ABSTRACT
Machine learning (ML) training and scoring fundamentally relies on linear algebra programs and more general tensor computations. Most ML systems utilize distributed parameter servers and similar distribution strategies for mini-batch stochastic gradient descent training. However, many more tasks in the data science and engineering lifecycle can benefit from efficient tensor computations. Examples include primitives for data cleaning, data and model debugging, data augmentation, query processing, numerical simulations, as well as a wide variety of training and scoring algorithms. In this survey tutorial, we first make a case for the importance of optimizing more general tensor computations, and then provide an in-depth survey of existing applications, optimizing compilation techniques, and underlying runtime strategies. Interestingly, there are close connections to data-intensive applications, query rewriting and optimization, as well as query processing and physical design. Our goal for the tutorial is to structure existing work, create common terminology, and identify open research challenges.

1 INTRODUCTION
Over the past decade(s), a wide variety of machine learning (ML) systems emerged from the ML/stats, data management, and high-performance computing communities. Early work focused on statistical computing platforms like R and ML algorithm libraries like scikit-learn [98], as well as specialized systems for clustering, matrix factorization [40], graph processing [126], and others. Later work also provided infrastructure for scalable, distributed computation [3, 104, 131] and hardware accelerators. Nowadays, most general-purpose ML systems focus exclusively on distributed parameter servers [1, 30, 57, 77, 115] and similar distribution strategies [17, 64, 101] for mini-batch training via stochastic gradient descent (SGD). Many more tasks in data science and engineering rely on linear algebra and numerical computation, but they often exhibit different characteristics and are implemented separately [9].

General Tensor Computations: In contrast to this narrowing focus on mini-batch SGD, in this tutorial, we make a case for supporting general linear algebra programs and tensor computations for a wide variety of applications and workload characteristics. Besides diverse ML algorithms and statistical learning [39], there are new compelling use cases. First, state-of-the-art data integration [31], feature and semantic type detection [50, 113, 132], and data cleaning [78, 121] all rely on ML. Second, there is work on data imputation, cleaning, and ML tightly interwoven with query processing [19], which requires integrated systems support [28, 32, 38, 44, 129]. Third, also data augmentation and simulation cleanly map to numerical computation. Recent work applies machine learning for more cost-effective weather forecasting [2] as well as simulations of fluid dynamics and material deformation [99]. Interestingly, both data augmentation and simulation allow for generating unlimited datasets. Fourth, even complex, enumeration-based algorithms for model debugging [106] as well as tree-based models [86] can be expressed and efficiently executed in linear algebra.

Optimizing Tensor Computations: Efficient and scalable system infrastructure for such use cases relies—due to complex, hierarchically composed primitives—on optimizing compilers and generating scalable runtime plans. Given increasing specialization, hand-crafting plans for different characteristics and deployments becomes infeasible. Automatic plan generation allows to seamlessly adapt to diverse workloads and data characteristics. In this context, a rapidly growing set of compilation and runtime techniques emerges. Our goal for this tutorial is to structure the space, create common terminology, and identify open research challenges. Common terminology and well-defined sub-areas would serve our community well by focusing efforts and simplifying reuse.

Tutorial Scope: Drawing from our experience building systems for linear algebra and tensor computations (e.g., SystemML [12],

ACM Reference Format:
SystemDS [11], Hummingbird [86], TQP [44], SimSQL [18, 39, 52, 82], we aim to survey the state-of-the-art from applications to compilation and runtime techniques. The technical background is summarized in Sections 2–4, and the tutorial format is as follows:

- **Tutorial Type:** Survey,
- **Preferred Duration:** 3 hours (1.5 hours would also be possible but only in reduced breadth and depth),
- **Target Audience:** Systems researchers and practitioners with basic applied ML background (we do not expect prior knowledge of state-of-the-art algorithms or system internals), and
- **Hands-on Tutorial:** no HW/SW requirements.

## 2 TENSOR COMPUTATIONS

In data-centric ML pipelines, there are multiple compelling use cases for more general tensor computations and respective system infrastructure in terms of compilation and runtime techniques.

