TF-Ranking: Scalable TensorFlow Library for Learning-to-Rank

Rama Kumar Pasumarthi, Sebastian Bruch, Xuanhui Wang, Cheng Li, Michael Bendersky, Marc Najork, Jan Pfeifer, Nadav Golbandi, Rohan Anil, Stephan Wolf
Google
{ramakumar,bruch,xuanhui,chgli,bemike,najork,janpf,nadavg,rohananil,stephanw}@google.com

ABSTRACT

Learning-to-Rank deals with maximizing the utility of a list of examples presented to the user, with items of higher relevance being prioritized. It has several practical applications such as large-scale search, recommender systems, document summarization and question answering. While there is widespread support for classification and regression based learning, support for learning-to-rank in deep learning has been limited. We introduce TensorFlow Ranking, the first open source library for solving large-scale ranking problems in a deep learning framework[1]. It is highly configurable and provides easy-to-use APIs to support different scoring mechanisms, loss functions and evaluation metrics in the learning-to-rank setting. Our library is developed on top of TensorFlow and can thus fully leverage the advantages of this platform. TensorFlow Ranking has been deployed in production systems within Google; it is highly scalable, both in training and in inference, and can be used to learn ranking models over massive amounts of user activity data, which can include heterogeneous dense and sparse features. We empirically demonstrate the effectiveness of our library in learning ranking functions for large-scale search and recommendation applications in Gmail and Google Drive. We also show that ranking models built using our model scale well for distributed training, without significant impact on metrics. The proposed library is available to the open source community, with the hope that it facilitates further academic research and industrial applications in the field of learning-to-rank.

CCS CONCEPTS

• **Information systems → Learning to rank**.

KEYWORDS

Learning-to-Rank; Information Retrieval; Recommender Systems; Machine Learning

ACM Reference Format:
Rama Kumar Pasumarthi, Sebastian Bruch, Xuanhui Wang, Cheng Li, Michael Bendersky,, Marc Najork, Jan Pfeifer, Nadav Golbandi, Rohan Anil, Stephan Wolf. 2019. TF-Ranking: Scalable TensorFlow Library for Learning-to-Rank. In *The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '19), August 4–8, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3292500.3330677

[1]The open source library is available at: https://github.com/tensorflow/ranking.

1 INTRODUCTION

With the high potential of deep learning for real-world data-intensive applications, a number of open source packages have emerged in recent years and are under active development, including TensorFlow [1], PyTorch [34], Caffe [21], and MXNet [10]. Supervised learning is one of the main use cases of deep learning packages. However, compared with the comprehensive support for classification or regression in open-source deep learning packages, there is a paucity of support for ranking problems.

A ranking problem is defined as a derivation of ordering over a list of items that maximizes the utility of the entire list. It is widely applicable in several domains, such as Information Retrieval and Natural Language Processing. Some important practical applications include web search, recommender systems, machine translation, document summarization, and question answering [28].

In general, a ranking problem is different from classification or regression tasks. While the goal of classification or regression is to predict a label or a value for each individual item as accurately as possible, the goal of ranking is to optimally sort the entire item list such that, for some notion of relevance, the items of highest relevance are presented first. To be precise, in a ranking problem, we are more concerned with the relative order of the relevance of items than their absolute magnitudes.

That ranking is a fundamentally different problem entails that classification or regression metrics and methodologies do not transfer effectively to the ranking domain. To fill this void, a number of metrics and a class of methodologies that are inspired by the challenges in ranking have been proposed in the literature. For example, widely-utilized metrics such as Normalized Discounted Cumulative Gain (NDCG) [20], Expected Reciprocal Rank (ERR) [8], Mean Reciprocal Rank (MRR) [14], Mean Average Precision (MAP), and Average Relevance Position (ARP) [48] are designed to emphasize the items that are ranked higher in the list.

Similarly, a class of supervised machine learning techniques that attempt to solve ranking problems—referred to as learning-to-rank [28]—has emerged in recent years. Broadly, the goal of learning-to-rank is to learn from labeled data a parameterized function that maps feature vectors to real-valued scores. During inference, this scoring function is used to sort and rank items.

Most learning-to-rank methods differ primarily in how they define surrogate loss functions over ranked lists of items during training to optimize a non-differentiable ranking metric, and by that measure fall into one of *pointwise*, *pairwise*, or *listwise* classes of algorithms. Pointwise methods [13, 17, 18] approximate ranking to a classification or regression problem and as such attempt to optimize the discrepancy between individual ground-truth labels and the absolute magnitude of relevance scores produced by the learning-to-rank model. On the other hand, *pairwise* [5, 22] or *listwise* [7, 42, 43] methods either model the pairwise preferences or define a loss over entire ranked list. Therefore, pairwise and listwise methods are more closely aligned with the ranking task [28].

There are other factors that distinguish ranking from other machine learning paradigms. As an example, one of the challenges facing learning-to-rank is the inherent biases [24, 46] that exist in labeled data collected through implicit feedback (e.g., click logs). Recent work on unbiased learning-to-rank [2, 25, 41] explores ways to counter position bias [24] in training data and produce a consistent and unbiased ranking function. These techniques work well with pairwise or listwise losses, but not with pointwise losses [41].

From the discussion above, it is clear that a library that supports the learning-to-rank problem has its own unique set of requirements and must offer a functionality that is specific to ranking. Indeed, a number of open source packages such as RankLib[2] and LightGBM [26] exist to address the ranking challenge.

Existing learning-to-rank libraries, however, have a number of important drawbacks. First, they were developed for small data sets (thousands of queries) and do not scale well to massive click logs (hundreds of millions of queries) that are common in industrial applications. Second, they have very limited support for sparse features and can only handle categorical features with a small vocabulary. Crucially, extensive feature engineering is required to handle textual features. In contrast, deep learning packages like TensorFlow can effectively handle sparse features through embeddings [30]. Finally, existing learning-to-rank libraries do not support the recent advances in unbiased learning-to-rank.

To address this gap, we present our experiences in building a scalable, comprehensive, and configurable industry-grade learning-to-rank library in TensorFlow. Our main contributions are:

- We propose an open-source library for training large scale learning-to-rank models using deep learning in TensorFlow.
- The library is flexible and highly configurable: it provides an easy-to-use API to support different scoring mechanisms, loss functions, example weights, and evaluation metrics.
- The library provides support for unbiased learning-to-rank by incorporating inverse propensity weights in losses and metrics.
- We demonstrate the effectiveness of our library by experiments on two large-scale search and recommendation applications, especially when employing listwise losses and sparse textual features.
- We demonstrate the robustness of our library in a large-scale distributed training setting.

