Compiling Classical ML Pipelines into Tensor Computations for One-size-fits-all Prediction Serving

Supun Nakandala[1], **Gyeong-In Yu**[2], **Markus Weimer**[3], **Matteo Interlandi**[3]
[1]University of California San Diego, [2]Seoul National University, [3]Microsoft
snakanda@eng.ucsd.edu, gyeongin@snu.ac.kr,
{mweimer,mainterl}@microsoft.com

Abstract

Recent advances in Deep Neural Networks (DNNs) and the subsequent explosion of DNN frameworks have fostered the creation of a new class of systems. ONNX, TVM, and TensorRT are notable examples of such systems: they share the same goal of providing a runtime for DNN model inference with state-of-the-art performance, ease of deployment on hardware accelerators (e.g., GPUs), and portability across platforms and devices. Yet, in the enterprise space data is mostly tabular and classical Machine Learning (ML) techniques such as tree methods are frequently used, often within complex pipelines composed of data featurizers and feature selection operators. Unfortunately, in the classical ML space no unified inference serving system exists. Therefore, developers are forced to resort to bespoken solutions or subpar performance. In this work we present HUMMINGBIRD: a system able to compile classical ML pipelines end-to-end into tensor computations. It thereby seamlessly leverages the features provided by DNN inference systems, e.g., ease of deployment, operator optimizations and GPU support. We discuss the challenges, our initial system prototype, and promising initial empirical results.

1 Introduction

Machine Learning (ML) infused applications are becoming ever more pervasive [18]. The expectation is that in the future, ML will play a major role in our day to day life [6, 9]. This outlook has been boosted by recent advances in Deep Neural Networks (DNNs) [24, 32, 23] and by the subsequent explosion in the number of DNNs frameworks [30, 13, 11, 12], DNN-specific hardware accelerators [25, 22, 1, 2], and DNN-enabled devices [3]. In this fast-evolving world, systems such as ONNX Runtime [8], TVM [21], TensorRT [17], and nGraph [14] have been developed with the goal of simplifying the deployment of trained DNN models by providing a target runtime for prediction serving. These systems mainly operate at the abstraction level of tensor operations, and are capable of executing arbitrary tensor computation graphs implemented in any of the major DNN frameworks, often by first running optimization passes, and by supporting different hardware backends.

Yet, if we look at how the majority of enterprises currently do ML [26], we notice that the situation is very different. For instance, in enterprises valuable data is often stored in tabular format [18], where classical ML techniques such as linear models and tree ensemble methods are often more effective. In this scenario, Data Scientists build *model pipelines* by composing data featurizers, feature selectors and ML models into Directed Acyclic Graphs (DAGs) of operators, whereby supporting end-to-end (1) model deployment, (2) optimizations, and (3) execution on hardware accelerators is arduous. The same tools and systems used for training the model pipelines are often used for prediction serving, not always with good performance results (e.g., if containerized execution is necessary to run Python code in a non-Python production environment [29]). Alternatively, bespoken solutions can be implemented [28, 5], but the process is not scalable e.g., among different products and devices, and if custom kernels are required for exploiting state-of-the-art hardware.

To overcome the above limitations, in this work we present HUMMINGBIRD: a system where trained classical ML pipelines are end-to-end compiled into tensor computations, therefore allowing the

execution of classical ML pipelines on DNN prediction serving runtimes. HUMMINGBIRD enables a significant reduction in engineering effort, and allows the exploitation of the many optimizations already available in DNN prediction serving systems, as well as enabling execution on hardware accelerators, and ease of deployment on devices (e.g., IoT) and platforms (e.g., web browser). In building HUMMINGBIRD we had to overcome several challenges. First, operators in classical ML pipelines are a mix of both linear algebra (*arithmetic*) operators (e.g., generalized linear models, feature scaling) and *algorithmic* operators (e.g., random forest, gradient boosting trees, feature hashing); compiling algorithmic operators into tensor computations is a non-trivial task. Second, since our focus is prediction serving, the requirement is low latency and efficient inference performance whereby we need to assure that compiled pipelines have reasonable performance. Third, we want to achieve system generality with support for many classical operators, while at the same time maintaining the ability to compile the source pipelines into many target environments including CPU, GPU, and other hardware accelerators.