### 2.1 Data Preparation and Cleaning

The inspiring tutorial by Dong and Rekatsinas [31] made a great case for the natural symbiosis of data integration and machine learning. Example tasks that heavily rely on ML—and thus, tensor computations—are data extraction, schema alignment, entity resolution, and data fusion. Interestingly, the same observation applies to other data preparation tasks such as data validation [110], data cleaning [45, 78], outlier/anomaly detection [127], missing value imputation [121], semantic type detection [50, 132], feature selection [123], feature engineering [113], and feature transformations [100]. For example, missing value imputation via chained equations (mice) [121] repeatedly extracts features with missing values, trains models (classifiers for categorical, regressors for numerical) using observed feature values as labels, and utilizes the models for missing value imputation. For this reason, implementing cleaning primitives in linear algebra is very compelling because it avoids unnecessary boundary crossing among systems and libraries.

### 2.2 Data Augmentation and Simulation

Data augmentation takes a small labeled dataset and generates many more synthetic examples via transformations and the original labels. Common transformations include reflections, translations, shearing, rotations, cropping, and mixup, which increase data coverage for improved generalization. The seminal AlexNet [72] paper heavily relied on data augmentation (by 2048x the original datasize), and since then it has become common practice. Recently, additional work also tunes data augmentation pipelines and their parameters [26], and pushes data augmentation as specialized kernels into model training to avoid data materialization [29]. Furthermore, machine learning is also applied for more cost-effective weather forecasting [2] as well as simulations of fluid dynamics and material deformation [99]. In this context, very simple MLP models are utilized and the simulation characteristics yield a wide variety of workload characteristics. Prior related work also focused on Monte Carlo sampling [51], and Markov chain simulation [18, 39]. Both data augmentation and simulation allow for generating unlimited datasets with interesting opportunities of directed sampling according to model accuracy as well as fusion.

### 2.3 Query Processing

Recently, tensor computations have been proposed for executing relational operators and even full queries. TCUDB [48] maps join operations into matrix multiplications for efficient execution on Tensor Cores [27, 91]. Raven [66] co-optimizes classical ML models and relational queries. During optimization, Raven can push relational operations such as projection and filters into the ML model as tensor operations. TQP [37, 44] maps Spark SQL queries into tensor computations. TQP supports the full TPC-H benchmark. TQP implements several relational operators (join, aggregation, group by, etc) as PyTorch tensor programs, and chains them together to form query plans which are executable on any hardware supported by PyTorch (e.g., CPU, GPU, TPU [61, 62], etc). Beyond using tensor computations for allowing queries to leverage hardware acceleration, TQP has also proposed tensor computations for query processing over unstructured data such as images, as well taking advantage of the auto-differentiation infrastructure in PyTorch for enabling differentiable queries [38]. While many preliminary results are very promising, other work has also pointed out remaining challenges of mapping queries to TPUs [47].

### 2.4 ML Algorithms and Debugging

Finally, there is rich literature on first- and second-order optimization, statistical learning, a variety of ML models, and more specialized fields such as robust optimization. Besides such algorithms—including tree-based models for both inference [86] and training— that naturally map to tensor computations, recently also model debugging, explanations, and fairness constraints have been elegantly expressed in linear algebra. Examples include linear-algebra-based slice finding [106], learning curve prediction for different slices [120], explanations via linear approximations [81, 85, 103], as well as constrained and unconstrained optimization for fairness and accuracy [89, 107, 133, 133].

## 3 COMPILATION TECHNIQUES

Compilation techniques in ML systems are inspired by programming language compilers, query optimization in DBMS, and optimizing HPC compilers. Some of the covered material overlaps with a previous SIGMOD 2017 tutorial [73] but existing work evolved significantly in the past six years.

### 3.1 Simplification Rewrites

#### Size Propagation:

As a basis for advanced compilation techniques, many systems first propagate size information [16] (e.g., dimensions and sparsity) and subsequently use this information for memory and cost estimation. A central challenge is sparsity estimation of intermediates, which is addressed via naïve metadata estimators [16], naïve bitset estimators [80, 118], density maps [67, 68], biased sampling [128], layered graphs [22, 23], and sketches [116]. Besides sparsity, there has been work on propagating other properties such as symmetry, constants, and storage formats [105].