Our current implementation of the TensorFlow Ranking library is by no means exhaustive. We envision that this library will provide a convenient open platform for hosting and advancing state-of-the-art ranking models based on deep learning techniques, and thus facilitate both academic research as well as industrial applications.

The remainder of this paper is organized as follows. Section 2 formulates the problem of learning to rank and provides an overview of existing approaches. In Section 3, we present an overview of the proposed learning-to-rank platform, and in Section 4 we present the implementation details of our proposed library. Section 5 showcases uses of the library in a production environment and demonstrates experimental results. Finally, Section 6 concludes this paper.

[2]Available at: https://sourceforge.net/p/lemur/wiki/RankLib/

2 LEARNING-TO-RANK

In this section, we provide a high-level overview of learning-to-rank techniques. We begin by presenting a formal definition of learning-to-rank and setting up notation.

2.1 Setup

Let $\mathcal{X}$ denote the universe of items and let $\boldsymbol{x} \in \mathcal{X}^n$ represent a list of n items and $x_i \in \boldsymbol{x}$ an item in that list. Further denote the universe of all permutations of size n by Π^n, where $\pi \in \Pi^n$ is a bijection from $[1 : n]$ to itself. π may be understood as a total ranking of items in a list where $\pi(i)$ yields the rank according to π of the i^{th} item in the list and $\pi^{-1}(r)$ yields the index of the item at rank r, and we have that $\pi^{-1}(\pi(i)) = i$. A ranking function $f : \mathcal{X}^n \to \Pi^n$ is a function that, given a list of n items, produces a permutation or a ranking of that list.

The goal of learning-to-rank, in broad terms, is to learn a ranking function f from training data such that items as ordered by f yield maximal utility. Let us parse this statement and discuss each component in more depth.

2.2 Training Data

We begin with a description of training data. Learning-to-rank, which is an instance of supervised learning, assumes the existence of a ground-truth permutation π^* for a given list of items $\boldsymbol{x}$.

In most real-world settings, ground-truth is provided in its more general form: a *set* of permutations or in other words a *partial* ranking. In a partial ranking $X_1 \succ X_2 \succ \ldots \succ X_k$, where X_is are k disjoint subsets of elements of $\boldsymbol{x} \in \mathcal{X}^n$ $(k \leq n)$, items in X_i are preferred over those in $X_{j>i}$, but within each X_i items may be permuted freely. When $k = n$ partial ranking reduces to a total ranking.

As a concrete example, consider the task of *ad hoc* retrieval where given a (textual) query the ranking algorithm retrieves a relevant list of documents from a large corpus. When constructing a training dataset, one may recruit human experts to examine an often very small subset of candidate documents for a given query and grade the documents' relevance with respect to that query on some scale (e.g., 0 for "not examined" or "not relevant" to 5 for "highly relevant"). A relevance grade for document x_i is considered its "label" y_i. Similarly, in training datasets that are constructed by implicit user feedback such as click logs, documents are either relevant and clicked $(y = 1)$ or not $(y = 0)$. In either case, a list of labels $\boldsymbol{y}$ induces a partial ranking of documents.

In order to simplify notation throughout the remainder of this paper and without loss of generality, we assume the existence of π^*, a correct total ranking. π^* can be understood as an ranking induced by a list of labels $\boldsymbol{y} \in \mathbb{R}^n$ for $\boldsymbol{x} \in \mathcal{X}^n$. As such, our training data set of m items can be defined as $S^m = \{(\boldsymbol{x}, \pi^*) \,|\, \boldsymbol{x} \in \mathcal{X}^n, \pi^* \in \Pi^n\}$ or equivalently $S^m = \{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{X}^n \times \mathbb{R}^n\}$.

Returning to the case of *ad hoc* retrieval, it is worth noting that each item $x_i \in \boldsymbol{x}$ is in fact a pair of query and document (q, d_i): It is generally the case that the pair (q, d_i) is transformed to a feature vector x_i.

2.3 Scoring Function

Directly finding a permutation π is difficult, as the space of all possible permutations is exponentially large. In practice, a score-and-sort approach is used instead. Let $h : \mathcal{X}^n \to \mathbb{R}^n$ be a scoring function that maps a list of items $\boldsymbol{x}$ to a list of scores $\hat{\boldsymbol{y}}$. Let $h(.)|_k$ denotes the k^{th} dimension of $h(.)$. As discussed earlier, h induces a permutation π such that $h(\boldsymbol{x})|_{\pi^{-1}(r)}$ is monotonically decreasing for increasing ranks r.

In its simplest form, the scoring function is univariate and can be decomposed into a per-item scoring function as shown in Equation 1, where $g : x \to \mathbb{R}$ maps a feature vector to a real-valued score.

$$h(\boldsymbol{x}) = [g(x_i), \forall\, 1 \leq i \leq n] = [g(x_1), g(x_2), ..., g(x_n)]. \tag{1}$$

The scoring function h is typically parameterized by a set of parameters θ and can be written as $h(.;\theta)$. Many parameterization options have been studied in the learning-to-rank literature including linear functions [23], boosted weak learners [45], gradient-boosted trees [6, 16], support vector machines [23, 25], and neural networks [5]. Our library offers deep neural networks as the basis to construct a scoring function. This framework facilitates more sophisticated scoring functions such as multivariate functions [3], where the scores of a group of items are computed jointly. Through a flexible API, the library also enables development and integration of arbitrary scoring functions into a ranking model.

