UNIVERSITY OF CALIFORNIA SAN DIEGO

Query Optimizations for Deep Learning Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Supun Nakandala

Committee in charge:

Professor Arun Kumar, Chair
Professor Loki Natarajan
Professor Yannis Papakonstantinou
Professor Lawrence Saul
Professor Geoffrey M. Voelker

2022

The dissertation of Supun Nakandala is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

DEDICATION

This dissertation is lovingly dedicated to my parents, Nilmini and Jayantha. Their love, encouragement, and sacrifices have helped me achieve great things in my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I am thankful to my Ph.D. advisor Professor Arun Kumar for giving me the opportunity and teaching me how to become an independent researcher. This dissertation would not have been possible without his support, guidance, and encouragement. I am also thankful for his valuable professional advice and support to pursue opportunities beyond my thesis research. I feel very fortunate to have him as my advisor.

Besides my advisor, my sincere thanks also go to the other members of my thesis committee: Professor Yannis Papakonstantinou, Professor Geoffrey Voelker, Professor Lawrence Saul, and Professor Loki Natarajan, for their feedback and comments that helped me complete my dissertation. I also thank Professor Victor Vianu and Professor Alin Deutsch for their valuable feedback to improve my work and presentation skills during the database seminar.

I thank my internship mentors: Dr. Vivek Narasayya at Microsoft Research, Professor Yannis Papakonstantinou at Amazon Web Services, and Dr. Matteo Internlandi at Microsoft, for the opportunity and the valuable mentorship that they gave me. These internships helped me gain valuable industrial research exposure, which I highly enjoyed. These internships were a key motivating reason for me to pursue a career in the industry.

I am also thankful to all my co-authors: Arun Kumar, Yannis Papakonstantinou, Loki Natarajan, Andrea LaCroix, John Bellettiere, Jordan Carlson, Paul R Hibbing, Jingjing Zou, Marta M Jankowska, Mikael Anne Greenwood-Hickman, Dori Rosenberg, Fatima Tuz-Zahra, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, Matteo Interland, Yuhao Zhang, Kabir Nagrecha, Liangde Li, Allen Ordookhanians, and Xin Li. I learned a lot from them and my dissertation would not have been possible without their collaboration and valuable contributions.

I was fortunate to make many new friends and colleagues during my Ph.D. studies at UC San Diego. I would like to especially mention the members of the ADALab, members of the Database Group, and my research collaborators from the UC San Diego Public Health

Department. I thank them for the insightful discussions and feedback on my work, papers, and talks. Finally, I would not be able to achieve anything without the support and understanding from my family and my beloved wife, Thanya.

The material in this dissertation is based on the following publications.

- Chapter 3 contains material from "Cerebro: A Data System for Optimized Deep Learning Model Selection" by Supun Nakandala, Yuhao Zhang, and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 13, Issue 12, July 2020. The dissertation author was the primary investigator and author of this paper.

- Chapter 4 Section 4.1 contains material from "The CNN Hip Accelerometer Posture (CHAP) Method for Classifying Sitting Patterns from Hip Accelerometers: A Validation Study" by Supun Nakandala, Mikael Anne Greenwood-Hickman, Marta M Jankowska, Dori Rosenberg, Fatima Tuz-Zahra, John Bellettiere, Jordan Carlson, Paul R Hibbing, Jingjing Zou, Andrea Z LaCroix, Arun Kumar, and Loki Natarajan, which appears in the Journal of Medicine & Science in Sports & Exercise Volume 53, Issue 11, November 2021. The dissertation author was a primary investigator and a primary author of this paper.

- Chapter 4 Section 4.2 contains material from "Intermittent Human-in-the-Loop Model Selection using Cerebro: A Demonstration" by Liangde Li, Supun Nakandala, and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 14, Issue 12, July 2021. The dissertation author's contribution was in the conceptualization of the system, parts of the implementation, and advising the junior student through the rest of the system implementation.

- Chapter 5 contains material from "A Tensor Compiler for Unified Machine Learning Prediction Serving" by Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi, which appears in Proceedings of

14^{th} USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020). The dissertation author was the primary investigator and author of this paper.

- Chapter 6 contains material from "Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations" by Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou, which appears in Proceedings of 2019 ACM SIGMOD International Conference on Management of Data. The dissertation author was the primary investigator and author of this paper.

- Chapter 7 Section 7.1 contains material from "Demonstration of Krypton: Optimized CNN Inference for Occlusion-based Deep CNN Explanations" by Allen Ordookhanians, Xin Lin, Supun Nakandala, and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 12, Issue 12, August 2019. The dissertation author's contribution was in the conceptualization of the system and advising the junior students through the system implementation.

- Chapter 7 Section 7.2 contains material from "Incremental and Approximate Computations for Accelerating Deep CNN Inference" by Supun Nakandala, Kabir Nagrecha, Arun Kumar, and Yannis Papakonstantinou, which appears in ACM Transactions on Database Systems Journal Volume 45, Issue 4, December 2020. The dissertation author was the primary investigator and author of this paper.

- Chapter 8 contains material from "Vista: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale" by Supun Nakandala and Arun Kumar, which appears in Proceedings of 2020 ACM SIGMOD International Conference on Management of Data. The dissertation author was the primary investigator and author of this paper.

- Chapter 9 contains material from "Nautilus: An Optimized System for Deep Transfer Learning over Evolving Training Datasets" by Supun Nakandala and Arun Kumar, which

appears in Proceedings of 2022 ACM SIGMOD International Conference on Management of Data. The dissertation author was the primary investigator and author of this paper.

My co-authors have kindly approved the inclusion of the aforementioned publications in my dissertation.

VITA

2015	B.Sc. in Computer Science, University of Moratuwa, Sri Lanka
2020	M.Sc. in Computer Science, University of California San Diego
2022	Ph.D. in Computer Science, University of California San Diego

PUBLICATIONS

Supun Nakandala and Arun Kumar, "Nautilus: An Optimized System for Deep Transfer Learning over Evolving Training Datasets", *Proceedings of the 2022 International Conference on Management of Data (SIGMOD 2022)*.

Liangde Li, **Supun Nakandala**, and Arun Kumar, "Intermittent Human-in-the-loop Model Selection using Cerebro: A Demonstration", *Proceedings of the 2021 Very Large Data Bases Conference (VLDB 2021)*, pages 2687-2690, 2021.

Supun Nakandala, Mikael Anne, Marta M. Jankowska, Dori Rosenberg, Fatima Tuz-Zahra, John Bellettiere, Jordan Carlson, Paul R. Hibbing, Jingjing Zou, Andrea Z. LaCroix, Arun Kumar, and Loki Natarajan, "The CNN Hip Accelerometer Posture (CHAP) Method for Classifying Sitting Patterns from Hip Accelerometers: A Validation Study in Older Adults", *Medicine & Science in Sports & Exercise*, 53(11):2445-2454, 2021

Supun Nakandala, Marta Jankowska, Fatima Tuz-Zahra, John Bellettiere, Jordan Carlson, Andrea LaCroix, Sheri Hartman, Dori Rosenberg, Jingjing Zou, Arun Kumar, and Loki Natarajan, "Application of Convolutional Neural Network Algorithms for Advancing Sedentary and Activity Bout Classification", *Journal for the Measurement of Physical Behaviour*, 4(2):102-110, 2021

Arun Kumar, **Supun Nakandala**, Yuhao Zhang, Side Li, Advitya Gemawat, and Kabir Nagrecha, "Cerebro: A Layered Data Platform for Scalable Deep Learning", *Proceedings of the 2021 Conference on Innovative Data Systems Research (CIDR 2021)*, 2021.

Supun Nakandala, Kabir Nagrecha, Arun Kumar, and Yannis Papakonstantinou, "Incremental and Approximate Computations for Accelerating Deep CNN Inference", *ACM Transactions on Database Systems (TODS) 45*, pages 1-42, 2020.

Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou, "Query Optimization for Faster Deep CNN Explanations", *ACM SIGMOD Record 49*, pages 61-68, 2020.

Supun Nakandala, Yuhao Zhang, and Arun Kumar, "Cerebro: A Data System for Optimized Deep Learning Model Selection", *Proceedings of the 2020 Very Large Data Bases Conference (VLDB 2020)*, pages 2159-2173, 2020.

Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi, "A Tensor Compiler for Unified Machine Learning Prediction Serving", *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, pages 899-917, 2020.

Supun Nakandala and Arun Kumar, "Vista: Declarative Feature Transfer from Deep CNNs at Scale", *Proceedings of the 2020 International Conference on Management of Data (SIGMOD 2020)*, pages 1685-1700, 2020.

Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, **Supun Nakandala**, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino, "Extending Relational Query Processing with ML Inference", *Proceedings of the 2020 Conference on Innovative Data Systems Research (CIDR 2020)*, 2020.

Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou, "Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations", *Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2019)*, pages 1589-1606. 2019.

Supun Nakandala, Gyeong-In Yu, Markus Weimer, and Matteo Interlandi, "Compiling Classical ML Pipelines into Tensor Computations for One-size-fits-all Prediction Serving", *Proceedings of the Systems for ML workshop at NeurIPS*, 2019.

Supun Nakandala, Yuhao Zhang, and Arun Kumar, "Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems", *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning (DEEM 2019)*, pages 1-4. 2019.

Allen Ordookhanians, Xin Li, **Supun Nakandala**, and Arun Kumar, "Demonstration of Krypton: Optimized CNN Inference for Occlusion-based Deep CNN Explanations", *Proceedings of the 2019 Very Large Data Bases Conference (VLDB 2019)*, pages 1894-1897, 2019.

ABSTRACT OF THE DISSERTATION

Query Optimizations for Deep Learning Systems

by

Supun Nakandala

Doctor of Philosophy in Computer Science

University of California San Diego, 2022

Professor Arun Kumar, Chair

Deep Learning (DL) has unlocked unstructured data for analytics. It has enabled new applications, insights, and value in various domains, including enterprises, domain sciences, and healthcare. However, DL workloads are highly resource-intensive and time-consuming, which hinder their adoption. Thus, optimizing them from a systems standpoint has attracted significant attention in recent years. In this dissertation, we fundamentally re-imagine DL workloads as data processing workloads and optimize them from a data management standpoint. Using a combination of abstractions already available in DL practice, new algorithms, system design, theoretical and empirical analysis, we show how classical query optimization ideas such as rewrites, multi-query optimization, materialization optimization, incremental view maintenance,

approximate query processing, and predicate push-down can be re-imagined in the context of DL workloads to optimize them. We show that our techniques can enable significant runtime and resource savings (even over 10X for some cases) for a variety of popular and important end-to-end DL workloads. Our work fills a critical technical gap in DL systems architecture and opens up new connections between query optimization and DL systems.

Chapter 1

Introduction

1.1 Motivation

Deep Learning (DL) [124] has unlocked unstructured data for analytics. It has enabled new applications, insights, and value in various domains, including enterprises [137], domain sciences [282], and even in critical application domains such as in healthcare [160, 148]. However, DL workloads are highly resource-intensive and time-consuming, which hinder their adoption. Thus, in recent years, optimizing them from a systems standpoint has attracted significant attention from the computer systems community.

In this dissertation, we fundamentally reimagine DL workloads as data processing workloads to optimize them from a data management standpoint. We identify and optimize three popular DL workload types: 1) model building, 2) model inference, and 3) transfer learning, which is a special type of model training workload that reuses pre-trained models. Figure 1.1 presents an illustration of these workload types and several workloads pertaining to each type.

However, one cannot directly apply existing data management techniques to optimize DL workloads. This is because DL workloads are significantly different from other popular data processing workloads at various levels, including the data model, computational model, execution

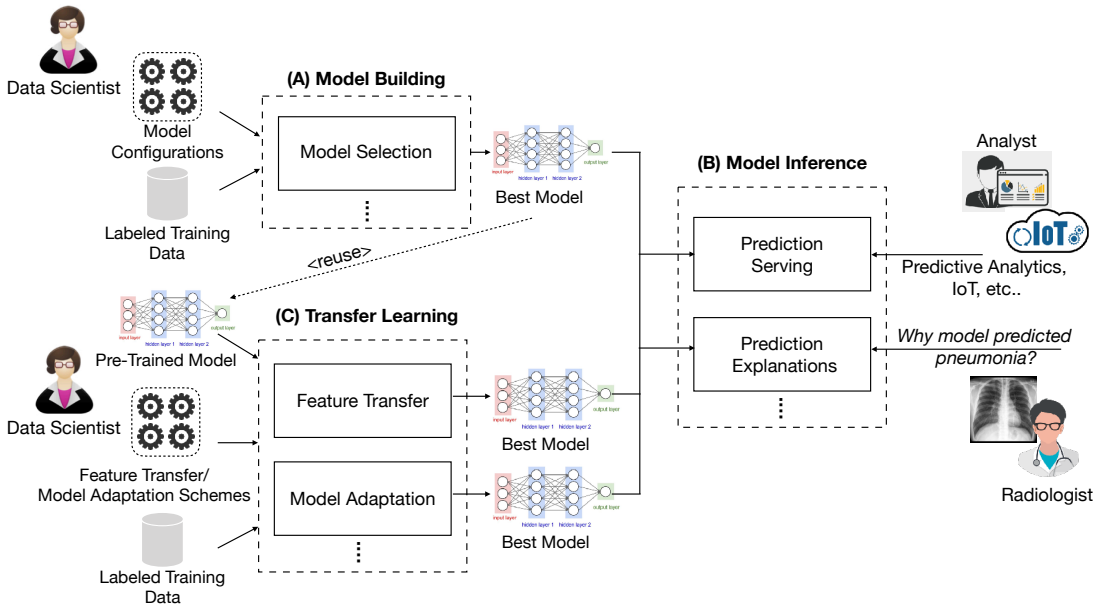


Figure 1.1: The three high-level DL workload types identified in this dissertation. (A) Model building workloads. (B) Model inference workloads. (C) Transfer learning workloads. For each workload type, we also present example workloads, which we focus on optimizing in this dissertation work.

characteristics, and user interaction modes. For example, DL workloads use [1]tensors as the data model and the neural computational graph as the computational model [124], which are unique to the DL workloads. DL workloads are also highly data-intensive. Training a DL model with good accuracy requires millions of training examples [108]. DL workloads are also highly compute- and memory-intensive [60]. DL model training is a complex iterative numerical optimization process [124], which can take weeks of computing time even on high-end hardware accelerators such as GPUs and TPUs. DL models can also have billions of parameters [257], which bloats their runtime memory footprint. Furthermore, DL model training is an empirical process and requires a significant amount of exploratory work to pick the best model for a given task. This exploratory process is also called model selection [124, 173]. Overall, these characteristics hinder DL adoption for many practitioners.

The computer systems community has identified the lack of specialized systems for DL

[1]Generalization of matrices to higher dimensions.

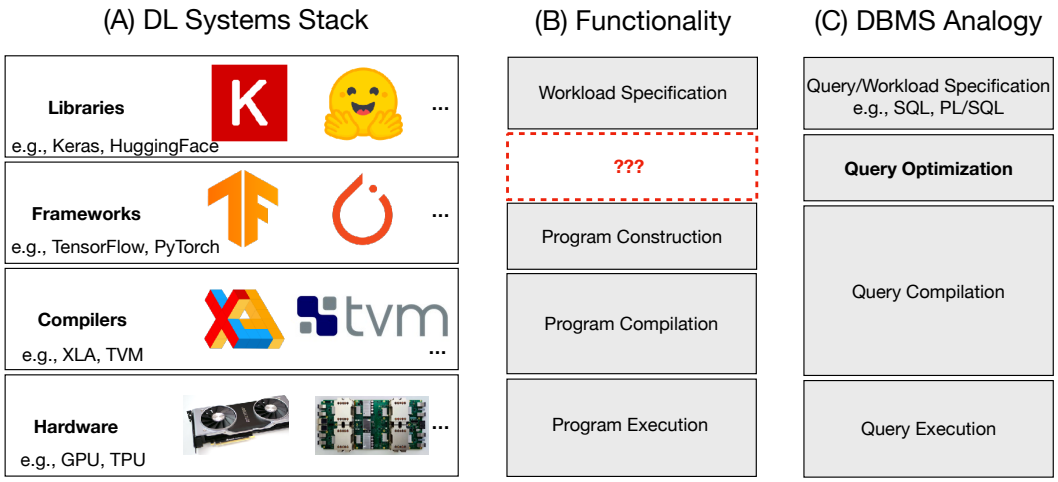


Figure 1.2: (A) DL systems stack (hardware, compilers, frameworks, and libraries). (B) Functionality performed by layers in the DL systems stack. (C) Database management systems analogy for the functionality performed by layers in the DL systems stack.

as a blocker for DL adoption and has invested in developing optimized systems for DL. These systems are collectively referred to as DL systems. They span the entire vertical cross-section of computing. An illustration of the DL systems stack is presented in Figure 1.2 (A). At the bottom-most layer, there are custom hardware accelerators (e.g., GPUs, TPUs, and custom ASICs) developed for DL [179]. On the layer above, there are compilers (e.g., TVM [93], XLA [52]) developed for DL. On the very top, there are frameworks (e.g., TensorFlow [54], PyTorch [228]) and libraries (e.g., Keras [131], HuggingFace [284]) developed for DL. In terms of functionality, the libraries layer provides application programming interfaces (APIs) for specifying various DL workloads. The frameworks layer contains the application logic needed to construct the final program. The compiler layer then compiles the program to executable code to execute on the hardware.

In terms of the layers of functionality, the DL systems stack bears strong similarities to the database management systems (DBMSs) architecture [237] as shown in Figure 1.2 (C). For instance, the workload specification aspect of the DL libraries is analogous to the query/workload

specification layer in DBMSs (e.g., SQL, PL/SQL). The program construction aspect of the DL frameworks and program compilation aspects of the DL compilers are collectively analogous to the query compilation layer in DBMSs. The program execution on the DL hardware is analogous to the query execution layer in DBMSs. However, one major research gap in the DL systems stack is the analogous of a query optimization layer. The query optimization layer [150] is a major component in the DBMS architecture, which enables achieving high efficiencies despite providing an easy-to-use declarative workload specification interface (e.g., SQL) for the users. The query optimization layer performs automated optimizations based on the query workload characteristics such as data access and computation patterns. For other data processing workloads such as relational query processing, query optimization is an extensively studied area [88].

The lack of a query optimization counterpart in the DL systems stack results in significant missed opportunities for optimization. Even though the DL compilers perform program-level compiler optimizations (e.g., operator fusion, loop optimizations [93]), they operate at a lower granularity and do not optimize for workload-level semantic, logical, and physical characteristics. Considering this, the thesis of this dissertation is that:

> *For a variety of end-to-end model building, model inference, and transfer learning workloads, by characterizing the semantic, logical, and physical characteristics of those workloads, novel query optimization-inspired techniques can be developed to significantly improve system efficiency and reduce resource costs of them.*

Using a combination of abstractions already available in DL practice, new algorithms, system design, theoretical and empirical analysis, we show how classical query optimization ideas such as rewrites [227], multi-query optimization [250], materialization optimization [96], incremental view maintenance [86], approximate query processing [120], and predicate push-down [272] can be re-imagined in the context of DL workloads to optimize them. We show that our approach can enable significant runtime and resource savings (even over 10X for some cases) for a variety of popular end-to-end DL workloads.

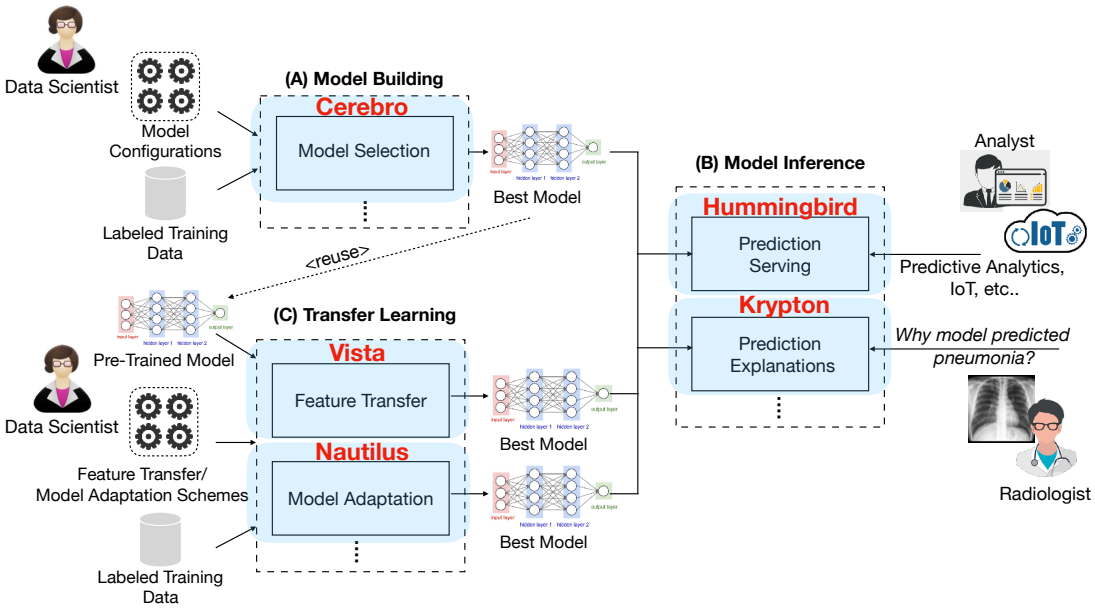
1.2 Technical Contributions

In this dissertation, we select five popular and important end-to-end DL workloads and optimize them from a data management standpoint. We chose these workloads based on our interactions with DL practitioners and domain scientists and the recent trends identified from influential DL publications. We identify semantic, logical, and physical characteristics of these workloads that are not leveraged by existing DL systems and develop new query optimization-inspired techniques to optimize them. Our physical optimizations operate at the lowest level and are the closest to the physical execution. Our logical optimizations are higher-level than physical optimizations. They rearrange the computations and data accesses of a workload while ensuring the exact output. Our semantic optimizations operate at the highest level and leverage the workload semantics to trade off the output accuracy with system/resource efficiency.

We bake in our techniques into specialized systems optimized for each DL workload. As much as possible, we try to leverage existing DL systems and data analytics systems and implement our ideas on top of these existing systems without making any changes to the internals of those systems. This approach reduces the implementation efforts and enables wider adoption of our techniques. It also allows us to automatically benefit from future advancements in the rest of the DL systems stack (e.g., compilers and architecture). We organize our contributions based on the type of DL workload they optimize: 1) model building, 2) model inference, and 3) transfer learning. Figure 1.3 summarizes our contributions.

1.2.1 Model Building Workloads

We focus on model selection workloads, which is an unavoidable step in model training. Model selection requires training and evaluating several model configurations (sometimes hundreds) before picking the best model. Thus, it significantly increases the resource costs of DL model training.



	Semantic Optimizations	Logical Optimizations	Physical Optimizations
Cerebro (Chapter 3)			✓
Hummingbird (Chapter 5)		✓	✓
Krypton (Chapter 6)	✓	✓	✓
Vista (Chapter 8)		✓	✓
Nautilus (Chapter 9)			✓

Figure 1.3: The different systems that we have developed, the workloads they optimize, and the type of optimizations they perform.

CEREBRO: Query Optimizations for DL Model Selection. We observed that most DL systems today focus on training one model at a time, reducing throughput and raising overall resource costs; some also sacrifice reproducibility. We propose CEREBRO, which is a new data system to raise DL model selection throughput at scale without raising resource costs and sacrificing reproducibility or accuracy. CEREBRO uses a new parallel DL model training strategy we call model hopper parallelism. Model hopper parallelism leverages the physical characteristics of the DL model training process to optimize it. Model hopper parallelism hybridizes task- and data-parallelism to mitigate the cons of these prior paradigms and offers the best of both worlds. Experiments on large benchmark datasets show that CEREBRO offers 3X to 10X runtime savings

relative to data-parallel systems like Horovod [252] and Parameter Server [187] and up to 8X memory/storage savings or up to 100X network savings relative to task-parallel systems [261].

This work is the subject of Chapter 3 and is joint work with Yuhao Zhang and Arun Kumar. A paper on this work appeared at the VLDB conference in 2020 [215]. In Chapter 4 we dive into the applications and extensions of CEREBRO. There we provide more details about a public health application case study that was supported by CEREBRO and details on how we extended CEREBRO to implement a novel human-in-the-loop model selection paradigm. The code for our system is open source and is available on GitHub: https://github.com/ADALabUCSD/cerebro-system.

1.2.2 Model Inference Workloads

Inference is the process of generating predictions from trained models. We focus on two inference workloads: 1) prediction serving and 2) prediction explanation.

HUMMINGBIRD: Query Optimizations for Classical ML Prediction Serving on DL Systems. We observed that while there are specialized systems for DL inference (e.g., TVM [93], ONNXRuntime [15]), classical ML inference workloads in enterprises are supported in an ad-hoc manner. Yet, classical machine learning workloads are very popular in enterprises and the lack of specialized systems support contributes to significant infrastructure complexity and increased costs [61]. In this work, we use query optimization-inspired techniques to develop a system called HUMMINGBIRD, which enables classical ML inference on DL systems. HUMMINGBIRD leverages the logical and physical characteristics of classical ML inference workloads and rewrites featurization operators and traditional ML models (e.g., decision trees) into a set of tensor operations supported by DL inference systems. We show that our approach not only reduces the infrastructure complexity by leveraging existing investments in DL inference systems, but can also enable significant runtime speedups of up to 3X against hand-crafted GPU kernels, and up to 1200X for predictive pipelines against state-of-the-art frameworks.

This work is the subject of Chapter 5. It was started as an internship project while interning

at Microsoft and is joint work with Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. A paper on this work appeared at the USENIX OSDI conference in 2020 [213]. The code for our system is open source and is available on GitHub: https://github.com/microsoft/hummingbird.

KRYPTON: Query Optimizations for Deep CNN Prediction Explanations. Deep Convolutional Neural Networks (CNNs) [171, 260] are a special type of DL model optimized for image prediction tasks. They match human-level accuracy in many prediction tasks, resulting in a growing adoption in various domains including healthcare [160, 148]. Naturally, "explaining" CNN predictions is a key concern for many users and occlusion-based explanations [290] (OBE) is a popular technique for understanding CNN predictions. In OBE, one occludes a region of the image using a patch and moves it around to produce a heat map of changes to the prediction probability. This approach is computationally expensive due to the large number of re-inference requests produced, which wastes time and raises resource costs.

In this work, we leverage the semantic, logical, and physical characteristics of the OBE workload and develop a system we call KRYPTON to optimize it. We cast the OBE workload as a new instance of the classical incremental view maintenance problem [86]. We create a novel and comprehensive algebraic framework for incremental CNN inference combining materialized views with multi-query optimization [250] to reduce computational costs. We also present two novel approximate inference optimizations that exploit the semantics of CNNs and the OBE task to further reduce runtimes. Experiments with real data and CNNs show that KRYPTON reduces runtimes by up to 5X (resp. 35X) to produce exact (resp. high-quality approximate) results without raising resource requirements.

This work is the subject of Chapter 6 and is joint work with Arun Kumar and Yannis Papakonstantinou. In Chapter 7, we dive into the extensions of KRYPTON. There we provide details on how we extended KRYPTON to support interactive CNN prediction diagnosis and support accelerating OBE for arbitrary CNN models.

A paper on this work appeared at the ACM SIGMOD conference in 2019 [211], and it was recognized as an honorable mention for the ACM SIGMOD 2019 best paper award for its novelty and innovation in the transfer of database knowledge to—seemingly—unrelated domains. It also received a SIGMOD 2020 research highlight award and was invited to the ACM TODS journal in 2020.

1.2.3 Transfer Learning Workloads

Transfer learning [268] is a crucial paradigm for democratizing DL, which reduces the high resource costs of DL training workloads. It does so by reusing pre-trained DL models instead of training new models for every new task. We present two scenarios where query optimization-inspired techniques can be used to optimize transfer learning workloads: 1) deep CNN feature transfer and 2) DL model adaptation.

VISTA: Query Optimizations for Deep CNN Feature Transfer. Deep CNNs provide a unique opportunity to holistically integrate image data in enterprise data analytics, which has hitherto relied mainly on structured data [137]. Since training deep CNNs from scratch is expensive and laborious, feature transfer has become popular: using a pre-trained CNN, one reads off certain intermediate output features from the model to represent images and combines them with other features for a downstream machine learning task. Since no single intermediate output will always offer the best accuracy in general, such feature transfer requires comparing many intermediate outputs.

We observed that the current approach to this process on top of DL systems such as TensorFlow [54] and scalable analytics systems such as Spark [289] is fraught with inefficiency due to redundant CNN inference and the potential for system crashes due to manual memory management. We propose VISTA, a system to mitigate such issues by elevating the feature transfer workload to a declarative level and formalizing the data model of CNN inference. VISTA leverages the logical and physical characteristics of CNN feature transfer workloads to optimize them. It

casts the deep CNN feature transfer workload as a novel instance of multi-query optimization and enables automated optimization of feature materialization trade-offs, memory usage, and system configuration. Experiments with real-world datasets and deep CNNs show that apart from enabling seamless feature transfer, VISTA reduces runtimes by 67%.

This work is the subject of Chapter 8 and is joint work with Arun Kumar. A paper on this work appeared at the ACM SIGMOD conference in 2020 [210]. The code for our system is open source and is available on GitHub: https://github.com/AdaLabUCSD/Vista.

NAUTILUS: Query Optimizations for DL Model Adaptation. In DL model adaptation workloads, one adapts a pre-trained model for a target task instead of training a new model from scratch. A common practice during this process is to freeze most pre-trained model parameters [194, 80] and adapt only the remaining. Since no single freezing scheme is universally the best, one often evaluates several model freezing schemes. Furthermore, this process is repeated whenever new training data become available for the target task [65, 73]. Today, one executes this workload by performing computations for the entire pre-trained model and repeats it every time new training data become available. This approach results in redundant computations in frozen model parts and causes system inefficiency issues.

We propose NAUTILUS, a system to reduce redundant computations and training overheads of DL model adaptation workloads. NAUTILUS leverages the physical characteristics of model adaptation workload and casts it as a new instance of multi-query optimization [250]. Experiments with end-to-end workloads on benchmark datasets show that NAUTILUS reduces DL model adaptation runtimes by up to 5X compared to the current practice.

This work is the subject of Chapter 9 and is joint work with Arun Kumar. A paper on this work will appear at the ACM SIGMOD conference in 2022 [12]. The code for our system is open source and is available on GitHub: https://github.com/AdaLabUCSD/Nautilus.

1.3 Summary and Impact

DL is an emerging data processing workload category that requires specialized systems support. DL systems stack bears strong similarities to other data processing system stacks such as DBMSs. However, one major difference of current DL systems stack compared to the DBMSs is the lack of a counterpart for query optimization. As a result, DL systems miss significant opportunities for workload optimizations. In this dissertation, we propose novel query optimization-inspired workload optimization techniques for DL systems. Our optimization techniques cover three major DL workload categories: 1) model building workloads, 2) model inference workloads, and 3) transfer learning workloads. We show that our techniques can significantly improve system efficiency and reduce resource costs of DL workloads. Our work fills critical technical gaps in DL systems architecture and opens up new connections between query optimization and DL systems.

Our interactions with domain experts, analysts, and engineers from academic institutions and industry helped us identify many practical bottlenecks faced when adopting DL. These bottlenecks became the research focus of this dissertation, and in some cases, we were able to transition our techniques and systems for real-world adoption. In particular, at the time of writing this document, the research systems discussed in this dissertation have had the following practical impacts:

- The CEREBRO system has been used by public health researchers at UC San Diego to train behavior prediction models on terabyte-scale accelerometer training data [127].
- Ideas from the CEREBRO system has been integrated into the Apache MADLib library, and VMware has shipped it to their enterprise customers for performing in-database DL model selection [275].
- The HUMMINGBIRD system has been open-sourced by Microsoft [38] and it has been

downloaded over 60,000 times using the PyPI package manager. Also, Microsoft has integrated the HUMMINGBIRD system to their ONNXRuntime and SynapseML systems.

- We have also started exploring the possibility of adopting ideas from the NAUTILUS system for public health use cases at UC San Diego.

Chapter 2

Background

2.1 Deep Learning

DL is a machine learning paradigm that uses artificial neurons as the building block [124]. DL models take raw features as input and perform a series of transformation operations to generate the final output predictions. These transformations are organized into groups called layers. Internally, a DL model is organized as a directed graph of layers. A layer contains multiple neurons and these neurons are connected to neurons in the input layers through learnable weight parameters. The output of a neuron is computed by some non-linear function of the sum of its inputs. Figure 2.1 (A) presents an illustration of a DL model.

The internal layers of a DL model are referred to as hidden layers. During DL model training, hidden layers automatically learn to extract a hierarchy of relevant features directly from raw data [290]. Thus, DL significantly deviates from the previous paradigms of machine learning, which require extensive human effort for engineering the relevant features for a task. In recent years, DL models have revolutionized unstructured data analytics (e.g., images, speech, text, time-series data), resulting in near-human accuracies for some prediction tasks [2].

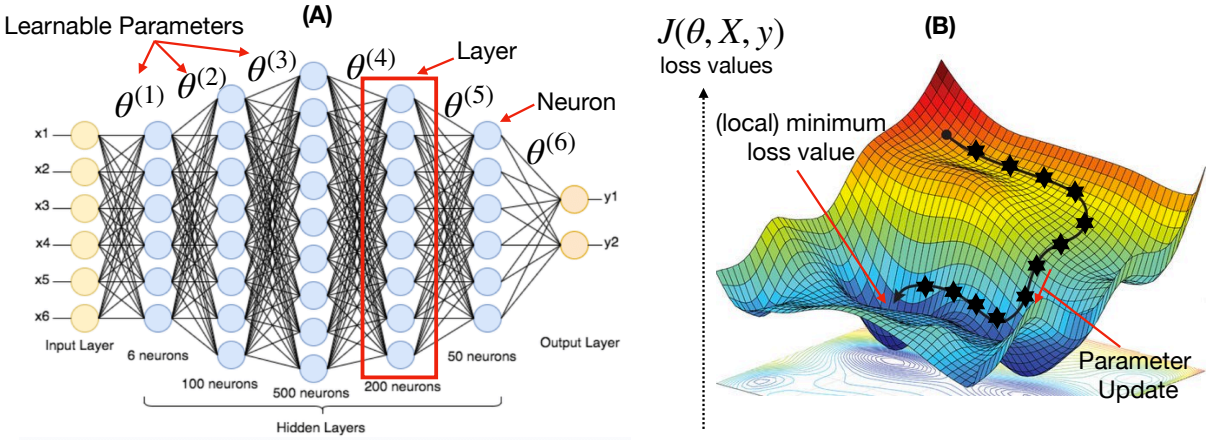


Figure 2.1: (A) Deep learning model. (B) Mini-batch stochastic gradient descent in action. $J(\theta, X, y)$ is the loss function; θ is the set of learnable parameters; (X, y) is the training data.

2.2 Mini-batch Stochastic Gradient Descent

DL model training tries to learn optimal values for the model parameters θ using some loss function J. The loss function essentially captures how good the model approximates the input-output (X, y) mappings in the training data. One such widely-used loss function is the cross-entropy loss function [124].

Given such a loss function, the mini-batch stochastic gradient descent (mini-batch SGD) method or its variants are used to train DL models. Mini-batch SGD is an iterative numerical optimization method, which performs multiple scans over the data using small data batches called mini-batches. A single scan is also called an epoch of training. In an epoch, SGD randomly samples a mini-batch of training data $(X^{(i:i+n)}, y^{(i:i+n)})$ and calculates the loss $J(\theta; X^{(i:i+n)}; y^{(i:i+n)})$. It then calculates the loss gradients with respect to all the learnable model parameters ∇_θ and updates the parameters to reduce the loss as per Equation 2.1. The parameter gradients are also weighted by a coefficient η, which is also called the learning rate.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; X^{(i:i+n)}; y^{(i:i+n)}) \tag{2.1}$$

Each parameter update tries to reduce the loss value and find optimal values for the model

parameters. This process is continued until the model has converged such that the loss value has reached a minimum value and it wouldn't improve any further. This process is pictorially depicted in Figure 2.1 (B). SGD is guaranteed to find the optimal values for the model parameters only for convex optimization problems. However, DL model training is a non-convex optimization problem. Yet, it has been shown that SGD can converge to a good local optimum in most cases. SGD is inherently sequential; deviating from sequential execution may lead to poor convergence behavior, typically raising the number of parameter updates needed for a given accuracy.

The computations needed for mini-batch SGD are performed in two stages: forward pass and backward pass.

2.2.1 Forward Pass of Computations

The mini-batch loss values $J(\theta; X^{(i:i+n)}; y^{(i:i+n)})$ needed for SGD are calculated using a forward computation pass over the model. During the forward pass, input feature values X of a mini-batch are propagated through the layers, where each layer performs some non-linear transformation. The final layer generates the output predictions. These predictions values and the actual label values y are then used to calculate the mini-batch loss value.

2.2.2 Backward Pass of Computations

Parameter gradients ∇_θ needed for mini-batch SGD are calculated using a backward computation pass over the model, which is also called backpropagation. Backpropagation leverages the graph structure of the model and the chain rule of differentiation to efficiently calculate the gradients. After the forward-pass and the loss computation, backpropagation calculates the loss gradient with respect to the model outputs and traverses backward from the model output layers towards the model input layers. Along its path, it propagates the loss gradient through the layers. The loss gradients with respect to the inputs of a layer, also called the

input gradients, are calculated using the loss gradient with respect to the output of that layer. Backpropagation also calculates the layer parameter gradients ∇_θ using the layer output gradients and uses them to update the learnable parameters.

2.3 Model Selection

The accuracy of a trained DL model heavily depends on the model architecture (e.g., model graph structure, number of layers, layer types, number of artificial neurons inside a layer) and also on the various training hyperparameter (e.g., training mini-batch size, learning rate, regularization values) used during training [124, 173]. Thus, training a DL model is seldom a one-off process. Practitioners often explore several different model architectures and hyperparameter values before picking the best configuration for their given task. This empirical process is referred to as model selection, and it is unavoidable because it is how one controls underfitting vs. overfitting [255]. Model selection is a major bottleneck for the adoption of DL among enterprises and domain scientists due to both the time spent and resource costs. In chapter 3, we provide more details about model selection along with a case study and dive deep into our techniques for optimizing it.

2.4 Model Inference

Inference is the process of generating predictions from a trained DL model. DL inference is a simpler process compared to the training process as it involves only the forward-pass of computations. Even though the model inference process is simple, it is a key contributor to the increased system complexity, performance concerns, and overall operational efficiency of a machine learning infrastructure [29]. For instance, each model is trained once but used multiple times for inference in a variety of environments. Thus inference dominates infrastructure complexity for deployment, maintainability, and monitoring. Second, model inference is often

in the critical path of interactive and analytical applications, hence its performance (in terms of latency and throughput) is an important concern for enterprises. Finally, model inference is responsible for 45-65% of the total cost of ownership of machine learning solutions [61]. In chapter 5, we provide more details about model inference and dive deep into our techniques for enabling classical ML inference on DL inference systems.

In addition to making basic predictions, some DL users often seek an "explanation" for why a DL model predicted a certain prediction. Hence, model explainability workloads have also become a popular and important DL workload category. Prediction explanations can help users trust DL models [242], especially in high-stakes applications such as radiology [256], and are a legal requirement for machine learning applications in some countries [276]. How to explain a DL model prediction is still an active research question. However, perturbation-based [290] and gradient-based [251] approaches are two widely used method families for explaining DL model predictions. In chapter 6, we provide more details about occlusion-based explanation, a widely used technique for explaining deep convolutional neural network (CNN) predictions, and dive deep into our techniques for optimizing it.

2.5 Deep Transfer Learning

The success of DL is mainly driven by how it extracts a hierarchy of relevant parameterized features from raw data, with the parameters learned automatically during training [124]. But, DL has a major bottleneck: model training is expensive. In many cases, it needs large training datasets (e.g., millions of records) and incurs high compute costs (e.g., days to weeks of GPU time). This hinders DL adoption, especially in low-resource settings. These bottlenecks can be overcome using a paradigm called deep transfer learning (DTL).

DTL leverages the fact that most of the features learned by a DL model when trained on a large dataset like Wikipedia, are general enough to be reused in other similar settings [245, 181,

229, 109]. Thus, instead of training a DL model from scratch, one can reuse a pre-trained model to 1) transfer features for a target task or 2) adapt the parameters of the DL model for the target task, instead of training a model from scratch. DTL is a crucial paradigm for democratizing DL. Thus, the DL community is investing in creating highly reusable pre-trained models, also called foundation models [76], for various domains. We provide more details about feature transfer and model adaptation workloads in chapter 8 and chapter 9, respectively, and also dive into our techniques for optimizing these workloads.

Chapter 3

CEREBRO: Query Optimizations for DL Model Selection

3.1 Introduction

In this chapter, we dive deeper into our techniques for optimizing distributed DL model selection. Model selection is an unavoidable step when training DL models where practitioners have to evaluate several different models configurations before picking the best model for their given task. Model selection significantly amplifies the DL model training costs and makes it a major bottleneck for adopting DL. Thus optimizing DL model selection from a systems standpoint is of major importance for democratizing DL.

Case Study. We present a real-world model selection scenario. Our public health collaborators at UC San Diego wanted to try DL models for identifying different activities (e.g., sitting, standing, stepping, etc.) of subjects from body-worn accelerometer data. The data were collected from a cohort of about 600 people and is labeled. Its size is 864 GB. During model selection, we tried different model architectures such as convolution neural networks (CNNs), long short-term memory models (LSTMs), and composite models such as CNN-LSTMs,

which now offer state-of-the-art results for multivariate time-series classification [220, 161]. Our collaborators also wanted to try different prediction window sizes (e.g., predictions generated every 5 seconds vs. 15 seconds) and alternative target semantics (e.g., sitting–standing–stepping or sitting vs. not sitting). The training process also involves tuning various hyper-parameters such as learning rate and regularization coefficient.

In the above scenario, it is clear that the model selection process generates dozens, if not hundreds, of different models that need to be evaluated in order to pick the best one for the prediction task. Due to the scale of the data and the complexity of the task, it is too tedious and time-consuming to manually steer this process by trying models one by one. Parallel execution on a cluster is critical for reasonable runtimes. Moreover, since our collaborators often changed the time windows and output semantics for health-related analyses, we had to rerun the whole model selection process over and over several times to get the best accuracy for their evolving task definitions. Finally, reproducible model training is also a key requirement in such scientific settings. All this underscores the importance of automatically scaling DL model selection on a cluster with high throughput.

System Desiderata. We identify the following key desiderata for a DL model selection system.

1) Scalability. DL often has large training datasets, larger than single-node memory and sometimes even disk. DL model selection is also highly compute-intensive. Thus, we desire out-of-the-box scalability to a cluster with large partitioned datasets (*data scalability*) and distributed execution (*compute scalability*).

2) High Throughput. Regardless of manual grid/random searches or more complex AutoML searches, a key bottleneck for model selection is *throughput*: how many training configurations are evaluated per unit time. Higher throughput enables DL users to iterate through more configurations in bulk, potentially reaching a better accuracy sooner.

3) Overall Resource Efficiency. DL training uses variants of mini-batch stochastic gradient descent (SGD) [71, 77, 79]. To improve efficiency, the model selection system has to avoid wasting resources and maximize resource utilization for executing SGD on a cluster. We identify 4 key components of resource efficiency: (1) *per-epoch efficiency*: time to complete an epoch of training; (2) *convergence efficiency*: time to reach a given accuracy metric; (3) *memory/storage efficiency*: amount of memory/storage used by the system; and (4) *communication efficiency*: the amount of network bandwidth used by the system. In cloud settings, compute, memory/storage, and network all matter for overall costs because resources are pay-as-you-go; on shared clusters, which are common in academia, wastefully hogging any resource is unethical.

4) Reproducibility. Ad hoc model selection with distributed training is a key reason for the "reproducibility crisis" in deep learning [280]. While some Web giants may not care about reproducibility for some use cases, this is a showstopper issue for many enterprises due to auditing, regulations, and/or other legal reasons. Most domain scientists also inherently value reproducibility.

Limitations of Existing Landscape. We compared existing approaches to see how well they cover the above desiderata. Unfortunately, each approach falls short on some major desiderata, as we summarize next. Figure 3.3 and Section 3.2 present our analysis in depth.

1) False Dichotomy of Task- and Data-Parallelism. Prior work on model selection systems almost exclusively focus on the task-parallel setting [186, 185, 149]. This ignores a pervasive approach to scale to large data on clusters: data partitioning (sharding). A disjoint line of work on data-parallel DL systems do consider partitioned data but focus on training one model at a time, not model selection workloads [252, 187]. Model selection on partitioned datasets is important because parallel file systems (e.g., HDFS for Spark), parallel RDBMSs, and "data lakes" typically store large datasets in that manner.

2) Resource Inefficiencies. Due to the false dichotomy, naively combining the above

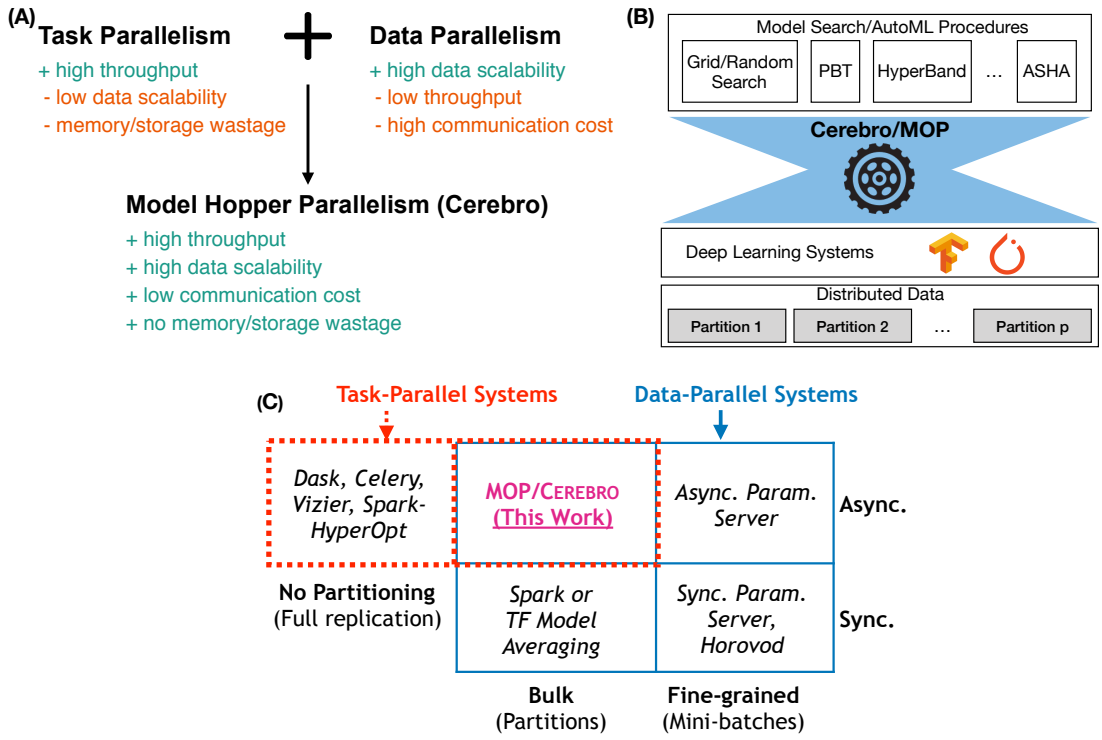


Figure 3.1: (A) Cerebro combines the advantages of both task- and data-parallelism. (B) System design philosophy and approach of CEREBRO/MOP (introduced in [214]): "narrow waist" architecture in which multiple model selection procedures and multiple deep learning tools are supported–unmodified–for specifying/executing DL computations. MOP is our novel resource-efficient distributed SGD execution approach. (C) Model Hopper Parallelism (MOP) as a hybrid approach of task- and data-parallelism. It is the first known form of *bulk asynchronous* parallelism, filling a major gap in the parallel data systems literature.

mentioned approaches could cause overheads and resource wastage (Section 3.2 explains more). For instance, using task-parallelism on HDFS requires extra data movement and potential caching, substantially wasting network and memory/storage resources. An alternative is remote data storage (e.g., S3) and reading repeatedly at every iteration of SGD. But this leads to orders of magnitude higher network costs by flooding the network with lots of redundant data reads. On the other hand, data-parallel systems that train one model at a time (e.g., Horovod [252] and Parameter Servers [187]) incur high communication costs, leading to high runtimes.

Overall, we see a major gap between task- and data-parallel systems today, which leads to substantially lower overall resource efficiency, i.e., when compute, memory/storage, and network

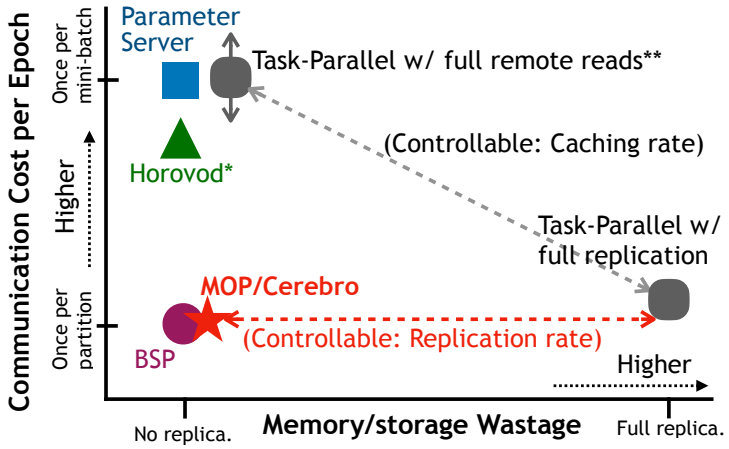


Figure 3.2: Conceptual comparison of MOP/CEREBRO with prior art on two key axes of resource efficiency: communication cost per epoch and memory/storage wastage. Dashed line means that approach has a controllable parameter. *Horovod uses a more efficient communication mechanism than Parameter Server (PS), leading to a relatively lower communication cost. **Task-Parallelism with full remote reads has varying communication costs (higher or lower than PS) based on dataset size.

are considered holistically.

Our Proposed System. We propose CEREBRO, a new system for DL model selection that mitigates the above issues with both task- and data-parallel execution. As Figure 3.1(A) shows, CEREBRO combines the advantages of both task- and data-parallelism, while avoiding the limitations of each. It raises model selection throughput without raising resource costs. Our target setting is *small clusters* (say, tens of nodes), which covers a vast majority (over 90%) of parallel ML workloads in practice [224]. We focus on the common setting of partitioned data on such clusters. Figure 3.1(B) shows the system design philosophy of CEREBRO: a narrow-waist architecture inspired by [173] to support multiple model selection search procedures and DL systems.

Summary of Our Techniques. At the heart of CEREBRO is a simple but novel hybrid of task- and data-parallelism we call *model hopper parallelism* (MOP) that fulfills all of our desiderata. MOP is inspired by the multi-query optimization (MQO) [250] in DBMSs and is based on our insight about a formal optimization theoretic property of SGD: *robustness to the random*

ordering of the data. Figure 3.1(C) positions MOP against prior approaches: it is the first known form of "Bulk Asynchronous" parallelism, a hybridization of the Bulk Synchronous parallelism common in the database world and task-parallelism common in the DL world. As Figure 3.2 shows, MOP has the network and memory/storage efficiency of BSP but offers much better model convergence behavior. Prior work has shown that the BSP approach for distributed SGD (also called "model averaging") has poor convergence behavior [116]. Overall, *considering all resources holistically–compute, memory/storage, and network–MOP can be the resource-optimal choice* in our target setting.

With MOP as its basis, CEREBRO devises an *optimizing scheduler* to efficiently execute DL model selection on small clusters. We formalize our scheduling problem as a mixed-integer linear program (MILP). We compare alternate candidate algorithms with simulations and find that a simple randomized algorithm has surprisingly good performance on all aspects (Section 3.5). We then extend our scheduler to support replication of partitions, fault tolerance, and elasticity out of the box (Sections 3.5.5 and 3.5.6). Such systems-level features are crucial for DL model selection workloads, which can often run for days. Overall, this work makes the following contributions:

- We present a new parallel SGD execution approach we call *model hopper parallelism* (MOP) that satisfies all the desiderata listed earlier by exploiting a formal property of SGD. MOP is applicable to *any* ML models trained with SGD. We focus primarily on DL models due to their growing popularity combined with the pressing issue of their resource-intensiveness.

- We build CEREBRO, a general and extensible DL model selection system using MOP. CEREBRO can support arbitrary DL models and data types, as well as multiple DL training frameworks and model selection search procedures. We integrate it with TensorFlow and PyTorch.

- We formalize the scheduling problem of CEREBRO and compare 3 alternatives (MILP solver, approximate, and randomized) using simulations. We find that a randomized scheduler works well in our setting.
- We extend CEREBRO to use partial data replication and support fault tolerance and elasticity.
- We perform extensive experiments on real model selection workloads with two large benchmark datasets: *ImageNet* and *Criteo*. CEREBRO offers 3x to 10x runtime gains over purely data-parallel systems and up to 8x memory/storage gains over purely task-parallel systems. CEREBRO also exhibits linear speedup behavior.

3.2 Tradeoffs of Existing Approaches

Most deep learning tools (e.g., TensorFlow) focus on the latency of training *one model at a time*, not on throughput. A popular way to raise throughput is *parallelism*. Thus, various multi-node parallel execution approaches have been studied. All of them fall short on some desiderata, as Figure 3.3 shows. We group these approaches into 4 categories:

Embarrassingly Task Parallel. Tools such as Python Dask, Celery, Vizier [122], and Ray [207] can run different training configurations on different workers in a task-parallel manner. Each worker can use logically sequential SGD, which yields the best convergence efficiency. This is also reproducible. There is no communication across workers during training, but the whole dataset must be copied to each worker, which does not scale to large partitioned datasets. Copying datasets to all workers is also *highly wasteful of resources*, both memory and storage, which raises costs. Alternatively, one can use remote storage (e.g., S3) and read data remotely every epoch. But such repeated reads wastefully flood the network with orders of magnitude extra redundant data, e.g., see a realistic cost calculation in Table 3.2.

Bulk Synchronous Parallel (BSP). BSP systems such as Spark and TensorFlow with

Desiderata	Embarrassing Task Parallelism (e.g., Dask, Celery,	Data Parallelism			Model Hopper Parallelism (Our Work)
		Bulk Synchronous (e.g., Spark)	Centralized Fine-grained (e.g., Async PS)	Decentralized Fine-grained (e.g., Horovod)	
Data Scalability	✗ No (Full Replication) Wasteful (Remote Reads)	✓ Yes	✓ Yes	✓ Yes	✓ Yes
Per-Epoch Efficiency	✓ High	✓ High	✗✗ Lowest	✗ Low	✓ High
SGD Conv. Efficiency	✓✓ Highest	✗✗ Lowest	↔ Medium	✓ High	✓✓ Highest
Mem/Storage Efficiency	✗✗ Lowest	✓ High	✓ High	✓ High	✓ High
Reproducibility	✓ Yes	✓ Yes	✗ No	✓ Yes	✓ Yes

Figure 3.3: Qualitative comparisons of existing systems on key desiderata for a model selection system.

model averaging [50] parallelize one model at a time. They partition the dataset across workers, yielding high memory/storage efficiency. They broadcast a model, train models independently on each worker's partition, collect all models on the master, average the weights (or gradients), and repeat this every epoch. Alas, this approach converges poorly for highly non-convex models; so, it is almost never used for DL training [265].

Centralized Fine-grained. These systems also parallelize one model at a time on partitioned data but at the finer granularity of each mini-batch. The most prominent example is Parameter Server (PS) [187]. PS is a set of systems for data-parallel DL model training. A typical PS consists of *servers* and *workers*; servers maintain the globally shared model weights, while workers compute SGD gradients on a locally stored data partition. Workers communicate with servers periodically to update and retrieve model weights. Based on the nature of these communications, PS has two variants: *synchronous* and *asynchronous*. Asynchronous PS is highly scalable but unreproducible; it often has poorer convergence than synchronous PS due to stale updates but synchronous PS has higher overhead for synchronization. All PS-style approaches have *high communication* due to their centralized all-to-one communications, which is proportional to the number of mini-batches and orders of magnitude higher than BSP.

Decentralized Fine-grained. The best example is Horovod [252]. It adopts HPC-style techniques to enable synchronous all-reduce SGD. While this approach is bandwidth optimal,

Table 3.1: Notation used in Section 3.3

Symbol	Description
S	Set of training configurations
p	Number of data partitions/workers
k	Number of epochs for S to be trained
m	Model size (uniform for exposition sake)
b	Mini-batch size
D	Training dataset ($\langle D \rangle$: dataset size, $\|D\|$: number of examples)

communication latency is still proportional to the number of workers, and the synchronization barrier can become a bottleneck. The total communication overhead is also proportional to the number of mini-batches and orders of magnitude higher than BSP.

3.3 Model Hopper Parallelism

We first explain how MOP works and its properties. Table 3.1 presents some notation. We also theoretically compare the communication costs of MOP and prior approaches.

3.3.1 Basic Idea of MOP

We are given a set S of training configurations ("configs" for short). For simplicity of exposition, assume for now each runs for k training iterations, also called k epochs–we relax this later[1]. Shuffle the dataset once and split into p partitions, with each partition located on one of p worker machines. Given these inputs, MOP works as follows. Pick p configs from S and assign one per worker (Section 3.5 explains how we pick the subset). On each worker, the assigned config is trained on the local partition for a single *sub-epoch*, which we also call a *training unit*. Completing a training unit puts that worker back to the idle state. An idle worker is

[1] Section 3.4.2 (Supporting Multiple AutoML Procedures) explains further how CEREBRO can support different configs being trained for different numbers of epochs.

then assigned a new config that has not already been trained and also not being currently trained on another worker. Overall, a model "hops" from one worker to another after a sub-epoch. Repeat this process until all configs are trained on all partitions, completing one epoch for each model. Repeat this every epoch until all configs in S are trained for k epochs. The invariants of MOP can be summarized as follows:

- *Completeness:* In a single epoch, each config is trained on all workers exactly once.
- *Model training isolation:* Two training units of the same config are not run simultaneously.
- *Worker/partition exclusive access:* A worker executes only one training unit at a time.
- *Non-preemptive execution:* Once started, individual training units run without preemption.

Insights Underpinning MOP. MOP exploits a formal property of SGD: *any random ordering* of examples suffices for convergence [71, 77]. Each of the p configs visits the data partitions in a different (pseudorandom) yet in sequential order. Thus, MOP offers high accuracy for all models, comparable to sequential SGD. While SGD's robustness has been exploited before in DL systems, e.g., in Parameter Server [187], MOP exploits it at the *partition level* instead of at the mini-batch level to reduce communication costs. This is possible because we connect this property with model selection workloads instead of training one model at a time.

Positioning MOP. As Figure 3.1(C) shows, MOP is a new hybrid of task- and data-parallelism that is a form of "bulk asynchronous" parallelism. Like task-parallelism, MOP trains many configs in parallel but like BSP, it runs on partitions. So, MOP is more fine-grained than task parallelism but more coarse-grained than BSP. MOP has no global synchronization barrier within an epoch. Later in Section 3.5, we dive into how CEREBRO uses MOP to schedule S efficiently and in a general way. Overall, while the core idea of MOP is simple–perhaps even obvious in hindsight–it has hitherto not been exploited in its full generality in DL systems.

Table 3.2: Communication cost analysis of MOP and other approaches. *Full replication. †Remote reads. ‡Parameters for the example: k = 20, $|S|$ = 20, p = 10, m = 1GB, $\langle D \rangle$ = 1TB, and $|D|/b$ = 100K.

	Comm. Cost	**Example**‡
Model Hopper Parallelism	$kmp\|S\| + m\|S\|$	4 TB
Task Parallelism (FR*)	$p\langle D \rangle + m\|S\|$	10 TB
Task Parallelism (RR†)	$k\|S\|\langle D \rangle + m\|S\|$	400 TB
Bulk Synchronous Parallelism	$2kmp\|S\|$	8 TB
Centralized Fine-grained	$2kmp\|S\| \left\lceil \frac{\|D\|}{bp} \right\rceil$	80 PB
Decentralized Fine-grained	$2km(p-1)\|S\| \left\lceil \frac{\|D\|}{bp} \right\rceil$	72 PB

Reproducibility. MOP does not restrict the visit ordering. So, reproducibility is trivial in MOP: log the worker visit order for each configuration per epoch and replay with this order. Crucially, this logging incurs very negligible overhead because a model hops only *once per partition*, not for every mini-batch, at each epoch.

3.3.2 Communication Cost Analysis

We summarize the communication costs of MOP and other approaches in Table 3.2. It also illustrates the communication costs in bytes for a realistic example based on our model selection case study in Section 3.1. MOP reaches the theoretical minimum cost of $kmp|S|$. Crucially, note that this cost does not depend on batch size, which underpins MOP's higher efficiency. BSP also has the same asymptotic cost but unlike MOP, BSP typically converges poorly for DL models and lacks sequential-equivalence. Fine-grained approaches like PS and Horovod have communication costs proportional to the number of mini-batches, which can be orders of magnitude higher. In our setting, p is under low 10s, but the number of mini-batches can even be 1000s to millions based on the batch size.

3.4 System Overview

We present an overview of CEREBRO, a DL system that uses MOP to execute DL model selection workloads.

3.4.1 User-facing API

CEREBRO API allows users to do 2 things: (1) register workers and data; and (2) issue a DL model selection workload. Workers are registered by IP addresses. As for datasets, CEREBRO expects a list of data partitions and their availability on each worker. We assume shuffling and partitioning are already handled by other means, since these are well studied. This common data ETL step is also orthogonal to our focus and is not a major part of the total runtime for iterative DL training.

CEREBRO takes the reference to the dataset, set of initial training configs, the AutoML procedure, and 3 user-defined functions: $input\_fn$, $model\_fn$, and $train\_fn$. It first invokes $input\_fn$ to read and pre-process the data. It then invokes $model\_fn$ to instantiate the neural architecture and potentially *restore* the model state from a previous *checkpointed* state. The $train\_fn$ is invoked to perform one sub-epoch of training. We assume validation data is also partitioned and use the same infrastructure for evaluation. During evaluation, CEREBRO marks model parameters as non-trainable before invoking $train\_fn$. We also support higher-level API methods for AutoML procedures that resemble the popular APIs of Keras [221]. Note that $model\_fn$ is highly general, i.e., CEREBRO supports *all* neural computational graphs on all data types supported by the underlying deep learning tool, including CNNs, RNNs, transformers, etc. on structured data, text, images, video, etc. More details of our APIs, including full method signatures and a fleshed out example of how to use CEREBRO are provided in the Appendix A.1.

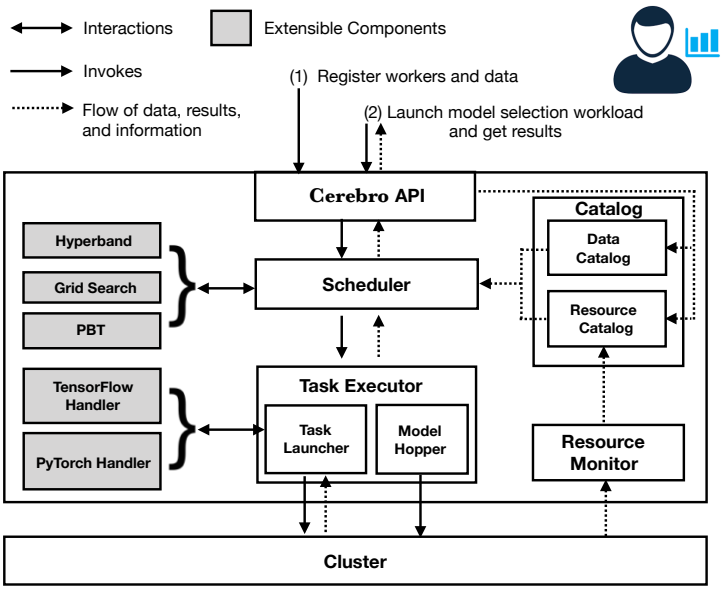


Figure 3.4: System architecture of CEREBRO.

3.4.2 System Architecture

We adopt an extensible architecture, as Figure 3.4 shows. This allows us to easily support multiple deep learning tools and AutoML procedures. There are 5 main components: (1) API, (2) Scheduler, (3) Task Executor, (4) Catalog, and (5) Resource Monitor. Scheduler is responsible for orchestrating the entire workload. It relies on worker and data availability information from the Catalog. Task Executor launches training units on the cluster and also handles model hops. Resource Monitor is responsible for detecting worker failures and updating the Resource Catalog. Section 3.5 explains how the Scheduler works and how we achieve fault tolerance and elasticity. Next, we describe how CEREBRO's architecture enables high system generality.

Supporting Multiple Deep Learning Tools. The functions $input\_fn$, $model\_fn$, and $train\_fn$ are written by users in the deep learning tool's APIs. We currently support TensorFlow and PyTorch (it is simple to add support for more). To support multiple such tools, we adopt a handler-based architecture to delineate tool-specific aspects: model training, checkpointing and restoring. Note that checkpointing and restoring is how CEREBRO realizes model hops. Task

Executor automatically injects the tool-specific aspects from the corresponding tool's handler and runs these functions on the workers. Overall, CEREBRO's architecture is highly general and supports virtually all forms of data types, DL model architectures, loss functions, and SGD-based optimizers.

Supporting Multiple AutoML Procedures Metaheuristics called AutoML procedures are common for exploring training configs. We now make a key observation about such procedures that underpins our Scheduler. Most AutoML procedures fit a *common template*: create an initial set of configs (S) and evaluate them after each epoch (or every few epochs). Based on the evaluations, terminate some configurations (e.g., as in Hyperband [186] and PBT [149]) or add new configurations (e.g., as in PBT). Grid/random search is a one-shot instance of this template. Thus, we adopt this template for our Scheduler. Given S, CEREBRO trains all models in S for one epoch and passes control back to the corresponding AutoML procedure for convergence/termination/addition evaluations. CEREBRO then gets a potentially modified set S' for the next epoch. This approach also lets CEREBRO support data re-shuffling after each epoch. But the default (and common practice) is to shuffle only once upfront. Grid/random search (perhaps the most popular in practice), Hyperband, and PBT (and more procedures) conform to this common template and are currently supported.

ASHA [185] and Hyperopt [70] are two notable exceptions to the above template, since they do not have a global synchronized evaluation of training configs after an epoch and are somewhat tied to task-parallel execution. While MOP/CEREBRO cannot ensure logically same execution as ASHA or HyperOpt on task-parallelism, it is still possible to emulate them on MOP/CEREBRO without any modifications to our system. In fact, our experiments with ASHA show that ASHA on CEREBRO has comparable–even slightly better!–convergence behavior than ASHA on pure task-parallelism (Section 3.6.3).

Table 3.3: Additional notation used in the MOP MILP formulation

Symbol	Description
$T \in \mathbb{R}^{\lvert S\rvert \times p}$	$T_{i,j}$ is the runtime of unit $s_{i,j}$ (i^{th} config on j^{th} worker)
C	Makespan of the workload
$X \in \mathbb{R}^{\lvert S\rvert \times p}$	$X_{i,j}$ is the start time of the execution of i^{th} config on j^{th} partition/worker
$Y \in \{0,1\}^{\lvert S\rvert \times p \times p}$	$Y_{i,j,j'} = 1 \iff X_{i,j} < X_{i,j'}$
$Z \in \{0,1\}^{\lvert S\rvert \times \lvert S\rvert \times p}$	$Z_{i,i',j} = 1 \iff X_{i,j} < X_{i',j}$
V	Very large value (Default: sum of training unit runtimes)

3.4.3 System Implementation Details

We prototype CEREBRO in Python using XML-RPC client-server package. Scheduler runs on the client. Each worker runs a single service. Scheduling follows a push-based model– Scheduler assigns tasks and periodically checks the responses from the workers. We use a shared network file system (NFS) as the central repository for models. Model hopping is realized implicitly by workers writing models to and reading models from this shared file system. Technically, this doubles the communication cost of MOP to $2kmp|S|$, still a negligible overhead. Using NFS greatly reduces engineering complexity to implement model hops.

3.5 Cerebro Scheduler

Scheduling training units on workers properly is critical because pathological orderings can under-utilize resources substantially, especially when DL model architectures and/or workers are heterogeneous. Consider the model selection workload shown in Figure 3.5(A). Assume workers are homogeneous and there is no data replication. For one epoch of training, Figure 3.5(B) shows an optimal task-parallel schedule for this workload with a 9-unit makespan. Figure 3.5(C) shows a non-optimal MOP schedulewith also 9 units makespan. But as Figure 3.5(D) shows, an optimal MOP schedule has a makespan of only 7 units. Overall, we see that MOP's training

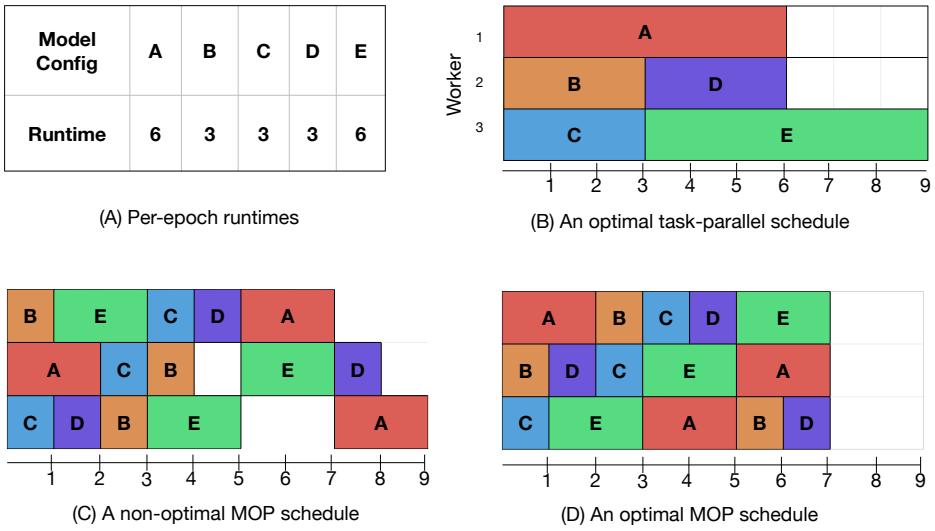


Figure 3.5: Gantt charts of task-parallel and MOP schedules for a sample model selection workload.

unit-based scheduling offers more flexibility to raise resource utilization. Next, we formally define the MOP-based scheduling problem and explain how we design our Scheduler.

