WindTunnel: Towards Differentiable ML Pipelines Beyond a Single Model

Gyeong-In Yu*
Seoul National University
gyeongin@snu.ac.kr

Saeed Amizadeh
Microsoft
saamizad@microsoft.com

Sehoon Kim
UC Berkeley
sehoonkim@berkeley.edu

Artidoro Pagnoni*
Carnegie Mellon University
apagnoni@andrew.cmu.edu

Ce Zhang
ETH Zurich
ce.zhang@inf.ethz.ch

Byung-Gon Chun
Seoul National University
FriendliAI
bgchun@snu.ac.kr

Markus Weimer
Microsoft
mweimer@microsoft.com

Matteo Interlandi
Microsoft
mainterl@microsoft.com

ABSTRACT

While deep neural networks (DNNs) have shown to be successful in several domains like computer vision, non-DNN models such as linear models and gradient boosting trees are still considered state-of-the-art over tabular data. When using these models, data scientists often author machine learning (ML) pipelines: DAG of ML operators comprising data transforms and ML models, whereby each operator is sequentially trained one-at-a-time. Conversely, when training DNNs, layers composing the neural networks are simultaneously trained using backpropagation.

In this paper, we argue that the training scheme of ML pipelines is sub-optimal because it tries to optimize a single operator at a time thus losing the chance of global optimization. We therefore propose WindTunnel: a system that translates a trained ML pipeline into a pipeline of neural network modules and jointly optimizes the modules using backpropagation. We also suggest translation methodologies for several non-differentiable operators such as gradient boosting trees and categorical feature encoders. Our experiments show that fine-tuning of the translated WindTunnel pipelines is a promising technique able to increase the final accuracy.

PVLDB Reference Format:
Gyeong-In Yu, Saeed Amizadeh, Sehoon Kim, Artidoro Pagnoni, Ce Zhang, Byung-Gon Chun, Markus Weimer, and Matteo Interlandi. WindTunnel: Towards Differentiable ML Pipelines Beyond a Single Model. PVLDB, 15(1): 11-20, 2022.
doi:10.14778/3485450.3485452

1 INTRODUCTION

The recent decade has witnessed two distinct trends in Machine Learning (ML). On one hand, the success of Deep Neural Networks (DNNs) has been the driving force of many recent advances in ML, pushing the limits of various tasks that use unstructured data such as image recognition [15, 49, 55], machine translation [11, 14, 60], and speech recognition [3, 16]. One of the key factors of this success was the power of backpropagation, which allows the DNNs to learn and extract important higher-level features for the given task. DNNs comprise multiple layers, which can be seen as multiple cascaded operators. These layers are trained simultaneously using backpropagation by which parameters can be globally estimated end-to-end to reach better minima.

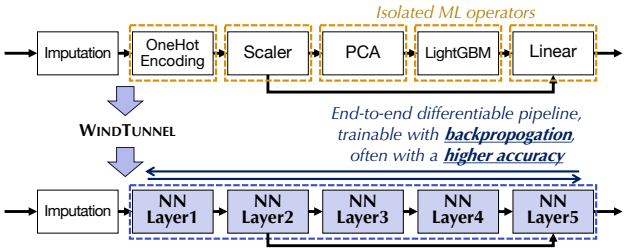


Figure 1: An illustration of WindTunnel. The input to WindTunnel is a ML pipeline and the output is its (partially) differentiable counterpart.

On the other hand, many real-world ML applications including recommendation [1, 6, 46, 47], click prediction [12, 20, 21], and malware prediction [19, 28, 38] use structured data, which is often represented in a tabular form within RDBMSs. These applications often use *classical*[1] *machine learning pipelines* composed of multiple data transformations and ML models [2, 41, 51, 56] rather than a single model. Such pipelines are Directed Acyclic Graphs (DAGs) of operators and are enriched by domain knowledge from practitioners and domain experts via feature engineering and model selection. However, these pipelines are trained sequentially by following the topological order specified in the DAG. They are not end-to-end differentiable, thus cannot take advantage of backpropagation in jointly optimizing the whole pipeline beyond a single model.

Inspired by these observations, in this paper we ask: *Can we combine the strength of backpropagation and ML pipelines?* To answer this question, we propose WindTunnel, a framework that translates operators of a given ML pipeline into differentiable Neural Network (NN) modules. The translated NN modules are wired together to form a WindTunnel pipeline (Figure 1), hence enabling end-to-end training via backpropagation. This allows us to bypass the greedy one-operator-at-a-time training scheme and boost the accuracy of the pipeline. During the translation phase, we can retain the information already acquired by training the original ML pipeline and provide a useful parameter initialization for the

*Work done while Gyeong-In Yu and Artidoro Pagnoni were at Microsoft.

Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:10.14778/3485450.3485452

[1]The term "classical" is generally used to diversify this type of ML from DNN-based approaches.

translated NN modules, making further training of WindTunnel pipeline more accurate and faster.

To demonstrate the benefits brought by WindTunnel, we conduct experiments on three large-scale real-world datasets with three ML pipelines made up of multiple operators. The results show that we can arrive at better accuracy by jointly training these operators. Furthermore, we find that WindTunnel provides informative knowledge transfer from pre-trained pipelines, along with efficient neural architecture that performs better than previous work [24]. WindTunnel currently supports several among data transforms and ML models (the full list is contained in Table 1).

Comparison with the state of the art approaches. WindTunnel has the following benefits compared to other approaches.

(1) Compared with the original ML pipeline that cannot optimize multiple operators in an end-to-end fashion, a WindTunnel pipeline has higher accuracy because of its ability to jointly fine-tune the pipeline with backpropagation. While jointly optimizing multiple operators, WindTunnel is also able to maintain the knowledge encoded in the structure of the original pipeline by experts, such as how input features are wired to operators and hyperparameters of the operators (how many trees in a Gradient Boosting Decision Tree (GBDT) model, the number of principal components for PCA, etc.).
(2) Compared with DNNs, WindTunnel leads to a higher accuracy because ML pipelines are often better than DNNs for handling tabular data, and WindTunnel successfully leverages such advantage in the translation. For example, Ke et al. [24] compared the performance of ML pipelines with various DNNs developed for tabular data including Wide&Deep [8], DeepFM [13], and PNN [45], and showed that the ML pipeline with LightGBM [23] outperforms all the DNNs for every dataset. Rendle et al. [48] also showed that Matrix Factorization [30] can outperform recent DNN-based approaches. One can try to manually design a DNN that matches the neural architecture of the WindTunnel pipeline, however, the DNN should be trained from scratch while WindTunnel provides an informative initialization point by transferring weights from the original ML pipeline.