In this paper we will describe how HUMMINGBIRD is able to address the above challenges by embracing techniques from both compilers and database optimizers. Furthermore, we will show that by following such approach, HUMMINGBIRD opens up a whole array of novel optimizations for classical ML pipelines. This includes cost-based operator compilation strategy selection, DAG transformations, and cross-operator optimizations. Early empirical results show that our approach can enable more than 10× speedup compared to original tools/systems for classical ML inference, while enabling seamless hardware acceleration using GPUs.

2 System Overview

HUMMINGBIRD takes in a pre-trained classical ML pipelines as input and compiles it into a DAG of tensor computations. Unlike DNN-based models, which are expressed using low-level tensor operators, classical ML methods are expressed using a mix of high-level arithmetic and algorithmic operators. Feature scaling, one-hot encoding, and random forest evaluation are examples of some of those operators. Fig. 1 (A) shows examples of two such pipelines. During the compilation process, HUMMINGBIRD first translates the given pipeline into an intermediate representation (IR) format. Before emitting the compiled tensor DAG, it invokes it's optimizer to perform optimization passes over the IR. The high-level architecture of HUMMINGBIRD is shown in Fig. 1 (B). The current version of HUMMINGBIRD is implemented as an extension of ONNXMLTools [15] and supports compiling Scikit-learn [31] pipelines into TorchScript [10], ONNX, and TVM output formats. Adding support for other ML tools (e.g., ML.NET [19]) is left to future work.

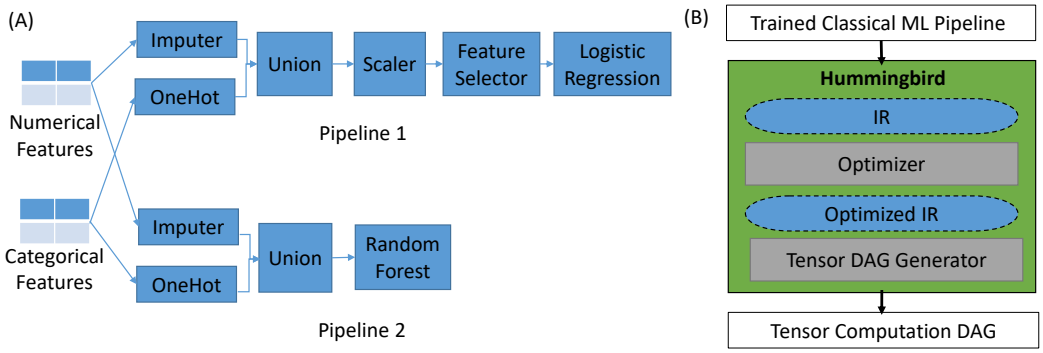


Figure 1: (A) Two examples of classical ML pipelines. (B) High-level architecture of HUMMINGBIRD.

Compiling Algorithmic Operators into Tensor Computations. One of the key challenges for HUMMINGBIRD is translating algorithmic operators into tensor computations. Algorithmic operators perform *inherently asymmetric data accesses and control flow decisions*. For example, in a decision tree ensemble potentially every tree is different from each other with respect to the structure, the decision variables, and the threshold values. But tensor operators such as matrix multiplication, index select, and tensor concatenation perform *symmetric* (bulk) *operations*. Thus, to cast algorithmic operators into tensor computations, we *introduce a degree of redundancy*. Based on the level of redundancy introduced, we can come up with different compilation strategies. The degree of redundancy is informed by model statistics such a tree-structure (for tree-based models) or sparsity (e.g., for linear models). HUMMINGBIRD can currently compile around 40 operators. We support the most popular classical ML models (e.g., tree ensembles, linear models) and featurizers (e.g., one

hot encoding, polynomial featurizer) including the ones that operate on strings. However, currently we cannot handle text featurizers such as TF-IDF. While this is a limitation, we believe its impact is low as deep learning-based models (e.g., [23]) are increasingly used for feature extraction from text. Due to space constraints we only explain the compilation of tree ensembles and related compilation strategies in §4. Next, we outline the set of optimizations currently implemented in HUMMINGBIRD.