3.5.1 Formal Problem Statement as MILP

Suppose the runtimes of each training unit, aka *unit times*, are given. These can be obtained with, say, a pilot run for a few mini-batches and then extrapolating (this overhead will be marginal). For starters, assume each of the p data partitions is assigned to only one worker. The objective and constraints of the MOP-based scheduling problem is as follows. Table 3.3 lists the additional notation used here.

$$\texttt{Objective:} \quad \min_{C,X,Y,Z} C \tag{3.1}$$

`Constraints:`

$$
\begin{aligned}
&\forall i, i' \in [1, \ldots, |S|]\ \forall j, j' \in [1, \ldots, p] \\
&(a)\ X_{i,j} \geq X_{i,j'} + T_{i,j'} - V \cdot Y_{i,j,j'} \\
&(b)\ X_{i,j'} \geq X_{i,j} + T_{i,j} - V \cdot (1 - Y_{i,j,j'}) \\
&(c)\ X_{i,j} \geq X_{i',j} + T_{i',j} - V \cdot Z_{i,i',j} \\
&(d)\ X_{i',j} \geq X_{i,j} + T_{i,j} - V \cdot (1 - Z_{i,i',j}) \\
&(e)\ X_{i,j} \geq 0 \\
&(f)\ C \geq X_{i,j} + T_{i,j}
\end{aligned}
\tag{3.2}
$$

We need to minimize makespan C, subject to the constraints on C, unit start times X, model training isolation matrix Y, and worker/partition exclusive access matrix Z. The constraints enforce some of the invariants of MOP listed in Section 3.3. Equations 2.a and 2.b ensure model training isolation. Equations 2.c and 2.d ensure worker exclusive access. Equation 2.e ensures that training unit start times are non-negative and Equation 2.f ensures that C captures the time taken to complete all training units.

Given the above, a straightforward approach to scheduling is to use an MILP solver like Gurobi [134]. The start times X then yield the actual schedule. But our problem is essentially an instance of the classical open-shop scheduling problem, which is known to be `NP-Hard` [123]. Since $|S|$ can even be 100s, MILP solvers may be too slow (more in Section 3.5.4); thus, we explore alternative approaches.

3.5.2 Approximate Algorithm-based Scheduler

For many special cases, there are algorithms with good approximation guarantees that can even be optimal under some conditions. One such algorithm is "vector rearrangement" [283, 117]. It produces an optimal solution when $|S| \gg p$, which is possible in our setting.

The vector rearrangement based method depends on two values: L_{max} (see Equation 3.3), the maximum load on any worker; and T_{max} (see Equation 3.4), the maximum unit time of any training configuration in S.

$$L_{max} = \max_{j \in [1,...,p]} \sum_{i=1}^{|S|} T_{i,j} \tag{3.3}$$

$$T_{max} = \max_{i \in [1,...,|S|], j \in [1,...,p]} T_{i,j} \tag{3.4}$$

If $L_{max} \geq (p^2 + p - 1) \cdot T_{max}$, this algorithm's output is optimal. When there are lots of configs, the chance of the above constraint being satisfied is high, yielding us an optimal schedule. But if the condition is not met, the schedule produced yields a makespan $C \leq C^* + (p-1) \cdot T_{max}$, where C^* is the optimal makespan value. This algorithm scales to large $|S|$ and p because it runs in polynomial time in contrast to the MILP solver. For more details on this algorithm, we refer the interested reader to [283, 117].

3.5.3 Randomized Algorithm-based Scheduler

The approximate algorithm is complex to implement in some cases, and its optimality condition may be violated often. Thus, we now consider a much simpler scheduler based on *randomization*. This approach is simple to implement and offer much more flexibility (explained more later). Algorithm 1 presents our randomized scheduler.

Given S, create $Q = \{s_{i,j} : \forall i \in [1,...,|S|], j \in [1,..,p]\}$, the set of all training units. Note that $s_{i,j}$ is the training unit of configuration i on worker j. Initialize the state of all models and workers to idle state. Then find an idle worker and schedule a random training unit from Q on it. This training unit must be such that its configuration is not scheduled on another worker and it corresponds to the data partition placed on that worker (Line 10). Then remove the chosen training unit from Q. Continue this process until no worker is idle and eventually, until Q is

Algorithm 1 Randomized Scheduling

Input: S
$Q = \{s_{i,j} : \forall i \in [1,\ldots,|S|], \forall j \in [1,\ldots,p]\}$
worker_idle $\leftarrow$ [true,...,true]
model_idle $\leftarrow$ [true,...,true]
while not empty(Q) **do**
  for $j \in [1,\ldots,p]$ **do**
    if worker_idle[j] **then**
      $Q \leftarrow$ shuffle(Q)
      for $s_{i,j'} \in Q$ **do**
        if model_idle[i] **and** $j' = j$ **then**
          Execute $s_{i,j'}$ on worker j
          model_idle[i] $\leftarrow$ false
          worker_idle[j] $\leftarrow$ false
          remove($Q, s_{i,j'}$)
          break
  wait WAIT_TIME

Algorithm 2 When $s_{i,j}$ finishes on worker j

model_idle[i] $\leftarrow$ true
worker_idle[j] $\leftarrow$ true

empty. After a worker completes training unit $s_{i,j}$ mark its model i and worker j as idle again as per Algorithm 2.

3.5.4 Comparing Different Scheduling Methods

We use simulations to compare the efficiency and makespans yielded by the three alternative schedulers. The MILP and approximate algorithm are implemented using Gurobi. We set a maximum optimization time of 5min for tractability sake. We compare the scheduling methods on 3 dimensions: 1) number of training configs (two values: 16 and 256), 2) number of workers (two values: 8 and 16), 3) homogeneity/heterogeneity of configs and workers.

Sub-epoch training time (unit time) of a training config is directly proportional to the compute cost of the config and inversely proportional to compute capacity of the worker. For the homogeneous setting, we initialize all training config compute costs to be the same and also

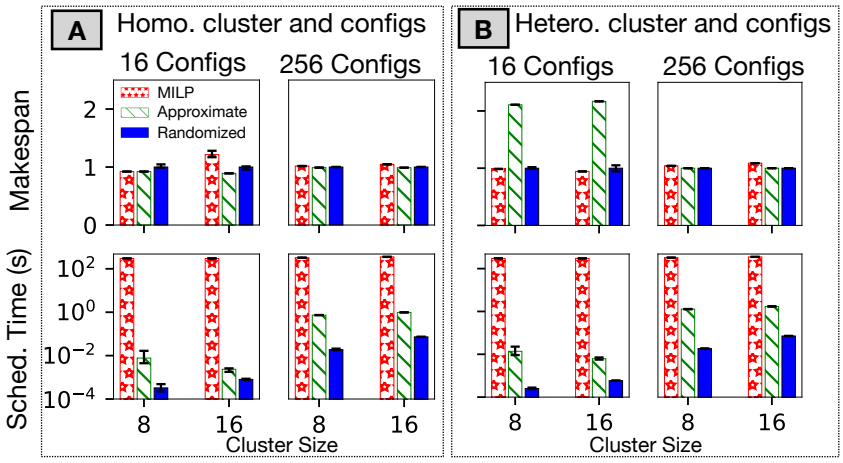


Figure 3.6: Scheduler runtimes and makespans of the schedules produced in different settings. Makespans are normalized with respect to that of Randomized. (A) Homogeneous cluster and homogeneous training configs. (B) Heterogeneous cluster and heterogeneous training configs.

all worker compute capacities to be the same. For the heterogeneous setting, training config compute costs are randomly sampled (with replacement) from a set of popular deep CNNs (n=35) obtained from [60]. The costs vary from 360 MFLOPS to 21000 MFLOPS with a mean of 5939 MFLOPS and standard deviation of 5671 MFLOPS. We provide these computational costs in the Appendix A.2. For worker compute capacities, we randomly sample (with replacement) compute capacities from 4 popular Nvidia GPUs: Titan Xp (12.1 TFLOPS/s), K80 (5.6 TFLOPS/s), GTX 1080 (11.3 TFLOPS/s), and P100 (18.7 TFLOPS/s). For each setting, we report the average of 5 runs with different random seeds set to the scheduling algorithms and also the min and max of all 5 runs. All makespans reported are normalized by the randomized scheduler's makespan.

The MILP scheduler sometimes performs poorer than the other two because it has not converged to the optimal in the given time budget. The approximate scheduler performs poorly when both the configs and workers are heterogeneous. It is also slower than the randomized scheduler.

Overall, the randomized approach works surprisingly well on all aspects: near-optimal makespans with minimal variance across runs and very fast scheduling. We believe this interesting superiority of the randomized algorithm against the approximation algorithm is due to some

fundamental characteristics of DL model selection workloads, e.g., large number of configurations and relatively low differences in compute capacities. We leave a thorough theoretical analysis of the randomized algorithm to future work. Based on these results, we use the randomized approach as the default Scheduler in CEREBRO.

3.5.5 Replica-Aware Scheduling

So far we assumed that a partition is available on only one worker. But some file systems (e.g., HDFS) often replicate data files, say, for reliability sake. We now exploit such replicas for more scheduling flexibility and faster plans.

The replica-aware scheduler requires an additional input: availability information of partitions on workers (an availability map). In replica-aware MOP, a training configuration need *not* visit all workers. This extension goes beyond open shop scheduling, but it is still `NP-Hard` because the open shop problem is a special case of this problem with a replication factor of one. We extended the MILP scheduler but it only got slower. So, we do not use it and skip its details. Modifying the approximate algorithm is also non-trivial because it is tightly coupled to the open shop problem; so, we skip that too. In contrast, the randomized scheduler can be easily extended for replica-aware scheduling. The only change needed to Algorithm 1 is in Line 10: instead of checking $j' = j$, consult the availability map to check if the relevant partition is available on that worker.

3.5.6 Fault Tolerance and Elasticity

We now explain how we make our randomized scheduler fault tolerant. Instead of just Q, we maintain two data structures Q and Q'. Q' is initialized to be empty. The process in Algorithm 1 continues until both Q and Q' are empty. When a training unit is scheduled, it will be removed from Q as before but now also *added* to Q'. It will be removed from Q' when it successfully

completes its training on the assigned worker. But if the worker fails before the training unit finishes, it will be moved back from Q' to Q. If the data partitions present on the failed worker are also available elsewhere, the scheduler will successfully execute the corresponding training units on those workers at a future iteration of the loop in Algorithm 1.

CEREBRO detects failures via the periodic heart-beat check between the scheduler and workers. Because the trained model states are always checkpointed between training units, they can be recovered and the failed training units can be restarted. Only the very last checkpointed model is needed for the failure recovery and others can be safely deleted for reclaiming storage. The same mechanism can be used to detect availability of new compute resources and support seamless scale-out elasticity in CEREBRO.

3.6 Experimental Evaluation

We empirically validate if CEREBRO can improve overall throughput and efficiency of DL model selection. We then evaluate CEREBRO in depth. Finally, we demonstrate CEREBRO's ability to support multiple AutoML procedures.

Datasets. We use two large benchmark datasets: *ImageNet* [108] and *Criteo* [102]. *ImageNet* is a popular image classification dataset. We choose the 2012 version and reshape the images to 112×112 pixels[2]. *Criteo* is an ad click classification dataset with numeric and categorical features. It is shipped under sparse representation. We one-hot encode the categorical features and densify the data. Only a 2.5% random sample of the dataset is used[2]. Table 3.4. summarizes the dataset statistics.

Workloads. For our first end-to-end test, we use two different neural architectures and grid search for hyper-parameters, yielding 16 training configs for each dataset. Table 3.5 offers

[2]We made this decision only so that all of our experiments can complete in reasonable amount of time. This decision does *not* alter the takeaways from our experiments.

Table 3.4: Dataset details. All numbers are after preprocessing and sampling of the datasets.

Dataset	On-disk size	Count	Format	Class
ImageNet	250 GB	1.2M	HDF5	1000
Criteo	400 GB	100M	TFRecords	Binary

Table 3.5: Workloads. †serialized sizes.

Dataset	Model arch.	Model size/MB†	Batch size	Learning rate	Reg. coeff.	Epochs
ImageNet	{VGG16, ResNet50}	VGG16: 792, ResNet50: 293	{32, 256}	$\{10^{-4}, 10^{-6}\}$	$\{10^{-4}, 10^{-6}\}$	10
Criteo	3-layer NN, 1000+500 hidden units	179	{32, 64, 256, 512}	$\{10^{-3}, 10^{-4}\}$	$\{10^{-4}, 10^{-5}\}$	5

the details. We use Adam [163] as our SGD method. To demonstrate generality, we also present results for HyperOpt and ASHA on CEREBRO in Section 3.6.3.

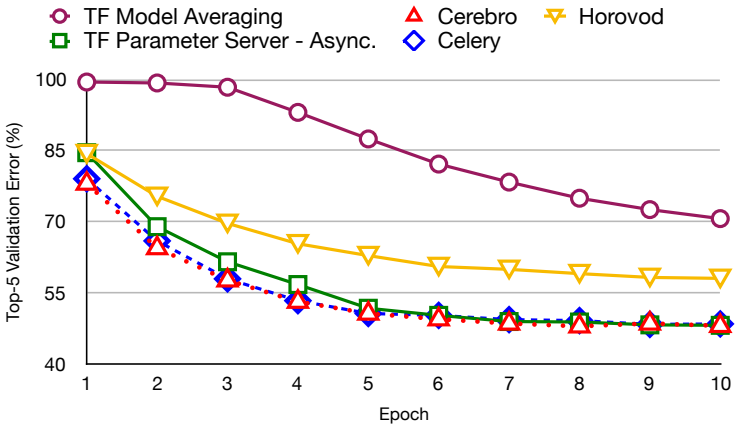
Experimental Setup. We use two clusters: CPU-only for *Criteo* and GPU-enabled for *ImageNet*, both on CloudLab [111]. Each cluster has 8 worker nodes and 1 master node. Each node in both clusters has two Intel Xeon 10-core 2.20 GHz CPUs, 192GB memory, 1TB HDD and 10 Gbps network. Each GPU cluster worker node has an extra Nvidia P100 GPU. All nodes run Ubuntu 16.04. We use TensorFlow v1.12.0 as CEREBRO's underlying deep learning tool. For GPU nodes, we use CUDA version 9.0 and cuDNN version 7.4.2. Both datasets are randomly shuffled and split into 8 equi-sized partitions.

3.6.1 End-to-End Results

We compare CEREBRO with 5 systems: 4 data-parallel–synchronous and asynchronous TensorFlow Parameter Server, Horovod, BSP-style TensorFlow model averaging–and 1 task-parallel (Celery). For Celery, we replicate datasets to each worker beforehand and stream them from disk, since they do not fit in memory. I/O time is trivial for DL models, where computation

System	ImageNet			Criteo		
	Runtime (hrs)	GPU Utili. (%)	Storage Footprint (GB)	Runtime (hrs)	CPU Utili. (%)	Storage Footprint (GB)
TF PS - Async	19.00	8.6	250	28.80	6.9	400
Horovod	5.42	92.1	250	14.06	16.0	400
TF Model Averaging	1.97	72.1	250	3.84	52.2	400
Celery	1.72	82.4	2000	3.95	53.6	3200
Cerebro	1.77	79.8	250	3.40	51.9	400

(A) Per-epoch makespans and CPU/GPU utilization.



(B) Learning curves of the respectively best configs on *ImageNet*.

Figure 3.7: End-to-end results on *ImageNet* and *Criteo*. For Celery, we report the runtime corresponding to the lowest makespan schedule. Celery's per-epoch runtime varies between 1.72-2.02 hours on *ImageNet*; on *Criteo*, 3.95-5.49 hours. Horovod uses GPU kernels for communication; hence high GPU utilization.

dominates; thus, they can be interleaved. We use TensorFlow features to achieve this. For all other systems, each worker node has one in-memory data partition. We do not include data copying in the end-to-end runtimes. For scheduling, Celery uses a FIFO queue and CEREBRO uses the randomized scheduler. All other systems train models sequentially.

Figure 9.6 presents the results. CEREBRO significantly improves the efficiency and throughput of model selection. On *ImageNet*, CEREBRO is over 10x faster than asynchronous PS, which has a GPU utilization as low as 9%! Synchronous PS was even slower. CEREBRO is 3x faster than Horovod. Horovod has high GPU utilization because it uses GPU for communication.

CEREBRO's runtime is comparable to model averaging, which is as expected. But note model averaging converges poorly. Celery's runtime is dependent on the execution order and thus we report the runtime on the optimal schedule. On *ImageNet*, Celery's runtime is comparable to CEREBRO. But note that Celery has a highly bloated 8x memory/storage footprint. Overall, Celery and CEREBRO have the best learning curves–this is also as expected because MOP ensures sequential equivalence for SGD, just like task-parallelism. Horovod converges slower due to its larger effective mini-batch size.

On *Criteo*, CEREBRO is 14x faster than synchronous PS and 8x faster than asynchronous PS. Both variants of PS report severe CPU under-utilization ($< 7\%$). CEREBRO is also 4x faster than Horovod. CEREBRO's runtime is comparable to model averaging, with about 52% CPU utilization. Celery is somewhat slower than CEREBRO due to a straggler issue caused by the highly heterogeneous model configs for *Criteo*. CEREBRO's MOP approach offers higher flexibility to avoid such straggler issues. A more detailed explanation is given in the Appendix A.3. All methods have almost indistinguishable convergence behavior on this dataset: all reached 99% accuracy quickly, since the class label is quite skewed.

Overall, CEREBRO is the *most resource-efficient* approach when compute, memory/storage, and network are considered holistically. It also has the *best accuracy behavior*, on par with task-parallelism.

3.6.2 Drill-down Experiments

Unless specified otherwise, we now show experiments on the GPU cluster, *ImageNet*, and a model selection workload of 8 configs (4 learning rates, 2 regularization values, and ResNet architectures) trained for 5 epochs. Each data partition is placed on only one worker.

Scalability. We study the speedups (strong scaling) of CEREBRO and Horovod as we vary the cluster sizes. Figure 3.8(A) shows the speedups, defined as the workload completion time on multiple workers vs a single worker. CEREBRO exhibits linear speedups due to MOP's

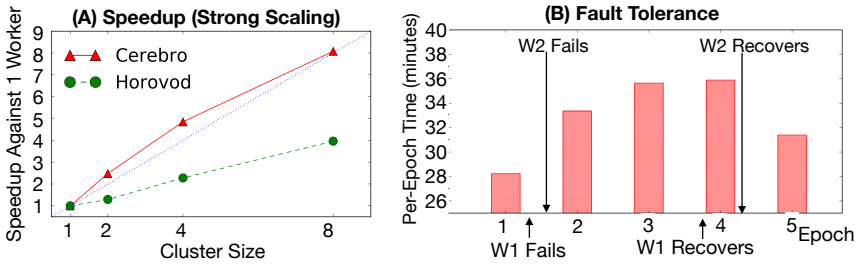


Figure 3.8: (A) Speedup plot (strong scaling). (B) Fault-tolerance.

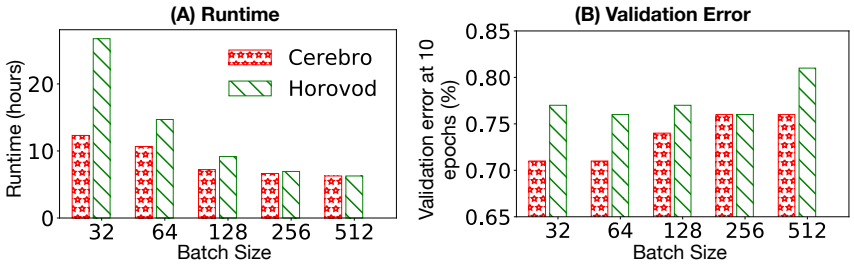


Figure 3.9: Effect of batch size on communication overheads and convergence efficiency. (A) Runtime against batch size. (B) The lowest validation error after 10 epochs against batch size.

marginal communication costs; in fact, it seems slightly super-linear here because the dataset fits entirely in cluster memory compared to the minor overhead of reading from disk on the single worker. In contrast, Horovod exhibits substantially sub-linear speedups due to its much higher communication costs with multiple workers.

Fault Tolerance. We repeat our drill-down workload with a replication factor of 3. We first inject two node failures and bring the nodes back online later. Figure 3.8(B) shows the time taken for each epoch and the points where the workers failed and returned online. Overall, we see CEREBRO's replica-aware randomized scheduler can seamlessly execute the workload despite worker failures.

Effect of Batch Size. We now evaluate the effect of training mini-batch size for CEREBRO and Horovod. We evaluate 5 batch sizes and report makespans and the validation error of the best model for each batch size after 10 epochs. Figure 3.9 presents the results. With batch size 32, Horovod is 2x slower than CEREBRO. However, as the batch size increases, the difference narrows since the relative communication overhead per epoch decreases. CEREBRO also runs

faster with larger batch size due to better hardware utilization. The models converge slower as batch size increases. The best validation error is achieved by CEREBRO with a batch size of 32. With the same setting, Horovod's best validation error is higher than CEREBRO; this is because its effective batch size is 256 (32×8). Horovod's best validation error is closer to CEREBRO's at a batch size of 256. Overall, CEREBRO's efficiency is more stable to the batch size, since models hop per sub-epoch, not per mini-batch.

Network and Storage Efficiency. We study the tradeoff between redundant remote reads (wastes network) vs redundant data copies across workers (wastes memory/storage). Task parallelism forces users to either duplicate the dataset to all workers or store it in a common repository/distributed filesystem and read remotely. CEREBRO can avoid both forms of resource wastage. We assume the whole dataset cannot fit on single-node memory. We compare CEREBRO and Celery in the following 2 settings:

Reading from remote storage (e.g., S3). In this setting, Celery reads data from a remote storage repeatedly each epoch. For CEREBRO each worker remotely reads one data partition and caches it. We change the data scale to evaluate effects on the makespan and the amount of remote reads. Figure 3.10 shows the results. Celery is slightly slower than CEREBRO due to remote read overheads. The most significant advantage of CEREBRO is its network bandwidth cost, which is over 10x lower than Celery's. After the initial read, CEREBRO only communicates models weights during training. In situations where reads and networks are not free (e.g., cloud providers), Celery will incur higher monetary costs than CEREBRO. These results show it is perhaps better to partition the dataset on S3, cache partitions on workers on the first read, and then run CEREBRO instead of Celery with full dataset reads from S3 per epoch to avoid copying.

Reading from distributed storage (e.g., HDFS). In this setting, the dataset is partitioned, replicated, and stored on 8 workers. We then load all local data partitions into each worker's memory. Celery performs remote reads for non-local partitions. We vary the replication factor to study its effect

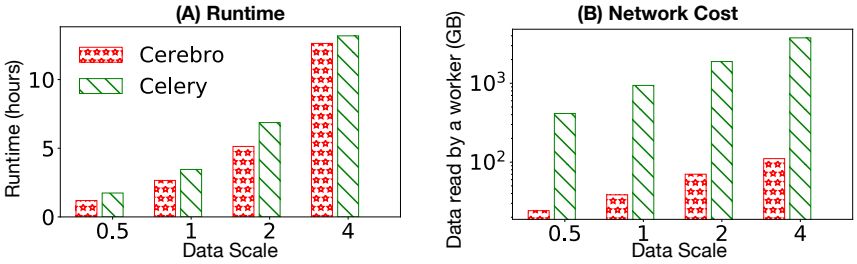


Figure 3.10: Reading data from remote storage.

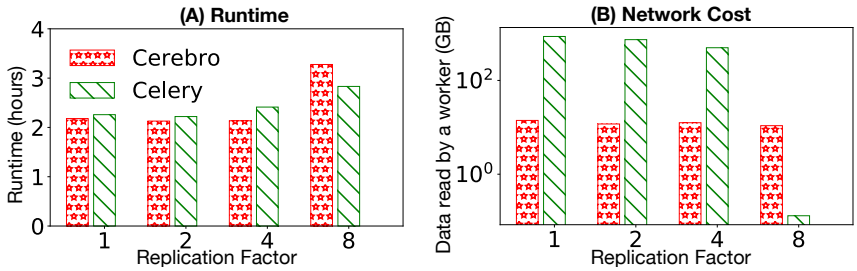


Figure 3.11: Reading data from distributed storage.

on the makespan and the number of remote reads. Figure 3.10 presents the results. For replication factors 1 (no replication), 2, and 4, CEREBRO incurs 100x less network usage and is slightly faster than Celery. But at a replication factor of 8 (i.e., full replication), CEREBRO is slightly slower due to the overhead of model hops. For the same reason, CEREBRO incurs marginal network usage, while Celery has almost no network usage other than control actions. Note that the higher the replication factor for Celery, the more memory/storage is wasted. CEREBRO offers the best overall resource efficiency–compute, memory/storage, and network put together–for DL model selection.

3.6.3 Experiments with AutoML Procedures

We experiment with two popular AutoML procedures: HyperOpt [70] and ASHA [185]. For HyperOpt, we compare CEREBRO and Spark as the execution backends. Spark is a backend supported natively by HyperOpt; it distributes only the models, i.e., it is task-parallel on fully replicated data. For ASHA, we compare CEREBRO and Celery as the execution backends. We

Table 3.6: Parameter grid used to randomly sample configuration for Section 3.6.3.

	Values sampled from
Model	[ResNet18, ResNet34]
Learning rate	$[10^{-5}, \ldots, 10^{-1}]$
Weight decay coefficient	$[10^{-5}, \ldots, 10^{-1}]$
Batch size	$[16, \ldots, 256]$

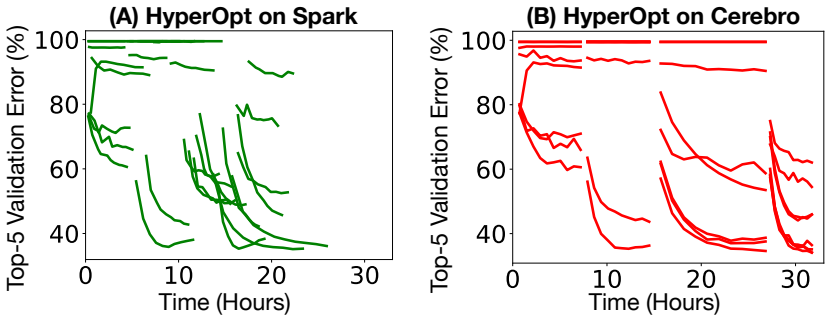


Figure 3.12: HyperOpt learning curves by time.

use *ImageNet*, GPU cluster, and PyTorch. Training configs are sampled from the grid shown in Table 3.6. For CEREBRO data is partitioned without replication; for Spark and Celery the dataset is fully replicated.

Both HyperOpt and ASHA keep exploring different configs until a resource limit is reached. For HyperOpt, this limit is the maximum number of configs; for ASHA, it is the maximum wall-clock time. During the exploration HyperOpt uses Bayesian sampling to generate new configs; ASHA uses random sampling. For both methods, the generated configs are dependent on the completion order of configs across task-parallel workers. Thus, it is impossible for CEREBRO to *exactly* replicate HyperOpt or ASHA ran with task-parallelism. However, we can closely *emulate* HyperOpt and ASHA on CEREBRO by making the number of simultaneously trained configs ($|S|$) equal to the number of workers (p) and without making any changes to CEREBRO.

HyperOpt. We run an experiment using HyperOpt with a max config budget of 32. We train each config for 10 epochs. With this configuration, HyperOpt on CEREBRO (resp. Spark) took 31.8 (resp. 25.9) hours. Figure 3.12 shows all learning curves. We found that the slightly

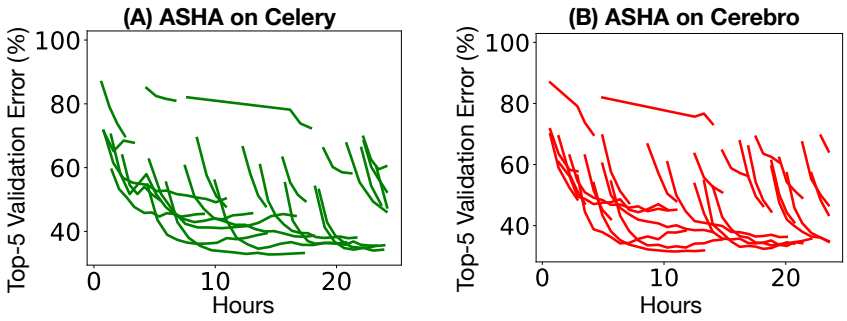


Figure 3.13: ASHA learning curves by time.

higher (23%) runtime of CEREBRO is mainly due to the lower degree of parallelism ($|S| = 8$). However, this issue can be mitigated by increasing the number of simultaneously trained configs. Although individual configs are not comparable across the two systems, the best errors achieved are close (34.1% on CEREBRO; 33.2% on Celery).

ASHA. We use ASHA with a max epoch budget (R) of 9, a selection fraction (η) of 3, and a time limit of 24hr. With these settings, ASHA trains for a maximum of 13 epochs over 3 stages: 1, 3, and 9 epochs. Only the more promising configurations are trained for more epochs. In the given time limit, ASHA on CEREBRO (resp. Celery) explored 83 (resp. 67) configs. Figure 3.13 shows all learning curves. Like HyperOpt, even though the configs are not directly comparable, the best errors achieved are close (31.9% on CEREBRO; 33.2% on Celery). More details about this experiment and experiments with another AutoML procedure (HyperBand) are presented in the Appendix A.4.

3.7 Discussion and Limitations

Open Source Systems. CEREBRO is open sourced and available for download [53]. MOP's generality also enabled us to emulate it on existing data-parallel systems. Pivotal/VMware collaborated with us to integrate MOP into Greenplum by extending the MADlib library [139] for running TensorFlow on Greenplum-resident data [198, 275]. Greenplum's customers are

interested in this for enterprise ML use cases, including language processing, image recognition, and fraud detection. We have also integrated CEREBRO into Apache Spark [106]. CEREBRO-Spark can run MOP on existing resource managers such as YARN and Mesos. Alternatively, one can also deploy CEREBRO as a standalone application by wrapping it as tasks accepted by the resource manager.

Other ML Model Families. We focused primarily on DL due to their growing popularity, high sensitivity to model configurations, and resource intensiveness. However, note that MOP and CEREBRO's ideas are directly usable for model selection of *any* ML models trainable with SGD. Examples include linear/logistic regression, some support vector machines, low-rank matrix factorization, and conditional random fields. In fact, since linear/logistic regression can be trivially expressed in the deep learning tools's APIs, CEREBRO will work out of the box for them. CEREBRO's high memory efficiency makes it easier for users to store the entire large datasets in distributed memory, which can significantly reduce runtimes of such I/O-bound ML models.

3.8 Conclusion

DL model selection is a highly resource-intensive step that is unavoidable when training DL models. The high resource costs of DL model selection often hinder DL adoption by practitioners. In this work, we propose MOP, a multi-query optimization-inspired technique for parallel SGD execution. MOP is a simple and highly general form of parallel SGD execution that raises the resource efficiency of DL model selection without sacrificing accuracy or reproducibility. MOP is also simple to implement, which we demonstrate by building CEREBRO, a fault-tolerant DL model selection system that supports multiple popular DL training systems and model selection procedures. Experiments with large benchmark datasets confirm the benefits of CEREBRO.

Chapter 3 contains material from "Cerebro: A Data System for Optimized Deep Learning

Model Selection" by Supun Nakandala, Yuhao Zhang, and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 13, Issue 12, July 2020. The dissertation author was the primary investigator and author of this paper. The code for our system is open source and is available on GitHub: https://github.com/ADALabUCSD/cerebro-system.

Chapter 4

Applications and Extensions of CEREBRO

4.1 Application: UCSD Public Health Data

4.1.1 Introduction

CEREBRO is being used for time-series analytics for our public health collaborators at UCSD. Our collaborators wanted to identify sitting patterns from hip-worn accelerometers. It has been shown that sitting patterns predict several healthy aging outcomes. These patterns can potentially be measured using hip-worn accelerometers, but current methods have limited ability to detect postural transitions. No off-the-shelf techniques suit our collaborators' task semantics. So, based on the literature on DL-based time series classification, we tried many DL models with TensorFlow, including deep CNNs, LSTMs, and composite CNN-LSTMs.

The size of the raw training dataset for this task was close to a 1 TB. We also performed repeated model selection on a shared GPU cluster at UCSD. Our collaborators also kept changing the prediction windows (e.g., 5s vs. 15s) and label semantics (e.g., sitting vs. not sitting), requiring us to rerun model selection over and over. It also underscores the importance of resource efficiency and the throughput of this process. We also found that existing distributed DL model selection systems scale poorly; they incur high network communication costs or face

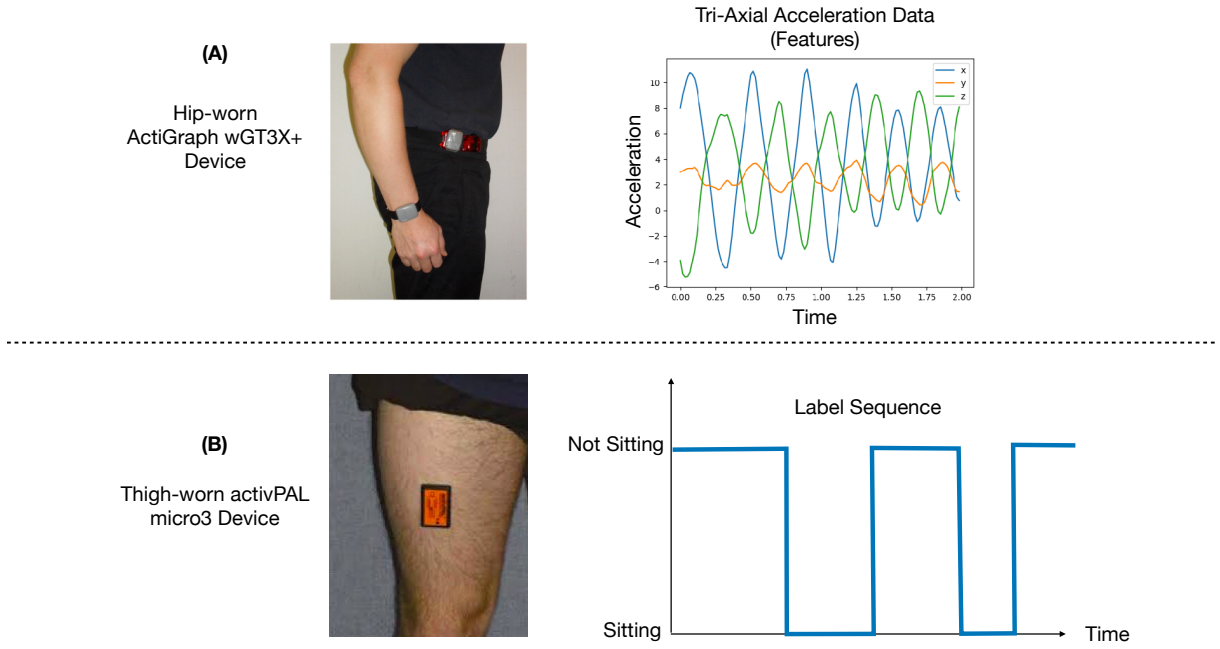


Figure 4.1: Devices used to generate the training data. (A) Hip-worn ActiGraph wGT3X+ device generates a tri-axial acceleration sequence. (B) Thigh-worn activPAL micro3 device generates a sitting vs not-sitting label sequence.

data scalability issues. This use case was a key motivation for building CEREBRO and migrating this workload to CEREBRO. Convolutional Neural Network Hip Accelerometer Posture (CHAP) models are the outcome of this work. They are now the state-of-the-art method for identifying sitting patterns from hip-worn accelerometer data for public health applications.

4.1.2 Training Data

Training data for CHAP models were obtained from the Adult Changes in Thought (ACT) study, an ongoing longitudinal cohort study that maintains an active enrollment of approximately 2,000 older adults (65 y old) in Washington State [243]. The participants were asked to wear a hip-worn ActiGraph wGT3X+ (see Figure 4.1 (A)), activated using ActiLife software to capture 30 Hz triaxial, and a thigh-worn activPAL micro3 (see Figure 4.1 (B)), activated using a 10s minimum threshold for labeling postural transitions. Participants were asked to wear both devices

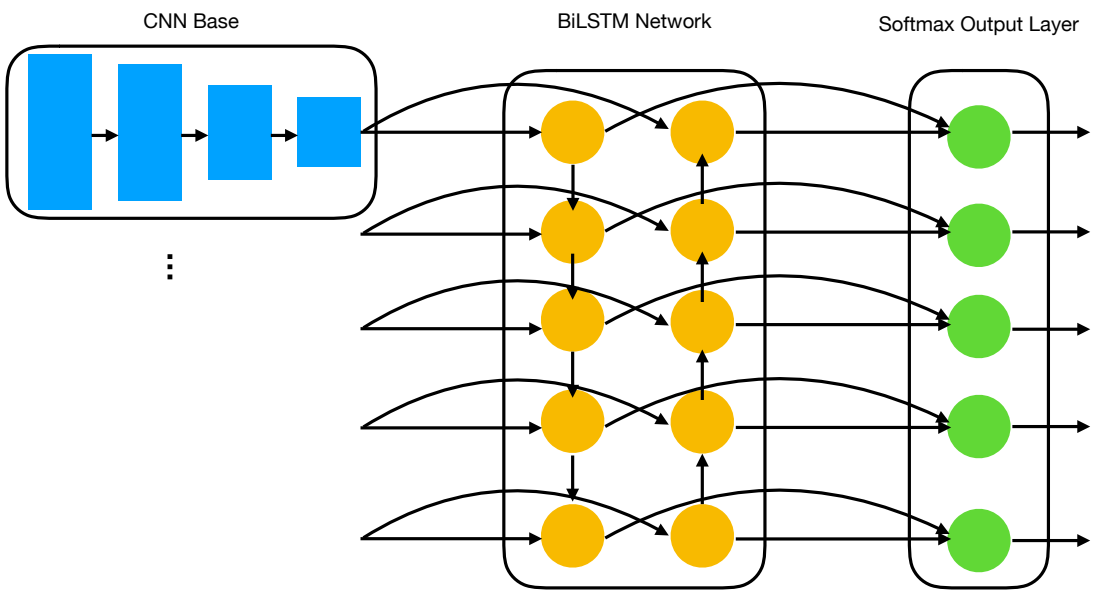


Figure 4.2: CNN-BiLSTM model architecture.

24-hours/day for one week. Participants also recorded self-reported sleep logs throughout their device wear. We selected data from 709 participants for our work.

4.1.3 Model Design

The CHAP model classifies sitting versus non-sitting behavioral postures and postural transitions from 10 Hz triaxial ActiGrah data (downsampled from 30 Hz via boxcar aggregation to reduce the size of the dataset). All computations were made on 10-second non-overlapping windows of continuous 10 Hz data, containing 100 triaxial acceleration values. The 10-second window size was chosen to align with activPAL's 10-second minimum threshold for labeling postural transitions. We used a model architecture family called CNN-BiLSTM architecture [241]. Figure 4.2 presents an illustration of the architecture of this model. It has three main components: 1) a CNN base, 2) a BiLSTM network, and 3) a softmax output layer akin to a logistic regression classifier. The first component automatically extracted features for identifying sitting versus non-sitting for each time point; the second component refined these features by considering

neighboring time points and the most likely sequence of events; the third component converted the extracted features to a final classification label (sitting or non-sitting).

4.1.4 Model Selection

We trained several model configurations on the training data and compared their performance when applied to the holdout validation data. Model configurations varied on four dimensions:

- BiLSTM window size (7 and 9 minutes).
- Number of neurons in a CNN layer (3200 and 6400 neurons).
- Learning rate (0.001 and 0.0001).
- Regularization coefficient (0.001 and 0.0001).

All possible unique combinations of domain values were tested, for a total of 16 unique model configurations tested. These comparisons enabled us to identify the best model configuration, based on several performance metrics. Metrics included overall and balanced classification accuracy, the ability to accurately capture transitions (i.e., changes in posture), sitting and non-sitting bout deviations, and Kolmogorov-Smirnov statistics for comparing CHAP-predicted vs. true (activPAL) labels of sitting and non-sitting bouts. Models with low accuracy or high variance, relative to competing models, on any of these metrics were eliminated.

We found that rigorous model selection is key to finding model configurations that result in good accuracies as the model accuracies varied significantly depending on the configuration (accuracies varied between 60%-93%). Three models performed equally well on all metrics, and these models were used to create a hybrid ensemble model that made classifications based on the majority vote. This ensemble model represented the complete CHAP model.

Table 4.1: Summary of experimental results.

Metric	Cut Point Method	TLBC	CHAP
Minute-level prediction accuracy	74%	83%	93%
Sit-to-stand transition sensitivity	73%	26%	83%
Sit-to-stand transition positive predictive value	30%	71%	83%
Mean sitting bout duration (activPAL 15.4 mins)	9.4 mins	49.4 mins	15.7 mins

4.1.5 Experimental Results

Using data from a test set, we compared the performance of CHAP to the performance of two other classification approaches that are commonly used in the field: 1) the standard ActiGraph cut-point (AG cut-point) method [84], and 2) a previously developed two-level behavior classification (TLBC) machine-learned model [113] designed to differentiate sitting from standing postures. Table 4.1 presents a summary of our experimental results.

For minute level sitting vs. non-sitting classification, CHAP performed better (93% agreement with activPAL) than other methods (74%-83% agreement). CHAP also outperformed other methods in its sensitivity to detecting sit-to-stand transitions: cut-point (73%), TLBC (26%), and CHAP (83%). CHAPs positive predictive value of capturing sit-to-stand transitions was also superior to other methods: cut-point (30%), TLBC (71%), and CHAP (83%). Day-level sitting pattern metrics, such as mean sitting bout duration, derived from CHAP did not differ significantly from activPAL, whereas other methods did: activPAL (15.4 mins mean sitting bout duration), CHAP (15.7 mins), cut-point (9.4 mins), TLBC (49.4 mins)

Overall, CHAP was the most accurate method for classifying sit-to-stand transitions and sitting patterns from free-living hip-worn accelerometer data in older adults. This promotes enhanced analysis of older adult movement data, resulting in more accurate measures of sitting patterns and opening the door for large-scale cohort studies into the effects of sitting patterns on healthy aging outcomes.

4.2 Extension: Intermittent Human-in-the-Loop Model Selection using CEREBRO

4.2.1 Introduction

In this work, we extend CEREBRO to implement a novel model selection paradigm that mitigates the drawbacks of existing ones. When training DL models practitioners have to perform model selection where they search over a potential configuration space of model architectures and training hyperparameters, in order to pick the best model.

Paradigms for Searching the Configuration Space. We found two main paradigms: 1) AutoML and 2) interactive human-in-the-loop. In the AutoML-based paradigm, the user will initiate a model selection workload by specifying a configuration search space and a canned AutoML procedure. AutoML procedures implement a search heuristic such as Bayesian optimization (e.g., HyperOpt [70]), evolutionary search (e.g., PBT [149]), and random search (e.g., ASHA [185]). It then uses the parallelism available in a cluster (or a single machine) to perform automated high-throughput configuration exploration. As model selection progresses, the user will receive the results of the explored configurations. Figure 4.3 (A) presents an illustration of this paradigm. While there are advanced AutoML procedure implementations of the above-mentioned search heuristics, recent surveys [78] have shown that ML practitioners often use simple techniques like grid search (explore all configurations) or random search (randomly sample configurations).

In interactive human-in-the-loop model selection [286, 87], the user retains full control over the search process. They will explicitly specify a configuration (or a few configurations) to be explored and wait until it finishes. Based on the results of the explored configurations and human intuition about the search space, they will specify the next configuration (or set of configurations) to explore. Figure 4.3 (B) presents an illustration of this paradigm.

False Dichotomy of Existing Paradigms. We compare the above two paradigms on two

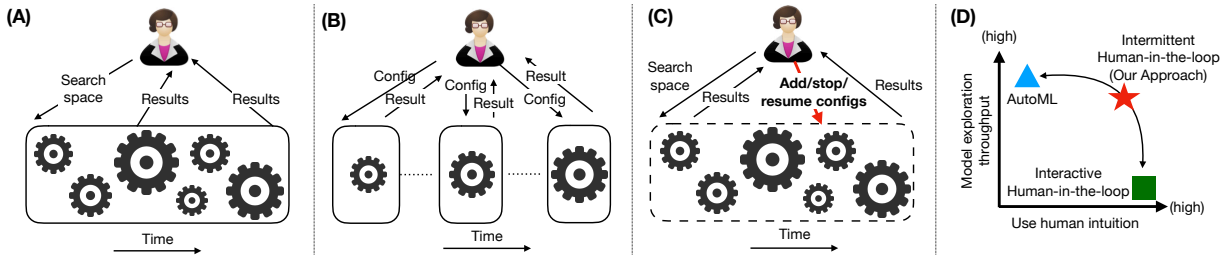


Figure 4.3: A) AutoML-based model selection. B) Interactive human-in-the-loop model selection. C) Our paradigm of *intermittent human-in-the-loop model selection*. D) Qualitative comparison of different paradigms.

dimensions: 1) model exploration throughput and 2) the ability to use human intuition. As shown in Figure 4.3 (D), AutoML-based model selection is capable of performing high-throughput model exploration. But the only time it relies on human intuition is during the initial specification of the search space. As a result, it may inefficiently explore the configuration space and incur significant resource costs, which could have been avoided by a simple human intervention! On the other hand, the human-in-the-loop model selection primarily relies on human intuition but operates at very-low throughput levels due to the inherent limitations of human interactivity. Also, it can be tedious and time-consuming. Overall, we see a major gap between AutoML-based and human-in-the-loop model selection paradigms today.

This Work. To overcome the above-mentioned drawbacks, we propose a new paradigm for model selection, which we call *intermittent human-in-the-loop model selection*. It is a hybrid of both AutoML-based and interactive human-in-the-loop model selection. However, unlike in the interactive human-in-the-loop paradigm, human exploration is not mandatory in our approach. As an analogy, the former is akin to instant messaging (IM), whereas our paradigm is akin to email threads or Slack channels. Without interactivity, the former becomes not usable. But our approach is more flexible due to asynchronous, spread-out-over-time yet stateful exchanges that can even subsume full interactivity. We implement our paradigm in the CEREBRO system [215], a system for resource-efficient deep learning model selection. We

also extend the CEREBRO system with a graphical user interface, a REST API, and perform changes to existing components to support our new paradigm. Our paradigm is an ideal fit for deep learning model selection workloads due to their long-running nature. But it is readily applicable to any other ML model family too. A short video of our system can be found on our project web page: `https://adalabucsd.github.io/cerebro.html`.

4.2.2 New Paradigm for Model Selection

Our intermittent human-in-the-loop model selection paradigm for model selection breaks the false dichotomy of AutoML-based and interactive human-in-the-loop model selection. It starts similar to the AutoML-based paradigm where the user specifies the search space and picks a canned AutoML procedure like Grid, Random, or even a more advanced AutoML procedure like HyperOpt. However, instead of passively waiting by just consuming the results of explored configurations, we enable the user to steer the model selection process. They can now create new individual configurations or batch of configurations using a refined search space, stop running configurations, and resume stopped configurations.

Creating new configurations outside the control of the AutoML procedure enables the user to inject human intuition into the overall model selection process. New configurations can also be created by first cloning an existing configuration along with its trained parameters and then by tweaking only some of the hyper-parameters like learning rate or batch size. Users can use this feature to make the model training adaptable based on human intuition. They can also dynamically reprioritize the training of some configurations over the others by using the stop and resume feature. Thus, as shown in Figure 4.3 (D) our paradigm can seamlessly navigate the exploration throughput and human intuition usage tradeoff space based on the available user interaction level.

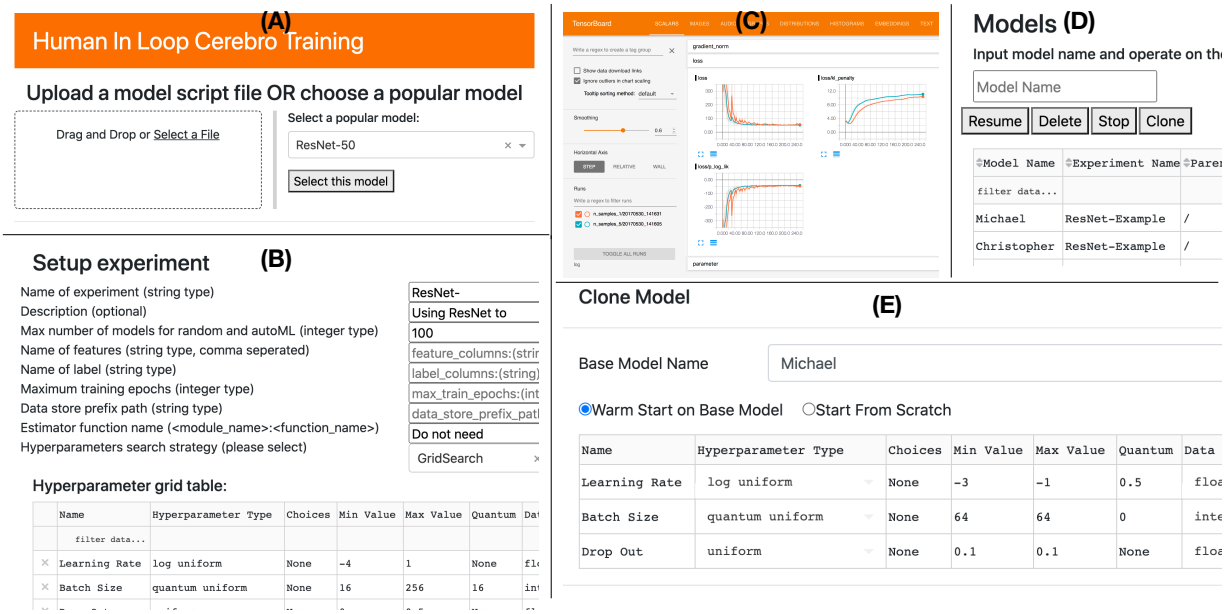


Figure 4.4: User interface for intermittent human-in-the-loop model selection. (A) UI to either pick a canned ML model (e.g., ResNet50) or upload a script file defining a custom model. (B) UI to specify experiment metadata, training data information, and configuration search space. (C) Model training performance visualization using embedded TensorBoard. (D) UI listing all configurations and controls to add/stop/resume configurations. (E) UI to create a drill-down model selection workload.

4.2.3 UIs for Intermittent Specification

System UI provides graphical controls that enable the user to perform intermittent human-in-the-loop model selection. It is implemented using Python Dash visualization library and runs in a web browser which makes it portable. It is integrated with a backend REST API to perform the user-requested actions.

The user will start interacting with our system by either picking a canned ML model from a roster or by uploading a Python script defining a custom ML model using the UI shown in Figure 4.4 (A). We currently support four popular deep learning models in our roster: ResNet50, MobileNet, BERT-base, and DistilBert. New models can be easily added to the roster. Also, the custom script option can support arbitrary Keras models. After picking a model, the user will be then prompted with the UI shown in Figure 4.4 (B) to specify a name, description, AutoML search

procedure, names of features and label columns, the path to the training data, and the maximum number of training epochs for any model. If a custom script is uploaded, the user is required to specify the entry point function name in that script. This entry point function should take a dictionary of configuration values as input and return compiled Keras model as output. The user is also required to specify the search spaces for the available configurations. The list of available configurations is fixed for a canned model. For a custom model, user can manually define the configurations that the model depends on. After specifying these values, the user can launch the model selection workload. While the workload is running, the user can visualize model training metrics, such as loss and accuracy, through an embedded TensorBoard UI as shown in Figure 4.4 (C). They can also add/stop/resume configurations using the controls shown in Figure 4.4 (D) or create a new drill-down workload on a refined search space using the UI shown in Figure 4.4 (E).

4.2.4 Decoupled System Architecture

Implementing our paradigm requires decoupling configuration exploration logic from model training and being able to multiplex the training of multiple configurations on a fixed set of resources. Otherwise, it is simply not possible to run multiple model selection workloads at the same time or even increase the model selection throughput of a single workload without provisioning more resources. While resource provisioning has become easy with cloud computing, cloud users also often need to limit their resource usages due to cost concerns. For others like domain science users, it may be simply not possible to provision more resources such as in fixed-sized campus clusters.

We implement our paradigm in the CEREBRO system. CEREBRO uses a novel parallel execution strategy called *model hopper parallelism* (MOP) that can significantly increase the model selection throughput without provisioning more resources. It does so by breaking the training of a single model configuration over partitioned data into multiple units called sub-epochs and multiplexing the training of multiple configurations–one epoch at a time–by asynchronously

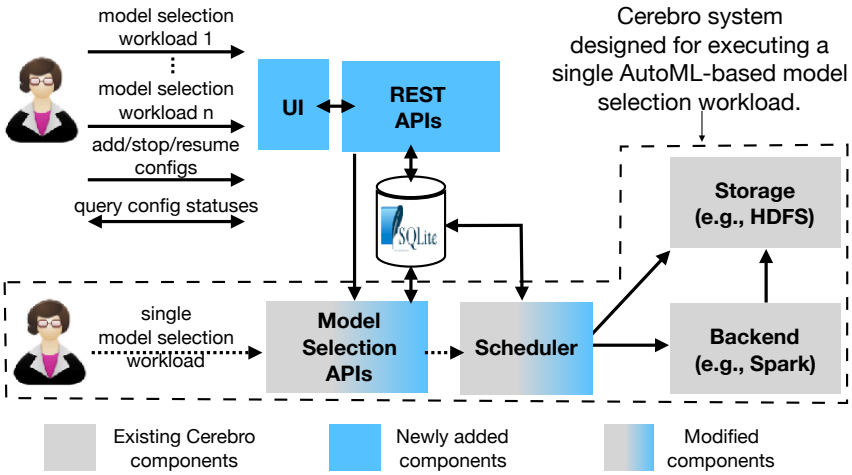


Figure 4.5: High-level system architecture diagram of CEREBRO along with the new components added to support our *intermittent human-in-the-loop model selection paradigm*.

scheduling sub-epochs on workers. A sub-epoch essentially trains the configuration for one pass over the locally available partition of the data. Originally, the CEREBRO system was designed to execute a single AutoML-based model selection workload at a time. Figure 4.5 presents the high-level architecture of the CEREBRO system.

We leverage the epoch-level scheduling template of CEREBRO to support our new paradigm. We also add a new graphical user interface (UI), a REST API and update CEREBRO's model selection APIs and scheduler to achieve our requirements. UI sends user requests to the model selection APIs through the REST API. We changed the model selection APIs such that they now write the configurations to an SQLite database instead of directly interacting with the scheduler. User-created configurations are also directly added to this database. The scheduler will then read all the configurations that need to be trained from this database and train them for one epoch. After completing training for one epoch it will update the training metrics of the configuration in the database. And this process will continue. Whenever the user wants to stop (resp. resume) a configuration, it will be marked as such in the database and will be ignored (resp. considered back) by the scheduler for training. Figure 4.5 presents the system architecture of the modified CEREBRO system.

Chapter 4 Section 4.1 contains material from "The CNN Hip Accelerometer Posture (CHAP) Method for Classifying Sitting Patterns from Hip Accelerometers: A Validation Study" by Supun Nakandala, Mikael Anne Greenwood-Hickman, Marta M Jankowska, Dori Rosenberg, Fatima Tuz-Zahra, John Bellettiere, Jordan Carlson, Paul R Hibbing, Jingjing Zou, Andrea Z LaCroix, Arun Kumar, and Loki Natarajan, which appears in the Journal of Medicine & Science in Sports & Exercise Volume 53, Issue 11, November 2021. The dissertation author was a primary investigator and a primary author of this paper. The CHAP models are open source and are available on GitHub: https://github.com/ADALabUCSD/DeepPostures.

Chapter 4 Section 4.2 contains material from "Intermittent Human-in-the-Loop Model Selection using Cerebro: A Demonstration" by Liangde Li, Supun Nakandala, and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 14, Issue 12, July 2021. The dissertation author's contribution was in the conceptualization of the system, parts of the implementation, and advising the junior student through the rest of the system implementation.

Chapter 5

HUMMINGBIRD: Query Optimizations for Classical ML Prediction Serving on DL Systems

5.1 Introduction

In this chapter, we dive deeper into our techniques for enabling classical machine learning inference on DL systems. Classical machine learning is the term used to broadly categorize all machine learning model families, which are not based on DL. While deep learning is highly popular for unstructured data analytics, classical machine learning is still the king for structured data analytics [56]—examples include predictive maintenance, customer churn prediction, and supply-chain optimizations [118]. A recent analysis by Amazon Web Services found that 50% to 95% of all ML applications in an organization are based on classical ML [61]. As a point of comparison with DL systems, scikit-learn, a popular library used for classical ML, is used about 5 times more than PyTorch [228] and TensorFlow [54] combined, and growing faster than both. Acknowledging this trend, classical ML capabilities have been recently added to DL systems,

such as the ONNX-ML [14] flavor in ONNX [42] and TensorFlow's TFX [67].

Predictive Pipelines. Unlike in DL, where the output of model training/model selection step is a single model, in classical ML the output is a *predictive pipeline*: a Directed Acyclic Graph (DAG) of operators. Such pipelines are typically comprised of up to tens of operators out of a set of hundreds [233] that fall into two main categories: (1) *featurizers*, which could be either stateless imperative code (e.g., string tokenization) or data transformations fit to the data (e.g., normalization); and (2) *models*, commonly decision tree ensembles or (generalized) linear models, fit to the data. Note that the whole pipeline is required to perform a prediction.

A Missing Abstraction. Today's featurizers and model implementations *are not expressed in a shared logical abstraction, but rather in an ad-hoc fashion* using programming languages such as R, Python, Java, C++, or C#. This hints to the core problem with today's approaches to model inference: *the combinatorial explosion of supporting many operators (and frameworks) across multiple target environments*. Figure 5.1 (top) highlights this visually by showing how existing solutions lead to an $O(N \times M)$ explosion to support N operators from various ML frameworks against M deployment environments (e.g., how to run a scikit-learn model on an embedded device?). Furthermore, [233] shows that the number of libraries used in data science (a metric correlated to N) increased by roughly $4\times$ in the last 2 years. It is expected that M is also destined to grow as ML is applied more widely across a broad range of enterprise applications and hardware (e.g., [30, 7, 147, 154, 46]). From the vantage point of implementing systems for model inference, this is a daunting proposition. We argue that any brute-force approach directly tackling all combinations would dilute engineering focus leading to costly and less optimized solutions. In fact, today, with very few exceptions (e.g., NVIDIA RAPIDS [13] for GPU), classical ML operators are only implemented for CPUs.

This state of affairs is in contrast with the DL space, where neural networks are authored using tensor transformations (e.g., multiplications, convolutions), providing an algebraic abstraction over computations. Using such abstractions rather than imperative code not only enables

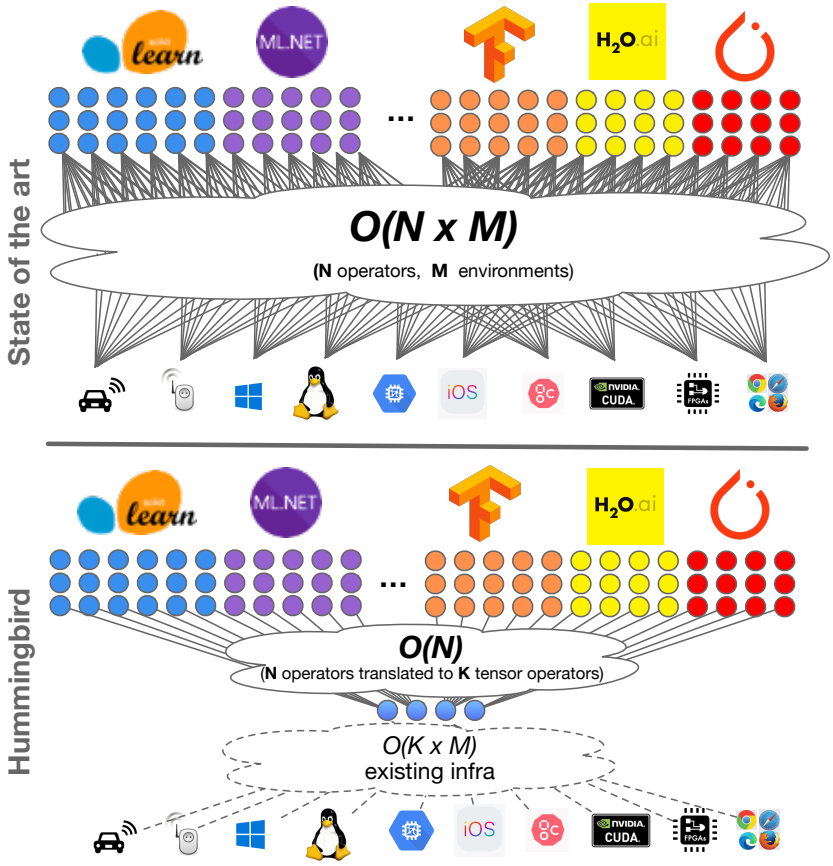


Figure 5.1: Prediction serving complexity: state-of-the-art (top) vs. HUMMINGBIRD (bottom).

evolved optimizations [52, 93] but also facilitates support for diverse environments (such as mobile devices [43], web browsers [48], and hardware accelerators [30, 154, 147]), unlocking new levels of performance and portability.

Our Solution. To bypass this $N \times M$ explosion in implementing classical ML operators, we built HUMMINGBIRD (HB for short). HUMMINGBIRD leverages query optimization-inspired techniques to translate a broad set of classical ML operators into a small set of K core operators, thereby reducing the cost to $O(N) + O(K \times M)$, as shown in Figure 5.1 (bottom). This is also the key intuition behind the ONNX model format [42] and its various runtimes [16]. However, with HUMMINGBIRD we take one further bold step: we demonstrate that this set of core operators can be reduced to tensor computations and therefore be executed over DL systems. This allows us to piggyback on existing investments in DL compilers, runtimes, and specialized hardware,

and reduce the challenge of "running K operators across M environments" for classical ML to just $O(N)$ operator translations. This leads to improved performance and portability, and reduced infrastructure complexity.

Contributions. In this work we answer three main questions:

1. Can classical ML operators (both linear algebra-based such as linear models, and algorithmic ones such as decision trees) be translated to tensor computations?

2. Can the resulting computations in tensor space be competitive with the imperative alternatives we get as input (e.g., traversing a tree)?

3. Can HUMMINGBIRD reduce software complexity and improve model portability?

Concretely, we: (1) port thousands of benchmark predictive pipelines to two DL systems (PyTorch and TVM); (2) show that we can seamlessly leverage hardware accelerators and deliver speedups of up to 3× against hand-crafted GPU kernels, and up to 1200× for predictive pipelines against state-of-the-art frameworks; and (3) qualitatively confirm improvements in software complexity and portability by enabling scikit-learn pipelines to run across CPUs and GPUs.

Organization. The remainder of the chapter is organized as follows. Section 5.2 provides some background, and Section 5.3 presents an overview of HUMMINGBIRD. Section 5.4 describes the translation from classical ML to tensor computations, whereas Section 5.5 discusses various optimizations. Section 5.6 presents our evaluation, then we conclude.

5.2 Background and Challenges

We first provide background on classical ML predictive pipelines. We then explain the challenges of translating classical ML operators and predictive pipelines into tensor computations.

5.2.1 Classical ML Predictive Pipelines

The result of the data science workflow over classical ML are predictive pipelines, i.e., Directed Acyclic Graphs (DAGs) of operators such as trained models, preprocessors, featurizers, missing-value imputers. The process of presenting a trained predictive pipeline with new data to obtain a prediction is referred to as inference. Packaging a trained pipeline into a single artifact is common practice [57]. These artifacts are then embedded inside host applications, or containerized and deployed in the cloud to perform model scoring [232, 101]. ML.NET (.NET-based), scikit-learn (Python-based), and H_2O (Java-based) are popular toolkits to train and generate pipelines. However, it is important to note that they are primarily optimized for training, not for inference. Predictive pipeline inference is challenging, as their operators are implemented in imperative code, and do not follow a shared logical or physical abstraction. Supporting every operator in all target environments requires a huge effort, which is why these frameworks have limited portability.

5.2.2 Challenges

HUMMINGBIRD combines the strength of classical ML pipelines on structured data [188] with the computational and operational simplicity of DL systems for model inference. To do so, it relies on a simple yet key observation: once a model is trained, it can be represented as a *prediction function* transforming input features into a prediction score (e.g., 0 or 1 for binary classification), regardless of the training algorithm used. The same observation naturally applies to featurizers fit to the data. Therefore, HUMMINGBIRD only needs to translate the prediction functions (not the training logic) for each operator in a pipeline into tensor computations and stitch them appropriately. Towards this goal, we identify two challenges.

1. **Challenge 1**: *How can we map classical predictive pipelines into tensor computations?* Pipelines are generally composed of operators (with predictive functions) of two classes: *al-*

gebraic (e.g., scalers or linear models) and *algorithmic* (e.g., one-hot encoder and tree-based models). While translating algebraic operators into tensor computations is straightforward, the key challenge for HUMMINGBIRD is the translation of algorithmic operators. Algorithmic operators perform arbitrary *data accesses and control flow decisions*. For example, in a decision tree ensemble potentially every tree is different from each other, not only with respect to the structure, but also the decision variables and the threshold values. Conversely, tensor operators perform *bulk operations* over the entire set of input elements.

2. **Challenge 2**: *How can we achieve efficient execution for tensor-translated classical ML operators?* The ability to translate predictive pipelines into DAGs of tensor operations does not imply adequate performance of the resulting DAGs. In fact, common wisdom would suggest the opposite: even though DL systems naturally support execution on hardware accelerators, tree-based methods and commonly used data transformations are well known to be difficult to accelerate [100], even using custom-developed implementations.

5.3 System Overview

In this section we explain our approach to overcome the challenges outlined in Section 5.2.2, and present HUMMINGBIRDS's architecture and implementation details. We conclude this section by explaining assumptions and limitations.

5.3.1 High-level Approach

In HUMMINGBIRD, we cast algorithmic operators into tensor computations. You will notice that this transformation *introduces redundancies*, both in terms of *computation* (we perform more computations than the original classical ML operators) and *storage* (we create data structures that store more than what we actually need). Although these redundancies might sound counter-intuitive at first, we are able to transform the arbitrary data accesses and control flow of the original

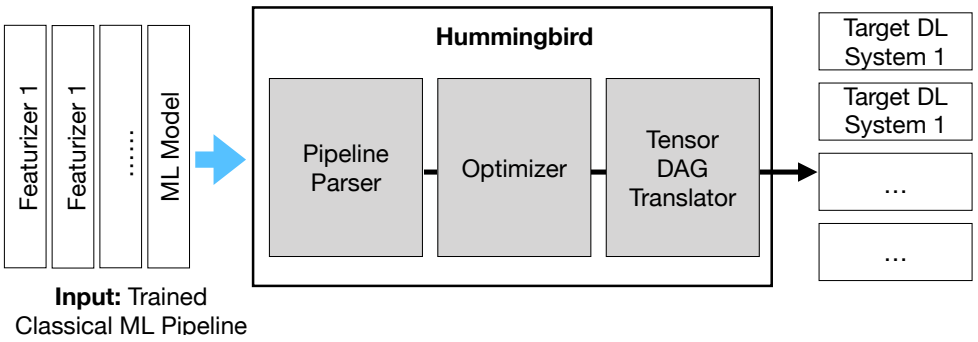


Figure 5.2: High-level architecture of HUMMINGBIRD.

operators into tensor operations that lead to efficient computations by leveraging state-of-the-art DL systems.

For a given classical ML operator, there exist different strategies for translating it to tensor computations, each introducing a different degree of redundancy. We discuss such strategies for representative operators in Section 5.4. The optimal tensor implementation to be used varies and is informed by model characteristics (e.g., tree-structure for tree-based models, or sparsity for linear models) and runtime statistics (e.g., batch size of the inputs). *Heuristics at the operator level*, *target-independent optimizations at the pipeline level*, and *target-specific optimizations at the execution level* enable HUMMINGBIRD to further improve predictive pipelines performance end-to-end. The dichotomy between target-independent and target-specific optimizations allow us to both (1) apply optimizations unique to classical ML and not captured by the DL systems; and (2) exploit DL system optimizations once the classical ML is lowered into tensor computations. Finally, HUMMINGBIRD is able to run end-to-end pipelines on the hardware platforms supported by the target DL system.