Multi-operator pipeline vs. Single model. At this point, the readers might wonder: *What's the difference between composing a ML pipeline with multiple operators and a single model? Can't we just replace the multi-operator pipeline with a single model?* We have evidence that this is not the current trend in data science. For instance, in [44] we crawled 6 million python notebooks on GitHub and joined this information with telemetry data on the internal usage within Microsoft of ML.NET [2]. The analysis suggested that the majority of Scikit-learn [41] pipelines used in public notebooks contain 2 or more operators (with a max length of 43), whereas in ML.NET telemetry the distribution is even more tail-skewed, with few pipelines having even up to hundreds of operators. This evidence suggests that multi-operator pipelines are widely used in practice both in the open-source domain and in industry. We attribute such trend to the additional information encoded by experts in the structure of the pipeline, including how to featurize the input data and how to wire the connection between operators.

Challenges of translating non-differentiable operators. Nevertheless, noticeable challenges arise when the pipeline involves operators that are intrinsically non-differentiable, such as decision trees or word tokenization. This requires us to develop new methods in translating non-differentiable operators into differentiable NN modules. To address this challenge, we develop translation methods for a selected set of non-differentiable operators. First, we propose a translation method that translates tree ensemble (e.g., GBDT) into a batch of Multi-Layer Perceptrons (MLPs), where each MLP corresponds to a tree in the ensemble. The translated NN module (i.e., batch of MLPs) directly inherits the decision procedure of the original tree ensemble, thus the learning capacity of the NN module varies according to hyperparameters of the ensemble like number of trees. Leveraging this additional knowledge infused by the ML experts relieves the burden of laborious neural architecture tuning. We also suggest multiple parametrization levels when optimizing the translated module to balance good fit and inductive bias.

For categorical features, we translate categorical feature encoders (e.g., one-hot encoding) into embedding lookup modules. By doing so, WindTunnel learns the dense representations of sparse categorical features by exploiting the information propagated from the final loss function. The translated embedding module inherits the data transformation procedure of the original encoder, following the same principle as GBDT translation. Conversely, the original ML pipeline uses a fixed encoding logic *regardless of the final prediction result.* To the best of our knowledge, this is the first work that proposes joint optimization of categorical encoders and downstream operators (e.g., GBDT) in ML pipelines. Although the embedding technique itself is well-recognized in ML community especially in the context of deep learning [42], combining classical ML models with the embedding technique were not possible without explicit use of models with latent parameters [46]. This is because the ML models did not allow backpropagating gradients to upstream operators, while the neural translation unlocks this capability.

Practical impact. WindTunnel will be open sourced as part of Hummingbird [31, 37]: a tool recently released [57] by Microsoft enabling inference of classical ML pipelines over hardware accelerators (e.g., GPUs). Hummingbird converts ML pipelines into non-differentiable tensor computations and thus can directly leverage the capabilities of DNN runtimes [7, 40]. Hummingbird is part of the PyTorch ecosystem [59], and is integrated with ONNXMLTools [58]. WindTunnel extends Hummingbird by enabling conversion of pipelines into differentiable modules, and therefore allowing the fine-tuning of pipelines along with fast inference. Enabling training of ML pipelines over DNN runtimes and hardware accelerators has been suggested as one of the important extension to Hummingbird both from the open-source community[2] and internal conversations with product partners within Microsoft.

Limitations. As one of the first systems that focus on the differentiable translation of ML pipelines, WindTunnel by no means provides a complete solution to this challenging problem. One major limitation is that there are some operators that we cannot translate into a differentiable format yet. Word tokenization, data cleansing, and imputation are such examples. These operators require sophisticated algorithms that are too difficult to parametrize.

Since we currently do not handle these operators, WindTunnel does not translate them and keep them as they are. Nevertheless, in

[2] https://github.com/microsoft/hummingbird/issues/165

Table 1: WindTunnel's currently supported ML operators.

Supported Operators			
linear models	normalizers	categorical encoders	
SVM	PCA	LDA	KMeans
naive bayes	random forest	gradient boosting trees	
matrix factorization		factorization machine	

all the cases we studied, these non-translatable operators are placed at the beginning of the pipeline and do not affect backpropagation through the rest of the translated pipeline. Hence, we can still compute gradients and jointly optimize the downstream operators, which are the more essential parts of the ML pipeline.

Another notable limitation is that WindTunnel requires more training time than classical ML pipelines. In particular, the fine-tuning stage is more than an order of magnitude slower than the training of the original ML pipeline. This is due to the large amount of computation required for optimizing the NN modules. One can consider using recent hardware that enables faster GEMM [53] or applying distributed training techniques for sparse parameters [26] to mitigate this problem. We leave these directions as a future work.

2 PIPELINE TRANSLATION

A classical machine learning pipeline is defined as a DAG of data-processing operators, and these operators are mainly divided into two categories: (1) the *arithmetic* operators and (2) the *algorithmic* operators. Arithmetic operators are typically described by a single mathematical formula. These operators are, in turn, divided into two sub-categories of *parametric* and *non-parametric* operators. Non-parametric operators define a fixed arithmetic operation on their inputs; for example, the logistic sigmoid function can be seen as a non-parametric arithmetic operator. In contrast, parametric operators involve numerical parameters on the top of their inputs in calculating the operators' outputs. For example, an affine transform is a parametric arithmetic operator where the parameters consist of the affine weights and biases. The parameters of these operators can be potentially tuned via some training procedure.

The algorithmic operators, on the other hand, are those whose operation is not described by a single mathematical formula but rather by an algorithm. For instance, the one-hot encoder that converts categorical features into one-hot vectors is an algorithmic operator that mainly implements the look-up operation. Given a DAG of arithmetic and algorithmic operators, we propose the following procedure for translating it into a differentiable format:

(1) For an arithmetic operator, translate the mathematical formula into a neural network (NN) module. In the case of parametric operator, copy the values of the operator's parameters into the resulting NN module.
(2) For an algorithmic operator, translate the operator by rewriting the algorithm as a differentiable operation.
(3) Compose all the resulting modules into a WindTunnel pipeline by following the dependencies in the original pipeline.

The final output of the above translation process is a pipeline of NNs that provides the same prediction results (unless the translation includes approximation described in Section 2.2) as the original pipeline. Note that Step 1 and 2 in the above procedure are where the actual translation happen, and will be described in details next.

2.1 Translating Arithmetic Operators

Arithmetic operators comprise non-parametric and parametric operators, and it is straightforward to translate the former into a NN module: the mathematical function of the operator can in fact be directly rewritten using the math API provided by a DL framework like PyTorch [40]. On the other hand, parametric operators are often implicitly derived from ML models[3], which are not straightforward to translate. ML models typically consist of three key components: (1) the prediction function, (2) the loss function, and (3) the learning algorithm. While the prediction function defines the functional form of the model, the learning algorithm and the loss function define how it is trained toward what objective, respectively. Take the popular linear Support Vector Machine (SVM) as an example: the prediction function is a linear combination of input features; the loss function is the Hinge loss, and the learning algorithm is gradient descent in the dual space.