3 System Optimizations

HUMMINGBIRD's optimizations can be broadly classified into three categories:

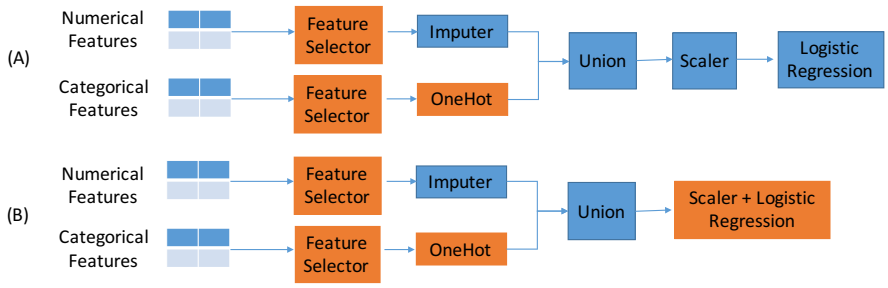


Figure 2: HUMMINGBIRD optimizations. (A) Feature selection push-down (DAG transformation), (B) Fusing feature scaling with logistic regression (cross-operator optimization).

DAG transformations. In classical ML pipelines there are opportunities to optimize the end-to-end pipeline through transformation rules, which are only applicable in the *prediction* setting. For example consider the transformed version of the Pipeline 1 of Fig. 1 shown in Fig. 2 (A). In Pipeline 1, before feeding the features to the linear model, there is a feature selection operator to discard not useful features. However, during prediction time this operator can be pushed down, similarly to projection push-down in databases. This avoids redundant computations such as scaling and one-hot encoding for discarded features, or even reading the feature at all.

For operators such as feature scaling, which performs 1-to-1 transformations, selection push-down can be easily implemented. However, for 1-to-n and n-to-1 operators such as one-hot encoding and polynomial featurizer, the operator will have to absorb the feature selection. After absorbing, it is possible that some of the original features can still be discarded as they are not used. Note that for some operators, such as feature normalizers, it is not possible to push-down the feature selection.

Even if the original pipeline doesn't have a feature selection operator, it is possible to inject one and then push it down to avoid redundant computations. 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.

Cross-operator optimizations. Our approach also opens up the opportunity to perform several cross-operator optimizations. This includes arithmetic rewrites and operator batching optimizations. For example the scaling operator and logistic regression model in the pipeline shown in Fig. 2 (A) can be merged into one operator which performs a single generic matrix multiplication (GEMM) operation. Furthermore, consider a stacked ensemble model which is composed of logistic regression, linear SVM, and a Naive Bayes models. While these models are conceptually different, during inference time all three of them are essentially performing a GEMM operation. Thus, it is possible to batch them together into one GEMM operation in order to reduce the overheads.

Runtime optimizations. Our compilation approach enables us to leverage runtime-dependant optimizations such as low-precision inference (e.g., in TensorRT, TVM), automatic operator fusion, and optimized kernel generation (e.g., TVM) which are not possible or hard to implement otherwise.

Cost-based compilation target selection. When compiling classical ML pipelines, for a given high-level operator there can be more than one compilation target. For example, in the case of decision tree-based models, HUMMINGBIRD currently supports three possible compilation strategies. In practice, none of the strategies are globally optimal, but instead they are all applicable in different situations based on the input model structure. For example, as we will see in the next section, one strategy HUMMINGBIRD uses to implement tree inference is to compute all internal decisions at once [34]. Clearly, as the size of the decision trees get bigger, this strategy will be significantly inefficient due to redundant computations. With this strategy we in fact perform $O(2^h)$ (h is the

height of the tree) computations whereas the original algorithmic operator needs to perform only $O(h)$ comparisons. Nevertheless, the above compilation strategy implement tree traversal as GEMM operations, therefore up to a certain depth level this strategy can be actually optimal performance-wise on modern hardware, where GEMM operations can run highly efficiently. The exact crossover point where one strategy outperforms the others is determined by the characteristics of the execution engine and the available hardware. For instance, we have experimental evidences showing that the above strategy performs better over shallow trees ($h \leq 3$), whereas tree-traversal-based strategies perform better over deep trees. Thus, in HUMMINGBIRD we use a cost model for compilation target selection, reminiscent to relational data management systems. Next we give details on the strategies HUMMINGBIRD uses to compile tree-base models into tensor computations.

4 Compilation Strategies for Tree-base Methods

Trained tree-based models are currently compiled by HUMMINGBIRD using three different strategies, based on run-time statistics (e.g., batch size) and tree structure.