5.3.2 System Architecture and Implementation

The high-level architecture of HUMMINGBIRD is shown in Figure 5.2. HUMMINGBIRD has three main components: (1) *Pipeline Parser*, (2) *Optimizer*, and (3) *Tensor DAG Translator*.

Pipeline Parser. In this phase, input pipelines are parsed one operator at a time, and each operator is *wrapped* into a *container* object. Each operator's container maintains (1) the inputs and outputs of the operator, and (2) the *operator signature* that codifies the operator type (e.g., "scikit-learn decision tree"). HUMMINGBIRD parser also introduces a set of *extractor functions* that are used to extract the parameters of each operator (e.g., weights of a linear regression, thresholds of a decision tree). Operator signatures dictate which extractor function should be used for each operator. At startup time, extractor functions are registered into a hash table, mapping operator signatures to the related extractor function. HUMMINGBIRD parser is extensible, allowing users to easily add new extractor functions. HUMMINGBIRD supports over 40 scikit-learn operators (listed in Table 5.2), as well as parsers for XGBoost [92], LightGBM [159], and ONNX-ML [14]. At the end of the parsing phase, the input pipeline is "logically" represented in HUMMINGBIRD as a DAG of containers storing all the information required for the successive phases. HUMMINGBIRD parser is based on skl2onnx [47].

Optimizer. In this phase, the DAG of containers generated in the parsing phase is traversed in topological order in two passes. During the first traversal pass, the Optimizer extracts the parameters of each operator via the referenced extractor function and stores them in the container. Furthermore, since HUMMINGBIRD supports different operator implementations based on the extracted parameters, the Optimizer annotates the container with the translation strategy to be used for that specific operator (5.5.1). During the second pass, HUMMINGBIRD tries to apply target-independent optimizations (5.5.2) over the DAG.

Tensor DAG Translator. In this last phase, the DAG of containers is again traversed in topological order and a *conversion-to-tensors function* is triggered based on each operator signatures. Each conversion function receives as input the extracted parameters and generates a PyTorch's *neural network module* composed of a small set of tensor operators (listed in Table 5.1). The generated module is then exported into the target DL systems format. The current version of HUMMINGBIRD supports PyTorch/TorchScript, ONNX, and TVM output formats. The target-

Table 5.1: PyTorch tensor operators used by the Tensor DAG Compiler.

`matmul, add, mul, div, lt, le, eq, gt, ge, &, |, ≪, ≫, bitwise_xor, gather, index_select, cat, reshape, cast, abs, pow, exp, arxmax, max, sum, relu, tanh, sigmoid, logsumexp, isnan, where`

Table 5.2: Scikit-learn operators currently supported in HUMMINGBIRD.

Supported ML Models
LogisticRegression, SVC, NuSVC, LinearSVC, SGDClassifier, LogisticRegressionCV, DecisionTreeClassifier/Regression, RandomForestClassifier/Regression, ExtraTreesClassifier/Regressor, GradientBoostingClassifier/Regression, HistGradientBoostingClassifier/Regressor, IsoltationForest, MLPClassifier, BernoulliNB, GaussianNB, MultinomialNB
Supported Featurizers
SelectKBest, VarianceThreshold, SelectPercentile, PCA, KernelPCA, TruncatedSVD, FastICA, SimpleImputer, Imputer, MissingIndicator, RobustScaler, MaxAbsScaler, MinMaxScaler, StandardScaler, Binarizer, KBinsDiscretizer, Normalizer, PolynomialFeatures, OneHotEncoder, LabelEncoder, FeatureHasher

specific optimizations are triggered at this level.

5.3.3 Assumptions and Limitations

In this work, we make a few simplifying assumptions. First, we assume that predictive pipelines are "pure", i.e., they do not contain arbitrary user-defined operators. There has been recent work [236] on compiling imperative UDFs (user-defined functions) into relational algebra, and we plan to make use of such techniques in HUMMINGBIRD in the future. Second, we do not support sparse data well. We found that current support for sparse computations on DL systems is primitive and not well optimized. We expect advances in DL systems to improve on this aspect—TACO [164] is a notable such example. Third, although we support string operators, we currently do not support text feature extraction (e.g., `TfidfVectorizer`). The problem in this case is twofold: (1) translating regex-based tokenizers into tensor computations is not trivial,

and (2) representing arbitrarily long text documents in tensors is still an open challenge. Finally, HUMMINGBIRD is currently limited by single GPU memory execution. Given that several DL systems nowadays support distributed processing [252, 189], we plan to investigate distributed inference as future work.

5.4 Translation

HUMMINGBIRD supports translating several algorithmic operators into tensor computations. Given their popularity [233], in Section 5.4.1 we explain our approach for tree-based models. Section 5.4.2 gives a summary of other techniques that we use for both algorithmic and arithmetic operators.

5.4.1 Translating Decision Tree-based Models

HUMMINGBIRD has three different strategies for translating tree-based models. Strategies differ based on the degree of redundancy introduced. Table 5.3 explains the notation used in this section. We summarize the worst-case runtime and memory footprints of each strategy in Table 5.4. HUMMINGBIRD currently supports only trees built over numerical values: support for missing and categorical values is under development. For the sake of presentation, we assume all decision nodes perform $<$ comparisons.

Strategy 1: `GEMM`. We cast the evaluation of a tree as a series of three GEneric Matrix Multiplication (`GEMM`) operations interleaved by two element-wise logical operations. Given a tree, we create five tensors which collectively capture the tree structure: A, B, C, D, and E. A captures the relationship between input features and internal nodes. B is set to the threshold value of each internal node. For any leaf node and internal node pair, C captures whether the internal node is a parent of that internal node, and if so, whether it is in the left or right sub-tree. D captures the count of the internal nodes in the path from a leaf node to the tree root, for which

Table 5.3: Notation used in Section 5.4.1

Symbol	Description
N, I, L, F, C	Ordered lists with all nodes, internal nodes, leaf nodes, features, and classes, respectively.
$X \in \mathbb{R}^{n \times \lvert F \rvert}$	Input records (n is the number of records).
$A \in \mathbb{R}^{\lvert F \rvert \times \lvert I \rvert}$	$A_{i,j} = \begin{cases} 1, & I_j \text{ evaluates } F_i \\ 0, & \text{Otherwise} \end{cases}$
$B \in \mathbb{R}^{\lvert I \rvert}$	$B_i = \texttt{ThresholdValue}(I_i)$
$C \in \mathbb{R}^{\lvert I \rvert \times \lvert L \rvert}$	$C_{i,j} = \begin{cases} -1, & L_j \in \texttt{RightSubTree}(I_i) \\ 1, & L_j \in \texttt{LeftSubTree}(I_i) \\ 0, & \text{Otherwise} \end{cases}$
$D \in \mathbb{R}^{\lvert L \rvert}$	$D_k = \sum_{k \in L \xrightarrow{path} \texttt{Root}} \mathbf{1}(k == \texttt{LeftChild}(\texttt{Parent}(k)))$
$E \in \mathbb{R}^{\lvert L \rvert \times \lvert C \rvert}$	$E_{i,j} = \begin{cases} 1, & L_i \xrightarrow{map\ to} C_j \\ 0, & \text{Otherwise} \end{cases}$

Table 5.4: Worst-case memory and runtime analysis of different tree translation strategies, assuming the number of input records and number of trees are fixed. The notation is explained in Table 5.3

Strategy	Memory	Runtime
GEMM	$O(\lvert F \rvert \lvert N \rvert + \lvert N \rvert^2 + \lvert C \rvert \lvert N \rvert)$	$O(\lvert F \rvert \lvert N \rvert + \lvert N \rvert^2 + \lvert C \rvert \lvert N \rvert)$
TT	$O(\lvert N \rvert)$	$O(\lvert N \rvert)$
PTT	$O(2^{\lvert N \rvert})$	$O(\lvert N \rvert)$

the internal node is the left child of its parent. Finally, E captures the mapping between leaf nodes and the class labels. Given these tensors, Algorithm 3 presents how we perform tree scoring for a batch of input records X. A graphical representation of an execution of the GEMM strategy is depicted in Figure 5.3.

The first GEMM is used to match each input feature with the internal node(s) using it. The following $<$ operations is used to evaluate all the internal decision nodes and produces a tensor of 0s and 1s based on the false/true outcome of the conditions. The second GEMM operation generates an encoding for the path composed by the true internal nodes, while the successive $==$ operation returns the leaf node selected by the encoded path. Finally, the third GEMM operation maps the

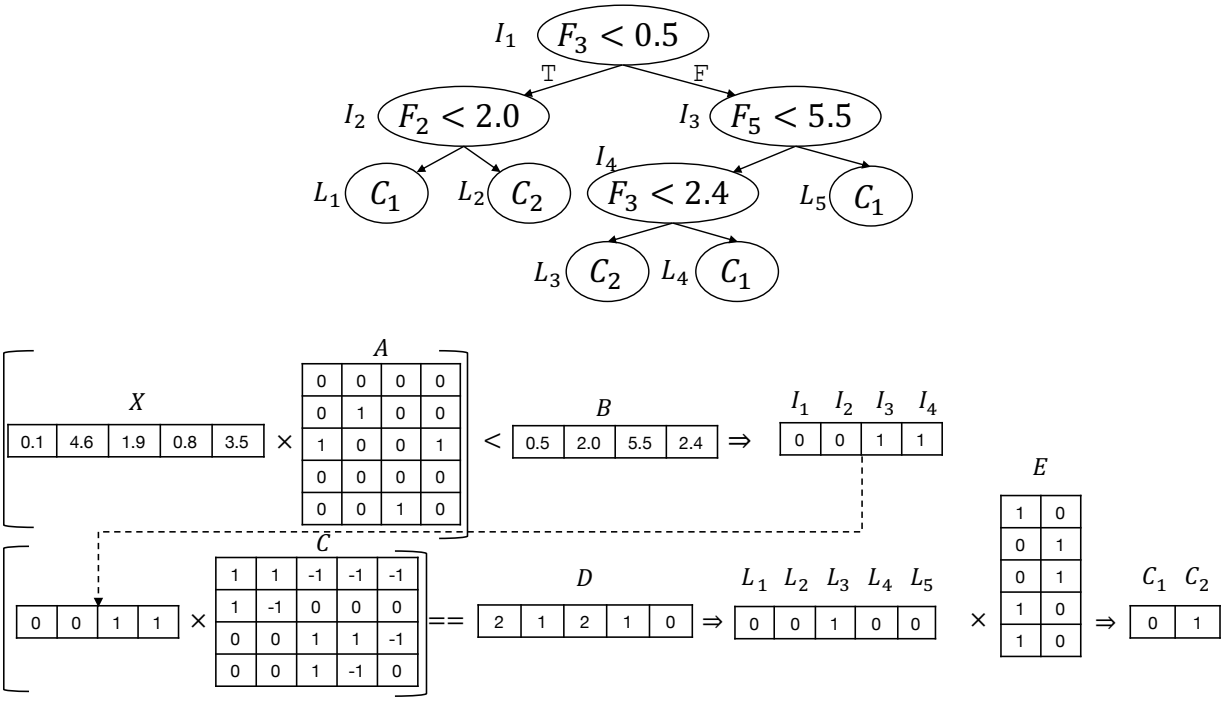


Figure 5.3: Compiling an example decision tree using the GEMM strategy.

Algorithm 3 GEMM Strategy (Notation explained in Table 5.3)

Input: $X \in \mathbb{R}^{n \times |F|}$, Input records
Output: $R \in \{0,1\}^{n \times |C|}$, Predicted class labels
// Evaluate all internal nodes
$T \leftarrow$ GEMM(X, A) $\quad //T \in \mathbb{R}^{n \times |I|}$
$T \leftarrow T < B$ $\quad //T \in \mathbb{R}^{n \times |I|}$
// Find the leaf node which gets selected
$T \leftarrow$ GEMM(T, C) $\quad //T \in \mathbb{R}^{n \times |L|}$
$T \leftarrow T == D$ $\quad //T \in \mathbb{R}^{n \times |L|}$
// Map selected leaf node to class label
$R \leftarrow$ GEMM(T, E) $\quad //R \in \mathbb{R}^{n \times |C|}$

selected leaf node to the class label.

This strategy can be easily applied to support tree ensembles and regression tasks too. For tree ensembles, we create the above 2-dimensional tensors for each tree and batch them together. As the number of leaf nodes and internal nodes can vary among trees, we pick the maximum number of leaf nodes and internal nodes for any tree as the tensor dimensions and pad the smaller

Table 5.5: Additional notation used in Strategy 2: `TreeTraversal`

Symbol	Description
$N_L \in \mathbb{Z}^{\lvert N\rvert}$	$N_{L_i} = \begin{cases} \texttt{LeftChild}(N_i), N_i \in I \\ i, \text{Otherwise} \end{cases}$
$N_R \in \mathbb{Z}^{\lvert N\rvert}$	$N_{R_i} = \begin{cases} \texttt{RightChild}(N_i), N_i \in I \\ i, \text{Otherwise} \end{cases}$
$N_F \in \mathbb{Z}^{\lvert N\rvert}$	$N_{F_i} = \begin{cases} k, (N_i \in I) \wedge (N_i \text{ evaluates } F_k) \\ 1, \text{Otherwise} \end{cases}$
$N_T \in \mathbb{R}^{\lvert N\rvert}$	$N_{T_i} = \begin{cases} \texttt{ThresholdValue}(N_i), N_i \in I \\ 0, \text{Otherwise} \end{cases}$
$N_C \in \mathbb{Z}^{\lvert N\rvert \times \lvert C\rvert}$	$N_{C_{i,k}} = \begin{cases} 1, (N_i \in L) \wedge (N_i \xrightarrow{map\ to} C_k) \\ 0, \text{Otherwise} \end{cases}$

tensor slices with zeros. During scoring, we invoke the batched variants of `GEMM` and logical operations and perform a final `ReduceMean` operation over the batched dimension to generate the ensemble output. For regression tasks, we initialize E with label values.

Strategy 2: TreeTraversal (TT). In the `GEMM` strategy, we incorporated a high degree of computational redundancy by evaluating all internal nodes and leaf nodes. Here, we try to reduce the computational redundancy by mimicking the typical tree traversal—but implemented using tensor operations. In this strategy, the tree structure is captured by five tensors: N_L, N_R, N_F, N_T, and N_C. We formally define these tensors in Table 5.5. The same column index (last dimension) across all tensors corresponds to the same tree node. N_L and N_R capture the indices of the left and right nodes for a given node. If the node is a leaf node, we set these to the index of the given node. Similarly, N_F and N_T capture the feature index and threshold value for each node, respectively. For leaf nodes, we set N_F to 1 and N_T to 0. Finally, N_C captures the class label of each leaf node. For internal nodes this can be any value; we set it to 0.

Given these tensors, Algorithm 4 presents how we perform scoring for a batch of input records X. We use `Gather` and `Where` operations which can be used to perform index-based slicing and conditional value selection. We first initialize an index tensor T_I corresponding to

Algorithm 4 `TreeTraversal` Strategy (Notation in Tables 5.5)

Input: $X \in \mathbb{R}^{n \times |F|}$, Input records
Output: $R \in \{0,1\}^{n \times |C|}$, Predicted class labels
// Initialize all records to point to k, with k the index of `Root` node.
$T_I \leftarrow \{k\}^n$ //$T_I \in \mathbb{Z}^n$
for $i \leftarrow 1$ **to** `TREE_DEPTH` **do**
    // Find the index of the feature evaluated by the current node. Then find its value.
    $T_F \leftarrow$`Gather`(N_F, T_I) //$T_F \in \mathbb{Z}^n$
    $T_V \leftarrow$`Gather`(X, T_f) //$T_V \in \mathbb{R}^n$
    // Find the threshold, left child and right child
    $T_T \leftarrow$`Gather`(N_T, T_I) //$T_T \in \mathbb{R}^n$
    $T_L \leftarrow$`Gather`(N_L, T_I) //$T_L \in \mathbb{Z}^n$
    $T_R \leftarrow$`Gather`(N_R, T_I) //$T_R \in \mathbb{Z}^n$
    // Perform logical evaluation. If true pick from T_L; else from T_R.
    $T_I \leftarrow$`Where`$(T_V < T_T, T_L, T_R)$ //$I \in \mathbb{Z}^n$
// Find label for each leaf node
$R \leftarrow$`Gather`(N_C, T_I) //$R \in \mathbb{Z}^n$

all records in X, which points to the root node. Using T_I, we `Gather` the corresponding feature indices and use them to `Gather` the corresponding feature values from X. Similarly, we also `Gather` left node indices, right node indices, and node thresholds. Using these gathered tensors, we then invoke a `Where` operation which checks for the tree node decisions. Based on the evaluation, for each record the `Where` operator either returns the left child index or right child index. To perform full tree scoring, the above steps have to be repeated until we reach a leaf node for all records in X. We exploit the fact that (1) `TREE_DEPTH` is a known property of the input model at translation time, and (2) all leaf nodes are at a depth $\leq$ `TREE_DEPTH`, to iterate for that fixed number of iterations to ensure that all records have found their corresponding leaf node. Tensors are created in such a way that if one of the indices reaches a leaf node before running for `TREE_DEPTH` iterations, the same class label will keep getting selected. At translation time, we unroll all iterations and remove the `for` loop to improve efficiency. For ensembles, we create tensors for each tree and batch them together. However, between trees the number of nodes and dimensions may differ, so we use the maximum node count for any tree as the dimension and pad the remaining elements.

Table 5.6: Additional notation used in Strategy 3

Symbol	Description
$I' \in \mathbb{Z}^{2^{D-1}}, L' \in \mathbb{Z}^{2^D}$	Internal and leaf nodes of the perfect tree ordered by level.
$N'_F \in \mathbb{Z}^{\lvert I' \rvert}$	$N'_{F_i} = k \iff I'_i$ evaluates F_k
$N'_T \in \mathbb{R}^{\lvert I' \rvert}$	$N'_{T_i} = \texttt{ThresholdValue}(I'_i)$
$N'_C \in \mathbb{Z}^{\lvert L' \rvert \times \lvert C \rvert}$	$N'_{C_{i,k}} = \begin{cases} 1, N_i \xrightarrow{map\ to} C_k \\ 0, \text{Otherwise} \end{cases}$

Strategy 3: PerfectTreeTraversal (PTT). Similar to the previous one, this strategy also mimics the tree traversal. However, here we assume the tree is a *perfect binary tree*. In a perfect binary tree, all internal nodes have exactly two children and all leaf nodes are at the same depth level. Assume we are given a non-perfect binary tree with a `TREE_DEPTH` of D, and L_k is a leaf node which is at a depth of $D_k < D$. To push L_k to a depth D, we replace L_k with a perfect sub-tree of depth $D - D_k$ and map all the leaf nodes of the sub-tree to C_k: the label of the original leaf node. The decision nodes in the introduced sub-tree are free to perform arbitrary comparisons as the outcome is the same along any path. By pushing all leaf nodes at depth $< D$ to a depth of D, we transform the original tree to a perfect tree with the same functionality.

Working on perfect trees enables us to get rid of N_L and N_R tensors as we can now calculate them analytically, which also reduces memory lookup overheads during scoring. Thus we create only three tensors to capture the tree structure: N'_F, N'_T, and N'_C (Table 5.6). They capture the same information as N_F, N_T, N_C but have different dimensions and have a strict condition on the node order. Both N'_F and N'_T have 2^{D-1} elements and the values correspond to internal nodes generated by level order tree traversal. N'_C has 2^D elements with each corresponding to an actual leaf node from left to right order.

Given these tensors, in Algorithm 5 we present how `PTT` works. From a high-level point of view, it is very similar to the `TT` strategy with only a few changes. First, the index tensor T_I is initialized to all ones as the root node is always the first node. Second, we get rid of finding the

Algorithm 5 `PTT` Strategy (Notation in Tables 5.6)

Input: $X \in \mathbb{R}^{n \times |F|}$, Input records
Output: $R \in \{0,1\}^{n \times |C|}$, Predicted class labels
// Initialize all records to point to the root node.
$T_I \leftarrow \{1\}^n$ //$T_I \in \mathbb{Z}^n$
for $i \leftarrow 1$ **to** `TREE_DEPTH` **do**
    // Find the index of the feature evaluated by the current node. Then find its value.
    $T_F \leftarrow$`Gather`(N_F, T_I) //$T_F \in \mathbb{Z}^n$
    $T_V \leftarrow$`Gather`(X, T_f) //$T_V \in \mathbb{R}^n$
    // Find the threshold
    $T_T \leftarrow$`Gather`(N_T, T_I) //$T_T \in \mathbb{R}^n$
    // Perform logical evaluation. If true pick left child; else right child.
    $T_I \leftarrow 2 \times T_I +$ `Where`$(T_V < T_T, 0, 1)$ //$I \in \mathbb{Z}^n$
// Find label for each leaf node
$R \leftarrow$`Gather`(N'_C, T_I) //$*R \in \mathbb{Z}^n$

left index and right index of a node and using them in the `Where` operation. Instead, the `Where` operation returns 0 for true case and 1 for the false case. By adding this to $2 \times T_I$ we get the index of the child for the next iteration. For ensembles, we use the maximum `TREE_DEPTH` of any tree as D for transforming trees to perfect trees. We create tensors separate for each tree and batch them together for N'_C. But for N'_F and N'_T instead of batching, we interleave them together in some order such that values corresponding to level i for all trees appear before values corresponding to level $i+1$ of any tree.

5.4.2 Summary of Other Techniques

Next, we discuss the other techniques used across classical ML operators to efficiently translate them into tensor computations.

Exploiting Automatic Broadcasting. Broadcasting [36] is the process of making two tensors shape compatible for element-wise operations. Two tensors are said to be shape compatible if each dimension pair is the same, or one of them is 1. At execution time, tensor operations implicitly repeat the size 1 dimensions to match the size of the other tensor, without allocating

memory. In HUMMINGBIRD, we heavily use this feature to execute some computation over multiple inputs. For example, consider performing a one-hot encoding operation over column $X_i \in \mathbb{R}^n$ with a vocabulary $V \in \mathbb{Z}^m$. In order to implement this using tensor computations, we `Reshape` X_i to $[n,1]$ and V to $[1,m]$ and calculate $R = \texttt{Equal}(X,V)$, $R \in \{0,1\}^{n \times m}$. The `Reshape` operations are for free because they only modify the metadata of the tensor. However, this approach performs redundant comparisons as it checks the feature values from all records against all vocabulary values.

Minimize Operator Invocations. Given two approaches to implement a classical ML operator, we found that often picking the one which invokes fewer operators outperforms the other—even if it performs some extra computations. Consider a featurizer that generates feature interactions. Given an input $X \in \mathbb{R}^{n \times d}$, with $d = |F|$, it generates a transformed output $R \in \mathbb{R}^{n \times \frac{d \cdot (d+1)}{2}}$ with $R_i = [X_{i,1}^2, ..., X_{i,d}^2, X_{i,1}X_{i,2}, ...X_{i,d-1}X_{i,d}]$. One way to implement this operator is to compute each new feature separately by first `Gather`ing the corresponding input feature columns, perform an element-wise `Mul`tiplication, and con`Cat`enate all new features. However, this approach requires performing $d^2 + d + 1$ operations and hence is highly inefficient due to high operator invocation overheads. Alternatively, one could implement the same operator as follows. First, `Reshape` X into $X' \in \mathbb{R}^{n \times d \times 1}$ and $X'' \in \mathbb{R}^{n \times 1 \times d}$. Then perform a batched `GEMM` using these inputs, which will create $R' \in \mathbb{R}^{n \times d \times d}$. Finally, `Reshape` R' to $R'' \in \mathbb{R}^{n \times d^2}$. Notice that each row in R'' has all the values of the corresponding row in R, but in a different order. It also has some redundant values due to commutativity of multiplication (i.e., $x_i x_j = x_j x_i$). Hence, we perform a final `Gather` to extract the features in the required order, and generate R. Compared to the previous one, this approach increases both the computation and the memory footprint roughly by a factor of two. However, we can implement feature interaction in just two tensor operations.

Avoid Generating Large Intermediate Results. Automatic broadcasting in certain cases can become extremely inefficient due to the materialization of large intermediate tensors. Consider the Euclidean distance matrix calculation, which is popular in many classical ML

operators (e.g., SVMs, KNN). Given two tensors $X \in \mathbb{R}^{n \times d}$ and $Y \in \mathbb{R}^{m \times d}$, the objective is to calculate a tensor $D \in \mathbb{R}^{n \times m}$, where $D_{i,j} = ||X_i - Y_j||_2^2$. Implementing this using broadcasting requires first reshaping X to $X' \in \mathbb{R}^{n \times 1 \times d}$, Y to $Y' \in \mathbb{R}^{1 \times m \times d}$, calculate $(X' - Y') \in \mathbb{R}^{n \times m \times d}$, and perform a final `Sum` over the last dimension. This approach causes a size blowup by a factor of d in intermediate tensors. Alternatively, a popular trick [59] is to use the quadratic expansion of $D_{i,j} = ||X_i||_2^2 + ||Y_j||_2^2 - 2 \cdot X_i Y_j^T$ and calculate the individual terms separately. This avoids generating large intermediate tensors.

Fixed Length Restriction on String Features. Features with strings of arbitrary lengths pose a challenge for HUMMINGBIRD. Strings are commonly used in categorical features, and operators like one-hot encoding and feature hashing natively support strings. To support string features, HUMMINGBIRD imposes a fixed length restriction, with the length being determined by the max size of any string in the vocabulary. Vocabularies are generated during training and can be accessed at translation time by HUMMINGBIRD. Fixed length strings are then encoded into an `int8`.

5.5 Optimizations

In this section we discuss the key optimizations performed by the HUMMINGBIRD's Optimizer: heuristics for picking operator strategies (Section 5.5.1) and target-independent optimizations (Section 5.5.2). Recall that our approach also leverages target-specific optimizations at the DL Compiler level. We refer to [24, 93] for runtime-specific optimizations.

5.5.1 Heuristics-based Strategy Selection

For a given classical ML operator, there can be more than one translation strategy available. In the previous section we explained three such strategies for tree-based models. In practice, no strategy consistently dominates the others, but each is preferable in different situations based on

the input and model structure. For instance, the GEMM strategy gets significantly inefficient as the size of the decision trees gets bigger because of the large number of redundant computations. This strategy performs $O(2^D)$ (D is the depth of the tree) computations whereas the original algorithmic operator needs to perform only $O(D)$ comparisons. Nevertheless, with small batch sizes or a large number of smaller trees, this strategy can be performance-wise optimal on modern hardware, where GEMM operations can run efficiently. With large batch sizes and taller trees, TT techniques typically outperform the GEMM strategy and PTT is slightly faster than vanilla TT due to the reduced number of memory accesses. But if the trees are too deep, we cannot implement PTT because the $O(2^D)$ memory footprint of the associated data structures will be prohibitive. In such cases, we resort to TT. The exact crossover point where GEMM strategy outperforms other strategies is determined by the characteristics of the tree model (e.g., number of trees, maximum depth of the trees), runtime statistics (e.g., batch size), and the underlying hardware (e.g., CPUs, GPUs). For instance, from our experiments (see Figure 5.9) we found that the GEMM strategy performs better for shallow trees ($D \leq 3$ on CPU, ≤ 10 on GPU) or for scoring with smaller batch sizes. For tall trees, using PTT when $D \leq 10$ give a reasonable trade-off between memory footprint and runtime, which leaves vanilla TreeTraversal the only option for very tall trees ($D > 10$). These heuristics are currently hard-coded.

5.5.2 Target-independent Optimizations

We discuss two novel optimizations, which are unique to HUMMINGBIRD. HUMMINGBIRD's approach of separating the prediction pipeline from training pipeline, and representing them in a logical DAG before translation into tensor computations facilitate the optimization of end-to-end pipelines.

Feature Selection Push-Down. Feature selection is a popular operation that is often used as the *final featurization step* as it reduces over-fitting and improves the accuracy of the classical ML model [105]. However, during inference, it can be pushed down in the pipeline

to avoid redundant computations such as scaling and one-hot encoding for discarded features or even reading the feature at all. This idea is similar to the concept of projection push-down in relation query processing [272] but through user-defined table functions, which in our case are the classical ML operators. For operators such as feature scaling, which performs 1-to-1 feature transformations, selection push-down can be easily implemented. However, for operators such as one-hot encoding and polynomial featurization, which perform 1-to-m or m-to-1 feature transformations, the operator will have to absorb the feature selection and stop generating those features. For example, say one-hot encoding is applied on a categorical feature column which has a vocabulary size of 10, but 4 of those features are discarded by the feature selector. In such cases, we can remove such features from the vocabulary. Note that for some "blocking" operators [182], such as normalizers, it is not possible to push-down the feature selection.

Feature Selection Injection. Even if the original pipeline doesn't have a feature selection operator, it is possible to inject one and then push it down. Linear models with L1 regularization (Lasso) is a typical example where feature selection is implicitly performed. The same idea can be extended to tree-based models to prune the features that are not used as decision variables. In both of these examples, the ML model also has to be updated to take into account the pruned features. For linear models we prune the zero weights; for tree models, we update the indices of the decision variables.

5.6 Experimental Evaluation

In our experimental evaluation we report two micro-benchmark experiments showing how HUMMINGBIRD performs compared to current state-of-the-art for inference over (1) tree ensembles (Section 5.6.1); (2) other featurization operators and classical ML models (Section 5.6.1). Then we evaluate the optimizations by showing: (1) the need for heuristics for picking the best tree-model implementation (Section 5.6.2); and (2) the benefits introduced by the target-

independent optimizations (Section 5.6.2). Finally, we conduct an end-to-end evaluation using pipelines (Section 5.6.3). We evaluate both CPUs and hardware accelerators (GPUs).

Hardware and Software Setup. For all the experiments (except when stated otherwise) we use an Azure NC6 v2 machine equipped with 112 GB of RAM, an Intel Xeon CPU E5-2690 v4 @ 2.6GHz (6 virtual cores), and an NVIDIA P100 GPU. The machine runs Ubuntu 18.04 with PyTorch 1.3.1, TVM 0.6, scikit-learn 0.21.3, XGBoost 0.9, LightGBM 2.3.1, ONNX runtime 1.0, RAPIDS 0.9, and CUDA 10. We run TVM with `opt_level` 3 when not failing; 0 otherwise.

Experimental Setup. We run all the experiments 5 times and report the truncated mean (by averaging the middle values) of the processor time. In the following, we use ONNX-ML to indicate running an ONNX-ML model (i.e., traditional ML part of the standard) on the ONNX runtime. Additionally, we use **bold numbers** to highlight the best performance for the specific setup (CPU or GPU). Note that both scikit-learn and ONNX-ML do not natively support hardware acceleration.

5.6.1 Micro-benchmarks

Tree Ensembles

Setup. This experiment is run over a set of popular datasets used for benchmarking gradient boosting frameworks [37]. We first do a 80%/20% train/test split over each dataset. Successively, we train a scikit-learn *random forest*, *XGBoost* [92], and *LightGBM* [159] models using the default parameters of the benchmark. Specifically, we set the number of trees to 500 and maximum depth to 8. For XGBoost and LightGBM we use the scikit-learn API. Note that each algorithm generates trees with different structures, and this experiment helps with understanding how HUMMINGBIRD behaves with various tree types and dataset scales. For example, XGBoost generates balanced trees, LightGBM mostly generates skinny tall trees, while random forest is a mix between the two. Finally, we score the trained models over the test dataset using different

Table 5.7: Batch Experiments (10K records at-a-time) for both CPU (6 cores) and GPU. Reported numbers are in seconds.

Algorithm	Dataset	Baselines (CPU)		HUMMINGBIRD CPU			Baselines (GPU)	HUMMINGBIRD GPU	
		Sklearn	ONNX-ML	PyTorch	TorchScript	TVM	RAPIDS	TorchScript	TVM
Rand. Forest	Fraud	**2.5**	7.1	8.0	7.8	3.0	N/A	0.044	**0.015**
	Epsilon	9.8	18.7	14.7	13.9	**6.6**	N/A	**0.13**	**0.13**
	Year	1.9	6.6	7.8	7.7	**1.4**	N/A	0.045	**0.026**
	Covtype	**5.9**	18.1	17.22	16.5	6.8	N/A	0.11	**0.047**
	Higgs	**102.4**	257.6	314.4	314.5	118.0	N/A	1.84	**0.55**
	Airline	1320.1	timeout	timeout	timeout	**1216.7**	N/A	18.83	**5.23**
LightGBM	Fraud	3.4	5.9	7.9	7.6	**1.7**	**0.014**	0.044	**0.014**
	Epsilon	10.5	18.9	14.9	14.5	**4.0**	0.15	0.13	**0.12**
	Year	5.0	7.4	7.7	7.6	**1.6**	**0.023**	0.045	0.025
	Covtype	51.06	126.6	79.5	79.5	**27.2**	N/A	0.62	**0.25**
	Higgs	198.2	271.2	304.0	292.2	**69.3**	0.59	1.72	**0.52**
	Airline	1696.0	timeout	timeout	timeout	**702.4**	5.55	17.65	**4.83**
XGBoost	Fraud	1.9	5.5	7.7	7.6	**1.6**	**0.013**	0.44	0.015
	Epsilon	7.6	18.9	14.8	14.8	**4.2**	0.15	0.13	**0.12**
	Year	3.1	8.6	7.6	7.6	**1.6**	**0.022**	0.045	0.026
	Covtype	42.3	121.7	79.2	79.0	**26.4**	N/A	0.62	**0.25**
	Higgs	126.4	309.7	301.0	301.7	**66.0**	0.59	1.73	**0.53**
	Airline	1316.0	timeout	timeout	timeout	**663.3**	5.43	17.16	**4.83**

batch sizes. We compare the results against HUMMINGBIRD with different DL inference system backends and an ONNX-ML version of the model generated using ONNXMLTools [33]. When evaluating over GPU, we also compared against NVIDIA RAPIDS Forest Inference Library (FIL) [45]. We don't compare against GPU implementations for XGBoost or LightGBM because we consider FIL as state-of-the-art [34]. For the CPU experiments, we use all six cores in the machine, while for request/response experiments we use one core. We set a timeout of 1 hour for each experiment.

Datasets. We use 6 datasets from NVIDIA's gbm-bench [37]. The datasets cover a wide spectrum of use-cases: from regression to multiclass classification, from 285K rows to 100M, and from few 10s of columns to 2K.

List of Experiments. We run the following set of experiments: (1) batch inference, both on CPU and GPU; (2) request/response where a single record is used for inference; (3) scaling experiments by varying batch sizes, both over CPU and GPU; (4) evaluation of how

HUMMINGBIRD behaves on different GPU generations; (5) dollar cost per prediction; (6) memory consumption; (7) validation of the produced output with respect to scikit-learn; and finally (8) time spent on translating the models.

Batch Inference. Table 5.7 reports the inference time for random forest, XGBoost and LightGBM models run over the 6 datasets. The batch size is set to $10K$ records. Looking at the CPU numbers from the table, we can see that:

1. Among the baselines, scikit-learn models outperform ONNX-ML implementations by 2 to 3×. This is because ONNX-ML v1.0 is not optimized for batch inference.

2. Looking at the HUMMINGBIRD's backends, there is not a large difference between PyTorch and TorchScript, and in general these backends perform comparable to ONNX-ML.

3. The TVM backend provides the best performance on 15 experiments out of 18. In the worst case TVM is 20% slower (than scikit-learn); in the best cases it is up to 2× faster compared to the baseline solutions.

Let us look now at the GPU numbers of Table 5.7:

1. Baseline RAPIDS does not support random forest nor multiclass classification tasks. For the remaining experiments, GPU acceleration is able to provide speedups of up to 300× compared to CPU baselines.[1]

2. Looking at HUMMINGBIRD backends, TorchScript is about 2 to 3× slower compared to RAPIDS. TVM is instead the faster solution on 14 experiments out of 18, with a 10% to 20% improvement wrt RAPIDS.

[1]The original FIL blog post [34] claims GPU acceleration to be in the order of 28× for XGBoost, versus close to 300× in our case (Airline). We think that the difference is in the hardware: in fact, they use 5 E5-2698 CPUs for a total of 100 physical cores, while we use a E5-2690 CPU with 6 (virtual) physical cores. Additionally, they use a V100 GPU versus a P100 in our case.

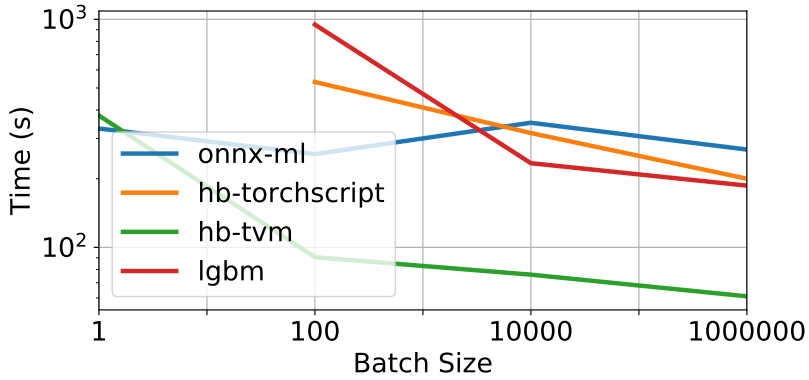


Figure 5.4: Performance with respect to scaling the batch size on CPU (Higgs, LightGBM), 6 cores

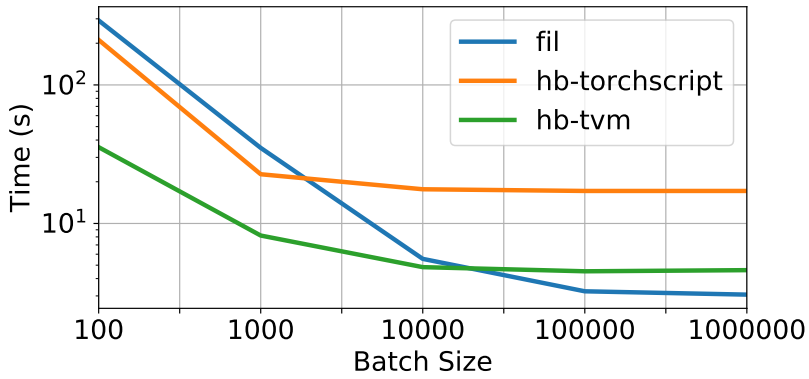


Figure 5.5: Performance with respect to scaling the batch size on GPU (Airline, LightGBM)

The results are somehow surprising: HUMMINGBIRD targets the high-level tensor APIs provided by PyTorch and TVM, and still it is able to outperform custom C++ and CUDA implementations.

Request/response.

In this scenario, one record is used for inference at a time. For this experiment we run inference over the entire test datasets, but with batch size equal to 1. We used the same datasets and setup of Section 5.6.1, except that (1) we removed the Airline dataset since no system was able to complete within the 1 hour timeout; and (2) we only use one single core. The results are depicted in Table 5.8:

1. Unlike the batch scenario, ONNX-ML is much faster compared to scikit-learn, in some

Table 5.8: Request/response times in seconds (one record at a time).

Algorithm	Dataset	Baselines		HUMMINGBIRD		
		Sklearn	ONNX-ML	PT	TS	TVM
Rand. Forest	Fraud	1688.22	**9.96**	84.95	75.5	11.63
	Epsilon	2945.42	32.58	153.32	134.17	**20.4**
	Year	1152.56	18.99	84.82	74.21	**9.13**
	Covtype	3388.50	35.49	179.4	157.8	**34.1**
	Higgs	timeout	**335.23**	timeout	timeout	450.65
LightGBM	Fraud	354.27	12.05	96.5	84.56	**10.19**
	Epsilon	40.7	29.28	167.43	148.87	**17.3**
	Year	770.11	16.51	84.55	74.05	**9.27**
	Covtype	135.39	209.16	854.07	822.93	**42.86**
	Higgs	timeout	**374.64**	timeout	timeout	391.7
XGBoost	Fraud	79.99	**7.78**	96.84	84.61	10.21
	Epsilon	121.21	27.51	169.03	148.76	**17.4**
	Year	98.67	17.14	85.23	74.62	**9.25**
	Covtype	135.3	197.09	883.64	818.39	**43.65**
	Higgs	timeout	585.89	timeout	timeout	**425.12**

Table 5.9: Peak memory consumption (in MB) for Fraud.

Framework	Random Forest	LightGBM	XGBoost
Sklearn	180	182	392
ONNX-ML	265	258	432
TorchScript	375	370	568
TVM	568	620	811

cases even more than 100×. The reason is that ONNX-ML is currently optimized for single record, single core inference, whereas scikit-learn design is more towards batch inference.

2. PyTorch and TorchScript, again, behave very similarly. For random forest they are faster than scikit-learn but up to 5× slower compared to ONNX-ML. For LightGBM and XGBoost they are sometimes on par with scikit-learn, sometime slower.

3. TVM provides the best performance in 11 cases out of 15, with a best case of 3× compared to the baselines.

These results are again surprising, considering that tensor operations should be more optimized for bulk workloads rather than request/response scenarios.

Scaling the Batch Size. We study how the performance of baselines and HUMMINGBIRD's backends change with the batch size. Figures 5.4 and 5.5 depicts the performance variation over CPU and GPU, respectively. We report only a few combinations of dataset / algorithm, but all the other combinations behave similarly. Starting with the CPU experiment, we can see that ONNX-ML has the best runtime for batch size of 1, but then its performance remains flat as we increase the batch size. TorchScript and scikit-learn did not complete within the timeout for batch equal to 1, but, past 100, they both scale linearly as we increase the batch size. TVM is comparable to ONNX-ML for batch of 1; for batches of 100 records it gets about 5× faster, while it scales like TorchScript for batches greater than 100. This is likely due to the fact that TVM applies a set of optimizations (e.g., operator fusion) that introduce a constant-factor speedup compared to TorchScript.

Looking at the GPU numbers (Figure 5.5), TorchScript and TVM again follow a similar trend, with TVM being around 3× faster than TorchScript. Both TVM and TorchScript plateau at about a batch size of $10K$. RAPIDS FIL is slower than TorchScript for small batch sizes, but it scales better than HUMMINGBIRD. This is because of its custom CUDA implementation that is able to better use hardware under higher utilization. Interestingly, FIL as well plateaus at around $100K$ records. The custom CUDA implementation introduces a 50% gain over HUMMINGBIRD with TVM runtime over large batches.

Scaling Hardware. We tested how RAPIDS FIL and HUMMINGBIRD (TorchScript and TVM) scale as we change the GPU model. For this experiment we tried both with a large batch size ($1M$ records, Figure 5.6 (a)) to maximize hardware utilization, and a smaller batch size ($1K$, Figure 5.7 (b)). We ran this on all datasets across random forest, LightGBM, XGBoost with similar results, and present the Airline dataset (the largest) with LightGBM as a representative sample. We tested on three NVIDIA devices: K80 (the oldest, 2014), P100 (2016), and V100 (2017). From the figures, in general we can see that: (1) RAPIDS FIL does not run on the K80 because it is an old generation; (2) with a batch size of 1K we get slower total inference time

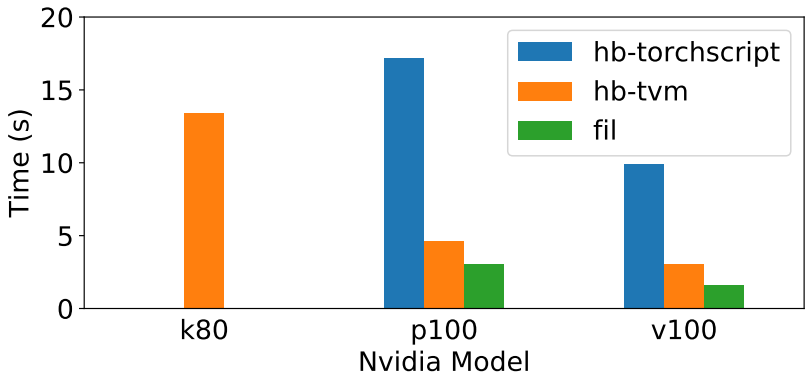


Figure 5.6: Performance across GPUs for Airline, LightGBM with batch size of 1*M*

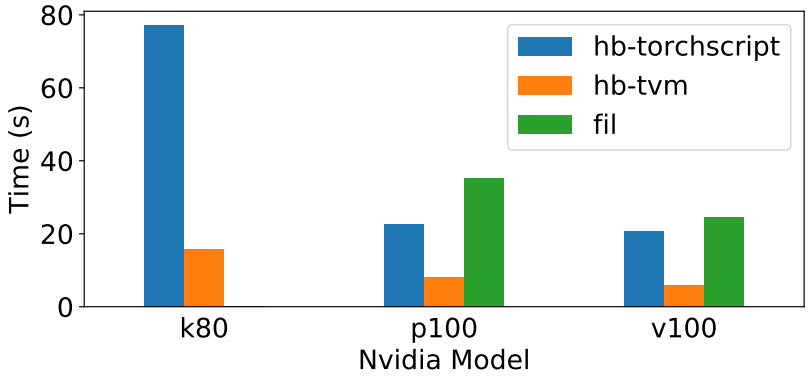


Figure 5.7: Performance across GPUs for Airline, LightGBM with batch size of 1*K*

because we don't utilize the full hardware; (3) TorchScript and TVM runtimes for HUMMINGBIRD scale similarly on different hardware, although TVM is consistently 4 to 7× faster; (4) FIL scales similarly to HUMMINGBIRD, although it is 50% faster on large batches, 3× slower for smaller batches; (5) TorchScript is not optimal in memory management because for batches of 1*M* it fails on the K80 with an OOM exception. Finally, we also were able to run HUMMINGBIRD on the new Graphcore IPU [30] over a single decision tree.

Cost. Figure 5.8 shows the cost comparison between the Azure VM instance equipped with GPU, and a comparable one without GPU (E8 v3). The plot shows the cost of executing 100k samples with a batch size of 1K for random forest. The cost is calculated based on the hourly rate of each VM divided by the amortized cost of a single prediction. We executed scikit-learn on the CPU and TorchScript and TVM on the GPU for comparison. We found that the CPU cost was

Table 5.10: Conversion times (in seconds) over one core.

Algorithm	Dataset	ONNX-ML	HUMMINGBIRD		
			PyTorch	TorchScript	TVM
Rand.Forest	Fraud	1.28	0.55	0.58	102.37
	Epsilon	7.53	2.63	2.67	108.64
	Year	7.11	2.77	2.86	69.99
	Covtype	9.87	2.16	2.2	106.8
	Higgs	8.25	2.41	2.44	103.77
	Airline	6.82	2.42	2.53	391.07
LightGBM	Fraud	1.34	0.98	1.06	3.42
	Epsilon	11.71	7.55	7.60	9.95
	Year	9.49	6.11	6.15	8.35
	Covtype	32.46	22.57	23.12	26.36
	Higgs	6.73	25.04	26.3	109
	Airline	11.52	6.38	6.47	8.19
XGBoost	Fraud	0.55	0.65	0.7	86.59
	Epsilon	6.86	25.89	25.94	113.4
	Year	5.66	23.4	23.54	110.24
	Covtype	9.87	2.16	2.20	106.8
	Higgs	6.73	25.04	26.3	109

significantly higher (between 10×-120×) across all experiments. [2] An interesting result was that the oldest GPU was the most cost effective, with the K80 and TVM having the lowest cost for 13 out of the 18 experiments (including LightGBM and XGBoost, not pictured). This result is explained by the fact that the K80 is readily available at significantly lower cost.

Memory Consumption. We measured the peak memory consumption over the Fraud dataset and for each algorithm. We used the `memory_usage` function in the `memory_profiler` library [40]. The numbers are reported in Table 5.9, and are the result of the execution over 1 core with a batch size of $1K$. As we can see, scikit-learn is always the most memory efficient. ONNX-ML consumes from 10% to 50% more memory, while HUMMINGBIRD with TorchScript runtime consumes from 50% to about 2× more memory than scikit-learn. Conversely, TVM consumes from 2× to 3× more memory with respect to scikit-learn. We think that TVM is more

[2] Note: airline times out for random forest for CPU with $1K$ batch.

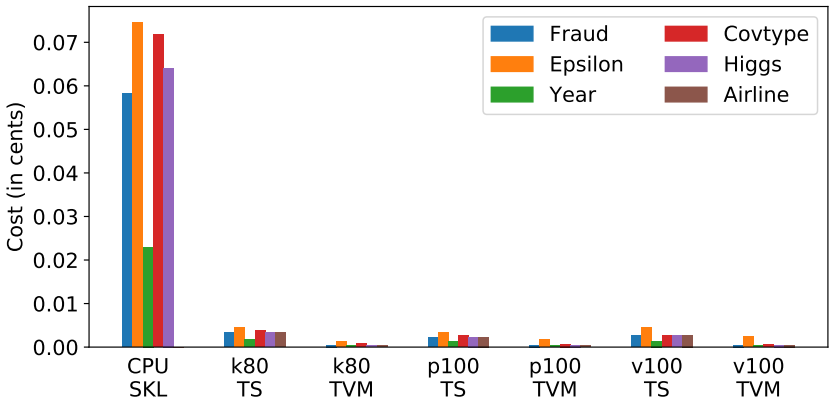


Figure 5.8: Cost for random forest 100k samples, batch size of $1K$.

memory hungry because it optimizes compute at the cost of memory requirements. Note that the batch size influences the total memory consumption.

Output Validation. Since we run tree ensemble models as tensor operations, we could introduce rounding errors over floating point operations. Therefore, we need to validate that indeed the outputs produced match. To evaluate this, we used the `numpy testing.assert_allclose` function, and we set the relative and absolute errors to 10^{-5}. We validate both the final scores and the probabilities (when available) for all combinations of datasets and algorithms. Out of the 18 experiments listed in Table 5.7, 9 of them returned no mismatches for HUMMINGBIRD, 12 in the ONNX-ML case. Among the mismatches, the worst case for HUMMINGBIRD is random forest with Covtype where we have 0.8% of records differing from the original scikit-learn output. For the Epsilon dataset, HUMMINGBIRD with random forest returns a mismatch on 0.1% of records. All the remaining mismatches effect less than 0.1% of records. Note that the differences are small. The biggest mismatch is of 0.086 (absolute difference) for Higgs using LightGBM. For the same experiment ONNX-ML has an absolute difference of 0.115.

Conversion Time. Table 5.10 shows the time it takes to convert a trained model into a target DL system. The numbers are related to the generation of models running on a single core. This cost occurs only once per model and are not part of the inference cost. As we can see,

converting a model to ONNX-ML can take up to a few tens of seconds; HUMMINGBIRD with PyTorch backend is constantly about 2× to 3× faster with respect to ONNX-ML in converting random forests models, while it varies for LightGBM and XGBModels. TorchScript models are generated starting from PyTorch models, and in general this further conversion step does not introduce any major overhead. Finally, conversion to TVM is much slower, and it might take more than 3 minutes. This is due to code generation and optimizations introduced in TVM.

As a final note: parallel (i.e., more than 1 core) and GPU execution introduced further conversion time overheads, especially on TVM. For instance, TVM can take up to 40 minutes to convert a random forest model for execution on GPU.

Operators

Setup. This micro-benchmark is a replication of the suite comparing scikit-learn and ONNX-ML operators [32]. We test all scikit-learn operators of the suite that are supported by both ONNX-ML and HUMMINGBIRD (minus tree ensembles models). The total number of tested operators is 13, and they are a mix of classical ML models (Logistic Regression, Support Vector Machines, etc.) and featurizers (e.g., Binarizer, Polynomial, etc.). For this micro-benchmark we score 1 million records.

Datasets. We use the Iris datasets [39] with 20 features.

List of Experiments. We run the following experiments: (1) batch inference over $1M$ records, both on CPU and GPU; (2) request/response over 1 record; (3) memory consumption and conversion time. All the output results are correct.

Batch Inference. The batch numbers are reported in Table 5.11. On CPU, scikit-learn is faster than ONNX-ML, up to 6× for polynomial featurizer, although in most of the cases the two systems are within a factor of 2. HUMMINGBIRD with TorchScript backend is competitive with scikit-learn, whereas with TVM backend HUMMINGBIRD is faster on 8 out of 13 operators, in

Table 5.11: Batch experiments for operators on both CPU (1 core) and GPU. Numbers are in milliseconds. (TS is short for TorchScript)

Operator	Baselines (CPU)		HUMMINGBIRD CPU		HUMMINGBIRD GPU	
	Sklearn	ONNX-ML	TS	TVM	TS	TVM
Log. Regres.	970	1540	260	**47**	**13**	15
SGDClass.	180	1540	270	**49**	**11**	15
LinearSVC	110	69	260	**51**	**12**	18
NuSVC	3240	4410	**2800**	3000	140	**72**
SVC	1690	2670	**1520**	1560	120	**41**
BernoulliNB	280	1670	290	**65**	**12**	14
MLPClassifier	930	1860	**910**	1430	**17**	31
Dec.TreeClass.	59	1610	560	**35**	**13**	16
Binarizer	98	75	**39**	59	**38**	**38**
MinMaxScaler	92	200	78	**57**	**38**	**38**
Normalizer	94	140	**83**	97	**39**	40
Poly.Features	4030	29160	6380	**3130**	**340**	error
StandardScaler	150	200	77	**58**	**38**	**38**

general a speedup of about 2× compared to scikit-learn. If now we focus to the GPU numbers, we see that HUMMINGBIRD with TorchScript backend compares favorably against TVM on 11 operators out of 13. This is in contrast with the tree ensemble micro-benchmark where the TVM backend was faster than the TorchScript one. We suspect that this is because TVM optimizations are less effective on these "simpler" operators. For the same reason, GPU acceleration does not provide the speedup we saw for the tree ensemble models. In general, we see around 2× performance improvement over the CPU runtime: only polynomial featurizer runs faster, with almost a 10× improvement. TVM returns a runtime error when generating the polynomial featurizer model on GPU.

Request/response. Table 5.12 contains the times to score 1 record. The results are similar to the request/response scenario for the tree ensemble micro-benchmark. Namely, ONNX-ML outperform both scikit-learn and HUMMINGBIRD in 9 out of 13 cases. Note, however, that all frameworks are within a factor of 2. The only outlier is polynomial featurizer which is about 10× faster on HUMMINGBIRD with TVM backend.

Table 5.12: Request/Response experiments for operators on CPU (single core). Reported numbers are in milliseconds.

Operator	Baselines		HUMMINGBIRD	
	Sklearn	ONNX-ML	TS	TVM
LogisticRegression	0.087	**0.076**	0.1	0.1
SGDClassifier	**0.098**	0.1	0.12	0.1
LinearSVC	0.077	**0.05**	0.11	0.1
NuSVC	0.086	**0.072**	4.1	0.14
SVC	0.086	**0.074**	2.3	0.12
BernoulliNB	0.26	**0.1**	0.07	0.11
MLPClassifier	0.15	0.11	**0.1**	0.12
DecisionTreeClassifier	0.087	**0.074**	0.44	0.12
Binarizer	0.064	**0.053**	0.063	0.1
MinMaxScaler	0.066	0.060	**0.058**	0.1
Normalizer	0.11	**0.063**	0.072	0.1
PolynomialFeatures	1.2	1	0.5	**0.1**
StandardScaler	0.069	**0.048**	0.059	0.1

Memory Consumption and Conversion Time. We measured the peak memory consumed and conversion time for each operator on each DL system. We used batch inference over 1K records. For memory consumption, the results are in line with what we already saw in Section 5.6.1. Regarding the conversion time, for ONNX-ML and HUMMINGBIRD with TorchScript, the conversion time is in the order of few milliseconds. The TVM backend is slightly slower but still in the order of few tens of milliseconds (exception for NuSVC and SVC which take up to 3.2 seconds). In comparison with the tree ensembles numbers (Table 5.10), we confirm that these operators are simpler, even from a translation perspective.

5.6.2 Optimizations

Tree Models Translation

Next we test the different tree-based models implementation to make the case for the heuristics.

Datasets. For this experiment we employ a synthetic dataset randomly generated with

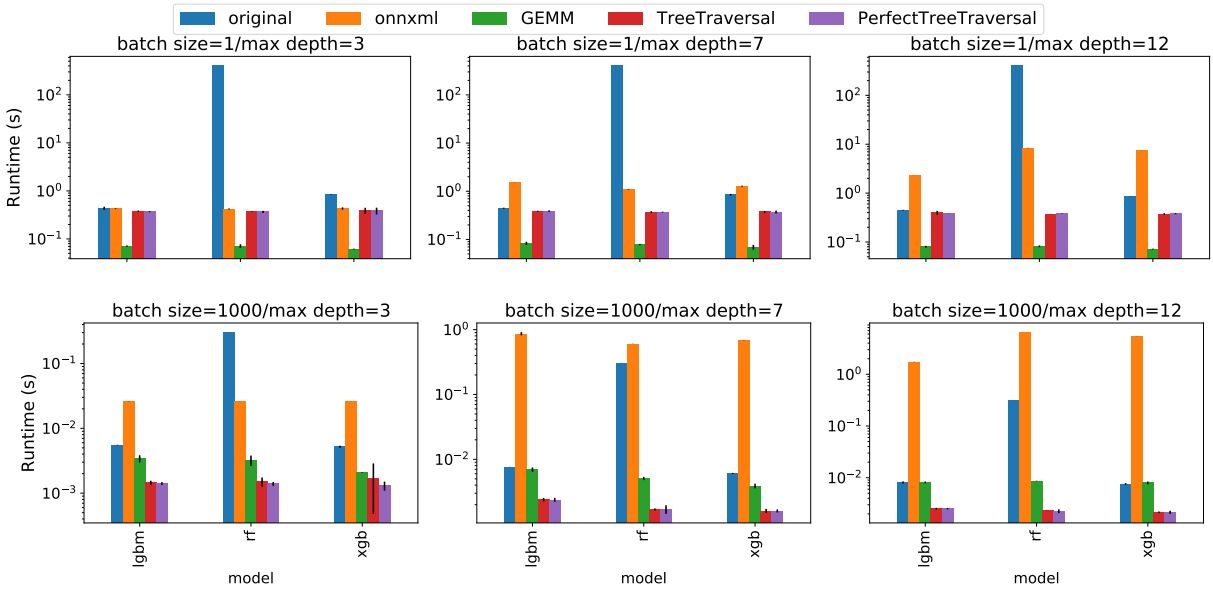


Figure 5.9: Comparison between the different tree strategies as we vary the batch size and depth.

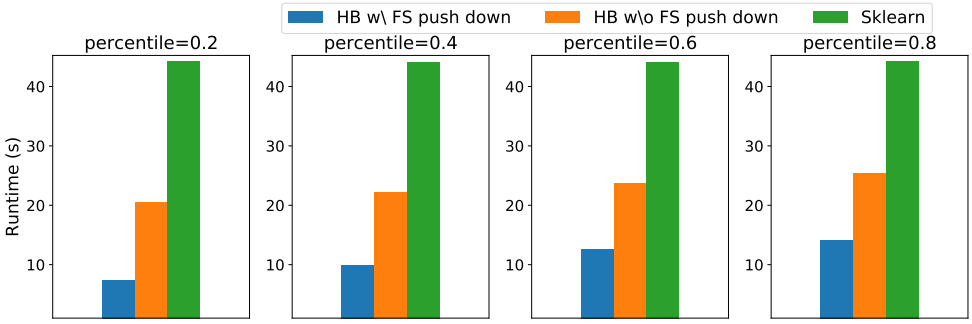


Figure 5.10: Feature selection push down.

5000 rows and 200 features.

Experiments Setup. We study the behavior of the tree implementations as we change the training algorithm, the batch size, and the tree depth. For each experiment we set the number of trees to 100. We use the TVM backend. Each experiment is run on 1 CPU core.

Results. Figure 5.9 shows the comparison between the different tree implementations, and the two scikit-learn and ONNX-ML baselines. In the top part of the figure we run all experiments using a batch size of 1; on the bottom part we instead use a batch size of $1K$. In the column on the left-hand side, we generate trees with a max depth of 3; 7 for the middle column, and 12 for

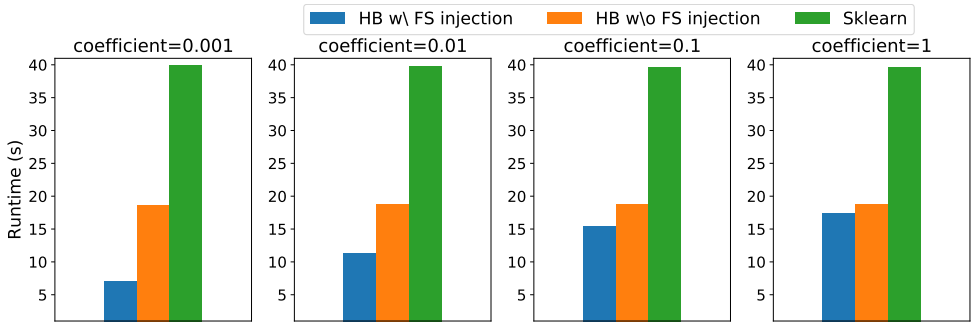


Figure 5.11: Feature selection injection.

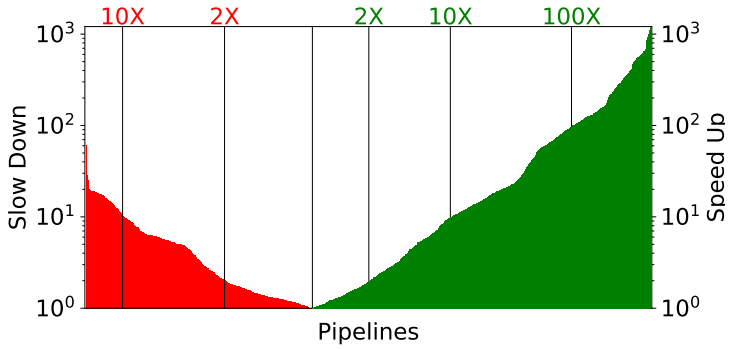


Figure 5.12: Speedup/slowdown of pipelines when using HUMMINGBIRD with respect to baseline Sklearn on CPU

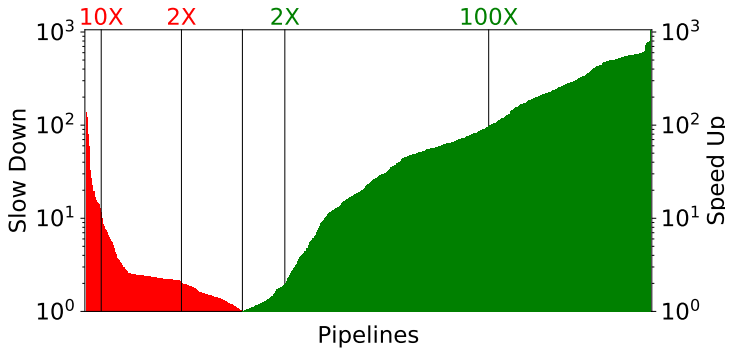


Figure 5.13: Speedup/slowdown of pipelines when using HUMMINGBIRD with respect to baseline Sklearn on GPU

column on the right-hand side. In general, two things are apparent: (1) HUMMINGBIRD is as fast as or better than the baselines; and (2) no tree implementation is always better than the others. The GEMM implementation outperforms the other two for small batch sizes, whereas TT and PTT are better over larger batch sizes. Between TT and PTT, the latter is usually the best performant

(although not by a large margin). PTT however creates balanced trees, and fails for very deep trees.

Target-independent Optimizations.

Next we test the optimizations described in Section 5.5.2.

Dataset. We use the Nomao dataset [41] with 119 features.

Feature Selection Push Down. In this experiment we measure the benefits of the feature selection push down. In Figure 5.10 we compare HUMMINGBIRD with and without feature selection push-down, and the baseline implementation of the pipelines in scikit-learn. We use a pipeline which trains a logistic regression model with L2 loss. The featurization part contains one-hot encoding for categorical features, missing value imputation for numerical values, followed by feature scaling, and a final feature selection operator (scikit-learn's `SelectKBest`). We vary the percentile of features that are picked by the feature selection operator. In general, we can see that HUMMINGBIRD without optimization is about 2× faster than scikit-learn in evaluating the pipelines. For small percentiles, the feature selection push-down optimization delivers a further 3×. As we increase the percentile of features that are selected, the runtime of HUMMINGBIRD both with and without optimizations increase, although with the optimization HUMMINGBIRD is still 2× faster than without.

Feature Selection Injection. In this experiment we evaluate whether we can improve the performance of pipelines with sparse models by injecting (and then pushing down) feature selection operators. The pipeline is the same as in the previous case but without the feature selection operator. Instead we train the logistic regression model with L1 regularization. In Figure 5.11 we vary the L1 regularization coefficient and study how much performance we can gain. Also in this case, with very sparse models we can see up to 3× improvement with respect to HUMMINGBIRD without optimization. Performance gains dissipate as we decrease the sparsity

of the model.

5.6.3 End-to-end Pipelines

Setup. In this experiment we test HUMMINGBIRD over end-to-end pipelines. We downloaded the 72 tasks composing the OpenML-CC18 suite [44]. Among all the tasks, we discarded all the "not pure scikit-learn" ML pipelines (e.g., containing also arbitrary Python code). We successively discarded all the pipelines returning a failure during training. 88% of the remaining pipelines are exclusively composed by operators supported by HUMMINGBIRD, for a total of 2328 ML pipelines. Among these, 11 failed during inference due to runtime errors in HUMMINGBIRD; we report the summary of executing 2317 pipelines. These pipelines contain an average of 3.3 operators, which is in line with what was observed in [233].

Datasets. For this experiment we have 72 datasets in total [44]. The datasets are a curated mix specifically designed for classical ML benchmarking. We did the typical 80%/20% split between training and inference. The smaller dataset has just 100 records, the bigger 19264, while the median value is 462. The minimum number of columns for a dataset is 4, the maximum 3072, with a median of 30.

Results. Figure 5.12 and Figure 5.13 summarize the speedup / slowdown introduced by HUMMINGBIRD when scoring all 2317 pipelines. As we can see, HUMMINGBIRD is able to accelerate about 60% of the pipelines on CPU (5.12). In general, the slowest pipeline gets about 60× slower with respect to scikit-learn, the fastest instead gets a 1200× speed up. The slowdowns are due to a couple of factors: (a) the datasets used for these experiments are quite small; (b) some pipelines contain largely sparse operations (i.e., SVM on sparse inputs); (c) several pipelines are small and do not require much computation (e.g., a simple inputer followed by a small decision tree). These three factors are highlighted also by the fact that even if we move computation to the GPU (5.13), still 27% of the pipelines have some slowdown. Note however

that (1) both sparse and small pipelines can be detected at translation time, and therefore we can return a warning or an error; (2) DL inference systems are continuously adding new sparse tensor operations (e.g., [49]); and (3) an option could be to add a specific DL system backend for sparse tensor operations (e.g., we have a prototype integration with TACO [164]). In general, DL inference systems are relatively young, and HUMMINGBIRD will exploit any future improvement with no additional costs.

With GPU acceleration (Figure 5.13), 73% of the pipelines show some speedup. The slowest pipeline gets about 130× slower with respect to scikit-learn, the fastest instead gets a speedup of 3 orders of magnitude. Some of the pipelines get worse from CPU to GPU execution. This is due to (1) sparsity; (2) small compute; and (3) data movements between CPU and GPU memory. Indeed we run all pipelines on GPU, even the ones for which in practice would not make much sense (e.g., a decision tree with 3 nodes). We leave as future work an extension to our heuristics for picking the right hardware backend.

5.7 Conclusion

Classical ML is widely used for structured data analytics in enterprises. Yet, establishing systems support for classical ML has not garnered the same attention as it has for DL, especially for inference workloads. Current bespoke system implementations for classical ML inference introduce significant infrastructure complexity and miss significant opportunities for optimization. In this work, we use query optimization-inspired techniques to translate classical ML pipelines to tensor computations. This approach enables us to leverage the DL inference systems for classical ML inference and also support heterogeneous hardware backends (e.g., CPUs, GPUs) for inference. The results are compelling: even though we target high-level tensor operations, we outperform custom C++ and CUDA implementations. To our knowledge, HUMMINGBIRD is the first system able to run classical ML inference on heterogeneous hardware.

Chapter 5 contains material from "A Tensor Compiler for Unified Machine Learning Prediction Serving" by Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi, which appears in Proceedings of 14^{th} USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020). The dissertation author was the primary investigator and author of this paper. The code for our system is open source and is available on GitHub: https://github.com/microsoft/Hummingbird.

Chapter 6

KRYPTON: Query Optimizations for Deep CNN Prediction Explanations

6.1 Introduction

In this chapter, we dive deeper into our techniques for optimizing occlusion-based deep CNN prediction explanations [290], or OBE for short. OBE is a popular technique for explaining CNN predictions and it works as follows: place a small square patch (usually gray) on the image to occlude those pixels. Rerun CNN inference, illustrated in Figure 6.1(a), on the occluded image. The probability of the predicted class will change, as Figure 6.1(b) shows. Repeat this process by moving the patch across the image to obtain a sensitivity *heatmap* of probability changes, as Figure 6.1(c) shows. This heatmap highlights regions of the image that were highly "responsible" for the prediction (red/orange color regions). Such localization of the regions of interest allows users to gain intuition on what "mattered" for the prediction. For instance, the heatmap can highlight the diseased areas of a tissue image, which a radiologist can then inspect more deeply for further tests. Overall, OBE is popular because it is easy for non-technical users to understand.