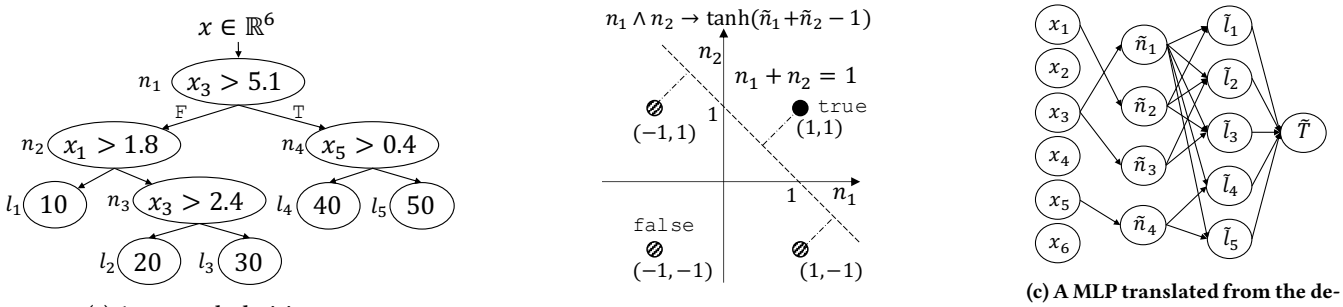
A crucial observation is that once the training is complete, the data-processing logic of any ML model can be completely defined by the prediction function regardless of the loss function and the learning algorithm. Hence, we can translate a parametric operator derived from a ML model by applying the translation method for non-parametric operators to the model's prediction function and properly initializing the parameters. For example, a linear SVM can be translated into a linear NN module of one output unit having the weights transferred from the trained SVM. It is worth noting that the translation of a ML pipeline into a pipeline of NN modules is uniquely done starting from the data-processing logic (i.e., prediction function in case of parametric operator derived from ML model), independently on how different parts of the ML pipeline have been trained. This enables us to translate different operators of a pipeline using the same formalism even though they might have been obtained via different learning algorithms or objectives.

2.2 Translating Algorithmic Operators: GBDT

While most ML models correspond to arithmetic operators that can be directly translated, some do not. One prominent example is GBDT whose prediction function is not differentiable. Instead, each prediction of a GBDT model is made by executing a sequence of `if-else` statements for each tree and computing the mean over the trees. In that respect, GBDT's prediction function is an algorithmic operator rather that an arithmetic one, which means we cannot use backpropagation. In order to jointly optimize GBDT with other operators, we should rewrite its prediction function as a differentiable function of tunable parameters. We use GBDT as a running example here because they are widely used in practical data science [44]. Naturally, the same approach applies over any tree-based model (e.g., decision trees, random forests, etc.).

We introduce parameters that fully determine GBDT's prediction function, and smooth the non-differentiable points of the function so that it can be differentiated. At a given internal node n of a binary decision tree of GBDT, the prediction function evaluates a boolean-valued function $n(x) = x_{i(n)} > \theta_n$, where x is a vector representing the input of the tree, $i(n)$ is the index of the feature examined at

[3] Some parametric operators are not derived from ML models (e.g., normalizer). Still, these operators can be translated using the same mechanism for parametric operators derived from ML models.



(a) An example decision tree.

(b) Logical cunjunction.

(c) A MLP translated from the decision tree of Figure 2a.

Figure 2: Translating a decision tree into a multi-layer perceptron.

node n, and θ_n is the decision threshold at node n. We smooth this non-differentiable function by making the function output a real number, $\tilde{n}(x) = tanh(\frac{1}{\tau}(x^T e_{i(n)} - \theta_n))$, where $e_{i(n)}$ is the canonical basis vector along the $i(n)$-th dimension of the feature space and τ is a temperature parameter. If $n(x)$ is `true`, $\tilde{n}(x)$ is close to 1, otherwise $\tilde{n}(x)$ is close to -1. We set τ as 1 throughout this paper. As we employ smaller τ, the differentiable approximation becomes steeper and degenerates into the original boolean-valued function.

Next, we note that the value of a leaf node is outputted as the final value of a tree if and only if the path from the root node to that leaf node is traversed. For example, in Figure 2a, the tree will output 30 (i.e. the value of leaf l_3) iff $n_1(x)$ is `false`, $n_2(x)$ is `true`, and $n_3(x)$ is `true`. As such, we denote the leaf activation function of l_3 as a conjunction of l_3's ancestors: $l_3(x) = \neg n_1(x) \wedge n_2(x) \wedge n_3(x)$. To get a differentiable approximation of the logical conjunction, we write $\tilde{l}_3(x) = tanh\left(\frac{1}{\tau}\left(-\tilde{n}_1(x) + \tilde{n}_2(x) + \tilde{n}_3(x) - C_{l_3} + 1\right)\right)$, where C_l is the total number of literals in the conjunction (the path length from the root to the leaf node l; e.g., $C_{l_3} = 3$). Figure 2b visualizes this approximation for 2 inputs. The equation $n_1 + n_2 = 1$ is a maximum-margin hyperplane between `true` and `false` evaluations of $n_1 \wedge n_2$. In the case of no approximation (i.e., $\tau \rightarrow 0$), one and only one of the leaf activation functions $\tilde{l}(x)$ evaluates to 1 for any given input x, while the rest are -1.

Having translated the function of internal and leaf nodes into the smooth functions described above, any decision tree $T(x)$ can be translated into a MLP $\tilde{T}(x)$ with two hidden layers. Figure 2c shows an example of this translation procedure. The first hidden layer implements a hidden unit $n(x)$ per each internal node. The second hidden layer allocates a hidden unit $l(x)$ for each leaf node. Finally, the output layer is defined as a linear layer with one unit, $\tilde{T}(x) = \sum_{l_i \in L} \frac{v_i}{2}(1 + \tilde{l}_i(x))$, where L is the set of all leaf nodes and v_i is the value of the leaf node l_i. Translation of GBDT or Random Forest follows directly by computing the average of $\tilde{T}(x)$ over the trees. We batch the computation of multiple MLPs using variants of `gemm` such as `baddbmm` and `addbmm`. Since each MLP only has tens (or one to two hundred) of hidden units, we cannot fully utilize the computation power of modern GPUs without batching them.

Note that when smoothing boolean-valued functions, we map `false` evaluation of $n(x)$ and $l(x)$ to a real number close to -1, not 0. Suppose we map `false` to a value close to 0 and use dropout [52] when training the translated module. In this case, dropping a `true` neuron by zeroing its output can be seen as flipping the evaluation from `true` to `false`, introducing unintended bias. Instead of using the logistic sigmoid function to map `false` to a number close to 0, we use *tanh* for smoothing the boolean functions to make the dropped neurons unbiased, meaning neither `true` nor `false`.

