



A Tensor Compiler for Unified Machine Learning Prediction Serving

Supun Nakandala^{c,*}, Karla Saur^m, Gyeong-In Yu^{s,*}, Konstantinos Karanasos^m,
Carlo Curino^m, Markus Weimer^m, Matteo Interlandi^m

^mMicrosoft, ^cUC San Diego, ^sSeoul National University

{<name>.<surname>}@microsoft.com, snakanda@eng.ucsd.edu, gyeongin@snu.ac.kr

Abstract

Machine Learning (ML) adoption in the enterprise requires simpler and more efficient software infrastructure—the bespoke solutions typical in large web companies are simply untenable. Model scoring, the process of obtaining predictions from a trained model over new data, is a primary contributor to infrastructure complexity and cost as models are trained once but used many times. In this paper we propose HUMMINGBIRD, a novel approach to model scoring, which compiles featurization operators and traditional ML models (e.g., decision trees) into a small set of tensor operations. This approach inherently reduces infrastructure complexity and directly leverages existing investments in Neural Network compilers and runtimes to generate efficient computations for both CPU and hardware accelerators. Our performance results are intriguing: despite replacing imperative computations (e.g., tree traversals) with tensor computation abstractions, HUMMINGBIRD is competitive and often outperforms hand-crafted kernels on micro-benchmarks on both CPU and GPU, while enabling seamless end-to-end acceleration of ML pipelines. We have released HUMMINGBIRD as open source.

1 Introduction

Enterprises increasingly look to Machine Learning (ML) to help solve business challenges that escape imperative programming and analytical querying [35]—examples include predictive maintenance, customer churn prediction, and supply-chain optimizations [46]. To do so, they typically turn to technologies now broadly referred to as “*traditional ML*”, to contrast them with Deep Neural Networks (DNNs). A recent analysis by Amazon Web Services found that 50 to 95% of all ML applications in an organization are based on traditional ML [38]. An analysis of 6M notebooks in public GitHub repositories [64] paints a similar picture: NumPy [69], Matplotlib [11], Pandas [7], and scikit-learn [62] are the four most used libraries—all four provide functions for traditional ML. As a point of comparison with DNN frameworks, scikit-learn is used about 5 times more than PyTorch [61] and

TensorFlow [13] combined, and growing faster than both. Acknowledging this trend, traditional ML capabilities have been recently added to DNN frameworks, such as the ONNX-ML [4] flavor in ONNX [25] and TensorFlow’s TFX [39].

When it comes to owning and operating ML solutions, enterprises differ from early adopters in their focus on long-term costs of ownership and amortized return on investments [68]. As such, enterprises are highly sensitive to: (1) complexity, (2) performance, and (3) overall operational efficiency of their software infrastructure [14]. In this work we focus on *model scoring* (i.e., the process of getting a prediction from a trained model by presenting it with new data), as it is a key driving factor in each of these regards. First, each model is trained once but used multiple times for scoring in a variety of environments, thus scoring dominates infrastructure complexity for deployment, maintainability, and monitoring. Second, model scoring is often in the critical path of interactive and analytical enterprise applications, hence its performance (in terms of latency and throughput) is an important concern for enterprises. Finally, *model scoring is responsible for 45-65% of the total cost of ownership of data science solutions* [38].

Predictive Pipelines. The output of the iterative process of designing and training traditional ML models is *not just* a model but 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 [64] 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 scoring: *the combinatorial explosion of supporting many operators*

*The work was done while the author was at Microsoft.

(and frameworks) across multiple target environments. Figure 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, [64] 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. Our expectation is that M is also destined to grow as ML is applied more widely across a broad range of enterprise applications and hardware (e.g., [1, 15, 30, 48, 49]). From the vantage point of implementing runtimes for model scoring, 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 [3] for GPU), traditional ML operators are only implemented for CPUs.

This state of affairs is in contrast with the DNN 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 evolved optimizations [33, 41] but also facilitates support for diverse environments (such as mobile devices [26], web browsers [32], and hardware accelerators [15, 48, 49]), unlocking new levels of performance and portability.

Our Solution. To bypass this $N \times M$ explosion in implementing traditional ML operators, we built HUMMINGBIRD (HB for short). HB leverages compilation and optimization techniques to translate a broad set of traditional 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 1 (bottom). This is also the key intuition behind the ONNX model format [25] and its various runtimes [6]. However, with HB 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 DNN frameworks. This allows us to piggyback on existing investments in DNN compilers, runtimes, and specialized-hardware, and reduce the challenge of “running K operators across M environments” for traditional ML to just $O(N)$ operator translations. This leads to improved performance and portability, and reduced infrastructure complexity.

Contributions. In this paper we answer three main questions:

1. Can traditional 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 HB help in reducing software complexity and improving model portability?

Concretely, we: (1) port thousands of benchmark predictive pipelines to two DNN backends (PyTorch and TVM);

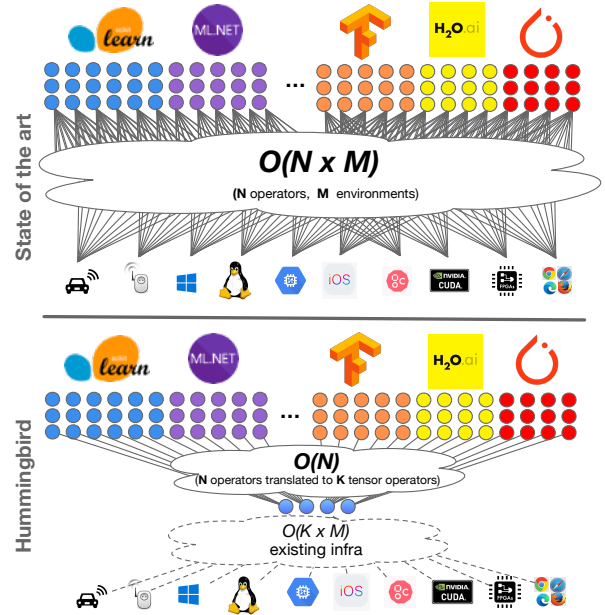


Figure 1: Prediction serving complexity: state-of-the-art (top) vs. HUMMINGBIRD (bottom).

(2) show that we can seamlessly leverage hardware accelerators and deliver speedups of up to $3\times$ against hand-crafted GPU kernels, and up to $1200\times$ 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.

HB is open source under the MIT license¹, and is part of the PyTorch ecosystem [28]. We are integrating HB with other systems, such as the ONNX converters [58].

Organization. The remainder of the paper is organized as follows. Section 2 provides some background, and Section 3 presents an overview of HB. Section 4 describes the compilation from traditional ML to tensor computations, whereas Section 5 discusses various optimizations. Section 6 presents our evaluation. Section 7 is related work, then we conclude.

