432	49.152 ± 0.446	48.966 ± 0.389
40	49.129 ± 0.54	9 48.978 ± 0.527	49.262 ± 0.489	49.155 ± 0.387	48.998 ± 0.450	49.024 ± 0.284
60	49.083 ± 0.34	8 49.224±0.338	49.027 ± 0.453	49.194 ± 0.396	49.107 ± 0.461	49.270 ± 0.429

5.4 Different Partitioning Heuristics

We consider workloads of (CIFAR-10, ResNet-18, Variance) with each of the Stripes and Blocks heuristics. The results of the runs can be seen in Tables 1 and 3, and in Figure 5 subfigures (1) and (3).

The results show that for both heuristics, there are combinations of $(E_{warmup}, E_{interval})$ that can train better models than vanilla DPT. It is particularly interesting that training using the Blocks heuristic shows comparable performance to training with both Stripes heuristic and vanilla DPT.

5.5 Different Importance Metrics

With the loss values generated by each example in forward passes across several epochs as our importance measure, we evaluate the effects of the choice of two different metrics: *average loss* and *loss variance*. We consider workloads of (CIFAR-10, ResNet-18, Stripes) with each of the above metrics. The results of the runs can be seen in Tables 1-2, and in Figure 5 subfigures (1)-(2). Loss variance as an importance metric performs marginally better than the average loss.

5.6 Added Overheads

The overheads of importance-aware DPT compared to vanilla DPT include (1) tracking importance data for each example at every epoch (a.k.a., importance tracking overhead) and (2) calculating the importance of examples and repartitioning the dataset based on heuristics at the beginning of each interval (a.k.a., heuristic overhead). In Table 6, we report the statistics on these overheads (in seconds) when we train ResNet-18 on CIFAR-10 for 100 epochs using four workers and the different 36 combinations of E_{warmup} and

Table 6: Overhead statistics (in seconds) of importance-aware DPT when training ResNet-18 on CIFAR-10 with the different 36 combinations of E_{warmup} and $E_{interval}$.

Quantity	Min	Average	Max
Importance tracking overhead (each epoch)	0.979	1.052	1.407
Heuristic overhead (each interval)	2.456	2.643	5.213
Total training time	715	721.556	758

 $E_{interval}$ (as reported in Tables 1-5). The importance tracking overhead is independent of E_{warmup} and $E_{interval}$, as it happens at every epoch, and on average accounts for 14.57% of the total wallclock time. However, we should note that this is a prototype implementation of importance-aware DPT, and many optimizations can be made to significantly reduce the overheads (e.g., getting the individual example losses and the mini-batch losses in the same forward pass or using MPI operations for calculating the importance of examples). By only requiring repartitioning at every $E_{interval}$, importance-aware DPT has the potential to significantly reduce the network and I/O overhead that vanilla DPT requires for fetching examples at each epoch, especially in large training settings consisting of hundreds of thousands or millions of examples.

6 Conclusion

In this paper, we proposed importance-aware DPT, a data-parallel training approach for deep neural networks, that partitions the dataset examples across the workers based on a notion of the importance of each example. Our empirical evaluation across a number of well-known image classification workloads suggests that by setting relevant values for the hyperparameters of this approach, most notably E_{warmup} and $E_{interval}$, we can find better models (in terms of best test accuracy) compared to when training with vanilla DPT. Future work can concentrate on, e.g., using hyperparameter tuning methods for finding the best values for the hyperparameters of importance-aware DPT and evaluating the effects of different importance metrics and measures.

Acknowledgement

This work has been supported by the ExtremeEarth project funded by European Union's Horizon 2020 Research and Innovation Programme under

Grant Agreement No. 825258. The authors would like to acknowledge funding from Vinnova for the Digital Cellulose Competence Center (DCC), Diary number 2016–05193. The computations for some of the experiments were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) at C3SE, partially funded by the Swedish Research Council through grant agreement no. 2022-06725 and no. 2018-05973. Artifacts are available in https://doi.org/10.5281/zenodo.785 5247 and https://github.com/ssheikholeslami/importance-aware-d ata-parallel-training.

References

- Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li, "Communicationefficient distributed deep learning: A comprehensive survey," arXiv preprint arXiv:2003.06307, 2020.
- [2] K. Vodrahalli, K. Li, and J. Malik, "Are all training examples created equal? an empirical study," arXiv preprint arXiv:1811.12569, 2018.
- [3] K. Chitta, J. M. Alvarez, E. Haussmann, and C. Farabet, "Training data distribution search with ensemble active learning," arXiv preprint arXiv:1905.12737, 2019.
- [4] H.-S. Chang, E. Learned-Miller, and A. McCallum, "Active bias: Training more accurate neural networks by emphasizing high variance samples," *Advances in Neural Information Processing Systems*, vol. 30, 2017.
- [5] A. Katharopoulos and F. Fleuret, "Not all samples are created equal: Deep learning with importance sampling," in *International Confer*ence on Machine Learning, PMLR, 2018, pp. 2525–2534.
- [6] M. Toneva, A. Sordoni, R. T. d. Combes, A. Trischler, Y. Bengio, and G. J. Gordon, "An empirical study of example forgetting during deep neural network learning," in *ICLR*, 2019.
- [7] D. Yin, A. Pananjady, M. Lam, D. Papailiopoulos, K. Ramchandran, and P. Bartlett, "Gradient diversity: A key ingredient for scalable distributed learning," in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2018, pp. 1998–2007.
- [8] P. Isola, J. Xiao, D. Parikh, A. Torralba, and A. Oliva, "What makes a photograph memorable?" *IEEE Transactions on Pattern Analysis* and Machine Intelligence, vol. 36, no. 7, pp. 1469–1482, 2013.

- [9] D. Arpit, S. Jastrzebski, N. Ballas, et al., "A closer look at memorization in deep networks," in *International Conference on Machine Learning*, PMLR, 2017, pp. 233–242.
- [10] S. Swayamdipta, R. Schwartz, N. Lourie, et al., "Dataset cartography: Mapping and diagnosing datasets with training dynamics," in Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2020, pp. 9275–9293.
- [11] S. Li, Y. Zhao, R. Varma, et al., "Pytorch distributed: Experiences on accelerating data parallel training," *Proceedings of the VLDB En*dowment, vol. 13, no. 12, 2020.
- [12] A. Paszke, S. Gross, F. Massa, et al., "Pytorch: An imperative style, high-performance deep learning library," Advances in neural information processing systems, vol. 32, 2019.
- [13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer* vision and pattern recognition, 2016, pp. 770–778.
- [14] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.
- [15] D. Zhuang, X. Zhang, S. Song, and S. Hooker, "Randomness in neural network training: Characterizing the impact of tooling," in *Proceedings* of Machine Learning and Systems, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, 2022, pp. 316–336.

Paper 5

Deep Neural Network Weight Initialization from Hyperparameter Tuning Trials

The 31st International Conference on Neural Information Processing (ICONIP), 2024

DEEP NEURAL NETWORK WEIGHT INITIALIZATION FROM HYPERPARAMETER TUNING TRIALS

Sina Sheikholeslami¹, Tianze Wang¹, Amir H. Payberah¹, Jim Dowling², and Vladimir Vlassov¹

¹ KTH Royal Institute of Technology, Stockholm, Sweden ² Hopsworks AB, Stockholm, Sweden

Abstract

Training of deep neural networks from scratch requires initialization of the neural network weights as a first step. Over the years, many policies and techniques for weight initialization have been proposed and widely used, including Kaiming initialization and different variants of random initialization. On the other hand, another requirement for starting the training stage is to choose and set suitable hyperparameter values, which are usually obtained by performing several hyperparameter tuning trials. In this paper, we study the suitability of weight initialization using weights obtained from different epochs of hyperparameter tuning trials and compare it to Kaiming uniform (random) weight initialization for image classification tasks. Based on an experimental evaluation using ResNet-18, ResNet-152, and InceptionV3 models, and CIFAR-10, CIFAR-100, Tiny ImageNet, and Food-101 datasets, we show that weight initialization from hyperparameter tuning trials can speed up the training of deep neural networks by up to 2x while maintaining or improving the best test accuracy of the trained models, when compared to random initialization.

1 Introduction

Training deep neural networks (DNNs) requires setting values for some parameters of the training process, e.g., learning rate, dropout rate, number of hidden layers, the amount of weight decay, and various settings of the model optimization algorithm, before starting the training stage. These parameters are referred to as hyperparameters (HPs), or meta-parameters for DNN model training, and their values are typically found during a stage called hyperparameter tuning (or hyperparameter optimization), where our goal

PAPER 5. DNN WEIGHT INITIALIZATION FROM HP TUNING

is to find the right combination for the value of different hyperparameters that maximize the prediction accuracy of the model.