---

1 For example, see the vectorized `decisionTree()` and `randomForest()` built-in functions—as well as their corresponding predict functions—in Apache SystemDS [11].
Rewrites: Applied rewrites then include traditional programming language rewrites—such as common subexpression elimination, constant folding, branch removal, and loop hoisting [4, 24]—as well as simplification rewrites for linear algebra expressions [16, 74, 119], dedicated dynamic programming approaches for matrix multiplication chains, and sparsity exploitation [12, 49]. Examples of systems that apply such rewrites are SystemDS, TensorFlow, and PyTorch. Graph substitutions are also applied to expressions in deep neural networks [35, 56]. Other rewrites include loop vectorization, and incremental computations [90, 108, 109]. Recent work aims to overcome the need for hand-crafting simplification rewrites by automatic rewrite generation and sum-product optimization based on meta-properties of operations [33, 55, 69, 125].

3.2 Operator Fusion and Code Generation
Operator fusion and code generation are used during ahead-of-time and just-in-time compilation in order to eliminate unnecessary intermediates, apply scan sharing, exploit sparsity, and specialize runtime plans to the underlying, increasingly specialized hardware and runtime strategies. Examples systems with dedicated code generators include BTO [10], Tupleware [25], Kasen [135], SystemDS [14, 33], Weld [94, 95], TACO [70], PlinyCompute [137], Julia [60], TensorFlow XLA, PyTorch, Tensor Comprehensions [122], TVM [20], NVIDIA TensorRT [92], DAPHNE [28], and Tuplex [117]. Recently, several systems were built on top of existing code generators (e.g., JAX on TensorFlow XLA, and TQP on TVM). MLIR [75] aims to avoid redundancy by providing compiler-infrastructure as a library with clearly defined dialects, which gains popularity because hardware vendors can provide specific dialects for their hardware devices, easing their adoption. Remaining challenges include increasing the fusion potential for integrated query processing and linear algebra / tensor operations with dynamic tensor shapes.

3.3 Operator Selection and Placement
Beyond dedicated systems for local and distributed computation, there are several ML systems with multiple backends. Examples include PyTorch [97], TensorFlow [1], SystemDS [11], Samsara [111], DaskML [104], and code generators like TF XLA and TVM [20]. Predominantly though, operator selection and placement is still done via heuristics and manual placement. SystemDS also automatically chooses local and distributed operations depending on memory estimates and budgets, as well as different physical operators based on data characteristics. Reinforcement learning has been successfully used to place neural network layers onto multiple heterogeneous hardware devices [83]. In addition to placing entire operators on devices, DTensors [43] in TensorFlow and PyTorch as well as federated matrices/frames in DAPHNE [28] further allow for more fine-grained placement of shards of tensors on heterogeneous devices. Recent work on compiler infrastructure for ASICs also include the spatial-temporal mapping of data flow graphs to die space [93] and "hardware islands" of multiple devices [6].

4 RUNTIME STRATEGIES
Underneath the evolving compilation techniques, there are important runtime strategies, especially regarding data representations, parallelization strategies, and dedicated runtime backends.

4.1 Data Representations
Overall, we observe an increasing specialization in terms of partitioning and tiling strategies of matrices/frames; as well as dense, sparse, and compressed tile representations. First, in terms of overall partitioning there are local tensors, distributed collections of tiles, as well as federated or shareded tensors with implicit/or explicit sharding information. Distributed collections of tiles originate from ML systems on data-parallel computing frameworks like Spark [131], Flink [3], or Dask [104]. Later, such abstractions have also been adopted for in-DBMS machine learning [52, 82, 112]. This representation has been proven to be very versatile, and recent work on tensor relational algebra makes a case for adopting it as a logical abstraction [54, 129]. Second, at the level of individual tiles, we see more and more specialized sparse [21, 91] and compressed [7, 34, 59, 65, 76, 124, 134] matrix representations as well as specialized data types [71, 91]. Unfortunately, the selection of such representations is still largely a manual trial and error process.