2 Background and Challenges

We first provide background on traditional ML and DNNs. We then explain the challenges of compiling traditional ML operators and predictive pipelines into tensor computations.

2.1 Traditional ML and DNNs

Traditional Predictive Pipelines. The result of the data science workflow over traditional ML are predictive pipelines, i.e., DAG of operators such as trained models, preprocessors, featurizers, and missing-value imputers. The process of presenting a trained predictive pipeline with new data to obtain a prediction is referred to in literature interchangeably as: model scoring/inference/serving, pipeline evaluation, or prediction serving. We favor model scoring in our writing.

¹<https://github.com/microsoft/hummingbird>

Packaging a trained pipeline into a single artifact is common practice [36]. These artifacts are then embedded inside host applications or containerized and deployed in the cloud to perform model scoring [43, 63]. ML.NET [36] (.NET-based), scikit-learn [62] (Python-based), and H₂O [9] (Java-based) are popular toolkits to generate pipelines. However, they are primarily optimized for training. Scoring predictive pipelines is challenging, as their operators are implemented in imperative code and do not follow a shared abstraction. Supporting every operator in all target environments requires a huge effort, which is why these frameworks have limited portability.

DNNs. Deep Neural Networks (DNNs) are a family of ML models that are based on artificial neurons [47]. They take raw features as input and perform a series of transformation operations. Unlike traditional ML, transformations in DNNs are drawn from a common abstraction based on tensor operators (e.g., generic matrix multiplication, element-wise operations). In recent years, DNNs have been extremely successful in vision and natural language processing tasks [45, 54]. Common frameworks used to author and train DNNs are TensorFlow [13], PyTorch [61], CNTK [10], and MXNet [12]. While these frameworks can also be used to perform model scoring, next we discuss systems specifically designed for that.

Runtimes for DNN Model Scoring. To cater to the demand for DNN model inference, a new class of systems has emerged. ONNX Runtime (ORT) [5] and TVM [41] are popular examples of such systems. These capitalize on the relative simplicity of neural networks: they accept a DAG of tensor operations as input, which they execute by implementing a small set of highly optimized operator kernels on multiple hardware. Focusing on just the prediction serving scenario also enables these systems to perform additional inference-specific optimizations, which are not applicable for training. HB is currently compatible with all such systems.

2.2 Challenges

HB combines the strength of traditional ML pipelines on structured data [56] with the computational and operational simplicity of DNN runtimes for model scoring. 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, HB only needs to compile 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.

Challenge 1: *How can we map traditional predictive pipelines into tensor computations?* Pipelines are generally composed of operators (with predictive functions) of two classes: *algebraic* (e.g., scalars 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 HB 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.

Challenge 2: *How can we achieve efficient execution for tensor-compiled traditional ML operators?* The ability to compile 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 tensor runtimes naturally support execution on hardware accelerators, tree-based methods and commonly used data transformations are well known to be difficult to accelerate [42], even using custom-developed implementations.

3 System Overview

In this section we explain our approach to overcome the challenges outlined in Section 2.2, and present HB's architecture and implementation details. We conclude this section by explaining assumptions and limitations.

3.1 High-level Approach

In HB, 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 traditional 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 operators into tensor operations that lead to efficient computations by leveraging state-of-the-art DNN runtimes.

For a given traditional ML operator, there exist different strategies for compiling it to tensor computations, each introducing a different degree of redundancy. We discuss such strategies for representative operators in Section 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, runtime-independent optimizations at the pipeline level, and runtime-specific optimizations at the execution level* enable HB to further improve predictive pipelines performance end-to-end. The dichotomy between runtime-independent and runtime-specific optimizations allow us to both (1) apply optimizations unique to traditional ML and not captured by the DNN runtimes; and (2) exploit DNN runtime optimizations once the traditional ML is lowered into tensor computations. Finally, HB is able to run end-to-end pipelines on the hardware platforms supported by the target DNN runtimes.

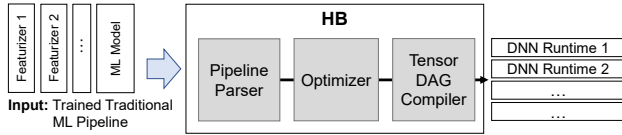


Figure 2: High-level architecture of HB.

3.2 System Architecture and Implementation

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

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”). HB 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). HB 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. HB parser is extensible, allowing users to easily add new extractor functions. HB currently supports over 40 scikit-learn operators (listed in Table 1), as well as parsers for XGBoost [40], LightGBM [51], and ONNX-ML [4]. At the end of the parsing phase, the input pipeline is “logically” represented in HB as a DAG of containers storing all the information required for the successive phases. HB parser is based on skl2onnx [31].

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 HB supports different operator implementations based on the extracted parameters, the Optimizer annotates the container with the compilation strategy to be used for that specific operator (5.1). During the second pass, HB tries to apply runtime-independent optimizations (5.2) over the DAG.

Tensor DAG Compiler. 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 2). The generated module is then exported into the target runtime format. The current version of HB supports PyTorch/TorchScript, ONNX, and TVM output formats. The runtime-specific optimizations are triggered at this level.

Table 2: 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 1: Scikit-learn operators currently supported in HB.

Supported ML Models
LogisticRegression, SVC, NuSVC, LinearSVC, SGDClassifier, LogisticRegressionCV, DecisionTreeClassifier/Regression, RandomForestClassifier/Regression, ExtraTreesClassifier/Regressor, GradientBoostingClassifier/Regression, HistGradientBoostingClassifier/Regressor, IsolationForest, 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

3.3 Assumptions and Limitations

In this paper, 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 [65] on compiling imperative UDFs (user-defined functions) into relational algebra, and we plan to make use of such techniques in HB in the future. Second, we do not support sparse data well. We found that current support for sparse computations on DNN runtimes is primitive and not well optimized. We expect advances in DNN frameworks to improve on this aspect—TACO [52] 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) compiling regex-based tokenizers into tensor computations is not trivial, and (2) representing arbitrarily long text documents in tensors is still an open challenge. Finally, HB is currently limited by single GPU memory execution. Given that several DNN runtimes nowadays support distributed processing [57, 66], we plan to investigate distributed inference as future work.

4 Compilation

HB supports compiling several algorithmic operators into tensor computations. Given their popularity [64], in Section 4.1 we explain our approach for tree-based models. Section 4.2 gives a summary of other techniques that we use for both algorithmic and arithmetic operators.

