RAX: Composable Learning-to-Rank using JAX

Rolf Jagerman
Google Research
jagerman@google.com

Xuanhui Wang
Google Research
xuanhui@google.com

Honglei Zhuang
Google Research
hlz@google.com

Zhen Qin
Google Research
zhenqin@google.com

Michael Bendersky
Google Research
bemike@google.com

Marc Najork
Google Research
najork@google.com

ABSTRACT

RAX is a library for *composable* Learning-to-Rank (LTR) written entirely in JAX. The goal of RAX is to facilitate easy prototyping of LTR systems by leveraging the flexibility and simplicity of JAX. RAX provides a diverse set of popular ranking metrics and losses that integrate well with the rest of the JAX ecosystem. Furthermore, RAX implements a system of ranking-specific function transformations which allows fine-grained customization of ranking losses and metrics. Most notably RAX provides `approx_t12n`: a function transformation (t12n) that can transform *any* of our ranking metrics into an approximate and differentiable form that can be optimized. This provides a systematic way to directly optimize neural ranking models for ranking metrics that are not easily optimizable in other libraries. We empirically demonstrate the effectiveness of RAX by benchmarking neural models implemented using Flax and trained using RAX on two popular LTR benchmarks: *WEB30K* and *Istella*. Furthermore, we show that integrating ranking losses with T5, a large language model, can improve overall ranking performance on the *MS MARCO* passage ranking task. We are sharing the RAX library with the open source community as part of the larger JAX ecosystem at https://github.com/google/rax.

CCS CONCEPTS

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

KEYWORDS

Learning to Rank; JAX

ACM Reference Format:
Rolf Jagerman, Xuanhui Wang, Honglei Zhuang, Zhen Qin, Michael Bendersky, and Marc Najork. 2022. RAX: Composable Learning-to-Rank using JAX. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22), August 14–18, 2022, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3534678.3539065

1 INTRODUCTION

Learning-to-Rank (LTR) concerns itself with learning a *ranking model* or *ranker* from labeled relevance data or interaction logs. Unlike traditional classification or regression problems, where the goal is to predict a label or value for each individual item, ranking problems aim to predict an *ordering* on a list of items in order to maximize the utility of that list. Ranking models are widely applicable and actively used in Web search, recommendation systems, question answering systems, and more. Due to the applicability of ranking models to many different domains, several open-source LTR libraries have been developed, including TF-Ranking [33] and XGBoost [9]. Although the current offering of ranking libraries is impressive, none of them function natively within JAX [4], a new numerical computing framework that is rapidly gaining popularity.

JAX provides tools for automatic differentiation, compilation to accelerators, and batching in the form of function transformations that compose [4]. Unlike Tensorflow [1] or PyTorch [34] which focus mostly on deep learning, JAX is a library for arbitrary differentiable programming. JAX makes it possible to express complex machine learning systems that leverage accelerators, while being written entirely in Python. Due to its ease-of-use, JAX has been used in computer vision [11], physics [41], differential privacy [43], reinforcement learning [3] and other fields of machine learning.

However, adoption of JAX in the Information Retrieval (IR) and Learning-to-Rank (LTR) communities is non-existent, as there is currently no library that provides ranking capabilities to the JAX ecosystem. This motivated us to build RAX to bridge this gap. However, implementing ranking functionality in JAX is challenging for a number of reasons. First, to be compatible with the JAX ecosystem, any ranking functionality in JAX needs to uphold the Pure and Statically Composed (PSC) function requirement of JAX (see Section 3.2). Second, implementing ranking metric optimization, a key technique for LTR, in a systematic way is not trivial. Existing LTR offerings implement approximate metrics as entirely separate loss implementations, which has a very loose connection between the metrics and the losses. It thus easily leads to code duplication and does not scale to new metrics. Finally, how to enable stochastic estimation [5, 45, 48] for all of the ranking losses in a systematic way, without introducing duplicated code, is desired but not immediately clear.

To overcome these challenges, this paper introduces RAX[1], the first ranking library in the JAX ecosystem. RAX provides three main components that make ranking functionality feasible in the JAX ecosystem: (1) ranking losses, (2) ranking metrics, and, (3) function transformations. First, the ranking losses provide implementations of standard ranking losses. Each ranking loss is implemented according to a consistent function signature which allows users of RAX to easily switch among different losses in the same code base. Second, the ranking metrics provide implementations of standard ranking metrics, which are highly configurable. For example, our

[1]RAX is available at https://github.com/google/rax

KDD '22, August 14–18, 2022, Washington, DC, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9385-0/22/08.
https://doi.org/10.1145/3534678.3539065

implementation of the NDCG metric [24] accepts customized gain and discount functions, which makes them re-usable in different contexts. Lastly, the function transformations provide a systematic way to support ranking metric optimization. One example is `rax.approx_t12n` which can transform any of our ranking metrics into an approximate form that can be optimized using gradient descent. Another example is `rax.gumbel_t12n` which transforms any of our ranking losses into stochastic versions via Gumbel sampling [5]. Each of the components of Rax is implemented using a functionally pure design, which makes them compatible with JAX and many libraries in the JAX ecosystem. Furthermore, Rax provides a number of examples that help users get started, including examples on how to use Rax to build ranking models with Flax [20] and Optax [22].

We validate the effectiveness of our new library Rax by running LTR benchmarks on both the *WEB30K* [36] and *Istella* [10] datasets. We experiment with a large number of ranking losses that are provided by Rax. Our results suggest constructing new ranking losses by lower bounding or approximating ranking metrics, a functionality made possible by Rax function transformations, can provide significantly better ranking models. Furthermore, we integrate several of the ranking losses offered by Rax with T5X, a JAX implementation of the T5 [39] model and test them by finetuning a T5 model on the *MS MARCO* [30] passage ranking dataset. We find that by fine-tuning on listwise ranking losses we can obtain significantly better results on this task than what we could obtain with just pointwise or pairwise losses.

2 RELATED WORK

Learning-to-Rank (LTR) is a widely studied area of Information Retrieval (IR). There are numerous algorithms, models and ranking losses and metrics that are relevant to both researchers and practicioners of LTR systems. Because of this, a large number of libraries exist that provide LTR functionality. Notable libraries include TF-Ranking[2] [33], XGBoost[3] [9], SVM-Rank[4] [25], sofia-ml[5], RankLib[6], QuickRank[7] [8], RankPy[8], PyLTR[9], jforest[10] [16], and, PyTorchLTR[11] [23], to name a few.