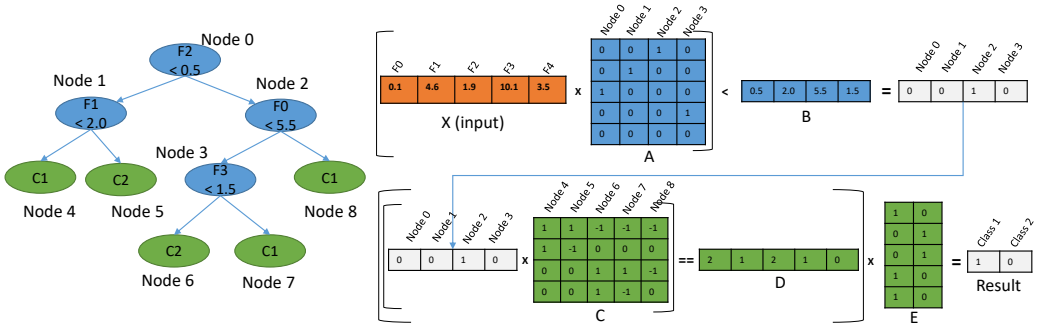


Figure 3: Decision Tree inference: GEMM strategy.

Strategy 1: GEMM. With this strategy we cast the evaluation of a decision tree as a series of three GEMM operations interleaved by two logical operators. Fig. 3 shows the high-level idea. Let m be the number features in a record, n be the number of internal nodes in the tree, l be the number of leaf nodes, and c be the number of classes.

We first create 5 matrices (A, B, C, D, and E) representing the structure of the decision tree. A is a $m \times n$ matrix having $A_{i,j}$ set to 1 if and only if the index of the feature being evaluate at the internal Node i is F j. Otherwise it is set to 0. Matrix B is a $1 \times n$ matrix with $B_{1,i}$ set to the threshold value of the internal Node i. We multiply the input X with A and then perform a *less than (<)* operation to get an indicator matrix denoting which internal nodes evaluated to true.[1] We next multiply the indicator matrix by the $n \times l$ matrix C. $C_{i,j}$ is set to 1 if internal node corresponding to row i is on the path to leaf node corresponding to node j from root with evaluating to true. It is set to -1 if the internal node is in the path and evaluates to false. Otherwise it is set to 0. The result of this multiplication operation is then subjected to *equal* condition with matrix D to get an indicator matrix denoting which leaf node evaluated to true. D is a $1 \times m$ matrix with $D_{1,i}$ set to the number of internal nodes in the path to the leaf node denoted by column i from root node which evaluates to true. The resultant indicator matrix is then multiplied by matrix E to get the final result. $E_{i,j}$ is set to 1 if and only if the leaf node corresponding to row i has class label j. Fig. 3 depicts this strategy for binary classification, but the approach can be easily extended to support multi-class and regression tasks. In the case of tree ensembles, we create the above matrices for each tree and batch them together to produce 3-dimensional tensors. As the number of leaf nodes and internal nodes can vary between trees, we pick the maximum number of leaf nodes and internal nodes for any tree as the tensor dimensions, and pad the smaller matrix slices with zeros. Similarly, when the input X contains batches with multiple records, we perform batched variants of GEMM and logical operators.

Strategy 2: TreeTraversal. In the GEMM strategy we incorporated two forms of redundancy: (1) storage redundancy by padding matrix slices; and (2) computational redundancy by evaluating all internal nodes and leaf nodes when only few of them actually need to be evaluated. In this second strategy, we try to reduce the computational redundancy by mimicking the typical tree traversal implemented using tensor operators. The high-level approach is shown in Fig. 4. Given a decision

[1] Here, we focus on $<$ nodes. Others are equally possible.