The HP tuning stage is usually started by specifying a search space of hyperparameters, which includes the list of hyperparameters and the possible or allowed values each can take. Then, an HP tuning algorithm (or approach) is selected based on the downstream task and the computational and time constraints. The search process involves multiple trials, each training a model with a combination of hyperparameter values. Over the years, many hyperparameter tuning algorithms have been proposed [1], with grid search, random search [2], Bayesian optimization [3], and Asynchronous Successive Halving Algorithm (ASHA) [4] being the most popular. In the HP tuning stage, the training data is usually divided into (smaller) training and validation sets. Then, the model will be trained for a reduced number of epochs (usually a fraction of the number of epochs used for full model training) on the (smaller) training set using different combinations of hyperparameter values. The resulting performance from various combinations is then evaluated against the validation sets. At the end of this stage, the hyperparameters required for model training are chosen, and we proceed to the training stage.

Prior to the full model training stage, however, we need to perform one additional step, i.e., weight initialization [5], after deciding the values of hyperparameters. This involves setting appropriate initial values for the DNN model for effective training, e.g., avoiding issues like exploding and vanishing gradients [6]. Over the years, researchers and practitioners have proposed numerous approaches for weight initialization [5], including random initialization, Xavier (Glorot) initialization [7], and Kaiming (He) initialization [8]. Different initialization approaches use different heuristics and techniques to provide a better starting point for the model training stage, based on, e.g., information about the model, priors on the distribution of the dataset, etc. Nevertheless, to the best of our knowledge, there is a lack of weight initialization schemes that would use computations or results from the hyperparameter tuning stage, where the model from the best-performing trial has already "learned" some useful information from the dataset used for hyperparameter tuning, and can be potentially reused to provide a more appropriate starting point for the full model training.

Inspired by the idea of reusing computations or results of different stages of creating a deep learning (DL) system [9], in this paper we try to understand how weight initialization using the results of the hyperparameter tuning stage compares to the current best practices for weight initialization, in particular, the default weight initialization in PyTorch which is a variant of Kaiming uniform initialization for convolutional layers¹. Our driving motivation was to see if we could somehow "reuse" some of the computations of the hyperparameter tuning stage in the training stage.

Contributions. In this paper, we:

- propose a novel weight initialization approach that uses computation results (i.e., model weights) of the hyperparameter tuning stage to speed up and enhance the model training stage, and
- through an experimental evaluation consisting of 232 training runs, we show that for some combinations of models and datasets, weight initialization from hyperparameter tuning trials can outperform random initialization in terms of time-to-target-accuracy (which translates into speedup of the training stage by up to 2x), while achieving similar or even better best test accuracy.

2 Background

Weight Initialization. Neural network weight initialization is a very crucial and well-studied subject in the machine learning and deep learning literature [5], [6], [10], [11]. Weights are parameters that the neural network learns during the training process. Each weight represents the strength of connections between neurons, and the weights in a neural network determine how it will react in the output given a certain input. The primary goal of the training stage is to learn the appropriate values of all the weights from the data so that the neural network would perform well on the training data and generalize well on unseen test data.

Weight initialization is a crucial step in the training process of neural networks. The training process of neural networks is iterative by nature and is dominated by methods of stochastic gradient descent and its variants, and most neural networks are strongly affected by the choice of initialization [12]; hence, we need a suitable initial point for the training process where weights are initialized. The choice of the initial point can affect the speed of convergence, determine whether the neural network converges to a point with low and high cost, and sometimes, whether it converges at all. Common weight initialization techniques implemented and used in different deep learning software frameworks include variants of Xavier (Glorot) and Kaiming (He) initialization. In this work, we propose a new method that

¹The list of available initializations in PyTorch: https://pytorch.org/docs/stable/nn.init.html

uses the best weight point found during hyperparameter tuning trials as the initial point for the model training stage.

Throughout the years, several weight initialization approaches based on "pre-training" have been proposed [13]–[17]. Our approach differs from this body of work, particularly in that they all include various computations and calculations (e.g., using Autoencoders) on the pre-trained neural network, whereas we directly use the exact weights from the winning hyperparameter tuning trial. We refer readers to [5] for a recent review of weight initialization strategies and approaches.

Hyperparameter Tuning and Optimization. Apart from model weights (parameters) that determine the model itself, most machine learning models have settings that we usually refer to as hyperparameters, e.g., the number of hidden layers in the model, choice of the optimizer, learning rate, etc. Hyperparameters specify the details of the learning process but are not part of the result of training the model. Some hyperparameters, e.g., batch size and number of hidden layers, affect the time and memory cost of the training process. Other hyperparameters, e.g., choice of optimizer and floating point precision, affect the quality of the learned model after the training process.

One can choose between two basic approaches for selecting hyperparameters, i.e., manually or automatically. Choosing a hyperparameter manually requires domain knowledge and a deep understanding of the model, the downstream task, and the training process itself. Apart from that, choosing manually would often involve a tedious process of trial and error by trying out different potential values. Automated selection of hyperparameters, often referred to as hyperparameter tuning (or optimization), lifts the requirement of domain knowledge and requires less manual effort.

In a typical setting of HP tuning, we need to define a search space, a search strategy, and a computation budget. The search space specifies the ranges of values of the hyperparameters that we are interested in optimizing. A search strategy defines how we navigate through the search space to find the optimal set of values for the given hyperparameters. Finally, the computation budget limits the amount of time and computation used for the search. Common search strategies include grid search, random search, Bayesian optimization, Asynchronous Successive Halving (ASHA), etc. The computation budget can be specified in wall-clock time, number of epochs, number of evaluations, etc. We refer readers to [1], [18], [19] for a detailed review of hyperparameter tuning and optimization methods.

Meta-learning and Weight Initialization. A number of approaches have been proposed to learn "initializers" or "policies" that can suggest suitable initial parameters. Among these proposals, [20] introduce MetaInit, an algorithm that learns to suitably initialize the parameters of a given neural network for a given task. Our work is different from this line of research in that we do not attempt to "learn" any set of weights for the purpose of initialization; instead, we propose to "reuse" the model weights that are already learned during hyperparameter tuning trials as initial weights for the model training stage.

3 Methodology

Based on the idea of reusing results and computations of one stage of DL systems in another, we propose a novel weight initialization approach that uses weights from the top-performing or "winning" hyperparameter tuning trials to initialize the model weights for the model training stage. Generally, the performance of the HP tuning trial is measured using the same performance metric as the training task, e.g., validation (test) accuracy for classification tasks. The assumption is that for one trial in the hyperparameter tuning stage to be the winner, the weights in the model of that trial have already accomplished learning from the data to some degree compared to random initialization. Furthermore, it might even be potentially a decent set of weights as it is the winner among all the other trials allowed in the hyperparameter tuning budget.

Reusing the weights of the best hyperparameter tuning trial requires a minor modification to the typical hyperparameter tuning loop to save the weights of the model at certain epochs during the trials. Figure 1 and Algorithm 1 show the needed modification. This essentially means that in addition to suitable hyperparameter values, we also save (and later reuse) the model weights from the best hyperparameter tuning trial.

Algorithm 1 Hyperparameter Tuning with Weight Saving
Require: Total number of hyperparameter tuning trials N , number of
epochs per trial E
1: for $i = 1$ to N do
2: Choose a set of hyperparameter values for the trial
3: for $e = 1$ to E do
4: Perform a forward and backward pass on the model
5: Update the model weights
6: Save the model weights to storage as weights_i_e
7: end for
8: end for

PAPER 5. DNN WEIGHT INITIALIZATION FROM HP TUNING



Figure 1: The modified hyperparameter tuning experiment, where in addition to suitable hyperparameter values, we also use the weights from the winning hyperparameter tuning trial to initialize the model for the model training stage.

We can see in Algorithm 1 that our only modification in a typical hyperparameter tuning trial corresponds to line 6, in which we save the model weights to storage after each epoch. Based on our specific initialization policy, we do this to choose a set of initial weights from any given epoch for the training stage. As an example, in Figure 2, we have shown the best test accuracy of models initialized with different configurations. The numbers on the X axis, i.e., 2, 5, 10, 15, 20, and 25, correspond to different values for e when initializing the model with weights i e before starting the training stage, where the value of i corresponds to the index of the "winning" hyperparameter tuning trial (i.e., the trial with the best final validation accuracy). We should mention that for very large models, saving all the weights of different model variants might require considerable storage space and time to write to external storage. However, one can mitigate this by keeping track of the performance metric of top-T models (with T being a predefined constant number and only triggering the model save function when better models, e.g., with regards to test accuracy, are found).

4 Experimental Evaluation

To evaluate our weight initialization approach, we perform a number of experiments on different models and datasets and compare weight initialization from hyperparameter tuning to PyTorch's default initialization scheme, which uses a combination of various techniques, including Kaiming uniform [8] for convolutional and linear layers. We repeat the experiments several times as a way to control for randomness.