2.4 Utility and Ranking Metrics

We now turn to the notion of utility. As noted in Section 1, the utility of an ordered list of items is often measured by a number of standard ranking-specific metrics. What makes ranking metrics unique and suitable for this task is that, in ranking, it is often desirable to have fewer errors at higher ranked positions; this principle is reflected in many ranking metrics:

$$RR(\pi, \boldsymbol{y}) = \frac{1}{\min_j\{y_{\pi^{-1}(j)} > 0\}}, \tag{2}$$

$$RP(\pi, \boldsymbol{y}) = \frac{\sum_{j=1}^{n} y_j \pi(j)}{\sum_{j=1}^{n} y_j}, \tag{3}$$

$$DCG(\pi, \boldsymbol{y}) = \sum_{j=1}^{n} \frac{2^{y_j} - 1}{\log_2(1 + \pi(j))}, \tag{4}$$

$$NDCG(\pi, \boldsymbol{y}) = \frac{DCG(\pi, y)}{DCG(\pi^*, \boldsymbol{y})}, \tag{5}$$

where $y_i \in \boldsymbol{y}$ are ground-truth labels that induce π^*, and $\pi(i)$ is the rank of the i^{th} item in $\boldsymbol{x}$. RR is the reciprocal rank of the first relevant item. RP is the positions of items weighted by their relevance values [48]. DCG is the Discounted Cumulative Gain [20], and NDCG is DCG normalized by the maximum DCG obtained from the ideal ranked list π^*.

Note that, given m evaluation samples $(\pi_k, \boldsymbol{y}_k)$, $1 \leq k \leq m$, the mean of the above metrics is calculated and reported instead. For example, the mean reciprocal rank (MRR) is defined as:

$$MRR = \frac{1}{m} \sum_{k=1}^{N} RR(\pi_k, \boldsymbol{y}_k).$$

Our library supports commonly used ranking metrics and enables easy development and addition of arbitrary metrics.

2.5 Loss Functions

Learning-to-rank seeks to maximize a utility or equivalently minimize a cost or loss function. Assuming there exists a loss function $\ell(.)$, the objective of learning-to-rank is to find a ranking function f^* that minimizes the empirical loss over training samples:

$$f^* = \underset{f:\mathcal{X}^n \to \Pi^n}{\text{argmin}} \frac{1}{m} \sum_{(\boldsymbol{x}, \pi^*) \in S^m} \ell(\pi^*, f(\boldsymbol{x})). \tag{6}$$

Replacing f with a scoring function h as is often the case yields the following optimization problem:

$$h^* = \underset{h:\mathcal{X}^n \to \mathbb{R}^n}{\text{argmin}} \frac{1}{m} \sum_{(\boldsymbol{x}, \boldsymbol{y}) \in S^m} \hat{\ell}(\boldsymbol{y}, h(\boldsymbol{x})), \tag{7}$$

where $\hat{\ell(.)}$ is a loss function equivalent to $\ell(.)$ that acts on scores instead of permutations induced by scores.

This setup naturally prefers loss functions that are differentiable. Most ranking metrics, however, are either discontinuous or flat everywhere due to the use of the sort operation and as such cannot be directly optimized by learning-to-rank methods. With a few notable exceptions [29, 45], most learning-to-rank approaches therefore define and optimize a differentiable surrogate loss instead. They do so by, among other techniques, creating a smooth variant of popular ranking metrics [35, 39]; deriving tight upper-bounds on ranking metrics [42]; bypassing the requirement that a loss function be defined altogether [6]; or, otherwise designing losses that are loosely related to ranking metrics [7, 43].

Our library supports a number of surrogate loss functions. As an example of a **pointwise** loss, the sigmoid cross-entropy for binary relevance labels $y_j \in \{0, 1\}$ is computed as follows:

$$\hat{\ell}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{j=1}^{n} y_j \log(p_j) + (1 - y_j)\log(1 - p_j) \tag{8}$$

where $p_j = \frac{1}{1+\exp(-\hat{y}_j)}$, $\hat{\boldsymbol{y}} \triangleq h(\boldsymbol{x})$ are scores computed by the scoring function h. As an example of a **pairwise** loss in our library, the pairwise logistic loss is defined as:

$$\hat{\ell}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \sum_{j=1}^{n} \sum_{k=1}^{n} \mathbb{I}(y_j > y_k) \log(1 + \exp(\hat{y}_k - \hat{y}_j))) \tag{9}$$

where $\mathbb{I}(\cdot)$ is the indicator function. Finally, as an example of a **listwise** loss [11], our library provides the implementation of Softmax Cross-Entropy, ListNet [7], and ListMLE [43] among others. For example, the Softmax Cross-Entropy loss is defined as follows:

$$\hat{\ell}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{j=1}^{n} y_j \log(\frac{\exp(\hat{y}_j)}{\sum_{j=1}^{n} \exp(\hat{y}_j)}) \tag{10}$$

2.6 Item Weighting

Finally, we conclude this section by a note on bias in learning-to-rank. As discussed in Section 1, a number of studies have shown that click logs exhibit various biases including position bias [24]. In short, users are less likely to examine and click items at larger rank positions. Ignoring this bias when training a learning-to-rank model or when evaluating ranked lists may lead to a model with less generalization capacity and inaccurate quality measurements.

Unbiased learning-to-rank [25, 41] looks at handling such biases in relevance labels. One proposed method [40, 41] is to compute Inverse Propensity Weights (IPW) for each position in the ranked list. By incorporating these scores in the training process (usually by way of re-weighting items during loss computation), one may produce a better ranking function. Similarly, IPW-weighted variants of evaluation metrics attempt to counter such biases during evaluation. Our library supports the incorporation of such weights into the training and evaluation processes.

3 PLATFORM OVERVIEW

Popular use-cases of learning-to-rank, such as search or recommender systems [38], have several challenges. Models are generally trained over vast amounts of user data, so efficiency and scalability are critical. Features may be comprised of dense, categorical, or sparse types, and are often missing for some data points. These applications also require fast inference for real-time serving.

It is against the backdrop of these challenges that we believe, at a high level, neural networks as a class of machine learning models, and TensorFlow [1] as a machine learning framework are suitable for practical learning-to-rank problems.

Take efficiency and scalability as an example. The availability of vasts amounts of training data, along with an increase in computational power and thereby in our ability to train deeper neural networks with millions of parameters in a scalable manner have led to rapid adoption of neural networks for a variety of applications. In fact, deep neural networks have gained immense popularity, with applications in Natural Language Processing, Computer Vision, Speech Signal Processing, and many other areas [19]. Recently, neural networks have been shown to be effective in applications of Information Retrieval as well [31].