Once the translation is complete, the question is which of the parameters should be declared as trainable. We suggest four levels of parametrization to balance good fit and inductive bias:

- L1: The weights and biases for computing the output layer $\tilde{T}$, initialized using leaf node values v, are declared as trainable.
- L2: In addition to L1, the biases for computing the first hidden layer $\tilde{n}$, initialized using the decision threshold values θ's at the internal nodes, are declared as trainable.
- L3: In addition to L2, the weights for computing the first hidden layer $\tilde{n}$, initialized using the canonical basis vectors $e_{i(n)}$ in the equation of $\tilde{n}(x)$, are declared as trainable.
- L4: In addition to L3, the weights (including the non-existing zero weights) and biases for computing the second hidden layer $\tilde{l}$ are declared as trainable.

As level number increases, we declare more parameters as trainable and as such increase the capacity of the MLP to fit to data better. While L1 and L2 can only change the leaf and the decision threshold values in the tree, L3 can additionally lead to examining a linear combination of features at each internal node rather than a single feature. Up to L4, the decision structure that determines whether or not to activate a leaf node by examining internal nodes on the path from the root to the leaf is preserved; whereas, at L4, we let this decision structure change. That is, L4 gives us a fully-connected and fully-trainable MLP initialized by a decision tree. This level can be disabled if, for example due to some governance constraints, the decision structure must be maintained for explainability.

2.3 Translating Algorithmic Operators for Categorical Features

Classical ML pipelines often convert categorical features into numerical values using non-differentiable, algorithmic operators. The simplest yet most popular technique is one-hot encoding [35], which generates sparse one-hot vectors out of categorical inputs. This operator is intrinsically non-differentiable since its inputs lie on discrete spaces such as integer or string.

Our observation is that one-hot encoding consumes raw categorical inputs, which means that we do not have to backpropagate further through its discrete inputs due to the absence of upstream operators. We adopt the embedding technique that has been studied

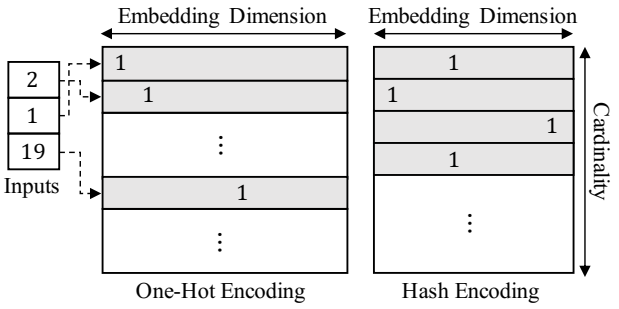


Figure 3: Translating one-hot encoder and hash encoder into embedding lookup modules.

intensively by the machine learning community. As shown in Figure 3, one-hot encoding can be seen as an embedding vector lookup operation with the embedding dimension matching the cardinality of categories. We can declare this embedding matrix as trainable in order to replace sparse one-hot vectors with dense representations and learn relationship between different categorical features, which is not possible with one-hot encoding. The same statement holds for hash encoding [34], except that data scientists can control the size of embedding dimension explicitly. Although in the original hash encoding we may have collisions between different categories, we have one row in the embedding matrix for each input category after translation. This means that even if the resulting vectors are equivalent, the "collision information" is actually stored into the embedding matrix. For example, the hash encoder in Figure 3 initially maps the first and fourth category to equivalent one-hot vectors, but they are in two different rows of the embedding matrix, and therefore they will be eventually trained differently.

By translating categorical encoders into embedding modules, we can use the well-recognized embedding technique along with arbitrary ML operators, which was not possible before. There indeed exist ML algorithms such as Matrix Factorization [30] and Factorization Machine [21, 46] that learn latent factors, which are conceptually equivalent to embedding parameters. Yet, the adoption of latent factors using these algorithms requires the ML operator to use a specific form of prediction function that explicitly models two-way interaction between features [46], which may not be desirable for certain type of tasks. In contrast, WindTunnel can combine embedding modules with arbitrary ML models, allowing data scientists to use any prediction function they want without restriction.

2.4 Fine-Tuning

After translating the operators into NN modules, one can jointly optimize the trainable parameters of the translated pipeline via backpropagation. We refer to this training process as *fine-tuning*. There are many scenarios for which this fine-tuning step can be useful. First, by fine-tuning the resulting pipeline on the original training data, we can potentially improve the generalization of the model since we are now jointly optimizing all the operators of the pipeline toward the final loss function. We empirically demonstrate this in Section 4. Second, as we discussed in Section 2.1, the translation process does not depend on the loss functions that different operators have been trained toward before. This means that once the translation is complete, the resulting pipeline can be fine-tuned toward a completely different objective that is more suitable for a given application. Third, fine-tuning can be used to adapt the model to new data that were not available before, which is not straightforward without re-training the original ML pipeline with the old and new data [24]. It is worth noting that other methods for fine-tuning such as boosting may increase the model size and complexity, while WindTunnel does not. Also, the ensemble model obtained by boosting can be seen as a pipeline containing multiple models that were not jointly optimized, so it can also benefit from our translation approach.

3 IMPLEMENTATION

Based on the translation mechanism described in Section 2, we have implemented prototypes of WindTunnel on different classical ML libraries (i.e., scikit-learn and ml.net). WindTunnel design is in fact simple, flexible, and easy to extend. The main component of WindTunnel is a *pre-defined mapping table* between the supported operators (listed in Table 1) and neural network modules implemented in PyTorch [40]. On top of the mapping table, WindTunnel provides a set of *converters* for extracting information from the trained ML operator and materialize the information into parameters of the corresponding NN module. We have different converters based on the ML framework the input pipeline was authored in. For the experiments in Section 4, we use scikit-learn [41] and PyTorch [40] for implementing ML pipelines.

During the translation process, WindTunnel refers to the proper converters and mapping table entries, and replaces the operator in the original pipeline with its differentiable counterpart. For the operators that we do not support, we either (1) cache the operators' outputs and reuse them in the fine-tuning stage; or (2) if streaming execution is provided by the ML framework, we stream data into the untouched operators and redirect their outputs to the WindTunnel pipeline. Caching is in general possible because the unsupported operators are often placed at the beginning of the pipeline thus their outputs do not change. We can consider this as a separate data pre-processing step, which is typically done before actual training.

We are currently working on adding the pre-defined mapping table containing the PyTorch implementations to Hummingbird[4].

4 EXPERIMENTS

In this section, we empirically evaluate the performance of WindTunnel. The main goal of the experiments is to show that we can improve the performance of ML pipelines by joint optimization instead of training each operator individually. We carry our experiments on binary classification tasks for three tabular datasets. We start with details about experimental setup, such as dataset description, pipeline composition, and training configurations.