Alas, OBE is highly expensive computationally. Deep CNN inference is already expensive;

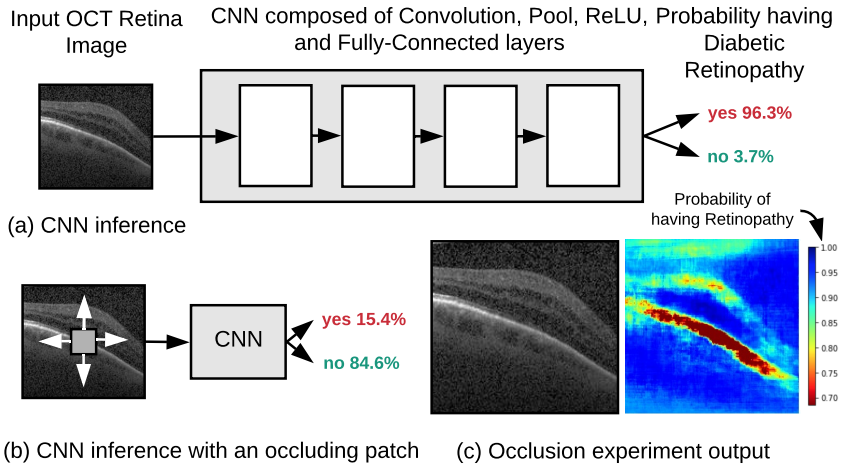


Figure 6.1: (a) Using a CNN to predict diabetic retinopathy in an OCT image/scan. (b) Occluding a part of the image changes the prediction probability. (c) By moving the occluding patch, a sensitivity heatmap can be produced.

OBE just amplifies it by issuing a large number of CNN re-inference requests (even 1000s). For example, [296] report 500,000 re-inference requests for 1 image, which took 1hr even on a GPU! Such long wait times can hinder users' ability to consume explanations and reduce their productivity. One could use more compute hardware, if available, since OBE is embarrassingly parallel across re-inference requests. But this may not always be affordable, especially for domain scientists, or feasible in all settings, e.g., in mobile clinical diagnosis. Extra hardware can also raise monetary costs, especially in the cloud.

In this work, we use a query optimizations-inspired lens to formalize, optimize, and accelerate OBE. We start with a simple but crucial observation: *the occluded images are not disjoint but share most of their pixels; so, most of CNN re-inference computations are redundant.* This observation leads us to connect OBE with two classical data management concerns: *incremental view maintenance* (IVM) and *multi-query optimization* (MQO). Instead of treating a CNN as a "blackbox," we open it up and formalize *CNN layers* as "queries." Just like how a relational query converts relations to other relations, a CNN layer converts *tensors* (multidimensional arrays) to other tensors. So, we reimagine OBE as *a set of tensor transformation queries* with incrementally updated inputs. With this fresh database-inspired view, we introduce several *novel CNN-specific*

query optimization techniques to accelerate OBE.

Our first optimization is *incremental inference*. We first *materialize* all tensors produced by the CNN. For every re-inference request, instead of rerunning inference from scratch, we treat it as an IVM query, with the "views" being the tensors. We rewrite such queries to *reuse* the materialized views as much as possible and recompute only what is needed, thus *avoiding computational redundancy*. Such rewrites are non-trivial because they are tied to the complex geometric dataflows of CNN layers. We formalize such dataflows to create a novel *algebraic rewrite framework*. We also create a "static analysis" routine to tell us up front how much computations can be saved. Going further, we batch all re-inference requests in OBE to reuse the *same* materialized views. This is a form of MQO, which we call *batched incremental inference*. We also create a GPU-optimized kernel for such execution. To the best of our knowledge, this is the first instance of IVM being combined with MQO in query optimization, at least for CNN inference.

We then introduce two novel *approximate inference* optimizations that allow users to tolerate some degradation in visual quality of the heatmaps produced to reduce runtimes further. These optimizations build upon our incremental inference optimization to trade off heatmap quality in a user-tunable manner. Our first approximate optimization, *projective field thresholding*, draws upon an idea from neuroscience and exploits the internal semantics of how CNNs work. Our second approximate optimization, *adaptive drill-down*, exploits the semantics of the OBE task and the way users typically consume the heatmaps produced. We also present intuitive automated parameter tuning methods to help users adopt these optimizations.

We prototype our ideas in the popular deep learning system PyTorch to create a tool we call KRYPTON. It works on both CPU and GPU and currently supports a few popular deep CNNs (VGG16, ResNet18, and InceptionV3). We perform a comprehensive empirical evaluation of KRYPTON with three real-world image datasets from recent radiology and computer vision papers. KRYPTON yields up to 35X speedups over the current dominant practice of running

re-inference with just batching for producing high-quality approximate heatmaps and up to 5X speedups for producing exact heatmaps. We then analyze the utility of each of our optimizations. Overall, this work makes the following contributions:

- To the best of our knowledge, this is the first work to formalize and optimize the execution of occlusion-based explanations (OBE) of CNN predictions from a data management standpoint.
- We cast OBE as an IVM problem to create a novel and comprehensive algebraic framework for incremental CNN inference. We also combine our IVM technique with an MQO-style technique to further reduce computational redundancy in CNN inference.
- We present two novel approximate inference optimizations for OBE that exploit the semantics of CNNs and properties of human perception.
- We prototype our ideas in a tool, KRYPTON, and perform an extensive empirical evaluation with real data and deep CNNs. KRYPTON speeds up OBE by even over an order of magnitude in some cases.

Outline. Section 6.2 explains our problem setup, assumptions, and CNN dataflow model. Section 6.3 (resp. Section 6.4) presents our incremental (resp. approximate) inference optimizations. Section 6.5 presents the experimental evaluation and we conclude in Section 6.6.

6.2 Setup and Preliminaries

We now state our problem formally and explain our assumptions. We then formalize the dataflow of the layers of a CNN, since these are required for understanding our techniques in Sections 6.3 and 6.4. Table 6.1 lists our notation.

Table 6.1: Notation used in this chapter.

Symbol	Meaning
f	Given deep CNN; input is an image tensor; output is a probability distribution over class labels
L	Class label predicted by f for the original image $I_{:img}$
$T_{:l}$	Tensor transformation function of layer l of the given CNN f
$\mathcal{P}$	Occlusion patch in RGB format
$S_{\mathcal{P}}$	Occlusion patch striding amount
G	Set of occlusion patch superimposition positions on $I_{:img}$ in (x,y) format
M	Heat map produced by the OBE workload
H_M, W_M	Height and width of M
$\circ_{(x,y)}$	Superimposition operator. $A \circ_{(x,y)} B$, superimposes B on top of A starting at (x,y) position
$I_{:l}$ ($I_{:img}$)	Input tensor of layer l (Input Image)
$O_{:l}$	Output tensor of layer l
$C_{I:l}, H_{I:l}, W_{I:l}$	Depth, height, and width of input of layer l
$C_{O:l}, H_{O:l}, W_{O:l}$	Depth, height, and width of output of layer l
$\mathcal{K}_{conv:l}$	Convolution filter kernels of layer l
$\mathcal{B}_{conv:l}$	Convolution bias value vector of layer l
$\mathcal{K}_{pool:l}$	Pooling filter kernel of layer l
$H_{\mathcal{K}:l}, W_{\mathcal{K}:l}$	Height and width of filter kernel of layer l
$S_{:l}; S_{x:l}; S_{y:l}$	Filter kernel striding amounts of layer l; $S_{:l} \equiv (S_{x:l}, S_{y:l})$, strides along width and height dimensions
$P_{:l}; P_{x:l}; P_{y:l}$	Padding amounts of layer l; $P_{:l} \equiv (P_{x:l}, P_{y:l})$, padding along width and height dimensions

6.2.1 Problem Statement and Assumptions

We are given a CNN f that has a sequence (or DAG) of *layers* l, each of which has a *tensor transformation function* $T_{:l}$. We are also given the image $I_{:img}$ for which the occlusion-based explanation (OBE) is desired, the class label L predicted by f on $I_{:img}$, an occlusion patch $\mathcal{P}$ in RGB format, and occlusion patch *stride* $S_{\mathcal{P}}$. We are also given a set of patch positions G

constructed either automatically or manually with a visual interface interactively. The OBE workload is as follows: produce a 2-D heat map M, wherein each value corresponds to a position in G and has the prediction probability of L by f on the occluded image $I^{'}_{x,y:img}$ (i.e., superimpose occlusion patch on image) or zero otherwise. More precisely, we can describe the OBE workload with the following logical statements:

$$W_M = \lfloor (\texttt{width}(I_{:img}) - \texttt{width}(\mathcal{P}) + 1)/S_{\mathcal{P}} \rfloor \tag{6.1}$$

$$H_M = \lfloor (\texttt{height}(I_{:img}) - \texttt{height}(\mathcal{P}) + 1)/S_{\mathcal{P}} \rfloor \tag{6.2}$$

$$M \in \mathbb{R}^{H_M \times W_M} \tag{6.3}$$

$$\forall\ (x,y) \in G: \tag{6.4}$$

$$I^{'}_{x,y:img} \leftarrow I_{:img}\ \circ_{(x,y)}\ \mathcal{P} \tag{6.5}$$

$$M[x,y] \leftarrow f(I^{'}_{x,y:img})[L] \tag{6.6}$$

Steps (6.1) and (6.2) calculate the dimensions of the heat map M. Step (6.5) superimposes $\mathcal{P}$ on $I_{:img}$ with its top left corner placed on the (x,y) location of $I_{:img}$. Step (6.6) calculates the output value at the (x,y) location by performing CNN inference for $I^{'}_{x,y:img}$ using f and picks the prediction probability of L. Steps (6.5) and (6.6) are performed *independently* for every occlusion patch position in G. In the *non-interactive* mode, G is initialized to $G = [0, H_M) \times [0, W_M)$. Intuitively, this represents the set of all possible occlusion patch positions on $I_{:img}$, which yields a full heat map. In the *interactive* mode, the user may manually place the occlusion patch only at a few locations at a time, yielding partial heat maps.

We assume the CNN is used for classification (or regression), since only such applications typically use OBE. One could create CNNs that predict an image "segmentation" instead, but labeling image segments for training such CNNs is tedious and expensive. Thus, most recent applications of CNNs in healthcare, sociology, and other domains rely on classification CNNs

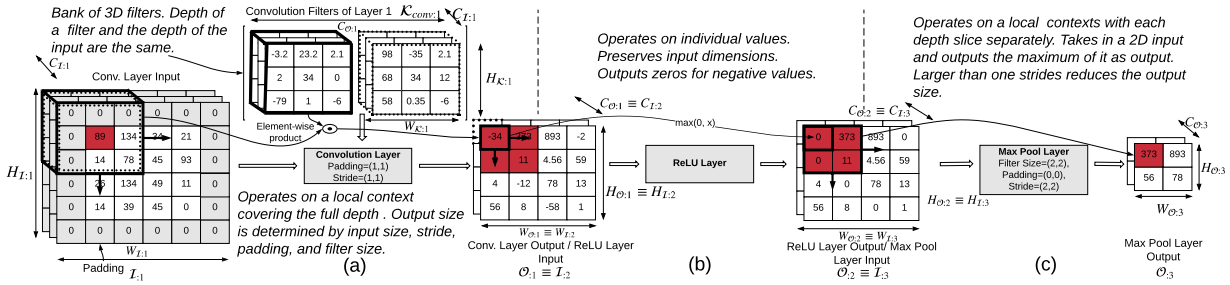


Figure 6.2: Simplified illustration of the key layers of a typical CNN. The highlighted cells (dark/red background) show how a small local spatial context in the first input propagates through subsequent layers. (a) Convolution layer (for simplicity sake, bias addition is not shown). (b) ReLU Non-linearity layer. (c) Pooling layer (max pooling). Notation is explained in Table 6.1.

and use OBE [160, 148, 204, 63, 278]. Other approaches to explain CNN predictions have been studied, but since they are orthogonal to our focus, we summarize them in Section 10.3. We assume f is from a roster of well-known deep CNNs; we currently support VGG16, ResNet18, and InceptionV3. We think this is a reasonable start, since most recent OBE applications use only such well-known CNNs from model zoos [6, 11]. But we note that our techniques are generic enough to apply to any CNN; we leave support for arbitrary CNNs to future work.

6.2.2 Deep Convolutional Neural Networks (CNNs)

CNNs are a type of neural networks specialized for images [180, 124]. CNNs typically surpass older hand-crafted image features such as SIFT and HOG in accuracy [195, 103]. CNNs are organized as *layers* of various types, each of which transforms a tensor (multidimensional array, typically 3-D) into another tensor: *Convolution* uses image filters from graphics to extract features, but with parametric filter weights (learned during training); *Pooling* subsamples features in a spatial-aware manner; *Batch-Normalization* normalizes the output tensor; *Non-Linearity* applies an element-wise non-linear function (e.g., ReLU); *Fully-Connected* is an ordered collection of perceptrons [124]. The output tensor of a layer can have a different width, height, and/or depth than the input. An image can be viewed as a tensor, e.g., a 224×224 RGB image is a

3-D tensor with width and height 224 and depth 3. A Fully-Connected layer converts a 1-D tensor (or a "flattened" 3-D tensor) to another 1-D tensor. For simplicity of exposition, we group CNN layers into 3 main categories based on the *spatial locality* of how they transform a tensor: (1) Transformations with a *global context*, e.g., Fully-Connected; (2) Transformations at the granularity of *individual elements*, e.g., ReLU or Batch Normalization; and (3) Transformations at the granularity of a *local spatial context*, e.g. Convolution or Pooling.

Global context granularity. Such layers convert the input tensor holistically into an output tensor without any spatial context, typically with a full matrix-vector multiplication. Fully-Connected is the only layer of this type. Since every element of the output will likely be affected by the entire input, such layers do not offer a major opportunity for faster incremental computations. Thankfully, Fully-Connected layers typically arise only as the last layer(s) in deep CNNs (and never in some recent deep CNNs), and they typically account for a negligible fraction of the total computational cost. Thus, we do not focus on such layers for our optimizations.

Individual element granularity. Such layers apply a "map()" function on the elements of the input tensor, as Figure 6.2(b) illustrates. Thus, the output has the same dimensions as the input. Non-Linearity (e.g., with ReLU) falls under this category. The computational cost is proportional to the "volume" of the input tensor (product of the dimensions). If the input is incrementally updated, only the corresponding region of the output will be affected. Thus, incremental inference for such layers is straightforward. The computational cost of the incremental computation is proportional to the volume of the updated region.

Local spatial context granularity. Such layers perform weighted aggregations of slices of the input tensor, called *local spatial contexts*, by multiplying them with a *filter kernel* (a tensor of weights). Thus, input and output tensors can differ in width, height, and depth. If the input is incrementally updated, the region of the output that will be affected is not straightforward to ascertain–this requires non-trivial and careful calculations due to the overlapping nature of how

filters get applied to local spatial contexts. Both Convolution and Pooling fall under this category. Since such layers typically account for the bulk of the computational cost of deep CNN inference, enabling incremental inference for such layers in the OBE context is a key focus of this Work. The rest of this section explains the machinery of the dataflow in such layers using our notation. Section 6.3 then uses this machinery to explain our optimizations.

Dataflow of Convolution Layers. A layer l has $C_{O:l}$ 3-D filter kernels arranged as a 4-D array $\mathcal{K}_{conv:l}$, with each having a smaller spatial width $W_{\mathcal{K}:l}$ and height $H_{\mathcal{K}:l}$ than the width $W_{I:l}$ and height $H_{I:l}$ of the input tensor $I_{:l}$ but the same depth $C_{I:l}$. During inference, c^{th} filter kernel is "strided" along the width and height dimensions of the input to produce a 2-D "activation map" $A_{:c} = (a_{y,x:c}) \in \mathbb{R}^{H_{O:l} \times W_{O:l}}$ by computing element-wise products between the kernel and the local spatial context and adding a bias value as per Equation (6.7). The computational cost of each of these small matrix products is proportional to the volume of the filter kernel. All the 2-D activation maps are then stacked along the depth dimension to produce the output tensor $O_{:l} \in \mathbb{R}^{C_{O:l} \times H_{O:l} \times W_{O:l}}$. Figure 6.2 (a) presents a simplified illustration of this layer.

$$\begin{aligned} a_{y,x:c} = \sum_{k=0}^{C_{I:l}} \sum_{j=0}^{H_{\mathcal{K}:l}-1} \sum_{i=0}^{W_{\mathcal{K}:l}-1} & \mathcal{K}_{conv:l}[c,k,j,i] \\ & \times I_{:l}[k, y - \lfloor \frac{H_{\mathcal{K}:l}}{2} \rfloor + j, x - \lfloor \frac{W_{\mathcal{K}:l}}{2} \rfloor + i] \\ & + \mathcal{B}_{conv:l}[c] \end{aligned} \tag{6.7}$$

Dataflow of Pooling Layers. Such layers behave essentially like Convolution layers but with a fixed (not learned) 2-D filter kernel $\mathcal{K}_{pool:l}$. These kernels aggregate a local spatial context to compute its maximum or average element. But unlike Convolution, Pooling operates independently on the depth slices of the input tensor. It takes as input a 3-D tensor O_l of depth $C_{I:l}$, width $W_{I:l}$, and height $H_{I:l}$. It produces as output a 3-D tensor $O_{:l}$ with the same depth $C_{O:l} = C_{I:l}$ but a different width of $W_{O:l}$ and height $H_{O:l}$. The filter kernel is typically strided over more

than one pixel at a time. Thus, $W_{O:l}$ and $H_{O:l}$ are usually smaller than $W_{I:l}$ and $H_{I:l}$, respectively. Figure 6.2(c) presents a simplified illustration of this layer. Overall, both Convolution and Pooling layers have a similar dataflow along the width and height dimensions, while differing on the depth dimension. Since OBE only concerns the width and height dimensions of the image and subsequent tensors, we can treat both these types of layers in a unified manner for our optimizations.

Relationship between Input and Output Dimensions. For Convolution and Pooling layers, $W_{O:l}$ and $H_{O:l}$ are determined by $W_{I:l}$ and $H_{I:l}$, $W_{\mathcal{K}:l}$ and $H_{\mathcal{K}:l}$, and two other parameters that are specific to that layer: *stride* $S_{:l}$ and *padding* $P_{:l}$. Stride is the number of pixels by which the filter kernel is moved at a time; it can differ along the width and height dimensions: $S_{x:l}$ and $S_{y:l}$, respectively. In practice, most CNNs have $S_{x:l} = S_{y:l}$. Typically, $S_{x:l} \leq W_{\mathcal{K}:l}$ and $S_{y:l} \leq H_{\mathcal{K}:l}$. In Figure 6.2, the Convolution layer has $S_{x:l} = S_{y:l} = 1$, while the Pooling layer has $S_{x:l} = S_{y:l} = 2$. For some layers, to help control the dimensions of the output to be the same as the input, one "pads" the input with zeros along the width and height dimensions. *Padding* $P_{:l}$ captures how much such padding extends these dimensions; once again, padding values can differ along the width and height dimensions: $P_{x:l}$ and $P_{y:l}$. In Figure 6.2 (a), the Convolution layer has $P_{x:l} = P_{y:l} = 1$. Given these parameters, width (similarly height) of the output tensor is given by the following formula:

$$W_{O:l} = (W_{I:l} - W_{\mathcal{K}:l} + 1 + 2 \times P_{x:l})/S_{x:l} \tag{6.8}$$

Computational Cost of Inference. Deep CNN inference is computationally expensive. Convolution layers typically account for a bulk of the cost (90% or more) [85]. Thus, we can roughly estimate the computational cost of inference by counting the number of *fused multiply-add* (FMA) floating point operations (FLOPs) needed for the Convolution layers. For example,

applying a Convolution filter with dimensions $(C_{I:l}, H_{\mathcal{K}:l}, W_{\mathcal{K}:l})$ to compute one element of the output tensor requires $C_{I:l} \cdot H_{\mathcal{K}:l} \cdot W_{\mathcal{K}:l}$ FLOPs, with each FLOP corresponding to one FMA. Thus, the total computational cost $Q_{:l}$ of a layer that produces output $O_{:l}$ of dimensions $(C_{O:l}, H_{O:l}, W_{O:l})$ and the total computational cost Q of the entire set of Convolution layers of a given CNN f can be calculated as per Equations (6.9) and (6.10).

$$Q_{:l} = (C_{I:l} \cdot H_{\mathcal{K}:l} \cdot W_{\mathcal{K}:l})(C_{O:l} \cdot H_{O:l} \cdot W_{O:l}) \tag{6.9}$$

$$Q = \sum_{l \ in \ f} Q_{:l} \tag{6.10}$$

6.3 Incremental Inference Optimizations

We start with a theoretical characterization of the speedups incremental inference can yield. We then dive into our novel algebraic framework to enable incremental CNN inference and combine it with our multi-query optimization for OBE.

6.3.1 Expected Speedups

In relational IVM, when a part of the input relation is updated, we recompute only the part of output that gets changed. We bring this notion to CNNs; a CNN layer is our "query" and the materialized feature tensor is our "relation." OBE updates only a part of the image; so, only some parts of the tensors need to be recomputed. We create an algebraic framework to determine which parts these are for a CNN layer (Section 6.3.2) and how to propagate updates across layers (Section 6.3.3). Given a CNN f and the occlusion patch, our framework determines using "static analysis" over f how many FLOPs can be saved, yielding us an upper bound on speedups.

More precisely, let the output tensor dimensions of layer l be $(C_{O:l}, H_{O:l}, W_{O:l})$. An incremental update recomputes a smaller local spatial context with width $W_{\mathcal{P}:l} \leq W_{O:l}$ and height

$H_{\mathcal{P}:l} \leq H_{O:l}$. Thus, the computational cost of incremental inference for layer l, denoted by $Q_{inc:l}$, is equal to the volume of the individual filter kernel times the total volume of the updated output, as given by Equation (6.11). The total computational cost for incremental inference, denoted Q_{inc}, is given by Equation (6.12).

$$Q_{inc:l} = (C_{I:l} \cdot H_{\mathcal{K}:l} \cdot W_{\mathcal{K}:l})(C_{O:l} \cdot H_{\mathcal{P}:l} \cdot W_{\mathcal{P}:l}) \tag{6.11}$$

$$Q_{inc} = \sum_{l \ in \ f} Q_{inc:l} \tag{6.12}$$

The above costs can be much smaller than $Q_{:l}$ and Q in Equations (6.9) and (6.10) earlier. We define the *theoretical speedup* as the ratio $\frac{Q}{Q_{inc}}$. It tells us how beneficial incremental inference can be in the best case *without* performing the inference itself. It depends on several factors: the occlusion patch size, its location, the parameters of layers (kernel dimensions, stride, etc.), and so on. Calculating it is non-trivial and requires careful analysis, which we perform. The location of patch affects this ratio because a patch placed in the corner leads to fewer updates overall than one placed in the center of the image. Thus, the "worst-case" theoretical speedup is determined by placing the patch at the center.

We perform a sanity check experiment to ascertain the theoretical speedups for a few popular deep CNNs. For varying occlusion patch sizes (with a stride of 1), we plot the theoretical speedups in Figure 6.3. VGG-16 has the highest theoretical speedups, while DenseNet-121 has the lowest. Most CNNs fall in the 2X–3X range. The differences arise due to the specifics of the CNNs' architectures: VGG-16 has small Convolution filter kernels and strides, which means full inference incurs a high computational cost ($Q = 15$ GFLOPs). Thus, VGG-16 benefits the most from incremental inference. Note the image size is assumed to be 224×224 for this plot; if the image is larger, the theoretical speedups will be higher.

While speedups of 2X-3X may sound "not that significant" in practice, we find that

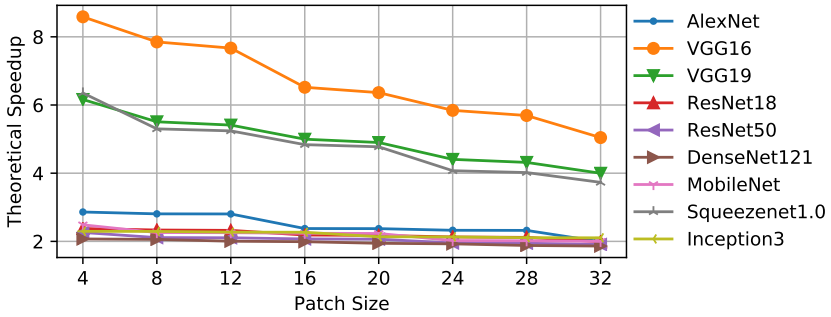


Figure 6.3: Theoretical speedups for popular deep CNN architectures with incremental inference.

they indeed are significant for at least two reasons. First, *users often wait in the loop* for OBE workloads for performing interactive diagnoses and analyses. Thus, even such speedups can improve their productivity, e.g., reducing the time taken on a CPU from about 6min to just 2min, or on a GPU from 1min to just 20s. Second, and equally importantly, incremental inference is the *foundation for our approximate inference* optimizations (Section 6.4), which amplify the speedups we achieve for OBE. For instance, the speedup for Inception3 goes up from only 2X for incremental inference to up to 8X with all of our optimizations enabled. Thus, incremental inference is critical to optimizing OBE.

6.3.2 Single Layer Incremental Inference

We now present our algebraic framework for incremental updates to the materialized output tensor of a CNN layer. As per the discussion in Section 6.2.2, we focus only on the non-trivial layers that operate at the granularity of a local spatial context (Convolution and Pooling). We call our modified version of such layers "incremental inference operations."

Determining Patch Update Locations. We first explain how to calculate the coordinates and dimensions of the *output update patch* of layer l given the *input update patch* and layer-specific parameters. Figure 6.4 presents a simplified illustration of these calculations. Our coordinate system's origin is at the top left corner. The input update patch is shown in red/dark

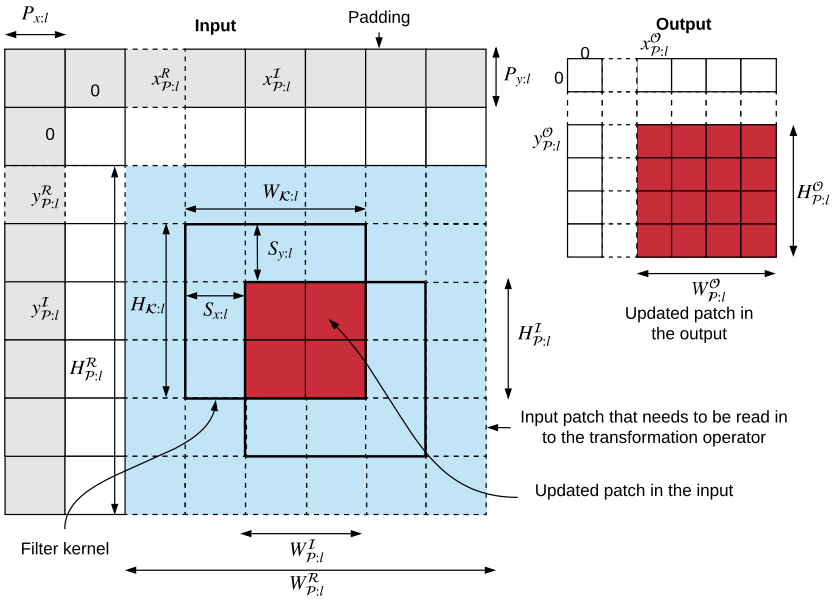


Figure 6.4: Simplified illustration of input and output update patches for Convolution/Pooling layers.

color and starts at $(x^I_{\mathcal{P}:l}, y^I_{\mathcal{P}:l})$, with height $H^I_{\mathcal{P}:l}$ and width $W^I_{\mathcal{P}:l}$. The output update patch starts at $(x^O_{\mathcal{P}:l}, y^O_{\mathcal{P}:l})$ and has a height $H^O_{\mathcal{P}:l}$ and width $W^O_{\mathcal{P}:l}$. Due to overlaps among filter kernel positions during inference, computing the output update patch requires us to read a slightly larger spatial context than the input update patch–we call this the "read-in context," and it is illustrated by the blue/shaded region in Figure 6.4. The read-in context starts at $(x^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l})$, with its dimensions denoted by $W^{\mathcal{R}}_{\mathcal{P}:l}$ and $H^{\mathcal{R}}_{\mathcal{P}:l}$. Table 6.2 summarizes all this additional notation for this section. The relationship between these quantities along the width dimension (similarly along the height dimension) can be expressed as follows:

$$x^O_{\mathcal{P}:l} = max\big(\lceil (P_{x:l} + x^I_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1)/S_{x:l} \rceil, 0\big) \tag{6.13}$$

$$W^O_{\mathcal{P}:l} = min\big(\lceil (W^I_{\mathcal{P}:l} + W_{\mathcal{K}:l} - 1)/S_{x:l} \rceil, W_{O:l}\big) \tag{6.14}$$

$$x^{\mathcal{R}}_{\mathcal{P}:l} = x^O_{\mathcal{P}:l} \times S_{x:l} - P_{x:l} \tag{6.15}$$

$$W^{\mathcal{R}}_{\mathcal{P}:l} = W_{\mathcal{K}:l} + (W^O_{\mathcal{P}:l} - 1) \times S_{x:l} \tag{6.16}$$

Table 6.2: Additional notation for Sections 6.3 and 6.4.

Symbol	Meaning
$x^I_{\mathcal{P}:l}, y^I_{\mathcal{P}:l}$	Start coordinates of input update patch for layer l
$x^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l}$	Start coordinates of read-in context for layer l
$x^O_{\mathcal{P}:l}, y^O_{\mathcal{P}:l}$	Start coordinates of output update patch for layer l
$H^I_{\mathcal{P}:l}, W^I_{\mathcal{P}:l}$	Height and width of input update patch for layer l
$H^{\mathcal{R}}_{\mathcal{P}:l}, W^{\mathcal{R}}_{\mathcal{P}:l}$	Height and width of read-in context for layer l
$H^O_{\mathcal{P}:l}, W^O_{\mathcal{P}:l}$	Height and width of output update patch for layer l
τ	Projective field threshold
$r_{drill-down}$	Drill-down fraction for adaptive drill-down

Equation (6.13) calculates the coordinates of the output update patch. As shown in Figure 6.4, padding effectively shifts the coordinate system and thus, $P_{x:l}$ is added to correct it. Due to overlaps among the filter kernels, the affected region of the input update patch (blue/shaded region in Figure 6.4) will be increased by $W_{\mathcal{K}:l} - 1$, which needs to be subtracted from the input coordinate $x^I_{\mathcal{P}:l}$. A filter of size $W_{\mathcal{K}:l}$ that is placed starting at $x^I_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1$ will see an update starting from $x^I_{\mathcal{P}:l}$. Equation (6.14) calculates the width of the output update patch which is essentially the number of filter kernel stride positions on the read-in input context. However, this value cannot be larger than the output size. Given these, a start coordinate and width of the read-in context are given by Equations (6.15) and (6.16); similar equations hold for the height dimension (skipped for brevity).

Incremental Inference Operation. For layer l, given the transformation function $T_{:l}$, the pre-materialized input tensor $I_{:l}$, input update patch $\mathcal{P}^O_{:l}$, and the above calculated coordinates and dimensions of the input, output, and read-in context, the output update patch $\mathcal{P}^O_{:l}$ is computed as follows:

$$\mathcal{U} = I_{:l}[:, x^{\mathcal{R}}_{\mathcal{P}:l} : x^{\mathcal{R}}_{\mathcal{P}:l} + W^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l} : y^{\mathcal{R}}_{\mathcal{P}:l} + H^{\mathcal{R}}_{\mathcal{P}:l}] \tag{6.17}$$

$$\mathcal{U} = \mathcal{U} \circ_{(x^{I}_{\mathcal{P}:l} - x^{\mathcal{R}}_{\mathcal{P}:l}), (y^{I}_{\mathcal{P}:l} - y^{\mathcal{R}}_{\mathcal{P}:l})} \mathcal{P}^{I}_{:l} \tag{6.18}$$

$$\mathcal{P}^{O}_{:l} = T_{:l}(\mathcal{U}) \tag{6.19}$$

Equation (6.17) slices the read-in context $\mathcal{U}$ from the pre-materialized input tensor $I_{:l}$. Equation (6.18) superimposes the input update patch $\mathcal{P}^{I}_{:l}$ on it. This is an in-place update of the array holding the read-in context. Finally, Equation (6.19) computes the output update patch $\mathcal{P}^{O}_{:l}$ by invoking $T_{:l}$ on $\mathcal{U}$. Thus, we avoid performing inference on all of $I_{:l}$, thus achieving incremental inference and reducing FLOPs.

6.3.3 Propagating Updates across Layers

Sequential CNNs. Unlike relational IVM, CNNs have many layers, often in a sequence. This is analogous to a sequence of queries, each requiring IVM on its predecessor's output. This leads to a new issue: correctly and automatically configuring the update patches across all layers of a CNN. Specifically, output update patch $\mathcal{P}^{O}_{:l}$ of layer l becomes the input update patch of layer $l+1$. While this seems simple, it requires care at the boundary of a local context transformation and a global context transformation, e.g., between a Convolution (or Pooling) layer and a Fully-Connected layer. In particular, we need to materialize the full updated output, not just the output update patches, since global context transformations lose spatial locality for subsequent layers.

Extension to DAG CNNs. Some recent deep CNNs have a more general directed acyclic graph (DAG) structure for layers. They have two new kinds of layers that "merge" two branches in the DAG: *element-wise addition* and *depth-wise concatenation*. Element-wise addition requires

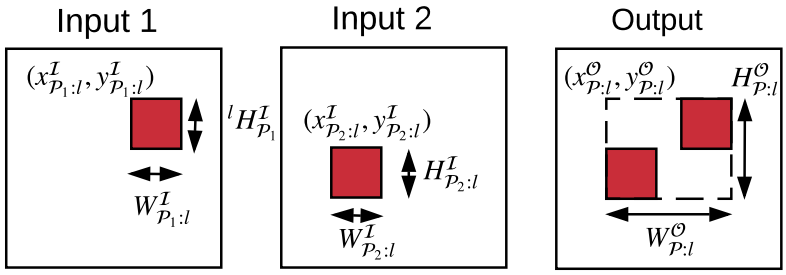


Figure 6.5: Illustration of bounding box calculation for differing input update patch locations for element-wise addition and depth-wise concatenation layers in DAG CNNs.

two input tensors with all dimensions being identical. Depth-wise concatenation takes two input tensors with the same height and width dimensions. We now face a new challenge–how to calculate the output update patch when the two input tensors differ on their input update patches locations and sizes? Figure 6.5 shows a simplified illustration of this issue. The first input has its update patch starting at coordinates $(x^I_{\mathcal{P}_1:l}, y^I_{\mathcal{P}_1:l})$ with dimensions $H^I_{\mathcal{P}_1:l}$ and $W^I_{\mathcal{P}_1:l}$, while the second input has its update patch starting at coordinates $(x^I_{\mathcal{P}_2:l}, y^I_{\mathcal{P}_2:l})$ with dimensions $H^I_{\mathcal{P}_2:l}$ and $W^I_{\mathcal{P}_2:l}$. This issue can arise with both element-wise addition and depth-wise concatenation.

We propose a simple unified solution: compute the *bounding box* of the input update patches. So, the coordinates and dimensions of both read-in contexts and the output update patch will be identical. Figure 6.5 illustrates this. While this will potentially recompute parts of the output that do not get modified, we think this trade-off is acceptable because the gains are likely to be marginal for the additional complexity introduced into our framework. Overall, the output update patch coordinate and width dimension are given by the following (similarly for the height dimension):

$$\begin{aligned} x^O_{P:l} &= \texttt{min}(x^I_{\mathcal{P}_1:l}, x^I_{\mathcal{P}_2:l}) \\ W^O_{\mathcal{P}:l} &= \texttt{max}(x^I_{\mathcal{P}_1:l} + W^I_{\mathcal{P}_1:l}, x^I_{\mathcal{P}_2:l} + W^I_{\mathcal{P}_2:l}) - \texttt{min}(x^I_{\mathcal{P}_1:l}, x^I_{\mathcal{P}_2:l}) \end{aligned} \tag{6.20}$$

6.3.4 Multi-Query Incremental Inference

OBE issues $|G|$ re-inference requests *in one go*. Viewing each request as a "query" makes the connection with multi-query optimization (MQO) [250] clear. The $|G|$ queries are also *not disjoint*, since the occlusion patch is typically small, which means most pixels are the same for each query. Thus, we now extend our IVM framework for re-inference with an MQO-style optimization fusing multiple re-inference requests. An analogy with relational queries would be having many incremental update queries on the same relation in one go, with each query receiving a different incremental update.

Batched Incremental Inference. Our optimization works as follows: materialize all CNN tensors *once* and *reuse* them for incremental inference across all $|G|$ queries. Since the occluded images share most of their pixels, parts of the tensors will likely be identical too. Thus, we can amortize the materialization cost. One might ask: why not just perform "batched" inference for the $|G|$ queries? Batched execution is standard practice on high-throughput compute hardware like GPUs, since it amortizes CNN set up costs, data movement costs, etc. Batch sizes are tuned to optimize hardware utilization. We note that batching is an *orthogonal* (albeit trivial) optimization compared to our MQO. Thus, we combine both of these ideas to execute incremental inference in a batched manner. We call this approach "batched incremental inference." Empirically, we find that batching alone yields limited speedups (under 2X), but our batched incremental inference amplifies the speedups. Algorithm 6 formally presents the batched incremental inference operation for layer l.

BATCHEDINCREMENTALINFERENCE first calculates the geometric properties of the output update patches and read-in contexts. A temporary tensor $\mathcal{U}$ is initialized to hold the input update patches with their read-in contexts. The **for** loop iteratively populates $\mathcal{U}$ with corresponding patches. Finally, $T_{:l}$ is applied to $\mathcal{U}$ to compute the output patches. We note that for the first layer, all input update patches will be identical to the occlusion patch. But for the later

Algorithm 6 BATCHEDINCREMENTALINFERENCE

Inputs:
$T_{:l}$: *Original Transformation function*
$I_{:l}$: *Pre-materialized input from original image*
$[\mathcal{P}^I{}_{1:l}, ..., \mathcal{P}^I{}_{n:l}]$: *Input patches*
$[(x^I_{\mathcal{P}_1:l}, y^I_{\mathcal{P}_1:l}), ..., (x^I_{\mathcal{P}_n:l}, y^I_{\mathcal{P}_n:l})]$: *Input patch coordinates*
$W^I_{\mathcal{P}:l}, H^I_{\mathcal{P}:l}$: *Input patch dimensions*

1: *Calculate* $[(x^O_{\mathcal{P}_1:l}, y^O_{\mathcal{P}_1:l}), ..., (x^O_{\mathcal{P}_n:l}, y^O_{\mathcal{P}_n:l})]$
2: *Calculate* $(W^O_{\mathcal{P}:1}, H^O_{\mathcal{P}:l})$
3: *Calculate* $[(x^{\mathcal{R}}_{\mathcal{P}_1:l}, y^{\mathcal{R}}_{\mathcal{P}_1:l}), ..., (x^{\mathcal{R}}_{\mathcal{P}_n:l}, y^{\mathcal{R}}_{\mathcal{P}_n} : l)]$
4: *Calculate* $(W^{\mathcal{R}}_{\mathcal{P}:l}, H^{\mathcal{R}}_{\mathcal{P}:l})$
5: *Initialize* $\mathcal{U} \in \mathbb{R}^{n \times \texttt{depth}(I_{:l}) \times H^{\mathcal{R}}_{\mathcal{P}:l} \times W^{\mathcal{R}}_{\mathcal{P}:l}}$
6: **for** `i in [1,...,n]` **do**
7: $\quad T_1 \leftarrow I_{:l}[:, x^{\mathcal{R}}_{\mathcal{P}_i:l} : x^{\mathcal{R}}_{\mathcal{P}_i:l} + W^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}_i:l} : y^{\mathcal{R}}_{\mathcal{P}_i:l} + H^{\mathcal{R}}_{\mathcal{P}:l}]$
8: $\quad T_2 \leftarrow T_1 \circ_{(x^I_{\mathcal{P}_i:l} - x^{\mathcal{R}}_{\mathcal{P}_i:l}), (y^I_{\mathcal{P}_i:l} - y^{\mathcal{R}}_{\mathcal{P}_i:l})} \mathcal{P}_{i:l}$
9: $\quad \mathcal{U}[i, :, :] \leftarrow T_2$
10: $[\mathcal{P}^O_{1:l}, ..., \mathcal{P}^O_{n:l}] \leftarrow T(\mathcal{U})$
11: **return** $[\mathcal{P}^O_{1:l}, ..., \mathcal{P}^O_{n:l}]$, $[(x^O_{\mathcal{P}_1:l}, y^O_{\mathcal{P}_1:l}), ..., (x^O_{\mathcal{P}_n:l}, y^O_{\mathcal{P}_n:l})]$, $(W^O_{\mathcal{P}:l}, H^O_{\mathcal{P}:l})$

layers, the update patches will start to deviate depending on their locations and read-in contexts.

GPU Optimized Implementation. Empirically, we found a dichotomy between CPUs and GPUs: BATCHEDINCREMENTALINFERENCE yielded expected speedups on CPUs, but it performed dramatically poorly on GPUs. In fact, a naive implementation of BATCHEDINCREMENTALINFERENCE on GPUs was *slower* than full re-inference! We now shed light on why this is the case and how we tackled this issue. The **for** loop in line 6 of Algorithm 6 is essentially preparing the input for $T_{:l}$ by copying values (slices of the materialized tensor) from one part of GPU memory to another sequentially. A detailed profiling of the GPU showed that these *sequential memory copies are a bottleneck* for GPU throughput, since they throttle it from exploiting its massive parallelism effectively. To overcome this issue, we created a custom CUDA kernel to perform input preparation more efficiently by *copying memory regions in parallel* for all items in the batched inference request. This is akin to a parallel **for** loop tailored for slicing the tensor. We

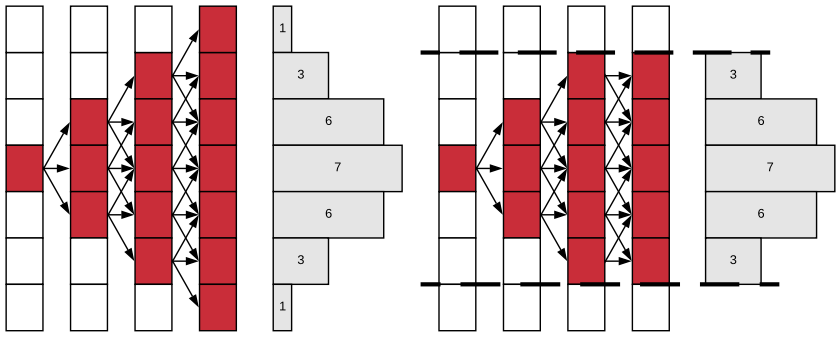
then invoke $T_{:l}$, which is already hardware-optimized by modern deep learning tools [95]. We defer more details on our custom CUDA kernel to the Appendix B.2. Also, since GPU memory might not be enough to fit all $|G|$ queries, the batch size for GPU execution might be smaller than $|G|$.

6.3.5 Putting it All Together

We summarize the end-to-end workflow of our incremental inference optimizations for OBE. We are given the CNN f, image $I_{:img}$, predicted class label L, occlusion patch $\mathcal{P}$ and its stride $S_{\mathcal{P}}$, and the set of occlusion patch positions G. Pre-materialize the output tensors of all layers of f with $I_{:img}$ as the input. Prepare occluded images ($I'_{(x,y):img}$) for all positions in G. For batches of $I'_{(x,y):img}$ as the input, invoke the transformations functions of the layers of f in topological order and calculate the corresponding entries of heat map M. For transformations with local spatial context, invoke BATCHEDINCREMENTALINFERENCE. For layer that precede a global context transformation, materialize the full updated output. For all other layers, invoke the original transformation function. M is now the output heat map.

6.4 Approximate Inference Optimizations

Since incremental inference is *exact*, i.e., it yields the same heat map as full inference, it does not exploit a capability of human perception: tolerance of some degradation in visual quality. Thus, we now build upon our IVM framework to create two novel heuristic approximate inference optimizations that trade off the heat map's quality in a user-tunable manner to accelerate OBE further. We note that our optimizations operate at the logical level and are complementary to more physical-level optimizations such as low-precision computation [206] and model pruning [138]. We first present the techniques and then explain how to tune them.



(a) Projective Field (b) Projective field thresholding

Figure 6.6: (a) Projective field growth for 1-D Convolution (filter size 2, stride 1). (b) Projective field *thresholding*; $\tau = 5/7$.

6.4.1 Projective Field Thresholding

The *projective field* of a CNN neuron is the slice of the output tensor that is connected to it [3]. It is a term from neuroscience to describe the effects of a retinal cell on the output of the eye's neuronal circuitry [107]. This notion sheds light on the *growth of the size* of the update patches through the layers of a CNN. The 3 kinds of layers (Section 6.2.2) affect the projective field size growth differently. Transformations at the granularity of individual elements do not alter the projective field size. Global context transformations increase it to the whole output. But local spatial context transformations, which are the most crucial, increase it *gradually* at a rate determined by the filter kernel's size and stride: additively in the size and multiplicatively in the stride. The growth of the projective field size implies the amount of FLOPs saved by IVM decreases as we go to the higher layers of a CNN. Eventually, the output update patch becomes as large as the output tensor. This growth is illustrated by Figure 6.6(a).

Our above observation motivates the main idea of this optimization, which we call projective field thresholding: *truncate* the projective field from growing beyond a given *threshold fraction* τ $(0 < \tau \leq 1)$ of the output size. This means inference in subsequent layers is approximate. Figure 6.6(b) illustrates the idea for a filter size 3 and stride 1. One input element is updated (shown in red/dark); the change propagates to 3 elements in the next layer and then 5, but it then

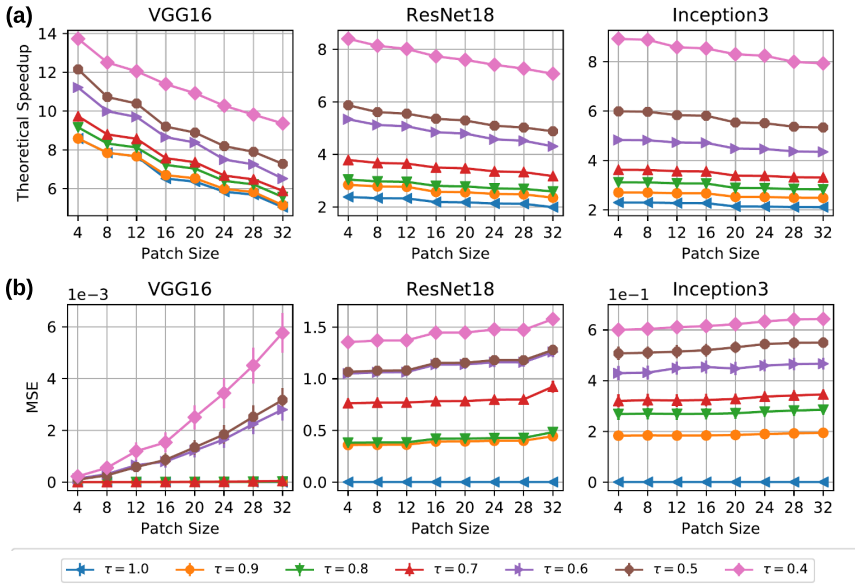


Figure 6.7: (a) Theoretical speedups with projective field thresholding. (b) Mean Square Error between exact and approximate output of final Convolution/Pooling layers.

gets truncated because we set $\tau = 5/7$. This approximation can alter the accuracy of the output values and the heat map's visual quality. Empirically, we find that modest truncation is tolerable and does not affect the heat map's visual quality too significantly.

To provide intuition on why the above happens, consider histograms on the side of Figures 6.6(a,b) that list the number of unique "paths" from the updated element to each element in the last layer. It resembles a Gaussian distribution, with the maximum paths concentrated on the middle element. Thus, for most of the output patch updates, truncation will only discard a few values at the "fringes" that contribute to an output element. Of course, we do not consider the weights on these "paths," which is dependent on the given trained CNN. Since the weights can be arbitrary, a tight formal analysis is unwieldy. But under some assumptions on the weights values (similar to the assumptions in [197] for understanding the "receptive field" in CNNs), in Appendix B.4 we show that this distribution does indeed converge to a Gaussian. Thus, while this idea is a heuristic, it is grounded in a common behavior of real CNNs. Overall, since most of the contributions to the output elements are concentrated around the center, such truncation is often affordable. Note that this optimization is only feasible *in conjunction with* our incremental

inference framework (Section 6.3) to reuse the remaining parts of the tensors and save FLOPs. We extend the formulas for the output-input coordinate calculations to account for τ. For the width dimension, the new formulas are as follows (similarly for the height dimension):

$$W^O_{\mathcal{P}:l} = \texttt{min}\big(\lceil (W^I_{\mathcal{P}:l} + W_{\mathcal{K}:l} - 1)/S_{x:l} \rceil, W^O_{\mathcal{P}:l}\big) \tag{6.21}$$

$$\text{If } W^O_{\mathcal{P}:l} > \texttt{round}(\tau \times W^O_{:l}) : \tag{6.22}$$

$$W^O_{\mathcal{P}:l} = \texttt{round}(\tau \times W^O_{:l}) \tag{6.23}$$

$$W^I_{\mathcal{P}_{new}:l} = W^O_{\mathcal{P}:l} \times S_{x:l} - W_{\mathcal{K}:l} + 1 \tag{6.24}$$

$$x^I_{\mathcal{P}:l} \mathrel{+}= (W^I_{\mathcal{P}:l} - W^I_{\mathcal{P}_{new}:l})/2 \tag{6.25}$$

$$W^I_{\mathcal{P}:l} = W^I_{\mathcal{P}_{new}:l} \tag{6.26}$$

$$x^O_{\mathcal{P}:l} = \texttt{max}\big(\lceil (P_{x:l} + x^I_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1)/S_{x:l} \rceil, 0\big) \tag{6.27}$$

Equation (6.21) calculates the width assuming no thresholding. But if the output width exceeds the threshold, it is reduced as per Equation (6.23). Equation (6.24) calculates the input width that would produce an output of width $W^O_{\mathcal{P}:l}$; we can think of this as making $W^I_{\mathcal{P}:l}$ the subject of Equation (6.21). If the new input width is smaller than the original input width, the starting x coordinate should be updated as per Equation (6.25) s.t. the new coordinates correspond to a "center crop" compared to the original. Equation (6.26) sets the input width to the newly calculated input width. Equation (6.27) calculates the x coordinate of the output update patch.

Theoretical Speedups. We modify our "static analysis" framework to determine the theoretical speedup of incremental inference (Section 6.3) to also include this optimization using the above formulas. Consider a square occlusion patch placed on the center of the input image. Figure 6.7 (a) plots the new theoretical speedups for varying patch sizes for 3 popular CNNs for different τ values. As expected, as τ goes down from 1, the theoretical speedup goes up for

all CNNs. Since lowering τ approximates the heat map values, we also plot the mean square error (MSE) of the elements of the exact and approximate output tensors produced by the final Convolution or Pooling layers on a sample (n=30) of real-world images. Figure 6.7 (b) shows the results. As expected, as τ drops, MSE increases. But interestingly, the trends differ across the CNNs due to their different architectural properties. MSE is especially low for VGG-16, since its projective field growth is rather slow relative to the other CNNs. We acknowledge that using MSE as a visual quality metric and tuning τ are both unintuitive for humans. We mitigate these issues in Section 6.4.3 by using a more intuitive quality metric and by presenting an automated tuning method for τ.

6.4.2 Adaptive Drill-Down

This heuristic optimization is based on our observation about a peculiar semantics of OBE that lets us modify how G (the set of occlusion patch locations) is specified and handled, especially in the non-interactive specification mode. We explain our intuition with an example. Consider a radiologist explaining a CNN prediction for diabetic retinopathy on a tissue image. The region of interest typically occupies only a tiny fraction of the image. Thus, it is an overkill to perform regular OBE for *every* patch location: most of the (incremental) inference computations are effectively "wasted" on uninteresting regions. In such cases, we modify the OBE workflow to produce an approximate heat map using a two-stage process, illustrated by Figure 6.8(a).

In stage one, we produce a lower resolution heat map by using a larger stride–we call it *stage one stride* S_1. Using this heat map, we identify the regions of the input that see the largest drops in predicted probability of the label L. Given a predefined parameter *drill-down fraction*, denoted $r_{drill-down}$, we select a proportional number of regions based on the probability drops. In stage two, we perform OBE only for these regions with original stride value (we also call this *stage two stride*, S_2) for the occlusion patch to yield a portion of the heat map at the original higher resolution. Since this process "drills down" adaptively based on the lower resolution heat

map, we call it adaptive drill-down. Note that this optimization also builds upon the incremental inference optimizations of Section 6.3, but it is *orthogonal* to projective field thresholding and can be used in addition.

Theoretical Speedups. We now define a notion of theoretical speedup for this optimization; this is independent of the theoretical speedup of incremental inference. We first explain the effects of $r_{drill-down}$ and S_1. Setting these parameters is an application-specific balancing act. If $r_{drill-down}$ is low, only a small region will need re-inference at the original resolution, which will save a lot of FLOPs. But this may miss some regions of interest and thus, compromise important explanation details. Similarly, a large S_1 also saves a lot of FLOPs by reducing the number of re-inference queries in stage one. But it runs the risk of misidentifying interesting regions, especially when the size of those regions are smaller than the occlusion patch size. We now define the theoretical speedup of adaptive drill-down as the ratio of the number of re-inference queries for regular OBE without this optimization to that with this optimization. We only need the counts, since the occlusion patch dimensions are unaltered, i.e., the cost of a re-inference query is the same with or without this optimization. Given a stride S, the number of re-inference queries is $\frac{H_{I_{img}}}{S} \cdot \frac{W_{I_{img}}}{S}$. Thus, the theoretical speedup is given by the following equation. Figure 6.8(b) illustrates how this ratio varies with S_1 and $r_{drill-down}$.

$$\texttt{speedup} = \frac{S_1^2}{S_2^2 + r_{drill-down} \cdot S_1^2} \tag{6.28}$$

6.4.3 Automated Parameter Tuning

We now present automated parameter tuning methods for easily configuring our approximate inference optimizations.

Tuning Projective Field Thresholding. As Section 6.4.1 explained, τ controls the visual

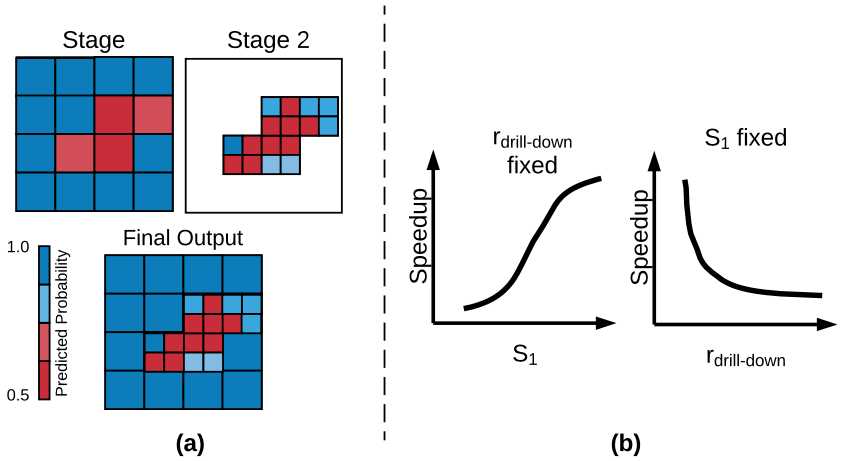


Figure 6.8: (a) Schematic illustration of the adaptive drill-down idea. (b) Conceptual depiction of the effects of S_1 and $r_{drill-down}$ on the theoretical speedup..

quality of the heat map. There is a spectrum of visual quality degradation: imperceptible changes to major structural changes. But mapping τ to visual quality directly is likely to be unintuitive for users. Thus, to measure visual quality more intuitively, we adopt a cognitive science-inspired metric called Structural Similarity (SSIM) Index, which is widely used to quantify human-perceptible differences between two images [279]. In our case, the two "images" are the original and approximate heat maps. SSIM is a number in $[-1,1]$, with 1 meaning a perfect match. SSIM values in the $[0.90,0.95]$ range are considered almost imperceptible distortions in many practical multimedia applications such as image compression and video encoding [279].

Our tuning process for τ has an offline "training" phase and an online usage phase. The offline phase relies on a set of sample images (default 30) from the same application domain. We compute SSIM for the approximate and exact heat maps for all sample images for a few τ values (default $1.0, 0.9, 0.8, \ldots, 0.4$). We then learn a second-degree polynomial curve for SSIM as a function of τ with these data points. Figure 6.9(a) illustrates this phase and the fit SSIM-τ curves for 3 different CNNs using sample images from an OCT dataset (Section 6.5). In the online phase, when OBE is needed on a given image, we expect the user to provide a *target SSIM* for the quality–runtime trade-off they want (1 yields the exact heat map). We can then use our learned curve to map this target SSIM to the lowest τ. Figure 6.9(b) shows the CDFs of

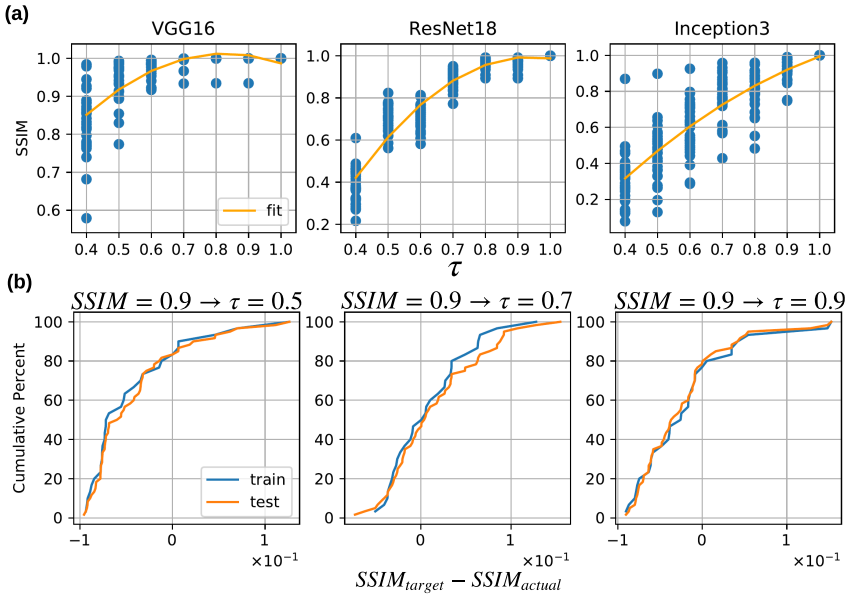


Figure 6.9: (a) Fitting a second-order curve for SSM against τ on a sample of the OCT dataset. (b) CDFs of deviation of actual SSIM from the target SSIM (0.9) with our auto-tuned τ, which turned out to be 0.5, 0.7, and 0.9 for VGG-16, ResNet-18, and Inception-V3, respectively.

differences between the target SSIM (0.9) and the actual SSIM yielded when using our auto-tuned τ on both the training set and a holdout test set (also 30 images). In 80% of the cases, the actual SSIM was *better* than the user-given target; never once did the actual SSIM go 0.1 below the target SSIM. This suggests that our auto-tuning method for τ works, is robust, and applicable to different CNNs.

Tuning Adaptive Drill-Down. As Section 6.4.2 explained, the speedup offered by adaptive drill-down is controlled by two parameters: stage one stride S_1 and drill-down fraction $r_{drill-down}$. We expect the user to provide $r_{drill-down}$ (default 0.25), since it captures the user's intuition about how large or small the region of interest is likely to be in the images in their specific application domain and dataset. We also expect the user to provide a "target speedup" ratio (default 3) for using this optimization to capture their desired quality-runtime trade-off. Higher the user's target speedup, the more we sacrifice the quality of the "non-interesting regions" ($1 - r_{drill-down}$ fraction of the heat map). Our automated tuning process sets S_1 using these two user-given settings. Unlike the tuning of τ, setting S_1 is more direct, since this optimization

relies on the number of re-inference queries, not SSIM. Let *target* denote the target speedup; the original occlusion patch stride is S_2. Equation 6.29 shows how we calculate S_1; it is obtained by making $S1$ the subject of Equation 6.28. Since S_1 cannot be larger than the image width W_{img} (similarly H_{img}) and due to the constraint of $(1 - r_{drill-down} \cdot \texttt{speedup})$ being positive, we also have an upper bound on the possible speedups as per Equation 6.30.

$$S_1 = \sqrt{\frac{target}{1 - r_{drill-down} \cdot target}} \cdot S_2 \tag{6.29}$$

$$speedup < \texttt{min}\left(\frac{W_{img}^2}{S_2^2 + r_{drill-down} \cdot W_{img}^2}, \frac{1}{r_{drill-down}}\right) \tag{6.30}$$

6.5 Experimental Evaluation

We integrated our optimization techniques with the popular deep learning framework PyTorch to create a tool we call KRYPTON. Implementation details of this integration are deferred to Appendix B.2. We now evaluate the speedups yielded by KRYPTON for OBE for different deep CNNs and datasets. We then drill into the contributions of each of our optimization techniques.

Datasets. We use 3 diverse real-world image datasets: *OCT*, *Chest X-Ray*, and *ImageNet*. *OCT* has about 84,000 optical coherence tomography retinal images with 4 classes: CNV, DME, DRUSEN, and NORMAL; CNV (choroidal neovascularization), DME (diabetic macular edema), and DRUSEN are varieties of diabetic retinopathy. *Chest X-Ray* has about 6,000 X-ray images with three classes: VIRAL, BACTERIAL, and NORMAL; VIRAL and BACTERIAL are varieties of pneumonia. Both *OCT* and *Chest X-Ray* are from a recent radiology study that applied deep CNNs to detect the respective diseases [160]. *ImageNet* is a benchmark dataset in computer

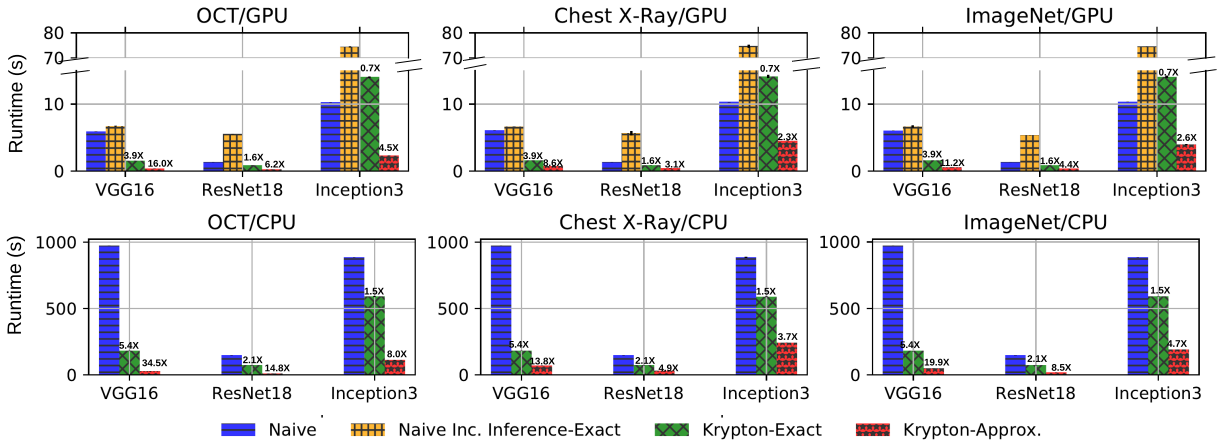


Figure 6.10: End-to-end runtimes of KRYPTON and baselines on all 3 datasets, 3 CNNs, and both GPU and CPU.

vision [246]; we use a sample of 1,000 images with 200 classes.

Workloads. We use 3 diverse ImageNet-trained deep CNNs: VGG16 [260], ResNet18 [136], and Inception3 [267], obtained from [23]. They complement each other in terms of model size, architectural complexity, computational cost, and our predicted theoretical speedups (Figure 6.3). For *OCT* and *Chest X-Ray*, the 3 CNNs were fine-tuned by retraining their final Fully-Connected layers as per standard practice. The details of fine-tuning are not relevant for the rest of our discussion; so, we present further details in the Appendix B.5. The OBE heatmaps are plotted using Python Matplotlib's `imshow` method using the `jet_r` color scheme; we set the maximum threshold to $\texttt{min}(1, 1.25p)$ and minimum to $0.75p$, where p is predicted class probability on a given image. All images are resized to the input size required by the CNNs (224×224 for VGG16 and ResNet18; 299×299 for Inception3); no additional pre-processing was done. The GPU-based experiments used a batch size of 128; for CPUs, the batch size was 16. All CPU-based experiments were executed with a thread parallelism of 8. All of our datasets, experimental scripts, and the KRYPTON codebase will be made publicly available on our project webpage.

Experimental Setup. We use a machine with 32 GB RAM, Intel i7-6700 3.40GHz CPU, and NVIDIA Titan X (Pascal) GPU with 12 GB memory. The machine runs Ubuntu 16.04 with

PyTorch version 0.4.0, CUDA version 9.0, and cuDNN version 7.1.2. All reported runtimes are the average of 3 runs, with 95% confidence intervals shown.

6.5.1 End-to- End Runtimes

We focus on the most common OBE scenario of producing the whole heatmap; G is automatically created ("non-interactive" mode). We use an occlusion patch of size 16 and stride of 4. We compare two variants of KRYPTON: KRYPTON-Exact uses only incremental inference (Section 6.3), while KRYPTON-Approximate uses our approximate inference optimizations too (Section 6.4). The main baseline is *Naive*, the current dominant practice of performing full inference for OBE with just only batching. We have another baseline on GPU: *Naive Inc. Inference-Exact*, which is a direct implementation of Algorithm 6 in PyTorch/Python without using our GPU-optimized CUDA kernel (Section 6.3.4). Note that *Naive Inc. Inference-Exact* is not relevant on CPU.

We set the adaptive drill-down parameters based on the semantics of each dataset's prediction task (Section 6.4.3). For *OCT*, since the region of interest is likely to be small, we set $r_{drill-down} = 0.1$ and $target = 5$. For *Chest X-Ray*, the region of interest can be large; so, we set $r_{drill-down} = 0.4$ and $target = 2$. For *ImageNet*, which is in between, we use the KRYPTON default of $r_{drill-down} = 0.25$ and $thittarget = 3$. Throughout, τ is auto-tuned with a target SSIM of 0.9 (Section 6.4.3). Figure 6.10 presents the results. Visual examples of the heatmaps produced are provided in the Appendix B.7.

Overall, we see KRYPTON offers significant speedups across the board on both GPU and CPU, with the highest speedups seen by KRYPTON-Approximate on *OCT* with VGG16: 16X on GPU and 34.5X on CPU. The highest speedups of KRYPTON-Exact are also on VGG16: 3.9X on GPU and 5.4X on CPU. The speedups of KRYPTON-Exact are identical across datasets for a given CNN, since it does not depend on the image semantics, unlike KRYPTON-Approximate due to its parameters. KRYPTON-Approximate sees the highest speedups on *OCT* because our

auto-tuning yielded the lowest $r_{drill-down}$, highest target speedup, and lowest τ on that dataset.

The speedups are lower with ResNet18 and Inception3 than VGG16 due to their architectural properties (kernel filter dimensions, stride, etc.) that make the projective field grow faster. Moreover, Inception3 has a complex DAG architecture with more branches and depth-wise concatenation, which limits GPU throughput for incremental inference. In fact, KRYPTON-Exact on GPU shows a minor slow-down (0.7X) with Inception3. But KRYPTON-Approximate still offers speedups on GPU with Inception3 (up to 4.5X). We also see that ResNet18 and VGG16 almost near their theoretical speedups (Figure 6.3) but Inception3 does not. Note that the theoretical speedup definition only counts FLOPs and does not account for memory stalls.

Finally, the speedups are higher on CPU than GPU; this is because CPU suffers less from memory stalls during incremental inferences. But the *absolute* runtimes are much lower on GPU as expected. Overall, KRYPTON reduces OBE runtimes substantially for multiple datasets and deep CNNs. We also ran an experiment in the "interactive" mode by reducing $|G|$. As expected, speedups go down with $|G|$ due to the reduction in amortization benefits. These additional results are presented in the Appendix B.1.

6.5.2 Ablation Study

We now analyze the contributions of our 3 optimizations individually. We compare the speedups of KRYPTON over *Naive* (batched inference) on both CPU and GPU, termed Empirical-CPU and Empirical-GPU respectively, against the theoretical speedups (explained in Sections 6.3 and 6.4).

Only Incremental Inference. We vary the patch size and set the stride to 4. Figure 6.11 shows the results. As expected, the speedups go down as the patch size increases. Empirical-GPU Naive yields no speedups because it does not use our GPU-optimized kernel, while Empirical-GPU does. But Empirical-CPU is closer to theoretical speedup and almost matches it on ResNet18. Thus, there is still some room for improvement to improve the efficiency of incremental inference

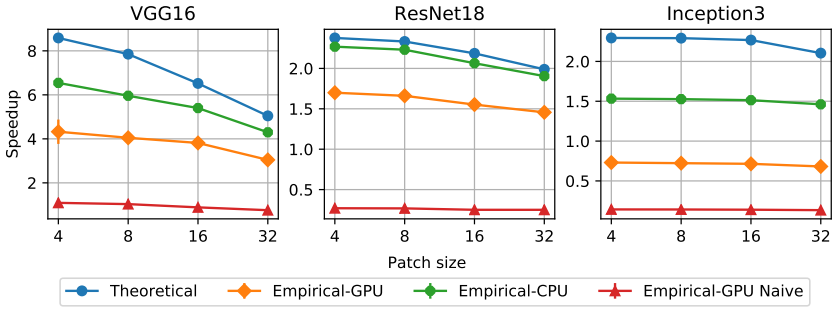


Figure 6.11: Speedups with only the incremental inference optimization (occlusion patch stride $S = 4$).