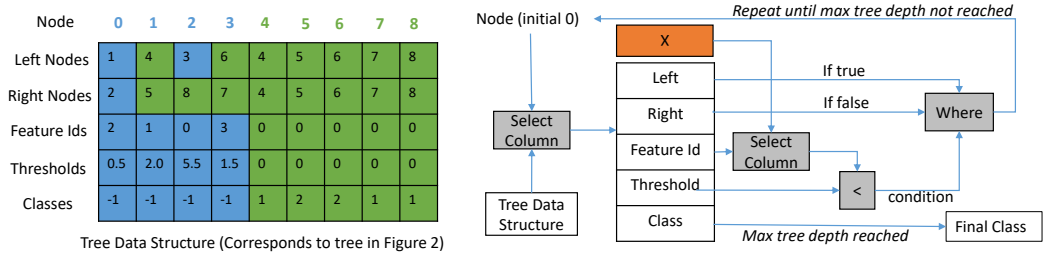


Figure 4: Decision Tree inference: TreeTraversal strategy.

tree we create a matrix maintaining the structure of the tree. Each column in this matrix corresponds to a tree node. The matrix has 5 rows and each row contains different information about each tree node. The first row contains the node id of the left child and the second row contains the node id of the right child. For leaf nodes the same parent node id is repeated. The third row contains the index of the feature that is being evaluated at each node. For leaf nodes this is set to zero. The fourth row contains the threshold value and for leaf nodes again this is set to zero. The last row contains the class label corresponding to each node and for internal nodes this is set to -1 (invalid).

Given the above tree data structure, starting with the initial node id of zero (root node), we slice the corresponding column from the structure matrix. We then select the feature id value and use that to select the corresponding feature value from the input (X). We then perform a *less than* check to determine whether the internal node is evaluated to true or false. Based on the evaluation we select either the left child id or right child id as the node id for the next iteration. This operation can be performed using the `Where` operator available in most tensor runtimes. To perform the full tree inference this process can be repeated until we reach a leaf node. However, instead of iterating in a loop we exploit the fact that we know the maximum depth of this tree to unroll the loop for max depth iterations. The tree data structure is created in way such that if it reaches leaf node before reaching the max depth, the same class label will be selected repeatedly. In the case of an ensemble with multiple trees, we batch individual tree data structures into a 3-dimensional tensor with number of tree nodes set to the maximum number of nodes in any tree. Similar to the first strategy, we pad the matrix slices corresponding to smaller trees and invoke the batched variants of the operators.

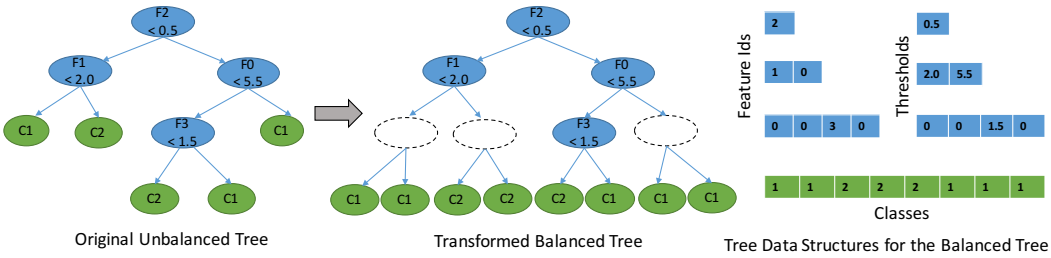
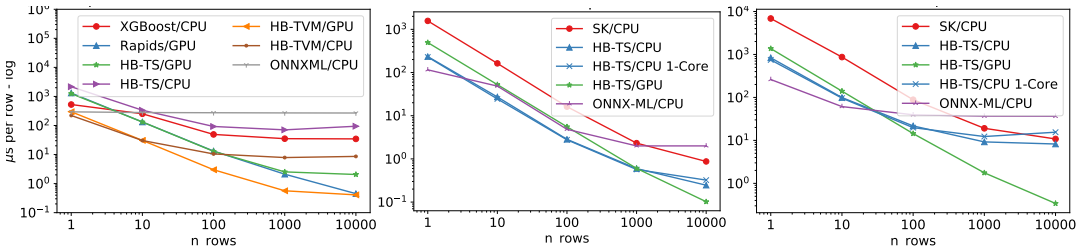


Figure 5: Decision Tree inference: BalancedTreeTraversal strategy.

Strategy 3: BalancedTreeTraversal. This strategy is similar to the previous one in spirit. However, in addition to incorporating redundancy to make all trees in the ensemble have the same number of nodes, it requires all trees in the ensemble to be complete balanced trees. Consider the example scenario shown in Fig. 5. Given a decision tree, we first find out the maximum depth of the tree. We then transform the decision tree by incorporate dummy internal nodes and replicating the corresponding leaf nodes to make the tree a balanced tree. We create three sets of data structures to maintain the structure of the decision tree: (1) indices of features checked by each internal node; (2) threshold value for each internal node; and (3) class labels. Features ids and threshold values are organized by depth levels. We exploit the balanced nature of the trees to remove the look-ups for finding the left and right child node ids of a given node. In the case of a tree ensemble, we select the maximum depth of any tree to transform the decision trees. Additionally, we batch the tree data structures corresponding to each tree, and invoke batched variants of the tensor operations.