4.1 Experimental Setup

Datasets. We conduct experiments on real-world datasets listed in Table 2. The Flight [39] dataset is used for predicting whether a scheduled flight will be delayed more than 15 minutes inclusive. We use records from the year of 2006 and 2007 as training set (about

[4]https://github.com/microsoft/hummingbird/tree/mainterl/fine-tune-trees.

Table 2: Statistics of datasets used in experiments. #Rec is the number of data records, #Num is the number of numerical features, #Cat is the number of categorical features, #Unq is the number of unique categories that appear in the training split (i.e. the sum of cardinalities of categorical features), and Positive ratio is the percentage of records with positive label.

Dataset	#Rec	#Num	#Cat	#Unq	Positive ratio
Flight	21.6M	2	6	694	20.4%
Avazu	40.4M	0	23	8.93M	17.0%
Criteo	45.8M	13	26	30.8M	25.6%

14M records), while the records from the year of 2008 are divided into two splits and used as validation set (Jan to Jun) and test set (Jul to Dec). The Avazu [4] and Criteo [9] datasets are from Kaggle competitions that call for click-through rate prediction models. We use the first 90% of the Criteo dataset as training set, while the next 5% and the last 5% is used as validation and test set, respectively. For the Avazu dataset, we use the same ratio for splitting the dataset after a random shuffling step, to ensure that the distribution of "day of week" feature is consistent between train-validation-test splits.

Data pre-processing. We experiment with three different data pre-processing schemes to show both the general applicability of WindTunnel, as well as the impact of different pre-processing operations over the embedding dimensions and final accuracy. The first pre-processing scheme (Pre1) drops categories that appear less than 25 times in the training set to reduce noise, followed by a binary encoder for handling categorical features. In the second pre-processing scheme (Pre2), we replace the binary encoder with a two-hot encoder, while the rest are left the same as the first scheme (Pre1). Two-hot encoder is similar to the well-known one-hot encoder [35], except that there are two "hot" elements (value of 1) in the resulting vector. Switching the pre-processing step from Pre1 to Pre2 allows us to test how accuracy changes when increasing the embedding dimensions. The last, most complex scheme (Pre3) is composed as follows: (1) drop the lower 1% categories by frequency; (2) drop categories that appear less than 10 times in the training set; (3) add additional categorical features by bucketizing numeric features using 32 bins; (4) apply two-hot encoding and target encoding [43]. We take inspiration from previous literature [24] for designing Pre3.

ML pipelines. We evaluate the performance improvements using three ML pipelines. Each pipeline not only serves as the source pipeline for WindTunnel translation, but also serves as a baseline for comparison. In the first pipeline (Pipe1), we use a logistic regression model following the pre-processing scheme. After translation, WindTunnel jointly optimizes the embedding modules translated from categorical encoders and the logistic regression module. The second pipeline (Pipe2) employs a GBDT model trained by LightGBM [23] after the pre-processing scheme. The third pipeline (Pipe3) is composed as follows: (1) process the data using the described pre-processing scheme; (2) train a LightGBM model with the processed data and label; (3) for each tree in the trained LightGBM model, create a one-hot vector that marks the index of the activated leaf as 1 and keeps others 0 using the leaf activation function $l(x)$ (see Section 2.2); (4) train a Factorization Machine [46] model with the output from (3) and label. We take inspiration from the winning solution of Kaggle competition [20] for designing Pipe3. These pipelines cover the most common ML algorithms (linear models and decision-tree variants) used in practice [22].

DNN baseline. In addition to the three ML pipelines described above, we compare DeepGBM [24], a state-of-the-art neural network that takes advantage of classical ML by distilling knowledge from gradient boosting machine. We do not compare with other DNN-based models because Ke et al. [24] already demonstrated that DeepGBM is consistently better than Wide&Deep [8], DeepFM [13] and PNN [45]. We also do not report results from ML pipelines using a single operator because they fall far behind other models.

Configurations. We set the LightGBM to create 64 leaves for each tree, and we construct 100 trees for all experiments that use LightGBM. The Factorization Machine (FM) model uses a latent dimension of 20 for all experiments. We set learning rate to 0.25 and 10^{-3} for training LightGBM and FM, respectively. Regarding the training of WindTunnel pipelines, we use the parametrization level L4 for GBDT-translated modules unless otherwise noted. Dropout [52] is applied to each NN layer of WindTunnel pipeline, with a zeroing probability of 0.1. We use the Adam [27] optimizer with a batch size of 4096 and weight decay of 10^{-6} for all experiments. Learning rate is set to 10^{-4} for the Flight and Avazu dataset, and 10^{-5} for the Criteo dataset. We select these rates by sweeping a grid of $\{10^{-2}, 10^{-3}, \cdots, 10^{-6}\}$ for learning rate and $\{10^{-5}, 10^{-6}, 10^{-7}\}$ for weight decay. We let the training process run until convergence. Regarding the experiments using DeepGBM, we use an open-source implementation. For the Flight and Criteo dataset, we use the hyperparameter setting described in the original literature. Since the literature did not use the Avazu dataset, we set the hyperparameters same as Criteo's.

4.2 Overall Performance

We first evaluate the overall performance of WindTunnel. The comparison results can be found in Table 3. As we can see, WindTunnel greatly improves AUC of the original pipeline by jointly optimizing the ML operators which were trained separately, providing up to 10.0% higher AUC. This demonstrates the power of end-to-end training and WindTunnel's ability to leverage such advantage. WindTunnel also surpasses DeepGBM by a significant margin (up to 3.4%) for all cases, except the Avazu dataset using Pre3 pre-processing where the margin is small (0.7760 vs. 0.7763). We credit the AUC gap to joint training of embedding modules (translated from categorical encoders) and downstream modules (translated from GBDT and FM), which is not possible in DeepGBM.

Impact of data pre-processing. We study the impact of using different pre-processing schemes, and find that it can largely affect the AUC of both the original pipeline and WindTunnel pipeline. First of all, the use of one-hot encoder [35] is not suitable for large-scale dataset because it requires too much host and GPU memory and computation power. The number of embedding dimension grows linearly with the number of unique categories (denoted as C), which makes training of both the original pipeline and WindTunnel pipeline extremely difficult. On the other hand, for the binary encoder [33] and two-hot encoder, the minimum required size of embedding dimension is roughly $log_2 C$ and $\sqrt{C}$, respectively.

Table 3: Overall performance comparision. We report AUC on test split following the previous work [24]. ML is the original ML pipeline, while W.T. is for WindTunnel. PreX means different preprocessing schemes, and PipeX denotes different ML pipelines. The best result is marked bold.