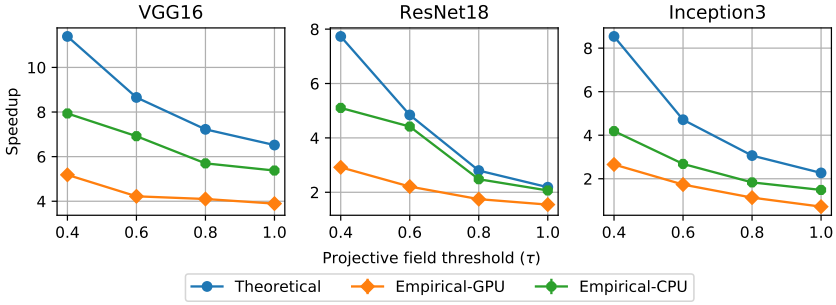


Figure 6.12: Speedups with incremental inference combined with only projective field thresholding.

in both environments.

Projective Field Thresholding. We vary τ from 1.0 (no approximation) to 0.4. Adaptive drill-down is disabled but note that this optimization builds on top of our incremental inference. The occlusion patch size is 16 and stride is 4. Figure 6.12 shows the results. The speedups go up steadily as τ drops for all 3 CNNs. Once again, Empirical-CPU nears the theoretical speedups on ResNet18, but the gap between Empirical-GPU and Empirical-CPU remains due to the disproportionate impact of memory stalls on GPU. Overall, this approximation offers some speedups in both environments, but has a higher impact on CPU than GPU.

Adaptive Drill-Down. Finally we study the effects of adaptive drill-down (again, on top of incremental inference) and disable projective field thresholding. The occlusion patch size is 16. Stage two stride is $S_2 = 4$. First, we vary $r_{drill-down}$, while fixing stage one stride ($S_1 = 16$). Figure 6.13 (a) shows the results. Next, we vary S_1, while fixing $r_{drill-down} = 0.25$.

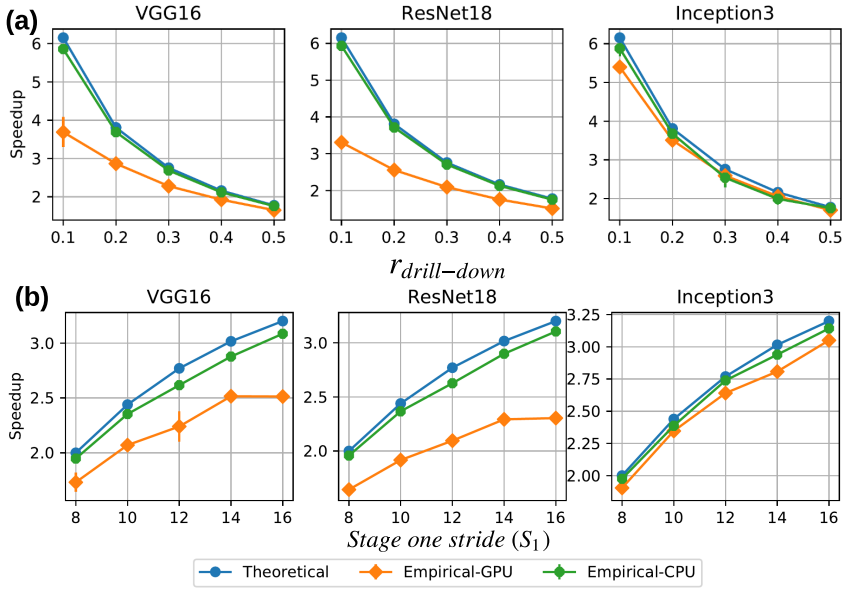


Figure 6.13: Speedups with incremental inference combined with adaptive drill-down. For (a), we set $S_1 = 16$. For (b), we set $r_{drill\_down} = 0.25$).

Figure 6.13 (b) shows the results. As expected, the speedups go up as $r_{drill-down}$ goes down or S_1 goes up, since fewer re-inference queries arise in both cases. Empirical-CPU almost matches the theoretical speedups across the board; in fact, even Empirical-GPU almost matches theoretical speedups on Inception3. Empirical-GPU flattens out at high S_1, since the number of re-inference queries drops, thus resulting in diminishing returns for the benefits of batched execution on GPU. Overall, this optimization has a major impact on speeding up OBE for all CNNs in both environments.

Memory Overhead. We compare our batched incremental inference against full re-inference on GPU. Our approach actually *reduces* memory footprint by 58%. We explain this result further in Appendix B.6.

6.5.3 Summary and Discussion

Overall, our experiments show that KRYPTON can substantially accelerate OBE, with up to 16X speedups on GPU and 34.5X speedups on CPU. The benefits of our optimizations depend

on the CNN's architectural properties. Our approximate inference optimizations also depend on the dataset's properties due to their tunable parameters, which KRYPTON can tune automatically. Finally, KRYPTON sees higher speedups on CPU than GPU but the runtimes are much lower on GPU. Overall, our optimizations in KRYPTON help reduce waiting times for OBE users by improving utilization of existing resources rather than forcing users to buy more resources.

6.6 Conclusion

Deep CNNs are gaining widespread adoption for image prediction tasks but their internal workings are unintuitive for most users. Thus, occlusion-based explanations (OBE) have become a popular mechanism for non-technical users to understand CNN predictions. But OBE is highly compute-intensive due to the large number of CNN re-inference requests produced. In this work, we formalize OBE from a data management standpoint and introduce several query optimization-inspired techniques to speed up OBE. Our techniques span exact incremental inference and multi-query optimization for CNN inference, as well as CNN-specific and human perception-aware approximate inference. Overall, our ideas yield even over an order of magnitude speedups for OBE in both CPU and GPU environments.

Chapter 6 contains material from "Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations" by Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou, which appears in Proceedings of 2019 ACM SIGMOD International Conference on Management of Data. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Extensions of KRYPTON

7.1 Extension: Interactive Diagnosis of CNN Predictions using KRYPTON

7.1.1 Introduction

In this work, we develop a system for interactive diagnosis of CNN predictions using occlusion-based explanations (OBE). Our system internally uses the KRYPTON engine to accelerate OBE using incremental and approximate inference techniques. Users can now also select a subregion of the image to run OBE using a cropping tool to exploit their intuitions about what regions might be more important. We also provide runtime estimations for the OBE workload. A short video of our system can be found here: https://youtu.be/1OWddbd4n6Y.

7.1.2 User Interface

The user interface is developed as a web application running in the browser. Figure 7.1 shows an image of the user interface. To run the OBE workload on an image users need to load the image from the file picker option. The image will be then displayed on the left-hand side

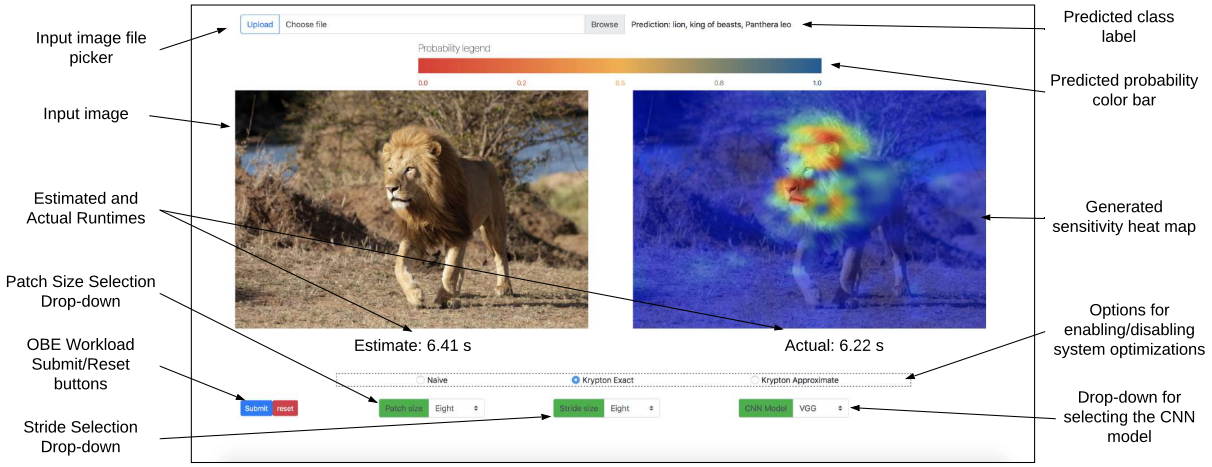


Figure 7.1: KRYPTON user interface. Users can load an input image, select a CNN model, and interactively diagnose the prediction by occluding parts of the full image or part of the image using the cropping tool. KRYPTON generates a sensitivity heatmap (right image) and iteratively refines it as the user progresses. NB: This figure is best viewed in color, as is standard in the visual computing literature.

panel of the interface. After loading the image, users have three options to choose from to run OBE: Naive, KRYPTON-Exact, and KRYPTON-Approximate. The Naive approach performs full CNN inference. KRYPTON-Exact performs our incremental CNN inference optimization, and KRYPTON-Approximate performs both our incremental and approximate optimizations.

Users can then select the occlusion patch size, stride, and the CNN model using the corresponding drop-down menus as shown in Figure 7.1. We allow the user to pick from 4 different patch sizes (4×4, 8×8, 16×16, 32×32) and 4 different stride values (2, 4, 8, 16). Currently, we support three different CNN models (VGG16, ResNet18, Inception3). After picking the above configuration values, the user can then initiate the OBE workload by clicking the "Submit" button. The system will also display an estimate for the runtime of the workload. After the workload is completed, the system will overlay the sensitivity heatmap on the original image and display it on the right-hand side panel. We also show the predicted class label (e.g., lion) and the actual workload execution time. On the heatmap, red color regions correspond to low predicted probabilities for the selected label (e.g., lion) and the blue color regions correspond to

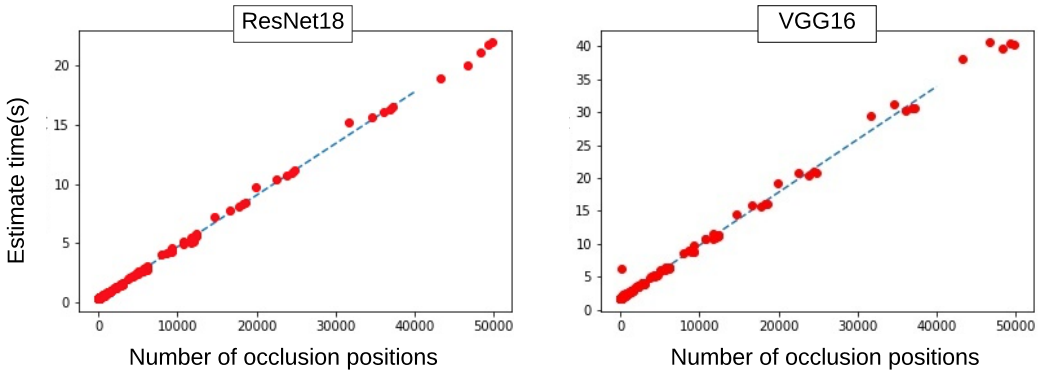


Figure 7.2: Runtime estimation using linear regression cost model (occlusion patch size = 16 and execution mode is exact).

high predicted probabilities for the selected label. So the red color regions are the most sensitive regions for the predicted class label. The color bar for the heatmap is also shown in the interface.

Instead of considering occlusion patch positions over the entire image, users can also focus on a selected small region. Considering all occlusion patch positions can be wasteful if the significant objects in an image are localized into a small region. For example, consider the image shown in Figure 7.1. It contains an image of a lion on a wilderness background. The main object in this image is the lion which occupies only a small region of the entire image. A user who wants to diagnose the prediction for this image can start the diagnosis by selecting a smaller region, which contains only the face and the body of the lion by cropping that region. This will execute faster than the full image. If the user is not satisfied with the produced heatmap, she can iteratively refine the selected region. The cropping of an image can be done simply by dragging on the image in the interface to select a rectangular selection area.

7.1.3 OBE Runtime Estimation

To make the UI more user-friendly, we provide runtime estimations for the OBE workload based on the configurations selected by the user. Showing the estimated runtime for the OBE workload will enable the user to make an informed decision when picking the OBE configurations

to trade-off the quality of the heatmap and the available time budget. The runtime of a single OBE workload depends on the width and height of the selected image region, the stride value, occlusion patch size, selected CNN model, and the execution mode (i.e. exact vs approximate). When we control for the occlusion patch size, CNN model, and the execution mode, the runtime of the workload is directly proportional to the number of different occlusion patch positions. We ran several offline experiments with different configurations and recorded the runtimes. These runtimes are then used to fit a linear regression cost model for each patch size, CNN model, and execution mode combination and are then used to predict the runtime for new configuration instances. Figure 7.2 shows the cost models generated for ResNet18 and VGG16 for a patch size of 16 with the exact execution mode.

7.2 Extension: Accelerating OBE for Arbitrary CNNs

In this work, we extend KRYPTON so that it can accelerate OBE for arbitrary PyTorch CNNs. To achieve this, we develop a high-level abstraction called `KryptonGraph` and automate the generation and execution of it. For a given CNN, `KryptonGraph` handles the incremental CNN inference of that CNN by using PyTorch. The high-level process for `KryptonGraph` generation and execution is shown in Figure 7.3 and works as follows:

1. Given a CNN model f, we use the utilities available in PyTorch to trace the structure of the CNN by providing a sample image as input. Since all CNNs are static in nature (i.e., the order of operator execution is not dependent on data), the structure obtained by tracing is guaranteed to be correct. The trace output is then exported to ONNX format [42], which is a convenient representation format for subsequent analysis.

2. Dropout [263] operators in the CNN model are simply ignored as they do not have any effect on CNN inference.

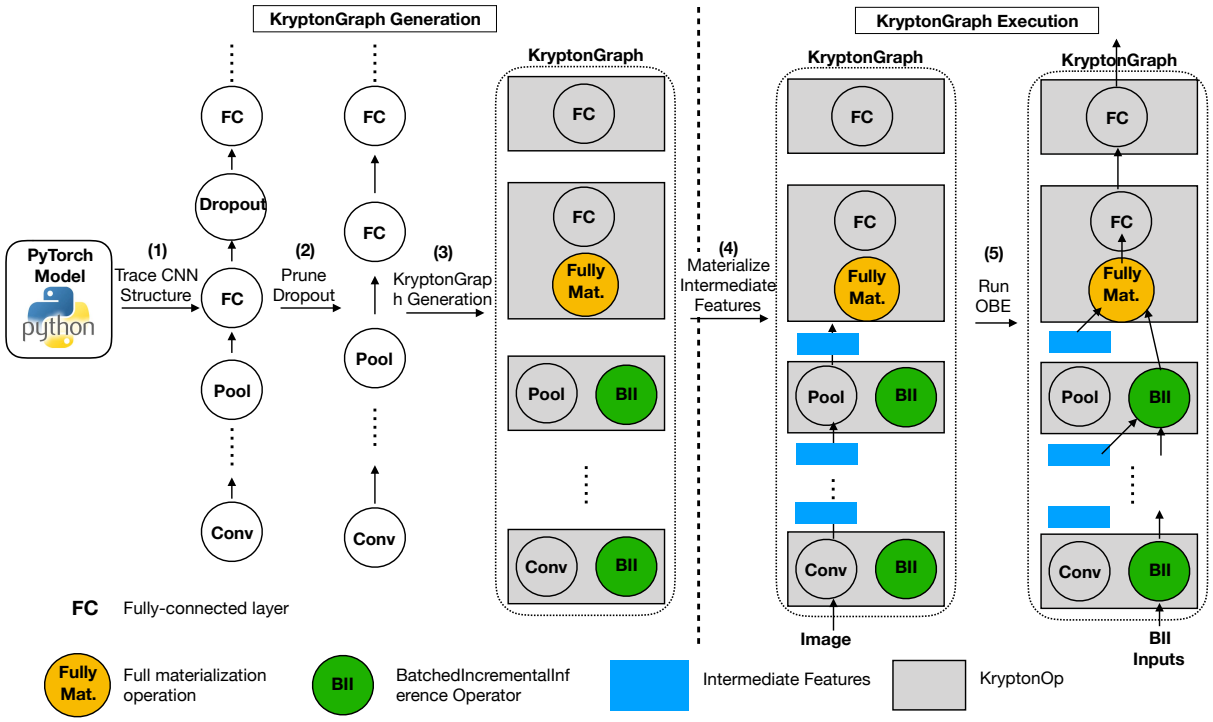


Figure 7.3: `KryptonGraph` generation and execution process. For brevity, only a subgraph of a linear CNN is shown. The same method also applies to arbitrary DAG like CNNs.

3. We then traverse the exported CNN model in topological order and create the corresponding `KryptonGraph`. For each operator T in the original CNN f, there will be a corresponding `KryptonOP` in the `KryptonGraph` that implements the `BatchedIncrementalInference` (Algorithm 6) for local context operators. Each `KryptonOp` also has a reference to the original CNN operator T, which will be used in the `BatchedIncrementalInference` method or directly invoked for global context operators that do not support incremental inference (e.g., fully-connected). Under the hood, KryptonOP is relying on the PyTorch framework for the actual execution of the corresponding CNN operator. The first global context operator that succeeds a local context operator will first fully materialize the updated input before invoking the full inference operator. Since all CNNs are created using a small number of low-level operators (e.g., convolution, pooling, and fully-connected), by implementing all corresponding types of `KryptonOps`, we are able to support arbitrary

PyTorch CNNs as input.

4. The generated `KryptonGraph` is then used for performing CNN inference for OBE. Given an input image $I_{:img}$ we first materialize all intermediate outputs corresponding to incremental inference operators using one full inference.

5. We then prepare occluded images ($I'_{(x,y):img}$) for all positions in G. For batches of $I'_{(x,y):img}$ as the input, we invoke the `KryptonGraph` in topological order and calculate the corresponding entries of heatmap M.

Chapter 7 Section 7.1 contains material from "Demonstration of Krypton: Optimized CNN Inference for Occlusion-based Deep CNN Explanations" by Allen Ordookhanians, Xin Lin, Supun Nakandala, and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 12, Issue 12, August 2019. The dissertation authors contribution was in the conceptualization of the system and advising the junior students through the system implementation.

Chapter 7 Section 7.2 contains material from "Incremental and Approximate Computations for Accelerating Deep CNN Inference" by Supun Nakandala, Kabir Nagrecha, Arun Kumar, and Yannis Papakonstantinou, which appears in ACM Transactions on Database Systems Journal Volume 45, Issue 4, December 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 8

VISTA: Query Optimizations for Deep CNN Feature Transfer

8.1 Introduction

In this chapter, we dive deeper into our techniques for optimizing deep convolutional neural network (CNN) feature transfer workloads. CNNs [180, 124] are a special type of DL model family optimized for image data and there is growing interest in using CNNs to exploit images in analytics applications that have hitherto relied mainly on structured data. But analytics systems today have a dichotomy: dataflow systems (e.g., Spark [289]) are popular for structured data [201, 5], while DL systems (e.g., TensorFlow [54]) are needed for CNNs. This dichotomy means the systems issues of workloads that combine both forms of data are surprisingly ill-understood.

Example (Based on [199]). Consider a data scientist, Alice, at an online fashion retailer building a product recommender system (see Figure 8.1). She uses structured features (e.g., price, brand, user clicks, etc.) to build an ML model (e.g., logistic regression, multi-layer perceptron, or decision tree) to predict product ratings. She then has a hunch that including product images can

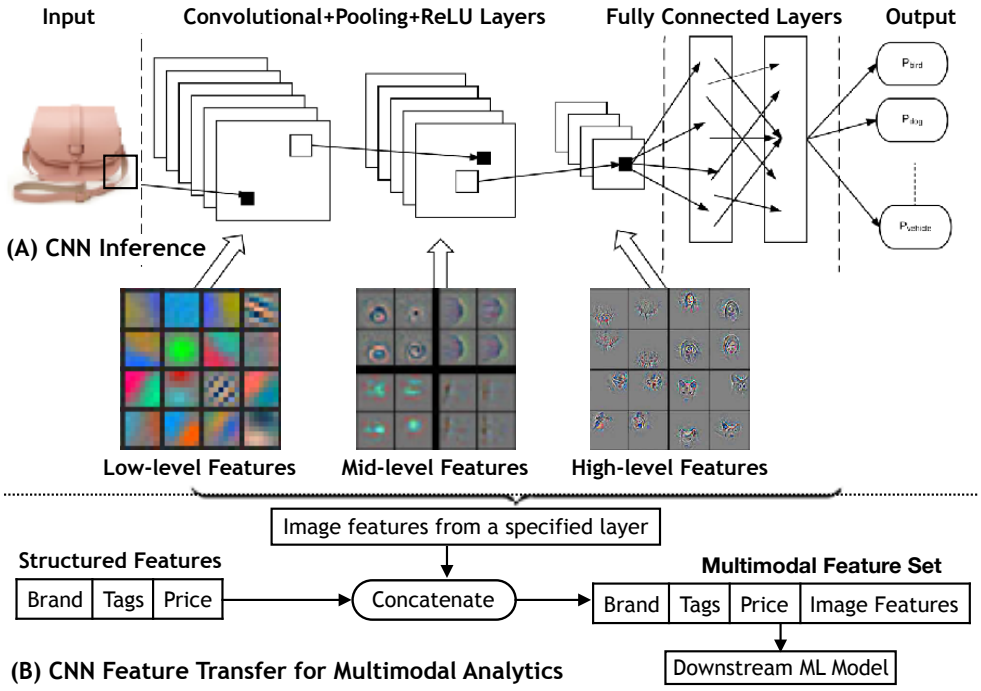


Figure 8.1: (A) Simplified illustration of a typical deep CNN and its hierarchy of learned feature layers(based on [290]). (B) Illustration of the CNN feature transfer workflow for multimodal analytics.

raise ML accuracy. So, she uses a pre-trained deep CNN (e.g., ResNet50 [136]) on the images to extract a *feature layer*: a vector representation of an image produced by the CNN. Deep CNNs produce a series of feature layers; each layer automatically captures a different level of abstraction from low-level patterns to high-level abstract shapes [124, 180], as Figure 8.1(A) illustrates. Alice concatenates her chosen feature layer with the structured features and trains her "downstream" model. Figure 8.1(B) illustrates this workflow. She then tries a few other feature layers instead to check if the downstream model's accuracy increases.

Importance of Feature Transfer. Feature transfer is a form of "transfer learning" that mitigates two key pains of training deep CNNs from scratch [226, 4, 25]: the number of labeled images needed is lower, often by an order of magnitude [25, 287], and the time and resource costs of training are lower, even by two orders of magnitude [4, 25]. Overall, feature transfer is now popular in many domains, including recommender systems [199], visual search [153]

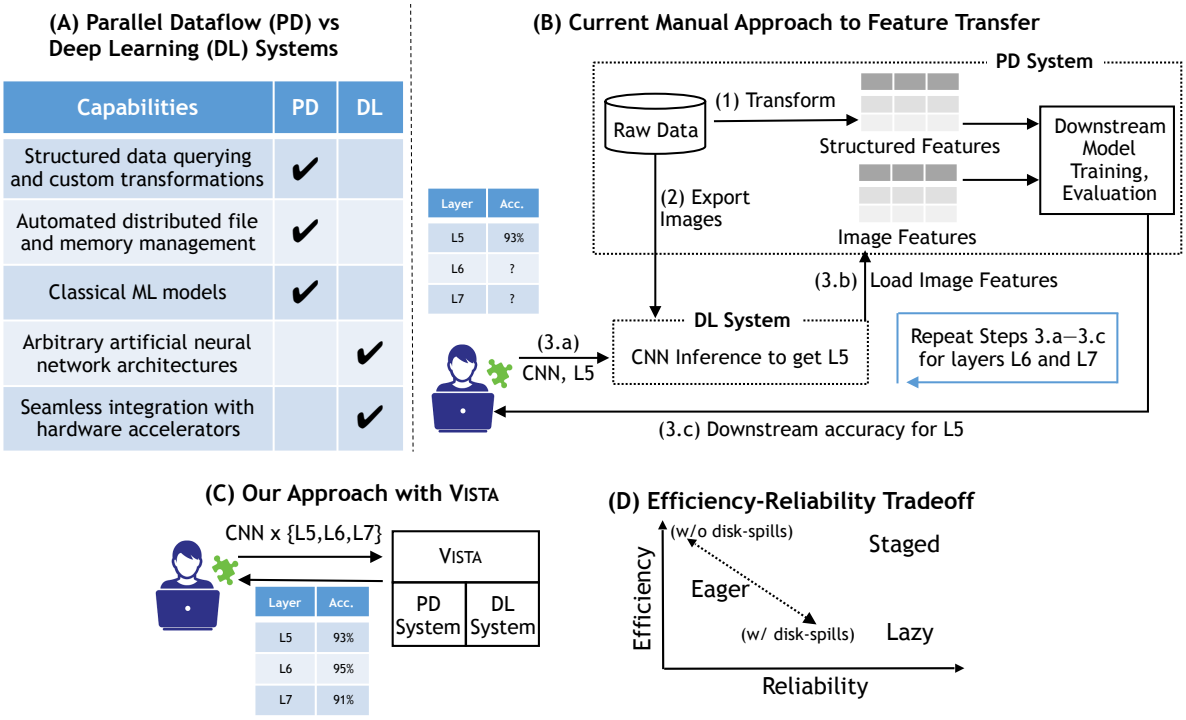


Figure 8.2: (A) Comparing the analytics-related capabilities of parallel dataflow (PD) systems and deep learning (DL) systems. (B) Current manual approach of executing feature transfer at scale straddling PD and DL systems. The steps in the manual workflow are numbered. Step 3 (a-b-c) is repeated for every feature layer of interest. (C) The "declarative" approach in VISTA. (D) Tradeoffs of alternative execution plans on efficiency (runtimes) and reliability (crash-proneness).

(product images), healthcare (tissue images) [114], nutrition science (food images) [17], and computational advertising (ad images).

Bottleneck: Trying Multiple Layers. Recent work in ML showed that it is *critical to try multiple layers* for feature transfer because different layers yield different accuracies and it is impossible to tell upfront which layer will be best [110, 287, 66, 25]. But trying multiple layers becomes a bottleneck for data scientists running large-scale ML on a cluster because it can slow down their analysis, e.g., from an hour to several hours (Section 8.4), and/or raise resource costs.

8.1.1 Current Approach and Systems Issues

The common approach to feature transfer at scale is a *tedious manual process* straddling DL systems and parallel dataflow (PD) systems. These systems present a dichotomy, as Figure 8.2(A) shows. PD systems support queries and manage distributed memory for structured data but do not support DL natively. DL systems support complex CNNs and hardware accelerators but need manual partitioning of files and memory for distributed execution. Moreover, data scientists often prefer decision tree-based ML models on structured data [10]; thus, a DL system alone is too limiting.

Figure 8.2(B) illustrates the manual process. Suppose Alice tries layer 5 (L5) to layer 7 (L7) (say) from a given CNN. She first runs CNN inference in DL system (e.g., TensorFlow) to materialize L5 for all images in her dataset. She loads this large data file with image features into PD system (e.g., Spark), joins it with the structured data, and trains a downstream multimodal ML model (e.g., using MLlib [201] or TensorFlow). She repeats this for L6 and then for L7. Apart from being tedious, this process faces two key systems issues:

(1) Inefficiency. Extracting a higher layer (say, L6) requires a *superset of the inference computations* needed for a lower layer (say, L5). So, the manual process may have high *computational redundancy*, which wastes runtime.

(2) Crash-proneness. One might ask: *why not write out all layers in one go to save time?* Alas, CNN feature layers can be very large, e.g., one of ResNet50's layers is 784KB but the image is only 14KB [136]. So, 10GB of data blows up to 560GB for just one layer! Forcing ML users to handle such large intermediate data files on in-memory PD systems can easily cause workload crashes due to exhausting available system memory. Alternatively, writing these feature files to disk and reading iteratively will incur significant overheads due to costly disk reads/writes, thus reducing efficiency further.

8.1.2 Our Proposed Approach

We resolve the above issues by elevating scalable feature transfer to a "declarative" level and automatically optimizing its execution. We want to retain the benefits of both PD and DL systems without reinventing their current capabilities (Figure 8.2(A)). Thus, we build a new data system we call VISTA *on top* of PD and DL systems, as Figure 8.2(C) illustrates. To make practical adoption easier, we believe it is crucial to *not modify the code* of the underlying PD and DL systems; this also lets us leverage future improvements to those systems. VISTA is based on three design decisions: (1) *Declarativity* to simplify specification, (2) *Execution Optimization* to reduce runtimes, and (3) *Automated Memory and System Configuration* to avoid memory-related workload crashes.

(1) Declarativity. VISTA lets users specify *what* CNNs and layers to try, but not *how* to run them. It invokes the DL system to run CNN inference, loads and joins image features with structured data, and runs downstream training on the PD system. Since VISTA, not the user, handles how layers are materialized, it can optimize execution in non-trivial ways.

(2) Execution Optimization. We characterize the memory use behavior of this workload in-depth, including key workload crash scenarios. This helps us bridge PD and DL systems, since PD systems do not understand CNNs and DL systems do not understand joins or caching. We compare alternative *execution plans* with different efficiency–reliability tradeoffs, as Figure 8.2(D) shows. The "Lazy" plan simply automates the manual process. It is reliable due to its low memory footprint, but it has high computational redundancy. At the other end, "Eager" materializes all layers of interest in one go (Section 8.3). It avoids redundancy but is prone to memory-related crashes if the intermediate data does not fit in memory. Alternatively, enabling disk spills for the Eager plan will avoid crashes but will be inefficient due to costly disk reads/writes. We then present a new plan used in VISTA that offers the best of both worlds: "Staged" execution; it *interleaves* the DL and PD systems' operations by enabling *partial CNN inference*.

(3) Automated Memory and System Configuration. Finally, we explain how key system tuning knobs affect this workload: apportioning memory for caching data, CNNs, and feature layers; data partitioning; and physical join operator. Using our insights, we build an *end-to-end automated optimizer* in VISTA to configure both the PD and DL systems to run this workload efficiently and reliably.

Implementation and Evaluation. We prototype VISTA on top of two PD systems, Spark and Ignite [72], with TensorFlow as the DL system. Our API is in Python. We perform an extensive empirical evaluation of VISTA using 2 real-world multimodal datasets and 3 deep CNNs. VISTA avoids many crash scenarios and reduces total runtimes by 58% to 92% compared to existing baseline approaches.

Our approach is inspired by the long line of work on multi-query optimization in RDBMSs [250]. But our execution plans and optimizer have no counterparts in prior work because they treat CNNs as black-box user-defined functions that they do not rewrite. In contrast, VISTA treats CNNs as first-class operations, understands their memory footprints, rewrites their inference, and optimizes this workload in a principled and holistic manner.

Overall, this work makes the following contributions:

- To the best of our knowledge, this is the first work on the systems principles of integrating PD and DL systems to optimize scalable feature transfer from CNNs.
- We characterize the memory use behavior of this workload in-depth, explain the efficiency–reliability tradeoffs of alternative execution plans, and present a new CNN-aware optimized execution plan.
- We create an automated optimizer to configure the system and optimize its execution to offer both high efficiency and high reliability.
- We prototype our ideas to build VISTA on top of a PD and DL system. We com-

pare VISTA against baseline approaches using multiple real-world datasets and deep CNNs. Unlike the baselines, VISTA never crashes and is also faster by 58% to 92%.

Outline. Section 8.2 formalizes the dataflow of the feature transfer workload, explains our assumptions, and provides an overview of VISTA. Section 8.3 dives into the systems tradeoffs and presents our optimizer. Section 8.4 presents the experiments and we conclude in Section 8.5.

8.2 Preliminaries and Overview

We now formally describe our problem setting, explain our assumptions, and present an overview of VISTA.

8.2.1 Definitions and Data Model

We start by defining some terms and notation to formalize the data model of *partial CNN inference*. We will use these terms in the rest of this chapter.

Definition 8.2.1 *A* tensor *is a multidimensional array of numbers. The* shape *of a d-dimensional tensor $t \in \mathbb{R}^{n_1 \times n_2 \times \ldots n_d}$ is the d-tuple $(n_1, \ldots n_d)$.*

A raw image is the (compressed) file representation of an image, e.g., JPEG. An image tensor is the numerical tensor representation of the image. Gray-scale images have 2-dimensional tensors; colored ones, 3-dimensional (with RGB pixel values). We now define some abstract data types and functions that will be used to explain our techniques.

Definition 8.2.2 *A* TensorList *is an indexed list of tensors of potentially different shapes.*

Definition 8.2.3 *A* TensorOp *is a function f that takes as input a tensor t of a fixed shape and outputs a tensor $t' = f(t)$ of potentially different, but also fixed, shape. A tensor t is said to be* shape-compatible *with f iff its shape conforms to what f expects for its input.*

Definition 8.2.4 *A* CNN *is a TensorOp f that is represented as a composition of n_l indexed TensorOps, denoted $f(\cdot) \equiv f_{n_l}(\dots f_2(f_1(\cdot))\dots)$, wherein each TensorOp f_i is called a* layer *and n_l is the* number of layers.[1] *We use $\hat{f}_i$ to denote $f_i(\dots f_2(f_1(\cdot))\dots)$.*

Definition 8.2.5 *A* FlattenOp *is a TensorOp whose output is a vector; given a tensor $t \in \mathbb{R}^{n_1 \times n_2 \times \dots n_d}$, the output vector's length is $\sum_{i=1}^{d} n_i$.*

Definition 8.2.6 CNN inference*. Given a CNN f and a shape-compatible image tensor t,* CNN inference *is the process of computing $f(t)$.*

Definition 8.2.7 Partial CNN inference*. Given a CNN f, layer indices i and $j > i$, and a tensor t that is shape-compatible with layer f_i,* partial CNN inference *$i \rightarrow j$ is the process of computing $f_j(\dots f_i(t)\dots)$, denoted $\hat{f}_{i \rightarrow j}$.*

All major CNN layers–convolutional, pooling, non-linearity, and fully connected–are just TensorOps. The above definitions capture a crucial aspect of partial CNN inference: data flowing through the layers produces a sequence of tensors.

8.2.2 Problem Statement and Assumptions

We are given two tables $T_{str}(\underline{ID}, X)$ and $T_{img}(\underline{ID}, I)$, where *ID* is the primary key (identifier), $X \in \mathbb{R}^{d_s}$ is the structured feature vector (with d_s features, including label), and I are raw images (say, as files on HDFS). We are also given a CNN f with n_l layers, a set of layer indices $L \subset [n_l]$ specific to f that are of interest for transfer learning, a downstream ML algorithm M (e.g., logistic regression), a set of system resources R (number of cores, system memory, and number of nodes). The feature transfer workload is to train M for each of the $|L|$ feature vectors obtained by concatenating X with the respective feature layers obtained by partial CNN inference; we can state it more precisely as follows:

[1] We use sequential (chain) CNNs for simplicity of exposition; it is easy to extend our definitions to DAG-structured CNNs such as DenseNet [145].

$$\forall\, l \in L: \tag{8.1}$$

$$T'_{img,l}(\underline{ID}, g_l(\hat{f}_l(I))) \;\leftarrow\; \text{Apply } (g_l \circ \hat{f}_l) \text{ to } T_{img} \tag{8.2}$$

$$T'_l(\underline{ID}, X'_l) \;\leftarrow\; T_{str} \bowtie T'_{img,l} \tag{8.3}$$

$$\text{Train } M \text{ on } T'_l \text{ with } X'_l \equiv [X, g_l(\hat{f}_l(I))] \tag{8.4}$$

Step (2) runs partial CNN inference to materialize layer l and flattens it with g_l, a shape-compatible FlattenOp. Step (3) concatenates structured and image features using a key-key join. Step (4) trains M on the concatenated feature vector. Pooling can be inserted before g to reduce dimensionality for M [66]. The current approach (Figure 8.2(B)) runs the above queries as such, i.e., materialize layers *manually* and *independently* as flat files and transfer them; we call this execution plan *Lazy*. This plan is cumbersome, inefficient due to *redundant* CNN inference, and/or is prone to workload crashes due to inadvertently mismanaged memory. Our goal is to resolve these issues. *Our approach is to elevate this workload to a declarative level, obviate manual feature transfer, automatically reuse intermediate results, and optimize the system configuration and execution for better reliability and efficiency.*

We make a few simplifying assumptions for tractability in this first work on this problem. First, we assume that f is from a roster of well-known CNNs. We currently support AlexNet [171], VGG16 [260], and ResNet50 [136] due to their popularity in real feature transfer applications [199, 278]. Second, we support only one image per data record. We leave support for arbitrary CNNs and multiple images per example to future work. Finally, we assume enough secondary storage is available for disk spills and optimize the use of distributed memory; this is a standard assumption in PD systems.

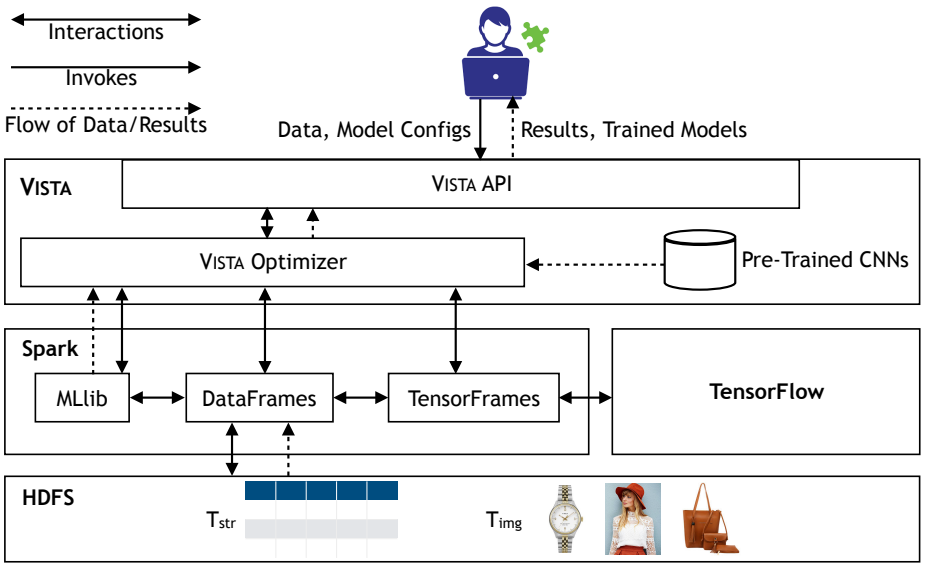


Figure 8.3: System architecture of the VISTA prototype on top of the Spark-TensorFlow combine. The prototype on Ignite-TenforFlow is similar and skipped for brevity.

8.2.3 System Architecture and API

We prototype VISTA as a library on top of Spark-TF and Ignite-TF environments. Due to space constraints, we explain the architecture of only the Spark-TF prototype; the Ignite-TF one is similar.

VISTA has three components, as Figure 8.3 illustrates: (1) a "declarative" API, (2) a roster of popular named deep CNNs with numbered feature layers, and (3) the VISTA optimizer. Our Python API expects 4 major groups of inputs. First is the system environment (memory, number of cores, and number of nodes). Second, a deep CNN f and the number of feature layers $|L|$ (starting from the top most layer) to explore. Third, the downstream ML routine M that handles the downstream model's evaluation, hyperparameter tuning, and model artifacts. Fourth, data tables T_{str} and T_{img} and statistics about the data.

Under the covers, VISTA invokes its optimizer to pick a fast and reliable set of choices for the logical execution plan, system configuration parameters, and physical execution decisions. After configuring Spark accordingly, VISTA runs within the Spark Driver process to control the execution. VISTA injects UDFs to run (partial) CNN inference, i.e., f, $\hat{f}_l$, g_l, and $\hat{f}_{i \to j}$

for the CNNs in its roster (currently, AlexNet, VGG16, and ResNet50). These UDFs specify the computational graphs for TF and invoke Spark's *DataFrames* and *TensorFrames* APIs with appropriate inputs based on our optimizer's decisions. Image and feature tensors are stored with our custom *TensorList* datatype. Finally, VISTA invokes downstream ML model training on the concatenated feature vector and obtains $|L|$ trained downstream models. *Overall,* VISTA *frees ML users from manually writing TF code for CNN feature transfer, saving features as files, performing joins, or tuning Spark for running this workload at scale.*

8.3 Tradeoffs and Optimizer

We now characterize the abstract memory usage behavior of our workload in depth. We then map our memory model to Spark and Ignite. Finally, we use these insights to explain three dimensions of efficiency-reliability tradeoffs and apply our analyses to design the VISTA optimizer.

8.3.1 Memory Use Characterization

It is important to understand and optimize the memory use behavior of the feature transfer workload, since mismanaged memory can cause frustrating workload crashes and/or excessive disk spills or cache misses that raise runtimes. Apportioning and managing distributed memory carefully is a central concern for modern distributed data processing systems. Since our work is not tied to any specific dataflow system, we create an *abstract model of distributed memory apportioning* to help us explain the tradeoffs in a generic manner. These tradeoffs involve apportioning memory between intermediate data, CNN/DL models and working memory for UDFs. Such tradeoffs affect both reliability (avoiding crashes) and efficiency. We then highlight interesting new properties of our workload that can cause unexpected crashes or inefficiency, if not handled carefully.

Abstract Memory Model. In distributed memory-based dataflow systems, a worker's System Memory is split into two main regions: Reserved Memory for OS and other processes and Workload Memory, which in turn is split into Execution Memory and Storage Memory. Figure 8.4(A) illustrates the regions. Execution Memory is further split into User Memory and Core Memory; for typical relational/SQL workloads, the former is used for UDF execution, while the latter is used for query processing. Best practice guidelines recommend allotting most of System Memory to Storage Memory, while having enough Execution Memory to reduce disk spills or cache misses [8, 19, 20]. OS Reserved Memory is typically a few GBs. Our workload requires rethinking memory apportioning due to interesting new issues caused by deep CNN models, (partial) CNN inference, feature layers, and the downstream ML task.

(1) The guideline of using most of System Memory for Storage and Execution no longer holds. In both Spark and Ignite, CNN inference in DL system (e.g., TF) uses System Memory *outside* Storage and Execution regions. If a DL model is used as the downstream ML model, it will also use memory outside of Storage and Execution regions. The memory footprint of DL models is non-trivial. For parallel query execution in PD systems, each execution thread will spawn its own DL model replica, multiplying the footprint.

(2) Many temporary objects are created when reading serialized DL models to initialize the DL system, for buffers to read inputs, and to hold features created by CNN inference. All these go under User Memory. The sizes of these objects depend on the number of examples in a data partition, the CNN, and L. These sizes could vary dramatically and also be very high, e.g., layer *fc6* of AlexNet has 4096 features but *conv5* of ResNet has over 400,000 features! Such complex memory footprint calculations will be tedious for ML users.

(3) The downstream ML routine (e.g., MLLib) also copies features produced by CNN inference into its own representations. Thus, Storage Memory should accommodate such intermediate data copies. Finally, Core Memory must accommodate the temporary objects created for join processing.

Mapping to Spark's Memory Model. Spark allocates User, Core, and Storage Memory regions of our abstract memory model from the JVM Heap Space. With default configurations, Spark allocates 40% of the Heap Memory to User Memory region. The rest of the 60% is shared between the Storage and Core Memory regions. The Storage Memory–Core Memory boundary in Spark is not static. If needed, Core Memory automatically borrows from the Storage Memory evicting data partitions to the disk. Conversely, if Spark needs to load more data to memory, it borrows from Execution Memory. But there is a maximum threshold fraction of Storage Memory (default 50%) that is immune to eviction.

Mapping to Ignite's Memory Model. Ignite treats both User and Core Memory regions as a single unified memory region and allocates the entire JVM Heap for it. This region is used to store the in-memory objects generated by Ignite during query processing and UDF execution. Storage Memory region of Ignite is allocated *outside* of JVM heap in the JVM native memory space. Unlike Spark, Ignite's in-memory Storage Memory region has a static size.

Memory-related Crash and Inefficiency Scenarios. The three issues explained above give rise to various unexpected workload crash scenarios due to memory errors, as well as system inefficiencies. Manually handling them could frustrate data scientists and impede their ML exploration.

(1) DL Execution Memory blowups. Serialized file formats of CNNs and downstream ML models often underestimate their in-memory footprints. Along with the replication by multiple threads, DL Execution Memory can be easily exhausted. If such blowups are not accounted when configuring the data processing system, and if they exceed available memory, the OS will kill the application.

(2) Insufficient User Memory. All UDF execution threads share User Memory for the CNNs, downstream ML models, and feature layer *TensorList* objects. If this region is too small due to a small overall Workload Memory size or due to a large degree of parallelism, such objects

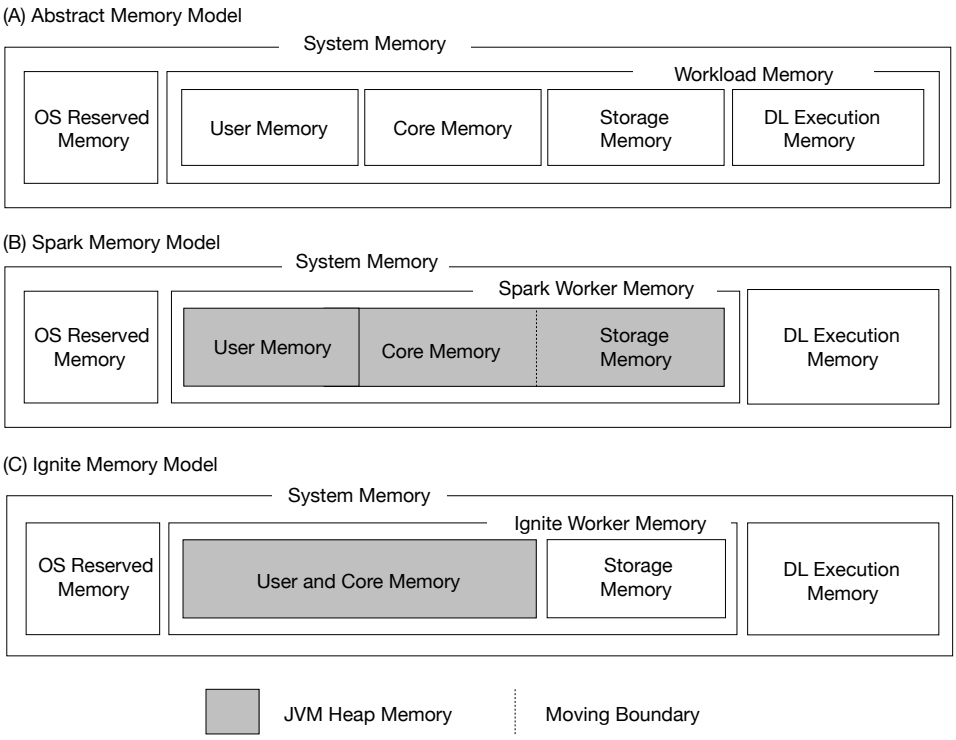


Figure 8.4: (A) Our abstract model of distributed memory apportioning. (B,C) How our model maps to Spark and Ignite.

might exceed available memory, leading to a crash with out-of-memory error.

(3) Very large data partitions. If a data partition is too big, the PD system needs a lot of User and Core Execution Memory for query execution operations (e.g., for the join in our workload and *MapPartition*-style UDFs in Spark). If Execution Memory consumption exceeds the allocated maximum, it will cause the system to crash with out-of-memory error.

(4) Insufficient memory for Driver Program. All distributed data processing systems require a Driver program that orchestrates the job among workers. In our case, the Driver reads and creates a serialized version of the CNN and broadcasts it to the workers. For the downstream ML model, the Driver may also have to collect partial results from workers (e.g., for *collect()* and *collectAsMap()* in Spark). Without enough memory for these operations, the Driver will crash.

Overall, several execution and configuration considerations matter for reliability and efficiency. Next, we delineate these systems tradeoffs precisely along three dimensions.

8.3.2 Three Dimensions of Tradeoffs

The dimensions we discuss are largely orthogonal to each other but they affect reliability and efficiency collectively.

Logical Execution Plan Tradeoffs

Figure 8.5(A) illustrates the *Lazy* plan (Section 8.2.2). As mentioned earlier, it has high computational redundancy; to see why, consider a popular deep CNN AlexNet with the last two layers *fc7 and fc8* used for feature transfer ($L = \{fc7, fc8\}$). This plan performs partial CNN inference for *fc7* (721 MFLOPS) independently of *fc8* (725 MFLOPS), incurring 99% redundant computations for *fc8*. An orthogonal issue is *join placement*: *should the join really come after inference?* Usually, the total size of all feature layers in L will be larger than the size of raw images in a compressed format such as JPEG. Thus, if the join is pulled below inference, as shown in Figure 8.5(B), the shuffle costs of the join will go down. We call this slightly modified plan *Lazy-Reordered*. But this plan still has computational redundancy. The only way to remove redundancy is to break the independence of the $|L|$ queries and fuse them.

Consider the *Eager* plan shown in Figure 8.5(C). It materializes all feature layers of L *in one go*, which avoids redundancy because CNN inference is not repeated. Features are stored as a *TensorList* in an intermediate table and joined with T_{str}. M is then trained on each feature layer (concatenated with X) projected from the *TensorList*. *Eager-Reordered*, shown in Figure 8.5(D), is a variant with the join pulled down. Alas, both of these plans have high memory footprints, since they materialize all of L at once. Depending on the memory apportioning (Section 8.3.1), this could cause workload crashes or a lot of disk spills, which in turn raises runtimes.

To resolve the above issues, we create a logically new execution plan we call *Staged* execution, shown in Figure 8.5(E). It splits partial CNN inference across the layers in L and invokes M on branches of the inference path; so, it stages out the materialization of the feature tensors. *Staged* offers the best of both worlds: it avoids computational redundancy, and it is

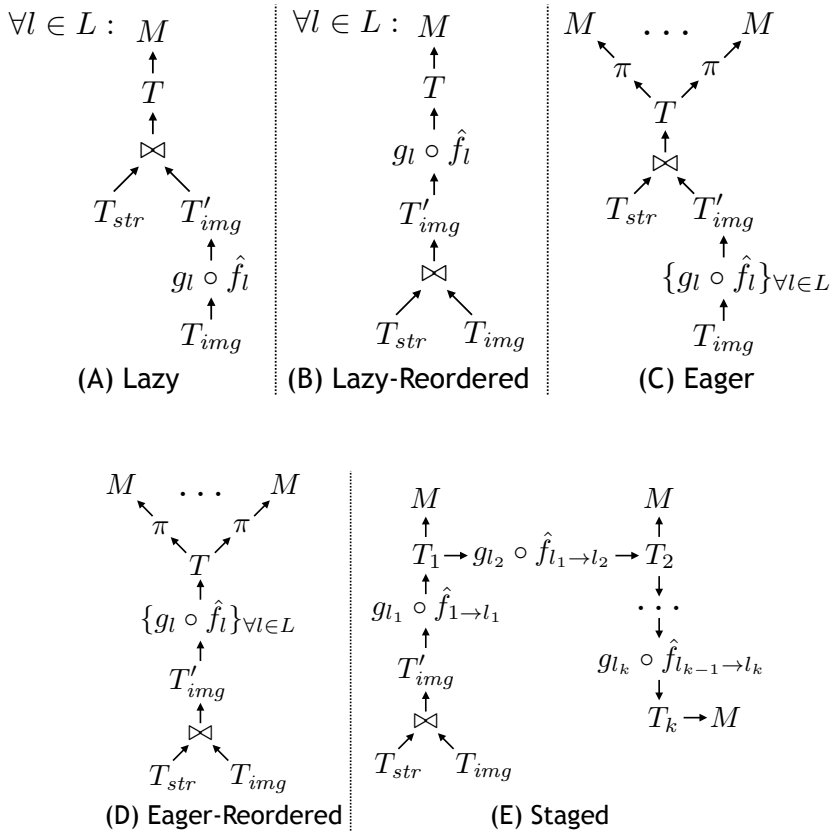


Figure 8.5: Alternative logical execution plans (let $k = |L|$). (A) Lazy, the de facto current approach. (B) Reordering the join operator in Lazy. (C) Eager execution plan. (D) Reordering the join operator in Eager. (E) Our new Staged execution plan.

reliable due to its lower memory footprints. Empirically, we find that *Eager* and *Eager-Reordered* are seldom much faster than *Staged* due to a peculiarity of deep CNNs. The former can be faster only if a CNN "quickly" (i.e., within a few layers and low FLOPs) converts the image to small feature tensors. But such an architecture is unlikely to yield high accuracy, since it loses too much information too soon [124]. Indeed, no popular deep CNN has such an architecture. Based on our above analysis, we find that it suffices for VISTA to only use our new *Staged* plan (validated in Section 8.4).

System Configuration Tradeoffs

Logical plans are generic and independent of the PD system. But as explained in Section 8.3.1, three key system configuration parameters matter for reliability and efficiency: degree of parallelism in a worker, data partition sizes, and memory apportioning. While the tuning of such parameters is well understood for SQL and MapReduce [141, 58], we need to rethink them due to the properties of CNNs and partial CNN inference.

Naively, one might choose the following settings that may work well for SQL workloads: the degree of parallelism is the number of cores on a node; allocate few GBs for User and Core Execution Memory; use most of the rest of memory for Storage Memory; use the default number of partitions in the PD system. But for the feature transfer workload, these settings can cause crashes or inefficiencies.

For example, a higher degree of parallelism increases the worker's throughput but also raises the CNN models' footprint, which in turn requires reducing Execution and Storage Memory. Reducing Storage Memory can cause more disk spills, especially for feature layers, and raise runtimes. Worse still, User Memory might also become too low, which can cause crashes during CNN inference. Lowering the degree of parallelism reduces the CNN models' footprint and allows Execution and Storage Memory to be higher, but too low a degree of parallelism means workers will get underutilized.[2] This in turn can raise runtimes, especially for the join and the downstream training. Finally, too low a number of data partitions can cause crashes, while too high a value leads to high overheads. Overall, we see multiple non-trivial systems tradeoffs that are tied to the CNN and its feature layer sizes. It is unreasonable to expect ML users to handle such tradeoffs manually. Thus, VISTA automates these decisions in a feature transfer-aware manner.

[2]In our current prototypes, every TF invocation by a worker uses all cores regardless of how many cores are assigned to that worker. But, one TF invocation per used core increases overall throughput and reduces runtimes.

Physical Execution Tradeoffs

Physical execution decisions are closer to the specifics of the underlying PD system. We discuss the tradeoffs of two such decisions that are common in PD systems and then explain what Spark and Ignite specifically offer.

First is the physical join operator used. The two main options for distributed joins are *shuffle-hash* and *broadcast*. In shuffle-hash join, base tables are hashed on the join attribute and partitioned into "shuffle blocks." Each shuffle block is then sent to an assigned worker over the network, with each worker producing a partition of the output table using a local sort-merge join or hash join. In broadcast join, each worker gets a copy of the smaller table on which it builds a local hash table before joining it with the outer table without any shuffles. If the smaller table fits in memory, broadcast join is typically faster due to lower network and disk I/O costs.

Second is the persistence format for in-memory storage of intermediate data. Since feature tensors can be much larger than raw images, this decision helps avoid/reduce disk spills or cache misses. The two main options are *deserialized* format or *compressed serialized* format. While the serialized format can reduce memory footprint and thus, reduce disk spills/cache misses, it incurs additional computational overhead for translating between formats. To identify potential disk spills/cache misses and determine which format to use, we estimate the size of intermediate data tables $|T_i|$ (for $i \in L$). VISTA can automatically estimate $|T_i|$ because it knows the sizes of the feature tensors in its CNN roster and understands the internal record format of the PD system.

Spark supports both shuffle-hash join and broadcast join implementations, as well as both deserialized and compressed serialized in-memory storage formats. In Ignite, data is shuffled to the corresponding worker node based on the partitioning attribute during data loading itself. Thus, a key-key join can be performed using a local hash join, if we use the same data partitioning function for both tables. Ignite stores intermediate in-memory data in a compressed binary format.

8.3.3 The Optimizer

We now explain how the VISTA optimizer navigates all the tradeoffs in a holistic and automated way to improve both reliability and efficiency. Table 8.1 lists the notation used.

Optimizer Formalization and Simplification. Table 8.1(A) lists the inputs given by the user. From these inputs, VISTA infers the sizes of the structured data table ($|T_{str}|$), the images table ($|T_{img}$), and all intermediate data tables ($|T_i|$ for $i \in L$) shown in Figure 8.5(E). VISTA also looks up the CNN's serialized size $|f|_{ser}$, runtime memory footprint $|f|_{mem}$, and runtime GPU memory footprint $|f|_{mem\_gpu}$ from its roster, in which we store these statistics. Then, VISTA calculates the runtime memory footprint of the downstream model $|M|_{mem}$ based on the specified M and the largest total number of features (based on L). For instance, for logistic regression, $|M|$ is proportional to the sum of structured features and the maximum number of CNN features for any layer ($\max_{l \in L} |g_l(\hat{f}_l(I))|$). Table 8.1(B) lists the variables whose values are set by the optimizer. We define two quantities that capture peak intermediate data sizes to help our optimizer set memory variables reliably:

$$s_{single} = \max_{1 \leq i \leq |L|} |T_i| \tag{8.5}$$

$$s_{double} = \max_{1 \leq i \leq |L|-1} (|T_i| + |T_{i+1}|) - |T_{str}| \tag{8.6}$$

The ideal objective is to minimize the overall runtime subject to memory constraints. As explained in Section 8.3.2, there are two competing factors: cpu and $mem_{storage}$. Raising cpu increases parallelism, which could reduce runtimes. But it also raises the DL Execution Memory needed, which forces $mem_{storage}$ to be reduced, thus increasing potential disk spills/cache misses for T_i's and raising runtimes. This tension is captured by the following objective function:

$$\min_{cpu, n_p, mem_{storage}} \frac{\tau + \max(0, \frac{s_{double}}{n_{nodes}} - mem_{storage})}{cpu} \tag{8.7}$$

Table 8.1: Notation for Section 8.3 and Algorithm 7.

Symbol	Description
	(A) Inputs given by user to VISTA
T_{str}	Structured features table
T_{img}	Images table
f	CNN model in our roster
L	Set of feature layer indices of f to transfer
M	Downstream ML routine
n_{nodes}	Number of worker nodes in cluster
mem_{sys}	Total system memory available in a worker node
mem_{GPU}	GPU memory if GPUs are available
cpu_{sys}	Number of cores available in a worker node
(B) System variables/decisions set by VISTA Optimizer	
$mem_{storage}$	Size of Storage Memory
mem_{user}	Size of User Memory
mem_{dl}	DL Execution Memory
cpu	Number of cores assigned to a worker
n_p	Number of data partitions
join	Physical join implementation (*shuffle* or *broadcast*)
pers	Persistence format (*serialized* or *deserailized*)
(C) Other fixed (but adjustable) system parameters	
mem_{os_rsv}	Operating System Reserved Memory (default: 3 GB)
mem_{core}	Core Memory as per system specific best practice guidelines (e.g. Spark default: 2.4 GB)
p_{max}	Maximum size of data partition (default: 100 MB)
b_{max}	Maximum broadcast size (default: 100 MB)
cpu_{max}	Cap recommended for *cpu* (default: 8)
α	Fudge factor for size blowup of binary feature vectors as JVM objects (default: 2)

In the numerator, τ captures the total compute and communication costs, which are effectively "constant" for this optimization. The second term captures disk spill costs for T_i's. The denominator captures the degree of parallelism. While this objective is ideal, it is impractical

and needlessly complicated for our purposes due to three reasons. (1) Estimating τ is tedious, since it involves join costs, data loading costs, etc. (2) More importantly, we hit a point of diminishing returns with *cpu* quickly, since CNN inference typically dominates total runtime and DL systems like TF, anyway uses all cores regardless of *cpu*. That is, this workload's speedup against *cpu* will be quite sub-linear (confirmed by Figure 8.12(C)). Empirically, we find that about 7 cores typically suffice; interestingly, a similar observation is made in Spark guidelines for purely relational workloads [19, 21]. Thus, we cap *cpu* at $cpu_{max} = 8$. (3) Given the cap on *cpu*, we can just drop the term minimizing disk spill/cache miss costs, since s_{double} will typically be smaller than the total memory (even after accounting for the CNNs) due to the above cap.

Overall, our insights above yield a simpler objective that is still a reasonable surrogate for minimizing runtimes:

$$\max_{cpu, n_p, mem_{storage}} \quad cpu \tag{8.8}$$

The constraints for the optimization are as follows:

$$1 \leq cpu \leq \min\{cpu_{sys}, cpu_{max}\} - 1 \tag{8.9}$$

$$mem_{user} = \begin{cases} \text{(a) M is stored in PD User Memory:} \\ \quad \max\{|f|_{ser} + cpu \times \alpha \times \lceil s_{single}/n_p \rceil, \\ \quad\quad cpu \times |M|_{mem}\} \\ \text{(b) M is stored in DL Execution Memory:} \\ \quad |f|_{ser} + cpu \times \alpha \times \lceil s_{single}/n_p \rceil \end{cases} \tag{8.10}$$

$$mem_{dl} = \begin{cases} \text{(a) M is stored in PD User Memory:} \\ \quad cpu \times |f|_{mem} \\ \text{(b) M is stored in DL Execution Memory:} \\ \quad \max\{cpu \times |f|_{mem}, cpu \times |M|_{mem}\} \end{cases} \tag{8.11}$$

$$\begin{aligned} mem_{os\_rsv} + mem_{dl} + mem_{user} + mem_{core} \\ + mem_{storage} < mem_{sys} \end{aligned} \tag{8.12}$$

$$n_p = z \times cpu \times n_{nodes}, \text{ for some } z \in \mathbb{Z}^+ \tag{8.13}$$

$$\lceil s_{single}/n_p \rceil < p_{max} \tag{8.14}$$

If GPUs are available:

$$cpu \times \max\{|f|_{mem\_gpu}, |M|_{mem\_gpu}\} < mem_{GPU} \tag{8.15}$$

Equation 8.9 caps *cpu* and leaves a core for the OS. Equation 8.10 captures User Memory for reading CNN models for invoking the DL system, copying materialized feature layers from the DL system and memory needed for M–if M is stored in PD User Memory. As execution threads in a single worker have access to shared memory, the serialized CNN model need not be replicated. Equation 8.11 captures the maximum DL Execution Memory. $cpu \times |f|_{mem}$ is the CNN inference memory needed. If the downstream ML model is also a DL model, DL Execution Memory should also account for holding M. Equation 8.12 constrains the total memory as per Figure 8.4. If

there are GPUs, maximum GPU memory footprint $cpu \times \max\{|f|_{mem\_gpu}, |M|_{mem\_gpu}\}$ should be bounded by available GPU memory mem_{GPU} as per Equation 8.15. Equation 8.13 requires n_p to be a multiple of the number of worker processes to avoid skews, while Equation 8.14 bounds the size of an intermediate data partition as per system guidelines [1].

Optimizer Algorithm. Given our above observations, the algorithm is simple: linear search on *cpu* to satisfy all constraints.[3] Algorithm 7 presents it formally. If the **for** loop completes without returning, there is no feasible solution, i.e., System Memory is too small to satisfy some constraints, say, Equation 8.12. In this case, VISTA notifies the user, and the user can provision machines with more memory. Otherwise, we have the optimal solution. The other variables are set based on the constraints. We set *join* to *broadcast* if the predefined maximum broadcast data size constraint is satisfied; otherwise, we set it to *shuffle*. Finally, as per Section 8.3.2, *pers* is set to *serialized*, if disk spills/cache misses are likely (based on the newly set $mem_{storage}$). This is a bit conservative, since not all pairs of intermediate tables might spill, but empirically, we find that this conservatism does not affect runtimes significantly (more in Section 8.4). We leave more complex optimization criteria to future work.

8.4 Experimental Evaluation

We empirically validate if VISTA is able to improve efficiency and reliability of feature transfer workloads. We then drill into how it navigates the tradeoff space.

Datasets. We use two real-world public datasets: *Foods* [17] and *Amazon* [137]. *Foods* has about $20,000$ examples with 130 structured numeric features such as nutrition facts along with their feature interactions and an image of each food item. The target represents if the food is plant-based or not. *Amazon* is larger, with about $200,000$ examples with structured features such

[3] we explain our algorithm for the CPU-only scenario with an MLLib downstream model. It is straightforward to extend to the other settings.

Algorithm 7 The VISTA Optimizer Algorithm.

Inputs: see Table 8.1(A)
Outputs: see Table 8.1(B)
for $x = \min\{cpu_{sys}, cpu_{max}\} - 1$ **to** 1 **do**
  $totalcores \leftarrow x \times n_{nodes}$
  $n_p \leftarrow \lceil \frac{s_{single}}{p_{max} \times totalcores} \rceil \times totalcores$
  $mem_{worker} \leftarrow mem_{sys} - mem_{os\_rsv} - x \times |f|_{mem}$
  $mem_{user} \leftarrow \max\{|f|_{ser} + x \times \alpha \times \lceil s_{single}/n'_p \rceil, x \times |M|_{mem}\}$
  if $mem_{worker} - mem_{user} > mem_{core}$ **then**
    $cpu \leftarrow x$
    $mem_{storage} \leftarrow mem_{worker} - mem_{user} - mem_{core}$
    $join \leftarrow shuffle$
    if $|T_{str}| < b_{max}$ **then**
      $join \leftarrow broadcast$
    $pers \leftarrow deserialized$
    if $mem_{storage} < s_{double}$ **then**
      $pers \leftarrow serialized$
    return $(mem_{storage}, mem_{user}, cpu, n_p, join, pers)$
throw Exception(No feasible solution)

as price, title, and categories, as well as a product image. The target is the sales rank, which we binarize as a popular product or not. We pre-processed title strings to get 100 numeric features (an "embedding") using Doc2Vec [176]. We convert the indicator vector of categories to 100 numeric features using PCA. All images are resized to 227×227 resolution, as needed by popular CNNs. Overall, *Foods* is about 300 MB in size; *Amazon* is 3 GB. While these can fit on a single node, multi-node parallelism helps reduce completion times for long running ML workloads; also note that intermediate data sizes during feature transfer can be even 50x larger.

Workloads. We use three ImageNet-trained deep CNNs: AlexNet [171], VGG16 [260], and ResNet50 [136], obtained from TF model zoo [11]. They complement each other in terms of model size [83]. We select the following layers for feature transfer from each: *conv5* to *fc8* from AlexNet ($|L| = 4$); *fc6* to *fc8* from VGG ($|L| = 3$), and top 5 layers from ResNet (from its last two layer blocks [136]). Following standard practices [287, 25], we apply max pooling on convolutional feature layers to reduce their dimensionality before using them for M[4].

[4]Filter width and stride for max pooling are set to reduce the feature tensor to a 2×2 grid of the same depth.

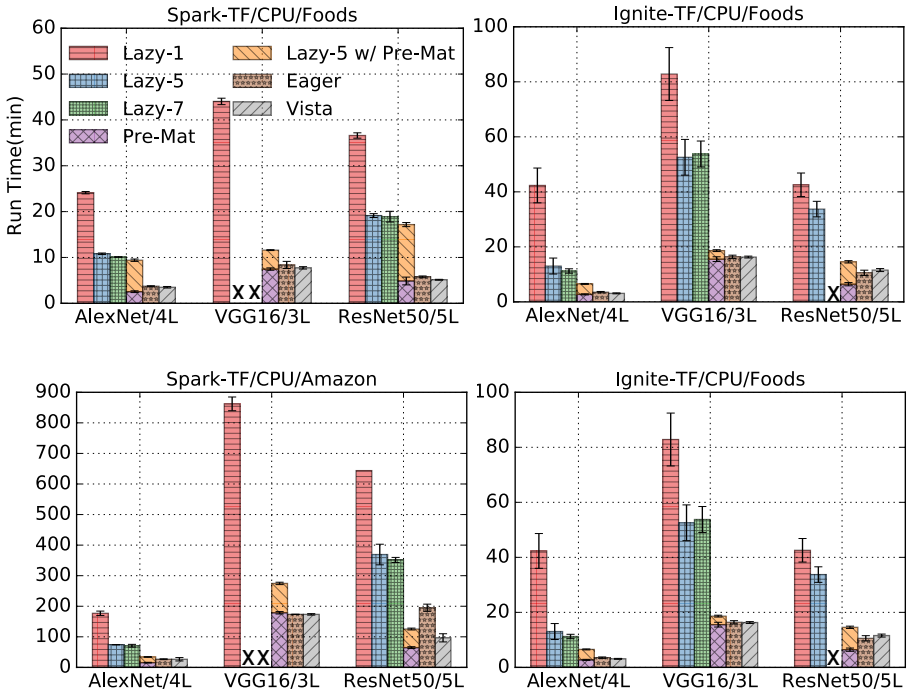


Figure 8.6: End-to-end reliability and efficiency. "×" means the workload crashed. Overall, VISTA offers the best or near-best runtimes and never crashes, while the alternatives are much slower or crash in some cases.

Experimental Setup. We use a cluster with 8 workers and 1 master in an OpenStack instance on CloudLab, a free and flexible cloud for research [111]. Each node has 32 GB RAM, Intel Xeon @ 2.00GHz CPU with 8 cores, and 300 GB Seagate Constellation ST91000640NS HDDs. All nodes run Ubuntu 16.04. We use Spark v2.2.0 with *TensorFrames* v0.2.9, TensorFlow v1.3.0, and Ignite v2.3.0. Spark runs in standalone mode. Each worker runs one Executor. HDFS replication factor is three; input data is ingested to HDFS and read from there. Ignite is configured with memory-only mode; each node runs one worker. All runtimes reported are the average of three runs with 90% confidence intervals. We chose these systems due to their popularity but the takeaways from our experiments are applicable to other DL systems (e.g., PyTorch [228]) and PD systems (e.g., Greenplum [18]) as well.