Out of these libraries, TF-Ranking [33] and XGBoost [9] are the most widely used open source software solutions for LTR. TF-Ranking is a "scalable learning-to-rank library in TensorFlow". The focus of TF-Ranking is on training neural ranking models on large-scale problems with hundreds of millions of training examples. Similarly, XGBoost is a scalable machine learning designed for tree boosting algorithms that can be applied to regression, classification and ranking. Both libraries focus strongly on *scalability* and provide strong ranking performance across existing benchmarks.

Rax is similar to these libraries in the fact that they all provide standard implementations of ranking losses and ranking metrics. However, as in the JAX ecosystem, Rax differentiates itself by placing emphasis on *usability* instead of *scalability*. One way Rax does this is by providing novel ways of re-purposing ranking metric implementations as ranking losses, which allows new ways of performing ranking metric optimization that was not easily accomplished in libraries such as TF-Ranking. Our design strictly consists of PSC functions (which will be expanded upon in Section 3.2). This design decision allows us to leverage the powerful machinery of JAX to both accelerate computation to GPU/TPU devices (via `jax.jit`) and scale computation to multi-device clusters (via `jax.pmap`). Furthermore, by focusing on ranking losses and metrics, Rax does not tie itself to any specific choice of model. This means that Rax could be used for both neural and tree algorithms as long as there are JAX libraries to support those models. At the time of writing, there are no libraries offering tree boosting functionality in JAX but there are numerous neural network libraries such as Flax [20] and Haiku [3]. This makes it possible to use Rax to train neural LTR models.

3 BACKGROUND

Before we describe the design of Rax, we first provide background on Learning-to-Rank (LTR) and JAX. Furthermore, this section establishes notation that will be used throughout this paper.

3.1 Learning-to-Rank

In this section, we will introduce the general problem of Learning-to-Rank (LTR) and our notation. Let $\mathcal{X}$ denote the universe of all possible items and $x \subset \mathcal{X}$ be a subset of n candidate items that we wish to rank. For example, in the case of ad-hoc retrieval we wish to rank a set of n candidate documents in response to a query, and in this setting each $x_i \in x$ represents a (query, document) pair. Similarly, for personalized recommendation, each $x_i \in x$ could represent a (user, item) pair when recommending a set of n items to a specific user. Furthermore let π be a total ordering of x, represented as a bijection of $\{1, 2, \ldots, n\} = [n]$ onto itself. The set of all possible n-sized total orderings, also called the symmetric group, is denoted as $S_n = \{\pi : [n] \to [n] \mid \pi \text{ bijective}\}$. We will denote with π^{-1} the inverse ordering such that $\pi^{-1}(\pi(i)) = i$. Let r be a *ranking function* that maps a set of candidate items $x \in \mathcal{X}^n$ to a total ordering $\pi \in S_n$:

$$r : \mathcal{X}^n \to S_n \tag{1}$$

Now, suppose there is a ground truth ordering $\pi^* \in \Pi^n$ for any subset of items x. Broadly speaking, the objective of LTR is to find a ranking function r that recovers the ground truth ordering π^*.

Because the set of permutations S_n is exponentially large, finding such a function r is a difficult problem to solve in general. In practice, instead of assuming a single optimal ordering π^*, it is common to assume there is a set of many optimal orderings $S_n^* \subseteq S_n$ [12]. These orderings are induced from *relevance labels*: real-valued labels that indicate the relevance of each item in x. We denote such relevance labels as $y \in \mathbb{R}^n$ and write $S_n^* = S_n^y$ as the set of optimal orderings induced by y:

$$S_n^y = \{\pi \in S_n \mid y_{\pi(i)} \geq y_{\pi(j)} \forall i < j\} \tag{2}$$

Furthermore, it is common practice to assume a score-and-sort approach. This means that, instead of finding a ranking function r,

[2]https://github.com/tensorflow/ranking
[3]https://github.com/dmlc/xgboost
[4]https://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html
[5]https://code.google.com/archive/p/sofia-ml/
[6]https://sourceforge.net/p/lemur/wiki/RankLib/
[7]http://quickrank.isti.cnr.it/
[8]https://bitbucket.org/tunystom/rankpy/src/master/
[9]https://github.com/jma127/pyltr
[10]https://github.com/yasserg/jforests
[11]https://github.com/rjagerman/pytorchltr

we are content with finding a *scoring function* f that maps a set of candidate items x to real-valued scores:

$$f : \mathcal{X}^n \to \mathbb{R}^n \tag{3}$$

The scores $s = f(x)$ can then be used to induce a total ordering π by sorting them. To this end, we define $rank(s_i \mid s)$ as the function that computes the 1-based rank of the i^{th} score s_i after sorting the scores s in descending order:

$$rank(s_i \mid s) \in \left\{\pi^{-1}(i) \mid \pi \in S_n^s\right\} \tag{4}$$

where S_n^s is defined similarly to S_n^y in Equation 2 with the difference that π is induced by scores s instead of labels y. In practice, f is often a parameterized function f_θ for which we wish to find the parameters θ. For example, f_θ could be a neural network where θ are the weights of the network. At this point, it is worth noting that an item $x_i \in \mathcal{X}$ is typically a representation of an item in some d-dimensional feature space. In other words $x_i \in \mathbb{R}^d$ and $x \in \mathbb{R}^{n \times d}$.

Given a set of items x and corresponding relevance labels y, it is now possible to define the risk (or reversely, the utility) of a scoring function f_θ by considering a ranking loss function ℓ:

$$\ell : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \tag{5}$$

Such a loss function expresses how well the ranking induced by the item scores $f_\theta(x)$ matches the ranking induced by the relevance labels y. With this in mind, LTR becomes a supervised learning problem where items and labels are sets and we wish to find a scoring function f_θ that minimizes empirical risk on a dataset $\mathcal{D}$:

$$\text{Risk}(f_\theta) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} \ell\left(f_\theta(x), y\right) \tag{6}$$

where $\mathcal{D}$ is defined as:

$$\mathcal{D} \subseteq \left\{(x, y) \mid (x, y) \in \mathcal{X}^n \times \mathbb{R}^n\right\} \tag{7}$$

Depending on the specific ℓ and f_θ, this formulation permits optimization via gradient descent and many existing LTR algorithms minimize this empirical risk estimate. In practical applications where $\mathcal{D}$ is large, a common learning strategy is that of Stochastic Gradient Descent (SGD). In SGD, the learner samples a batch of elements $\mathcal{B} \subset \mathcal{D}$ at each learning step and performs a gradient descent update on the sampled elements. The effectiveness of SGD in optimizing the empirical risk for LTR is widely documented in the literature [2, 6, 7, 25, 38, 42, 47]. As the focus of this paper is on a new library for LTR, we will focus mostly on ranking losses and metrics. As a result we will not go into detail about different possible LTR learning strategies and simply assume we wish to optimize a neural network f_θ via standard SGD.