4.1 Compiling Tree-based Models

HB has three different strategies for compiling tree-based models. Strategies differ based on the degree of redundancy introduced. Table 3 explains the notation used in this section. We summarize the worst-case runtime and memory footprints of each strategy in Table 4. HB 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.

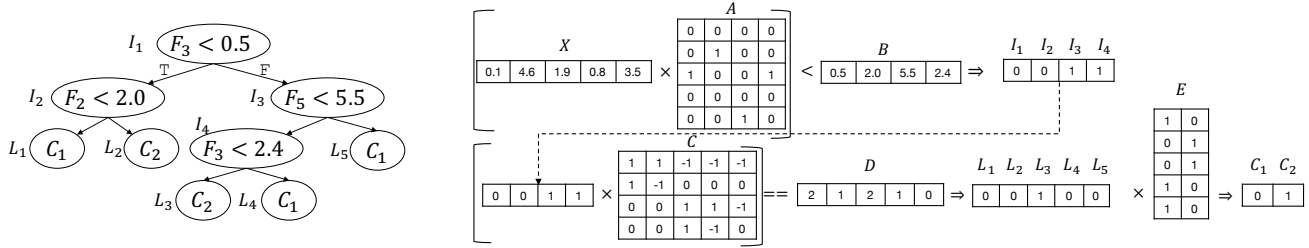


Figure 3: Compiling an example decision tree using the GEMM strategy.

Table 3: Notation used in Section 4.1

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

Table 4: Worst-case memory and runtime analysis of different tree compilation strategies, assuming the number of input records and number of trees are fixed. The notation is explained in Table 3.

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

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 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 1 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 3.

The first GEMM is used to match each input feature with

Algorithm 1 GEMM Strategy (Notation explained in Table 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 */
 $T \leftarrow \text{GEMM}(X, A)$  //  $T \in \mathbb{R}^{n \times |I|}$ 
 $T \leftarrow T < B$  //  $T \in \mathbb{R}^{n \times |I|}$ 
/* Find the leaf node which gets selected */
 $T \leftarrow \text{GEMM}(T, C)$  //  $T \in \mathbb{R}^{n \times |L|}$ 
 $T \leftarrow T == D$  //  $T \in \mathbb{R}^{n \times |L|}$ 
/* Map selected leaf node to class label */
 $R \leftarrow \text{GEMM}(T, E)$  //  $R \in \mathbb{R}^{n \times |C|}$ 

```

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 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 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. 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

Table 5: Additional notation used in Strategy 2: TreeTraversal

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

Algorithm 2 TreeTraversal Strategy (Notation in Tables 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. */
   $T_F \leftarrow \text{Gather}(N_F, T_I)$  //  $T_F \in \mathbb{Z}^n$ 
   $T_V \leftarrow \text{Gather}(X, T_F)$  //  $T_V \in \mathbb{R}^n$ 
  /* Find the threshold, left child and right child */
   $T_T \leftarrow \text{Gather}(N_T, T_I)$  //  $T_T \in \mathbb{R}^n$ 
   $T_L \leftarrow \text{Gather}(N_L, T_I)$  //  $T_L \in \mathbb{Z}^n$ 
   $T_R \leftarrow \text{Gather}(N_R, T_I)$  //  $T_R \in \mathbb{Z}^n$ 
  /* Perform logical evaluation. If true pick from  $T_L$ ;
else from  $T_R$ . */
   $T_I \leftarrow \text{Where}(T_V < T_T, T_L, T_R)$  //  $I \in \mathbb{Z}^n$ 
end
/* Find label for each leaf node */
 $R \leftarrow \text{Gather}(N_C, T_I)$  //  $R \in \mathbb{Z}^n$ 

```

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 2 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 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 compilation 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 compile 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.

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.

Table 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.
$N'_F \in \mathbb{Z}^{ I' }$	$N'_{F_i} = k \iff I'_i \text{ evaluates } F_k$
$N'_T \in \mathbb{R}^{ I' }$	$N'_{T_i} = \text{ThresholdValue}(I'_i)$
$N'_C \in \mathbb{Z}^{ L' \times C }$	$N'_{C_{i,k}} = \begin{cases} 1, N_i \xrightarrow{\text{map to}} C_k \\ 0, \text{Otherwise} \end{cases}$

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 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 3 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 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