Model	Flight			Avazu			Criteo		
	Pre1	Pre2	Pre3	Pre1	Pre2	Pre3	Pre1	Pre2	Pre3
ML (Pipe1)	0.6783	0.6847	0.7126	0.6896	0.7264	0.7553	0.7167	0.7442	0.7769
ML (Pipe2)	0.7358	0.7427	0.7507	0.7521	0.7550	0.7718	0.7739	0.7781	0.7925
ML (Pipe3)	0.7519	0.7547	0.7467	0.7597	0.7616	0.7728	0.7838	0.7884	0.7954
DeepGBM	0.7793	0.7695	0.7726	0.7682	0.7680	0.7760	0.7965	0.7918	0.7972
W.T. (Pipe1)	0.6913	0.6910	0.7244	0.7588	0.7589	0.7637	0.7750	0.7804	0.7903
W.T. (Pipe2)	0.7790	0.7897	0.7960	**0.7753**	**0.7742**	**0.7763**	0.8006	0.8053	0.8041
W.T. (Pipe3)	**0.7829**	**0.7906**	**0.7989**	0.7746	0.7718	0.7753	**0.8014**	**0.8058**	**0.8048**

Table 4: AUC of the Criteo dataset using different translation scopes.

Model	Pre2	Pre3
ML (Pipe2)	0.7781	0.7925
GBDT2NN (Pipe2)	0.7962	0.7998
W.T. (Pipe2)	0.8053	0.8041

Table 5: AUC of the Criteo dataset using different parameter initialization regimes.

Model	Pre2		Pre3	
	Cold	Warm	Cold	Warm
W.T. (Pipe2)	0.7983	0.8053	0.7990	0.8041
W.T. (Pipe3)	0.7966	0.8058	0.7975	0.8048

Table 6: AUC of the Criteo dataset using different GBDT parametrization levels and dataset sizes (1%, 10%, 100%).

Model	100%	10%	1%
ML (Pipe2)	0.7781	0.7777	0.7657
W.T. (Pipe2, L2)	0.7924	0.7874	0.7685
W.T. (Pipe2, L3)	0.8051	0.7912	0.7699
W.T. (Pipe2, L4)	0.8053	0.7906	0.7691

Second, Pre2 shows better AUC on all cases using ML pipelines compared to Pre1. Similarly, WindTunnel pipelines tends to work better with Pre2 than Pre1. This means that both ML and WindTunnel pipelines prefer two-hot encoding to binary encoding. We attribute this trend to the high separability of two-hot vectors compared to binary vectors at the cost of larger embedding dimension. Note that DeepGBM prefers Pre1 to Pre2 because it explicitly selects top-N elements of the input vector based on the amount of information computed by LightGBM and use only them for training the GBDT-distilled neural network. Due to this feature selection policy, when we use two-hot encoder that produces larger encoded vector, some elements of the vector with meaningful information are dropped, thus resulting in worse AUC compared to binary encoder.

If we go one step further and compare Pre2 and Pre3, Pre3 shows better AUC in most cases. This means that if we carefully design the pre-processing scheme and adopt more sophisticated feature engineering, we can achieve better results compared to using simple, less-optimized pre-processing scheme.

Discussion. From this experiment, we notice the importance of getting a proper pre-processing scheme. We deem developing neural translation of the pre-processing operators a promising direction able to improve the performance. This also aligns with recent trends in computer vision domain that learns to augment input images by parametrizing and tuning the pre-processing scheme [10, 25, 32]. Developing new types of pre-processing operators that are naturally tunable (e.g., embedding) could also be an alternative solution.

4.3 Ablation Study

Next, we evaluate the performance of WindTunnel with varying configurations. As a representative for the datasets we study, we select the largest one (i.e, Criteo) and conduct experiments on it. We also focus on the settings with Pre2 & Pre3 schemes and Pipe2 & Pipe3 pipelines, because we produce top results with these settings.

Joint optimization. We study the impact of joint optimization in more details. Table 4 reports additional results by translating only the GBDT model of Pipe2, not categorical encoders (denoted by "GBDT2NN"). Note that a GBDT model is already an ensemble of multiple trees, so translating only the GBDT model (GBDT2NN) also employs joint optimization of multiple MLP modules. From the results, we can observe a clear pattern that as the scope of translation gets wider, the AUC increases (ML < GBDT2NN < WindTunnel). This further supports the claim that joint optimization is a promising technique able to improve the performance. In particular, the gap between GBDT2NN and WindTunnel suggests that the neural translation of pre-processing operators (categorical encoder) is indeed effective in improving the accuracy.

Parameter initialization and architecture. In this set of experiments, we show that the translation of trained ML operators provides informative initialization of WindTunnel pipelines. We experiment with two regimes of initialization for the parameters of WindTunnel pipeline: (1) in the *cold start* regime the parameters are randomly initialized (denoted by "Cold"); and (2) in the *warm start* regime the parameters are carried over from the original ML pipeline (denoted by "Warm", used as default setting for other WindTunnel experiments).

Table 5 shows that the warm start outperforms the cold start, which means that the parameters extracted from the original ML pipeline provide an informative initialization for WindTunnel. Interestingly, the cold start regime performs better than DeepGBM. This shows that WindTunnel not only delivers meaningful information by parameter initialization, but also provides a good neural architecture that can achieve better results than the baseline.

GBDT parametrization level and dataset size. As described in Section 2.2, the proposed translation of GBDT increases the capacity of the model if we adopt higher parametrization levels (L3 or L4). We evaluate the effect of the parametrization level in Table 6, using the combination of Pre2 and Pipe2. From the results, we can observe the significant gap between L2 and L3, verifying that the extra capacity helps improving the performance. Yet, L2 still outperforms the original ML pipeline, due to WindTunnel's ability to jointly optimize the categorical encoders and GBDT. We also evaluate trade-offs between flexibility and inductive bias come from different levels. For this, we use subsets of the training split with various sampling ratios (1%, 10%, 100%). As the subset size decreases, the gap between L2 and the other two level closes. This trend shows overfitting of L3 and L4 in small data experiments and how it is avoided by L2 that has much smaller capacity. In other words, lower levels provide a natural regularization mechanism in small data experiments. The results also show that the gap between the original pipeline (ML) and WindTunnel gets wider as the subset size increases. This suggests that WindTunnel has better scalability (in terms of accuracy) than the original pipeline by exploiting the extra model capacity that comes from the joint optimization.

5 RELATED WORKS

End-to-end training of ML pipelines. Milutinovic et al. [36] proposes the end-to-end training of ML pipelines via propagating gradients across multiple differentiable operators. This work however has no discussion about non-differentiable operators, while we attempt to backpropagate through non-differentiable (e.g., GBDT) and non-trainable operators (e.g., categorical encoding). Additionally, this work requires users to manually write "backward" code for operators from non-NN libraries (e.g., scikit-learn), while WindTunnel exploits the automatic differentiation capabilities of DL libraries by neural translation.