3.2 JAX

JAX is an "extensible system for function transformations that compose: differentiation, vectorization, JIT-compilation to GPU/TPU, and more" [4]. JAX can transform python functions using various transformations such as:

- `jax.grad`: Differentiates a function with respect to its inputs.
- `jax.vmap`: Vectorizes a function by mapping it across a batch dimension of its inputs.
- `jax.jit`: Compiles a function using Accelerated Linear Algebra (XLA).

To enable these transformations, JAX is designed to operate only on Pure and Statically Composed (PSC) functions. PSC functions are functions that satisfy the following two requirements [15]:

- *Functionally pure*: the function does not read/write global state and is deterministic so that it always produces the same output given the same inputs.
- *Statically-composed* (relative to a set of primitive functions): the function should be representable as a static data dependency graph on a set of primitives (addition, multiplication, XLA operators, etc.).

Although the PSC requirement restricts the class of python functions that can be transformed with JAX, it turns out that many machine learning systems can be implemented in this paradigm. For example, a single training step of a neural network can be entirely written as a PSC function. This makes it possible to use JAX to accelerate the compute-dominated parts of the machine learning workload using XLA, while still allowing the dynamism of Python to orchestrate the overall logic of the entire system.

The design of JAX has proven to be a powerful framework for implementing machine learning systems. For example, JAX was found to be consistently faster than alternatives for differential privacy applications [43]. Furthermore, JAX has demonstrated state-of-the-art results on the MLPerf benchmark [27], in several cases outperforming other frameworks such as Tensorflow.

JAX provides the building blocks needed to build machine learning systems in the form of function transformations. However, it does not provide out-of-the-box libraries that address the needs of researchers. Because of this a large ecosystem of well-tested and actively developed JAX libraries has emerged. Several libraries offer implementations of neural networks, most notably Flax [20] and Haiku [21]. Other libraries provide optimizers that can be used to optimize parameterized functions, for example Optax [22]. Moreover several libraries for specialized domains of machine learning exist: for example RLax [3] for reinforcement learning, PIX [3] for image processing, Scenic [11] for computer vision, and, JRaph [17] for graph neural networks. Since most of these libraries can inter operate, for example one could use an optimizer from Optax to optimize the parameters of a neural network designed with Flax, it is common practice for researchers to pick and choose a subset of libraries to fulfill their research needs.

By and large, the JAX ecosystem seems to follow a Unix-like philosophy [29] of developing libraries that do a single thing well and that can easily inter operate with each other. This stands in contrast with monolithic frameworks, such as TensorFlow [1] and PyTorch [34], which combine all of this functionality in a single library. To prevent re-inventing the wheel and to match this design philosophy, our library RAX focuses solely on serving the *ranking-specific* needs of researchers. As such, RAX does not provide modeling functionality, optimizers or data pipelines as those are better served by other libraries in the JAX ecosystem. Instead, RAX specifically provides ranking *losses*, ranking *metrics* and a set of novel *function transformations* that allow direct ranking metric optimization. RAX follows the JAX convention of strictly using PSC functions. This allows RAX to inter operate with nearly all libraries in the JAX ecosystem.

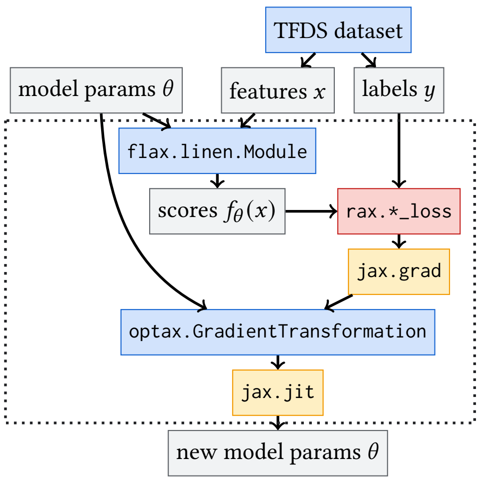

Figure 1: A single training step as a composition of functions from different libraries in the JAX ecosystem. Yellow nodes indicate core JAX functionality, blue nodes indicate other JAX libraries and the red node is the Rax ranking loss. The dotted line indicates the PSC function boundary.

4 DESIGN OF RAX

The design of Rax is broken down into three main components: ranking losses, ranking metrics and function transformations. The API of Rax is a flat collection of functions where the naming convention indicates the type of function:

- `rax.*_loss`: Losses such as `rax.pairwise_hinge_loss`.
- `rax.*_metric`: Metrics such as `rax.ndcg_metric`.
- `rax.*_t12n`: Transformations such as `rax.approx_t12n`.

An overview of how Rax interacts with JAX and other libraries in the JAX ecosystem is illustrated in Figure 1. This figure displays how functions from different libraries are composed into a single training step of a neural network. At the center of this composition is the Rax training loss that we wish to optimize. Notice how the entire training step function (as outlined in the dotted line) is represented as a *single* PSC function that is compiled using `jax.jit`. The resulting training step benefits from optimizations offered by the XLA compiler including fusion of many different parts of the training step. This means the training step, written entirely in Python, is capable of efficiently running on an accelerated device such as a TPU or GPU without any extra implementation effort. A more complete code example that demonstrates how Rax can be used to train a ranking model is provided in Appendix A.

4.1 A Unified Ranking Loss Design

As introduced in Section 3.1, ranking losses are functions that express, for a given set of candidate items, how well the ranking induced by the predicted item scores $s = f_\theta(x)$ match the ranking induced by the corresponding relevance labels y. Ranking losses are commonly categorized into three categories: pointwise losses, pairwise losses and listwise losses [28]. The loss function signature across these three categories are different because each loss function operates at a different level of abstraction: pointwise losses are defined on a single item x_i, pairwise losses on a pair of items (x_i, x_j), and, listwise losses on a list of items $[x_1, x_2, \ldots, x_n]$:

$$\ell_{\text{pointwise}} : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \tag{8a}$$
$$\ell_{\text{pairwise}} : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} \tag{8b}$$
$$\ell_{\text{listwise}} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \tag{8c}$$

The fact that these loss function signatures are different is problematic from an engineering perspective. Users would have to carefully craft their input to match the chosen loss function. As a result, users would not be able to easily change to a different ranking loss without changing the call signature.