Algorithm 3 PTT Strategy (Notation in Tables 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. */
   $T_F \leftarrow \text{Gather}(N_F, T_I)$  //  $T_F \in \mathbb{Z}^n$ 
   $T_V \leftarrow \text{Gather}(X, T_F)$  //  $T_V \in \mathbb{R}^n$ 
  /* Find the threshold */
   $T_T \leftarrow \text{Gather}(N_T, T_I)$  //  $T_T \in \mathbb{R}^n$ 
  /* Perform logical evaluation. If true pick left child;
  else right child. */
   $T_I \leftarrow 2 \times T_I + \text{Where}(T_V < T_T, 0, 1)$  //  $I \in \mathbb{Z}^n$ 
end
/* Find label for each leaf node */
 $R \leftarrow \text{Gather}(N'_C, T_I)$  //  $R \in \mathbb{Z}^n$ 
```

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.

4.2 Summary of Other Techniques

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

Exploiting Automatic Broadcasting. Broadcasting [21] 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 HB, we heavily use this feature to execute some computation over multiple inputs. For example, consider performing an 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 $R = \text{Equal}(X, V)$, $R \in \{0, 1\}^{n \times m}$. 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 an ML operator, we found that often picking the one which invokes fewer operators outperforms the other—even if it performs 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 $R_i = [X_{i,1}^2, \dots, X_{i,d}^2, X_{i,1}X_{i,2}, \dots, X_{i,d-1}X_{i,d}]$. One way to implement this operator is to compute each new feature separately by first Gathering the corresponding in-

put feature columns, perform an element-wise Multiplication, and concatenate all new features. However, this approach requires performing $d^2 + d + 1$ operations and hence is highly inefficient due to high operator scheduling 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 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 [37] 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 intermediate tensors.

Fixed Length Restriction on String Features. Features with strings of arbitrary lengths pose a challenge for HB. Strings are commonly used in categorical features, and operators like one-hot encoding and feature hashing natively support strings. To support string features, HB 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 compile time by HB. Fixed length strings are then encoded into an int8.

5 Optimizations

In this section we discuss the key optimizations performed by the HB's Optimizer: heuristics for picking operator strategies (Section 5.1) and runtime-independent optimizations (Section 5.2). Recall that our approach also leverages runtime-specific optimizations at the Tensor Compiler level. We refer to [8, 41] for runtime-specific optimizations.

5.1 Heuristics-based Strategy Selection

For a given classical ML operator, there can be more than one compilation 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 8) 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.

5.2 Runtime-independent Optimizations

We discuss two novel optimizations, which are unique to HB. HB’s approach of separating the prediction pipeline from training pipeline, and representing them in a logical DAG before compilation 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 ML model [44]. However, during scoring, 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 but through user-defined table functions, which in our case are the 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 [55], 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.

6 Experimental Evaluation

In our experimental evaluation we report two micro-benchmark experiments showing how HB performs compared to current state-of-the-art for inference over (1) tree ensembles (Section 6.1.1); (2) other featurization operators and ML models (Section 6.1.2). Then we evaluate the optimizations by showing: (1) the need for heuristics for picking the best tree-model implementation (Section 6.2.1); and (2) the benefits introduced by the runtime-independent optimizations (Section 6.2.2). Finally, we conduct an end-to-end evaluation using pipelines (Section 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.*

6.1 Micro-benchmarks

6.1.1 Tree Ensembles

Setup. This experiment is run over a set of popular datasets used for benchmarking gradient boosting frameworks [22]. We first do a 80%/20% train/test split over each dataset. Successively, we train a scikit-learn *random forest*, XGBoost [40], and LightGBM [51] 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 HB 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 batch sizes. We compare the results against HB with different runtime backends and an ONNX-ML version of the model generated using ONNXMLTools [18]. When evaluating over GPU, we also compared against NVIDIA RAPIDS Forest Inference Library (FIL) [29]. We don't compare against GPU implementations for XGBoost or LightGBM because we consider FIL as state-of-the-art [19]. 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 [22]. 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 one single record is scored at a time; (3) scaling experiments by varying batch sizes, both over CPU and GPU; (4) evaluation on how HB behaves on different GPU generations; (5) dollar cost per prediction; (6) memory consumption; (7) validation of the produced output wrt scikit-learn; and finally (8) time spent on compiling the models.

Batch Inference. Table 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 \times . This is because ONNX-ML v1.0 is not optimized for batch inference.
2. Looking at the HB'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 \times faster compared to the baseline solutions.

Let us look now at the GPU numbers of Table 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 \times compared to CPU baselines.²
2. Looking at HB backends, TorchScript is about 2 to 3 \times slower compared to RAPIDS. TVM is instead the faster solution on 14 experiments out of 18, with a 10% to 20% improvement wrt RAPIDS.

²The original FIL blog post [19] claims GPU acceleration to be in the order of 28 \times for XGBoost, versus close to 300 \times 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.

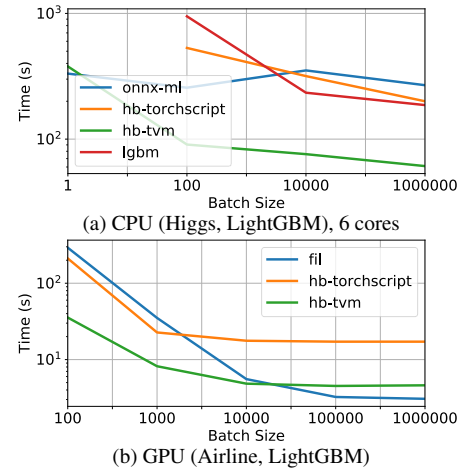


Figure 4: Performance wrt scaling the batch size.

The results are somehow surprising: HB 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 single record is scored 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 6.1.1, 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 8:

1. Unlike the batch scenario, ONNX-ML is much faster compared to scikit-learn, in some cases even more than 100 \times . 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 \times 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 \times 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 HB's backends change with the batch size. Figures 4a and 4b 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