Our two main evaluation metrics are (i) best test accuracy, which is the

maximum (top-1) accuracy of the model on the test set during training, and (ii) time-to-target-accuracy (TTA), which is measured as the number of epochs it takes for the model to surpass a relatively high test accuracy as a milestone. The former metric indicates the model performance for a configuration, while the latter suggests the effect of a configuration on training time. To summarize, we want to know if our initialization approach can result in better models (higher test accuracy) while speeding up training (lower TTA). We choose the target accuracies based on our observations from the training curves of each model/dataset pair.

4.1 Experiment Setup

Hyperparameter Search Space. For the ResNet models, we use SGD with momentum (0.9) and follow a search space inspired by Zhang et al. [21] and common practice: a set of possible learning rate values of {0.01, 0.03, 0.05, 0.1, 0.2, 0.3} and a set of possible weight decay values of {0.0003, 0.001, 0.003}. Details about the hyperparameter tuning for the InceptionV3 model can be found in 4.6. We try a subset of the different combinations using random search. We use PyTorch [22] on Ray [23], [24] for hyperparameter tuning and parallel execution of trials while assigning a different random seed to each trial. For hyperparameter tuning of experiments that use the CIFAR-10, CIFAR-100, and Food-101 datasets, the training datasets provided by PyTorch were randomly partitioned into 80/20 train/validation splits. The corresponding subsections below explain more details on hyperparameter tuning for each task.

Weight Initialization Experiments. The goal of this set of experiments is to investigate if we can use the model weights obtained during the hyperparameter tuning to initialize the model before starting the training round. We select a number of common model/dataset combinations (e.g., ResNet-18 on CIFAR-10). For each combination, we tune a number of modelindependent hyperparameters (i.e., learning rate and weight decay) for a number of trials using random search, with no early stopping. Within each hyperparameter tuning trial, we save the model weights after every epochf. We then rank the trials in descending order of final validation accuracy and specify the winning hyperparameter tuning trial. Moving on to the model training stage, for weight initialization we use weights from several epochs of the winning trial. We then train the model for a number of epochs, and report the best test accuracy of each training trial. As the baseline for comparison, we use random initialization as implemented in PyTorch, which in particular uses Kaiming uniform initialization for convolutional layers.

PAPER 5. DNN WEIGHT INITIALIZATION FROM HP TUNING

Randomness Control and Reproducibility. When training deep neural networks, we should deal with many sources of randomness, including non-determinisms in hardware, software frameworks, and optimization algorithms and computations [25], [26]. This stochasticity can drastically influence the performance of models and make it hard for researchers to draw strong conclusions from experimental evaluations. To alleviate this and allow for reasonable reproducibility of our results, We use a predefined set of global random seeds, repeat each set of experiments several times, and report the averages and standard deviations for each set of results².

4.2 EXP1: ResNet-18 on CIFAR-10

Our first set of experiments deals with tuning and training ResNet-18 on the CIFAR-10 dataset³. The CIFAR-10 dataset contains $60000 \ 32 \times 32$ color images in 10 classes (6000 images per class). There are 50000 training images and 10000 test images.

Hyperparameter Tuning. For hyperparameter tuning, we used the same search space described in 4.1. We performed 10 tuning trials, with each trial consisting of 30 epochs of training with a batch size of 256. The winning trial had achieved a final validation accuracy of 88.48% with a learning rate of 0.01 and weight decay rate of 0.0003.

Training Configurations. Using the hyperparameters and weights from the winning hyperparameter tuning trial, we try 7 different weight initializations and compare them to a baseline in which we initialize the model weights using PyTorch's default scheme. These 7 different sets of weights are taken from epochs #2, #5, #10, #15, #20, #25, and the final epoch (HP Final, #30). Each configuration is trained for 200 epochs, and we repeat the training 8 times (with 8 different global seeds). Based on the training curves of the models we set the target to 90.00%. The results of this experiment are also presented in Table 1.

We can see that all the models that use weight initialization from the hyperparameter tuning stage achieve this milestone significantly faster than the baseline approach (random initialization), and most of them achieve a higher best test accuracy on average compared to the baseline. To verify the

²Code and raw results from our experiments can be found in https://github.com/s sheikholeslami/dnn-weight-initialization-from-hp-tuning.

³CIFAR-10 and CIFAR-100 datasets: https://www.cs.toronto.edu/~kriz/cifar.ht ml



Figure 2: Best test accuracy after 200 epochs of training ResNet-18 on CIFAR-10, using different weight initialization configurations. Numerical values are reported in Table 1.

significance of the results, we performed a paired t-test and Mann–Whitney U-test on results from "Epoch 25" and "Random" configurations, and the results were: t-statistic=3.521, p-value=0.00970; U1=3.0, p-value=0.00108, both indicating a significant difference in terms of best test accuracy. This specific configuration also shows an speedup of 2.027x in terms of TTA compared to random initialization.

4.3 EXP2: ResNet-18 on CIFAR-100

In this experiment, we tune and train ResNet-18 this time on the CIFAR-100 dataset. CIFAR-100 is similar to CIFAR-10 in terms of dimensions and total number of examples, but it has 100 classes containing 600 images each.

Hyperparameter Tuning. For hyperparameter tuning, we used the same search space described in 4.1. We performed 10 tuning trials, with each trial consisting of 40 epochs of training with a batch size of 256. The winning trial had achieved a final validation accuracy of 57.74% with a learning rate of 0.1 and weight decay rate of 0.0003.

Training Configurations. Using the hyperparameters and weights from the winning hyperparameter tuning trial, we try 6 different weight initializations and compare them to a baseline in which we initialize the model weights using PyTorch's default scheme. These 6 different sets of weights



Figure 3: First epoch to reach the target (90%) test accuracy when training ResNet-18 on CIFAR-10, using different weight initialization configurations.

are taken from epochs #5, #10, #15, #30, #35, and the final epoch (HP Final, #40). Each configuration is trained for 200 epochs, and we repeat the training 8 times (with 8 different global seeds). Based on the training curves of the models we set the target to 75.00%. The results of this experiment are presented in Figure 4 and Table 2.

4.4 EXP3: ResNet-18 on Tiny ImageNet

In this experiment, we tune and train ResNet-18 on the Tiny ImageNet⁴ dataset. Tiny ImageNet is a small-scale version of the larger ImageNet dataset, and contains 100000 downsized 64×64 color images in 200 classes as the training set, as well as 50 images for validation and 50 images for test in each class.

Hyperparameter Tuning. For hyperparameter tuning, we used the same search space described in 4.1. We performed 12 tuning trials, with each trial consisting of 40 epochs of training with a batch size of 256. The winning trial had achieved a final validation accuracy of 35.94% with a learning rate of 0.3 and a weight decay rate of 0.0003.

Training Configurations. Using the hyperparameters and weights from the winning hyperparameter tuning trial, we try 6 different weight initial-

⁴Accessible from https://image-net.org/download-images.php
Table 1: Results from the experiments on ResNet-18 and CIFAR-10. Each trial consists of training the model for 200 epochs. The experiments for each configuration (row) are repeated with 8 different random seeds, and the average values are reported. The TTA in this table indicates the first epoch in which the model achieves at least a 90.00% test accuracy.

Weight Initialization	Best Test Accuracy	TTA (#Epoch)
Epoch 2	$94.198 {\pm} 0.108$	24.0
Epoch 5	$94.289 {\pm} 0.107$	21.25
Epoch 10	$94.319 {\pm} 0.085$	20.75
Epoch 15	$94.339 {\pm} 0.166$	18.5
Epoch 20	$94.351 {\pm} 0.103$	14.625
Epoch 25	$94.391{\pm}0.061$	14.375
HP Final Epoch	$94.316{\pm}0.061$	12.75
Random (Kaiming + Uniform)	$94.231{\pm}0.098$	29.125

izations and compare them to a baseline in which we initialize the model weights using PyTorch's default scheme. These 6 different sets of weights are taken from epochs #5, #10, #15, #30, #35, and the final epoch (HP Final, #40). Each configuration is trained for 200 epochs, and we repeat the training 8 times (with 8 different global seeds). Based on the final accuracies and training curves, we set the target to 40.00%. The results of this experiment are presented in Figure 5 and Table 3.

4.5 EXP4: ResNet-152 on CIFAR-100

For this experiment, we change the model from ResNet-18 to ResNet-152, which is the larger variant of the ResNet family of models [27], and train it on the CIFAR-100 dataset. Tiny ImageNet is a small-scale version of the larger ImageNet dataset, and contains 100000 downsized 64×64 color images in 200 classes as the training set, as well as 50 images for validation and 50 images for test in each class.