Neural networks can also process heterogeneous features more naturally. Evidence [4] suggests that neural networks learn effective representations of categorical and sparse features. These include words or characters in Natural Language Processing [30], phonemes in Speech Processing [36], or raw pixels in Computer Vision [27]. Such "representation learning" is usually achieved by way of converting raw, unstructured data into a dense real valued vector, which, in turn, is treated as a vector of implicit features for subsequent neural network layers. For example, discrete elements such as words or characters from a finite vocabulary are transformed using an embedding matrix—also referred to as embedding layer or embeddings [4]—which maps every element of the vocabulary to a learned, dense representation. This particular representation learning is useful for learning-to-rank over documents, web pages or other textual data.

Given these properties, neural networks are excellent candidates for modeling a score-and-sort approach to learning-to-rank. In particular, the scoring function $h(.)$ in Equation 1 can be parameterized by a neural network. In such a setup, feature representations can be jointly learned with the parameters of $h(.;\theta)$ while minimizing the objective $\hat{\ell}$ averaged over the training data. This is the general setup we adopt and implement in the TensorFlow Ranking library.

TensorFlow Ranking is built on top of TensorFlow, a popular open-source library for large scale training, evaluation and serving of machine learning and deep learning models. TensorFlow supports high performance tensor (multi-dimensional vector) manipulation and computation via what is referred to as "computational graphs." A computational graph expresses the logic of a sequence of tensor manipulations, with each node in the graph corresponding to a single tensor operation (*op*). For example, a node in the computation graph may multiply an "input" tensor with a "weight" tensor, a subsequent node may add a "bias" tensor to that product, and a final node may pass the resultant tensor through the Sigmoid function.

The concept of a computation graph with tensor operations simplifies the implementation of the *backpropagation* [9] algorithm for training neural networks. In a forward pass, the values at each node are computed by composing a sequence of tensor operations, and in a backward pass, gradients are accumulated in the the reverse fashion. TensorFlow enables such propagation of gradients through automatic differentiation [1]: each operation in the computation graph is equipped with a gradient expression with respect to its input tensors. In this way, the gradient of a complex composition of tensor operations can be automatically inferred during a backward pass through the computation graph. This allows for composition of a large number of operations to construct deeper networks.

Another computationally attractive property of the TensorFlow framework is its support, via TensorFlow Estimator [12], of distributed training of neural networks. TensorFlow Estimator is an abstract library that takes a high-level training, evaluation, or prediction logic and hides the execution logic from developers. An `Estimator` object encapsulates two major abstract components that can be further customized: (1) `input_fn`, which reads in data from a persistent storage and creates tensors for features and labels, and (2) `model_fn` which processes input features and labels, and depending on the mode (`TRAIN`, `EVAL`, `PREDICT`), returns a loss value, evaluation metrics, or predictions. The computation graph expressed within the `model_fn` may depend on the mode. This is particularly useful for learning-to-rank because the model may need entire lists of items during training (to compute a listwise loss, for example), but during serving, it may score each item independently. The model function itself can be expressed as a combination of a `logits` builder and a `Head` abstraction, where the logits builder generates the values in the forward computation of the graph, and the `Head` object defines loss objectives and associated metrics.

These abstractions, along with modular design, the ability to distribute the training of a neural network and to serve on a variety of high-performance hardware platforms (CPU/GPU/TPUs) are what make the TensorFlow ecosystem a suitable platform for a neural network-based learning-to-rank library. In the next section, we discuss how the design principles in TensorFlow and the `Estimator` workflow inspire the design of the TensorFlow Ranking library.

4 COMPONENTS

Motivated by design patterns in TensorFlow and the `Estimator` framework, TensorFlow Ranking constructs computational subgraphs via callbacks for various aspects of a learning-to-rank model such as scoring function $h(.)$, losses $\hat{\ell}(.)$, and evaluation metrics (see Section 2). These subgraphs are combined together in the

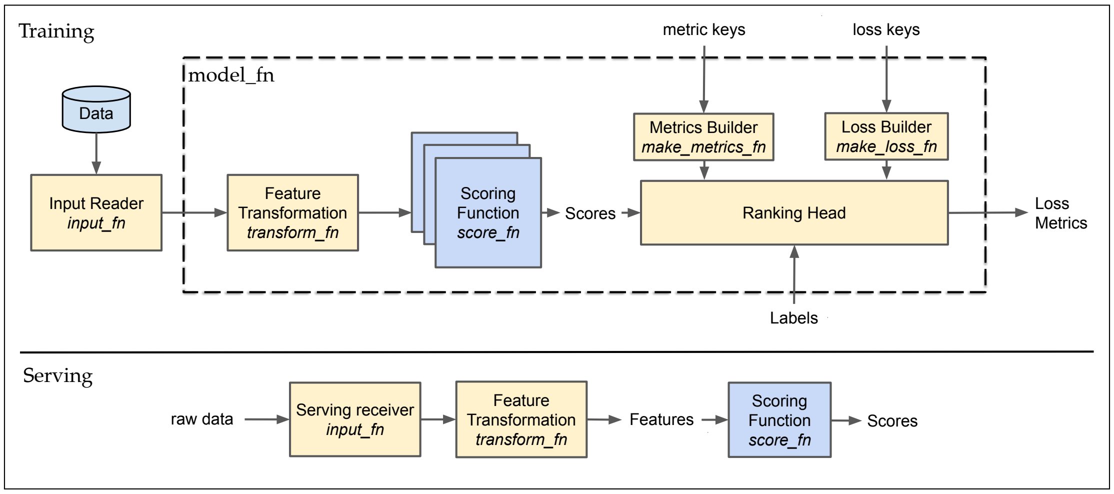

Figure 1: TensorFlow Ranking Architecture

`Estimator` framework, via a custom callback to create a `model_fn`. The `model_fn` itself can be decomposed into a `logits` builder, which represents the output of the neural network layers, and a `Head` abstraction, which encapsulates loss and metrics. This decomposition is particularly suitable to learning-to-rank problems in the score-and-sort approach. The `logits` builder corresponds to the scoring function, defined in Section 2.3, and the `Head` corresponds to losses and metrics.

It is important to note that this decomposition also provides modularity and the ability to switch between various combinations of scoring functions and ranking heads. For example, as we show in the code examples below, switching between a pointwise or a listwise loss, or incorporating embedding for sparse features into the model can be expressed in a single line code change. We found this modularity to be valuable in practical applications, where rapid experimentation is often required.

The overall architecture of TensorFlow Ranking library is illustrated in Figure 1. The key components of the library are: (1) data reader, (2) transform function, (3) scoring function, (4) ranking loss functions, (5) evaluation metrics, (6) ranking head, and (7) a `model_fn` builder. In the remainder of this section, we will describe each component in detail.