To resolve this problem, Rax implements the ranking losses using a unified design that encodes all three categories of ranking losses. To do so, Rax adopts the most general formulation, the listwise ranking loss, as the unified ranking loss signature. We can express the pointwise and pairwise losses using this unified signature by changing the output of the loss to be a tensor instead of a scalar. This leads to the following function signatures:

$$\ell_{\text{pointwise}} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n \tag{9a}$$
$$\ell_{\text{pairwise}} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^{n^2} \tag{9b}$$
$$\ell_{\text{listwise}} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \tag{9c}$$

A different perspective on this is that, given a list of items x and relevance labels y, the pointwise losses treat such a list as a batch of n samples, the pairwise losses treat it as a batch of (at most) n^2 samples and the listwise losses treat it as a single sample. So far, we have assumed the ranking loss is applied to a single list, but in practice it is more common to compute such losses on a batch $\mathcal{B}$ containing $b = |\mathcal{B}|$ lists. This changes the signature of the losses to:

$$\ell_{\text{pointwise}} : \mathbb{R}^{b \times n} \times \mathbb{R}^{b \times n} \to \mathbb{R}^{b \times n} \tag{10a}$$
$$\ell_{\text{pairwise}} : \mathbb{R}^{b \times n} \times \mathbb{R}^{b \times n} \to \mathbb{R}^{b \times n^2} \tag{10b}$$
$$\ell_{\text{listwise}} : \mathbb{R}^{b \times n} \times \mathbb{R}^{b \times n} \to \mathbb{R}^{b} \tag{10c}$$

We note that this formulation of the losses is not immediately useful for optimization. The output of each loss needs to be reduced to a scalar value in order to use the loss for optimization.

4.1.1 Batch Reduction of Ranking Losses. Common strategies for reducing a batch of losses to a scalar value include mean reduction and sum reduction. More generally we can denote with $A : \mathbb{R}^d \to \mathbb{R}$ the space of all possible aggregation functions. For example sum-reduction $a_{\text{sum}} \in A$ can be defined as follows:

$$a_{\text{sum}}(z) = \sum_{z_i \in z} z_i \tag{11}$$

With this formulation, the losses can be reduced to a scalar value by expressing them as a composition of ℓ with an aggregation function $a \in A$. For example:

$$\ell = \ell_{\text{pointwise}} \circ a \tag{12}$$

More generally, we can now write *any* ranking loss with the following unified signature:

$$\ell : \mathbb{R}^{b \times n} \times \mathbb{R}^{b \times n} \times A \to \mathbb{R}\,. \tag{13}$$

Example usage[12] of several Rax ranking losses, including the pointwise MSE loss [28], pairwise hinge loss [28] and Softmax loss [33] are shown in Listing 1.

Listing 1: Pointwise, pairwise and listwise ranking losses.

```
rax.pointwise_mse_loss(s, y)
rax.pairwise_hinge_loss(s, y, reduce_fn=jax.numpy.sum)
rax.softmax_loss(s, y, reduce_fn=jax.numpy.mean)
```

It is worth noting that this design of ranking losses in Rax is a form of inversion of control [14]. The ranking loss gives the user *control* over the desired reduction behavior by allowing them to inject a custom `reduce_fn`. This is different than libraries such as TensorFlow or PyTorch, where one would have to supply a configuration constant that indicates the type of loss reduction. Such configuration constants can make it difficult to customize the reduction behavior to more advanced use-cases. By accepting a `reduce_fn` keyword argument in Rax, users have full control over the desired reduction behavior. Note that this does not make typical reduction scenarios such as sum or mean reduction any more complicated, as one can simply pass in standard JAX functions such as `jax.numpy.sum` or `jax.numpy.mean`.

4.1.2 *Distributed Reduction of Ranking Losses.*

A major benefit of the approach described so far is that it enables more advanced cases where custom reduction logic is needed. For example, in distributed machine learning, it may be necessary to reduce loss values across devices before performing a parameter update step. Since JAX provides several parallel computing primitives, it becomes extremely easy to use a Rax loss in a distributed learning setting. An example of a distributed sum aggregation is provided in Listing 2.

Listing 2: Distributed loss reduction in Rax. This assumes there is a named batch axis called "device"

```
def psum(a, where):
  if where is not None:
    a = jax.numpy.where(where, a, 0.)
  return jax.lax.psum(a, axis_name="device")

rax.pairwise_hinge_loss(scores, labels, reduce_fn=psum)
```

Note that this code is significantly simpler compared to parallel training primitives that exist in other frameworks such as TensorFlow or PyTorch. The design of JAX, and by proxy the design of the ranking losses in Rax, obviates the need for strategy scopes, custom distributed optimizers and other complex abstractions because the parallel primitives are transparent to the user and can be customized effortlessly.

Finally, all the ranking losses are implemented as PSC functions. This makes computing the gradient of a loss a trivial application of a `jax.grad` transformation as shown in Listing 3.

Listing 3: Computing the gradient of a ranking loss.

```
grad_fn = jax.grad(rax.pairwise_hinge_loss)
grad_fn(scores, labels)
```

[12] In the actual implementation of Rax the loss also accepts an optional boolean tensor called `where`. This is needed to model batches with a ragged structure via padding.

4.2 Ranking Metrics

Ranking metrics are an important component for evaluating IR systems. There are a number of standard ranking metrics that are commonly used in practice to measure how well a ranker performs on a particular task. Most ranking metrics are designed to reward correctness at the top of the ranked list more than at the bottom. Recall that $rank(s_i \mid s)$ indicates the rank of the i^{th} score s_i after sorting the scores s in descending order. We can use this to define several common ranking metrics via aggregation over the ranks, for example:

$$DCG(s, y) = \sum_{i=1}^{n} \frac{\text{gain}(y_i)}{\text{discount}(rank(s_i \mid s))} \tag{14}$$

$$NDCG(s, y) = DCG(s, y)/DCG(y, y) \tag{15}$$

where gain$(\cdot)$ and discount$(\cdot)$ are functions that map relevance labels to gains and ranks to discounts respectively. Common choices are $\text{gain}(y_i) = 2^{y_i} - 1$ and $\text{discount}(r) = \log_2(r + 1)$. Furthermore, some metrics accept a cutoff value k that indicates the rank at which results are cut off. We denote such a cutoff function as follows:

$$cutoff(s_i \mid s, k) = \mathbb{1}[rank(s_i \mid s) \le k] \tag{16}$$

In other words $cutoff(s_i \mid s, k)$ is 1 if the i^{th} score is one of the k largest scores of s, and 0 otherwise. Some examples of cutoff-based ranking metrics are:

$$NDCG@k(s, y) = \frac{1}{DCG@k(y, y)} \sum_{i=1}^{n} \frac{cutoff(s_i \mid s, k) \cdot \text{gain}(y_i)}{\text{discount}(rank(s_i \mid s)} \tag{17}$$

$$Recall@k(s, y) = \frac{\sum_{i=1}^{n} cutoff(s_i \mid s, k)\, \mathbb{1}[y_i > 0]}{\sum_{i=1}^{n} \mathbb{1}[y_i > 0]} \tag{18}$$

$$Prec@k(s, y) = \frac{\sum_{i=1}^{n} cutoff(s_i \mid s, k)\, \mathbb{1}[y_i > 0]}{\sum_{i=1}^{n} cutoff(s_i \mid s, k)} \tag{19}$$

$$AP(s, y) = \frac{\sum_{i=1}^{n} Prec@rank(s_i \mid s)(s, y) \cdot \mathbb{1}[y_i > 0]}{\sum_{i=1}^{n} \mathbb{1}[y_i > 0]} \tag{20}$$

Rax provides implementations for a number of standard ranking metrics. The function signature of a ranking metric function m largely follows the same signature as that of the loss functions:

$$m : \mathbb{R}^{b \times n} \times \mathbb{R}^{b \times n} \times A \longrightarrow \mathbb{R}\,. \tag{21}$$

An example is given in Listing 4. Note how the keyword argument `topn=20` is used for some metrics to indicate the cutoff rank k.

Listing 4: Example ranking metrics in Rax.

```
rax.ndcg_metric(s, y)
rax.mrr_metric(s, y)
rax.precision_metric(s, y, topn=20)
rax.recall_metric(s, y, topn=20)
```

A key distinction that sets ranking metrics apart from ranking losses is their resistance to direct optimization. All ranking metrics are rank-based, meaning they need to compute $rank(s_i \mid s)$ to obtain 1-based ranks. Although the gradient of the *rank* operation is well-defined, it is fairly useless as it is zero everywhere. As a result, direct optimization of ranking metrics is considered a difficult problem. In the next section, we will describe how the ranking metric implementations in Rax can be *transformed* to allow approximate metric optimization.

4.3 Differentiable Metrics via Function Transformations

As described in Section 4.2, ranking metrics are difficult to optimize due to the *rank* operation introducing zero-gradients. Existing work has looked at finding approximations of ranks that have a useful non-zero gradients. Examples of methods utilizing this approach are SoftRank [44], SmoothDCG [46], ApproxNDCG [37] and NeuralSort [18, 35]. The general idea is that the rank function $rank(s_i \mid s)$ can be replaced with an approximation that has non-zero gradients. ApproxNDCG [37] constructs such an approximation by writing the rank function as a sum of indicator functions and then replacing the indicator with a sigmoid:

$$rank(s_i \mid s) = 1 + \sum_{j \neq i} \mathbb{1}[s_j > s_i] \tag{22a}$$

$$\approx 1 + \sum_{j \neq i} \text{sigmoid}(s_j - s_i) = \overline{rank}(s_i \mid s) \tag{22b}$$

Existing libraries such as TF-Ranking have offered such approximations as separate ranking loss implementations. However, there are two problems with this approach: First, it is not easy to extend the general idea of approximate metric optimization to other ranking metrics without having to write entirely new implementations. Second, the existing work on approximate metric optimization has mostly focussed on approximating metrics such as NDCG, but has neglected work on metrics that necessarily require a cutoff value such as NDCG@k or Recall@k.

4.3.1 Approximate Metrics. In Rax we solve both issues simultaneously by allowing users to provide their own *rank* and *cutoff* functions. In effect, this allows users to replace both *rank* and *cutoff* with differentiable approximations that can be optimized. The way this is accomplished is by generalizing the function signature of the metrics from their standard form to a more generic one:

$$m : \mathbb{R}^{b \times n} \times \mathbb{R}^{b \times n} \times A \times R \times C \to \mathbb{R}. \tag{23}$$

where R is the space of all possible rank functions and C is the space of all possible cutoff functions. In other words, users can freely inject their own rank and cutoff functions to customize the ranking metrics. For example, by supplying the approximate *rank* function defined in Equation 22b we can obtain the standard ApproxNDCG loss. Note that Rax provides a `rax.approx_t12n` transformation for ease-of-use that automatically injects sigmoid-based approximations as demonstrated in Listing 5.

Listing 5: Transforming a ranking metric to an approximate differentiable loss.

```
loss = -rax.ndcg_metric(s, y, rank_fn=rax.approx_ranks)
# is equivalent to:
loss = rax.approx_t12n(rax.ndcg_metric)(s, y)
```

4.3.2 Approximations for Cutoff-based Metrics. In addition to supporting custom *rank* functions, Rax metrics also accept custom *cutoff* functions. For example, we can define an approximate cutoff function as follows:

$$cutoff(s_i \mid s) = \mathbb{1}[rank(s_i \mid s) \leq k] \tag{24a}$$

$$\approx \text{sigmoid}\left(\overline{rank}(s_i \mid s) - k\right) = \overline{cutoff}(s_i \mid s) \tag{24b}$$

This can be used to construct approximations for cutoff-based metrics. Consider the definition of NDCG@k in Equation 17. By plugging in an approximations for both *cutoff* and *rank* we obtain an approximate version of this metric that has non-zero gradients:

$$ApproxNDCG@k(s, y) = \frac{1}{DCG@k(y, y)} \sum_{i=1}^{n} \frac{\overline{cutoff}(s_i \mid s, k) \cdot \text{gain}(y_i)}{\text{discount}(\overline{rank}(s_i \mid s)} \tag{25}$$

The pluggable *rank* and *cutoff* functions make it possible to construct approximate forms of our entire offering of ranking metrics. In fact, any future metric that follows this same signature, can be used for approximate metric optimization. This permits re-use of all the ranking metrics in Rax in an optimization context.