Hyperparameter Tuning. For hyperparameter tuning, we used the same search space described in 4.1. We performed 10 tuning trials, with each trial consisting of 80 epochs of training with a batch size of 128. The winning trial had achieved a final validation accuracy of 51.13% with a learning rate of 0.01 and a weight decay rate of 0.003.

Training Configurations. Using the hyperparameters and weights from the winning hyperparameter tuning trial, we try three different weight initializations and compare them to a baseline in which we initialize the model



PAPER 5. DNN WEIGHT INITIALIZATION FROM HP TUNING

Figure 4: Best test accuracy after 200 epochs of training ResNet-18 on CIFAR-100, using different weight initialization configurations. Numerical values are reported in Table 2.

weights using PyTorch's default scheme. These three sets of weights are taken from epochs #65, #75 and the final epoch (HP Final, #80). Each configuration is trained for 200 epochs, and we repeat the training four times (with four different global seeds). Based on the final accuracies and training curves, we set the target to 75.00%. The results of this experiment are presented in Figure 6 and Table 4.

4.6 EXP5: InceptionV3 on Food-101

For this experiment, we use the InceptionV3 network [28], and train it on the Food-101 dataset [29]. The Food-101 dataset contains 101000 color images in 101 classes and 750 training and 250 test images per each class.

Hyperparameter Tuning. For hyperparameter tuning, we used the following search space: learning rate values sampled from a loguniform distribution between 0.0001 and 0.1, momentum from a uniform distribution between 0.8 and 0.99, weight decay from a loguniform distribution between 0.00001 and 0.001, and the maximum number of iterations (T_max) for the Cosine Annealing learning rate scheduler from the set of possible values of 50, 100, 200. We performed 12 tuning trials, with each trial consisting of 20 epochs of training with a batch size of 64. The winning trial had achieved a final validation accuracy of 40.50% with a learning rate of 0.021.

Table 2: Results from the experiments on ResNet-18 and CIFAR-100. Each trial consists of training the model for 200 epochs. The experiments for each configuration (row) have been repeated with 8 different random seeds and the average values are reported. The TTA in this table indicates the first epoch in which the model achieves at least a 75.00% test accuracy.

Weight Initialization	Best Test Accuracy	TTA (#Epoch)
Epoch 5	$77.044 {\pm} 0.191$	122.75
Epoch 10	$77.152 {\pm} 0.219$	123.25
Epoch 15	$77.112 {\pm} 0.138$	123.125
Epoch 30	$77.292{\pm}0.181$	124.5
Epoch 35	$77.124 {\pm} 0.287$	124.25
HP Final Epoch	$76.947{\pm}0.193$	124.25
Random (Kaiming + Uniform)	$77.080{\pm}0.299$	122.375

Table 3: Results from the experiments on ResNet-18 and Tiny ImageNet. Each trial consists of training the model for 200 epochs. The experiments for each configuration (row) are repeated with 8 different random seeds and the average values are reported. The TTA in this table indicates the first epoch in which the model achieves at least a 40.00% test accuracy.

Weight Initialization	Best Test Accuracy	TTA (#Epoch)
Epoch 5	$44.062 {\pm} 0.536$	153.75
Epoch 10	$44.085 {\pm} 0.403$	153.25
Epoch 15	$43.885 {\pm} 0.245$	153.5
Epoch 30	$44.328 {\pm} 0.479$	153.875
Epoch 35	$44.362 {\pm} 0.415$	154.125
HP Final Epoch	44.223 ± 0.275	154.25
Random (Kaiming + Uniform)	$44.428 {\pm} 0.485$	154.5

Training Configurations. Using the hyperparameters and weights from the winning hyperparameter tuning trial, we try four different weight initializations and compare them to a baseline in which we initialize the model weights using PyTorch's default scheme. These four sets of weights are taken from epochs #5, #10, #15, and the final epoch (HP Final, #20). Each configuration is trained for 50 epochs, and we repeat the training for each configuration eight times (with eight different global seeds). Based on the final accuracies and training curves, we set the target to 70.00%. The results of this experiment are presented in Figure 7, Figure 8, and Table 5. Based on the results, we can see that for this combination of model and dataset, weight initialization from hyperparameter tuning trials clearly out-





Figure 5: Best test accuracy after 200 epochs of training ResNet-18 on Tiny ImageNet, using different weight initialization configurations. Numerical values are reported in Table 3.

performs random initialization in terms of both TTA and best test accuracy.

4.7 Further Discussion

Summary of Findings. Overall, our experimental evaluation shows that for some models and datasets, e.g., ResNet-18 on CIFAR-10 or CIFAR-100, and InceptionV3 on Food-101, weight initialization using hyperparameter tuning trials can outperform random initialization in terms of timeto-target-accuracy, as well as best test accuracy. It is also interesting to note that using weights from a later epoch of the hyperparameter tuning trial does not necessarily result in a better test accuracy; e.g., we can see in Figure 2 that initializing the model with weights from epoch #25 result in the best performance, and better than HP Final.

For ResNet-18 on Tiny ImageNet, and ResNet-152 on CIFAR-100, random initialization achieves a higher best test accuracy, but other initializations can achieve better TTAs. One possible reason for this difference in performance is that for ResNet-18 on Tiny ImageNet, and ResNet-152 on CIFAR-100, our underlying training regimes (irrespective of weight initialization policy) do not result in a test accuracy in the so-called stateof-the-art region, which is not the case for ResNet-18 on CIFAR-10 and CIFAR-100. This can be investigated using further, more thorough experiments; however, we believe our findings from these experiments are interesting enough to motivate further research on this topic.



Figure 6: Best test accuracy after 200 epochs of training ResNet-152 on CIFAR-100, using different weight initialization configurations. Numerical values are reported in Table 4.

Storage Requirements. Our modified hyperparameter tuning algorithm, in its general form, as specified in Algorithm 1, requires that we save the model weights to storage after each epoch, hence using storage space. When tuning ResNet-152 on CIFAR-100, the size of each set of weights is 224 Megabytes (MB), so a complete hyperparameter tuning experiment in which we run 10 trials with 80 epochs each would require 179.2 Gigabytes (GB) of storage to save all the weights. Saving the weights after each epoch can become impractical or lead to a bottleneck when tuning considerably larger models; however, one can always modify the algorithm so that only weights from specific epochs are saved, e.g., the weights from the final epoch.

Table 4: Results from the experiments on ResNet-152 and CIFAR-100. Each trial consists of training the model for 200 epochs. The experiments for each configuration (row) are repeated with 4 different random seeds and the average values are reported. The TTA in this table indicates the first epoch in which the model achieves at least a 75.00% test accuracy.

Weight Initialization	Best Test Accuracy	TTA (#Epoch)
Epoch 65	$80.23 {\pm} 0.189$	144.0
Epoch 75	$80.168 {\pm} 0.107$	143.25
HP Final Epoch	$80.072 {\pm} 0.285$	142.5
Random (Kaiming $+$ Uniform)	$80.372 {\pm} 0.169$	143.25

Table 5: Results from the experiments on InceptionV3 and Food-101. Each trial consists of training the model for 50 epochs. The experiments for each configuration (row) are repeated with 8 different random seeds and the average values are reported. The TTA in this table indicates the first epoch in which the model achieves at least a 70.00% test accuracy.

Weight Initialization	Best Test Accuracy	TTA (#Epoch)
Epoch 5	$74.93{\pm}0.115$	28.75
Epoch 10	$75.935 {\pm} 0.23$	23.25
Epoch 15	$76.572 {\pm} 0.155$	20.5
HP Final Epoch	$76.894{\pm}0.158$	18.375
Random (Kaiming $+$ Uniform)	$73.385{\pm}1.073$	33.875



Figure 7: Best test accuracy after 50 epochs of training InceptionV3 on Food-101, using different weight initialization configurations. Numerical values are reported in Table 5. Training with each configuration was repeated 8 times.



Figure 8: First epoch to reach the target (70%) test accuracy when training InceptionV3 on Food-101, using different weight initialization configurations.

5 Conclusion

In this paper, we proposed a novel weight initialization approach that uses computation results (i.e., model weights) of the hyperparameter tuning stage to speed up and enhance the model training stage and evaluated its performance through a number of experiments on common models and datasets in the image classification domain. Our main research question was to understand how to effectively perform weight initialization using the weights from the hyperparameter tuning stage and how it compares to the current best practices for weight initialization, in particular, the default weight initialization in PyTorch. The results of our experiments using ResNet-18 and ResNet-152 models and CIFAR-10, CIFAR-100, and Tiny ImageNet datasets show that for some combinations of models and datasets, weight initialization in terms of time-to-target-accuracy, while maintaining or improving the best test accuracy of the learned model.