4.1 Reading data using `input_fn`

The `input_fn`, as shown in Figure 1, reads in data either from a persistent storage or data generated by another process, to produce dense and sparse tensors of the appropriate type, along with the labels. The library provides support for several popular data formats (`LIBSVM`, `tf.SequenceExample`). Furthermore, it allows developers to define customized data readers.

Consider the case when the user would like to construct a custom `input_fn`. The user defines a parser for a batch of serialized datapoints that returns a feature dictionary containing 2-D and 3-D tensors for "context" and "per-item" features respectively. This user-defined parser function is passed to a dataset builder, which generates features and labels. Per-item features are those features that are particular to each example or item in the list of items. Context features, on the other hand, are independent of the items in the list. In the context of document search, features extracted from documents are per-item features and other document-independent features (such as query/session/user features) are considered context features. We represent per-item features by a 3-D tensor (where dimensions correspond to queries, items, and feature values) and context features by a 2-D tensor (where dimensions correspond to queries and feature values). The following code snippet shows how one may construct a custom `input_fn`.

```
def _parse_single_datapoint(serialized):
  """User defined logic to parse a batch of serialized datapoints to
      a dictionary of 2-D and 3-D tensors."""
  return features

def input_fn(file_pattern):
  """Generate features and labels from input data."""
  dataset = tfr.data.build_ranking_dataset_with_parsing_fn(
  file_pattern=file_pattern,
  parsing_fn=_parse_single_datapoint, ...)
  features = dataset.make_one_shot_iterator().get_next()
  label = tf.squeeze(features.pop(_LABEL_FEATURE), axis=2)
  return features, label
```

4.2 Feature Transformation with `transform_fn`

As discussed in Section 3, sparse features such as words or n-grams can be transformed to dense features using (learned) embeddings. More generally, any raw feature may require some form of transformation. Such transformations may be implemented in `transform_fn`, a function that is applied to the output of `input_fn`. The library provides standard functions to transform sparse features to dense features based on feature definitions. A feature definition in TensorFlow is a user-defined `tf.FeatureColumn` [12], an expression of the type and attributes of features. Given feature definitions, the transform function produces dense 2-D or 3-D tensors for context and per-item features respectively. The following code snippet demonstrates an implementation of the `transform_fn`. Context and per-item feature definitions are passed to the encode_listwise_features function, which in turn returns dense tensors for each of context and per-item features.

```python
def make_transform_fn():
  def _transform_fn(features, mode):
    context_feature_columns = {"unigrams": embedding_column(
      categorical_column("unigrams", dimension=10, vocab_list=
      _VOCAB_LIST))}
    example_feature_columns = {"utility": numeric_column("utility",
        shape=(1,), default_value=0.0, dtype=float32)}
    # 2-D context tensors and 3-D per-item tensors.
    context_features, example_features = tfr.feature.
      encode_listwise_features(
        features,
        input_size=2,
        context_feature_columns=context_feature_columns,
        example_feature_columns=example_feature_columns)
    return context_features, example_features
  return _transform_fn
```

4.3 Feature Interactions using scoring_fn

The library allows users to build arbitrary scoring functions defined in Section 2.3. A scoring function takes batched tensors in the form of 2-D context features and 3-D per-item features, and returns a score for a single item or a group of items. The scoring function is supplied to the model builder via a callback. The model uses the scoring function internally to generate scores during training and inference, as shown in Figure 1. The signature of the scoring function is shown below, where mode, params, and config are input arguments for model_fn that supply model hyperparameters and configuration for distributed training [12]. The code snippet below, constructs a 3-layer feedforward neural network with ReLUs [32].

```python
def make_score_fn():
  def _score_fn(context_features, group_features, mode, params,
      config):
    """Scoring network for feature interactions."""
    net = concat(layers.flatten(group_features.values()), layers.
      flatten(context_features.values()))
    for i in range(3):
      net = layers.dense(net, units=128, activation="relu")
    logits = layers.dense(cur_layer, units=1)
    return logits
  return _score_fn
```

4.4 Ranking Losses

Here, we describe the APIs for building loss functions defined in Section 2.5. Losses in TensorFlow are functions that take in inputs, labels and a weight, and return a weighted loss value. The library has a pre-defined set of pointwise, pairwise and listwise ranking losses. The loss key is an enum over supported loss functions. These losses are exposed using the factory function tfr.losses.make_loss_fn that takes a loss key (name) and a weights tensor, and returns a loss function compatible with Estimator. The code snippet below shows how to built the loss function for softmax cross-entropy loss.

```python
# Define loss key(s).
loss_key = tfr.losses.RankingLossKey.SOFTMAX_LOSS
# Build the loss function.
loss_fn = tfr.losses.make_loss_fn(loss_key, ...)
# Generating loss value from the loss function.
loss_scalar = loss_fn(scores, labels, ...)
```

4.5 Ranking Metrics

The library provides an API to compute most common ranking metrics defined in Section 2.4. Similar to loss functions, a metric can be instantiated using a factory function that takes a metric key and a weights tensor, and returns a metric function compatible with Estimator. The metric function itself takes in predictions, labels, and weights to compute a scalar measure. The metric key is an enum over supported metric functions, described in Section 2.4. During evaluation, the library supports computing both weighted and unweighted metrics, for example weights defined in Section 2.6, which facilitates evaluation in the context of unbiased learning-to-rank. The code snippet below shows how to build metric function.

```python
def eval_metric_fns():
  """Returns a dict from name to metric functions."""
  metric_fns = {
      "metric/ndcg@5": tfr.metrics.make_ranking_metric_fn(
          tfr.metrics.RankingMetricKey.NDCG, topn=5)
  }
  return metric_fns
```

4.6 Ranking Head

In the Estimator workflow, the Head API is an abstraction that encapsulates losses and metrics: Given a pair of Estimator-compatible loss function and metric function along with scores from a neural network, Head computes the values of the loss and metric and produces model predictions as output. The library provides a Ranking Head: a Head object with built-in support for ranking losses built in Section 4.4 and ranking metrics of Section 4.5. The signature of Ranking Head is shown below.

```python
def _train_op_fn(loss):
  """Defines train op used in ranking head."""
  return tf.contrib.layers.optimize_loss(
      loss=loss,
      global_step=tf.train.get_global_step(),
      learning_rate=hparams.learning_rate,
      optimizer="Adagrad")

ranking_head = tfr.head.create_ranking_head(
    loss_fn=tfr.losses.make_loss_fn(_LOSS),
    eval_metric_fns=eval_metric_fns(),
    train_op_fn=_train_op_fn)
```