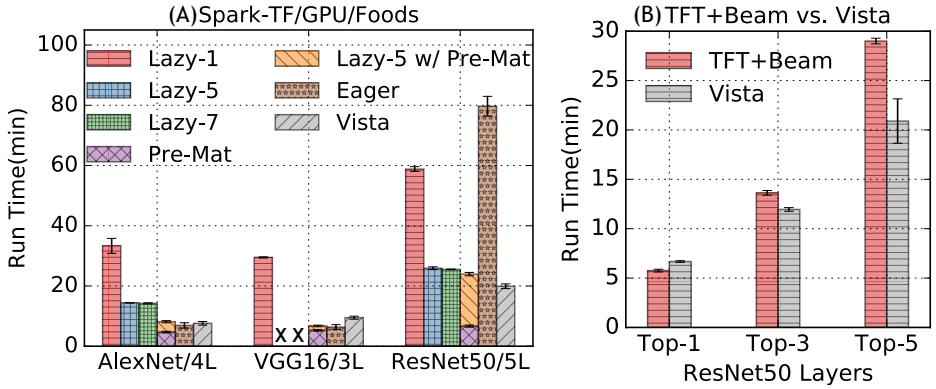


Figure 8.7: (A) End-to-end reliability and efficiency on GPU. "×" is a workload crash. (B) Comparing TFT+Beam vs. VISTA on *Foods*/CPU.

8.4.1 End-to-End Reliability and Efficiency

We compare VISTA with five baselines: three naive and two strong. *Lazy-1* (1 CPU per Executor), *Lazy-5* (5 CPUs), and *Lazy-7* (7 CPUs) capture the current dominant practice of layer at a time execution (Section 8.2). Spark is configured based on best practices [8, 19] (29 GB JVM heap, shuffle join, deserialized, and defaults for all other parameters, including n_p and memory apportioning). Ignite is configured with a 4 GB JVM heap, 25 GB off-heap Storage Memory, and n_p set to the default 1024. *Lazy-5 with Pre-mat* and *Eager* are strong baselines based on our tradeoff analyses in Section 8.3.2. In *Lazy-5 with Pre-mat*, the lowest layer specified (e.g., *conv5* for AlexNet) is materialized beforehand and used instead of raw images for all subsequent CNN inference; *Pre-mat* is time spent on pre-materializing the lowest layer specified. *Eager* is an alternative plan explained in Section 8.3.2; we use 5 CPUs per Executor. For *Lazy-5 with Pre-mat* and *Eager*, we explicitly apportion CNN Inference memory, Storage Memory, User Memory, and Core Memory to avoid workload crashes. Note that *Lazy-5 with Pre-mat* and *Eager* actually need parts of our code from VISTA. As for M, we run logistic regression for 10 iterations. Figure 8.6 presents the results.

Overall, VISTA improves reliability and/or efficiency across the board. On Spark-TF, *Lazy-5* and *Lazy-7* crash on both datasets for VGG16. On Ignite-TF, *Lazy-7* crashes for all CNNs on *Amazon*, while for ResNet50, *Lazy-7* on *Foods* also crashes. These are due to memory

related crash scenarios explained in Section 8.3.1. On Ignite-TF, *Eager* on *Amazon* also crashes for ResNet50 due to intermediate data exhausting the total available system memory. When *Eager* does not crash and the intermediate data fits in memory, its efficiency is comparable to VISTA, which validates our analysis in Section 8.3.2. However, when the size of the intermediate data does not fit in memory, as with *Amazon* on Spark for ResNet50, *Eager* incurs significant overheads due to costly disk spills. *Lazy-5 with Pre-mat* does not crash, but its runtimes are comparable to *Lazy-5* and mostly higher than VISTA. This is because the layers of AlexNet and ResNet are much larger than the images, which raises data I/O and join costs.

More careful tuning could avoid the crashes with *Lazy*. But that forces ML users to waste time wrestling with low-level systems issues–time they can now spend on further ML analysis. Compared to *Lazy-7*, VISTA is 62%–72% faster; compared to *Lazy-1*, 58%–92%. These gains arise because VISTA removes redundancy in partial CNN inference and reduces disk spills. Of course, the exact gains depend on the CNN and L: if more of the higher layers are tried, the more redundancy there is and the faster VISTA will be.

Experiments on a GPU. We ran GPU experiments on Spark-TensorFlow environment using the *Foods* dataset. Experimental setup is a single node machine with 32 GB RAM, Intel i7-6700 @ 3.40GHz CPU with 8 cores, 1 TB Seagate ST1000DM010-2EP1 SSD, and Nvidia Titan X (Pascal) 12GB GPU. Figure 8.7 presents the results. In this setup *Lazy-5* and *Lazy-7* crash with VGG16. For ResNet50, *Eager* takes significantly more time to complete compared to VISTA due to costly disk spills. Overall, the experimental results on both CPU and GPU settings confirm the benefits of an automatic optimizer such as ours for improving reliability and efficiency, which could reduce both user frustration and costs.

Comparing against TF Transform+Beam. TensorFlow Transform (TFT) [9] is a library for pre-processing input data for TF. It can wrap CNN models as pre-processing functions and be run on Apache Beam at scale to generate ML-ready features in TFRecord format. So, TFT is akin

to the role of TensorFrames in VISTA. We compare TFT+Beam against VISTA on *Foods*/ResNet50 with varying number of layers explored. For TFT+Beam, we first join the structured data with images and then extract and write out features for all the layers in one go (similar to our *Eager* plan). We then train a 3-layer MLP (each hidden layer has 1024 units) for 10 iterations using distributed TF/Horovod. We use Apache Flink (v1.9.2) as the backend runtime for Beam. For Flink, through trial and error, we chose a working configuration with a parallelism of 32, JVM heap of 25GB, and User Memory fraction of 60% (default 30%). Increasing the parallelism, reducing the heap size, and/or User Memory fraction resulted in various memory-related crashes. For VISTA we use Spark backend and use TF/Horovod to train the same downstream MLP. Figure 8.7 (B) presents the results.

When exploring only the last layer, TFT+Beam is slightly faster than VISTA. However, when exploring more layers, VISTA starts to clearly outperform TFT+Beam. This is because extracting all the layers in one go puts significant memory pressure on Flink, causing costly disk spills. VISTA's staged materialization plan keeps the memory pressure to a minimum. The fact that we had to manually figure out a working configuration for Flink to run this workload underscores the importance of automatically tuning these parameters without any user intervention. It also shows that the trade-offs discussed in Section 8.3.1 are generally applicable when integrating DL and PD systems.

8.4.2 Accuracy

All approaches in Figure 8.6 (including VISTA) yield identical downstream models (and thus, same accuracy) for a given CNN layer. For both *Foods* and a sample of *Amazon* (20,000 records) datasets we evaluate the downstream logistic regression model test F1 score with (1) only using structured features, (2) structured features combined with "Histogram of Oriented Gradients (HOG)" [103] based image features, and (3) structured features combined with CNN based image features from different layers of AlexNet and ResNet. Figure 8.8 presents the results.

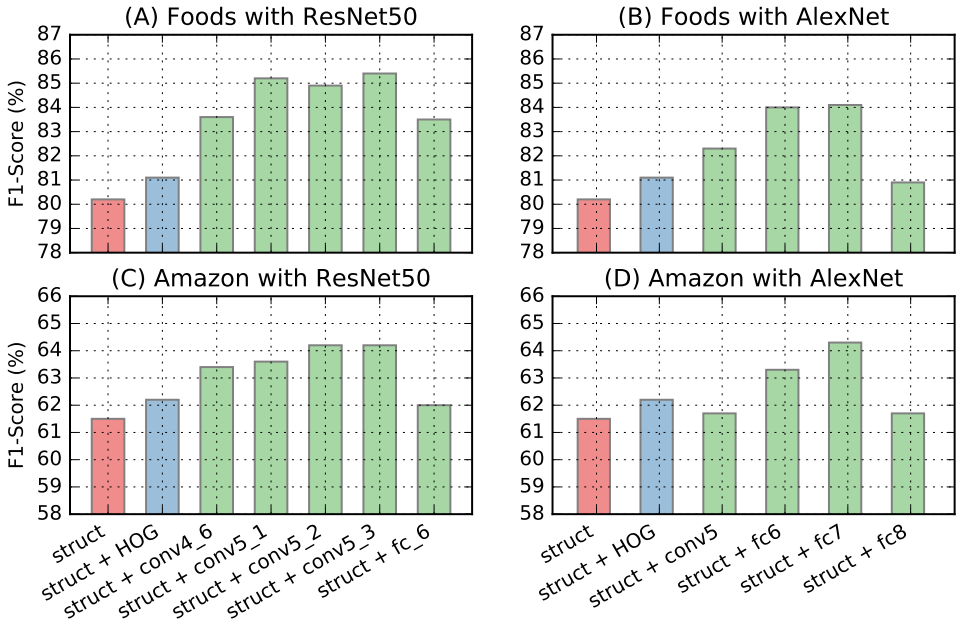


Figure 8.8: Test F1 scores for various sets of features for training a logistic regression model with elastic net regularization with $\alpha = 0.5$ and a regularization value of 0.01.

In all cases incorporating image features improves the classification accuracy, and CNN features offer significantly higher lift in accuracy than traditional HOG features. We saw F1 score lifts of 3% to 5% for the logistic regression model with feature transfer on test datasets (20% of the data). As expected, the lift varies across CNNs and layers. For instance, on *Foods*, structured features alone give 80.2% F1 score. Adding ResNet50's *conv-5-3* layer raises it to 85.4%, a large lift in ML terms. But using the last layer *fc-6* gives only 83.5%. *Amazon* exhibited similar trends. These results reaffirm the need to try multiple layers and thus, the need for a system such as VISTA to simplify and speed up this process. We also tried a decision tree as the downstream ML model. For *Foods* it yielded a test F1 score of 88.5% and for *Amazon* an F1 score of 61.4%. However, in both cases incorporating CNN features didn't improve the accuracy significantly. We believe this is because the depths of the conventional decision tree models are not large enough to reap the benefits of CNN features. We leave the analysis on the suitability of different ML models for CNN feature transfer for future work as it is orthogonal to our work.

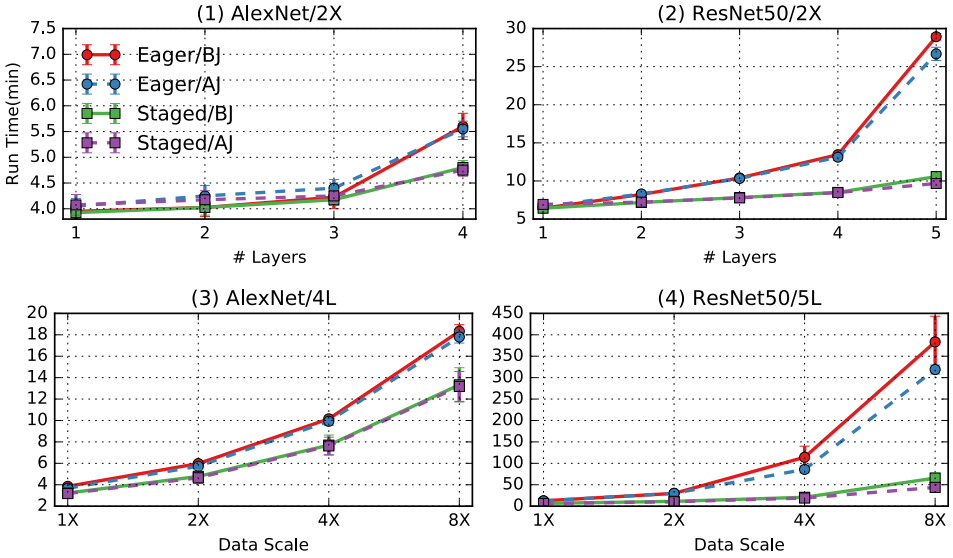


Figure 8.9: Runtimes of logical execution plan alternatives for varying data scale and number of feature layers explored.

8.4.3 Drill-Down Analysis of Tradeoffs

We now analyze how VISTA navigates the tradeoffs explained in Section 8.3 using Spark-TF prototype of VISTA. We use the less resource-intensive *Foods* dataset but alter it semi-synthetically for some experiments to study VISTA runtimes in new operating regimes. In particular, when specified, vary the data scale by replicating records (say, "4X") or varying the number of structured features (with random values). For uniformity sake, unless specified otherwise, we use all 8 workers, fix *cpu* to 4, and fix Core Memory to 60% of JVM heap. Other parameters are set by Algorithm 7. The layers explored for each CNN are the same as before.

Logical Execution Plan Decisions. We compare four combinations: *Eager* or *Staged* combined with inference *After Join (AJ)* or *Before Join (BJ)*. We vary both $|L|$ (dropping lower layers) and data scale for AlexNet and ResNet. Figure 8.9 shows the results. The runtime differences between all plans are insignificant for low data scales or low $|L|$ on both CNNs. But as $|L|$ or the data scale goes up, both *Eager* plans get much slower, especially for ResNet (Figure 8.9(2,4)); this is due to disk spills of large intermediate data. Across the board, *AJ*

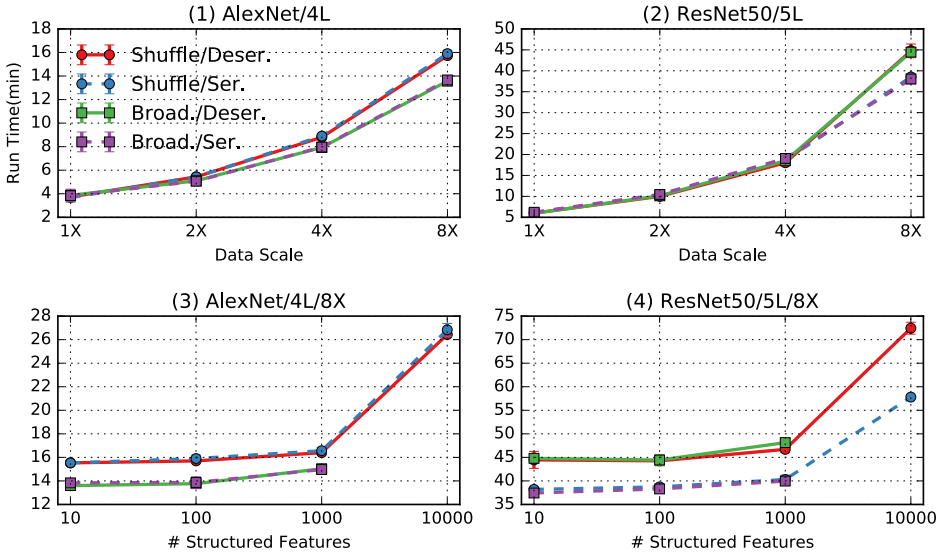


Figure 8.10: Runtimes of physical plan choices for varying data scale and number of structured features.

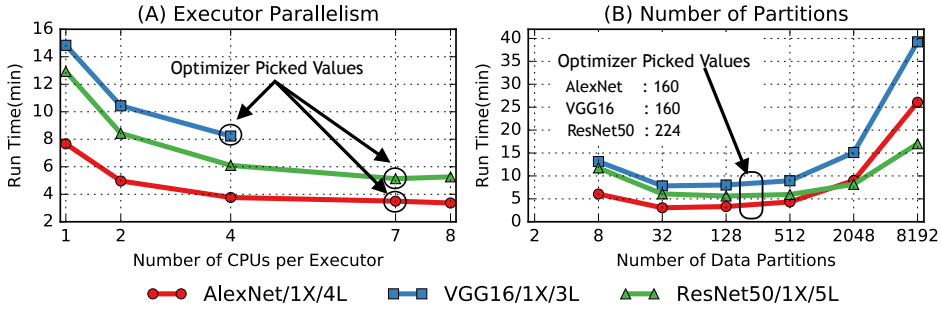


Figure 8.11: Varying system configuration parameters. Logical and physical plan choices are fixed to *Staged/AJ* and *Shuffle/Deser.*.

plans are mostly comparable to their *BJ* counterparts but marginally faster at larger scales. The takeaway is that these results validate our choice of using only *Staged/AJ* in VISTA, viz., Plan (E) in Figure 8.5.

Physical Plan Decisions. We compare four combinations: *Shuffle* or *Broadcast* join and *Serialized (Ser.)* or *Deserialized (Deser.)* persistence format. We vary both data scale and number of structured features ($|X_{str}|$) for both AlexNet and ResNet. The logical plan used is *Staged/AJ*. Figure 8.10 shows the results. On ResNet, all four plans are almost indistinguishable regardless of the data scale (Figure 8.10(2)), except at the 8X scale, when the *Ser.* plans slightly

outperform the *Deser.* plans. On AlexNet, the *Broadcast* plans slightly outperform the *Shuffle* plans (Figure 8.10(1)). Figure 8.10(3) shows that this gap remains as $|X_{str}|$ increases but the *Broadcast* plans crash eventually. On ResNet, however, Figure 8.10(4) shows that both *Ser.* plans are slightly faster than their *Deser.* counterparts but the *Broadcast* plans still crash eventually. The takeaway is that no one combination is always dominant, validating the utility of an automated optimizer like ours to make these decisions.

Optimizer Correctness. We vary *cpu* and n_p while explicitly apportioning the memory regions based on the chosen *cpu* value. We pick *Staged/ AJ/Shuffle/Deser.* as the logical-physical plan combination. Figures 8.11(A,B) show the results for all CNNs. As explained in Section 8.3.3, the runtime decreases with *cpu* for all CNNs, but VGG eventually crashes (beyond 4 cores) due to the blowup in CNN Inference Memory. The runtime decrease with *cpu* is sub-linear though. To drill into this issue, we plot the speedup against *cpu* on 1 node for data scale 0.25X (to avoid disk spills). Figure 8.12(C) shows the results: the speedups plateau at 4 cores. As mentioned in Section 8.3.3, this is as expected, since CNN inference dominates total runtimes and TF always uses all cores regardless of *cpu*. Overall, we can see that the VISTA optimizer (Algorithm 7) picks either optimal or near-optimal *cpu* values; AlexNet: 7, VGG16: 4, and ResNet50: 7.

Figure 8.11(B) shows non-monotonic behaviors with n_p. At low n_p, Spark crashes due to insufficient Core Memory for the join. As n_p goes up, runtimes go down, since Spark uses more parallelism (up to 32 cores). Eventually, runtimes rise again due to Spark overheads for running too many tasks. In fact, when $n_p > 2000$, Spark compresses task status messages, leading to high overhead. The VISTA optimizer (Algorithm 7) sets n_p at 160, 160, and 224 for AlexNet, VGG, and ResNet respectively, which yield close to the fastest runtimes. The takeaway is that these settings involve non-trivial CNN-specific efficiency tradeoffs and thus, an automated optimizer like ours can free ML users from such tedious tuning.

Scalability. We evaluate the scaleup (weak scaling) and speedup (strong scaling) of the logical-physical plan combination of *Staged/After Join/Shuffle/Deserialized* for varying number

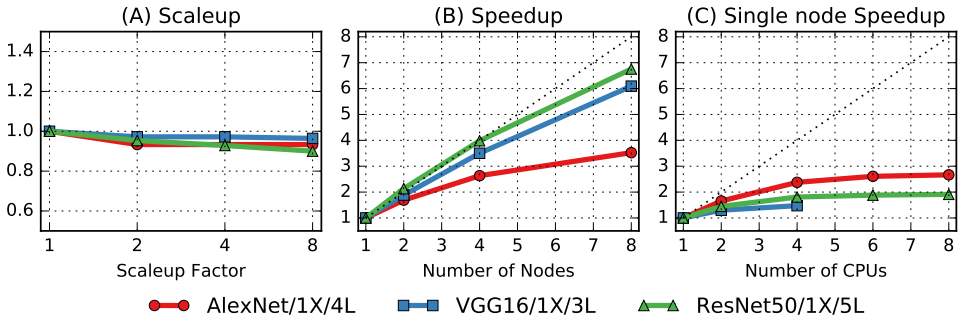


Figure 8.12: (A,B) Scaleup and speedup on cluster. (C) Speedup for varying *cpu* on one node with 0.25x data. Logical and physical plan choices are fixed to *Staged/AJ* and *Shuffle/Deser.*.

of worker nodes (and also data scale for scaleup). While CNN inference and M are embarassingly parallel, data reads from HDFS and the join can bottleneck scalability. Figures 8.12 (A,B) show the results. We see near-linear scaleup for all 3 CNNs. But Figure 8.12 (B) shows that the AlexNet sees a markedly sub-linear speedup, while VGG and ResNet exhibit near-linear speedups. To explain this gap, we drilled into the Spark logs and obtained the time breakdown for data reads and CNN inference coupled with the first iteration of logistic regression for each layer. For all 3 CNNs, data reads exhibit sub-linear speedups due to the notorious "small files" problem of HDFS with the images [22]. But for AlexNet in particular, even the second part is sub-linear, since its absolute compute time is much lower than that of VGG or ResNet. Thus, Spark overheads become non-trivial in AlexNet's case.

Summary of Results. VISTA reduces runtimes (even up to 10x) and avoids memory-related crashes by automatically handling the tradeoffs of logical execution plan, system configuration, and physical plan. Our new *Staged* execution plan offers both high efficiency and reliability. CNN-aware system configuration for memory apportioning, data partitioning, and parallelism is critical. Broadcast join marginally outperforms shuffle join but crashes at larger scales. Serialized disk spills are marginally faster than deserialized. Overall, VISTA automatically optimizes such complex tradeoffs, freeing ML users to focus on their ML exploration.

8.5 Conclusion

The success of deep CNNs presents new opportunities for exploiting images and other unstructured data in data-driven applications that have hitherto relied mainly on structured data. But realizing the full potential of this integration requires data analytics systems to evolve and elevate CNNs as first-class citizens for query processing, optimization, and system resource management. We take a first step in this direction by integrating parallel dataflow and DL systems to support and optimize a key emerging workload in this context: feature transfer from deep CNNs. By enabling more declarative specification and by formalizing partial CNN inference, VISTA automates much of the data management and systems-oriented complexity of this workload and enables automated optimizations that improve system reliability and efficiency.

Chapter 8 contains material from "Vista: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale" by Supun Nakandala and Arun Kumar, which appears in Proceedings of 2020 ACM SIGMOD International Conference on Management of Data. The dissertation author was the primary investigator and author of this paper. The code for our system is open source and is available on GitHub: https://github.com/AdaLabUCSD/Vista.

Chapter 9

NAUTILUS: Query Optimizations for DL Model Adaptation

9.1 Introduction

In this chapter, we dive deeper into our techniques for optimizing DL model adaptation workloads. DL model adaptation is an effective technique to mitigate the high training data and computation requirements of DL model building workloads. With model adaptation, one adapts the parameters of a pre-trained model instead of training a new model from scratch. Model adaptation is a form of transfer learning. In this chapter, we refer to model adaptation workloads simply as deep transfer learning (DTL) workloads.

Example Use Case: Consider a data scientist tasked to develop a named entity recognition model to identify disease entities from clinical text. She is provided a large unlabeled clinical text dataset. For this task, she decides to adopt the DTL paradigm. She downloads a pre-trained model (e.g., BERT [109]) from a model hub [284], removes the last few layers in the model, and adds a few new layers on top of it. She also *freezes* most of the pre-trained layers and trains only the new layers and the final few layers of the pre-trained model. She explores several freezing

schemes and training hyperparameter values (e.g., learning rates) to find the model with the best accuracy.

DTL leverages the fact that most of the features learned by a DL model when trained on a large dataset like Wikipedia are general enough to be reused in other similar settings. While both pre-trained and newly added layer parameters can be trained, one can freeze the parameters from most pre-trained layers as they are generic enough to be directly reused [245, 181, 229, 109]. This approach also reduces the chances of model overfitting [245, 200] and the compute costs as computations needed to update the frozen layers can be avoided. However, different freezing schemes and training hyperparameters can lead to different model accuracies. Hence, model selection is unavoidable for DTL. Overall, DTL significantly reduces the labeled data requirements (e.g., from millions to few thousand).

Even though DTL significantly reduces the large training data requirement, it does not eliminate it. Often, practitioners create training datasets by manually labeling the data. We also observed that in many domain applications, data labeling is seldom a one-off process [65, 73, 205, 254]. Practitioners often update the training dataset intermittently by labeling new data and evaluate the model accuracy to ensure that they have labeled a sufficient amount of data to train a model that meets their target accuracy.

Example Use Case (Continued): To create the labeled dataset for training, our data scientist adopts the *active learning paradigm* (AL) [205, 254]. AL operates in cycles. Each cycle, she labels a new batch of data and trains the model on all the labeled data up to that cycle. The model trained in the current cycle is used to sample the most informative data for the next cycle using some sampling technique (e.g., uncertainty, diversity [254]). Figure 9.1 (A, B) presents an illustration of her workflow.

AL is an emerging paradigm that focuses on reducing data labeling efforts and it too requires periodic model selection. However, AL is not the only paradigm that requires periodic

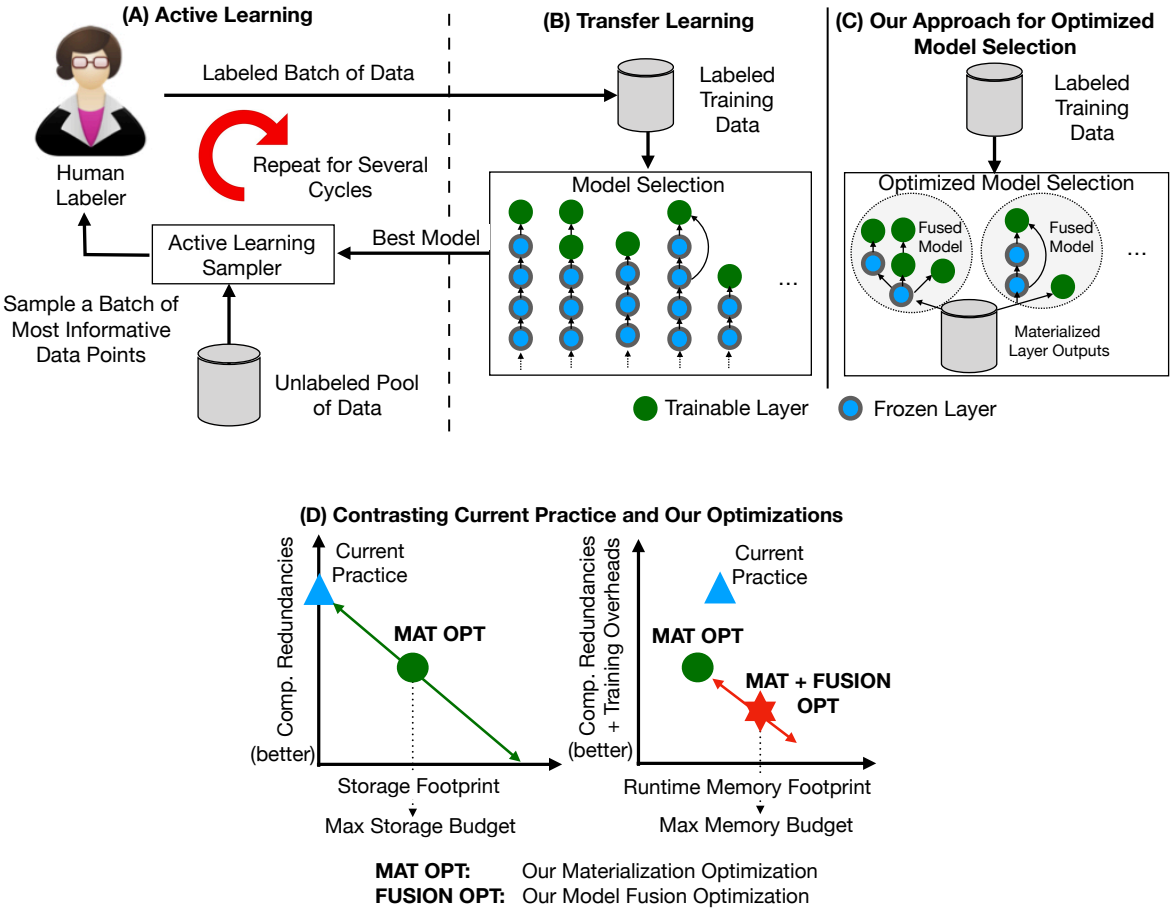


Figure 9.1: (A) Human labeler labels batches of most informative data. (B) A pre-trained model is adapted for a target task. (C) Our approach for optimized DTL model selection performs materialization and model fusion optimizations. (D) Contrasting the current practice and our approach on different trade-off spaces.

model selection during data labeling. For example, other popular data labeling approaches–such as simple manual annotation, crowd workers (e.g., SageMaker Ground Truth [253]), and programmatic supervision [238]–may also require periodic model selection to evaluate the benefit of labeling more data on model accuracy.

9.1.1 Current Practice and Inefficiencies

Today, one executes the DTL workload by training a DL model with frozen layers as it is and repeats the process for all model selection cycles [97, 205]. This leads to incurring redundant computations in frozen layers as they are repeatedly invoked with the same inputs to generate the same output. We identify three types of redundancies:

- **Redundancies across training epochs**: DL model training is iterative. Each iteration, also called an epoch, reads the full dataset and feeds it through the layers. This leads to redundant computations across training epochs.

- **Redundancies across models**: Practitioners have to perform model selection where they explore several different layer freezing schemes and training hyperparameters (e.g., learning rate, batch size). Thus, a model selection workload can contain models that share frozen layers. Independently training them leads to redundant computations across models.

- **Redundancies across model selection cycles**: Model selection is repeated for every new snapshot of training data. This leads to redundant computations across model selection cycles.

Overall, these redundancies are problematic, at least for three main reasons. First, they increase the model selection runtimes and impede human productivity. Human labelers may have to wait longer until model selection completes to proceed to the next data labeling cycle. Second, they lead to higher monetary costs, especially in pay-as-you-go environments in the cloud. Third, they also lead to significantly higher energy consumption and associated environmental issues, which are expected to further amplified by the wide adoption of DL in many domains [68].

9.1.2 Our Proposed Approach

In this work, we use a database-inspired lens to formalize, optimize, and accelerate the DTL model selection in the presence of frozen layers. We reimagine DTL model selection as a novel instance of *multi-query optimization* (MQO) [250] and perform two data management-inspired optimizations:

- **Materialization Optimization:** We materialize intermediate layer outputs from a chosen set of frozen layers on the first time they are computed and avoid repeated recomputations. However, the size of the intermediate layer outputs can be orders of magnitude (even up to 100X) larger than the input data, and it may not be possible to materialize all frozen layers due to storage constraints [126]. Even if it is possible to materialize all frozen layers, it may be the case that some outputs can be computed faster using others instead of loading them. Therefore, the challenge is to pick an optimal set of frozen layers that can reduce model selection runtimes subject to a storage budget as shown in Figure 9.1 (D). This optimization is an instance of view selection being combined with MQO to optimize DL workloads [96]. By doing so, we reduce all three types of computational redundancies.

- **Model Fusion Optimization:** Even after the materialization optimization, there can be remaining frozen layers shared among models in the workload. Thus, we propose model fusion optimization, which is inspired by pipelined multi-query execution in relational query processing [104]. It builds on top of our materialization optimization and reduces the redundant computations by fusing multiple models and eliminating frozen common sub-expressions in the models. It also amortizes model training overheads and I/O overheads [203]. However, excessive model fusion can increase the runtime memory footprint and cause workload crashes. The challenge is to pick an optimal set of models to be fused, such that it reduces model selection runtimes subject to a runtime memory budget as shown in Figure 9.1 (D).

Our optimization techniques are orthogonal to the data labeling scheme used. Thus, we can support all kinds of data labeling schemes–such as active learning, simple manual labeling, crowd workers, and programmatic supervision–in a unified manner.

We implement our techniques into a system we call NAUTILUS. It runs on top of the popular DL libraries Keras and TensorFlow [54]. NAUTILUS is tailored to limited-resource settings such as workstations and personal computers, which cover a vast majority of DTL use cases [205]. NAUTILUS provides easy-to-use APIs to specify the DTL workload over evolving training data and optimizes DTL model selection. We evaluate NAUTILUS empirically on five workloads, including one from an influential NLP publication [109], on two benchmark ML datasets: CoNLL [271] and Malaria [235]. NAUTILUS avoids many compute redundancies and significantly reduces training and I/O overheads, enabling up to 5X reductions in model selection runtimes. Thus, NAUTILUS significantly reduces system resource costs and also improves human productivity (i.e., less waiting time between model selection cycles) for DTL workloads. Overall, this work makes the following contributions:

- To the best of our knowledge, this is the first work to formalize and optimize deep transfer learning (DTL) workloads over evolving training data from a data management standpoint.

- We reimagine iterative training of DL models with frozen layers as a new instance of MQO and present a materialization optimization technique to reduce redundant computations of DTL workloads.

- We present model fusion optimization, which builds on top of our materialization optimization, to further reduce redundant computations and model training overheads of DTL workloads.

- We implement our ideas into a data system called NAUTILUS and perform an extensive empirical evaluation using 5 end-to-end workloads on two benchmark ML datasets. NAU-

Table 9.1: Notation used in Section 9.2

Symbol	Description
$M = (L, E)$	A DL model with L layers and E edges.
$f(l)$	$f(l) = True/False$. Function indicating layer l is frozen during model training.
$m(l)$	$m(l) = True/False$. Function indicating layer l is materializable.
ϕ	Set of training hyperparameters.
Q	$Q = \{(M_1, \phi_i), \ldots, (M_n, \phi_n)\}$. Set of model and training hyperparameter pairs.
D	Labeled dataset. D_k corresponds to the dataset snapshot at time k. D_k^{train} and D_k^{valid} are training and validation splits of D_k, respectively.
$g(M, \phi, D)$	$g(M, \phi, D)$. Training function that takes in a M, ϕ, and D. After training M using ϕ on D^{train}, returns the model accuracy on D^{valid}.
ΔL^+,ΔL^-	Added (ΔL^+) or removed (ΔL^-) layers.
ΔE^+,ΔE^-	Added (ΔE^+) or removed (ΔE^-) edges.
ΔD_k^+	New labeled data for the model selection cycle k.

TILUS reduces DTL model selection runtimes by up to 80% in some cases.

Outline: Section 9.2 formalizes the workload. Section 9.3 provides an overview of NAUTILUS and implementation details. Section 9.4 dives into the system optimizations. Section 9.5 presents the experiments and we conclude in Section 9.6.

9.2 Preliminaries

We present the formal problem description. The notation used is explained in Table 9.1.

9.2.1 Definitions and Data Model

We start by defining some terms and notation to formalize the DTL workload. We will use these terms in the rest of this chapter.

Definition 9.2.1 *A **layer** is a function l that takes a list of input tensors $t_1, t_2, \ldots, t_m$ ($m \geq 1$) of fixed shape and outputs a tensor $t' = l(t_1, t_2, \ldots, t_m)$ of potentially different, but fixed shape. A list of tensors $t_1, t_2, \ldots, t_m$ are said to be shape-compatible with l iff their shapes conform to what l expects for its inputs.*

Definition 9.2.2 *A **model** $M = (L, E)$ is a directed acyclic graph (DAG) of layers $L = l_1, \ldots, l_n$ and edges E between the layers.*

Definition 9.2.3 *A layer l is **frozen** if its learnable parameters are not updated during training. A layer with no learnable parameters is also frozen. $f(l)$ is a function that indicates a layer l is frozen or not.*

Frozen layers incur redundant computations. However, a frozen layer that has a non-frozen ancestor doesn't incur redundant computations. Thus, we introduce the notion of a *materializable* layer to identify layers that will contribute to redundant computations.

Definition 9.2.4 *A layer l is **materializable** if it's a model input layer (i.e., $l \in I$) or is a frozen layer with all its parent layers being materializable. $m(l)$ is a function that indicates a layer l is materializable or not.*

9.2.2 Workload Formalization

We are given a candidate set of model and training hyperparameter pairs $Q = \{(M_i, \phi_i) : i \in 1 \ldots n\}$. Each candidate model M_i is adapted from a source pre-trained model $M_{src} = (L_{src}, E_{src})$ and its pre-trained layers are frozen according to some scheme. We assume that we have access to

a model training function that trains a candidate model M_i using hyperparameters ϕ_i on a training split of D_k^{train} and returns validation accuracy on a validation split D_k^{valid}. We represent this model training function as $g(M_i, \phi_i, D_k)$. We then perform model selection to find the best candidate model based on validation accuracy and repeat it whenever the dataset snapshot changes from D_k to D_{k+1}. More precisely, we describe the workload as follows:

$\forall D_k \in \{D_0, D_1, \dots\}$:

$$\underset{(M_i,\phi_i)\in Q}{\arg\max}\, g(M_i, \phi_i, D_k) \tag{9.1}$$

$$\begin{aligned} M_i &= (L_i, E_i) \\ L_i = \Delta L^+ \cup (L_{src} - \Delta L^-),\ & E_i = \Delta E^+ \cup (E_{src} - \Delta E^-) \end{aligned} \tag{9.2}$$

$$\begin{aligned} \Delta E^+ \subseteq (\Delta L^+ \times \Delta L^+) \cup (L_{src} \times \Delta L^+) \\ \cup (\Delta L^+ \times L_{src}) \cup (L_{src} \times L_{src}) \end{aligned} \tag{9.3}$$

$$D_{k+1} = D_k \cup \Delta D_k^+ \tag{9.4}$$

Equation 9.1 captures the model selection step. Equation 9.2 captures the structure of a candidate model M_i, which is obtained by adding a new set of layers ΔL^+ and edges ΔE^+ to M_{src} and removing a set of existing layers $\Delta L^- (\subset L_{src})$ and edges $\Delta E^- (\subseteq E_{src})$ from M_{src}, while ensuring the DAG structure and the shape compatibility of all layer inputs. Equation 9.3 captures the structure of ΔE^+, which has four different types of edges based on the originating and terminating layer type. Finally, Equation 9.4 captures how the next labeled data snapshot

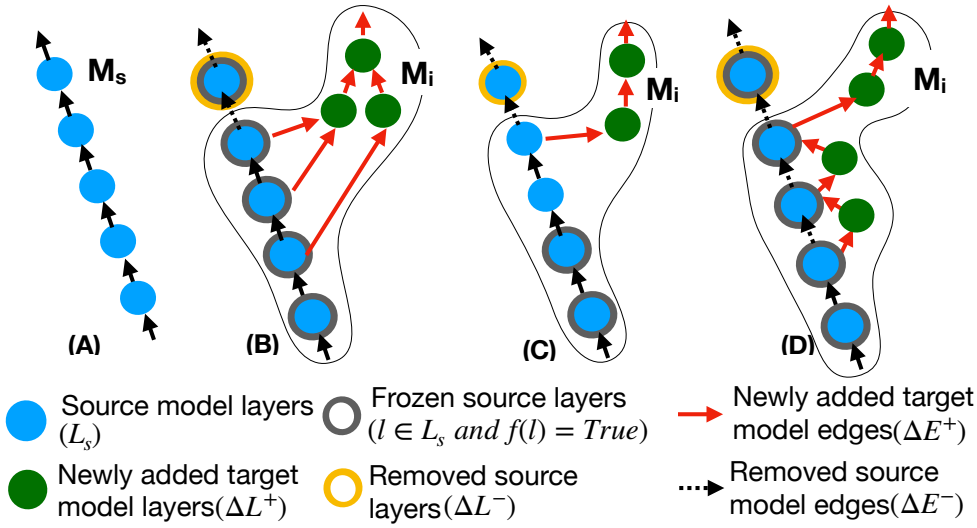


Figure 9.2: Model adaptation schemes. (A) Source model M_S. (B) Feature transfer. (C) Fine-tuning. (D) Adapter training.

D^{k+1} is obtained by adding a new set of labeled data records ΔD_k^+ to the current snapshot D_k.

9.2.3 Popular Model Adaptation Schemes

We found three model adaptation schemes that are popular among practitioners: features transfer, adapter training, and fine-tuning. They can be treated as special cases of our general workload formalization that impose specific structural properties on the newly added edges ΔE^+ and the layer freezing scheme.

- **Feature Transfer:** In this scheme, one freezes all layers in L_{src} (i.e., $f(l) = True$, $\forall l \in L_{src}$) and restricts ΔE^+ to only contain edges between newly added layers or edges from a source model layer to a newly added layer (i.e., $\Delta E^+ \subseteq (\Delta L^+ \times \Delta L^+) \cup (L_{src} \times \Delta L^+)$). Common practice is to add new layers on top of the penultimate layer, any other top-level layer, or a collection of top-level layers in M_{src} [109]. The structure of an example M_i that uses the feature transfer scheme is shown in Figure 9.2 (B).

- **Fine-Tuning:** This scheme is similar to feature transfer but there is at least one pre-

trained layer l in the adapted model that is unfrozen (i.e., $f(l) = False$). Parameters of all such layers along with the parameters of the newly added layers ΔL^+ are learned during training [109]. While one can unfreeze all layers in M_{src}, researchers have shown that freezing most of the lower-level layers in M_{src} and fine-tuning only the top few layers can achieve similar results to fine-tuning all layers [181]. This also avoids the risk of pre-trained information in M_{src} getting overwritten due to overfitting, which can easily happen in transfer learning settings with limited training data [245, 200]. The structure of an example M_i which uses the fine-tuning scheme is shown in Figure 9.2 (C).

- **Adapter Training:** In this scheme, one freezes most of the layers in L_{src}, but not necessarily all (i.e., $f(l) = True,\ \exists l \in L_{src}$). However, ΔE^+ can be more general with having edges between newly added layers, edges going from source model layers to newly added layers, and also edges going from newly added layers to source model layers (i.e., $E^+ \cap (L^+ \times L_{src}) \neq \phi$). The common practice is to add small bottleneck layers called *adapters* between the layers of M_{src} [239, 230, 143]. While one can add adapters to all layers in M_{src}, researchers have shown that adding adapters only to the top-level layers can be as effective as adding adapters to all the layers [244]. The structure of an example M_i which uses the adapter training scheme is shown in Figure 9.2 (D).

Model Selection for Popular Approaches: Model selection is unavoidable for any DL model training as one has to tune the training hyperparameters like batch size, regularization, learning rate, and number of training epochs. In addition to the above common training hyperparameters, popular transfer learning schemes also need additional architectural tuning. For example, feature transfer needs exploring features from several layers or layer combinations. Fine-tuning needs exploring different layer freezing schemes (e.g., up to which layer to freeze?). Adapter training also needs exploring different adapters and adapter placement schemes (e.g., to which layers to add adapters?). It has been shown that all these schemes can be equally

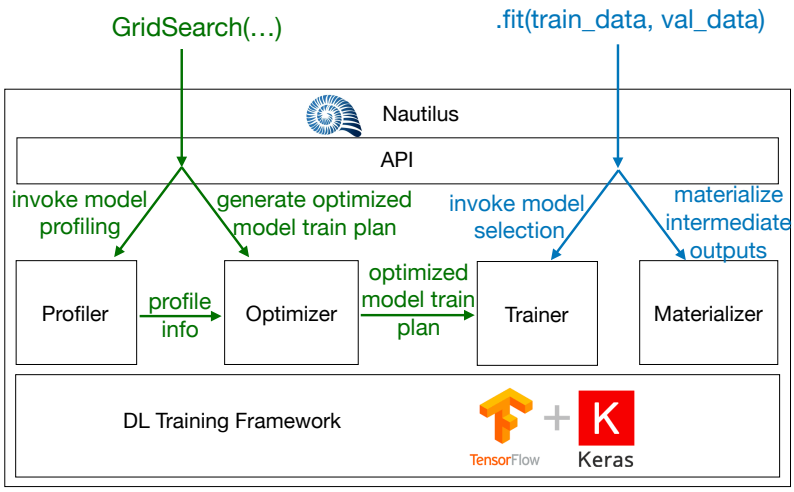


Figure 9.3: High-level architecture of NAUTILUS and the interactions between system components. fit(...) method is called for every model selection cycle.

competitive for a wide variety of transfer learning tasks [229, 245]. Thus, practitioners need to explore multiple schemes before picking the best one.

9.3 System Overview

We implement NAUTILUS on top of TensorFlow and Keras libraries. NAUTILUS optimizes DTL model selection. It has 5 main components: API, Profiler, Optimizer, Materializer, and Trainer. Figure 9.3 presents the architecture of NAUTILUS. Next, we provide details on NAUTILUS's components.

- **API:** NAUTILUS's API is inspired by libraries like Scikit-Learn and Keras. Users create a model selection object by specifying a parameter search space and a user-defined model initialization function. The model initialization function encapsulates the logic to interpret the search parameter values. It takes in an instance of parameter values ϕ_i and returns a Keras model ready to be trained. Thus, NAUTILUS can support both architectural tuning parameters (e.g., which layers to add, prune, or freeze) and training hyperparameters (e.g., learning rate) in a unified manner. Users can also override the default system config

values used by the optimizer. These include storage and runtime memory budget, expected maximum number of training records, disk throughput, and compute throughput values. Users initiate model selection by calling the *fit(...)* method and passing a batch of training and validation data. It is called for every model selection cycle.

- **Profiler:** When a user initializes a workload, NAUTILUS internally invokes its Profiler. Profiler invokes the user-defined model initialization function to initialize all models, profiles them, and finally stores the initialized model checkpoints on disk. Model checkpoints are artifacts consisting of model architecture, weights, and the optimizer. They capture all the details that the DL framework needs to train the model. For profiling, it uses the features available in TensorFlow. The profiling information includes shapes of all intermediate output tensors and the forward-pass layer compute costs in FLOPs.

- **Optimizer:** The optimizer takes in the profiling information and system configuration values and generates an optimized model training plan. The optimized plan combines both our materialization and model fusion optimizations. Due to the model fusion optimization, an optimized plan can correspond to more than one model in the original model selection workload. It then generates the model checkpoints for the optimized model training plan by reading the original model checkpoints and stores the new model checkpoints on disk. It also creates a model checkpoint that is used to generate the outputs of the chosen materialized layers. We discuss system optimizations in more detail in Section 9.4.

- **Materializer:** When a user initiates a model selection step by passing a new batch of labeled data, the API calls the Materializer to update both the labeled dataset and outputs of chosen materialized layers. The Materializer reads the output materialization model checkpoint, generates the intermediate outputs, and stores them on disk. One could also store the outputs in DRAM. However, their size can be significant (e.g., 10s of GBs) and

can exhaust DRAM. Also, DL models are often compute-intensive, and I/O overheads can be mitigated by prefetching. Thus, if there is excess DRAM available, we rely on the OS disk cache to cache the intermediate outputs.

- **Trainer:** The Trainer trains the models on the labeled training dataset according to the optimizer-generated training plan and saves the trained model parameters on disk. It extends the model training feature in Keras to support training a model with multiple optimizers with each optimizer operating on a separate trainable branch of a model. This feature is needed for our model fusion optimization, which we discuss in more detail in Section 9.4.3. Finally, the Trainer returns the model that has the best validation accuracy back to the user.

  In the current version, the Trainer supports single-node model training with or without GPU support. If multiple GPUs are available, the Trainer can also train models in data-parallel manner. We have focused on supporting DL models that fit in single-node/device memory during training (i.e., DRAM for CPU training and GPU's memory for GPU training) as many practical DTL applications operate in low-resource settings. Furthermore, the runtime memory usage of DL models is significantly reduced by pre-trained layer freezing, making most DTL models trainable on single-node/device memory. We discuss more details about runtime memory usage in Section 9.4.3.

9.4 System Optimizations

We first introduce the notion of *multi-model graph*, the core data structure used by our optimizations. We then dive into more details of our optimizations. We conclude the section by characterizing the attainable theoretical speedups. Table 9.2 presents the additional notation used in this section.

Table 9.2: Additional Notation used in Section 9.4

Symbol	Description
I, O	Input (I) and output (O) layers in model M.
U, V	Materializable (U) and materialized (V) layers in model M.
B_{disk}, B_{mem}	Disk storage budget (B_{disk}) and runtime memory budget (B_{mem}).
M^{opt}	Optimal reuse plan model for M.
$c_{comp}(l), c_{load}(l)$	Functions for estimating the computation cost and data load cost of layer l.
$s_{disk}(l), s_{mem}(l)$	Functions for estimating the storage usage and runtime memory usage of layer l.
$C(M)$	Training cost of model M.
$s_{mem}(M)$	Peak memory usage for training model M.
$q(l,M)$	$q(l,M) =$ is a function indicating the layer l is *pruned*, present and *computed*, or present and *loaded* (i.e., $l \in I$) in M.
r	Expected maximum training data records.

9.4.1 Multi-Model Graph

We create an information graph composed of all candidate models in a model selection workload, which we call a *multi-model graph*. It is inspired by the AND view graph in relational multi-query optimization [133]. But we adapt it to the DL model selection context by leveraging the properties of DL models and training. Next, we define some helper terms and formalize the multi-model graph.

Definition 9.4.1 *An **expression** for a layer l in a model M is a DAG of layers with model input layers I as sources and l as the sink.*

Definition 9.4.2 *An expression is a **materializable expression**, iff the sink layer of the corre-*

sponding DAG is materializable.

Definition 9.4.3 *Two layers l_i and l_j are said to be **identical**, i.e., $l_i \equiv l_j$, if both of them are of same type, have identical configuration values, and identical trainable parameter values. Two expressions e_i and e_j are said to be identical, i.e., $e_i \equiv e_j$, if both of them have the same DAG structure and all corresponding layer pairs are identical.*

Definition 9.4.4 *A model $M = (L, E)$ is called a **multi-model graph** for the models $M_1, M_2, \ldots, M_n$, iff for every output layer of every M_i, there is an expression in M that is identical to the expression of M_i's corresponding output layer.*

Constructing the Multi-Model Graph: For a model selection workload with a set of models $M_1, M_2, \ldots, M_n$, we construct the multi-model graph M by merging all the materializable identical sub-expressions in them. If layer l in model M_i is materializable (i.e., $l \in U_i$), the corresponding layer l in the multi-model M is also materializable (i.e., $l \in U$), and vice-versa. Only the materializable layers in the multi-model need to be considered for materialization (i.e., $V \subseteq U$). We use 4 metrics to capture the runtime and layer output characteristics of all layers in M. They can be obtained by profiling the original models in the workload. We represent these values normalized for a single training record. They include:

- $c_{comp}(l)$, which captures the layer **computation cost** in terms of floating-point operations (FLOPS). It includes both the forward- and backward-pass computation costs. The forward cost can be directly obtained from the profiling information. However, profiling information provided by DL frameworks often does not include backward-pass cost. Hence, as per the standard practice [128, 62], we use the forward cost to estimate the backward cost. For a trainable (i.e., not frozen) layer, we set it to thrice the number of forward-pass FLOPs to account for forward-pass, input gradient, and parameter gradient computations. For a frozen but not materializable layer, we set it to twice the number of forward-pass FLOPs to

account for both forward and input gradient computations. For a materializable layer, we set it to the forward-pass FLOPs as there is no back-propagation happening.

- $s_{disk}(l)$, which captures the layer **output size on disk** in Bytes. We estimate it using the output tensor dimensions and data type.

- $c_{load}(l)$, which captures the layer **output loading cost from disk** in terms of missed compute FLOPs. We calculate it by first estimating the disk read time and multiplying it by the FLOPs throughput of the system. We ignore the data transfer time from DRAM to GPU memory as disk load time dominates the total time. Both compute throughput and disk read speed affect $c_{load}(l)$. We use pre-configured values for them, which match the characteristics of the available hardware.

- $s_{mem}(l)$, which captures the layer **output size in memory** in Bytes. We estimate it using the output tensor dimensions and data type, similar to the on-disk output size. However, for a composite layer that consists of several basic layers (e.g., a transformer layer composed of several dense, addition, and layer normalization layers [274]), we estimate it by summing the size of all child layer output tensors. We treat composite layers differently to account for all the intermediate output tensors that the backward-pass may need. We explain this in more detail in Section 9.4.3.

9.4.2 Materialization Optimization

We formally present our materialization optimization problem and present a mixed-integer linear programming-based solution.

Materialization Optimization Problem

Our goal is to find an optimal set of intermediate layer outputs to materialize subject to a storage constraint. Given a set of such layers, we rewrite the model graphs to reuse these intermediate outputs during training. We assume that the model selection workload is fixed and repeated for all model selection cycles. Thus, we focus on materialization optimization for a single model selection cycle. Intermediate layer materialization incurs compute and I/O costs. However, it is amortized by the iterative DL model training costs, which get further amplified by multiple candidate models and multiple model selection cycles. Therefore, we ignore the computation and I/O costs for materializing the chosen layers and optimize for minimizing the total model training time. We estimate the storage footprint using a pre-configured maximum number of records r and reuse the obtained optimal materialization plan for all model selection cycles until the training dataset size reaches that limit. We explain how we relax the max training records constraint in Section 9.4.2.

Let $M = (L, E)$ be the multi-model graph for the set of models $\{M_i = (L_i, E_i) : \forall i \in 1, \ldots, n\}$. U is the set of materializable layers in M and V is the optimal set of materialized layers ($V \subseteq U$). I_i and O_i are the input and output layers of M_i, respectively. $C(M)$ is a function that estimates the training cost of model M (for one input record in FLOPs) and $q(l, M)$ is a function that indicates the presence of layer l in model M. We first introduce the notion of an optimal reuse plan model that captures how we should rewrite a model graph to reuse the materialized layer outputs in V and then explain how we find V.

Definition 9.4.5 *M_i^{opt} is called the **optimal reuse plan model** for M_i, iff*

1. *It has the same output layers as M_i (i.e., $O_i^{opt} = O_i$).*
2. *Every layer l in M_i^{opt} is also in M_i (i.e., $L_i^{opt} \subseteq L_i$).*
3. *For every layer l in M_i^{opt}, parents of l in M_i^{opt} are same as its parents in M_i; or l is in V (i.e., $l \in V$).*

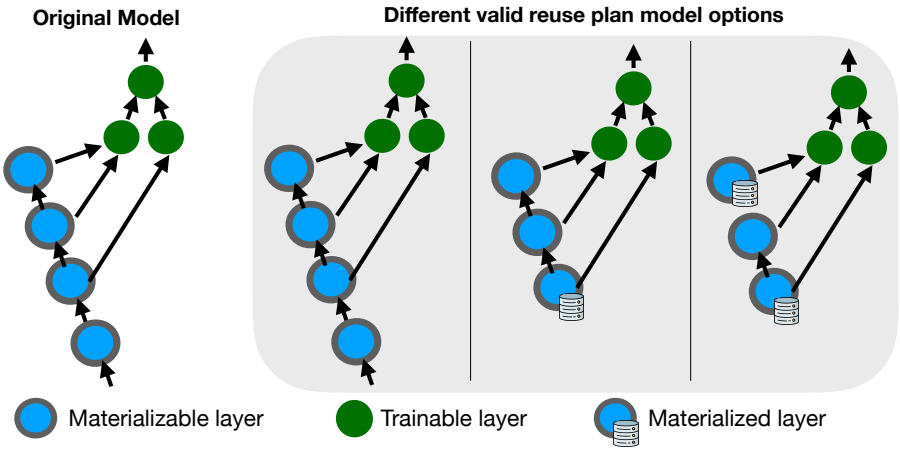


Figure 9.4: Valid reuse plan model options for a model with materializable layers.

4. *Has the lowest training cost $C(M_i^{opt})$ out of all such candidates.*

Training M_i^{opt} is equivalent to training M_i as both perform logically equivalent operations. M_i^{opt} can be obtained from $M_i = (L_i, E_i)$ by taking one of the following three actions for every layer $l \in L_i$: (1) pruning i.e., $q(l, M_i^{opt}) = pruned$, (2) retaining and computing i.e., $q(l, M_i^{opt}) = computed$, and (3) retaining and loading as an input i.e., $q(l, M_i^{opt}) = loaded$. Figure 9.4 shows an example model graph and several valid reuse plan models. We estimate the training cost of an optimal reuse plan model $C(M_i^{opt})$ by summing all layer compute costs and input loading costs as follows:

$$C(M_i^{mathitopt}) = \sum_{l \in L_i} \mathbb{1}\{q(l, M_i^{opt}) = computed\} \cdot c_{comp}(l) + \mathbb{1}\{q(l, M_i^{opt}) = loaded\} \cdot c_{load}(l) \tag{9.5}$$

Equation 9.5 makes the simplifying assumption that layer computations and input loadings do not overlap during training. However, DL model training operates in pipelined fashion on mini-batches of training data, and it may be possible to hide some of the data load costs by pre-fetching the data. Nevertheless, our formulation provides a reasonable upper bound for the total model training cost that is sufficient for our purpose.

With the optimal reuse plan model M_i^{opt} and its training cost $C(M_i^{opt})$ defined, the materialization optimization problem can be formally expressed as follows:

$$\underset{V, M_i^{opt} \forall i \in \{1,\ldots,n\}}{\arg\min} \sum_{i=1}^{n} C(M_i^{opt}) \cdot r \cdot \texttt{epochs}(\phi_i) \tag{9.6}$$

subject to:

$$\sum_{l \in V} s(l) \cdot r \leq B_{disk} \tag{9.7}$$

Equation 9.6 minimizes the total model training time. The model training time of a single model M_i is estimated by multiplying the training cost of the corresponding optimal reuse plan model M_i^{opt}, the maximum number of training records r, and the number of training epochs $\texttt{epochs}(\phi_i)$. $\texttt{epochs}(\phi_i)$ is a training hyperparameter provided by the user. Equation 9.7 ensures that the materialized layer outputs do not exhaust the disk storage budget B_{disk}.

Mixed Integer Linear Programming Formulation

We present a mixed-integer linear programming (MILP) formulation of the materialization optimization problem. To our knowledge, this is the first time an MILP formulation is used for materialization optimization in a DL systems context, although it has been previously used in other data management contexts [96]. Applying MILP techniques to this problem is possible because of our multi-model graph formalization, which we generate by leveraging the DAG nature and the presence of frozen layers in DL model graphs.

Let $l_{i,j}$ be the j^{th} layer of i^{th} model $M_i = (L_i, E_i)$ in the multi-model M. u_k is the k^{th} layer of the set of materializable layers U in M. We introduce three sets of binary indicator variables X, Y, and Z as follows:

for all $i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,|L_i|\}$, $k \in \{1,\ldots,|U|\}$

$$
\begin{aligned}
&(a)\ X_{i,j} = \mathbb{1}\{q(l_j, M_i^{opt}) \neq \mathit{pruned}\} \\
&(b)\ Y_{i,j} = \mathbb{1}\{q(l_j, M_i^{opt}) = \mathit{computed}\} \\
&(c)\ Z_k = \mathbb{1}\{u_k \in V\}
\end{aligned}
\tag{9.8}
$$

$X_{i,j}$ indicates whether the j^{th} layer of M_i is present in M_i^{opt}. $Y_{i,j}$ indicates whether the j^{th} layer of M_i is computed in M_i^{opt}. $X_{i,j}$ and $Y_{i,j}$ collectively determine $q(l_{i,j}, M_i^{opt})$. Z_k indicates whether the materializable layer u_k of M is materialized. With these indicator variables defined, the MILP-based approach for the materialization optimization problem can be expressed as follows:

$$
\begin{aligned}
\underset{X,Y,Z}{\arg\min} \sum_{\forall\ i,j} \big(& X_{i,j} \cdot c_{load}(l_{i,j}) \\
& + Y_{i,j} \cdot (c_{comp}(l_{i,j}) - c_{load}(l_{i,j}))\big) \cdot r \cdot \texttt{epochs}(\phi_i)
\end{aligned}
\tag{9.9}
$$

subject to:

$$
\begin{aligned}
&(a)\ l_{i,j} \in O \implies X_{i,j} \geq 1,\ \forall i,j \\
&(b)\ X_{i,j} - Y_{i,j} \geq 0,\ \forall i,j \\
&(c) \sum_{l_{i,k} \in \texttt{parents}(l_{i,j})} X_{i,k} - Y_{i,j} \geq 0,\ \forall i,j \\
&(d)\ u_k \equiv l_{i,j} \implies X_{i,j} - Y_{i,j} \leq Z_k,\ \forall i,j,k \\
&(e) \sum_{u_k \in U} Z_k \cdot s_{disk}(u_k) \cdot r \leq B_{disk}
\end{aligned}
\tag{9.10}
$$

Equation 9.9 is equivalent to the optimization objective presented in Equation 9.6. Equation 9.10 (a) ensures that output layers of all models are not pruned and avoids the trivial solution

where all layers are pruned. Equation 9.10 (b) ensures a computed layer is not pruned and Equation 9.10 (c) ensures parents of a computed layer are also not pruned. Equation 9.10 (d) ensures that only the materialized layers are loaded from the disk and Equation 9.10 (e) ensures that the cumulative size of the materialized layers does not exhaust the storage budget, which is equivalent to Equation 9.7.

Given the above, a straightforward approach to optimization is to use an MILP solver like Gurobi [134]. Z indicates the materialized layers and X and Y can be used to construct the optimal reuse plan models. If disk storage budget is not a critical resource, Z may contain materialized layers that do not get used in reuse plan models. Such layers can be discarded using a post-processing step. It can be shown that the above materialization optimization problem is NP-hard using a reduction from the known NP-hard Knapsack problem [286, 166]. However, we found that an MILP solver-based approach finds the optimal solution within a short execution time (e.g., few 10s of seconds) at the scale of practical DTL model selection workload sizes. We provide more details on MILP execution time in Section 9.5.3.

Incremental Feature Materialization

For a new batch of labeled data, we materialize it and also incrementally update the outputs of the chosen materialized layers. We repeat this until the pre-configured maximum number of training records r is reached. When we reach the maximum number of training records, we use an exponential backoff scheme with a factor of 2 to update r (i,e., $r \leftarrow 2 \times r$). We then rerun the materialization optimization to find a new set of materialized layers and materialize them. Overall, our exponential backoff scheme to update r provides a good balance between materialization overheads and storage wastage.

Algorithm 8 : **FUSEMODELS**(Q, B_{mem}, V)

$Q' = \{(M_i, M_i^{opt}, \phi_i) | \ \forall i \in [1, \ldots, |Q|]\}$
while there are not-considered fusible model pairs in Q' **do**
  $P = \{(i, j) | \text{all not-considered fusible model pair indices}\}$
  $M_{i,j} \leftarrow$multi-model for M_i and $M_j, \forall (i, j) \in P$
  $M_{i,j}^{opt} \leftarrow$optimal reuse plan model for $M_{i,j}, \forall (i, j) \in P$
  $c_{i,j} \leftarrow C(M_i^{opt}) + C(M_j^{opt}) - C(M_{i,j}^{opt}), \forall (i, j) \in P$
  $i_*, j_* \leftarrow \underset{(i,j) \in P}{\arg\max}\, c_{i,j}$, such that $s_{mem}(M_{i,j}^{opt}) \leq B_{mem}$
  $Q' \leftarrow Q' \cup \{(M_{i_*,j_*}, M_{i_*,j_*}^{opt}, \phi_{i_*} \cup \phi_{j_*})\}$
  $Q' \leftarrow Q' \setminus \{(M_{i_*}, M_{i_*}^{opt}, \phi_{i_*}), (M_{j_*}, M_{j_*}^{opt}, \phi_{j_*})\}$
return $\{(M_i, \phi_i) | \ \forall i \in 1, \ldots, |Q'|\}$

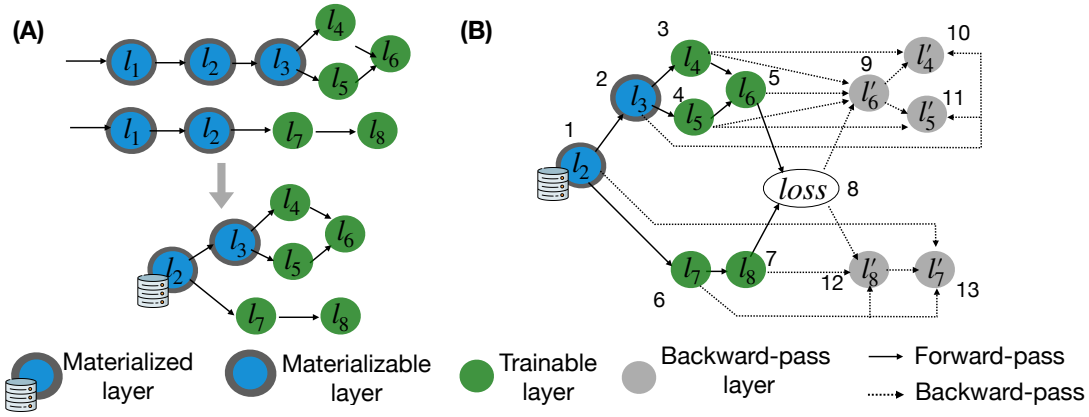
9.4.3 Model Fusion Optimization

In model fusion, we partition the set of models into groups such that the fused models corresponding to the partitions reduce the redundant computations with the highest margin while ensuring the runtime memory budget B_{mem} is not exhausted. We first explain our approach for finding such a partitioning. We then explain how to adapt the optimal reuse plan model for the fused model setting and also explain how we estimate fused model memory footprint.

Partitioning the Set of Models

We leverage the pipelining nature of the mini-batch SGD training method to train a fused model, which operates on one mini-batch at a time. We also ensure that all models in a partition have the same training batch size. Otherwise, they cannot be fused during training. Given such a partition, we create the multi-model for the partition and find the optimal reuse plan model. Multi-model creation will fuse only the materializable layers that do not require any training. Therefore, the training optimizer for the fused model's reuse plan model can be represented as the set of optimizers from the source models where each optimizer operates on the corresponding trainable branch.

However, finding the optimal partitioning requires considering all candidate partitions



(C)

Layer names

	l_2	l_3	l_4	l_5	l_6	l_7	l_8	$loss$	l'_6	l'_4	l'_5	l'_8	l'_7
2	1	1											
3	1	1	1										
4	1	1	1	1									
5	1	1	1	1	1								
6	1	1	1	1	1	1							
7	1	1	1	1	1	1	1						
8	1	1	1	1	1	1	1	1					
9	1	1	1	1	1	1	1	1	1				
10	1	1	1	1		1	1	1	1	1			
11	1	1		1		1	1	1	1		1		
12	1					1	1	1				1	
13	1					1						1	1

Topological traversal order

Figure 9.5: (A) A candidate partition containing two source models and the corresponding optimal reuse-plan model. (B) Augmenting reuse plan model with nodes to represent the backward-pass of training. (C) Topological traversal-based live tensor analysis for the model graph shown in (B).

(i.e., all possible model groupings), which is exponential in the number of models in the model selection workload. Furthermore, the training cost ($C(M^{opt})$) and the peak runtime memory usage ($s_{mem}(M^{opt})$) of the optimal reuse plan model for each partition's multi-model is not available in constant time. Thus, we use a greedy heuristic that only considers pair of models to be fused at a time. The high-level approach is presented in the `FuseModels` procedure of Algorithm 8.

`FuseModels` takes in a set of model and training hyperparameter pairs Q, runtime memory budget B_{mem}, set of materialized layers V, and returns a set of fused model and training hyperparameter pairs. For every model M_i in Q, we first find the optimal reuse plan model M_i^{opt} that reuses materialized intermediate layers in V and create a new set Q' containing (M_i, M_i^{opt}, ϕ_i)

triples. P is the set of all possible model pairs wherein both models have the same training batch size. For every candidate model pair indices (i, j) in P, we create the multi-model $M_{i,j}$ and find the optimal reuse plan model $M_{i,j}^{opt}$. We explain how we find the optimal reuse plan model $M_{i,j}^{opt}$ given V in Section 9.4.3. We also estimate the runtime memory usage $s_{mem}(M_{i,j}^{opt})$ and ensure that it does not exceed the runtime memory budget B_{mem}. Details on how we estimate $s_{mem}(M_{i,j}^{opt})$ are provided in Section 9.4.3. From the fusible model pairs, we pick the pair that will result in the highest cost reduction. We add the fused model back to Q' and remove the source models M_i and M_j from Q'. The training hyperparameters for a fused model are derived by combining source hyperparameters ϕ_i and ϕ_j. We repeat this process until there are no more fusible models.

Optimal Plan Given a Set of Materialized Layers

The optimal reuse plan model $M_{i,j}^{opt}$ for a fused model $M_{i,j}$ that reuses materialized layers V can be found by adapting the MILP presented in Section 9.4.2. The main difference here is that the set of materialized layers is already determined. Thus, we no longer need the indicator variable Z (Equation 9.8 (c)) and also remove the constraints (d) and (e) in Equation 9.10. We use $\{M_{i,j}\}$ as the set of input models. In this case, the multi-model M corresponding to the set of input models will be the same as the input model $M_{i,j}$. After the optimization, $M_{i,j}^{opt}$ can be obtained from the resulting indicator variable values X and Y. While most MILP problems are NP-hard, it has been shown that the resulting MILP problem can be solved in PTIME via a reduction to the MAX-FLOW problem [286].

Estimating Peak Runtime Memory Usage

Estimating the peak runtime memory usage of training a DL model is a challenging task as it depends on various factors including both workload- and system-specific. One could estimate peak runtime memory usage of a model by actually running the model and observing the memory usage. However, our model fusion optimization evaluates many model candidates, which is in the

order of $O(n^3)$ where n is the number of source models. Estimating memory usage at runtime would require creating checkpoints for many potential fused models and running them, which could add massive overheads. Thus, we decided to use an analytical model to estimate peak runtime memory usage. We found that our analytical model is accurate enough to avoid any out-of-memory workload crashes at runtime.