Our work serves as a starting point in the research direction of reusing computation results in hyperparameter tuning of DNN models to speed up the training process after hyperparameter tuning. Based on our empirical study, we envision that further investigation of our approach can result in weight initialization approaches that, in turn, lead to possibly faster and more efficient training pipelines that also train better models. Future work includes expanding our studies in both theoretical and empirical dimensions to further understand how to effectively reuse computation results from hyperparameter tuning. To this end, performing more experiments using more types of DNN models, datasets, and downstream tasks in different domains can be an interesting starting point.

References

- M. Feurer and F. Hutter, "Hyperparameter optimization," Automated machine learning: Methods, systems, challenges, pp. 3–33, 2019.
- [2] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization.," Journal of machine learning research, vol. 13, no. 2, 2012.
- [3] J. Wu, X.-Y. Chen, H. Zhang, L.-D. Xiong, H. Lei, and S.-H. Deng, "Hyperparameter optimization for machine learning models based on bayesian optimization," *Journal of Electronic Science and Technology*, vol. 17, no. 1, pp. 26–40, 2019.
- [4] L. Li, K. Jamieson, A. Rostamizadeh, et al., "A system for massively parallel hyperparameter tuning," *Proceedings of Machine Learning* and Systems, vol. 2, pp. 230–246, 2020.

- [5] M. V. Narkhede, P. P. Bartakke, and M. S. Sutaone, "A review on weight initialization strategies for neural networks," *Artificial intelli*gence review, vol. 55, no. 1, pp. 291–322, 2022.
- [6] D. Arpit, V. Campos, and Y. Bengio, "How to initialize your network? robust initialization for weightnorm & resnets," Advances in Neural Information Processing Systems, vol. 32, 2019.
- [7] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.
- [8] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.
- [9] L. Li, E. Sparks, K. Jamieson, and A. Talwalkar, "Exploiting reuse in pipeline-aware hyperparameter tuning," arXiv preprint arXiv:1903.05176, 2019.
- [10] G. Thimm and E. Fiesler, "Neural network initialization," in From Natural to Artificial Neural Computation: International Workshop on Artificial Neural Networks Malaga-Torremolinos, Spain, June 7–9, 1995 Proceedings 3, Springer, 1995, pp. 535–542.
- [11] N. Weymaere and J.-P. Martens, "On the initialization and optimization of multilayer perceptrons," *IEEE Transactions on Neural Net*works, vol. 5, no. 5, pp. 738–751, 1994.
- [12] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.
- [13] G. Li, H. Alnuweiri, Y. Wu, and H. Li, "Acceleration of back propagation through initial weight pre-training with delta rule," in *IEEE International Conference on neural networks*, IEEE, 1993, pp. 580– 585.
- [14] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layerwise training of deep networks," Advances in neural information processing systems, vol. 19, 2006.
- [15] T. Le Paine, P. Khorrami, W. Han, and T. S. Huang, "An analysis of unsupervised pre-training in light of recent advances," in 3rd International Conference on Learning Representations, ICLR 2015, 2015.
- [16] T. H. Trinh, M.-T. Luong, and Q. V. Le, "Selfie: Self-supervised pretraining for image embedding," arXiv preprint arXiv:1906.02940, 2019.

- [17] A. Ruiz-Garcia, M. Elshaw, A. Altahhan, and V. Palade, "Stacked deep convolutional auto-encoders for emotion recognition from facial expressions," in 2017 International Joint Conference on Neural Networks (IJCNN), IEEE, 2017, pp. 1586–1593.
- [18] T. Yu and H. Zhu, "Hyper-parameter optimization: A review of algorithms and applications," arXiv preprint arXiv:2003.05689, 2020.
- [19] B. Bischl, M. Binder, M. Lang, et al., "Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges," Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 13, no. 2, e1484, 2023.
- [20] Y. N. Dauphin and S. Schoenholz, "Metainit: Initializing learning by learning to initialize," Advances in Neural Information Processing Systems, vol. 32, 2019.
- [21] M. Zhang, J. Lucas, J. Ba, and G. E. Hinton, "Lookahead optimizer: K steps forward, 1 step back," Advances in neural information processing systems, vol. 32, 2019.
- [22] A. Paszke, S. Gross, F. Massa, et al., "Pytorch: An imperative style, high-performance deep learning library," Advances in neural information processing systems, vol. 32, 2019.
- [23] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," arXiv preprint arXiv:1807.05118, 2018.
- [24] P. Moritz, R. Nishihara, S. Wang, et al., "Ray: A distributed framework for emerging {ai} applications," in 13th USENIX symposium on operating systems design and implementation (OSDI 18), 2018, pp. 561–577.
- [25] D. Zhuang, X. Zhang, S. Song, and S. Hooker, "Randomness in neural network training: Characterizing the impact of tooling," *Proceedings* of Machine Learning and Systems, vol. 4, pp. 316–336, 2022.
- [26] D. Picard, "Torch. manual_seed (3407) is all you need: On the influence of random seeds in deep learning architectures for computer vision," arXiv preprint arXiv:2109.08203, 2021.
- [27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer* vision and pattern recognition, 2016, pp. 770–778.
- [28] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of* the *IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[29] L. Bossard, M. Guillaumin, and L. Van Gool, "Food-101 – mining discriminative components with random forests," in *Computer Vision* – *ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds., Cham: Springer International Publishing, 2014, pp. 446–461, ISBN: 978-3-319-10599-4.

Paper 6

Utilizing Large Language Models for Ablation Studies in Machine Learning and Deep Learning

The 5th Workshop on Machine Learning and Systems (EuroMLSys), co-located with the 20th European Conference on Computer Systems (EuroSys), 2025

UTILIZING LARGE LANGUAGE MODELS FOR Ablation Studies in Machine Learning and Deep Learning

Sina Sheikholeslami¹, Hamid Ghasemirahni¹, Amir H. Payberah¹, Tianze Wang¹, Jim Dowling², and Vladimir Vlassov¹

¹ KTH Royal Institute of Technology, Stockholm, Sweden ² Hopsworks AB, Stockholm, Sweden

Abstract

In Machine Learning (ML) and Deep Learning (DL) research, ablation studies are typically performed to provide insights into the individual contribution of different building blocks and components of an ML/DL system (e.g., a deep neural network), as well as to justify that certain additions or modifications to an existing ML/DL system can result in the proposed improved performance. Although dedicated frameworks for performing ablation studies have been introduced in recent years, conducting such experiments is still associated with requiring tedious, redundant work, typically involving maintaining redundant and mostly-identical versions of code that correspond to different ablation trials. Inspired by the recent promising performance of Large Language Models (LLMs) in the generation and analysis of ML/DL code, in this paper we discuss the potential of LLMs as facilitators of ablation study experiments for scientific research projects that involve or deal with ML and DL models. We first discuss the different ways in which LLMs can be utilized for ablation studies and then present the prototype of a tool called ABLATIONMAGE, that leverages LLMs to semi-automate the overall process of conducting ablation study experiments. We showcase the usability of ABLATIONMAGE as a tool through three experiments, including one in which we reproduce the ablation studies from a recently published applied DL paper.

1 Introduction

Deep neural networks (DNNs) are becoming increasingly larger in size and complexity. In just a decade, the size of practical and popular networks has grown from around 62 million parameters (AlexNet [1]) to hundreds of billions of parameters (e.g., Megatron-Turing NLG [2], PaLM [3], Falcon [4], Vision Transformer (ViT) [5], and Llama 3 [6]). Meanwhile, through the introduction of new architectures and training approaches, different modules and mechanisms for constructing and training DNNs have become widely adapted (e.g., inception modules [7], residual connections [8], and multihead attention [9]).

A common practice in Machine Learning (ML) and Deep Learning (DL) research and creating ML/DL systems is to start with an established network architecture (e.g., a Transformer [9]) or training method (e.g., Adam optimizer [10]) a "baseline" and try to improve that architecture or training method with regard to specific downstream tasks. The new architecture or method will then become different from the baseline in terms of the number of "components" that may have been added to or removed from the baseline. After that, to evaluate the new architecture or method, its performance on a number of benchmarks will be compared to that of the baseline as well as the state-of-the-art.

While such experimental results can be enough to determine the performance of the new architecture/method "as a whole" compared to the baseline, they may not provide information about the contribution of the different components of the architecture/method to its performance. To that end, a simple technique is to perform a systematic experiment known as ablation study [11], [12]. Essentially, in an ablation study, the changed components are added/removed from the baseline one by one or in groups, and the performance of each of these different "configurations" is then compared to the baseline as well as the final architecture/method. This way, one can reason about the individual contribution of the different added, removed, or modified components to the overall performance.

However, while performing an ablation study seems to be a straightforward task, it is still missing from many of the scientific publications in the various fields that deal with ML/DL. This is partly due to its added cost and manual effort, as it requires performing extra experiments and maintaining multiple versions of the code required for defining and training the different configurations. Although in recent years, dedicated frameworks for ablation studies in ML/DL have been introduced to address these challenges [12], [13], many practitioners still choose to either perform ablation studies manually (e.g., to avoid adding another framework or library to their project) or forgo them entirely.