4.7 Model Builder

A model builder, model_fn, is what puts all the different pieces together: scoring function, transform function, losses and metrics via ranking head. Recall that the model_fn returns operations related to predictions, metrics, and loss optimization. The output of model_fn and the graph constructed depends on the mode TRAIN, EVAL, or PREDICT. These are all handled internally by the library through make_groupwise_ranking_fn. The signature of a model builder, along with the overall flow to build a ranking Estimator is shown in Figure 1. The components of the ranking library can be used to construct a ranking model in several ways. The inbuilt model builder, make_groupwise_ranking_fn, provides a ranking model with multivariate scoring function and configurable losses and metrics. The user can also define a custom model builder which can use components from the library, such as losses or metrics.

```python
input_fn = tfr.data.read_batched_sequence_example_dataset(
      file_pattern,...)
ranking_estimator = estimator.Estimator(
      model_fn=tfr.model.make_groupwise_ranking_fn(
          group_score_fn=make_score_fn(),
          group_size=group_size,
          transform_fn=make_transform_fn(),
          ranking_head=ranking_head),
      params=hparams)
# Training loop.
for _ in range(num_train_steps):
  ranking_estimator.train(input_fn(TRAIN_FILES),...)
# Evaluation.
ranking_estimator.evaluate(input_fn(EVAL_FILES),...)
```

Ranking models have a crucial training-serving discrepancy. During training the model receives a list of items, but during serving, it could potentially receive independent items which are generated by a separate retrieval algorithm. The ranking model builder handles this by generating a graph compatible with serving requirements and exporting it as a `SavedModel` [33]: a language agnostic object that can be loaded by the serving code, which can be in any low-level language such as C++ or Java.

5 USE CASES

Tensorflow Ranking is already deployed in several production systems at Google. In this section, we demonstrate the effectiveness of our library for two real-world ranking scenarios: *Gmail search* [40, 47] and *document recommendation in Google Drive* [38]. In both cases the model is trained on large quantities of click data that is beyond the capabilities of existing open source learning-to-rank packages, e.g., RankLib. In addition, in the Gmail setting, our model contains sparse textual features that cannot be naturally handled by the existing learning-to-rank packages.

5.1 Gmail Search

In one set of experiments, we evaluate several ranking models trained on search logs from Gmail. In this service, when a user types a query into the search box, five results are shown and user clicks (if any) are recorded and later used as relevance labels. To preserve user privacy, we remove personal information and anonymize data using k-anonymization. We obtain a set of features that consists of both dense and sparse features. Sparse features include word- and character-level n-grams derived from queries and email subjects. The vocabulary of n-grams is pruned to retain only n-grams that occur across more than k users. This is done to preserve user privacy, as well as to promote a common vocabulary for learning a shared representations across users. In total, we collect about 250M queries and isolate 10% of those to construct an evaluation set. Losses and metrics are weighted by Inverse Propensity Weighting [40] computed to counter position bias.

5.2 Document Recommendation in Drive

Quick Access in Google Drive [38] is a zero-state recommendation engine that surfaces documents currently relevant to the user when she visits the Drive home screen. We evaluate several ranking models trained on user click data over these recommended results. The set of features consists of mostly dense features, as described in Tata et al. [38]. In total we collected about 30M instances and set aside 10% of the set for evaluation.

Table 1: Model performance with various loss functions. ΔM denotes % improvement in metric M over the *Sigmoid Cross Entropy* baseline. Best performance per column is in bold.

(a) Gmail Search	ΔMRR	ΔARP	ΔNDCG
Sigmoid Cross Entropy (Pointwise)	–	–	–
Logistic Loss (Pairwise)	+1.52	+1.64	+1.00
Softmax Cross Entropy (Listwise)	**+1.80**	**+1.88**	**+1.57**
(b) Quick Access	**ΔMRR**	**ΔARP**	**ΔNDCG**
Sigmoid Cross Entropy (Pointwise)	–	–	–
Logistic Loss (Pairwise)	+0.70	+1.86	+0.35
Softmax Cross Entropy (Listwise)	**+1.08**	**+1.88**	**+1.05**

5.3 Model Effectiveness

5.3.1 Setup and Evaluation. We consider a simple 3-layer feed-forward neural network with ReLU [32] non-linear activation units and dropout regularization [37]. We train models using pointwise, pairwise, and listwise losses defined in Section 2.5, and use `Adagrad` [15] to optimize the objective. We set the learning rate to 0.1 for Quick Access, and 0.3 for Gmail Search.

The models are evaluated using the metrics defined in Section 2.4. Due to the proprietary nature of the models, we only report relative improvements with respect to a given baseline. Due to the large size of the evaluation datasets, all the reported improvements are statistically significant.

5.3.2 Effect of listwise losses. Table 1 summarizes the impact of different loss functions as measured by various ranking metrics for Gmail and Drive experiments respectively. We observe that a listwise loss performs better than a pairwise loss, which is in turn better than a pointwise loss. This observation confirms the importance of listwise losses for ranking problems over pointwise and pairwise losses, both in search and recommendation settings.

5.3.3 Incorporating sparse features. Unlike other approaches to learning to rank, like linear models, SVMs, or GBDTs, neural networks can effectively incorporate sparse features like query or document text. Neural networks handle sparse features by using embedding layers, which map each sparse value to a dense representation. These embedding matrices can be jointly trained along with the neural network parameters, allowing us to learn effective dense representations for sparse features in the context of a given task (e.g., search or recommendation). As prior work shows, these representations can substantially improve model effectiveness on large-scale collections [44].

To demonstrate this point, Table 2 reports the relative improvements from using sparse features in addition to dense features on Gmail Search. We use an embedding layer of size 20 for each sparse feature. We note that adding sparse features significantly boosts ranking quality across metrics and loss functions. This confirms the importance of sparse textual features in large-scale applications, and the effectiveness of TensorFlow Ranking in employing these features.