4.2 Parallelization Strategies
Over the last decade, a wide variety of parallelization strategies has been devised, often designed to exploit the characteristics of the underlying hardware and compute infrastructure. First, data-parallel operations follow an SPMD (single-program, multiple-data) model on distributed collections. A variety of physical operators for broadcast-based, shuffle-based, and specialized operations has been proposed and integrated into ML systems [12, 53, 111]. Second, for use cases like hyper-parameter tuning, cross-validation, and embarrassingly-parallel programs, task parallelism (e.g., via parallel for loops) and hybrid parallelism (e.g., concurrent data-parallel jobs on large-scale datasets) has been adopted as well [15, 63, 114]. Third, irregular workloads—as used in reinforcement learning—are addressed with task-dependency-graphs and future-based scheduling [79, 84] as well as tightly integrated architectures of CPU drivers and hardware accelerators [46]. Fourth, distributed minibatch training relies on parameter servers [1, 30, 57, 77, 115] and similar distribution strategies [41]. Here, we often differentiate data- and model-parallel parameter servers, which hold data and model partitions, respectively. Given the challenges of efficient data exchange and synchronization barriers, compared to tuned single-node implementations, recent work added dedicated parallelization for training multiple models [87, 88], sampled, independent subet training [130], as well as sparsity and locality exploitation for sparse and skewed parameter access (e.g., matrix factorization) [101, 102].

4.3 Alternative Backends
Although many ML systems comprise custom runtime backends—with the help of communication libraries such as gRPC, MPI, and NCCL [36]—there is commonly used infrastructure. The first generation of parameter servers [30, 115] often relied on parameter management on Key/Value-stores. Similarly, Function-as-a-Service (FaaS) ML systems—which aim for low start-up costs and automatic elasticity—communicate over rather slow Key-Value Stores and even object stores like S3 [58]. A very popular infrastructure are general-purpose data-parallel computation frameworks like Spark [131], Flink [3], and Dask [104]. Other backends include Ray [84] for irregular task-parallelism, SQL with dedicated matrix/vector types
and recursive computation [52], and federated learning backends (e.g., SystemDS federated [8] and TensorFlow federated [42, 64]) with additional integration of privacy enhancing technologies. The individual ML system backends also implement specific techniques for providing efficient data access. Examples include buffer-pool-like eviction of live variables from GPU to CPU memory or from CPU memory to disk [13], dedicated tile layouts (aka page layouts) with reordered and padded rows [5], as well as in-memory and disk-based index structures for out-of-core data [68, 96, 136].

5 BIOGRAPHIES

The tutorial presenters have backgrounds from industry and academia, and have built various systems with different architectures.

Matthias Boehm: Matthias Boehm is a full professor for large-scale data engineering at Technische Universität Berlin and the BiPOFID research center. His research group focuses on high-level, data science-centric abstractions as well as systems and tools to execute these tasks in an efficient and scalable manner. From 2018 through 2022, Matthias was a BMK-endowed professor for data management at Graz University of Technology, Austria, and a research area manager for data management at the co-located Know-Center GmbH. Prior to 2018, he was a postdoc and research staff member at IBM Research - Almaden, CA, USA, with a major focus on compilation and runtime techniques for declarative, large-scale machine learning in Apache SystemML. Matthias received his Ph.D. from Dresden University of Technology, Germany in 2011 with a dissertation on cost-based optimization of integration flows.

Matteo Interlandi: Matteo Interlandi is a Principal Scientist in the Gray Systems Lab (GSL) at Microsoft, working at the intersection between Machine Learning and Database Systems. Before Microsoft, he was a Postdoctoral Scholar at the University of California, Los Angeles. Prior to joining UCLA, he was Research Associate at the Qatar Computing Research Institute and at the Institute for Human and Machine Cognition. Matteo received his Ph.D. from the University of Modena and Reggio Emilia. Matteo’s work has received a best demo award at VLDB 2022, an honorable mention at SIGMOD 2021 and was featured in the “Best of VLDB 2016”.

Christopher Jermaine: Chris Jermaine is a J.S. Abercrombie Professor of Engineering and Chair, of the CS department at Rice University. He studies data analytics: how to analyze, store, retrieve, and manipulate large and heterogeneous data sets. Within this problem space, most of his work focuses on: (1) The systems-oriented problems that arise when building software to manage and process large and diverse data sets, especially systems for machine learning; and (2) The difficulties that arise when applying statistical methods to such data sets. Chris received a BA from the Mathematics Department at UCSD, an MSc from the Computer Science and Engineering Department at OSU, and a PhD from the College of Computing at Georgia Tech. He is the recipient of a 2008 Alfred P. Sloan Foundation Research Fellowship, a National Science Foundation CAREER award, a 2007 ACM SIGMOD Best Paper Award, a 2009 ACM SIGKDD Best Paper Runner-Up, a 2017 ICDE Best Paper Award, and a 2019 VLDB Best Paper Runner-Up.

REFERENCES