Meanwhile, a recent trend in the ML/DL research community is to use Large Language Models (LLMs) and different prompting techniques to provide novel solutions or to enhance the existing solutions for various downstream tasks [14]–[16]. In particular, LLMs have shown promising results in neural architecture search [17], [18] and hyperparameter optimization [19]–



Figure 1: Overview of AblationMage.

[21]. Considering that the training data for the leading publicly available LLMs includes a large corpus of codes and documents related to DNN training available on the web, LLMs have shown good potential in generating sound and relevant code for different ML/DL stages and tasks [22].

Contributions: Motivated by these efforts, we (i) investigate how LLMs can be leveraged for performing ablation experiments, specifically by helping in the design of the ablation study, generating correct and coherent artifacts for performing the ablation trials, and analyzing and presenting the results. We then (ii) introduce ABLATIONMAGE, a tool that leverages an LLM to semi-automate the process of conducting ablation studies in ML/DL. Finally, we (iii) evaluate the usability of ABLATIONMAGE using three examples related to common scenarios related to ablation studies. To the best of our knowledge, ABLATIONMAGE is the first LLM-based dedicated tool for automating and conducting ablation study experiments.

2 AblationMage

We now discuss the prototype implementation of ABLATIONMAGE, a reusable and customizable tool for ablation experiments¹. An overview of ABLA-TIONMAGE is shown in Figure 1. ABLATIONMAGE receives the original or annotated source code of the system alongside any other documentation, such as dataset properties or the related research paper, as the input. It can then provide suggestions for annotations related to trials (if the code does not have any annotations), and the user can modify and verify them. The annotated code is then analyzed to find and extract the associated code snippets. Then, a full prompt is generated by ABLATIONMAGE, and

 $^{^1{\}rm The}$ source code of AblationMage and results from the experiments will be released upon acceptance of the paper.

an LLM will be queried. We refer to this as the first call.

The response of the LLM will include the executable source code(s), instructions for conducting the trials, description of added trials, and any code added for analysis and presentation. Once the user executes the provided code(s), in case there are any errors or issues related to the correctness, they can use ABLATIONMAGE to perform a *follow-up call*, which prompts the LLM to fix the specific issues. The user can repeat this process until desired artifacts are created. Example prompts and LLM responses for each of these types of calls (first or follow-up call) can be found in the Appendix.

Currently, ABLATIONMAGE supports two types of annotations: *explicit* annotations and hint annotations. Explicit annotations require the user to explicitly annotate the lines of code that may correspond to ablation trials. This is particularly useful for model/layer ablation trials, and the users can add "#ABLATABLE_COMPONENT" as comments to the lines of code that correspond to adding different layers to a model (see Listing 1 as an example). The users can also provide hint annotations, which are natural language descriptions of the desired ablation trials that are added in the form of comment blocks before related parts of the code (e.g., one might place a hint annotation before the definition of a function that creates a model, as shown in Listing 3).

Depending on how sophisticated the original implementation (base code) is, the LLM might still be able to "understand" the logic of the base code and help in both the design and implementation of the ablation trials without any explicit annotations or correct placement of hint annotations; however, the LLMs seem to benefit from documentation and code comments in code understanding [23].

ABLATIONMAGE is implemented in Python and uses HuggingFace's Chat Templates to communicate with LLMs. The current version supports the OpenAI and Anthropic APIs, but it can be easily extended to support other standard LLM APIs. Next, we will evaluate the usability of ABLA-TIONMAGE while using Claude Sonnet 3.5 as the LLM backend.

3 AblationMage

We now discuss the prototype implementation of ABLATIONMAGE, a reusable and customizable tool for ablation experiments. An overview of ABLATION-MAGE is shown in Figure 1. ABLATIONMAGE receives the original or annotated source code of the system alongside any other documentation, such as dataset properties or the related research paper, as the input. It can then provide suggestions for annotations related to trials (if the code does not have any annotations), and the user can modify and verify them. The annotated code is then analyzed to find and extract the associated code snippets. Then, a full prompt is generated, and an LLM backend will be queried in a *first call*.

The response of the LLM will include the executable source code(s), instructions for conducting the trials, description of added trials, and any code added for analysis and presentation. Once the user executes the provided code(s), in case there are any errors or issues related to the correctness, they can use ABLATIONMAGE to perform a *follow-up call*, which prompts the LLM to fix the specific issues. The user can repeat this process until desired artifacts are created. Example prompts and LLM responses for each of these types of calls can be found in the Appendix.

Currently, ABLATIONMAGE supports two types of annotations: *explicit* annotations and hint annotations. Explicit annotations require the user to explicitly annotate the lines of code that may correspond to ablation trials. This is particularly useful for model/layer ablation trials, and the users can add "#ABLATABLE_COMPONENT" as comments to the lines of code that correspond to adding different layers to a model (see Listing 1 as an example). The users can also provide hint annotations, which are natural language descriptions of the desired ablation trials that are added in the form of comment blocks before related parts of the code (e.g., one might place a hint annotation before the definition of a function that creates a model, as shown in Listing 3).

Depending on how sophisticated the original implementation (base code) is, the LLM might still be able to "understand" the logic of the base code and help in both the design and implementation of the ablation trials without any explicit annotations or correct placement of hint annotations; however, the LLMs seem to benefit from documentation and code comments in code understanding [23].

ABLATIONMAGE is implemented in Python and uses HuggingFace's Chat Templates to communicate with LLMs. The current version supports the OpenAI and Anthropic APIs, but it can be easily extended to support other standard LLM APIs. Next, we will evaluate the usability of ABLA-TIONMAGE while using Claude Sonnet 3.5 as the LLM backend.

4 Evaluation & Discussion

To evaluate the usability of our prototype implementation of ABLATION-MAGE, we conduct three experiments: in the first experiment, we use AB-LATIONMAGE to generate code for an ablation study of a few layers of a Convolutional Neural Network (CNN) trained on the CIFAR-10 dataset, where we annotate the lines of code that correspond to defining the layers of the CNN with #ABLATABLE_COMPONENT comments. In the second experiment, we try to reproduce the ablation studies mentioned in a recently

Listing 1 Annotating specific lines of the original code to specify ablation trials (explicit annotation).

```
class CIFAR1OCNN(nn.Module):
    def __init__(self):
        supr(CIFAR1OCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1) #ABLATABLE_CONPONENT
        self.pool = nn.MaxPool2d(2, 2)
        self.fcl = nn.Linear(128 * 4 * 4, 512) #ABLATABLE_CONPONENT
        self.fc2 = nn.Linear(512, 10)
        self.dropout = nn.Dropout(0.5)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
...
```

Listing 2 First call to an LLM using ABLATIONMAGE.

python ablationmage.py first-call -a anthropic -m claude-3-5-sonnet-20241022

published paper [24]. In the aforementioned paper, the authors discuss an ablation study in their manuscript and have made their code repository publicly available. However, the said repository does not contain the code related to their ablation studies. We are interested to see if we can use ABLATIONMAGE to generate the code to perform the ablation studies given the original implementation by the authors. Finally, in the third experiment, we use ABLATIONMAGE for a feature ablation study of the Higgs Boson Machine Learning Challenge dataset [25] when used for training an XGBoost [26] classifier.

4.1 Experiment 1: Layer Ablation of a CNN

For this experiment, we start with a typical PyTorch code that trains a simple CNN on the CIFAR-10 dataset. We are interested in performing a *layer ablation* experiment, in which the goal is to study the relative contribution of a convolutional layer (conv2) as well as a fully connected layer (fc1). To this end, we annotate the model creation code lines corresponding to those layers with #ABLATABLE_COMPONENT comments. Listing 1 shows the annotated code of the model creation function.

The next step is to pass this annotated code to ABLATIONMAGE, and wait for it to initiate an API Call to an LLM and return the results (this corresponds to the *first call* as discussed in the previous section). Listing 1 shows how to initiate a first call. The user can expect to receive the response after a few seconds. The response is returned in the form of a text file containing both the full versions of the modified source file(s), as well as instructions on how to run the ablation study, an overview of the changes made to the original source code, as well as additional notes, e.g., regarding the plots or how to interpret possible results.

We take the portion of the result that corresponds to the full source code for the ablation study to replace the original source code, and we run it. The baseline trial executes successfully, but the script fails when trying to execute the first layer ablation trial. Upon looking at the error logs and the stack trace, we realize this is due to the fact that when the layer is removed, the input of its following layer should be modified to match the output of the preceding layer. We paste the relevant parts of the stack trace in a file called output_result.txt and then use ABLATIONMAGE to initiate a follow-up call.