We identity three main types of memory usage that dominate the overall usage: (1) memory to store the parameter tensors, (2) workspace memory for performing layer operations, and (3) memory to store the layer outputs needed for back-propagation. We calculate the first type by using the dimensions and data types of the parameter tensors. The second type depends on the DL system. We rely on the user to set a value for this (e.g., 1GB). The third type depends on both the model architecture and the DL system and it often dominates the overall memory usage. Next, we discuss more details on how we estimate it.

The backward-pass of DL training needs access to the layer output tensors generated during the forward-pass. For example, the backward-pass operation of linear algebra-based layers such as Dense and Convolutional layers need access to the forward-pass layer input to calculate the parameter gradient. Some non-linear transformation layers like ReLU need access to layer output to calculate the input gradient. And some other non-linear transformation layers like MaxPooling need access to both layer input and output to calculate the input gradient. Thus, the DL system will accumulate layer output tensors during the forward-pass and gradually release them during the backward-pass. However, the exact order by which output tensors are accumulated and released is determined by the specific order by which layer operations are performed and also on how aggressively memory is allocated and deallocated by the DL system. Popular DL frameworks like TensorFlow and PyTorch execute operations in a topological order [269, 234].

We estimate the memory needed for storing layer output tensors by performing a topological traversal-based *live tensor analysis*. We augment the optimal reuse plan of the fused model by adding nodes needed to represent the backward-pass. We also add a node to represent the loss

computation and add edges between every output layer and the new loss node. This loss node is responsible for calculating the loss for all trainable model branches using the corresponding optimizer. It also ensures that our fused model adheres to the two-phase (i.e., forward and backward) training template supported by DL systems. For every non-materializable layer l_i, we add a node l_i' to represent the backward-pass computation of that layer. We set $s_{mem}(l_i')$ to be same as $s_{mem}(l_i)$. We treat all backward-pass computations uniformly and add edge dependencies as follows:

- Output from the forward-pass layer by adding (l_i, l_i') edge.
- Input(s) to the forward-pass layer by adding $\{(l_p, l_i') : \forall l_p \in parent(l_i)\}$ edges.
- Backward-pass output gradient(s) from the child layers by adding $\{(l_s', l_i') : \forall l_s \in child(l_i)\}$ edges.

Figure 9.5 (A) presents an example candidate partition containing two source models from the model selection workload and the corresponding reuse plan model. The reuse plan model is obtained by performing both our materialization and model fusion optimizations. Notice that each model's trainable layers are in a separate branch. We then augment the reuse plan model with nodes and edge dependencies needed to represent the backward-pass as shown in Figure 9.5 (B). To estimate peak runtime memory for storing layer outputs, we perform a topological traversal over the created graph structure while keeping track of the live output tensors. The output tensor size of layer l for a single training record is given by $s_{mem}(l)$. The order by which each node is visited during the topological traversal is denoted in the figure. Figure 9.5 (C) presents the live tensor analysis. When processing a node, we assume all its inputs and output tensors are live. We release a tensor if it is not needed for the current or future nodes. For example, as highlighted in Figure 9.5 (B), when processing the loss node, the l_1 output tensor is not live as it is not used by any node that is yet to be visited.

Notice that there might be more than one possible topological order and as a result, the order used by our analysis may differ from the order used by the training framework. However, for any topological traversal order, we can claim that the maximum number of live tensors is only one more than the number of live tensors needed when processing the loss node. This is because the loss node acts as a barrier needing the entire forward-pass to be completed before starting the backward-pass. Overall, we found that our approach can provide reasonable memory usage upper bounds that are sufficient for our model fusion optimization.

9.4.4 Theoretical Speedups

We define the theoretical speedup as the ratio between the total training cost of all model layers and the total training cost of only the non-materializable (i.e., $m(l) = \mathit{False}$) model layers, as per Equation 9.11. It assumes complete avoidance of computational redundancies without accounting for data movement overheads. It will be equivalent to the speedup achieved by our materialization optimization in the hypothetical case of zero load cost and a disk storage budget to materialize all materializable layers.

$$\frac{\sum_{i=1}^{|Q|} \sum_{l \in L_i} c_{comp}(l) \cdot \texttt{epochs}(\phi_i)}{\sum_{i=1}^{|Q|} \sum_{l \in L_i} \mathbb{1}[m(l) = \mathit{False}] \cdot c_{comp}(l) \cdot \texttt{epochs}(\phi_i)} \tag{9.11}$$

9.5 Experimental Evaluation

We now present an extensive empirical evaluation seeking to answer the following questions. (1) How does NAUTILUS compare with current practice and other baselines on runtimes, accuracy, and resource utilization? (2) How much NAUTILUS's optimizations contribute to the overall runtime reductions?

Datasets: We use two benchmark datasets: *CoNLL-2003* [271] and *Malaria* [235].

CoNLL-2003 is a text dataset and the prediction task is named entity recognition. *Malaria* is an image dataset and the prediction task is identifying Malaria from blood cell images. For *CoNLL-2003* and *Malaria*, we have unlabeled data pools of sizes 10,000 and 8,000 records, respectively.

Workloads: We run 5 end-to-end workloads covering feature transfer, fine-tuning, and adapter training. Table 9.3 summarizes the transfer learning and the model selection configuration values of the workloads. Feature transfer workloads (*FTR-**) use BERT-base as the source model. *FTR-1* explores 6 feature transfer strategies that are same as the ones reported in [109]. *FTR-2* and *FTR-1* explore 4 and 1 transfer strategies, respectively. The fine-tuning workload *FTU* uses the popular computer vision model ResNet-50 [136] and we vary the number of fine-tuned residual layer blocks from the top. The adapter training workload *ATR* also uses the BERT-base model. We use Houlsby [143] type adapters and vary the number of layers with adapters from the top. For *FTR-** workloads we add a new transformer layer on top of the extracted features. For all workloads, we add a new Softmax classification layer on top of the last hidden layer. *FTR-** and *ATR* workloads use the *CoNLL-2003* dataset; *FTU* workload uses the *Malaria* dataset.

For all workloads, we generate the labeled dataset iteratively. For each cycle, we label 500 records with a 400/100 train/validation split, perform model selection on all labeled data up to that point, and repeat the process for 10 cycles. We simulate the human labeler by programmatically releasing the labels.

Experimental Setup: We use a machine with 32 GB RAM, Intel i7 3.40GHz CPU, 1TB SSD, and NVIDIA Titan X GPU with 12 GB memory. It runs Ubuntu 18.04 with TensorFlow version 2.4, CUDA version 11.0, and cuDNN version 7.5. For our optimizer, we set the disk read throughput to 500 MB/s and the compute throughput to 6 TFLOP/s, which is 50% of the theoretical FLOPS rate of the Titan X GPU. These hardware settings are configurable by the user. We report average of 3 runtimes with 95% confidence intervals.

Table 9.3: Model selection configurations of workloads.

Workload	Tuning Parameters				# Models
	Transfer Learning Approach	Batch Size	Learning Rate	Epochs	
FTR-1	**Feature Transfer** from: {embedding, second last hidden, last hidden, sum last 4 hidden, concat last 4 hidden, sum all hidden}	{16, 32}	$\{5,3,2\} \times 10^{-5}$	{5}	36
FTR-2	**Feature Transfer** from: {second last hidden, last hidden, sum last 4 hidden, concat last 4 hidden}	{16, 32}	$\{5,3,2\} \times 10^{-5}$	{5}	24
FTR-3	**Feature Transfer** from: {concat last 4 hidden}	{16, 32}	$\{5,3,2\} \times 10^{-5}$	{5, 10}	12
ATR	**Adapter Training** for: {last hidden, last 2 hidden, last 3 hidden, last 4 hidden}	{16, 32}	$\{5,3,2\} \times 10^{-5}$	{5}	24
FTU	**Fine-tuning** : {last 3 hidden, last 6 hidden, last 9 hidden, last 12 hidden}	{16, 32}	$\{5,3,2\} \times 10^{-5}$	{5}	24

9.5.1 End-to-End Runtimes

Model Selection Time: We first evaluate the total model selection time for 4 different approaches. This enables us to isolate NAUTILUS's ability to reduce DTL model selection runtimes, which is independent of the data labeling approach and the labeling time. The four approaches that we evaluate are *Current Practice*, *MAT-ALL*, NAUTILUS, and *FLOPs Optimal*. *Current Practice* is the naive baseline, which trains unmodified models independently and repeats the process for all cycles. It incurs the highest level of redundancies. *MAT-ALL* is a strong baseline that materializes all materializable layers and uses them during training, irrespective of whether it is efficient to compute them rather than loading them. Note that *MAT-ALL* needs parts of our code from NAUTILUS. NAUTILUS is our optimizer-picked plan, which performs both our materialization and model fusion optimizations. We execute it with a disk storage budget (B_{disk}) of 25 GBs and a runtime memory budget (B_{mem}) of 10 GBs. *FLOPs Optimal* is calculated by dividing the *Current Practice* time by the theoretical speedup. Figure 9.6 presents the results.

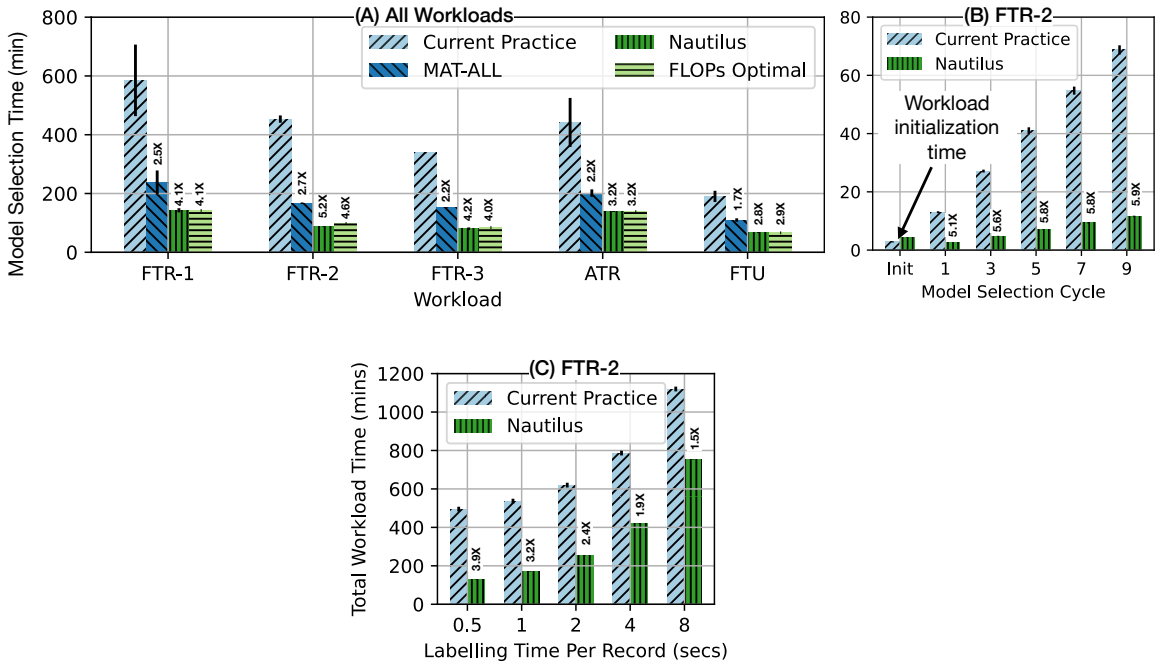


Figure 9.6: (A) Total model selection time. (B) Model selection time breakdown by model selection cycle for *FTR-2* (only the odd numbered model selection cycles are shown due to space constraints). (C) Total time for *FTR-2* including data labeling time.

NAUTILUS offers significant speedups with the highest speedup of 5.2X seen on *FTR-2*. The highest speedup for *MAT-ALL* is also for *FTR-2*, which is 2.7X. In all cases, NAUTILUS outperforms the *MAT-ALL* baseline. In the worst case, *MAT-ALL* is 48% slower than NAUTILUS for the *FTR-2* workload. *MAT-ALL* is slower than NAUTILUS because it incurs higher IO overheads due to reading all layers, even though the machine used has fast SSDs. NAUTILUS selectively loads some layers only as needed and recomputes the remaining to reduce the overall runtimes. One can also allocate more DRAM memory, materialize all layers in DRAM for *MAT-ALL*, and access with a small load cost. But this will be resource-inefficient and will likely cost more for a user due to Pareto tradeoffs. In contrast, NAUTILUS pushes the cost versus performance Pareto frontier to achieve higher efficiency at lower resource costs. We performed a cost estimation for executing the *FTR-1* workload with 10,000 records using the Google cloud computing cost calculator [125] and found that NAUTILUS can reduce the workload cost by 76% (0.97 $/hr for

MAT-ALL vs. 0.55 $/hr for NAUTILUS). NAUTILUS will incur less cost mainly because it will incur less DRAM cost.

In all cases, NAUTILUS achieves slightly better or competitive runtimes to the *FLOPs Optimal* runtime. This is because NAUTILUS significantly amortizes the training and I/O overheads, which are not accounted for in the FLOPS reduction-based theoretical speedup calculation. Also, NAUTILUS's speedups vary based on the characteristics of the workload. For example, speedups are generally higher for *FTR-** workloads compared to *ATR* or *FTU*, as the latter workloads have more trainable layers. Also, absolute runtimes are lower for the *FTU* compared to other workloads, as the former uses a less compute-intensive model. Overall, NAUTILUS reduces DTL model selection runtimes substantially for all workloads.

Model Selection Time Breakdown: Figure 9.6 (B) presents the model selection time breakdown by model selection cycle. *Current Practice* and NAUTILUS take 2.7 and 4.4 minutes to initialize the workload, respectively. By drilling into the workload initialization time, we found that NAUTILUS spends 63% of time creating the original model checkpoints, which is also performed by the *Current Practice*. Additionally, NAUTILUS spends 12% time profiling the original models, 3% time generating the optimized plan, and 21% time generating model checkpoints for the optimized plan. NAUTILUS's speedups are slightly lower in the early cycles compared to the later ones. This is because the later cycles have more training data, and the effect of fixed overheads is less pronounced in them. If there are no reuse opportunities, NAUTILUS will incur a one-time cost for model profiling and running the optimizer, which is less than 1% of the total workload runtime of *Current Practice*.

Total Workload Time: Finally, we evaluate the total workload time for the *FTR-2* workload for different data labeling runtime values. The *CoNLL-2003* dataset used for the *FTR-2* workload has 20 words per record on average. Hence, we vary the labeling time per data record between 0.5 seconds and 8 seconds. The 0.5 seconds case can be considered as a multi-labeler scenario (e.g., cloud labelers); 8 seconds scenario can be considered as a single-labeler scenario.

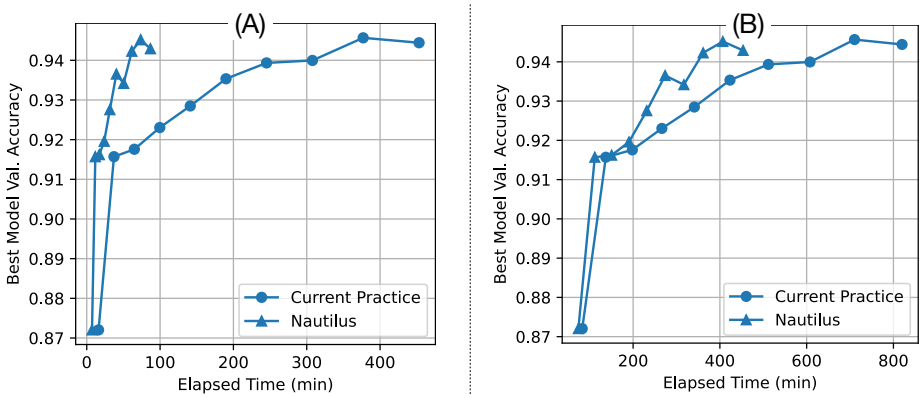


Figure 9.7: *FTR-2* learning curves with (A) zero and (B) 4 seconds/label data labeling cost values.

For the 0.5 seconds scenario, NAUTILUS achieves a speedup of 3.9X compared to the *Current Practice*. For the 8 seconds scenario, NAUTILUS's speedup reduces to 1.5X, as higher data labeling time dominates the overall workload time.

9.5.2 Accuracy

Both *Current Practice* and NAUTILUS perform logically equivalent SGD training. Thus, they both should achieve the same statistical efficiency. To validate this, we plot the best model's validation accuracy against the elapsed model selection time for the *FTR-2* workload. Figure 9.7 (A) presents the results. We see that both approaches achieve very similar validation accuracies after every model selection cycle. However, NAUTILUS achieves them 5X faster. We repeat the experiment with a data labeling time of 4 seconds/label and plot the best validation accuracy against elapsed total time as shown in Figure 9.7 (B). In this case, NAUTILUS is 2X faster.

9.5.3 Drill-Down Analysis

Contribution of Our Optimizations: We run the end-to-end workloads using NAUTILUS but disable either the materialization (*MAT OPT*) or the model fusion (*FUSE OPT*) optimization. Figure 9.8 presents the results. For all cases except *ATR*, running Nautilus w/o

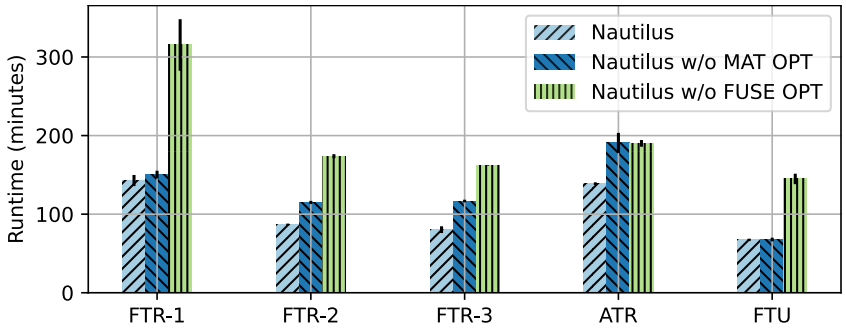


Figure 9.8: Model selection time with and without *MAT* and *FUSE* optimizations.

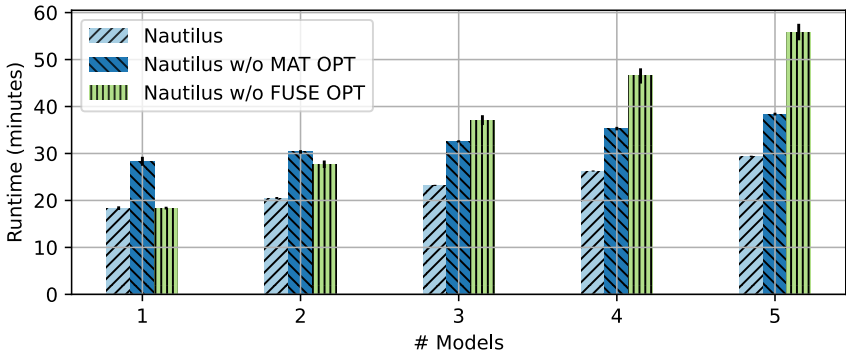


Figure 9.9: Model selection time for different number of models with and without *MAT* and *FUSE* optimizations.

FUSE OPT causes more slowdown than running w/o *MAT OPT*. The highest slowdown for NAUTILUS w/o *FUSE OPT* is for *FTR-1*, which is 54.7%; for w/o *MAT OPT*, it is for *FTR-3*, which is 31.2%. For *FTU*, NAUTILUS's runtime does not change w/o *MAT OPT*. This is because ResNet-50 is a less compute-intensive model and NAUTILUS computes all materializable layers instead of loading them. In 4 out of 5 end-to-end experiments, *FUSE OPT* contributes more than the *MAT OPT*. But combining both optimizations achieves even lower runtimes. The benefits of each optimization vary based on workload and hardware characteristics.

We also run an experiment where we vary the number of models in the model selection workload. For this, we use *FTR-2* and fix the feature transfer strategy to the concatenation of the last four layers, fix the batch size to 16, and vary the number of explored learning rates. Figure 9.9 presents the results. When the number of models is less than or equal to 2, running NAUTILUS w/o *MAT OPT* causes more slowdown than running it w/o *FUSE OPT*. However, when the number of

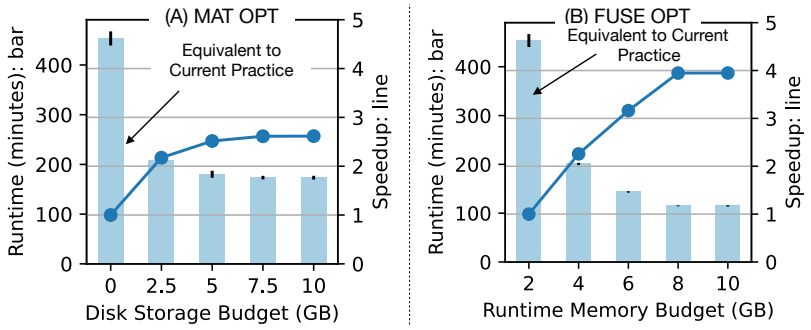


Figure 9.10: *FTR-2* model selection time using (A) *MAT OPT* vs. storage budget and (B) *FUSE OPT* vs. memory budget.

models increases, running w/o *FUSE OPT* causes more slowdown. With more models, *FUSE OPT* has more opportunities to avoid redundant computations and amortize training and I/O overheads. Also, with only 1 model, *FUSE OPT* doesn't give any benefits as there are no opportunities for model fusion.

Finally, we run *FTR-2* by only using *MAT OPT* or *FUSE OPT*, and vary the disk storage budget B_{disk} and runtime memory budget B_{mem}, respectively. Running *MAT OPT* with a B_{disk} of 0 GBs is equivalent to the *Current Practice*. Running *FUSE OPT* with a B_{mem} of 2 GBs is also equivalent to the *Current Practice* as it does not fuse any models. For *FUSE OPT* we also ensure that the training process does not consume more than the allocated B_{mem} from the available GPU memory. Thus, this experiment also shows that our memory estimation approach is capable of avoiding workload crashes due to out-of-memory errors. As B_{disk} is increased, *MAT OPT* runtime decreases and plateaus after 7.5 GBs where it achieves 2.6X speedup compared to the *Current Practice*. As the B_{mem} is increased, *FUSE OPT* runtime also decreases and plateaus after 8 GBs where it achieves a 4.0X speedup. NAUTILUS combines the benefits of both optimizations and achieves the lowest runtimes.

System Resources Utilization: We evaluate the GPU utilization and cumulative disk reads/writes for executing the *FTR-2*. Figure 9.11 presents the results. NAUTILUS yields a higher average GPU utilization of 66% compared to the 57% of *Current Practice*. It also performs 4.3X

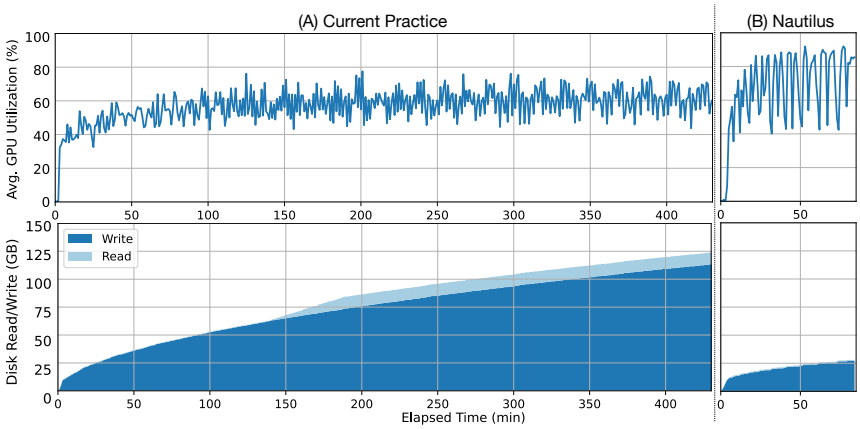


Figure 9.11: Average GPU utilization and cumulative disk reads and writes for executing the *FTR-2*.

fewer disk writes and 11.8X fewer disk reads. This is because *Current Practice* checkpoints the entire original model after every model training, which is around 400-500MBs. But most of the parameters in them are frozen parameters and do not need repeated checkpointing. In contrast, NAUTILUS checkpoints modified model graphs with most frozen parameters pruned. Writing less amount of data also helps with better page caching for the reads.

9.6 Conclusion

DTL model adaptation, where one adapts a pre-trained model for a new target task, is a crucial paradigm for democratizing deep learning. Yet, the current practice of executing DTL workloads faces significant resource inefficiency and usability issues. In this work, we formalize the DTL workload from a data management standpoint and enable two multi-query optimization-inspired optimizations: materialization optimization and model fusion optimization. Our optimizations leverage the physical characteristics of the DTL model adaptation workloads to optimize them. We implement our optimizations in a data system we call NAUTILUS. NAUTILUS reduces DTL model selection runtimes by even up to 80% and significantly improves usability and resource usage.

Chapter 9 contains material from "Nautilus: An Optimized System for Deep Transfer Learning over Evolving Training Datasets" by Supun Nakandala and Arun Kumar, which will appear in Proceedings of 2022 ACM SIGMOD International Conference on Management of Data. The dissertation author was the primary investigator and author of this paper. The code for our system is open source and is available on GitHub: https://github.com/AdaLabUCSD/Nautilus.

Chapter 10

Related Work

10.1 Related Work for CEREBRO

Systems for Model Selection: Google Vizier [122], Ray Tune [192], SparkDL [21], Dask-Hyperband [249], and Spark-Hyperopt [51] are systems for model selection. Vizier, Ray, and Dask-Hyperband are pure task-parallel systems that implement some AutoML procedures. SparkDL and Spark-Hyperopt use Spark for execution but distribute configs in a task-parallel manner with full data replication. CEREBRO offers higher overall resource efficiency compared to pure task- or pure data-parallel approaches.

AutoML Procedures: AutoML procedures such as Hyperband [186] and PBT [149] are orthogonal to our work and exist at a higher abstraction level. They fit a common template of per-epoch scheduling in CEREBRO. While ASHA [185] does not fit this template, CEREBRO can still emulate it well and offer similar accuracy. Bayesian optimization is a class of AutoML procedures, some of which have a high degree of parallelism for searching configs (e.g., Hyperopt [70]); CEREBRO supports such procedures. Some others run a sequential search, leading to a low degree of parallelism (e.g., [165, 69]); these may not be a fit for CEREBRO.

Distributed SGD Systems: There is much prior work on data-parallel distributed SGD,

including centralized fine-grained (e.g., [222, 297, 152, 146]) and decentralized fine-grained (e.g., [281, 191, 222]). These are all complementary to our work because they train one model at a time, while we focus on parallel model selection. As we showed, such approaches have higher communication complexity and thus, higher runtimes than MOP in our setting. Also, since CEREBRO performs logically sequential SGD, it ensures theoretically best convergence efficiency. CROSSBOW [167] proposes a new variant of model averaging for single-server multi-GPU setting. But it is also complementary to our work, since it also trains one model at a time. Overall, our work breaks the dichotomy between data- and task-parallel approaches, thus offering better overall resource efficiency.

Hybrid Parallelism in ML Systems: MOP is inspired by the classical idea of process migration in OS multiprocessing [64]. We bring that notion to the data-partitioned cluster setting. This generic idea has been used before in limited contexts in ML systems [172, 74]. The closest to our work is [90], which proposes a scheme for training many homogeneous CNNs on a homogeneous GPU cluster. They propose a ring topology to migrate models, resembling a restricted form of MOP. But their strong homogeneity assumptions can cause stalls in general model selection workloads, e.g., due to heterogeneous neural architectures and/or machines. In contrast, we approach this problem from first principles and formalize it as an instance of open shop scheduling. This powerful abstraction lets CEREBRO support arbitrary deep nets and data types, as well as heterogeneous neural architectures and machines. It also enables CEREBRO to support replication, fault tolerance, elasticity, and arbitrary AutoML procedures, unlike prior work. SystemML also supports a hybrid of task- and data-parallelism for better plan generation for linear algebra-based classical ML on top of MapReduce [75]. CEREBRO is complementary due to its focus on deep nets and SGD's data access pattern, not linear algebra-based classical ML. Finally, a recent benchmark study suggested that communication bottlenecks inherent in pure data-parallelism imply hybrid parallelism is crucial for scalable ML systems [270]. Our work validates that suggestion for deep learning workloads.

Multi-Query and Other System Optimizations: MOP is also inspired by multi-query optimization (MQO) [250]. A recent line of work in the database literature studies MQO for deep learning, including staging and sharing work in CNN transfer learning [210] and batched incremental view maintenance for CNN inference [211, 223, 212]. CEREBRO furthers this research direction. All these MQO techniques are complementary and can be used together. Several works optimize the internals of deep net or SGD systems, including communication-computation pipelining [216], new compilation techniques [151], model batching [217], and execution on compressed data [184]. They are complementary to CEREBRO, since they optimize lower-level issues. MOP's generality enables CEREBRO to be hybridized with such ideas.

Scheduling: Gandiva [285], Tiresias [129], and SLAQ [292] are cluster scheduling frameworks for deep learning. They focus on lower-level primitives such as resource allocation and intra-server locality for reducing mean job completion times. CEREBRO is complementary as it exists at a higher abstraction level and focuses on model selection throughput. How compute hardware is allocated is outside our scope. There is a long line of work on job scheduling in the operations research and systems literatures [142, 81, 121]. Our goal is *not* to create new scheduling algorithms but to apply known techniques to a new ML systems setting.

10.2 Related Work for HUMMINGBIRD

PyTorch [228], TensorFlow [54], MXNet [28], CNTK [27] are DNN frameworks that provide easy-to-use (tensor-based) APIs for authoring DNN models, and heterogeneous hardware support for both training and inference. Beyond these popular frameworks, inference runtimes such as ONNX [15], nGraph [31], TVM [93], and TensorRT [35] provide optimizations and efficient execution targets, specifically for inference. To prove the versatility of our approach, we have tested HUMMINGBIRD with both PyTorch and TVM. HUMMINGBIRD uses a two-level, logical-physical optimization approach. First, logical optimizations are applied based on the

operators composing the pipeline. Afterwards, physical operator implementations are selected based on model statistics, and physical rewrites, which are externally implemented by the DL system, are executed (e.g., algebraic rewrites, operator fusion). Willump [170] uses a similar two-level optimization strategy, although it targets Weld [225] as its low level runtime and therefore it cannot natively support inference on hardware accelerators. Conversely, HUMMINGBIRD casts ML pipelines into tensor computations and takes advantage of DL inference systems to ease the deployment on target environments. Other optimizers for predictive pipelines, such as Pretzel [182], only target logical optimizations. We have integrated HUMMINGBIRD into Raven [157] as part of our bigger vision for optimizing ML prediction pipelines.

10.3 Related Work for KRYPTON

Methods for Explaining CNN Predictions: Perturbation-based and gradient-based are the two main kinds of methods. Perturbation-based methods observe the output of the CNN by modifying regions of the input image [290, 296, 242]. OBE belongs to this category. Gradient-based methods generate a sensitivity heatmap by computing the partial derivatives of model outputs with respect to every input pixel [259, 251, 266]. The recently proposed "Integrated Gradients" (IGD) method belongs into this category [266]. Empirically, we found that OBE produces higher quality heatmaps with better localized regions of interest compared to IGD, while being competitive on runtime. In practice, however, OBE is usually the method of choice for domain scientific users, especially in radiology [256, 202], since it is easy to understand for non-technical users and typically produces high-quality and well-localized heatmaps.

Faster CNN Inference: EVA^2 [82] is a custom software-hardware integrated stack for exploiting temporal redundancy in video frames. While one can map OBE to a video, EVA^2 will still perform motion estimation computations on whole frames and not exploit spatial redundancy across frames as our batched IVM does. Since our optimizations are at the logical level, they are

also applicable to any compute hardware. CBinfer performs change-based approximate CNN inference to accelerate real-time object recognition on video [85]. Similarly, NoScope accelerates object classification on video streams using model cascades [156]. Our focus is on accelerating the OBE workload for images, not video streams. Our IVM and approximate inference optimizations exploit the semantic properties of OBE, not general object recognition. Both of these tools are orthogonal to our focus.

Query Optimization: Our work is inspired by the long line of work on relational IVM [96, 132, 183], but ours is the first work to use the IVM lens for OBE with CNNs. Our novel algebraic IVM framework is closely tied to the dataflow of CNN layers, which transform tensors in non-trivial ways. Our work is related to the IVM framework for linear algebra in [219]. They focus on bulk matrix operators and incremental addition of rows. We do not deal with bulk matrix operators or addition of rows but more fine-grained CNN inference computations and in-place updates to image pixels due to occlusions. Also related is the IVM framework for distributed multi-dimensional array database queries in [295]. An interesting connection is that CNN layers with local spatial context can be viewed as a variant of spatial array join-aggregate queries. But our work enables end-to-end IVM for entire CNNs, not just one-off spatial queries involving data materialization and loading. Our focus is on popular deep learning systems, not array databases. Finally, we also introduce novel CNN-specific and human perception-aware optimizations to accelerate OBE.

Our work is also inspired by relational MQO [250, 178], but our focus is CNN inference, not relational queries. To the best of our knowledge, ours is the first work to combine MQO with IVM, at least in the context of CNN inference. Our approximate inference optimizations are inspired by approximate query processing (AQP) techniques [120]. But unlike statistical approximations of aggregations over relations, our techniques are novel CNN-specific and human perception-aware heuristics tailored to accelerating OBE.

10.4 Related Work for VISTA

Multimodal Analytics: Transfer learning is used for other kinds of multimodal analytics too, e.g., image captioning [158]. Our focus is on integrating images with structured data. A related but orthogonal line of work is "multimodal learning" in which deep neural networks are trained from scratch on images [218, 264]; this incurs high costs for resources and labeled data, which feature transfer mitigates.

Multimedia Systems: The multimedia and database systems communities have studied "content-based" image retrieval, video retrieval, and similar queries over multimedia data [55, 155]. But they typically used non-CNN features such as SIFT and HOG [103, 195] not learned or hierarchical CNN features, although there is a resurgence of interest in CBIR with CNN features [277, 288]. Such systems are orthogonal to our work, since we focus on CNN feature transfer, not retrieval queries on multimedia data. One could integrate VISTA with multimedia databases.

Query Optimization: Our work is inspired by a long line of work on optimizing queries with UDFs, multi-query optimization (MQO), and self-tuning DBMSs. For instance, [140, 89, 26] studied the problem of optimizing complex relational queries with UDF-based predicates. Unlike such works on queries with UDFs in the `WHERE` clause, our work can be viewed as optimizing UDFs expressed in the `SELECT` clause for materializing CNN feature layers. VISTA is the first system to bring the general idea of MQO to complex CNN feature transfer workloads, which has been studied extensively for SQL queries [250]. We do so by formalizing partial CNN inference operations as first-class citizens for query processing. In doing so, our work expands a recent line of work on materialization optimizations for feature selection in linear models [168, 291] and integrating ML with relational joins [174, 91, 248, 175]. Finally, our work also expands the work in database systems on optimizing memory usage based on data access patterns of queries [98]. But ours is the first work to study this issue in depth for CNN feature transfer queries.

System Auto-tuning: There is much prior work on auto-tuning the configuration of RDBMSs, Hadoop/MapReduce, and Spark for relational workloads (e.g., [141, 58]). Our work is inspired by these works but ours is the first to focus on the CNN feature transfer workload. We explain the new efficiency and reliability issues caused by CNNs and feature layers and apply our insights for CNN-aware auto-tuning in our setting that straddles PD and DL systems.

10.5 Related Work for NAUTILUS

Materialization Optimizations: Our work is inspired by the long line of work on reusing intermediates to optimize ML workloads [168, 291, 262, 273, 286, 231, 210, 194], but ours is the first to apply it to optimize DTL over evolving training data. Prior work in VISTA system [210] also uses feature materialization to optimize DL feature transfer-based multi-modal analytics. However, it supports only linear DL model graphs and features extracted from only one layer at a time. Also, VISTA's focus is on training classical ML models (e.g., linear regression) on the extracted features from a fixed dataset and not DTL over evolving data. NAUTILUS generalizes VISTA in 3 dimensions. It supports 1) DAG structured DL model graphs with arbitrary feature compositions, 2) evolving labeled datasets, and 3) all 3 popular transfer learning paradigms. Intermediate feature materialization is also used in AUTOFREEZE [194] to optimize the fine-tuning of a single BERT [109] model. NAUTILUS supports arbitrary DL models, all popular transfer learning paradigms, and also model selection.

NAUTILUS's cost model-based materialization optimization extends the optimal reuse plan formulation in HELIX system [286]. Specifically, we represent models in a DTL model selection workload using an optimizable graph structure called multi-model graph and jointly solve the materialized intermediate output selection and the reuse plan generation in a single MILP formulation.

Joint Model Training Optimizations: NAUTILUS's model fusion is a form of common

sub-expression elimination (CSE). CSE is also used in several other systems to eliminate redundant data pre-processing steps [217, 193, 293, 130]. NAUTILUS extends this to also eliminate redundancies in materializable layers. However, existing systems require the user to select the set of models to fuse [217, 293], adopt a trial-and-error approach [193] to find the set of models, or trains each source model in a separate GPU [130]. NAUTILUS uses profiling information to estimate fused model memory footprint and automatically picks an optimal set of models to fuse.

DL Model Selection: Several systems have been proposed to optimize DL model selection [215, 122, 192, 258, 51, 294, 190]. However, the focus of all these systems is on utilizing the parallelism available in a cluster to scale the DL model selection. In contrast, the focus of NAUTILUS is on DL model selection in low-resource settings such as workstations or PCs. Also, NAUTILUS focuses on the human-in-the-loop setting with iteratively generated labeled data.

NAUTILUS supports two popular model selection procedures: grid and random search, which cover an overwhelming majority of model selection applications [78]. However, there are other more complex model selection procedures proposed in the literature [186, 185, 70, 149, 209]. We leave adding support for them to future work.

Other DL System Optimizations: Various other techniques can be also used to optimize DTL. They include operator fusion [93, 52], hybrid parallel execution [151], layer batching [217, 293, 193], model compression [135], and model distilling [247]. They are complementary to the optimizations performed by NAUTILUS since they mainly optimize lower-level operator execution. There also exist systems that support training larger than GPU memory models [240, 177, 94]. They are complementary to our work and can be combined with NAUTILUS to train or fuse larger models.

NAUTILUS's runtime memory estimation operates at a higher level that is independent of the exact memory allocation/deallocation behavior of the underlying DL system. However, one can also try to mimic the exact behavior and obtain improved memory usage estimates as in [119]. Nevertheless, our approach provides reasonable upper-bounds sufficient for our requirement.

Chapter 11

Conclusion and Future Work

In this dissertation, we take a first step in filling an important research gap in DL systems architecture: the lack of a query optimization layer. We show that by fundamentally reimagining DL workloads as data processing workloads, we can develop novel query optimization-inspired techniques to optimize DL workloads. Our techniques take inspiration from classical query optimization ideas such as query rewriting, multi-query optimization, materialization optimizations, incremental view maintenance, etc., and leverage the semantic, logical, and physical characteristics of DL workloads to optimize them. We show that by doing so, we can enable significant efficiency improvements and resource savings (sometimes even over 10X) for a variety of popular and important DL workloads covering three popular DL workload types: 1) model building, 2) model inference, and 3) transfer learning. Our work makes interesting new connections between classical query optimization techniques and DL systems and also takes an important step towards the goal of democratizing deep learning.

11.1 Future Work Related to CEREBRO

Model parallelism: In CEREBRO, we focused on DL models that fit in a single GPU memory. However, DL model sizes have grown at a faster rate than GPU memory sizes, and

some models no longer fit in single GPU memory [119]. One way to tackle this limitation is to partition a DL model across multiple GPUs and communicate updates between GPUs. This approach is called model parallelism. It has poor scalability with runtime speedups being quite sub-linear [151]. However, such large model training workloads are also not free from the need for model selection. Thus, model hopper parallelism style ideas can be combined with model parallelism to improve the system and resource efficiency of large model training workloads.

Model batching: At the other extreme of the model sizes, small DL models are also common (e.g., for IoT). Small DL models substantially under-utilize GPU capacity. Batching small models for training on the GPU can raise resource efficiency. While there is some recent work in this space, they modify the internals of the DL frameworks [217]. We believe multi-query optimization-inspired new approaches that intelligently batch DL models at the neural computation graph level can have broader adoption and impact.

More high-level APIs: In CEREBRO, we focussed on supporting basic model selection APIs, which can support hyperparameter tuning and architecture tuning. But as DL adoption grows, newer applications will emerge and new design patterns for model-building will appear. For example model building workloads for time-series or graph data have non-traditional data access patterns and manipulations to run SGD, e.g., configuring and chunking time windows in time series or retrieving features of a node's neighbors in a graph. These patterns may be twists on prior model selection patterns and/or contain new forms of sub-tasks that overlap in data and/or computation. It would be interesting future work to explore how the general API template we laid out for CEREBRO can be extended to support these workloads.

11.2 Future Work Related to HUMMINGBIRD

Tensors as an abstraction for general data processing: In project HUMMINGBIRD, we translated classical ML inference pipelines into tensor programs and leveraged DL compilers to

seamlessly execute them over heterogeneous hardware (e.g., CPUs, GPUs, ASICs). A natural question to ask is what other data processing workloads can be translated to tensor programs to benefit from the investments in DL systems. We believe graph processing and relational operators, two use cases very different from DL, in high demand, are two candidates that complement quite well with DL workloads. Much work is needed in understanding how to translate graph processing and relational operator algorithms into tensor computations [169].

Support for sparse tensor operations: We found that current support for sparse tensor operations in DL systems is primitive and not well optimized. Thus we restricted the set of target tensor operators supported in HUMMINGBIRD only to dense operations. However, these dense operations can incur non-trivial overheads due to the redundancies introduced in the translation process. As DL systems mature and improve the support for sparse tensor operations [164], it would be interesting future work to explore how sparse tensor operations can be leveraged to further improve the benefits of HUMMINGBIRD.

11.3 Future Work Related to KRYPTON

IVM-friendly CNNs: In KRYPTON, we use IVM-based incremental inference as a post-hoc optimization to accelerate CNN inference. Going further, "IVM-friendliness" can be baked into the very model selection process that crafts the CNN architecture so that the model is both accurate and amenable to fast explanations [208].

Other CNN workloads: Our IVM framework for propagating incremental changes through the layers of a CNN is highly general and can be used to support arbitrary workloads which require re-inference in the presence of incremental input changes. Thus, we believe our techniques may have applicability in several other CNN workloads, including object detection, image segmentation, and even video analytics [85, 82, 212].

Explainability beyond OBE and CNNs: Even though we focused on OBE workloads

in this work, we believe incremental inference-based techniques can have broader applicability across a variety of other feature perturbation-based explainability methods such as LIME [242] and SHAP [196]. Extending explanation support for other data modalities such as time-series [212] and graph data is also an interesting future direction.

11.4 Future Work Related to VISTA

Deep integration between PD and DL systems: VISTA currently supports one image per data example and a roster of popular CNNs. Nothing in Vista makes it difficult to relax these assumptions. For instance, supporting arbitrary CNNs requires static analysis of TF computational graphs. It would be interesting future work to extend VISTA to relax these assumptions.

Feature transfer from other types of neural models: Apart from image analytics, natural language processing (NLP) is another domain where feature transfer from pre-trained models has proven to be useful [245]. BERT [109] is one such popular pre-trained NLP model. Similar to CNNs, in order to pick the best layer, one has to explore multiple layer outputs. Also, aggregating features from multiple layers using concatenation or element-wise addition is also common. Thus, a feature layer in BERT depends on *multiple* input layers. VISTA can be extended to support such feature transfer workloads by generalizing our staged materialization plan to support arbitrary DAG architectures.

11.5 Future Work Related to NAUTILUS

Advanced model selection methods: In NAUTILUS, we assume that the model selection workload specification is fixed and is repeated for all model selection cycles. While this approach can support both grid and random search-based model selection, which covers an overwhelming majority of practical and research DTL workloads, it would be interesting future work to add

support for more complex model selection methods (e.g., ASHA [185], Hyperband [186]).

Advanced layer freezing schemes: We also assume that the layer freezing scheme used for adapting the source model is also fixed throughout the entire model training. However, few dynamic layer freezing schemes have also been proposed in the literature [144, 115, 99]. Techniques in NAUTILUS can be extended to support such dynamic layer freezing schemes.

Appendix A

Appendix: CEREBRO

A.1 CEREBRO API Usage Example

In this Section, we present a detailed example on how the CEREBRO API can be used to perform the *ImageNet* model selection workload explained in Section 3.6.

Before invoking the model selection workload users have to first register workers and data. This can be done as per the API methods shown in Figure A.1, A.2 and A.3.

```
#### API method to register workers ###
# worker_id:    Id of the worker
# ip       :    Worker IP
#
# Example usage:
#   register_worker(0, 10.0.0.1)
#   register_worker(1, 10.0.0.2)
#   ....
#   register_worker(7, 10.0.0.8)
#######################################
register_worker(worker_id, ip)
```

Figure A.1: Registering Workers.

Next, users need to define the initial set of training configurations as shown in Figure A.4. Users also need to define three functions: $input\_fn$, $model\_fn$, and $train\_fn$. $input\_fn$

```
## API method to register a dataset ###
# name     :    Name of the dataset
# num_partitions: # of partitions
#
# Example usage:
#   register_dataset("ImageNet", 8)
#######################################
register_worker(name, num_partitions)
```

Figure A.2: Registering Dataset.

```
## API method to register partition ###
#  availability
# dataset_name  : Name of the dataset
# partition_id  : Id of the partition
# worker_id     : Id of the worker
# file_path     : File path on the
#                 worker
#
# Example usage:
#   register_partition("ImageNet",
#                      'train',
#                      0, 0,
#                      "/imagnet/trn_0")
#######################################
register_partition(dataset_name,
                   data_type,
                   partition_id,
                   worker,
                   file_path)
```

Figure A.3: Registering Partitions.

as shown in Figure A.5, takes in the file path of a partition, performs pre-processing, and returns in-memory data objects. Inside the *input_fn* users are free to use their preferred libraries and tools provided they are already installed on the worker machines. These in-memory data objects are then cached in the worker's memory for later usage.

After the data is read into the worker's memory, CEREBRO then launches the model selection workload. This is done by launching training units on worker machines. For this CEREBRO first invokes the user defined *model_fn*. As shown in Figure A.6, it takes in the

```
S = []
for batch_size in [64, 128]:
    for lr in [1e-4, 1e-5]:
        for reg in [1e-4, 1e-5]:
            for model in ['ResNet', 'VGG']:
                config = {
                    'batch_size': batch_size,
                    'learn_rate': lr,
                    'regularization': reg,
                    'model': model
                }
                S.append(config)
```

Figure A.4: Initial Training Configurations.

```
#### User defined input function  #####
# file_path:    File path of a local
#               data partition
#
# Example usage:
#   processed_data = input_fn(file_path)
########################################
def input_fn(file_path):
    data = read_file(file_path)
    processed_data = preprocess(data)
    return processed_data
```

Figure A.5: User-defined input function.

training configuration as input and initializes the model architecture and training optimizer based on the configuration parameters. Users are free to use their preferred tool for defining the model architecture and the optimizer. After invoking the *model_fn*, CEREBRO injects a checkpoint restore operation to restore the model and optimizer state from the previous checkpointed state.

After restoring the state of the model and the optimizer, CEREBRO then invokes the user provided *train_fn* to perform one sub-epoch of training. As shown in Figure A.7, it takes in the data, model, optimizer, and training configuration as input and returns convergence metrics. Training abstractions used by different deep learning tools are different and this function abstracts it out from the CEREBRO system. After the *train_fn* is complete the state of the model and the optimizer is checkpointed again.

```
#### User defined model function  #####
# config:    Training configuration
#
# Example usage:
#   model, opt = model_fn(config)
########################################
def model_fn(config):
    if config['model'] == 'VGG':
        model = VGG()
    else:
        model = ResNet()

    opt = Adam(lr=config['learning_rate'])
    return model, opt
```

Figure A.6: User-defined model function.

```
#### User defined train function  #####
# data     : Preprocessed data
# model    : Deep learning model
# optimizer: Training optimizer
# config   : Training config
#
# Example usage:
#   loss = train_fn(data, model, optimizer,
#                   config)
########################################
def train_fn(data, model, optimizer, config)
    X, Y = create_batches(data,
                          config['batch_size'])
    losses = []
    for batch_x, batch_y in (X, Y):
        loss = train(model, opt,
                     [batch_x, batch_y])
        losses.append(loss)
    return reduce_sum(losses)
```

Figure A.7: Train function.

For evaluating the models, we assume the evaluation dataset is also partitioned and perform the same process. We mark the model parameters as non-trainable before passing it to the *train_fn*. After a single epoch of training and evaluation is done, CEREBRO aggregates the convergence metrics from all training units from the same configuration to derive the epoch-level convergence metrics. Convergence metrics are stored in a configuration state object which keeps

track of the training process of each training configuration. At the end of an epoch, configuration state objects are passed to the *automl_mthd* implementation for evaluation. It returns a set of configurations that needs to be stopped and/or the set of new configurations to start. For example in the case of performing Grid Search for 10 epochs, the *automl_mthd* will simply check whether an initial configuration has been trained for 10 epochs, and if so it will mark it for stopping.

A.2 CNN Compute Costs

Table A.1 lists the computational costs of the CNNs used for the simulation experiment in Section 3.5.4, which compares different scheduling methods. These costs were obtained from a publicly available benchmark[1].

A.3 Straggler Issue in Celery

One potential issue that could impact task-parallel systems' performance is load balancing. Given the large variance of runtimes for deep-nets training, the scheduling generated by Celery could lead to severe straggler issues that impairs the end-to-end runtime of the whole workload. On the other hand, CEREBRO suffers far less from this problem because it operates on a finer granularity; our tasks are chunked into sub-epochs and hence it is less likely for long-running stragglers to appear.

We take the Criteo tests showed in Section 3.6.1 as example. Without any prior or domain knowledge, it is impossible to know the runtime of each task before-hand and therefore Celery could schedule a plan like Figure A.8. The execution suffers from the straggler config#0 and needs 27.4 hrs to run.

However, if with a proper estimation/profiling of the runtimes/workloads, it is possible to fix this straggler issue with a carefully curated schedule as showed in Figure A.9. This schedule

[1] https://github.com/albanie/convnet-burden

Table A.1: Computation costs of the CNNs used for the simulation experiment comparing different scheduling methods.

Model	FLOPs
AlexNet	727 MFLOPs
CaffeNet	724 MFLOPs
SqueezeNet1-0	837 MFLOPs
SqueezeNet1-1	360 MFLOPs
VGG-f	727 MFLOPs
VGG-m	2 GFLOPs
VGG-s	3 GFLOPs
VGG-m-2048	2 GFLOPs
VGG-m-1024	2 GFLOPs
VGG-m-128	2 GFLOPs
VGG-vd-16-atrous	16 GFLOPs
VGG-vd-16	16 GFLOPs
VGG-vd-19	20 GFLOPs
GoogleNet	2 GFLOPs
ResNet18	2 GFLOPs
ResNet34	4 GFLOPs
ResNet50	4 GFLOPs
ResNet101	8 GFLOPs
ResNet152	11 GFLOPs
ResNext-50-32x4d	4 GFLOPs
ResNext-101-32x4d	8 GFLOPs
ResNext-101-64x4d	16 GFLOPs
Inception-V3	6 GFLOPs
SE-ResNet-50	4 GFLOPs
SE-ResNet-101	8 GFLOPs
SE-ResNet-152	11 GFLOPs
SE-ResNeXt-50-32x4d	4 GFLOPs
SE-ResNeXt-101-32x4d	8 GFLOPs
SENet	21 GFLOPs
SE-BN-Inception	2 GFLOPs
DenseNet121	3 GFLOPs
DenseNet161	8 GFLOPs
DenseNet169	3 GFLOPs
DenseNet201	4 GFLOPs
MobileNet	579 MFLOPs

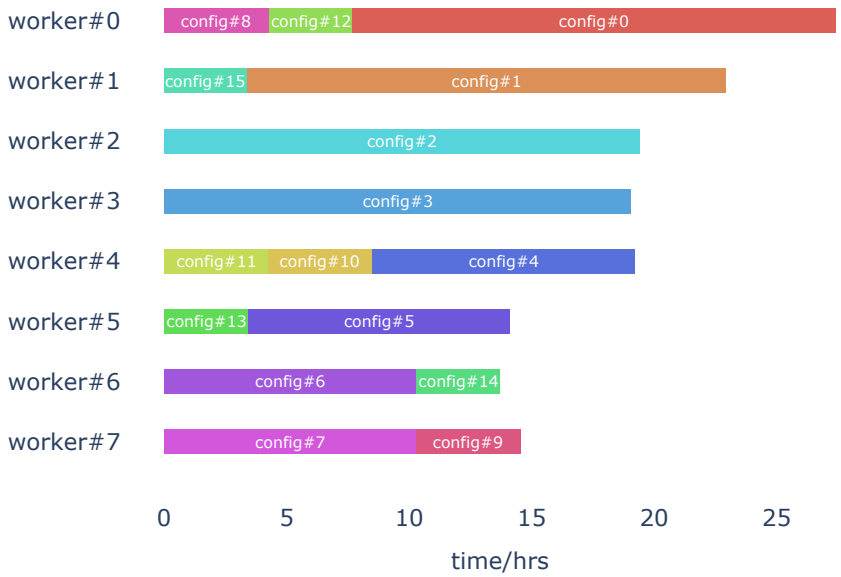


Figure A.8: An unbalanced work schedule generated by Celery for Criteo tests.

drastically reduces the runtime to 19.7 hrs.

In Section 3.6.1, we decided to show the runtime with the best-possible scheduling for Celery, as we do not wish to unfairly punish the adversarial systems, and load balancing/runtime estimation of deep learning workloads are out of the scope of this work. We believe these decisions can ultimately help the reader focus on the benefits and advantages of our system.

A.4 AutoML Procedures

A.4.1 Experiments with HyperBand

We compare CEREBRO and Celery for executing Hyperband [186]; a popular AutoML search procedure. We use *ImageNet*, GPU cluster, and PyTorch. Training configs are randomly sampled from the grid shown in Table 3.6. For CEREBRO each data partition is only available on one worker; for Celery the dataset is fully replicated.

Hyperband combines random search with early stopping. It starts with a fixed set of model configs and trains them for a given number of epochs in the first "rung." After completion,

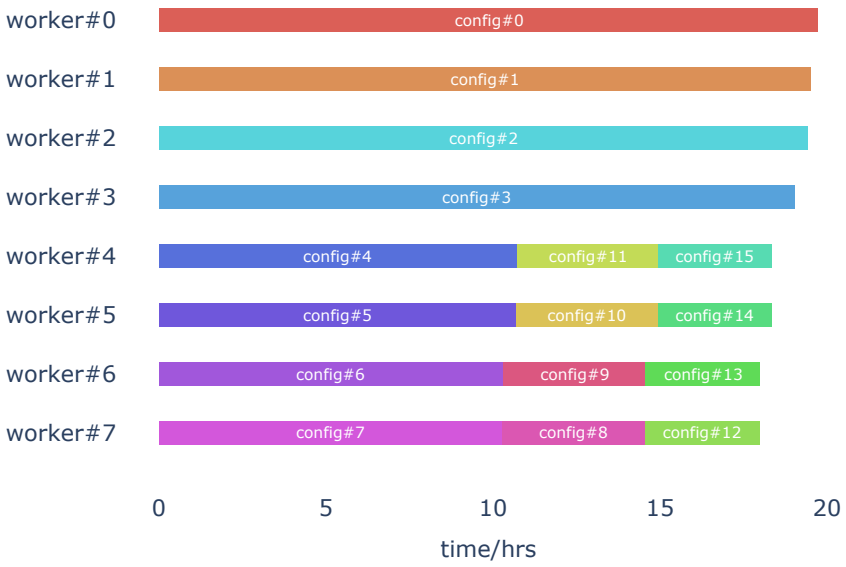


Figure A.9: Best possible work schedule with Celery for Criteo tests.

it picks a subset of the best models and promotes them to the next rung; this is repeated several times until a max epoch budget is hit. We run an experiment with a max resource budget (R) of 25 epochs and a downsampling rate (η) of 3, two parameters from the Hyperband paper. Figure A.10 compares the learning curves of the configs run by Hyperband on CEREBRO and Celery.

We see that CEREBRO and Celery have almost indistinguishable convergence behaviors, validating our claim that MOP benefits from SGD's robustness to random data ordering. As Figure A.11 shows, both systems have similar completion times (42.1hr for Celery; 43.5hr for CEREBRO). Some configs finish sooner on Celery than their counterparts on CEREBRO. This is because CEREBRO's per-epoch scheduling template enforces all configs to be scheduled for the same epoch. But in Celery, configs in their last rung can finish earlier without waiting for other configs.

A.4.2 Experiments with ASHA

ASHA combines random search with early stopping. It starts with a set of model configs and trains them for a fixed number of epochs in the first "rung." After a training config finishes its

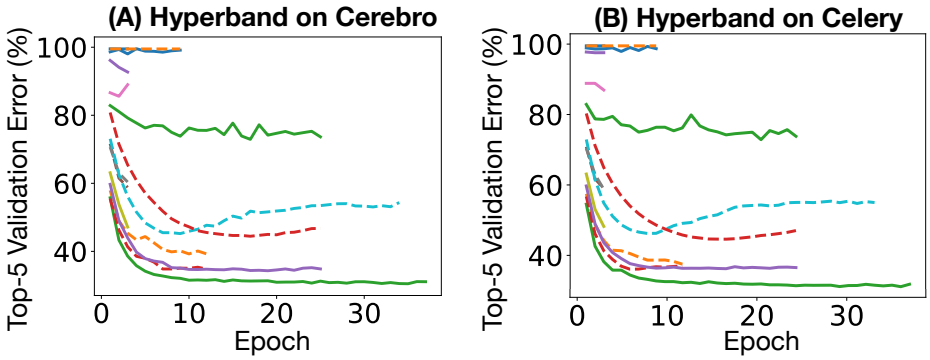


Figure A.10: Hyperband learning curves by epochs.

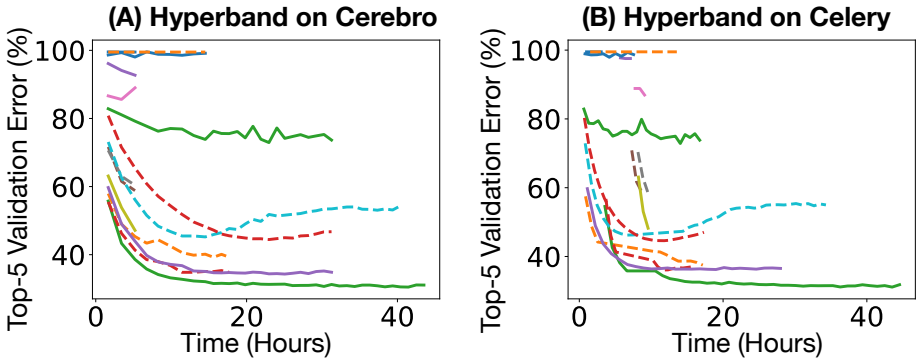


Figure A.11: Hyperband learning curves by time.

current rung, it is assigned to a pool of completed configs for that rung. ASHA will then pick a promising config from this pool based on a selection fraction (η) and promote it to the next rung for training. If the selection fraction is already exhausted, a new config will be created and trained for the first rung. A model can be promoted between rungs until it is trained for a maximum number of rungs/epochs. This process is continued until the allocated time budget for model selection workload is reached.

ASHA's decisions on configs are dependent on the wallclock completion order of configs across task-parallel workers. Thus, it is impossible to exactly replicate a run of ASHA on task-parallelism in CEREBRO. However, we can indeed emulate ASHA on CEREBRO without making any changes to CEREBRO. We run an experiment using ASHA with a max resource budget (R) of 9 epochs, a selection fraction (η) of 3, and a time limit of 24hr.

In the given time limit, ASHA on CEREBRO (resp. Celery) explored 83 (resp. 67) configs.

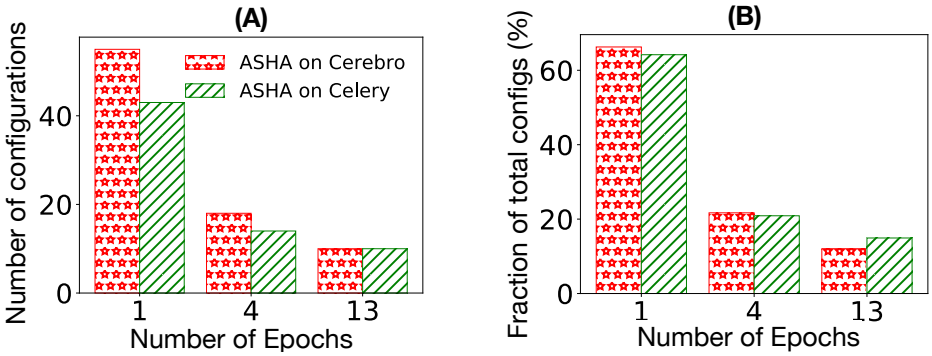


Figure A.12: Number of configs vs. the amount of epochs they were run for by. (A) Count of configs and (B) Fraction of total config count.

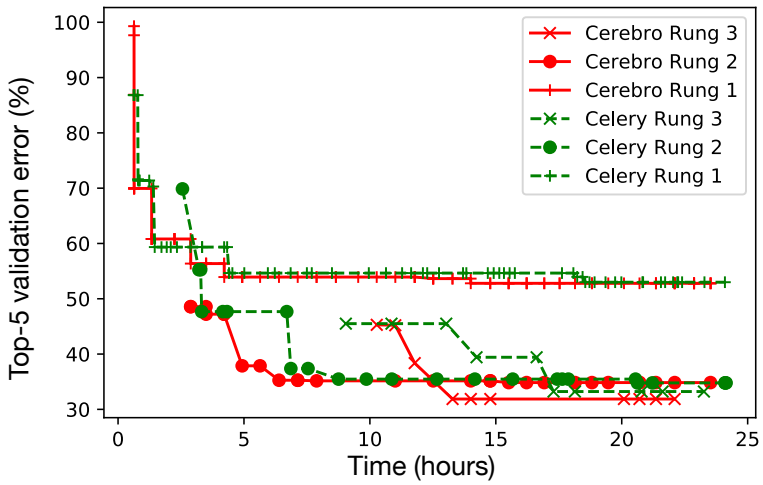


Figure A.13: Best validation error for each rung of ASHA.

Figure 3.13 shows all learning curves. Though the individual configs are not comparable across the two systems, the best errors achieved are close (31.9% on CEREBRO ; 33.2% on Celery). A serendipity is that ASHA-on-CEREBRO seems to perform slightly *better* than the regular task-parallel version in the ASHA! We believe this is because the epoch-level synchronization in CEREBRO actually helps ASHA pick and promote better configs due to its knowledge of a larger set of configs. Regular ASHA gains this knowledge spread over time, which makes it prone to more wrong promotions. Figure A.12 confirms our intuition: ASHA-on-CEREBRO explores more configs in the lower rungs than regular ASHA. Also, as Figure A.13 shows, ASHA-on-CEREBRO reaches lower errors for all rungs sooner than regular ASHA. We leave a more rigorous

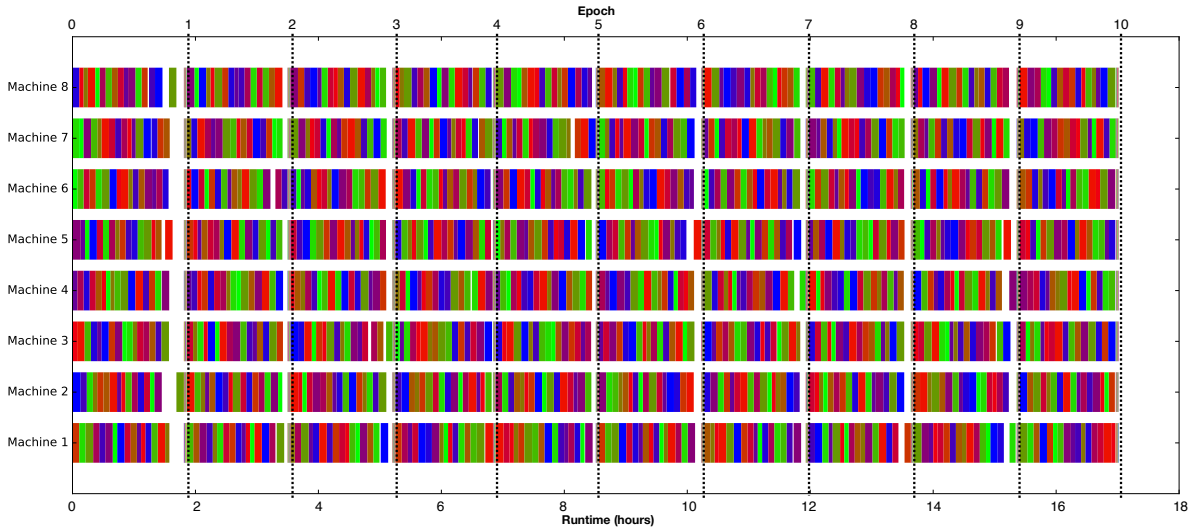


Figure A.14: Gantt chart corresponding to the schedule produced by CEREBRO for the *ImageNet* workload. Each color corresponds to a different training configuration. Best viewed in color.

statistical analysis of this apparent superiority of ASHA-on-CEREBRO over regular ASHA to future work.

A.5 Gantt Chart

Figure A.14 presents the Gantt chart corresponding to the scheduler produced by CEREBRO for the *ImageNet* workload. Each color bar corresponds to a different training configuration (16 in total).

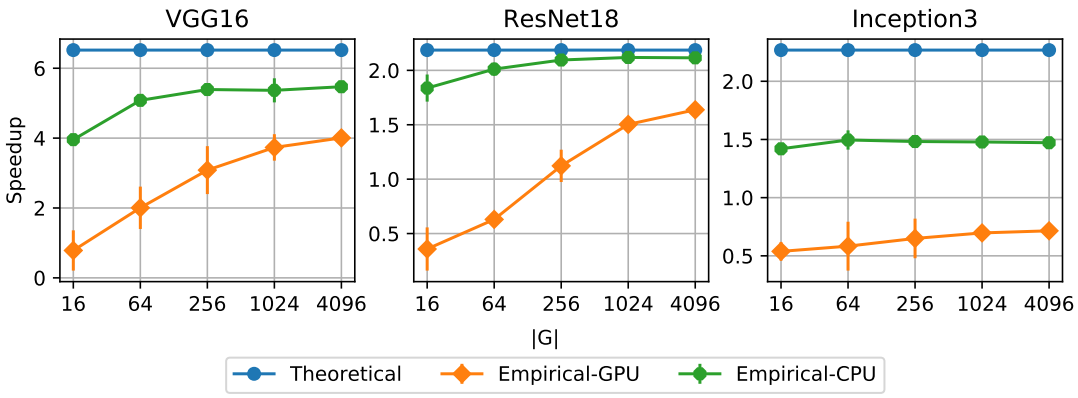


Figure B.1: Interactive mode execution of incremental inference with *G*s of different sizes

Appendix B

Appendix: KRYPTON

B.1 Interactive Mode Execution

We evaluate interactive-mode incremental inference execution (no approximate inference optimizations) with *G*s of different sizes. Similar to non-interactive mode experiments presented in Section 6.5, all experiments are run in batched mode with a batch size of 16 for CPU based experiments and a batch size 128 for GPU based experiments. If the size of G (formally $|G|$) or the remainder of G is smaller than the batch size, that value is used as the batch size (e.g. $|G| = 16$ results in a batch size of 16). Figure B.1 presents the final results.

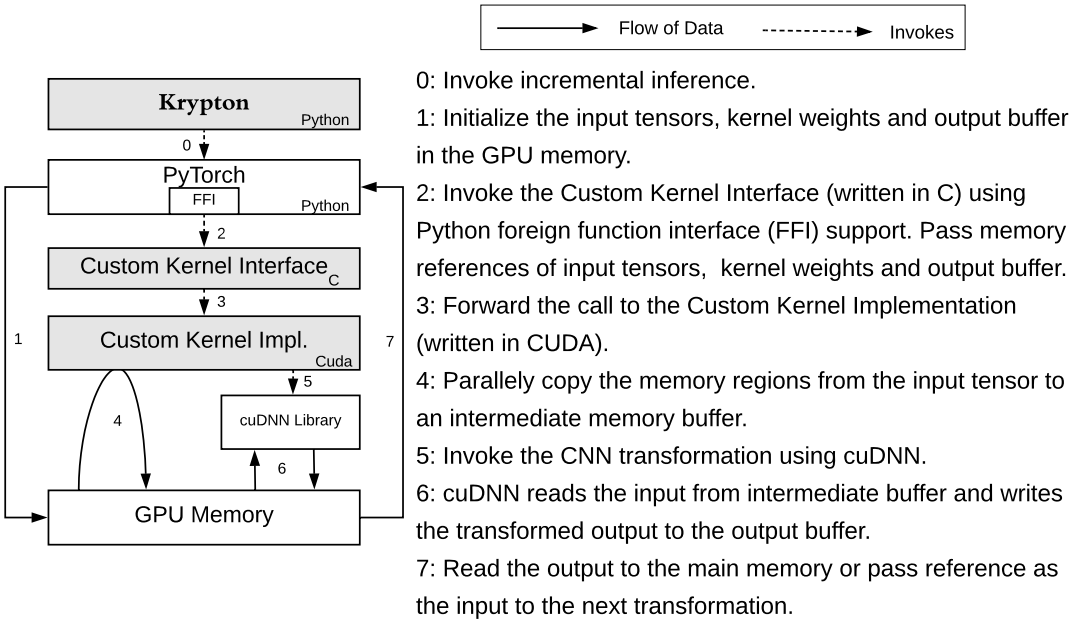


Figure B.2: Custom GPU Kernel integration architecture

B.2 Integration into PyTorch

For the CPU environment we implemented KRYPTON purely on top of the PyTorch toolkit using it's tensor slicing and stitching capabilities as per Algorithm 6. However, for the GPU environment such iterative memory copying operations introduce high overheads as the many GPU cores now have to idle wait for the slow memory copy operations. To overcome this we extended PyTorch by adding a custom GPU kernel which optimizes the input preparation for *incremental inference* by invoking parallel memory copy operations. This custom kernel is integrated to PyTorch using Python foreign function interface (FFI). Python FFI integrates with the Custom Kernel Interface layer which then invokes the Custom Memory Copy Kernel Implementation. The high-level architecture of the Custom Kernel integration is shown in Figure B.2.