5 Experimental Evaluation

We run some experiments to show (1) how HUMMINGBIRD performs compared to current state of the art wrt inference over tree ensembles, and (2) end-to-end performance over complete pipelines.



(a) Micro-benchmark (b) Pipeline 1 (c) Pipeline 2

Figure 6: HUMMINGBIRD performance evaluation.

Experimental Setup. We use an Azure NC6 v2 machine with 112 GB of RAM, Intel Xeon CPU E5-2690 v4 @ 2.60GHz, and Nvidia P100 GPU. The machine runs Ubuntu 18.04 with PyTorch 1.2.0, TVM 0.5, sklearn 0.21.3, XGBoost 0.90, ONNX Runtime 0.5, Rapids 0.9, and CUDA 10.0.

Gradient Boosting Micro-benchmark. We first run a micro-benchmark to evaluate the performance of translating tree-based models. We trained a XGBoost [20] model. We evaluate HUMMINGBIRD (strategy 3) with TVM (HB-TVM) and PyTorch [2] (HB-TS) backends against three baselines: XGBoost, ONNX-ML [7], and Nvidia Rapids Forest Inference Library (FIL) [16]. We use a synthetic dataset which has 100 features and 10,000 records. We vary the inference batch size and use 1000 trees with max tree depth set to 10. From the results we can see that HB-TVM outperforms the baselines (for batch size of 10k we are on par with Rapids) both on CPU and GPU. HUMMINGBIRD with PyTorch back-end is on par with the natives CPU and GPU implementations for small batch sizes, whereas it is 3× to 5× slower for larger batches. This is mainly due to TVM ability to fuse operators into efficient kernels and scheduling tuning. Note that ONNX-ML's scaling is linear here, as its current implementation of tree inference is neither optimized for batch or multicore execution.

End-to-end Pipeline Evaluation. For this set of experiments, we use the two pipelines shown in Fig. 1 (A), and train them on the Forest Covertype dataset [33]. For the feature selector operator we set the number of features to 15. For the random forest model we use 100 trees with max depth set to 10. We compare inference performance for the original pipeline implementations in Scikit-Learn and ONNX-ML against HUMMINGBIRD with PyTorch backed on varying batch sizes. The results are shown in Fig. 6b and c. In both pipelines,: (1) ONNX-ML performs best for batch of size but then it is not able to take advantage of bigger batch sizes; (2) HUMMINGBIRD always performs better than sklearn, while it performs better than ONNX-ML for large batches; (3) multicore execution do not improve performance for HUMMINGBIRD wrt single core; and (4) GPU execution provides better performance for large batches, up to 15× for pipeline 2.

6 Related Work

PRETZEL [28] is a white box prediction serving system for classical ML pipelines. PRETZEL optimizes resource usage by pooling memory and threads among operators and allows similar pipelines to share operators. In contrast, HUMMINGBIRD casts ML pipelines into tensor computations. Hence, it can take advantage of existing serving systems for DNN models and ease the deployment on many target environments. RAPIDS [4] is a ML library specifically built for NVIDIA GPUs. RAPIDS provides GPU implementations for Pandas DataFrames and classical ML algorithms. Although RAPIDS provides high-performance data manipulation and ML training implementations, inference tasks have been only added lately [16]. HUMMINGBIRD, on the other hand, provides prediction serving on various hardware and platforms, by taking advantage of existing solutions for DNN, hence minimizing engineering efforts.

7 Conclusions

Prediction serving systems for DNNs are maturing rapidly, whereas prediction serving for classical ML pipeline is still limited to ad-hoc solutions, or poor performance and limited portability. In this paper, we propose a framework to compile full pipelines into tensor operations such that DNN runtimes can be directly used for scoring classical ML models end-to-end. Our early results show that our approach is able to outperform custom (C++ and CUDA) implementations. We are currently working on porting more algorithms, exploring additional optimizations unlocked by our approach, as well as integrating HUMMINGBIRD with other optimizers [27].