Tree-based models and neural networks. There have been early works [5, 18, 50] that initialize parameters of a MLP by using a trained decision tree. However, these works have several limitations: (1) the resulting MLP cannot back-propagate gradients to upstream (or downstream) operators because the input and output layers are not designed with end-to-end training in mind; and (2) the MLP is not well-suited for adopting dropout [52] due to the use of logistic sigmoid activation that introduces bias. They also did not demonstrate generalizability on tree ensemble models like GBDT or Random Forest, and only experimented with a single decision tree that is unlikely used in practice.

DJINN [17] initializes a MLP in a different way, where the depth of the decision tree is used to decide the number of layers. Weights are randomly initialized, while the information on the tree is retained only for sparsely connecting the neurons. We instead extract more information from a GBDT model including tree structure and decision thresholds to initialize the parameters. DNDT [61] suggests to build tree-like neural networks for interpretability. This is different from our approach of handling trees because: (1) it builds a tree-like neural network using random weight initialization, while we retain the behavior of trained trees by neural translation; and (2) our translated pipeline learns to use all features for making decision at each internal node (L3 and L4), while this work uses a single feature at each neuron and requires a wider network whose number of neurons grows exponentially as the number of features grows. dNDF [29] combines neural networks (CNNs) and decision tree classifiers by enabling backpropagation, with a focus on computer vision tasks. Similar to DNDT, dNDF also starts with a random initialization of tree parameters.

Finally, DeepGBM [24] *distills* trees into neural networks by transferring the knowledge of tree outputs and feature importance learned by GBDT. Given this distilled neural network, DeepGBM incorporates an additional embedding-based neural network called *CatNN* for handling categorical features only. For this, DeepGBM requires several hyperparameters such as the number of layers and hidden units (for both the distilled NN and CatNN), weights for controlling the strength between knowledge distillation and final loss, the number of features used for training the distilled NN, how to group trees for distillation, and so on. Instead, WindTunnel directly translates a ML pipeline so the structure of resulting pipeline solely depends on the structure of the original one.

Making a specific model differentiable. Stoyanov et al. [54] deals with algorithmic operator beyond trees. Specifically, they directly minimizes the empirical risk of a Markov Random Field (MRF) by backpropagating through the inference algorithm. While this work deals with a system composed of a single model (MRF), our work handles pipelines with multiple operators. We can adopt their approach when we encounter ML pipelines containing MRF.

Neural translation for fast inference. Finally, while in this work we focus on translating ML operators into differentiable modules for fine-tuning, in the Hummingbird project [37] we translate ML operators (including unsupported ones in WindTunnel) into tensor operations without requiring the operations to be differentiable. This allow us to run inference of end-to-end pipelines (1) completely on DL frameworks without any additional data conversion overhead; and (2) on hardware accelerators specialized for tensor operations. We are currently working on adding training (fine-tuning) support in Hummingbird through WindTunnel.

6 CONCLUSIONS

Inspired by the existing gap between classical ML pipelines and neural networks, we propose WindTunnel, a framework for translating pipelines of ML operators into neural networks and further *jointly* fine-tuning them. As part of the translation procedure, we also propose techniques for translating popular non-differentiable operators including GBDT and categorical encoders. The experimental results show that the translation with knowledge transfer followed by the fine-tuning leads to significant accuracy improvements over the original pipeline and state-of-the-art NNs. Furthermore, we see that our translation mechanism can be seen as an approach for designing neural network architectures for a given task that is inspired by the classical ML pipeline structure for that task. We deem this work as a step towards filling the gap between classical ML pipelines and neural networks over tabular data.

ACKNOWLEDGMENTS

This work was supported (in part) by IITP grant funded by the Korea government (MSIT) (No.2015-0-00221, No.2021-0-01343), and by ICT R&D program of MSIT/IITP (No.2017-0-01772).

REFERENCES

[1] Eugene Agichtein, Eric Brill, and Susan Dumais. 2006. Improving Web Search Ranking by Incorporating User Behavior Information. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 19–26.

[2] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupré, Vadim Eksarevskiy, Senja Filipi, Tom Finley, Abhishek Goswami, Monte Hoover, Scott Inglis, Matteo Interlandi, Najeeb Kazmi, Gleb Krivosheev, Pete Luferenko, Ivan Matantsev, Sergiy Matusevych, Shahab Moradi, Gani Nazirov, Justin Ormont, Gal Oshri, Artidoro Pagnoni, Jignesh Parmar, Prabhat Roy, Mohammad Zeeshan Siddiqui, Markus Weimer, Shauheen Zahirazami, and Yiwen Zhu. 2019. Machine Learning at Microsoft with ML. NET. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2448–2458.

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. 2016. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *Proceedings of the 33rd International Conference on Machine Learning*. 173–182.

[4] Avazu. 2021. *Avazu Click-Through Rate Prediction Dataset.* Retrieved Sep 23, 2021 from https://www.kaggle.com/c/avazu-ctr-prediction/data/

[5] Arunava Banerjee. 1997. Initializing Neural Networks using Decision Trees. *Computational Learning Theory and Natural Learning Systems* 4 (1997), 3–15.

[6] Christopher J.C. Burges. 2010. *From RankNet to LambdaRank to LambdaMART: An Overview.* Technical Report MSR-TR-2010-82. Retrieved Sep 23, 2021 from https://www.microsoft.com/en-us/research/publication/from-ranknet-to-lambdarank-to-lambdamart-an-overview/

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. 578–594.

[8] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. 7–10.

[9] CriteoLabs. 2021. *Kaggle Display Advertising Challenge Dataset.* Retrieved Sep 23, 2021 from https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/

[10] Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. 2019. AutoAugment: Learning Augmentation Strategies from Data. In *Proceedings of the 32nd IEEE Conference on Computer Vision and Pattern Recognition*. 113–123.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 17th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186.

[12] Thore Graepel, Joaquin Quiñonero Candela, Thomas Borchert, and Ralf Herbrich. 2010. Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. In *Proceedings of the 27th International Conference on Machine Learning*. 13–20.

[13] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 1725–1731.

[14] Hany Hassan, Anthony Aue, Chang Chen, Vishal Chowdhary, Jonathan Clark, Christian Federmann, Xuedong Huang, Marcin Junczys-Dowmunt, William Lewis, Mu Li, Shujie Liu, Tie-Yan Liu, Renqian Luo, Arul Menezes, Tao Qin, Frank Seide, Xu Tan, Fei Tian, Lijun Wu, Shuangzhi Wu, Yingce Xia, Dongdong Zhang, Zhirui Zhang, and Ming Zhou. 2018. Achieving Human Parity on Automatic Chinese to English News Translation. arXiv:1803.05567 [cs.CL]

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[16] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal processing magazine* 29, 6 (2012), 82–97.