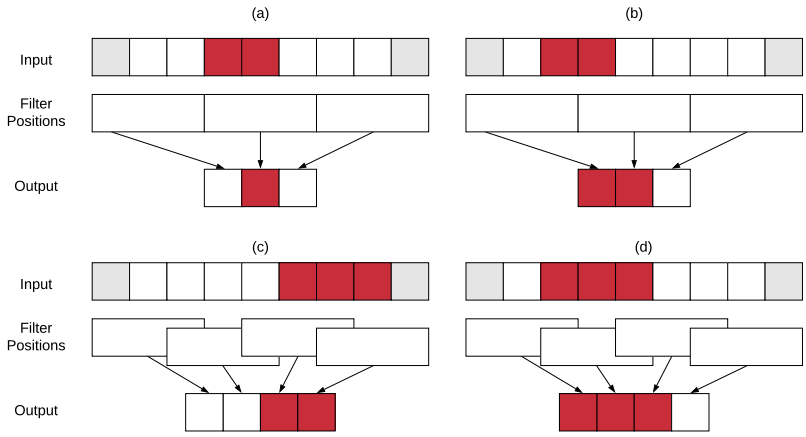


Figure B.3: Illustration of special cases for which actual output size will be smaller than the value given by Equation (6.13). (a) and (b) show cases where the filter stride is equal to the filter size. (c) and (d) show situations where the position of the modified patch affecting the size of the output patch.

B.3 Special Cases for Incremental Inference

There are special cases under which the output patch size can be smaller than the values calculated in Section 6.3.2. Consider the simplified 1-D case shown in Figure B.3 (a), where the filter stride[1] (3) is the same as the filter size (3). In this case, the size of the output update patch is one less than the value calculated by Equation (6.14). But this is not the case for the situation shown Figure B.3 (b), which has the same input patch size but placed at a different location. Another case arises when the modified patch is placed at the edge of the input, as shown in Figure B.3 (c). In this case, it is impossible for the filter to move freely through all positions, since it hits the input boundary. However, it is not the case for the modified patch shown in Figure B.3 (d). In KRYPTON, we do not treat these cases separately but rather use the values calculated by Equation (6.14) for the width dimension (similarly for the height dimension), since they act as an upper bound. In the case of a smaller output patch, KRYPTON reads and updates a slightly bigger patch to preserve uniformity. This also requires updating the starting coordinates of the patch, as shown in Equation (B.1). This sort of uniform treatment is required for performing

[1]Note that stride is typically less than or equal to filter size.

batched inference operations, which gives significant speedups compared to per-image inference.

$$\begin{aligned} &\text{If } x_{\mathcal{P}}^{O} + W_{\mathcal{P}}^{O} > W_O: \\ &x_{\mathcal{P}}^{O} = W_O - W_{\mathcal{P}}^{O}; x_{\mathcal{P}}^{I} = W_I - W_{\mathcal{P}}^{I}; x_{\mathcal{P}}^{\mathcal{R}} = W_I - W_{\mathcal{P}}^{\mathcal{R}} \end{aligned} \tag{B.1}$$

B.4 Effective Projective Field Size

We formalize the effective projective field growth for the one dimensional scenario with n convolution layers (assuming certain conditions). This proof is motivated by a similar proof in [197] which characterizes the effective growth rate of the receptive field in a CNN.

The input is $u(t)$ where

$$u(t) = \begin{cases} 1, & t = 0 \\ 0, & t \neq 0 \end{cases} \tag{B.2}$$

and $t = 0, 1, -1, 2, -2, ...$ indexes the input pixels.

Each layer has the same kernel $v(t)$ of size k. The kernel signal can be formally defined as

$$v(t) = \sum_{m=0}^{k-1} w(m)\delta(t-m) \tag{B.3}$$

where $w(m)$ is the weight for the m^{th} pixel in the kernel. Without loosing generality, we can assume the weights are normalized, i.e. $\sum_m w(w) = 1$. The output signal of the n^{th} layer $o(t)$ is simply $o = u * v * ... * v$, convolving u with n such vs. To compute the convolution, we can use the

Discrete Time Fourier Transform to convert the signals into the Fourier domain, and obtain

$$\begin{aligned} U(\omega) &= \sum_{t=-\infty}^{\infty} u(t)e^{-j\omega t} = 1,\ V(\omega) \\ &= \sum_{t=-\infty}^{\infty} v(t)e^{-j\omega t} = \sum_{m=0}^{k-1} w(m)e^{-j\omega t} \end{aligned} \tag{B.4}$$

Applying the convolution theorem, we have the Fourier transform of o is

$$\begin{aligned} \mathcal{F}(o) &= \mathcal{F}(u * v * ... * v)(\omega) = U(\omega).V(\omega)^n \\ &= \left(\sum_{m=0}^{k-1} w(m)e^{-j\omega t} \right)^n \end{aligned} \tag{B.5}$$

With inverse Fourier transform

$$o(t) = \frac{1}{2\pi} \int_{-\pi}^{\pi} \Big(\sum_{m=0}^{k-1} w(m)e^{-j\omega t} \Big)^n e^{j\omega t} d\omega \tag{B.6}$$

The space domain signal $o(t)$ is given by the coefficients of $e^{-j\omega t}$. These coefficients turn out to be well studied in the combinatorics literature [112]. It can be shown that if $\sum_m w(m) = 1$ and $w(m) \geq 0 \ \forall \ m$, then

$$\begin{aligned} o(t) &= p(S_n = t) \\ \text{where } S_n &= \sum_{i=1}^{n} X_i \text{ and } p(X_i = m) = w(m) \end{aligned} \tag{B.7}$$

From the central limit theorem, as $n \to \ \infty$, $\sqrt{n}(\frac{1}{n}S_n - \mathbb{E}[X]) \sim \mathcal{N}(0, Var[X])$ and $S_n \sim \mathcal{N}(n\mathbb{E}[X]), nVar[X])$. As $o(t) = p(S_n = t)$, $o(t)$ also has a Gaussian shape with

$$\mathbb{E}[S_n] = n \sum_{m=0}^{k-1} mw(m) \tag{B.8}$$

$$Var[S_n] = n \left(\sum_{m=0}^{k-1} m^2 w(m) - \Big(\sum_{m=0}^{k-1} mw(m) \Big)^2 \right) \tag{B.9}$$

Table B.1: Train-validation-test split size for each dataset.

	Train	Validation	Test
OCT	50,382	16,853	16, 857
Chest X-Ray	3,463	1,237	1,156

This indicates that $o(t)$ decays from the center of the projective field squared exponentially according to the Gaussian distribution. As the rate of decay is related to the variance of the Gaussian and assuming the size of the effective projective field is one standard deviation, the size can be expressed as

$$\sqrt{Var[S_n]} = \sqrt{nVar[X_i]} = O(\sqrt{n}) \tag{B.10}$$

On the other hand stacking more convolution layers would grow the theoretical projective field linearly. But the effective projective field size is shrinking at a rate of $O(1/\sqrt{n})$.

B.5 Fine-tuning CNNs

For *OCT* and *Chest X-Ray*, the 3 ImageNet-trained CNNs are fine-tuned by retraining the final Fully-Connected layer. We use a train-validation-test split of 60-20-20 and the exact numbers for each dataset are shown in Table B.1. Cross-entropy loss with L2 regularization is used as the loss function and Adam [162] is used as the optimizer. We tune learning rate $\eta \in [10^{-2}, 10^{-4}, 10^{-6}]$ and regularization parameter $\lambda \in [10^{-2}, 10^{-4}, 10^{-6}]$ using the validation set and train for 25 epochs. Table B.2 shows the final train and test accuracies.

B.6 Memory Overhead of IVM

We evaluate the memory overhead of IVM approach, with no projective field thresholding ($\tau = 1.0$) and a projective field thresholding value of $\tau = 0.6$, compared to the full CNN inference.

Table B.2: Train and test accuracies after fine-tuning.

	Model	Accuracy(%)		Hyperparams.	
		Train	Test	η	λ
OCT	VGG16	79	82	10^{-4}	10^{-4}
	ResNet18	79	82	10^{-2}	10^{-4}
	Inception3	71	81	10^{-2}	10^{-6}
Chest X-Ray	VGG16	75	76	10^{-4}	10^{-4}
	ResNet18	78	76	10^{-4}	10^{-6}
	Inception3	74	76	10^{-4}	10^{-2}

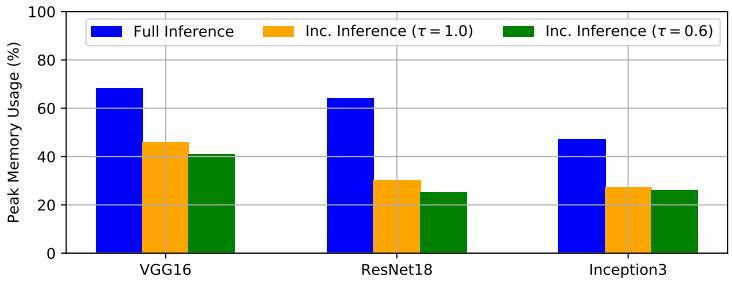


Figure B.4: Peak GPU memory usage when performing CNN inference on a batch of 128 images.

For this we record the peak GPU memory utilization while the CNN models perform inference on image batches of size 128. We found that incremental inference approach can enable up to 58% lower memory overhead (see Figure B.4). Krypton materializes a single copy of all CNN layers corresponding to the unmodified image and reuses it across a batch of occluded images with IVM. For IVM the size of required memory buffers are much smaller than the full inference as only the updated patches need to be propagated.

B.7 Visual Examples

Figure B.5 presents occlusion heat maps for a sample image from each dataset with (a) *incremental inference* for different *projective field threshold* values and (b) *incremental inference* with *adaptive drill-down* for different *projective field threshold* values. The predicted class label

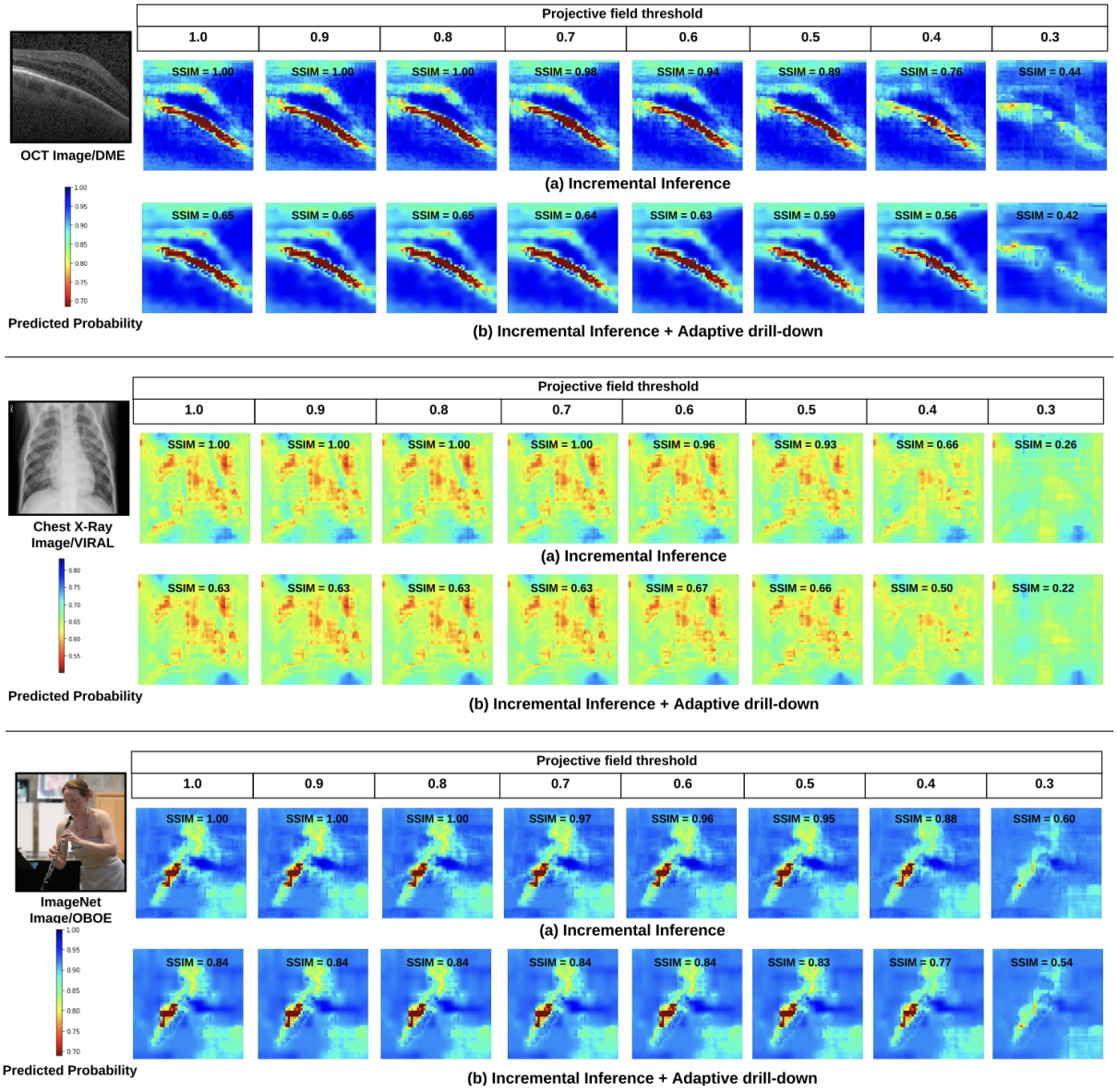


Figure B.5: Occlusion heat maps for sample images (CNN model = VGG16, occlusion patch size = 16, patch color = black, occlusion patch stride $(S \text{ or } S_2)$ = 4. For *OCT* $r_{drill\_down} = 0.1$ and $target = 5$. For *Chest X-Ray* $r_{drill\_down} = 0.4$ and $target = 2$. For *ImageNet* $r_{drill\_down} = 0.25$ and $target = 3$). For a projective field threshold value of 0.3 we see significant degradation of heat map quality due to the significant information loss from truncation.

for *OCT*, *Chest X-Ray*, and *ImageNet* are DME, VIRAL, and OBOE respectively.

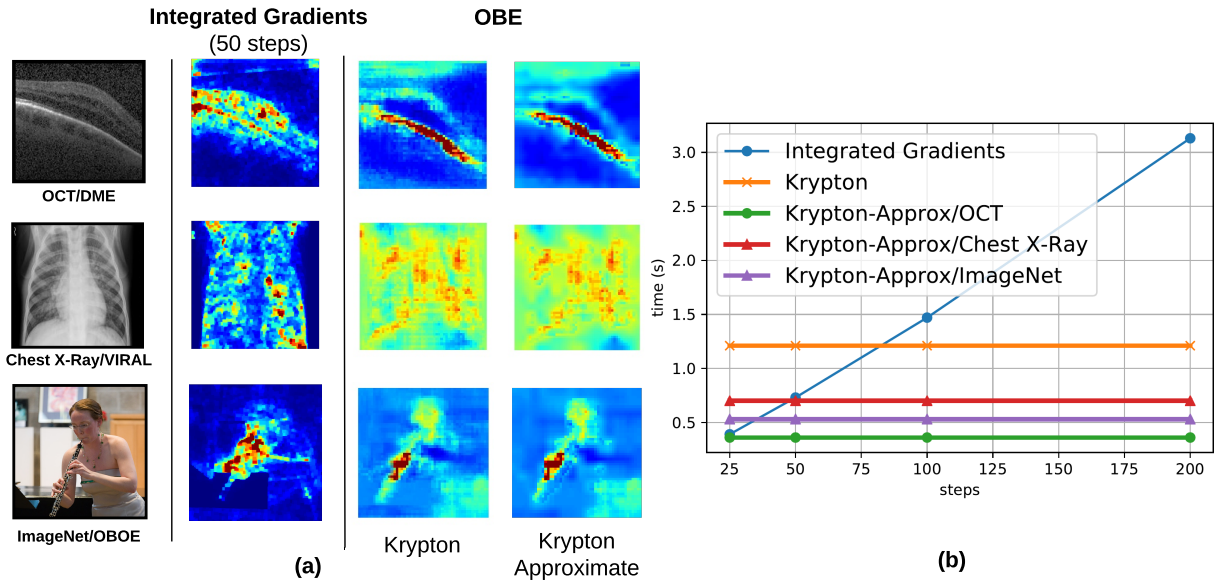


Figure B.6: Comparison of integrated gradients method against OBE. (a) Heat maps generated by integrated gradients method with a step size of 50. The three color channel gradients of a pixels at the same point are aggregated using L2 norm

B.8 Integrated Gradients Method

We evaluate the runtime and visual quality of the generated heat maps for integrated gradients (IGD) [266] and OBE methods on three representative images from our datasets (see Figure B.6). In general, OBE can better localize relevant regions from the input images. IGD method requires tuning a hyper-parameter called *steps* which determines the number steps to be used in the gradient integration approximation. Increasing steps improves both the runtime and heat map quality of the IGD method. For performing OBE we used the same hyper-parameters that were used in Section 6.5.1.

Appendix C

Appendix: VISTA

C.1 Estimating Intermediate Data Sizes

We explain the size estimations in the context of Spark. Ignite also uses an internal format similar to the Spark. Spark's internal binary record format is called "Tungsten record format," shown in Figure C.2. Fixed size fields (e.g., float) use 8 B. Variable size fields (e.g., arrays) have an 8 B header with 4 B for the offset and 4 B for the length of the data payload. The data payload is stored at the end of the record. An extra bit tracks null values.

VISTA estimates the size of intermediate tables $T_l \ \forall l \in L$ in Figure 8.5(E) based on its knowledge of the CNN. For simplicity, assume *ID* is a long integer and all features are single precision floats. Let $|X|$ denote the number of features in X. $|T_{str}|$ and $|T_{img}|$ are straightforward to calculate, since they are the base tables. For $|T_i|$ with feature layer $l = L[i]$, we have:

$$|T_i| = \alpha_1 \times (8 + 8 + 4 \times |g_l(\hat{f}_l(I))|) + |T_{str}| \tag{C.1}$$

Equation C.1 assumes deserialized format; serialized (and compressed) data will be smaller. But these estimates suffice as safe upper bounds.

Figure C.3 shows the estimated and actual sizes. We see that the estimates are accurate for

```
/**
 * * * * * * * * * * * * * * * * * * Input Parameters * * * * * ** * * * * *
 * name         : Name given to the experiment
 * mem_sys      : System memory available on a worker node
 * n_nodes      : # of nodes in the cluster
 * cpu_sys      : # of CPUs available on a worker node
 * model        : CNN model name. Possible values
 * n_layers     : # of layers from the last layer of the CNN to be explored
 * start_layer  : Starting layer of the ConvNet.
 * ml_func      : Function pointer which implements the downstream ML model
 * struct_input : Input path for the strucutred input
 * images_input : Input path for the images
 * n            : # of records
 * dS           : # of structured features
**/
vista = Vista("vista-example", 32, 8, 8, 'alexnet', 4, 0, ml_func,
            'hdfs://../foods.csv', 'hdfs://.../foods-images', 20129, 130)


//Initiate CNN feature transfer workload. Function returns the training
//accuracies for each evaluated layer
train_accuracies = vista.run()
```

Figure C.1: VISTA API and sample usage showing values for the input parameters and invocation.

the deserialized in-memory data with a reasonable safety margin. Interestingly, *Eager* is not that much larger than *Staged* for AlexNet. This is because among its four layers explored the 4^{th} layer from the top is disproportionately large while for the other two layer sizes are more comparable. Serialized is smaller than deserialized as Spark compresses the data. Interestingly, AlexNet feature layers seem more compressible; we verified that its features had many zero values. On average, AlexNet features had only 13.0% non-zero values while VGG16's and ResNet50's had 36.1% and 35.7%, respectively.

C.2 Pre Materializing a Base Layer

Often data scientists are interested in exploring few of the top most layers. Hence a base layer can pre-materialized before hand for later use of exploring other layers. This can save computations and thereby reduce the runtime of the CNN feature transfer workload.

However, the CNN feature layer sizes (especially for conv layers) are generally larger

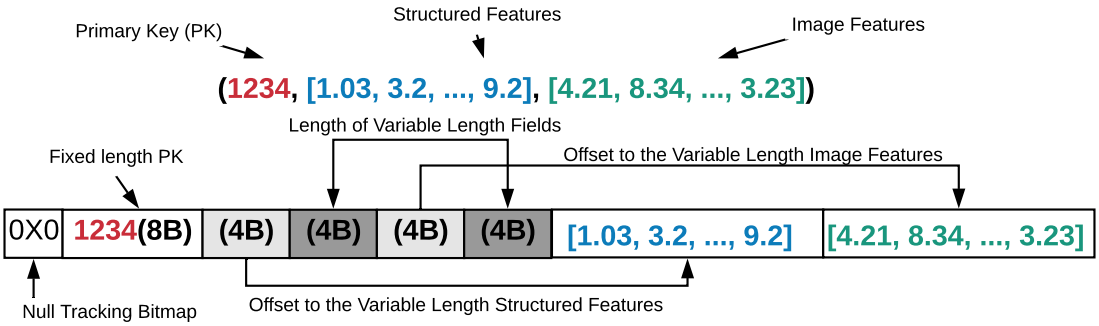


Figure C.2: Spark's internal record storage format.

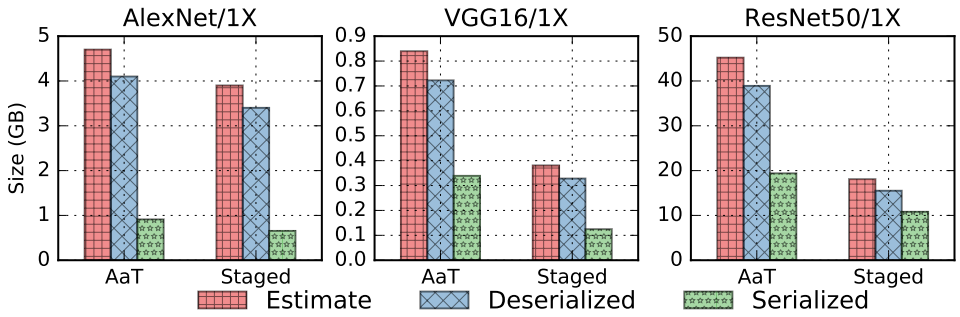


Figure C.3: Size of largest intermediate table.

Table C.1: Sizes of pre-materialized feature layers for the Foods dataset (size of raw images is 0.26 GB).

	Materialized Layer Size (GB) (layer index starts from the last layer)			
	1^{st}	2^{nd}	4^{th}	5^{th}
AlexNet	0.08	0.14	0.72	
VGG16	0.08	0.20	1.19	
ResNet50	0.08	2.65	3.45	11.51

than the compressed image formats such as JPEG (see Table C.1). This not only increases the secondary storage requirements but also increases the IO cost of the CNN feature transfer workload both when initially reading data from the disk and during join time when shuffling data over the network.

We perform a set of experiments using the Spark-TF system to explore the effect of pre-materializing a base layer (1, 2, 4, and 5^{th} layers from top). For evaluating the ML model for

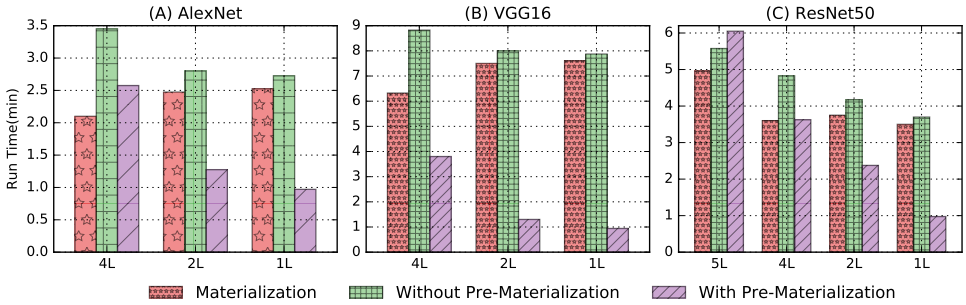


Figure C.4: Runtimes comparison for using pre-materialized features from a base layer

the base layer no CNN inference is required. But for the other layers partial CNN inference is performed starting from the base layer using the *Staged/After Join/Deserialized/Shuffle* logical-physical plan combination. Experimental set up is same as in Section 8.4.3.

For AlexNet and VGG16 when materializing 4^{th}, 2^{nd}, and 1^{st} layers from the top, the materialization time increases as evaluating higher layer requires more computations (see Figure. C.4 (A) and (B)). However, for ResNet50 there is a sudden drop from the materialization time of 5^{th} layer features to the materialization time of 4^{th} layer features. This can be attributed to the high disk IO overhead of writing out 5^{th} layer image features which are $\sim$3 times larger than that of 4^{th} layer (see Figure. C.4 (C)). Therefore, for ResNet50 starting from a pre-materialized feature layer, instead of raw images, may or may not decrease the overall CNN feature transfer workload runtime.

C.3 Runtime Breakdown

We drill-down into the time breakdowns of the workloads on Spark-TF environment and explore where the bottlenecks occur. In the downstream logistic regression (LR) model, the time spent for training the model on features from a specific layer is dominated by the runtime of the first iteration. In the first iteration partial CNN inference has to be performed starting either from raw images or from the image features from the layer below and the later iterations will be operating on top of the already materialized features. Input read time is dominated by reading

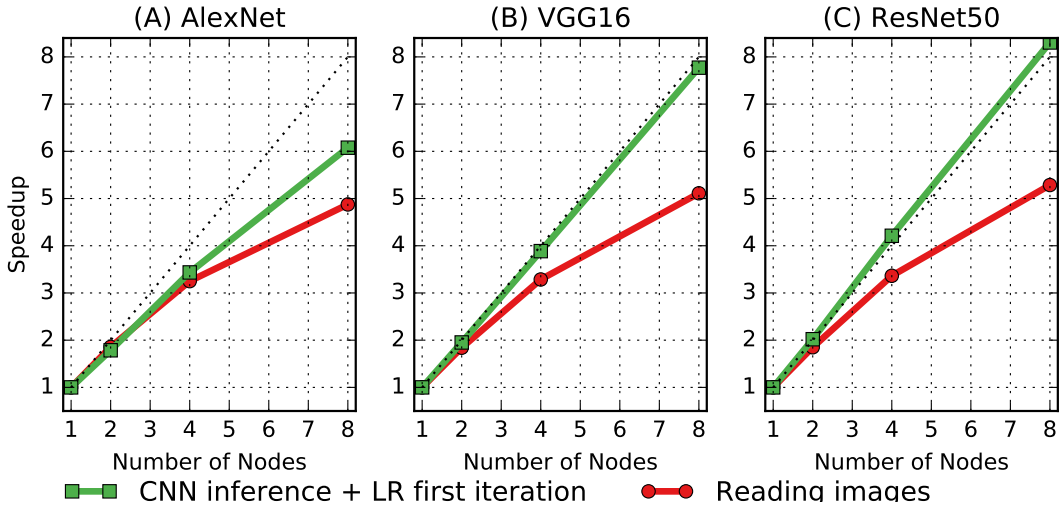


Figure C.5: Drill-down analysis of Speedup Curves.

Table C.2: Runtime breakdown for the image data read time and 1^{st} iteration of the logistic regression model (Layer indices starts from the top and runtimes are in minutes).

		ResNet50/5L				AlexNet/4L				VGG16/3L			
		Number of nodes				Number of nodes				Number of nodes			
		1	2	4	8	1	2	4	8	1	2	4	8
Layer	5	19.0	9.5	4.5	2.3								
	4	3.8	1.8	0.9	0.4	3.7	2.1	1.2	0.7				
	3	2.7	1.3	0.7	0.4	2.4	1.3	0.7	0.5	43.0	22.0	11.0	5.4
	2	2.6	1.3	0.6	0.3	1.1	0.6	0.3	0.2	1.0	0.5	0.3	0.2
	1	1.8	0.9	0.4	0.2	0.3	0.2	0.1	0.1	0.3	0.2	0.1	0.1
	total	29.9	14.8	7.1	3.6	7.5	4.2	2.3	1.5	44.3	22.7	11.4	5.7
Read images		3.7	2.0	1.1	0.7	3.9	2.1	1.2	0.8	4.6	2.5	1.4	0.9

images as there are lot of small files compared to the one big structured data file [22]. Table C.2 summarizes the time breakdown for the CNN feature transfer workload. It can be seen that most of the time is spent on performing the CNN inference and LR 1^{st} iteration on the first layer (e.g 5^{th} layer from top for ResNet50) where the CNN inference has to be performed starting from raw images.

We also separately analyze the speedup behavior for the input image reading and the sum of CNN inference and LR 1^{st} iteration times (see Figure C.5). When we separate out the sum of CNN inference and LR 1^{st} iteration times, we see slightly super linear speedups for ResNet50,

near linear speedups for VGG16, and slightly better sub-linear speedups for AlexNet.

Bibliography

[1] Adaptive Execution in Spark. https://issues.apache.org/jira/browse/SPARK-9850. Accessed March 31, 2020.

[2] AI Device for Detecting Diabetic Retinopathy Earns Swift FDA Approval. https://www.aao.org/headline/first-ai-screen-diabetic-retinopathy-approved-by-f. Accessed September 31, 2018.

[3] Basic Operations in a Convolutional Neural Network - CSE@IIT Delhi. http://www.cse.iitd.ernet.in/~rijurekha/lectures/lecture-2.pptx. Accessed September 31, 2018.

[4] Benchmarks for Popular CNN Models. https://github.com/jcjohnson/cnn-benchmarks. Accessed March 31, 2020.

[5] Big Data Analytics Market Survey Summary. https://www.forbes.com/sites/louiscolumbus/2017/12/24/53-of-companies-are-adopting-big-data-analytics/#4b513fce39a1. Accessed March 31, 2020.

[6] Cafee Model Zoo. https://github.com/BVLC/caffe/wiki/Model-Zoo. Accessed September 31, 2018.

[7] Cerebras Chip. https://www.wired.com/story/power-ai-startup-built-really-big-chip/.

[8] Distribution of Executors, Cores and Memory for a Spark Application running in Yarn. https://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application. Accessed March 31, 2020.

[9] Get Started with TensorFlow Transform. https://www.tensorflow.org/tfx/transform/get_started. Accessed March 31, 2020.

[10] Kaggle Survey: The State of Data Science and ML. https://www.kaggle.com/surveys/2017. Accessed March 31, 2020.

[11] Models and Examples Built with TensorFlow. https://github.com/tensorflow/models. Accessed September 31, 2018.

[12] Nautilus: An Optimized System for Deep Transfer Learning over Evolving Training Datasets - [Technical Report]. https://adalabucsd.github.io/papers/TR_2022_Nautilus.pdf. Accessed March 1, 2022.

[13] Nvidia RAPIDS. https://developer.nvidia.com/rapids.

[14] ONNX ML. https://github.com/onnx/onnx/blob/master/docs/Operators-ml.md.

[15] ONNX Runtime. https://github.com/microsoft/onnxruntime.

[16] ONNX Supported Frameworks and Backends. https://onnx.ai/supported-tools.html.

[17] Open Food Facts Dataset. https://world.openfoodfacts.org/. Accessed March 31, 2020.

[18] Parallel Postgres for Enterprise Analytics at Scale. https://pivotal.io/pivotal-greenplum. Accesses January 31, 2018.

[19] Spark Best Practices. http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/. Accessed March 31, 2020.

[20] Spark Memory Management. https://0x0fff.com/spark-memory-management/. Accessed March 31, 2020.

[21] SparkDL: Deep Learning Pipelines for Apache Spark. https://github.com/databricks/spark-deep-learning. Accessed March 31, 2020.

[22] The Small Files Problem of HDFS. http://blog.cloudera.com/blog/2009/02/the-small-files-problem/. Accessed March 31, 2020.

[23] Torch Vison Models. https://github.com/pytorch/vision/tree/master/torchvision/models. Accessed September 31, 2018.

[24] TorchScript Documentation. https://pytorch.org/docs/stable/jit.html.

[25] Transfer Learning with CNNs for Visual Recognition. http://cs231n.github.io/transfer-learning/. Accessed March 31, 2020.

[26] Efficient Evaluation of Queries with Mining Predicates. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 529–. IEEE Computer Society, 2002.

[27] CNTK. https://docs.microsoft.com/en-us/cognitive-toolkit/, 2018.

[28] MXNet. https://mxnet.apache.org/, 2018.

[29] ESG Technical Validation: Dell EMC Ready Solutions for AI: Deep Learning with Intel. https://tinyurl.com/2p94kcrd, 2019.

[30] Graphcore IPU. https://www.graphcore.ai/, 2019.

[31] nGraph. https://www.ngraph.ai/, 2019.

[32] ONNX-ML vs Sklearn Benchmark. https://github.com/xadupre/scikit-learn_benchmarks, 2019.

[33] ONNXMLTools. https://github.com/onnx/onnxmltools, 2019.

[34] RAPIDS Forest Inference Library. https://medium.com/rapids-ai/rapids-forest-inference-library-prediction-at-100-million-rows-per-second-19558890bc35, 2019.

[35] Tensor-RT. https://developer.nvidia.com/tensorrt, 2019.

[36] Broadcasting Semantic. https://www.tensorflow.org/xla/broadcasting, 2020.

[37] Gradient Boosting Algorithm Benchmark. https://github.com/NVIDIA/gbm-bench, 2020.

[38] Hummingbird's Github Repository. https://github.com/microsoft/hummingbird, 2020.

[39] Iris Dataset. https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html, 2020.

[40] Memory Profiler for Python. https://github.com/pythonprofilers/memory_profiler, 2020.

[41] Nomao Dataset. https://www.openml.org/d/1486, 2020.

[42] ONNX. https://github.com/onnx/onnx/blob/master/docs/Operators.md, 2020.

[43] ONNX Portable Format. https://www.infoworld.com/article/3223401/onnx-makes-machine-learning-models-portable-shareable.html, 2020.

[44] OpenML-CC18 Benchmark. https://www.openml.org/s/99, 2020.

[45] RAPIDS cuML. https://github.com/rapidsai/cuml, 2020.

[46] Sambanova: Massive Models for Everyone. https://sambanova.ai/, 2020.

[47] skl2onnx Converter. https://github.com/onnx/sklearn-onnx/, 2020.

[48] Tensorflow JS. https://www.tensorflow.org/js, 2020.

[49] The Status of Sparse Operations in Pytorch. https://github.com/pytorch/pytorch/issues/9674, 2020.

[50] Script for Tensorflow Model Averaging, Accessed January 31, 2020. https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/utils/avg_checkpoints.py.

[51] Scaling Out Search with Apache Spark, Accessed January 31, 2021.

[52] XLA: Optimizing Compiler for Machine Learning : TensorFlow, Accessed January 31, 2021.

[53] Cerebro Documentation, Accessed July 5, 2020. https://adalabucsd.github.io/cerebro-system/.

[54] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016.

[55] Donald A Adjeroh and Kingsley C Nwosu. Multimedia Database Management–Requirements and Issues. *IEEE Multimedia*, pages 24–33, 1997.

[56] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Gowdal, Matteo Interlandi, Alekh Jindal, Kostantinos Karanasos, Subru Krishnan, Brian Kroth, Jyoti Leeka, Kwanghyun Park, Hiren Patel, Olga Poppe, Fotis Psallidas, Raghu Ramakrishnan, Abhishek Roy, Karla Saur, Rathijit Sen, Markus Weimer, Travis Wright, and Yiwen Zhu. Cloudy with high chance of DBMS: A 10-year prediction for Enterprise-Grade ML. *arXiv e-prints*, page arXiv:1909.00084, Aug 2019.

[57] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupré, Vadim Eksarevskiy, Senja Filipi, Tom Finley, Abhishek Goswami, Monte Hoover, Scott Inglis, Matteo Interlandi, Najeeb Kazmi, Gleb Krivosheev, Pete Luferenko, Ivan Matantsev, Sergiy Matusevych, Shahab Moradi, Gani Nazirov, Justin Ormont, Gal Oshri, Artidoro Pagnoni, Jignesh Parmar, Prabhat Roy, Mohammad Zeeshan Siddiqui, Markus Weimer, Shauheen Zahirazami, and Yiwen Zhu. Machine Learning at Microsoft with ML.NET. In Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis, editors, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 2448–2458. ACM, 2019.

[58] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1009–1024. ACM, 2017.

[59] Samuel Albanie. Euclidean Distance Matrix Trick. *Retrieved from Visual Geometry Group, University of Oxford*, 2019.

[60] Samuel Albanie. Memory Consumption and FLOP Count Estimates for Convnets, Accessed January 31, 2020. `https://github.com/albanie/convnet-burden`.

[61] Amazon. The Total Cost of Ownership (TCO) of Amazon Sagemaker. `https://pages.awscloud.com/rs/112-TZM-766/images/Amazon_SageMaker_TCO_uf.pdf`, 2020.

[62] Dario Amodei and Danny Hernandez. AI and Compute, Accessed January 31, 2021. `https://openai.com/blog/ai-and-compute`.

[63] Farhad Arbabzadah, Grégoire Montavon, Klaus-Robert Müller, and Wojciech Samek. Identifying Individual Facial Expressions by Deconstructing a Neural Network. In Bodo Rosenhahn and Bjoern Andres, editors, *Pattern Recognition - 38th German Conference, GCPR 2016, Hannover, Germany, September 12-15, 2016, Proceedings*, volume 9796 of *Lecture Notes in Computer Science*, pages 344–354. Springer, 2016.

[64] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books LLC, 2018.

[65] AWS. Automate Data Labeling, Accessed January 31, 2021.

[66] Hossein Azizpour, Ali Sharif Razavian, Josephine Sullivan, Atsuto Maki, and Stefan Carlsson. Factors of Transferability for a Generic ConvNet Representation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 38(9):1790–1802, 2016.

[67] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD 17, page 13871395, New York, NY, USA, 2017. Association for Computing Machinery.

[68] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? In Madeleine Clare Elish, William Isaac, and Richard S. Zemel, editors, *FAccT '21: 2021 ACM Conference on Fairness, Accountability, and Transparency, Virtual Event / Toronto, Canada, March 3-10, 2021*, pages 610–623. ACM, 2021.

[69] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011*, pages 2546–2554, 2011.

[70] James Bergstra, Daniel Yamins, and David D. Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 115–123. JMLR.org, 2013.

[71] Dimitri P. Bertsekas. A New Class of Incremental Gradient Methods for Least Squares Problems. *Society for Industrial and Applied Mathematics Journal on Optimization*, 7(4):913–926, April 1997.

[72] Shamim Bhuiyan, Michael Zheludkov, and Timur Isachenko. High Performance In-Memory Computing with Apache Ignite. *Lulu. com*, 2017.

[73] Benjamin Biering. Getting Started with AI: How Much Data Do You Need?, Accessed January 31, 2021.

[74] Matthias Boehm, Arun Kumar, and Jun Yang. *Data Management in Machine Learning Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.

[75] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas R. Burdick, and Shivakumar Vaithyanathan. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB*, 7(7):553–564, 2014.

[76] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri S. Chatterji, Annie S. Chen, Kathleen Creel, Jared Quincy Davis, Dorottya Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah D. Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark S. Krass, Ranjay Krishna, and Rohith Kuditipudi. On the Opportunities and Risks of Foundation Models. *CoRR*, abs/2108.07258, 2021.

[77] Léon Bottou. Curiously Fast Convergence of some Stochastic Gradient Descent Algorithms. Unpublished open problem offered to the attendance of the Symposium on Learning and Data Science 2009 conference, 2009.

[78] Xavier Bouthillier and Gaël Varoquaux. *Survey of Machine-Learning Experimental Methods at NeurIPS2019 and ICLR2020*. PhD thesis, Inria Saclay Ile de France, 2020.

[79] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[80] Andrew Brock, Theodore Lim, James Millar Ritchie, and Nicholas J Weston. Freeze-out: Accelerate training by progressively freezing layers. In *NIPS 2017 Workshop on Optimization: 10th NIPS Workshop on Optimization for Machine Learning*, 2017.

[81] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag, 3rd edition, 2001.

[82] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. EVA^2: Exploiting Temporal Redundancy in Live Computer Vision. In Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi, editors, *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pages 533–546. IEEE Computer Society, 2018.

[83] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An Analysis of Deep Neural Network Models for Practical Applications. *CoRR*, abs/1605.07678, 2016.

[84] Jordan A Carlson, John Bellettiere, Jacqueline Kerr, Jo Salmon, Anna Timperio, Simone JJM Verswijveren, and Nicola D Ridgers. Day-level Sedentary Pattern Estimates Derived from Hip-worn Accelerometer Cut-points in 8–12-year-olds: Do they Reflect Postural Transitions? *Journal of Sports Sciences*, 37(16):1899–1909, 2019.

[85] Lukas Cavigelli, Philippe Degen, and Luca Benini. Cbinfer: Change-based Inference for Convolutional Neural Networks on Video Data. In *Proceedings of the 11th International Conference on Distributed Smart Cameras*, pages 1–8. ACM, 2017.

[86] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. 1991.

[87] Chengliang Chai and Guoliang Li. Human-in-the-loop Techniques in Machine Learning. *Data Engineering*, 37:16, 2020.

[88] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 34–43, 1998.

[89] Surajit Chaudhuri and Kyuseok Shim. Optimization of Queries with User-defined Predicates. *ACM Trans. Database Syst.*, 24(2):177–228, June 1999.

[90] Chun-Fu (Richard) Chen, Gwo Giun (Chris) Lee, Yinglong Xia, W. Sabrina Lin, Toyotaro Suzumura, and Ching-Yung Lin. Efficient Multi-training Framework of Image Deep Learning on GPU Cluster. In *2015 IEEE International Symposium on Multimedia, ISM 2015*, pages 489–494. IEEE Computer Society, 2015.

[91] Lingjiao Chen, Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. Towards Linear Algebra over Normalized Data. *PVLDB*, 10(11):1214–1225, 2017.

[92] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[93] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 579–594, Berkeley, CA, USA, 2018. USENIX Association.

[94] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *CoRR*, abs/1604.06174, 2016.

[95] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759*, 2014.

[96] Rada Chirkova and Jun Yang. Materialized Views. *Foundations and Trends® in Databases*, 4(4):295–405, 2012.

[97] Franois Chollet. Transfer Learning & Fine-tuning, Accessed January 31, 2021.

[98] Hong-Tai Chou and David J DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Algorithmica*, 1(1-4):311–336, 1986.

[99] Alexandra Chronopoulou, Christos Baziotis, and Alexandros Potamianos. An Embarrassingly Simple Approach for Transfer Learning from Pretrained Language Models. *arXiv preprint arXiv:1902.10547*, 2019.

[100] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. InferLine: ML Inference Pipeline Composition Framework. *CoRR*, abs/1812.01776, 2018.

[101] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. 2017.

[102] CriteoLabs. Kaggle Contest Dataset Is Now Available for Academic Use!, Accessed January 31, 2020. `https://ailab.criteo.com/category/dataset`.

[103] Navneet Dalal and Bill Triggs. Histograms of Oriented Gradients for Human Detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, pages 886–893. IEEE Computer Society, 2005.

[104] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in Multi-Query Optimization. In Peter Buneman, editor, *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*. ACM, 2001.

[105] Manoranjan Dash and Huan Liu. Feature Selection for Classification. *Intelligent Data Analysis*, 1(3):131–156, 1997.

[106] Databricks. Resource-efficient Deep Learning Model Selection on Apache Spark, Accessed May 30, 2020. `https://bit.ly/3esN3JT`.

[107] Saskia EJ de Vries, Stephen A Baccus, and Markus Meister. The Projective Field of a Retinal Amacrine Cell. *Journal of Neuroscience*, 31(23):8595–8604, 2011.

[108] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A Large-scale Hierarchical Image Database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009)*, pages 248–255. IEEE Computer Society, 2009.

[109] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[110] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 647–655. JMLR.org, 2014.

[111] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[112] Steffen Eger. Restricted Weighted Integer Compositions and Extended Binomial Coefficients. *J. Integer Seq*, 16(13.1):3, 2013.

[113] Katherine Ellis, Jacqueline Kerr, Suneeta Godbole, John Staudenmayer, and Gert Lanckriet. Hip and Wrist Accelerometer Algorithms for Free-living Behavior Classification. *Medicine and science in sports and exercise*, 48(5):933, 2016.

[114] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level Classification of Skin Cancer with Deep Neural Networks. *Nature*, 542(7639):115–118, 2017.

[115] Bjarke Felbo, Alan Mislove, Anders Søgaard, Iyad Rahwan, and Sune Lehmann. Using Millions of Emoji Occurrences to Learn Any-domain Representations for Detecting Sentiment, Emotion and Sarcasm. In *Proceedings of the 2017 Conference on Empirical Methods*

in Natural Language Processing, pages 1615–1625, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[116] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a Unified Architecture for In-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 325–336. Association for Computing Machinery, 2012.

[117] Tibor Fiala. An Algorithm for the Open-shop Problem. *Mathematics of Operations Research*, 8(1):100–109, 1983.

[118] FirmAI. Machine Learning and Data Science Applications in Industry. `https://github.com/firmai/industry-machine-learning`.

[119] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. Estimating GPU Memory Consumption of Deep Learning Models. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1342–1352. ACM, 2020.

[120] Minos N Garofalakis and Phillip B Gibbons. Approximate Query Processing: Taming the TeraBytes. In *VLDB*, pages 343–352, 2001.

[121] J. V. Gautam, H. B. Prajapati, V. K. Dabhi, and S. Chaudhary. A Survey on Job Scheduling Algorithms in Big Data Processing. In *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–11, 2015.

[122] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John 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*, KDD '17, pages 1487–1495. Association for Computing Machinery, 2017.

[123] Teofilo Gonzalez and Sartaj Sahni. Open Shop Scheduling to Minimize Finish Time. *J. ACM*, 23(4):665–679, October 1976.

[124] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.

[125] Google. Google Cloud Pricing Calculator, Accessed January 31, 2021.

[126] Google-Research. Huge Embedding Output File Issue #91, Accessed January 31, 2021.

[127] Mikael Anne Greenwood-Hickman, Supun Nakandala, Marta M Jankowska, Dori E Rosenberg, Fatima Tuz-Zahra, John Bellettiere, Jordan A Carlson, Paul R Hibbing, Jingjing Zou, and Andrea Z Lacroix. The CNN Hip Accelerometer Posture (CHAP) Method for Classifying Sitting Patterns from Hip Accelerometers: A Validation Study. *Medicine and Science in Sports and Exercise*, 53(11):2445, 2021.

[128] Roger Grosse. CSC321 Lecture 6: Backpropagation, Accessed January 31, 2021. http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/slides/lec6.pdf.

[129] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019*, pages 485–500. USENIX Association, 2019.

[130] Hui Guan, Laxmikant Kishor Mokadam, Xipeng Shen, Seung-Hwan Lim, and Robert M. Patton. FLEET: Flexible Efficient Ensemble Training for Heterogeneous Deep Neural Networks. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[131] Antonio Gulli and Sujit Pal. *Deep Learning with Keras*. Packt Publishing Ltd, 2017.

[132] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[133] Himanshu Gupta and Inderpal Singh Mumick. Selection of Views to Materialize in a Data Warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

[134] Gurobi. Gurobi Optimization, Accessed January 31, 2021. https://www.gurobi.com.

[135] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint arXiv:1510.00149*, 2015.

[136] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[137] Ruining He and Julian McAuley. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *proceedings of the 25th international conference on world wide web*, pages 507–517. International World Wide Web Conferences Steering Committee, 2016.

[138] Yihui He, Xiangyu Zhang, and Jian Sun. Channel Pruning for Accelerating Very Deep Neural Networks. In *International Conference on Computer Vision (ICCV)*, volume 2, 2017.

[139] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[140] Joseph M. Hellerstein and Michael Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 267–276. ACM, 1993.

[141] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 261–272. www.cidrdb.org, 2011.

[142] Willy Herroelen, Bert De Reyck, and Erik Demeulemeester. Resource-constrained Project Scheduling: A Survey of Recent Developments. *Computers & Operations Research*, 25(4):279–302, 1998.

[143] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-Efficient Transfer Learning for NLP. In *International Conference on Machine Learning*, pages 2790–2799, 2019.

[144] Jeremy Howard and Sebastian Ruder. Universal Language Model Fine-tuning for Text Classification. *arXiv preprint arXiv:1801.06146*, 2018.

[145] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely Connected Convolutional Networks. pages 4700–4708, 2017.

[146] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, and James Cheng. FlexPS: Flexible Parallelism Control in Parameter Server Architecture. *PVLDB*, 11(5):566–579, 2018.

[147] Intel. Machine learning fpga. `https://www.intel.com/content/www/us/en/products/docs/storage/programmable/applications/machine-learning.html`, 2020.

[148] Mohammad Tariqul Islam, Md Abdul Aowal, Ahmed Tahseen Minhaz, and Khalid Ashraf. Abnormality Detection and Localization in Chest X-Rays using Deep Convolutional Neural Networks. *CoRR*, abs/1705.09850, 2017.

[149] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks. *CoRR*, abs/1711.09846, 2017.

[150] Matthias Jarke and Jurgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys (CsUR)*, 16(2):111–152, 1984.

[151] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019*. mlsys.org, 2019.

[152] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-Aware Distributed Parameter Servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD 17, pages 463–478. Association for Computing Machinery, 2017.

[153] Yushi Jing, David Liu, Dmitry Kislyuk, Andrew Zhai, Jiajing Xu, Jeff Donahue, and Sarah Tavel. Visual Search at Pinterest. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1889–1898, 2015.

[154] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. pages 1–12, 2017.

[155] Oya Kalipsiz. Multimedia Databases. In *Information Visualization, 2000. Proceedings. IEEE International Conference on*, pages 111–115. IEEE, 2000.

[156] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. NoScope: Optimizing Neural Network Queries Over Video at Scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[157] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. Extending Relational Query Processing with ML Inference. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[158] Andrej Karpathy and Li Fei-Fei. Deep Visual-Semantic Alignments for Generating Image Descriptions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(4):664–676, April 2017.

[159] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017.

[160] Daniel S Kermany, Michael Goldbaum, Wenjia Cai, Carolina CS Valentim, Huiying Liang, Sally L Baxter, Alex McKeown, Ge Yang, Xiaokang Wu, and Fangbing Yan. Identifying Medical Diagnoses and Treatable Diseases by Image-based Deep Learning. *Cell*, 172(5):1122–1131, 2018.

[161] Tae-Young Kim and Sung-Bae Cho. Predicting Residential Energy Consumption Using CNN-LSTM Neural Networks. *Energy*, 182:72–81, 2019.

[162] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[163] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2015.

[164] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[165] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017*, volume 54 of *Proceedings of Machine Learning Research*, pages 528–536. PMLR, 2017.

[166] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Pearson Education India, 2006.

[167] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *PVLDB*, 12(11):1399–1412, 2019.

[168] Pradap Konda, Arun Kumar, Christopher Ré, and Vaishnavi Sashikanth. Feature Selection in Enterprise Analytics: A Demonstration using an R-based Data Analytics System. *PVLDB*, 6(12):1306–1309, 2013.

[169] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. Tensors: An Abstraction for General Data Processing. *Proceedings of the VLDB Endowment*, 14(10):1797–1804, 2021.

[170] Peter Kraft, Daniel Kang, Deepak Narayanan, Shoumik Palkar, Peter Bailis, and Matei Zaharia. Willump: A Statistically-Aware End-to-end Optimizer for Machine Learning Inference. *arXiv e-prints*, page arXiv:1906.01974, Jun 2019.

[171] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information*

Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States, pages 1106–1114, 2012.

[172] Arun Kumar, Matthias Boehm, and Jun Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1717–1722. Association for Computing Machinery, 2017.

[173] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. Model Selection Management Systems: the Next Frontier of Advanced Analytics. *SIGMOD Rec.*, 44(4):1722, May 2016.

[174] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. Learning Generalized Linear Models Over Normalized Data. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1969–1984. ACM, 2015.

[175] Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, and Volker Markl. Bridging the Gap: Towards Optimization Across Linear and Relational Algebra. In Foto N. Afrati, Jacek Sroka, and Jan Hidders, editors, *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2016, San Francisco, CA, USA, July 1, 2016*, page 1. ACM, 2016.

[176] Quoc Le and Tomas Mikolov. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1188–1196, 2014.

[177] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. TFLMS: Large Model Support in TensorFlow by Graph Rewriting. *CoRR*, abs/1807.02037, 2018.

[178] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable Multi-Query Optimization for SPARQL. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 666–677. IEEE, 2012.

[179] Yann LeCun. Deep Learning Hardware: Past, Present, and Future. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 12–19. IEEE, 2019.

[180] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.

[181] Jaejun Lee, Raphael Tang, and Jimmy Lin. What Would Elsa Do? Freezing Layers During Transformer Fine-Tuning. *arXiv preprint arXiv:1911.03090*, 2019.

[182] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In Andrea C. Arpaci-Dusseau and Geoff

Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 611–626. USENIX Association, 2018.

[183] Alon Y Levy, Alberto O Mendelzon, and Yehoshua Sagiv. Answering Queries Using Views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 95–104. ACM, 1995.

[184] Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Xi Wu, Jeffrey F. Naughton, and Jignesh M. Patel. Tuple-Oriented Compression for Large-Scale Mini-Batch Stochastic Gradient Descent. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1517–1534. Association for Computing Machinery, 2019.

[185] Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A System for Massively Parallel Hyperparameter Tuning. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[186] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017.

[187] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14*, pages 583–598. USENIX Association, 2014.

[188] Ping Li. Robust Logitboost and adaptive base class (ABC) Logitboost. In *In Proceedings of the Twenty-Sixth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI10)*.

[189] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.*, 2020.

[190] Side Li and Arun Kumar. Towards an Optimized GROUP BY Abstraction for Large-Scale Machine Learning. *Proc. VLDB Endow.*, 14(11):2327–2340, 2021.

[191] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 3049–3058. PMLR, 2018.

[192] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A Research Platform for Distributed Model Selection and Training. *CoRR*, abs/1807.05118, 2018.

[193] Rui Liu, Sanjan Krishnan, Aaron J Elmore, and Michael J Franklin. Understanding and Optimizing Packed Neural Network Training for Hyper-Parameter Tuning. *arXiv preprint arXiv:2002.02885*, 2020.

[194] Yuhan Liu, Saurabh Agarwal, and Shivaram Venkataraman. AutoFreeze: Automatically Freezing Model Blocks to Accelerate Fine-tuning. *CoRR*, abs/2102.01386, 2021.

[195] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.

[196] Scott M Lundberg and Su-In Lee. A Unified Approach to Interpreting Model Predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

[197] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. Understanding the Effective Receptive Field in Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, pages 4898–4906, 2016.

[198] MADLib. User Documentation for Apache MADlib, Accessed May 30, 2020. `https://bit.ly/3epbEyS`.

[199] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. Image-based Recommendations on Styles and Substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 43–52, 2015.

[200] Michael McCloskey and Neal J Cohen. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. In *Psychology of Learning and Motivation*, volume 24, pages 109–165. Elsevier, 1989.

[201] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, and Sean Owen. Mllib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[202] Tim Miller. Explanation in Artificial Intelligence: Insights from the Social Sciences. *arXiv preprint arXiv:1706.07269*, 2017.

[203] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and Mitigating Data Stalls in DNN Training. *Proceedings of the VLDB Endowment*, 2021.

[204] Sharada Prasanna Mohanty, David P. Hughes, and Marcel Salathé. Using Deep Learning for Image-Based Plant Disease Detection. *CoRR*, abs/1604.03169, 2016.

[205] Robert Monarch. *Human-in-the-Loop Machine Learning. Active Learning and Annotation for Human-centered AI.* Manning Early Access Program (MEAP). Manning Publications, 2019.

[206] Bert Moons and Marian Verhelst. A 0.3–2.6 TOPS/W Precision-scalable Processor for Real-time Large-scale ConvNets. In *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on*, pages 1–2. IEEE, 2016.

[207] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*, pages 561–577. USENIX Association, 2018.

[208] Mohammad Motamedi, Felix Portillo, Mahya Saffarpour, Daniel Fong, and Soheil Ghiasi. Resource-Scalable CNN Synthesis for IoT Applications. *arXiv preprint arXiv:1901.00738*, 2018.

[209] Kabir Nagrecha and Arun Kumar. Hydra: A System for Large Multi-Model Deep Learning. *arXiv preprint arXiv:2110.08633*, 2021.

[210] Supun Nakandala and Arun Kumar. Vista: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1685–1700. Association for Computing Machinery, 2020.

[211] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. Incremental and Approximate Inference for Faster Occlusion-Based Deep CNN Explanations. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1589–1606. Association for Computing Machinery, 2019.

[212] Supun Nakandala, Kabir Nagrecha, Arun Kumar, and Yannis Papakonstantinou. Incremental and Approximate Computations for Accelerating Deep CNN Inference. *ACM Trans. Database Syst.*, 0(ja), 2020.

[213] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 899–917. USENIX Association, November 2020.

[214] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*, pages 1–4, 2019.

[215] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.*, 13(12):2159–2173, July 2020.

[216] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 1–15. Association for Computing Machinery, 2019.

[217] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating Deep Learning Workloads through Efficient Multi-Model Execution. In *NeurIPS Workshop on Systems for Machine Learning*, December 2018.

[218] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal Deep Learning. In *ICML*, 2011.

[219] Milos Nikolic, Mohammed ElSeidy, and Christoph Koch. LINVIEW: Incremental View Maintenance for Complex Analytical Queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 253–264. ACM, 2014.

[220] Shu Lih Oh, Eddie Y.K. Ng, Ru San Tan, and U. Rajendra Acharya. Automated Diagnosis of Arrhythmia Using Combination of CNN and LSTM Techniques with Variable Length Heart Beats. *Computers in Biology and Medicine*, 102:278–287, 2018.

[221] Tom O'Malley. Hyperparameter Tuning with Keras Tuner, Accessed January 31, 2020. `https://blog.tensorflow.org/2020/01/hyperparameter-tuning-with-keras-tuner.html?linkId=81371017`.

[222] Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony K.H. Tung, Yuan Wang, Zhongle Xie, Meihui Zhang, and Kaiping Zheng. SINGA: A Distributed Deep Learning Platform. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, pages 685–688. Association for Computing Machinery, 2015.

[223] Allen Ordookhanians, Xin Li, Supun Nakandala, and Arun Kumar. Demonstration of Krypton: Optimized CNN Inference for Occlusion-Based Deep CNN Explanations. *PVLDB*, 12(12):1894–1897, 2019.

[224] Szilard Pafka. Big RAM is Eating Big Data
- Size of Datasets Used for Analytics, Accessed January 31, 2020. `https://www.kdnuggets.com/2015/11/big-ram-big-data-size-datasets.html`.

[225] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.*, 11(9):1002–1015, 2018.

[226] Sinno Jialin Pan and Qiang Yang. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[227] Yannis Papakonstantinou and Vasilis Vassalos. Query Rewriting for Semistructured Data. *ACM SIGMOD Record*, 28(2):455–466, 1999.

[228] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic Differentiation in PyTorch. In *NIPS-W*, 2017.

[229] Matthew E Peters, Sebastian Ruder, and Noah A Smith. To Tune or Not to Tune? Adapting Pretrained Representations to Diverse Tasks. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*, pages 7–14, 2019.

[230] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulic, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. Adapterhub: A framework for adapting transformers. In Qun Liu and David Schlangen, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, EMNLP 2020 - Demos, Online, November 16-20, 2020*, pages 46–54. Association for Computational Linguistics, 2020.

[231] Arnab Phani, Benjamin Rath, and Matthias Boehm. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. SIGMOD, 2021.

[232] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data Lifecycle Challenges in Production Machine Learning: A Survey. *SIGMOD Record*, 47(2):17–28, 2018.

[233] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Matteo Interlandi, Avrilia Floratou, Konstantinos Karanasos, Wentao Wu, Ce Zhang, Subru Krishnan, Carlo Curino, and Markus Weimer. Data Science Through the Looking Glass and What We Found There. *arXiv preprint arXiv:1912.09536*, 2019.

[234] PyTorch. The Topological Sorting Algorithm for Computation Graphs in PyTorch, Accessed January 31, 2021. `https://github.com/pytorch/pytorch/blob/v1.2.0/caffe2/core/nomnigraph/include/nomnigraph/Graph/TopoSort.h\#L26`.

[235] Sivaramakrishnan Rajaraman, Sameer K Antani, Mahdieh Poostchi, Kamolrat Silamut, Md A Hossain, Richard J Maude, Stefan Jaeger, and George R Thoma. Pre-trained Convolutional Neural Networks as Feature Extractors Toward Improved Malaria Parasite Detection in Thin Blood Smear Images. *PeerJ*, 6:e4568, 2018.

[236] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB Endow.*, 11(4):432444, December 2017.

[237] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems (3. ed.)*. McGraw-Hill, 2003.

[238] Alexander Ratner, Stephen H. Bach, Henry R. Ehrenberg, Jason Alan Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid Training Data Creation with Weak Supervision. *Proc. VLDB Endow.*, 11(3):269–282, 2017.

[239] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. Learning Multiple Visual Domains with Residual Adapters. In *Advances in Neural Information Processing Systems*, pages 506–516, 2017.

[240] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training. *CoRR*, abs/2101.06840, 2021.

[241] Maryem Rhanoui, Mounia Mikram, Siham Yousfi, and Soukaina Barzali. A CNN-BiLSTM Model for Document-level Sentiment Analysis. *Machine Learning and Knowledge Extraction*, 1(3):832–847, 2019.

[242] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 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*, pages 1135–1144, 2016.

[243] Dori Rosenberg, Rod Walker, Mikael Anne Greenwood-Hickman, John Bellettiere, Yunhua Xiang, KatieRose Richmire, Michael Higgins, David Wing, Eric B Larson, and Paul K Crane. Device-assessed Physical Activity and Sedentary Behavior in a Community-based Cohort of Older Adults. *BMC Public Health*, 20(1):1–13, 2020.

[244] Andreas Rücklé, Gregor Geigle, Max Glockner, Tilman Beck, Jonas Pfeiffer, Nils Reimers, and Iryna Gurevych. AdapterDrop: On the Efficiency of Adapters in Transformers. *arXiv preprint arXiv:2010.11918*, 2020.

[245] Sebastian Ruder, Matthew E Peters, Swabha Swayamdipta, and Thomas Wolf. Transfer Learning in Natural Language Processing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*, pages 15–18, 2019.

[246] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, and Michael Bernstein. Imagenet Large Scale Visual Recognition Challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[247] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, A Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[248] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning Linear Regression Models over Factorized Joins. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 3–18. ACM, 2016.

[249] Scott Sievert, Tom Augspurger, and Matthew Rocklin. Better and Faster Hyperparameter Optimization with Dask. In *Proceedings of the 18th Python in Science Conference*, pages 118–125, 2019.

[250] Timos K. Sellis. Multiple-query Optimization. *ACM Trans. Database Syst.*, 13(1):23–52, March 1988.

[251] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 618–626. IEEE, 2017.

[252] Alexander Sergeev and Mike Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TF. *CoRR*, abs/1802.05799, 2018.

[253] Amazon Web Services. SageMaker Ground Truth, Accessed January 31, 2021. `https://aws.amazon.com/sagemaker/groundtruth/`.

[254] Burr Settles. Active Learning Literature Survey. 2009.

[255] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: from Theory to Algorithms*. Cambridge University Press, 2014.

[256] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep Learning in Medical Image Analysis. *Annual review of biomedical engineering*, 19:221–248, 2017.

[257] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[258] Scott Sievert, Tom Augspurger, Matthew Rocklin, Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe. Better and Faster Hyperparameter Optimization with Dask. In *Proceedings of the 18th Python in Science Conference*, pages 118–125, 2019.

[259] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *arXiv preprint arXiv:1312.6034*, 2013.

[260] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[261] Ask Solem and Celery-contributors. Celery: Distributed Task Queue, Accessed January 31, 2020. http://www.celeryproject.org/.

[262] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 535–546. IEEE Computer Society, 2017.

[263] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The jJurnal of Machine Learning Research*, 15(1):1929–1958, 2014.

[264] Nitish Srivastava and Ruslan Salakhutdinov. Multimodal Learning with Deep Boltzmann Machines. volume 15, pages 2949–2980. JMLR.org, January 2014.

[265] Hang Su and Haoyu Chen. Experiments on Parallel Training of Deep Neural Network using Model Averaging. *CoRR*, abs/1507.01239, 2015.

[266] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic Attribution for Deep Networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3319–3328, 2017.

[267] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 1–9. IEEE Computer Society, 2015.

[268] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A Survey on Deep Transfer Learning. In *International conference on artificial neural networks*, pages 270–279. Springer, 2018.

[269] TensorFlow. The Topological Sorting Algorithm for Computation Graphs in TensorFlow, Accessed January 31, 2021. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/grappler/utils/topological_sort.h.

[270] Anthony Thomas and Arun Kumar. A Comparative Evaluation of Systems for Scalable Linear Algebra-Based Analytics. *PVLDB*, 11(13):2168–2182, 2018.

[271] Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, pages 142–147, 2003.

[272] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.

[273] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1285–1300. ACM, 2018.

[274] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[275] VMware. Model Selection for Deep Neural Networks on Greenplum Database, Accessed May 30, 2020. `https://bit.ly/2AaQLc2`.

[276] Paul Voigt and Axel Von dem Bussche. *The EU General Data Protection Regulation (GDPR)*, volume 18. Springer, 2017.

[277] Ji Wan, Dayong Wang, Steven Chu Hong Hoi, Pengcheng Wu, Jianke Zhu, Yongdong Zhang, and Jintao Li. Deep Learning for Content-based Image Retrieval: A Comprehensive Study. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 157–166, 2014.

[278] Yilun Wang and Michal Kosinski. Deep Neural Networks are More Accurate than Humans at Detecting Sexual Orientation from Facial Images. *Journal of Personality and Social Psychology*, 114(2):246, 2018.

[279] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Trans. Image Process.*, 13(4):600–612, 2004.

[280] Pete Warden. The Machine Learning Reproducibility Crisis, Accessed January 31, 2020. `https://petewarden.com/2018/03/19/the-machine-learning-reproducibility-crisis`.

[281] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised Deep Learning with Partial Gradient Exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC 16, pages 84–97. Association for Computing Machinery, 2016.

[282] Sarah Webb. Deep Learning for Biology. *Nature*, 554(7690):555–558, 2018.

[283] Gerhard J. Woeginger. The Open Shop Scheduling Problem. In *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018*, volume 96 of *LIPIcs*, pages 4:1–4:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[284] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, and Morgan Funtowicz. Huggingface's Transformers: State-of-the-art Natural Language Processing, 2019.

[285] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*, pages 595–610. USENIX Association, 2018.

[286] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. HELIX: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.*, 12(4):446–460, December 2018.

[287] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How Transferable are Features in Deep Neural Networks? In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3320–3328, 2014.

[288] Joe Yue-Hei Ng, Fan Yang, and Larry S Davis. Exploiting Local Features from Deep Networks for Image Retrieval. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 53–61, 2015.

[289] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012*, pages 15–28. USENIX Association, 2012.

[290] Matthew D Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. In *European Conference on Computer Vision*, pages 818–833. Springer, 2014.

[291] Ce Zhang, Arun Kumar, and Christopher Ré. Materialization Optimizations for Feature Selection Workloads. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 265–276. ACM, 2014.

[292] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. SLAQ: Quality-Driven Scheduling for Distributed Machine Learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 390–404. Association for Computing Machinery, 2017.

[293] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. Retiarii: A Deep Learning Exploratory-Training Framework. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 919–936, 2020.

[294] Yuhao Zhang, Frank Mcquillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches. *Proc. VLDB Endow.*, 14(10):1769–1782, 2021.

[295] Weijie Zhao, Florin Rusu, Bin Dong, Kesheng Wu, and Peter Nugent. Incremental View Maintenance over Array Data. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 139–154. ACM, 2017.

[296] Luisa M. Zintgraf, Taco S. Cohen, Tameem Adel, and Max Welling. Visualizing Deep Neural Network Decisions: Prediction Difference Analysis. *CoRR*, abs/1702.04595, 2017.

[297] Yongqiang Zou, Xing Jin, Yi Li, Zhimao Guo, Eryu Wang, and Bin Xiao. Mariana: Tencent Deep Learning Platform and Its Applications. *PVLDB*, 7(13):1772–1777, 2014.