Table 7: Batch Experiments (10K records at-a-time) for both CPU (6 cores) and GPU. Reported numbers are in seconds.

Algorithm	Dataset	Baselines (CPU)		HB CPU			Baselines (GPU)		HB GPU	
		Sklearn	ONNX-ML	PyTorch	TorchScript	TVM	RAPIDS FIL	TorchScript	TVM	
Rand. Forest	Fraud	2.5	7.1	8.0	7.8	3.0	not supported	0.044	0.015	
	Epsilon	9.8	18.7	14.7	13.9	6.6	not supported	0.13	0.13	
	Year	1.9	6.6	7.8	7.7	1.4	not supported	0.045	0.026	
	Covtype	5.9	18.1	17.22	16.5	6.8	not supported	0.11	0.047	
	Higgs	102.4	257.6	314.4	314.5	118.0	not supported	1.84	0.55	
	Airline	1320.1	timeout	timeout	timeout	1216.7	not supported	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	not supported	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	not supported	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	

Table 8: Request/response times in seconds (one record at-a-time).

Algorithm	Dataset	Baselines		HB		
		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 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

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 4b), 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

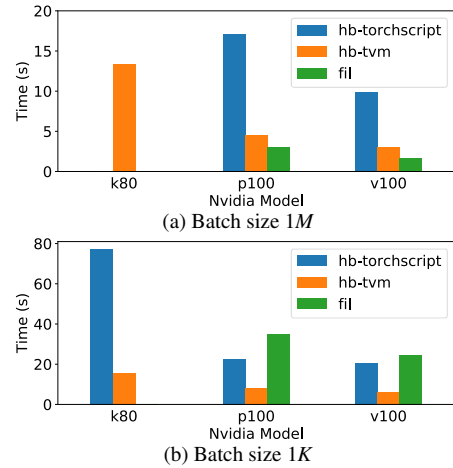


Figure 6: Performance across GPUs for Airline, LightGBM

slower than TorchScript for small batch sizes, but it scales better than HB. 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 HB with TVM runtime over large batches.

Scaling Hardware. We tested how RAPIDS FIL and HB (TorchScript and TVM) scale as we change the GPU model. For this experiment we tried both with a large batch size (1M records, Figure 6 (a)) to maximize hardware utilization, and a smaller batch size (1K, Figure 6 (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:

Table 10: Conversion times (in seconds) over one core.

Algorithm	Dataset	ONNX-ML	HB		
			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

(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 because we don't utilize the full hardware; (3) TorchScript and TVM runtimes for HB scale similarly on different hardware, although TVM is consistently 4 to 7× faster; (4) FIL scales similarly to HB, 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 1M it fails on the K80 with an OOM exception. Finally, we also were able to run HB on the new Graphcore IPU [15] over a single decision tree.

Cost. Figure 7 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 significantly higher (between 10×-120×) across all experiments.³ 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 [2]. The numbers are reported in Table 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 effi-

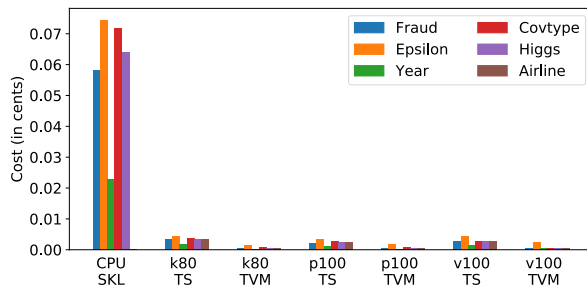


Figure 7: Cost for random forest 100k samples, batch size of 1K.

cient. ONNX-ML consumes from 10% to 50% more memory, while HB with TorchScript runtime consumes from 50% to about 2× more memory than scikit-learn. Conversely, TVM consumes from 2× to 3× more memory wrt scikit-learn. We think that TVM is more 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 7, 9 of them returned no mismatches for HB, 12 in the ONNX-ML case. Among the mismatches, the worst case for HB is random forest with Covtype where we have 0.8% of records differing from the original scikit-learn output. For the Epsilon dataset, HB 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 10 shows the time it takes to convert a trained model into a target framework. 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; HB with PyTorch backend is constantly about 2× to 3× faster wrt 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 compilation 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.

³Note: airline times out for random forest for CPU with 1K batch.

6.1.2 Operators

Setup. This micro-benchmark is a replication of the suite comparing scikit-learn and ONNX-ML operators [17]. We test all scikit-learn operators of the suite that are supported by both ONNX-ML and HB (minus tree ensembles models). The total number of tested operators is 13, and they are a mix of 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 [23] 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.

Table 11: Batch experiments for operators on both CPU (1 core) and GPU. Numbers are in milliseconds. (TS is short for TorchScript)

Operator	Baselines (CPU)		HB CPU		HB 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

Batch Inference. The batch numbers are reported in Table 11. On CPU, scikit-learn is faster than ONNX-ML, up to $6\times$ for polynomial featurizer, although in most of the cases the two systems are within a factor of 2. HB with TorchScript backend is competitive with scikit-learn, whereas with TVM backend HB is faster on 8 out of 13 operators, with in general a speedup of about $2\times$ compared to scikit-learn. If now we focus to the GPU numbers, we see that HB 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 instead saw for the tree ensemble models. In general, we see around $2\times$ performance improvement over the CPU runtime: only polynomial featurizer runs faster, with almost a $10\times$ improvement. TVM returns a runtime error when generating the polynomial featurizer model on GPU.

Request/response. Table 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 HB 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 $10\times$ faster on HB with TVM backend.

Table 12: Request/Response experiments for operators on CPU (single core). Reported numbers are in milliseconds.

Operator	Baselines		HB	
	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 framework. We used batch inference over 1K records. For memory consumption, the results are in line with what we already saw in Section 6.1.1. Regarding the conversion time, for ONNX-ML and HB 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 10), we confirm that these operators are simpler, even from a compilation perspective.

6.2 Optimizations

6.2.1 Tree Models Implementation

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 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 runtime backend. Each experiment is run on 1 CPU core.

Results. Figure 8 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 column on the right-hand side. In general, two things are apparent: (1) HB is as fast as or better than the

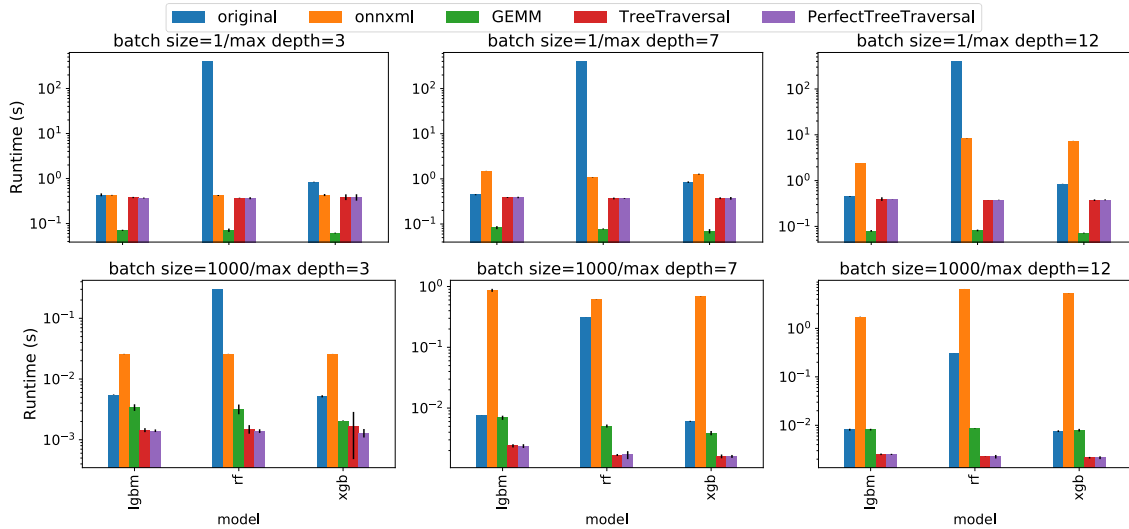


Figure 8: Comparison between the different tree strategies as we vary the batch size and depth.

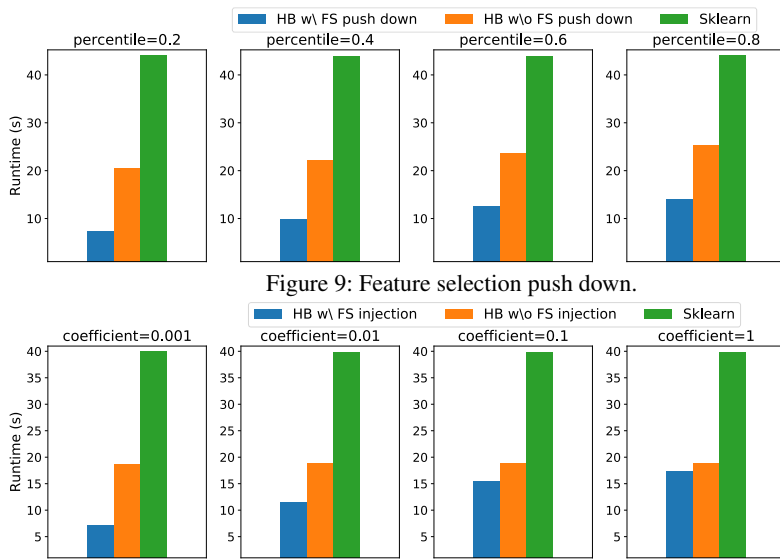


Figure 9: Feature selection push down.

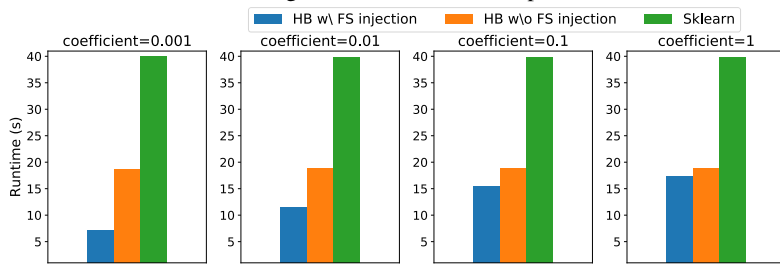


Figure 10: Feature selection injection.

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.

6.2.2 Runtime-independent Optimizations.

Next we test the optimizations described in Section 5.2.

Dataset. We use the Nomao dataset [24] with 119 features.