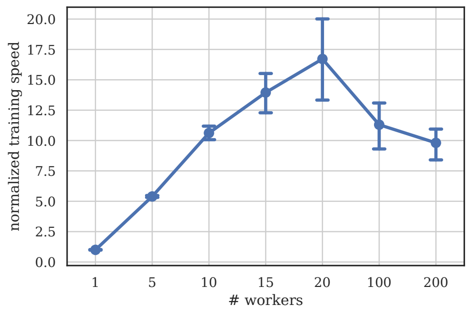

Figure 2: Normalized training speed for Gmail Search as a function of number of workers.

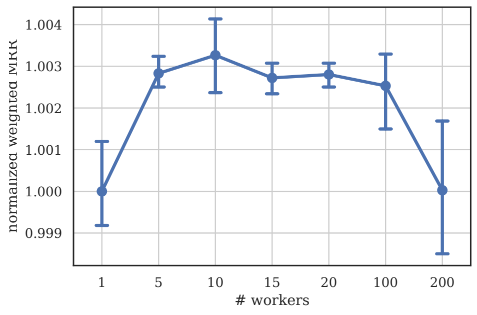

Figure 3: Normalized weighted Mean Reciprocal Rank for Gmail Search as a function of number of workers.

5.4 Distributed Training

Distributed training is crucial for applications of learning-to-rank to large scale datasets. Models built using the `Estimator` class allow for easy switching between local and distributed training, without changing the core functionality of the model. This is achieved via the `Experiment` class [12] given a distribution strategy.

Distributed training using TensorFlow consists of a number of worker tasks and parameter servers. We investigate the scalability of the model built for Gmail Search, the larger of our two datasets. We examine the effect of increasing the number of workers on the training speed. For the distributed strategy, we use between-graph replication, where each worker replicates the graph on a different subset of the data, and asynchronous training, where the gradients from different workers are asynchronously collected to update the gradients for the model.

For each of the following experiments, the number of workers ranges between 1 and 200, while the number of training epochs is fixed at 20 million. For robustness, each configuration is run 5 times, and the 95% confidence intervals are plotted.

5.4.1 Effect of scaling on training speed. We look at the impact of increasing the number of workers on the training speed in Figure 2. We measure training speed by the number of training steps executed per second. Due to the proprietary nature of the data, we report training speed normalized by the average training speed for one worker. The training scales up linearly in the initial phase, however when the pool of workers becomes too large, we are faced with two confounding factors. First is the communication overhead for gradient updates. Second, more workers stay idle waiting for data to become available. Therefore, the I/O costs begin to dominate, and the total training time stagnates and even slows down, as seen in the 20+ worker region of Figure 2.

Table 2: Model performance with dense and sparse textual features. ΔM denotes % improvement in metric M over the corresponding baseline, when only dense features are used.

	ΔMRR	ΔARP	ΔNDCG
Sigmoid Cross Entropy (Pointwise)	+6.06	+6.87	+3.92
Logistic Loss (Pairwise)	+5.40	+6.25	+3.51
Softmax Cross Entropy (Listwise)	+5.69	+6.25	+3.70

5.4.2 Effect of scaling on metrics. In Figure 3, we examine the impact of increasing the number of workers on weighted MRR. Due to the proprietary nature of the data, we report the metric value normalized by the average metric for one worker.

We observe that scaling does not generally have a significant impact on MRR, with two exceptions: using (a) a single worker, and (b) a large worker pool. We believe that the former requires more training cycles to achieve a comparable MRR. In the latter setting, gradient updates aggregated from too many workers become inaccurate, sending the model in a direction that is not well-aligned with the true gradient. However, in both cases, the overall effect on MRR is very small (roughly 0.03%), demonstrating the robustness of scaling with respect to model performance.

6 CONCLUSION

In this paper we introduced TensorFlow Ranking—a scalable learning-to-rank library in TensorFlow. The library is highly configurable and has easy-to-use APIs for scoring mechanisms, loss functions, and evaluation metrics. Unlike the existing learning-to-rank open source packages which are designed for small datasets, TensorFlow Ranking can be used to solve real-world, large-scale ranking problems with hundreds of millions of training examples, and scales well to large clusters. TensorFlow Ranking is already deployed in several production systems at Google, and in this paper we empirically demonstrate its effectiveness for Gmail search and Quick Access in Google Drive [38]. Our experiments show that TensorFlow Ranking can (a) leverage listwise loss functions, (b) effectively incorporate sparse features through embeddings, and (c) scale up without a significant drop in metrics. TensorFlow Ranking is available to the open source community, and we hope that it facilitates further academic research and industrial applications.

7 ACKNOWLEDGEMENTS

We thank the members of the TensorFlow team for their advice and support: Alexandre Passos, Mustafa Ispir, Karmel Allison, Martin Wicke, Clemens Mewald and others. We extend our special thanks to our collaborators, interns and early adopters: Suming Chen, Zhen Qin, Chirag Sethi, Maryam Karimzadehgan, Makoto Uchida, Yan Zhu, Qingyao Ai, Brandon Tran, Donald Metzler, Mike Colagrosso, Patrick McGregor and many others at Google who helped in evaluating and testing the early versions of TF-Ranking.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *12th USENIX Symposium on Operating Systems Design and Implementation*. 265–283.

[2] Qingyao Ai, Jiaxin Mao, Yiqun Liu, and W Bruce Croft. 2018. Unbiased Learning to Rank: Theory and Practice. In *2018 ACM SIGIR International Conference on Theory of Information Retrieval*. 1–2.

[3] Qingyao Ai, Xuanhui Wang, Nadav Golbandi, Michael Bendersky, and Marc Najork. 2018. Learning Groupwise Scoring Functions Using Deep Neural Networks. *arXiv preprint arXiv:1811.04415* (2018).

[4] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 8 (2013), 1798–1828.

[5] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. In *22nd International Conference on Machine Learning*. 89–96.

[6] Christopher J.C. Burges. 2010. *From RankNet to LambdaRank to LambdaMART: An Overview*. Technical Report Technical Report MSR-TR-2010-82. Microsoft Research.

[7] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *24th International Conference on Machine Learning*. 129–136.

[8] Olivier Chapelle, Donald Metzler, Ya Zhang, and Pierre Grinspan. 2009. Expected Reciprocal Rank for Graded Relevance. In *18th ACM Conference on Information and Knowledge Management*. 621–630.

[9] Yves Chauvin and David E Rumelhart. 2013. *Backpropagation: theory, architectures, and applications*. Psychology Press.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274* (2015).