The follow-up call is made similarly to the first call, but instead of the original source file(s), it requires the path to the new source file(s) - which we just executed - as well as the path to output_result.txt. The response to the follow-up call also comes in a similar format to the response to the first call. Upon replacing the source code(s) with the new version of the code and attempting the execution, the full ablation study experiment runs without a problem. The output of running the study includes several plots that can be used to compare the baseline and the different ablation trials, as well as a JSON file that includes the raw results of all the trials. We hence conclude that ABLATIONMAGE can be used as an aid in conducting ablation study experiments.

4.2 Experiment 2: Reproducing the Ablation Studies of a Research Paper

We now consider a different scenario: conducting an ablation study on a paper or code authored by other researchers. In cases where authors include an ablation study in their paper, they may or may not provide the code needed to reproduce it. We aim to evaluate whether ABLATIONMAGE can assist in such situations, specifically when an original implementation is available for a research paper that discusses an ablation study, but the code to reproduce the study is unavailable.

We take [24] as a relevant example of this case among the papers we had recently read. In the aforementioned paper, the authors propose an approach using Graph Representation Learning to automate the triage of emergency patients, i.e., to classify each patient's emergency treatment condition. As part of their system, they use a model based on GraphSAGE [27] as one of the possible classifiers. Since the model based on GraphSAGE showed the best performance among the different classifiers, they performed an ablation study on the relative contribution of the different layers of the model to the classification accuracy on the test set. They report their results Listing 3 Example of a hint annotation for model ablation.

```
#ABLATION_HINT_START
# The ablation study should consist of the following layer ablation trials on GraphSAGE:
# - the second layer removed
# - the third layer removed
# - the fourth layer removed
# - the second, third, and fourth layers removed, while the number of output neurons of the first layer is 8
# - the second, third, and fourth layers removed, while the number of output neurons of the first layer is 64
# ABLATION_HINT_END
class GraphSAGE(torch.nn.Module):
    def __init__(self, dim_in, dim_out):
...
```

and discuss the outcome of the different trials; however, their implementation on their publicly available repository does not contain the code for conducting the ablation study.

We add a description of the desired ablation trials in the form of a *hint* annotation and add it before the place in the code where the model is defined. Listing 3 shows the annotation we added to the original source code. Upon providing the source code to ABLATIONMAGE, the response from the first call got the correct implementation of different trials, but it missed an import of a required module. This required a simple, one-liner fix and could be done by the user, but nevertheless, we made a follow-up call with ABLA-TIONMAGE, and the output contained the missing import statement. The main part of the code was correct, and the trials were executed successfully, but there was an error related to the results visualization code that was added by the LLM. Again, this was something that could be fixed easily, but we used another follow-up call, and the final code was executed without any errors. We were able to replicate the study as described by the authors and witness mostly similar results. We further confirmed the correctness of the generated code by comparing it with the ablation study code provided by the authors of the original article upon our request.

4.3 Experiment 3: Feature Ablation of the Higgs Boson Challenge Dataset

We now look at another component of an ML/DL system that may benefit from ablation study experiments: the dataset used for training the model. Depending on the type and modality of the data, the examples within a dataset can have several ablatable dimensions (and in a multimodal datasets, modalities themselves can be ablated), but perhaps the most typical type of data ablation is *feature ablation*. In a feature ablation study, we remove individual (or groups of) features (e.g., channels in an image, or columns in a tabular dataset) from the training dataset, train the

Listing 4 Example of a hint annotation for feature ablation.

```
def main():
```

```
data = pd.read_csv('training.csv')
#ABLATION_HINT_START
# The ablation study should consist of the following feature ablation trials:
# - remove 'DER_mass_transverse_met_lep'
# - remove 'DER_mass_vis'
# - remove 'DER_pt_h'
# - remove 'DER_dilatei_jet_jet'
# - remove 'DER_dilatei_jet_jet'
# - remove 'DER_mass_jt_jet'
# - remove pairs of the above features, one pair at a time
#ABLATION_HINT_END
data['Label'] = data['Label'].map({'s': 1, 'b': 0})
```

same model on each variation of the dataset, and examine the difference in the performance of the model. This simple examination can provide us with useful information on the importance of different dataset features.

To demonstrate the capability of ABLATIONMAGE for data ablation studies, we perform a feature ablation study on the Higgs Boson ML Challenge Dataset [25], a well-known tabular dataset. The training data consists of 250000 particle collision events, and we want to train an XGBoost classifier to determine whether a collision event is a signal, or background noise. Each event has an ID column, 30 feature columns, a weight column, and a label column. The test set comprises 550000 events with an ID and 30 feature columns. We are interested to know about the individual and pairwise importance of 6 of the features to the performance of the classifier trained on the dataset.

We add a description of the desired ablation trials as a hint annotation and add it immediately after the line of code in which we load the training dataset. Listing 4 shows the annotation we added to the original source code. Upon providing the source code to ABLATIONMAGE, the response from the first call gets the correct implementation required to execute all the trials.

4.4 Discussion

The results of these experiments verify the usability and potential of ABLA-TIONMAGE as a tool to semi-automate the design and execution of ablation studies. Regarding the possibility of full automation, although LLMs have shown promising performance in generating ML/DL code, the users should still verify the correctness of the generated codes to make sure they correspond exactly to the target ablation study experiments. That being said, a natural next step for improving ABLATIONMAGE would be to provide more automation for the correctness/verification loop, possibly eliminating or reducing the need for multiple executions of possibly faulty code, e.g., by detecting and fixing common errors through static code analysis [28], [29].

Another possible challenge stems from the limited maximum number of input and output tokens and the limited context length of current LLMs [30]. This, in particular, may make it challenging for LLMs to understand and/or modify multiple source files simultaneously, especially in cases where the ML/DL system implementation spans many files and possibly hundreds of thousands of lines of code. This challenge can be alleviated by providing more sophisticated annotations and asking the LLM to provide experimentation code for the ablation trials one at a time. This may also ensure that the context length growth stays within the context length of the current publicly available LLMs such as GPT, Claude, and DeepSeek. This is particularly possible for code generation and code understanding tasks related to ablation studies, since the trials in an ablation study are intrinsically independent of each other (i.e., each trial can be generated and performed independently of the others). Nevertheless, we should also note the rapid improvement of LLMs in this regard; e.g., OpenAI's recent o3-mini model has a context window of 200K tokens and can have 100K tokens in its outputs, showing a remarkable increase compared to GPT-4's 8192-token context window and maximum output size. These improvements can enable LLMs to assist users with much more complex ablation studies. Our tests of ABLATIONMAGE on code repositories of complex DL frameworks and systems, while using more than a hundred source and documentation files as input documents, show that LLMs can successfully analyze and generate the desired code.

Limitations of existing dedicated ablation study frameworks (AutoAblation [12] and ABLATOR [13]) compared to utilizing state-of-the-art LLMs for conducting ablation studies include their dependence on their developers for implementing and providing support for different ML and DL frameworks, and the initial investment and learning curve required to learn their workflows, which might not be justified for smaller or simpler projects. We predict that as more users adopt LLM-based tools for their coding tasks [31], using LLM-based tools such as ABLATIONMAGE for conducting ablation studies will also become more common.

A natural extension to ABLATIONMAGE would be to provide full automation using agentic frameworks such as LangGraph [32] and AutoGen [33]. By leveraging these frameworks, a complete ablation study workflow can be structured as interactions among multiple specialized agents (e.g., for code generation and optimization, experiment execution, and result analysis), each potentially utilizing different LLMs and iterating over generated artifacts and outputs (such as errors and stack traces) until a desired outcome

PAPER 6. ABLATIONMAGE

System Prompt	LLM Agent Response
You are assisting me in creating code for ablation studies. I'm going to give you a runnable and correct script that trains a machine learning model. In the script, there may be lines that have a comment in the form of .# ABLATABLE_COMPONENT, These are related to ablation studies. Also, they might be comment block that start with #ABLATION_HINT_START and end with #ABLATION_HINT_END, these contain a description of the desired ablation study. When you see lines that have #ABLATABLE_COMPONENT or blocks of commented out lines enclosed in #ABLATABLE_COMPONENT or blocks of commented out lines enclosed in #ABLATABLE_COMPONENT or blocks of commented out lines enclosed in #ABLATABLE_COMPONENT or blocks of commented out lines the original script in such a way that the mentioned component is removed but the code is still runnable and correct.	<pre>I'll help you create a modified version of the script that implements the ablation study for the GraphSAGE layers. The study will test different architectures by removing layers and varying the number of neurons. Here's the complete modified code with ablation study capabilities: '`python if model_config['type'] == 'full': self.sage1 = SAGEConv(d, 32, aggre='max') self.sage3 = SAGEConv(32, 16, aggre='max') self.sage3 = SAGEConv(32, segre='max') self.sage5 = SAGEConv(32, segre='max') self.sage5 = SAGEConv(5, dim_out, aggre='max') self.sage5 = SAGEConv(5, dim_out, aggre='max')</pre>
Attached Files:	<pre>self.sage1 = SAGEConv(dim_in, 64, aggr='max') self.sage3 = SAGEConv(64_16_aggr='max')</pre>
Annotated Source Code Additional Documents	<pre>self.sage4 = SAGECONV(6, 10, aggr='max') self.sage5 = SAGECONV(8, dim_out, aggr='max')</pre>

Figure 2: The prompt prepared by ABLATIONMAGE for the "first call" to the LLM and part of the response. The user provides the path to the annotated source code and other documents.

is achieved. Meanwhile, a human-in-the-loop can validate artifacts and provide high-quality guidance for complex tasks, ensuring correctness and refining the workflow as needed.