Feature Selection Push Down. In this experiment we measure the benefits of the feature selection push down. In Figure 9 we compare HB 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

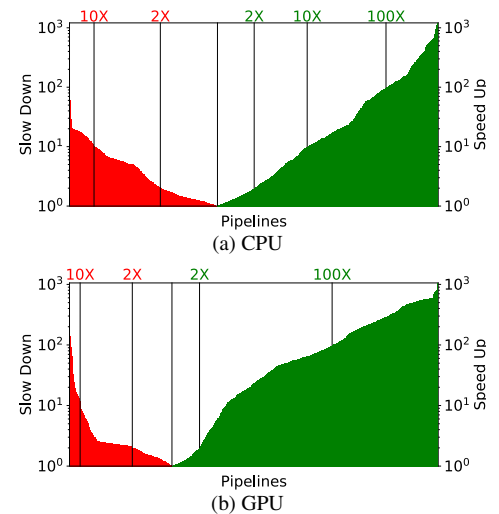


Figure 12: Speedup/slowdown of pipelines when using HB wrt baseline Sklearn.

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 HB without optimization is about $2\times$ faster than scikit-learn in evaluating the pipelines. For small percentiles, the feature selection push-down optimization delivers a further $3\times$. As we increase the percentile of features that are selected, the runtime of HB both with and without optimizations increase, although with the optimization HB is still $2\times$ 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 10 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 $3\times$ improvement wrt HB without optimization. Performance gains dissipate as we decrease the sparsity of the model.

6.3 End-to-end Pipelines

Setup. In this experiment we test HB over end-to-end pipelines. We downloaded the 72 tasks composing the OpenML-CC18 suite [27]. 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 HB, for a total of 2328 ML pipelines. Among these, 11 failed during inference due to runtime errors in HB; 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 elsewhere [64].

Datasets. For this experiment we have 72 datasets in total [27]. The datasets are a curated mix specifically designed for 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 12 summarizes the speedup / slowdown introduced by HB when scoring all 2317 pipelines. As we can see, HB is able to accelerate about 60% of the pipelines on CPU (11a). In general, the slowest pipeline gets about $60\times$ slower wrt scikit-learn, the fastest instead gets a $1200\times$ 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 (11b), still 27% of the pipelines have some slowdown. Note however that (1) both sparse and small pipelines can be detected at compile time, and therefore we can return a warning or an error; (2) DNN frameworks are continuously adding new sparse tensor operations (e.g., [34]); and (3) an option could be to add a specific runtime backend for sparse tensor operations (e.g., we have a prototype integration with TACO [52]). In general, DNN frameworks are relatively young, and HB will exploit any future improvement with no additional costs.

With GPU acceleration (Figure 11b), 73% of the pipelines show some speedup. The slowest pipeline gets about $130\times$ slower wrt 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.

7 Related Work

PyTorch [61], TensorFlow [13], MXNet [12], CNTK [10] 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 [5], nGraph [16], TVM [41], and TensorRT [20] provide optimizations and efficient execution targets, specifically for inference. To prove the versatility of our approach, we have tested HB with both PyTorch and TVM. HB 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 DNN runtime, are executed (e.g., algebraic rewrites, operator fusion). Willump [53] uses a similar two-level optimization strategy, although it targets Weld [60] as its low level runtime and therefore it cannot natively support inference on hardware accelerators. Conversely, HB casts ML pipelines into tensor computations and takes advantage of DNN serving systems to ease the deployment on target environments. Other optimizers for predictive pipelines, such as Pretzel [55], only target logical optimizations. We have integrated HB into Raven [50] as part of our bigger vision for optimizing ML prediction pipelines.

Several works deal with executing trees (ensembles) [29, 59, 67] on hardware accelerators. These systems provide a custom implementation of the PTT strategy specific to the target hardware (e.g., NVIDIA GPUs for RAPIDS FIL [29], FPGAs for [59]), and where computation is parallelized along on the tree-dimension. Alternatively, HB provides three tree inference strategies, including two novel strategies (GEMM and TT), and picks the best alternative based on the efficiency and redundancy trade-off.

8 Conclusions

In this paper, we explore the idea of using DNN frameworks as generic compilers and optimizers for heterogeneous hardware. Our use-case is “traditional” ML inference. We ported 40+ data featurizers and traditional ML models into tensor operations and tested their performance over two DNN frameworks (PyTorch and TVM) and over different hardware (CPUs and GPUs). The results are compelling: even though we target high-level tensor operations, we are able to outperform custom C++ and CUDA implementations. To our knowledge, HUMMINGBIRD is the first system able to run traditional ML inference on heterogeneous hardware.

9 Acknowledgements

We thank the anonymous reviewers and our shepherd, Chen Wenguang, for their feedback and suggestions to improve the paper. We would also like to thank Nellie Gustafsson, Gopal Vashishtha, Emma Ning, and Faith Xu for their support.

References

- [1] Cerebras Chip. <https://www.wired.com/story/power-ai-startup-built-really-big-chip/>.
- [2] Memory profiler for Python.
- [3] Nvidia RAPIDS. <https://developer.nvidia.com/rapids>.
- [4] ONNX ML. <https://github.com/onnx/onnx/blob/master/docs/Operators-ml.md>.
- [5] ONNX Runtime. <https://github.com/microsoft/onnxruntime>.
- [6] ONNX Supported Frameworks and Backends. <https://onnx.ai/supported-tools.html>.
- [7] Pandas. <https://pandas.pydata.org/>.
- [8] TorchScript Documentation. <https://pytorch.org/docs/stable/jit.html>.
- [9] H2O Algorithms Roadmap. <https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/product/flow/images/H2O-Algorithms-Road-Map.pdf>, 2015.
- [10] CNTK. <https://docs.microsoft.com/en-us/cognitive-toolkit/>, 2018.
- [11] Matplotlib. <https://matplotlib.org/>, 2018.
- [12] MXNet. <https://mxnet.apache.org/>, 2018.
- [13] TensorFlow. <https://www.tensorflow.org>, 2018.
- [14] Esg technical validation: Dell emc ready solutions for ai: Deep learning with intel. <https://www.esg-global.com/validation/esg-technical-validation-dell-emc-ready-solutions-for-ai-deep-learning-with-intel>, 2019.
- [15] Graphcore IPU. <https://www.graphcore.ai/>, 2019.
- [16] nGraph. <https://www.ngraph.ai/>, 2019.
- [17] ONNX-ML vs Sklearn Benchmark. https://github.com/xadupre/scikit-learn_benchmarks, 2019.
- [18] ONNXMLTools. <https://github.com/onnx/onnxmltools>, 2019.
- [19] RAPIDS Forest Inference Library. <https://medium.com/rapids-ai/rapids-forest-inference-library-prediction-at-100-million-rows-per-second-19558890bc35>, 2019.
- [20] Tensor-RT. <https://developer.nvidia.com/tensorrt>, 2019.
- [21] Broadcasting Semantic. <https://www.tensorflow.org/xla/broadcasting>, 2020.
- [22] Gradient Boosting Algorithm Benchmark. <https://github.com/NVIDIA/gbm-bench>, 2020.