[17] K. D. Humbird, J. L. Peterson, and R. G. McClarren. 2018. Deep Neural Network Initialization With Decision Trees. *IEEE Transactions on Neural Networks and Learning Systems* 30, 5 (2018), 1286–1295.

[18] Irena Ivanova and Miroslav Kubat. 1995. Initialization of Neural Networks by Means of Decision Trees. *Knowledge-Based Systems* 8, 6 (1995), 333–344.

[19] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting Concept Drift in Malware Classification Models. In *Proceedings of the 26th USENIX Security Symposium*. 625–642.

[20] Yuchin Juan, Wei-Sheng Chin, and Yong Zhuang. 2014. *3 Idiots' Approach for Display Advertising Challenge.* Retrieved Sep 23, 2021 from https://github.com/ycjuan/kaggle-2014-criteo

[21] Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. 2016. Field-aware Factorization Machines for CTR Prediction. In *Proceedings of the 10th ACM Conference on Recommender Systems*. 43–50.

[22] Kaggle. 2020. *State of Data Science and Machine Learning 2020.* Retrieved Sep 23, 2021 from https://www.kaggle.com/kaggle-survey-2020

[23] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 3149–3157.

[24] Guolin Ke, Zhenhui Xu, Jia Zhang, Jiang Bian, and Tie-Yan Liu. 2019. DeepGBM: A Deep Learning Framework Distilled by GBDT for Online Prediction Tasks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 384–394.

[25] Jang-Hyun Kim, Wonho Choo, and Hyun Oh Song. 2020. Puzzle Mix: Exploiting Saliency and Local Statistics for Optimal Mixup. In *Proceedings of the 37th International Conference on Machine Learning*. 5275–5285.

[26] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks. In *Proceedings of the 14th EuroSys Conference*. 1–15.

[27] Diederik P. Kingma and Jimmy Lei Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference for Learning Representations*.

[28] J. Zico Kolter and Marcus A. Maloof. 2006. Learning to Detect and Classify Malicious Executables in the Wild. *Journal of Machine Learning Research* 7 (2006), 2721–2744.

[29] Peter Kontschieder, Madalina Fiterau, Antonio Criminisi, and Samuel Rota Bulò. 2016. Deep Neural Decision Forests. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*. 4190–4194.

[30] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (2009), 30–37.

[31] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: An abstraction for general data processing. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1797–1804.

[32] Sungbin Lim, Ildoo Kim, Taesup Kim, Chiheon Kim, and Sungwoong Kim. 2019. Fast AutoAugment. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 6665–6675.

[33] Will McGinnis. 2016. *Binary Encoder for Categorical Variables.* Retrieved Sep 23, 2021 from https://contrib.scikit-learn.org/category_encoders/binary

[34] Will McGinnis. 2016. *Hashing Encoder for Categorical Variables.* Retrieved Sep 23, 2021 from https://contrib.scikit-learn.org/category_encoders/hashing

[35] Will McGinnis. 2016. *One-hot Encoder for Categorical Variables.* Retrieved Sep 23, 2021 from https://contrib.scikit-learn.org/category_encoders/onehot

[36] Mitar Milutinovic, Atılım Güneş Baydin, Robert Zinkov, William Harvey, Dawn Song, Frank Wood, and Wade Shen. 2017. End-to-end Training of Differentiable Pipelines Across Machine Learning Frameworks. *NIPS AutoDiff workshop* (2017).

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

[38] Lakshmanan Nataraj, Sreejith Karthikeyan, Gregoire Jacob, and Bangalore S. Manjunath. 2011. Malware Images: Visualization and Automatic Classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*. 1–7.

[39] ASA Section on Statistical Computing. 2021. *Data Expo 2009 - Airline on-time performance.* Retrieved Sep 23, 2021 from https://community.amstat.org/jointscsg-section/dataexpo/dataexpo2009/

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 8026–8037.

[41] 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. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, 85 (2011), 2825–2830.

[42] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In *Proceedings of the 16th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. 2227–2237.

[43] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: unbiased boosting with categorical features. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 6639–6649.

[44] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Matteo Interlandi, Avrilia Floratou, Konstantinos Karanasos, Wentao Wu, Ce Zhang, Subru Krishnan, Carlo Curino, and Markus Weimer. 2019. Data Science Through the Looking Glass and What We Found There. arXiv:1912.09536 [cs.LG]

[45] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. 2016. Product-based Neural Networks for User Response Prediction. In *Proceedings of the 16th IEEE International Conference on Data Mining*. 1149–1154.

[46] Steffen Rendle. 2010. Factorization machines. In *Proceedings of the 10th IEEE International Conference on Data Mining*. 995–1000.

[47] Steffen Rendle. 2013. Scaling Factorization Machines to Relational Data. *Proceedings of the VLDB Endowment* 6, 5 (2013), 337–348.

[48] Steffen Rendle, Walid Krichene, Li Zhang, and John Anderson. 2020. Neural Collaborative Filtering vs. Matrix Factorization Revisited. In *Proceedings of the 14th ACM Conference on Recommender Systems*. 240–248.

[49] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. Imagenet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[50] Ishwar Krishnan Sethi. 1990. Entropy Nets: From Decision Trees to Neural Networks. *Proc. IEEE* 78, 10 (1990), 1605–1613.

[51] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. 2017. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *Proceedings of the 33rd IEEE International Conference on Data Engineering*. 535–546.

[52] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[53] Dusan Stosic. 2020. *Training Neural Networks with Tensor Core.* Retrieved Sep 23, 2021 from https://nvlabs.github.io/eccv2020-mixed-precision-tutorial/files/dusan_stosic-training-neural-networks-with-tensor-cores.pdf

[54] Veselin Stoyanov, Alexander Ropson, and Jason Eisner. 2011. Empirical Risk Minimization of Graphical Model Parameters Given Approximate Inference, Decoding, and Model Structure. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*. 725–733.

[55] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning*. 6105–6114.

[56] H2O team. 2021. *H2O.* Retrieved Sep 23, 2021 from https://github.com/h2oai/h2o-3

[57] Hummingbird team. 2021. *Hummingbird.* Retrieved Sep 23, 2021 from https://github.com/microsoft/hummingbird

[58] ONNX team. 2021. *ONNXMLTools.* Retrieved Sep 23, 2021 from https://github.com/onnx/onnxmltools

[59] PyTorch team. 2021. *PyTorch Ecosystem.* Retrieved Sep 23, 2021 from https://pytorch.org/ecosystem/

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 5998–6008.

[61] Yongxin Yang, Irene Garcia Morillo, and Timothy M. Hospedales. 2018. Deep Neural Decision Trees. arXiv:1806.06988 [cs.LG]