4.3.3 Bounded Metrics. Another advantage of accepting custom *rank* and *cutoff* functions in the Rax ranking metrics is that we are not restricted to only sigmoid-based approximations. For example, existing work [2] has has explored upper-bounding the ranks instead of approximating them which in turn constructs a lower-bound for some metrics. As long as the upper-bound on the ranks is differentiable, this makes the lower-bound for the metric suitable for optimization. A straightforward differentiable upper-bound on the ranks is the hinge function:

$$rank(s_i \mid s) = 1 + \sum_{j \neq i} \mathbb{1}[s_j > s_i] \leq 1 + \sum_{j \neq i} max(0, 1 + s_j - s_i) \tag{26}$$

The design of Rax permits plugging such bounds directly into our collection of ranking metrics, as demonstrated in Listing 6. Note that, depending on the specific metric, in order to obtain a *lower* bound on the metric, we need to either supply a lower or upper bound for either the ranks or cutoff functions. For simplicity, Rax offers a `rax.bound_t12n` function that supplies upper bounds for ranks and lower bounds for cutoffs automatically.

Listing 6: Differentiable lower bound of NDCG.

```
upperbound_rank_fn = functools.partial(
  rax.approx_ranks,
  step_fn=lambda x: jax.nn.relu(1. + x))
loss = -rax.ndcg_metric(s, y, rank_fn=upperbound_rank_fn)
# is equivalent to:
loss = rax.bound_t12n(rax.ndcg_metric)(s, y)
```

4.3.4 Gumbel Sampling. Finally, recent work has shown that in order to effectively optimize an approximate metric loss, it is important to perform Gumbel-sampling on the scores [5], which is a form of stochastic estimation of the loss [48]. To support this, Rax provides a function transformation called `gumbel_t12n`. This function transformation replicates the scores a number of times, and adds gumbel noise to them. Such a transformation can be applied to any ranking loss, including the losses for approximate or bounded metrics. See Listing 7 for an example that combines the gumbel transformation with an approximate metric transformation.

Listing 7: Creating a GumbelApproxNDCG [5] loss as a composition of Gumbel and Approx transformations.

```
loss_fn = rax.gumbel_t12n(rax.approx_t12n(rax.ndcg_metric))
loss = loss_fn(s, y, key=jax.random.PRNGKey(0))
```

It is important to note here that this makes the loss stochastic, due to the sampling of random gumbel noise. As such, the returned loss function requires a new argument `key` which acts as the random state on which the random operations such as gumbel noise are based. To ensure the Rax functionality strictly adheres to the PSC function requirements, a global random state is prohibited, which means the random state needs to be passed as an argument. This is fairly standard behavior for JAX functions, for example all the functions in `jax.random` require a `key` argument.

Overall, the flexibility offered by Rax opens up new ways of performing ranking metric optimization. For example, Rax makes it possible to compute approximations and bounds for metrics that are traditionally considered hard to optimize such as *NDCG@K* and *Recall@K*. All of this is accomplished without introducing code duplication by carefully constructing a generic metric function signature and then transforming the functions using `rax.approx_t12n`, `rax.bound_t12n`, and, `rax.gumbel_t12n`. Moreover, this approach scales to future metric implementations as long as they adhere to the metric function definition of Equation 23.

5 EXPERIMENTS

To validate the effectiveness of our library Rax we have conducted several experiments on two large-scale LTR benchmark datasets *WEB30K* [36] and *Istella* [10]. Furthermore we have performed experiments for finetuning a passage ranking task on *MS MARCO* [30] using a Rax ranking loss with T5X[13]: a JAX implementation of the T5 [39] large model.

5.1 LTR Benchmarks

For the LTR benchmarks on the *WEB30K* and *Istella* datasets we use a standard supervised LTR setup. Both datasets are comprised of a collection of queries and corresponding documents to be ranked. Each query-document pair is represented as a d-dimensional feature vector, where $d = 136$ for *WEB30K* and $d = 220$ for *Istella*. Furthermore, each query-document pair has a corresponding relevance label $y \in \{0, 1, 2, 3, 4\}$, where a higher value indicates a higher relevance of the document. Since the *Istella* dataset has no validation set we split the training set into 90% training and 10% validation manually. For *WEB30K* we use Fold 1 to conduct our experiments.

We train a neural network with the various ranking losses offered by Rax. The network architecture, implemented in Flax [20] and optimized with Optax [3], is a feedforward neural network with hidden layers of size $[1024, 512, 256]$, where ReLU activations, batch normalization and dropout is used at each hidden layer. A hyperparameter sweep is performed, where the best run is selected by evaluating it on held-out validation data. For the losses that optimize a specific metric (e.g. ApproxAP, ApproxRecall@20, etc.) we choose the best run in terms of the metric being optimized on the validation set. For other losses we select the best run by its NDCG on the validation set. For the optimizer we try both Adam [26] with a learning rate $\in \{0.0001, 0.0003, 0.001, 0.003, 0.01, 0.03\}$ and Adagrad [13] with a learning rate $\in \{0.01, 0.03, 0.1, 0.3\}$. We tune the batchnorm momentum $\in \{0.9, 0.99, 0.999\}$ and, for the approx losses, additionally tune the temperature parameter $\in \{1, 10, 100\}$ for the sigmoid function. Each network is trained for 100,000 steps where each step uses a batch size of 128. The entire training procedure runs on a TPU and a single training run finishes in less than 3 hours. We evaluate the ranking models using NDCG, NDCG@10, Average Precision (AP) and Recall@20. For the AP and Recall@20 metrics we convert graded relevance to binary relevance by considering items with $y \in \{0, 1, 2\}$ as not relevant and items with $y \in \{3, 4\}$ as relevant.

[13]https://github.com/google-research/t5x

5.2 *MS MARCO* Passage Ranking

We also conduct experiments on the *MS MARCO* passage ranking dataset. The dataset contains a corpus of more than 8.8 million passages and questions with binary labels on relevant passages. The task is to rank the passages for each question based on their relevance. There are more than 530,000 questions in the "train" partition, and the evaluation is usually conducted on the "dev" partition of around 6,800 questions.

In our experiments, we first use a dual-encoder retriever [31] to retrieve the top-1000 passages for each question. Then we concatenate the question with each candidate passage in a similar setting to [32] and feed the concatenated strings into a T5 [40] model. We use the T5X implementation in JAX and initialize the model with T5-large checkpoint. We fine-tune the T5 ranker with multiple ranking losses implemented in Rax. Due to memory limitation, we sample 36 passages from the retrieved passages of each question as a list during fine-tuning. As a baseline, we also fine-tune the T5 ranker with the pointwise cross-entropy loss on a data set with a balanced number of relevant and irrelevant question-passage pairs.