- [23] Iris dataset. https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html, 2020.
- [24] nomao dataset. <https://www.openml.org/d/1486>, 2020.
- [25] ONNX. <https://github.com/onnx/onnx/blob/master/docs/Operators.md>, 2020.
- [26] ONNX Portable format. <https://www.infoworld.com/article/3223401/onnx-makes-machine-learning-models-portable-shareable.html>, 2020.
- [27] OpenML-CC18 Benchmark. <https://www.openml.org/s/99>, 2020.
- [28] Pytorch Ecosystem. <https://pytorch.org/ecosystem/>, 2020.
- [29] RAPIDS cuML. <https://github.com/rapidsai/cuml>, 2020.
- [30] Sambanova: Massive Models for Everyone. <https://sambanova.ai/>, 2020.
- [31] skl2onnx Converter. <https://github.com/onnx/sklearn-onnx/>, 2020.
- [32] Tensorflow JS. <https://www.tensorflow.org/js>, 2020.
- [33] Tensorflow XLA. <https://www.tensorflow.org/xla>, 2020.
- [34] The Status of Sparse Operations in Pytorch. <https://github.com/pytorch/pytorch/issues/9674>, 2020.
- [35] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avriela 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.
- [36] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupre, Vadim Eksarevskiy, Senja Filipi, Tom Finley, et al. Machine learning at Microsoft with ML.NET. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '19, page 2448–2458, New York, NY, USA, 2019. Association for Computing Machinery.
- [37] Samuel Albanie. Euclidean distance matrix trick. 2019.
- [38] 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.
- [39] 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 1387–1395, New York, NY, USA, 2017. Association for Computing Machinery.
- [40] 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.

- [41] 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.
- [42] 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.
- [43] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.
- [44] Manoranjan Dash and Huan Liu. Feature selection for classification. *Intelligent data analysis*, 1(3):131–156, 1997.
- [45] 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.
- [46] FirmAI. Machine Learning and Data Science Applications in Industry. <https://github.com/firmai/industry-machine-learning>.
- [47] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.
- [48] Intel. Machine learning fpga. <https://www.intel.com/content/www/us/en/products/docs/storage/programmable/applications/machine-learning.html>, 2020.
- [49] 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, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diermuth 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, 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. *CoRR*, abs/1704.04760, 2017.
- [50] 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.
- [51] 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.
- [52] 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.
- [53] 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.
- [54] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.
- [55] 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.
- [56] Ping Li. Robust logitboost and adaptive base class (abc) logitboost. In *n Proceedings of the Twenty-Sixth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI'10)*.
- [57] 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.
- [58] Faith Xu Matteo Interlandi, Karla Saur. Accelerate traditional machine learning models on GPU with ONNX Runtime. <https://cloudblogs.microsoft.com/opensource/2020/09/29/accelerate-machine-learning-models-gpu-onnx-runtime-hummingbird/>, 2020.
- [59] M. Owaida, H. Zhang, C. Zhang, and G. Alonso. Scalable inference of decision tree ensembles: Flexible design for cpu-fpga platforms. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sep. 2017.
- [60] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, et al. Evaluating end-to-end optimization for data analytics applications in weld. *Proceedings of the VLDB Endowment*, 11(9):1002–1015, 2018.
- [61] 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.

- [62] 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.
- [63] 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.
- [64] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Matteo Interlandi, Avrielia 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, 2019.
- [65] 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):432–444, December 2017.
- [66] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.
- [67] Toby Sharp. Implementing decision trees and forests on a gpu. In David Forsyth, Philip Torr, and Andrew Zisserman, editors, *Computer Vision – ECCV 2008*, pages 595–608, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [68] Credit Suisse. The apps revolution manifesto—volume 1: The technologies. <https://aka.ms/enterprise-application-lifespan>, 2012.
- [69] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

A Artifact Appendix

A.1 Abstract

Hummingbird compiles trained traditional ML models into tensor computation for faster inference. Hummingbird allows users to score models both on CPU and hardware accelerators.

A.2 Artifact check-list

- **Program:** PyTorch, ONNX Runtime, TVM.
- **Data set:** Fraud, Epsilon, Year, Covtype, Higgs, Airline, Iris, Nomao, OpenMLCC-18.
- **Run-time environment:** Ubuntu 18.04.
- **Hardware:** Azure NC6 v2 machine.
- **Experiments:** tree-models (Random Forest, XGBoost, LightGBM), operators (LogisticRegression, SGDClassifier, LinearSVC, NuSVC, SVC, BernoulliNB, MLPClassifier, DecisionTreeClassifier, Binarizer, MinMaxScaler, Normalizer, PolynomialFeatures, StandardScaler), end-to-end pipelines.
- **Public link:** <https://github.com/microsoft/hummingbird>.
- **Code licenses:** MIT.

A.3 Description

A.3.1 How to access

Hummingbird is open source and can be accessed directly from <https://github.com/microsoft/hummingbird>. Otherwise, Hummingbird can also be downloaded from pip with `pip install hummingbird-ml`.

A.3.2 Hardware dependencies

No specific hardware dependencies. The artifact has been evaluated on different NVIDIA GPU generations (K80, P100, V100) but it should work on any hardware supported by the target DNN runtime.

A.3.3 Software dependencies

Hummingbird requires Python ≥ 3.5 , numpy ≥ 1.15 , onnxconverter-common $\geq 1.6.0$, scikit-learn $\geq 0.21.3$, torch $\geq 1.3.1$. Additional dependencies for reproducing the results are onnxruntime ≥ 1.0 , onnxmltools $\geq 1.6.0$, xgboost ≥ 0.90 and lightgbm ≥ 2.2 , psutil, memory-profiler.

A.3.4 Data sets

For the experiments on tree algorithms we used Fraud ⁴, Epsilon ⁵, Year ⁶, Covtype ⁷, Higgs ⁸, and Airline ⁹. For the experiments on operators we instead used Iris ¹⁰. Finally, for the pipeline experiments we used OpenML-CC18 [27]. The experiment scripts automate the download and preparation of all the datasets.

A.4 Installation