[11] Wei Chen, Tie-Yan Liu, Yanyan Lan, Zhi-Ming Ma, and Hang Li. 2009. Ranking Measures and Loss Functions in Learning to Rank. In *Advances in Neural Information Processing Systems*. 315–323.

[12] Heng-Tze Cheng, Zakaria Haque, Lichan Hong, Mustafa Ispir, Clemens Mewald, Illia Polosukhin, Georgios Roumpos, D Sculley, Jamie Smith, David Soergel, et al. 2017. Tensorflow estimators: Managing simplicity vs. flexibility in high-level machine learning frameworks. In *23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1763–1771.

[13] Wei Chu and Zoubin Ghahramani. 2005. Preference Learning with Gaussian Processes. In *22nd International Conference on Machine Learning*. 137–144.

[14] Nick Craswell. 2009. Mean reciprocal rank. In *Encyclopedia of Database Systems*. Springer, 1703–1703.

[15] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12 (July 2011), 2121–2159.

[16] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of Statistics* 29, 5 (2001), 1189–1232.

[17] Norbert Fuhr. 1989. Optimum Polynomial Retrieval Functions Based on the Probability Ranking Principle. *ACM Transactions on Information Systems* 7, 3 (1989), 183–204.

[18] Fredric C. Gey. 1994. Inferring Probability of Relevance Using the Method of Logistic Regression. In *17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 222–231.

[19] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep Learning*. MIT Press Cambridge.

[20] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems* 20, 4 (2002), 422–446.

[21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *22nd ACM International Conference on Multimedia*. 675–678.

[22] Thorsten Joachims. 2002. Optimizing Search Engines Using Clickthrough Data. In *8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 133–142.

[23] Thorsten Joachims. 2006. Training linear SVMs in linear time. In *12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 217–226.

[24] Thorsten Joachims, Laura Granka, Bing Pan, Helene Hembrooke, and Geri Gay. 2005. Accurately Interpreting Clickthrough Data As Implicit Feedback. In *28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 154–161.

[25] Thorsten Joachims, Adith Swaminathan, and Tobias Schnabel. 2017. Unbiased Learning-to-Rank with Biased Feedback. In *10th ACM International Conference on Web Search and Data Mining*. 781–789.

[26] 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 *Advances in Neural Information Processing Systems 30*. 3146–3154.

[27] Yann LeCun and Yoshua Bengio. 1995. Convolutional networks for images, speech, and time series. In *The Handbook of Brain Theory and Neural Networks*, Michael A Arbib (Ed.). MIT Press, 255–258.

[28] Hang Li. 2011. Learning to rank for information retrieval and natural language processing. *Synthesis Lectures on Human Language Technologies* 4, 1 (2011), 1–113.

[29] Donald A Metzler, W Bruce Croft, and Andrew Mccallum. 2005. *Direct maximization of rank-based metrics for information retrieval*. CIIR report 429. University of Massachusetts.

[30] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*. 3111–3119.

[31] Bhaskar Mitra and Nick Craswell. 2017. Neural Models for Information Retrieval. *arXiv preprint arXiv:1705.01509* (2017).

[32] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted Boltzmann machines. In *27th International Conference on Machine Learning*. 807–814.

[33] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, high-performance ML serving. *arXiv preprint arXiv:1712.06139* (2017).

[34] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *AutoDiff Workshop at NIPS 2017*.

[35] Tao Qin, Tie-Yan Liu, and Hang Li. 2010. A General Approximation Framework for Direct Optimization of Information Retrieval Measures. *Information Retrieval* 13, 4 (2010), 375–397.

[36] Miikka P Silfverberg, Lingshuang Jack Mao, and Mans Hulden. 2018. Sound Analogies with Phoneme Embeddings. *Proc. of the Society for Computation in Linguistics (SCiL)* (2018), 136–144.

[37] 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.

[38] Sandeep Tata, Alexandrin Popescul, Marc Najork, Mike Colagrosso, Julian Gibbons, Alan Green, Alexandre Mah, Michael Smith, Divanshu Garg, Cayden Meyer, et al. 2017. Quick Access: Building a Smart Experience for Google Drive. In *23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1643–1651.

[39] Michael Taylor, John Guiver, Stephen Robertson, and Tom Minka. 2008. SoftRank: Optimizing Non-smooth Rank Metrics. In *1st International Conference on Web Search and Web Data Mining*. 77–86.

[40] Xuanhui Wang, Michael Bendersky, Donald Metzler, and Marc Najork. 2016. Learning to Rank with Selection Bias in Personal Search. In *39th International ACM SIGIR conference on Research and Development in Information Retrieval*. 115–124.

[41] Xuanhui Wang, Nadav Golbandi, Michael Bendersky, Donald Metzler, and Marc Najork. 2018. Position Bias Estimation for Unbiased Learning to Rank in Personal Search. In *11th ACM International Conference on Web Search and Data Mining*. 610 –618.

[42] Xuanhui Wang, Cheng Li, Nadav Golbandi, Michael Bendersky, and Marc Najork. 2018. The LambdaLoss Framework for Ranking Metric Optimization. In *27th ACM International Conference on Information and Knowledge Management*. 1313–1322.

[43] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. 2008. Listwise Approach to Learning to Rank: Theory and Algorithm. In *25th International Conference on Machine Learning*. 1192–1199.

[44] Chenyan Xiong, Zhuyun Dai, Jamie Callan, Zhiyuan Liu, and Russell Power. 2017. End-to-End Neural Ad-hoc Ranking with Kernel Pooling. In *40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 55–64.

[45] Jun Xu and Hang Li. 2007. AdaRank: A Boosting Algorithm for Information Retrieval. In *30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 391–398.

[46] Yisong Yue, Rajan Patel, and Hein Roehrig. 2010. Beyond Position Bias: Examining Result Attractiveness As a Source of Presentation Bias in Clickthrough Data. In *19th International Conference on World Wide Web*. 1011–1018.

[47] Hamed Zamani, Michael Bendersky, Xuanhui Wang, and Mingyang Zhang. 2017. Situational Context for Ranking in Personal Search. In *26th International Conference on World Wide Web*. 1531–1540.

[48] Mu Zhu. 2004. *Recall, precision and average precision*. Technical Report. Department of Statistics and Actuarial Science, University of Waterloo.