We evaluate the performance of our rankers on the "dev" partition of the *MS MARCO* dataset. The inference is conducted on all the retrieved passages and we use MRR@10 as the evaluation metric which is common practice for this dataset.

6 RESULTS

6.1 LTR Benchmarks

First, let us look at the results for the *WEB30K* dataset as indicated by the first 4 columns in Table 1. We note that, again, the Softmax loss is a strong baseline but does not outperform all other losses. On this dataset we find that ranking metric optimization has several benefits. For example the Rax implementation of ApproxAP strongly outperforms all other methods on the AP metric. Similarly, the BoundRecall@20 achieves the highest Recall@20. Finally, we find that NDCG and NDCG@10 are strongly correlated. As a result, methods that optimize for either one generally achieve a high score on both metrics. We note that BoundNDCG@10, a novel loss offered by Rax, achieves the highest NDCG on this dataset.

Second, we direct our attention to the *Istella* dataset as indicated by the latter 4 columns in Table 1. We observe that the Rax implementation of Softmax is very strong on this dataset, and in fact it outperforms the TF-Ranking implementation of Softmax on nearly all metrics. We note that the TF-Ranking version of ApproxNDCG performs strong on the NDCG metric. Upon further inspection

Table 1: LTR benchmark results on the test split of *WEB30K* and *Istella*. Bold numbers indicate the best result for each metric and ▲ indicates significantly ($p < 0.05$, t-test with Benferronni correction) better performance than the TF-Ranking Softmax baseline. All the Approx* and Bound* losses use Gumbel-sampling with 8 Gumbel samples.

	WEB30K				*Istella*			
	NDCG	NDCG@10	AP	Recall@20	NDCG	NDCG@10	AP	Recall@20
TF-Ranking Softmax	71.36	48.71	20.78	36.66	80.90	71.96	62.18	90.71
TF-Ranking ApproxNDCG	71.13	48.28	20.96	35.84	**81.12**	71.92	62.25	89.73
Rax Softmax	71.30	48.55	20.76	36.53	80.99	**72.11**	**62.40**	**90.82**
Rax Pointwise MSE	71.28	48.68	20.54	36.42	80.06	70.68	60.65	89.19
Rax Pairwise Logistic	70.53	47.45	18.53	34.88	80.39	71.15	61.19	90.59
Rax ApproxNDCG	71.08	48.14	20.29	35.10	79.88	70.06	60.33	88.70
Rax ApproxNDCG@10	71.31	48.57	20.53	35.29	79.91	70.21	60.46	88.88
Rax ApproxAP	68.89	44.18	**21.38**▲	36.70	79.56	69.67	60.27	88.29
Rax ApproxRecall@20	69.09	44.56	20.99	**36.90**	79.84	70.23	60.62	89.59
Rax BoundNDCG	70.96	47.96	20.42	35.02	80.69	71.30	61.67	89.01
Rax BoundNDCG@10	**71.40**	**48.87**	20.98	36.68	80.61	71.36	61.64	90.52
Rax BoundAP	69.20	44.51	20.79	35.41	79.83	70.02	60.77	87.80
Rax BoundRecall@20	68.16	42.98	21.02	36.80	78.72	68.59	58.99	88.93

Table 2: Performance of T5-large ranker on *MS MARCO* trained with different ranking losses. The best performance is bolded. The Rax ApproxNDCG@10 loss uses Gumbel-sampling with 8 Gumbel samples.

Ranking loss	Dev MRR@10
BERT TF-Ranking Ensemble [19]	42.13
T5-Large + Rax Pointwise Sigmoid	41.84
T5-Large + Rax Pairwise Logistic	41.79
T5-Large + Rax ApproxNDCG@10	41.62
T5-Large + Rax Softmax	**42.74**

of this loss we find that the Gumbel sampling procedure used in TF-Ranking applies an extra log-softmax transformation on the sampled scores, which the Rax version does not do and may explain the differences in performance. Generally, we find that ranking metric optimization does not perform very well on this dataset and is not able to outperform the Softmax loss. We hypothesize that this is due to the fact that metric-based losses are typically non-convex. This non-convexity may cause ranking models to get stuck in local optima, which seems more problematic on the *Istella* dataset.

Overall, our results show that Rax is able to compete with TF-Ranking, a state-of-the-art LTR library. Our results suggest that there is not a clear single superior loss. Generally the Softmax loss performs strong across all metrics. In some cases, depending on the metric and dataset, we find that ranking metric optimization can be beneficial. However, the exact properties of various ranking metric optimizations are not well understood and our results suggest that further study should be conducted, which we leave as future work.

6.2 *MS MARCO* Passage Ranking

The results of the *MS MARCO* Passage Ranking task are displayed in Table 2. We find that the standard pointwise sigmoid cross-entropy loss is a strong baseline for this task. Interestingly, the ApproxNDCG loss, which was a strong baseline on *WEB30K* and *Istella* performs less well on this dataset. One possible reason is the extreme sparseness of *MS MARCO*, where there is effectively a single relevant result for each list. Finally, we find that Softmax performs best, achieving a high MRR@10. The number is slightly higher than the ensemble of multiple BERT-based rankers also fine-tuned with different ranking losses reported in [19]. However, other factors such as different pre-trained models or different retrieval results may also contribute to the improvement.

7 CONCLUSION

In this paper we introduced Rax – the first Learning-to-Rank (LTR) library in the JAX ecosystem. The library provides implementations for a number of standard ranking losses and ranking metrics. Furthermore, Rax provides a set of novel function transformations that make it possible to re-purpose the ranking metrics as differentiable ranking losses by injecting approximations and/or bounds to rank and cutoff functions. Unlike existing libraries, this allows for approximate metric optimization on our entire offering of ranking metrics in a systematic way. The design of Rax makes it easy to explore new approximations and bounds for approximate metric optimization, as well as their Gumbel versions, which opens up new possibilities for research.

There are several directions for future work. First, Rax provides a number of ranking losses and metrics but is by no means exhaustive. Our initial offering of ranking losses can be expanded and we encourage the open source community to collaborate with us to do so. Second, the design of Rax opens up new ways of performing ranking metric optimization, including new ways to re-purpose metrics as losses via differentiable rank and cutoff functions. We leave studying the exact properties of various approximations and bounds of those functions as future work.

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 ({OSDI} 16)*. 265–283.

[2] Aman Agarwal, Kenta Takatsu, Ivan Zaitsev, and Thorsten Joachims. 2019. A general framework for counterfactual learning-to-rank. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 5–14.