Hummingbird can be installed from pip with `pip install hummingbird-ml` or by cloning the code available on GitHub and by calling `python setup.py install` from the main directory. Hummingbird will automatically detect the available backends at runtime. We refer to <https://github.com/microsoft/hummingbird/blob/master/TROUBLESHOOTING.md> for problems related to installations.

A.5 Experiment workflow

The scripts for the experiments are divided in three main folders: *trees*, *operators* and *pipelines*. Each folder contains a README.md file containing the specific instructions for that particular set of experiments.

Trees: This directory contains the script to generate the result of Section 6.1.1. We suggest to start with running `python run.py -dataset fraud,year,covtype,epsilon` (skipping `higgs/airline`) because the complete script (which can be run with just `python run.py`) over all backends and datasets takes more than one day to complete. After the script is run for the first time, the datasets and trained models are cached (in `datasets` and `models` folders, respectively), so that following executions will be faster. Several other arguments can be changed in the script (e.g., batch size, number of trees, etc.).

The output of the above commands is a JSON file reporting the training time and accuracy (if the model is not cached), and prediction (process) time in seconds, as well the peak memory used. The baseline is then compared against Hummingbird with PyTorch (`hb-pytorch`), TorchScript (`hb-torchscript`) and TVM (`hb-tvm`) backends. The entry `is_same_output` specifies whether the results of the translated models match those of the baseline (up to a tolerance of 10^{-6}). If the result is `false`, the script can be re-run with the `-validate` flag on to check the percentage of wrong results. The `-gpu` flag can be used to run the experiments on GPU.

Operators: This directory contains the scripts to reproduce the experiments of Section 6.1.2. The scripts are configured to run scikit-learn and compare it against ONNX-ML, TorchScript and TVM (the last 2 using Hummingbird), for the Iris dataset over 1 core, and with batch of 1M. `python run.py` runs the benchmark for CPU, `python run.py -gpu` runs the benchmark for GPU.

Pipelines: This directory contains the script to reproduce the experiments of Section 6.3. There are two main scripts to run for this experiment:

- `openml_pipelines.py` is used to download and train all the scikit-learn pipelines of the openML-CC18 benchmark.
- `run.py` is used to run evaluate the performance of scikit-learn and Hummingbird over the trained pipelines.

⁴<https://www.kaggle.com/mlg-ulb/creditcardfraud>

⁵<https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html>

⁶<https://archive.ics.uci.edu/ml/datasets/yearpredictionmsd>

⁷<https://archive.ics.uci.edu/ml/datasets/covtype>

⁸<https://archive.ics.uci.edu/ml/datasets/HIGGS>

⁹http://kt.ijs.si/elena_ikononovska/data.html

¹⁰<https://archive.ics.uci.edu/ml/datasets/iris>

This experiment is composed of two steps. The first step in this experiment is the generation of the prediction pipelines. This can be achieved by running `python openml_pipelines.py | tee openml-cc18.log`. This script takes several hours to run. While executing, this script will log the number of successfully trained pipelines, as well as additional statistics. Once completed, the `openml-cc18.log` file contains the statistics. Per task statistics are logged into the relative folder.

Once the first step is completed, in the second step we evaluate the scoring time of the generated pipelines, and compare the speed-ups introduced by Hummingbird against scikit-learn. This experiment can be executed both on CPU and GPU, and in both cases it takes about an hour. `python run.py` runs inference over all the generated pipelines, while `python run.py -gpu` can be used for GPU execution.

A.6 Evaluation and expected result

In May we open sourced Hummingbird (blog post: <https://azuredata.microsoft.com/articles/ebd95ec0-1eae-44a3-90f5-c11f5c916d15>). Since then we have been pushing our internal code into the open source repository, but the 2 versions do not match yet. Specifically:

- TVM integration is not complete. In our internal version we re-implemented all the operators directly in TVM's Relay but this is not a good strategy in the long term. In the open source version, we directly export Relay graphs from PyTorch models. However the exporter does not cover PyTorch 100% yet. We are however working with the TVM community for bringing full support of TVM in Hummingbird (we suggest to check the related issue #232 on Hummingbird's GitHub if interested). In practice, this means that: (1) not all operators are currently exportable into TVM; and (2) the performance we reported in the paper for TVM can be a bit different.
- The optimizer is not yet open sourced. This means that Figures 9 and 10 are not reproducible as of now. We hope to be able to bring the optimizer open source in the coming months.

Besides the above two limitations, the scripts allow the reproduction of the following main results of the paper:

- `trees` allows the reproduction of the results of Tables 7, 9 and 10. Please check the above description for specifics.

- `operators` allows the reproduction of the results of Table 11 (however not all operators will run on the TVM backend). Again, please check the related description for specifics.
- `pipelines` allows the reproduction of the results of Figure 12. Also in this case we don't cover yet 100% of the operators, but we are close.

Keep in mind that running all the experiments for completely reproducing the results will take several days.

A.7 Experiment customization

The above mentioned scripts can be customized by running them with different input arguments. For instance, Table 8 in the paper can be reproduced by setting the batch size to 1 (using the `-batch_size` argument.) in the `run.py` script.

A.8 Notes

The numbers in the paper were run on the reported VM, however:

- As this is an Azure VM, the underlying machine can receive upgrades necessitating the reinstallation of the NVidia drivers.
- The original experiments were run inside the context of an Nvidia-docker container. This setup should not have a large impact on results

Additionally, a few operators are not yet available in the open source version of Hummingbird, therefore the final coverage reported in the log file for the pipelines will be different than the one reported in the paper. To check the expected coverage once all the operators are open source, the script allows to add new operators. The same consideration holds for the operators experiment.

As a final note: to allow third-party reproducibility, we are open sourcing all the scripts used for the experiments.

A.9 AE Methodology

Submission, reviewing and badging methodology:

- <https://www.usenix.org/conference/osdi20/call-for-artifacts>