[2] For the PyTorch backend, we compile models into TorchScript for better performance.

References

[1] Celebra Chip. https://www.wired.com/story/power-ai-startup-built-really-big-chip/.

[2] Graphcore IPU. https://www.graphcore.ai/.

[3] Iphone's Neural Engine. https://www.wired.com/story/how-apple-makes-ai-chip-powering-iphones-fancy-tricks/.

[4] Nvidia RAPIDS. https://developer.nvidia.com/rapids.

[5] NVIDIA RAPIDS cuML. https://github.com/rapidsai/cuml.

[6] NY Post: How AI will change every aspect of our lives. https://nypost.com/2018/09/07/how-artificial-intelligence-will-change-every-aspect-of-our-lives/.

[7] ONNX ML. https://github.com/onnx/onnx/blob/master/docs/Operators-ml.md.

[8] ONNX Runtime. https://github.com/microsoft/onnxruntime.

[9] ORACLE Magazine: It's pervasive, AI is everywhere. https://blogs.oracle.com/oraclemagazine/its-pervasive-ai-is-everywhere.

[10] TorchScript Documentation. https://pytorch.org/docs/stable/jit.html.

[11] CNTK. https://docs.microsoft.com/en-us/cognitive-toolkit/, 2018.

[12] MXNet. https://mxnet.apache.org/, 2018.

[13] TensorFlow. https://www.tensorflow.org, 2018.

[14] nGraph. https://www.ngraph.ai/, 2019.

[15] ONNXMLTools. https://github.com/onnx/onnxmltools, 2019.

[16] RAPIDS Forest Inference Library. https://medium.com/rapids-ai/rapids-forest-inference-library-prediction-at-100-million-rows-per-second-19558890bc35, 2019.

[17] Tensor-RT. https://developer.nvidia.com/tensorrt, 2019.

[18] 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.

[19] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupre, Vadim Eksarevskiy, Eric Erhardt, Costin Eseanu, Senja Filipi, Tom Finley, Abhishek Goswami, Monte Hoover, Scott Inglis, Matteo Interlandi, Shon Katzenberger, Najeeb Kazmi, Gleb Krivosheev, Pete Luferenko, Ivan Matantsev, Sergiy Matusevych, Shahab Moradi, Gani Nazirov, Justin Ormont, Gal Oshri, Artidoro Pagnoni, Jignesh Parmar, Prabhat Roy, Sarthak Shah, Mohammad Zeeshan Siddiqui, Markus Weimer, Shauheen Zahirazami, and Yiwen Zhu. Machine learning at microsoft with ML.NET, 2019.

[20] 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.

[21] 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.

[22] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38:8–20, March 2018.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv e-prints*, page arXiv:1810.04805, Oct 2018.

[24] Hany Hassan Awadalla, Anthony Aue, Chang Chen, Vishal Chowdhary, , Christian Federmann, Xuedong Huang, Marcin Junczys-Dowmunt, Will Lewis, , Shujie Liu, Tie-Yan Liu, Renqian Luo, Arul Menezes, Tao Qin, Frank Seide, Xu Tan, Fei Tian, Lijun Wu, Shuangzhi Wu, Yingce Xia, Dongdong Zhang, Zhirui Zhang, and Ming Zhou. Achieving human parity on automatic chinese to english news translation. arXiv:1803.05567, March 2018.

[25] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, June 2017.

[26] Kaggle. Kaggle data science survey. https://www.kaggle.com/surveys/2017, 2019.

[27] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. Extending relational query processing with ml inference, 2019.

[28] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, October 2018. USENIX Association.

[29] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Markus Weimer, and Matteo Interlandi. From the edge to the cloud: Model serving in ML.NET. *IEEE Data Eng. Bull.*, 41(4):46–53, 2018.

[30] 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.

[31] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[32] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, Dec 2015.

[33] UCI Machine Learning Repository. Covertype Data Set. https://archive.ics.uci.edu/ml/datasets/Covertype, 2019.

[34] Gyeong-In Yu, Saeed Amizadeh, Artidoro Pagnoni, Byung-Gon Chun, Markus Weimer, and Matteo Interlandi. Making classical machine learning pipelines differentiable: A neural translation approach. *CoRR*, abs/1906.03822, 2019.