5 Summary

In this paper, we investigated the potential of LLMs as aids for performing ablation studies in ML/DL. We highlighted a number of ways in which LLMs can be a facilitator for ablation studies, in the different steps of study design, experimentation, and analysis & presentation, and discussed the prototype implementation of ABLATIONMAGE, an LLM-based tool that can semi-automate the process of conducting an ablation study by exploiting explicit and hint annotations provided by the users. Using three experiments, including one in which we reproduced the ablation studies of a recently published paper, we evaluated the usability of ABLATIONMAGE and briefly reviewed its current challenges and possible extensions.

Appendix: Example Prompts and LLM Responses

Here we provide examples of the prompts that ABLATIONMAGE creates as well as the responses from Claude Sonnet 3.5, the LLM backend we used for our experimental evaluation. These prompts and responses correspond to the second experiment, in which we reproduced the ablation studies of a recently published paper by adding hint annotations to the original code.

PAPER 6. ABLATIONMAGE

		LLM Agent Response
System Prompt		I added two important imports at the beginning of the file:
Previously, I asked you to output a script based on some requirements, but it failed to run successfully. I will give you the script you provided, as well as the stack trace of errors or any other error message outputted from running that script. Try to fix it, and provide the full code (not only hints for modifications, but the full code) in response.		""python from torch_geometric.nn import SAGEConv import torch.nn.functional as F "" These imports are necessary because:
Attached Files: Previously executed source code Feedback / Error Output	7	 SAECconv is the GraphSAEC convolution layer implementation from PyTorch Geometric F' contains various functions like 'relu()', 'dropout()', and 'log_softmax()' that we use in the modelThe rest of the code remains the same. This should now run without the 'NameError' you encountered.

Figure 3: The prompt prepared by ABLATIONMAGE for the "follow-up call" to the LLM and part of the response.

The prompts and snippets of the responses for the first call are shown in Figure 2, while Figure 3 shows the same for the first follow-up call AB-LATIONMAGE makes. In this case, the "Feedback/Error Output" was the stack trace of errors after the execution of the code provided by the LLM in response to the first call.

References

- A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural* information processing systems, vol. 25, 2012.
- [2] S. Smith, M. Patwary, B. Norick, et al., "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," arXiv preprint arXiv:2201.11990, 2022.
- [3] A. Chowdhery, S. Narang, J. Devlin, et al., "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.
- [4] G. Penedo, Q. Malartic, D. Hesslow, et al., "The refined web dataset for falcon llm: Outperforming curated corpora with web data, and web data only," arXiv preprint arXiv:2306.01116, 2023.
- [5] M. Dehghani, J. Djolonga, B. Mustafa, et al., "Scaling vision transformers to 22 billion parameters," in *International Conference on Machine Learning*, PMLR, 2023, pp. 7480–7512.
- [6] A. Dubey, A. Jauhri, A. Pandey, et al., "The llama 3 herd of models," arXiv preprint arXiv:2407.21783, 2024.
- [7] C. Szegedy, W. Liu, Y. Jia, et al., "Going deeper with convolutions," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2015, pp. 1–9.

- [8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer* vision and pattern recognition, 2016, pp. 770–778.
- [9] A. Vaswani, N. Shazeer, N. Parmar, et al., "Attention is all you need," in Advances in Neural Information Processing Systems, I. Guyon, U. V. Luxburg, S. Bengio, et al., Eds., vol. 30, Curran Associates, Inc., 2017.
- [10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv e-prints, arXiv-1412, 2014.
- [11] R. Meyes, M. Lu, C. W. de Puiseau, and T. Meisen, "Ablation studies in artificial neural networks," arXiv preprint arXiv:1901.08644, 2019.
- [12] S. Sheikholeslami, M. Meister, T. Wang, A. H. Payberah, V. Vlassov, and J. Dowling, "Autoablation: Automated parallel ablation studies for deep learning," in *Proceedings of the 1st Workshop on Machine Learning and Systems*, 2021, pp. 55–61.
- [13] I. Fostiropoulos and L. Itti, "Ablator: Robust horizontal-scaling of machine learning ablation experiments," in *International Conference* on Automated Machine Learning, PMLR, 2023, pp. 19–1.
- [14] J. Wang, Z. Liu, L. Zhao, et al., "Review of large vision models and visual prompt engineering," *Meta-Radiology*, p. 100 047, 2023.
- [15] K. Tanahashi, Y. Inoue, Y. Yamaguchi, et al., "Evaluation of large language models for decision making in autonomous driving," arXiv preprint arXiv:2312.06351, 2023.
- [16] S. Yang, O. Nachum, Y. Du, J. Wei, P. Abbeel, and D. Schuurmans, "Foundation models for decision making: Problems, methods, and opportunities," arXiv preprint arXiv:2303.04129, 2023.
- [17] G. Jawahar, M. Abdul-Mageed, L. V. Lakshmanan, and D. Ding, "Llm performance predictors are good initializers for architecture search," arXiv preprint arXiv:2310.16712, 2023.
- [18] A. Chen, D. Dohan, and D. So, "Evoprompting: Language models for code-level neural architecture search," Advances in Neural Information Processing Systems, 2023.
- [19] M. R. Zhang, N. Desai, J. Bae, J. Lorraine, and J. Ba, "Using large language models for hyperparameter optimization," arXiv e-prints, arXiv-2312, 2023.
- [20] S. Liu, C. Gao, and Y. Li, "Large language model agent for hyperparameter optimization," arXiv preprint arXiv:2402.01881, 2024.

- [21] H. Ghasemirahni, A. Farshin, M. Scazzariello, M. Chiesa, and D. Kostić, "Deploying stateful network functions efficiently using large language models," in *Proceedings of the 4th Workshop on Machine Learning and Systems*, 2024, pp. 28–38.
- [22] J. Xu, J. Li, Z. Liu, et al., "Large language models synergize with automated machine learning," Transactions on Machine Learning Research, 2024, ISSN: 2835-8856. [Online]. Available: https://openreview. net/forum?id=RDEaIfOiJM.
- [23] D. Song, H. Guo, Y. Zhou, et al., "Code needs comments: Enhancing code LLMs with comment augmentation," in Findings of the Association for Computational Linguistics: ACL 2024, L.-W. Ku, A. Martins, and V. Srikumar, Eds., Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 13640–13656. DOI: 10. 18653/v1/2024.findings-acl.809. [Online]. Available: https: //aclanthology.org/2024.findings-acl.809.
- [24] A. Defilippo, P. Veltri, P. Lió, and P. H. Guzzi, "Leveraging graph neural networks for supporting automatic triage of patients," *Scientific Reports*, vol. 14, no. 1, p. 12548, 2024.
- [25] C. Adam-Bourdarios, G. Cowan, C. Germain, I. Guyon, B. Kégl, and D. Rousseau, "The higgs boson machine learning challenge," in *NIPS* 2014 workshop on high-energy physics and machine learning, PMLR, 2015, pp. 19–55.
- [26] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, 2016, pp. 785–794.
- [27] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," Advances in neural information processing systems, vol. 30, 2017.
- [28] P. Louridas, "Static code analysis," *Ieee Software*, vol. 23, no. 4, pp. 58–61, 2006.
- [29] X. Guan and C. Treude, "Enhancing source code representations for deep learning with static analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 64– 68.
- [30] H. Jin, X. Han, J. Yang, et al., "Llm maybe longlm: Self-extend llm context window without tuning," arXiv preprint arXiv:2401.01325, 2024.
- [31] D. Zheng, Y. Wang, E. Shi, H. Zhang, and Z. Zheng, "How well do llms generate code for different application domains? benchmark and evaluation," arXiv preprint arXiv:2412.18573, 2024.

- [32] LangChain-AI, LangGraph, https://github.com/langchain-ai/ langgraph, Accessed: 2025-02-09, 2025.
- [33] Q. Wu, G. Bansal, J. Zhang, *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," *arXiv preprint arXiv:2308.08155*, 2023.