[3] Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, et al. 2020. *The DeepMind JAX Ecosystem*. http://github.com/deepmind

[4] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

[5] Sebastian Bruch, Shuguang Han, Michael Bendersky, and Marc Najork. 2020. A stochastic treatment of learning to rank scoring functions. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 61–69.

[6] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*. 89–96.

[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 *Proceedings of the 24th international conference on Machine learning*. 129–136.

[8] Gabriele Capannini, Domenico Dato, Claudio Lucchese, Monica Mori, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, and Nicola Tonellotto. 2015. QuickRank: a C++ Suite of Learning to Rank Algorithms. In *IIR*.

[9] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[10] Domenico Dato, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2016. Fast ranking with additive ensembles of oblivious and non-oblivious regression trees. *ACM Transactions on Information Systems (TOIS)* 35, 2 (2016), 1–31.

[11] Mostafa Dehghani, Alexey Gritsenko, Anurag Arnab, Matthias Minderer, and Yi Tay. 2021. Scenic: A JAX Library for Computer Vision Research and Beyond. *arXiv preprint arXiv:2110.11403* (2021).

[12] Fernando Diaz, Bhaskar Mitra, Michael D Ekstrand, Asia J Biega, and Ben Carterette. 2020. Evaluating stochastic rankings with expected exposure. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 275–284.

[13] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research* 12, 7 (2011).

[14] Mohamed Fayad and Douglas C Schmidt. 1997. Object-oriented application frameworks. *Commun. ACM* 40, 10 (1997), 32–38.

[15] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* (2018).

[16] Yasser Ganjisaffar, Rich Caruana, and Cristina Lopes. 2011. Bagging Gradient-Boosted Trees for High Precision, Low Variance Ranking Models. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information (SIGIR '11)*. 85–94.

[17] Jonathan Godwin*, Thomas Keck*, Peter Battaglia, Victor Bapst, Thomas Kipf, Yujia Li, Kimberly Stachenfeld, Petar Veličković, and Alvaro Sanchez-Gonzalez. 2020. *Jraph: A library for graph neural networks in jax*. http://github.com/deepmind/jraph

[18] Aditya Grover, Eric Wang, Aaron Zweig, and Stefano Ermon. 2019. Stochastic optimization of sorting networks via continuous relaxations. *arXiv preprint arXiv:1903.08850* (2019).

[19] Shuguang Han, Xuanhui Wang, Mike Bendersky, and Marc Najork. 2020. Learning-to-rank with bert in tf-ranking. *arXiv preprint arXiv:2004.08476* (2020).

[20] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. 2020. *Flax: A neural network library and ecosystem for JAX*. http://github.com/google/flax

[21] Tom Hennigan, Trevor Cai, Tamara Norman, and Igor Babuschkin. 2020. *Haiku: Sonnet for JAX*. http://github.com/deepmind/dm-haiku

[22] Matteo Hessel, David Budden, Fabio Viola, Mihaela Rosca, Eren Sezener, and Tom Hennigan. 2020. *Optax: composable gradient transformation and optimisation, in JAX!* http://github.com/deepmind/optax

[23] Rolf Jagerman and Maarten de Rijke. 2020. Accelerated Convergence for Counterfactual Learning to Rank. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. Association for Computing Machinery, New York, NY, USA.

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

[25] Thorsten Joachims. 2002. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 133–142.

[26] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[27] Sameer Kumar, Yu Wang, Cliff Young, James Bradbury, Naveen Kumar, Dehao Chen, and Andy Swing. 2021. Exploring the limits of Concurrency in ML Training on Google TPUs. *Proceedings of Machine Learning and Systems* 3 (2021), 81–92.

[28] Tie-Yan Liu. 2011. Learning to rank for information retrieval. (2011).

[29] M McIlroy, EN Pinson, and BA Tague. 1978. UNIX time-sharing system. *The Bell system technical journal* 57, 6 (1978), 1899–1904.

[30] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. 2016. MS MARCO: A human generated machine reading comprehension dataset. In *CoCo@ NIPS*.

[31] Jianmo Ni, Chen Qu, Jing Lu, Zhuyun Dai, Gustavo Hernández Ábrego, Ji Ma, Vincent Y Zhao, Yi Luan, Keith B Hall, Ming-Wei Chang, et al. 2021. Large Dual Encoders Are Generalizable Retrievers. *arXiv preprint arXiv:2112.07899* (2021).

[32] Rodrigo Nogueira, Zhiying Jiang, Ronak Pradeep, and Jimmy Lin. 2020. Document Ranking with a Pretrained Sequence-to-Sequence Model. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 708–718.

[33] Rama Kumar Pasumarthi, Sebastian Bruch, Xuanhui Wang, Cheng Li, Michael Bendersky, Marc Najork, Jan Pfeifer, Nadav Golbandi, Rohan Anil, and Stephan Wolf. 2019. Tf-ranking: Scalable tensorflow library for learning-to-rank. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2970–2978.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[35] Przemysław Pobrotyn and Radosław Białobrzeski. 2021. NeuralNDCG: Direct Optimisation of a Ranking Metric via Differentiable Relaxation of Sorting. *arXiv preprint arXiv:2102.07831* (2021).

[36] Tao Qin and Tie-Yan Liu. 2013. Introducing LETOR 4.0 Datasets. *CoRR* abs/1306.2597 (2013). http://arxiv.org/abs/1306.2597

[37] 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.

[38] Zhen Qin, Le Yan, Honglei Zhuang, Yi Tay, Rama Kumar Pasumarthi, Xuanhui Wang, Michael Bendersky, and Marc Najork. 2021. Are Neural Rankers still Outperformed by Gradient Boosted Decision Trees?. In *International Conference on Learning Representations*.

[39] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[40] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[41] Samuel Schoenholz and Ekin Dogus Cubuk. 2020. Jax md: a framework for differentiable physics. *Advances in Neural Information Processing Systems* 33 (2020), 11428–11441.

[42] D Sculley. 2009. Large scale learning to rank. In *NIPS 2009 Workshop on Advances in Ranking*.

[43] Pranav Subramani, Nicholas Vadivelu, and Gautam Kamath. 2021. Enabling fast differentially private sgd via just-in-time compilation and vectorization. *Advances in Neural Information Processing Systems* 34 (2021).

[44] Michael Taylor, John Guiver, Stephen Robertson, and Tom Minka. 2008. Softrank: optimizing non-smooth rank metrics. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*. 77–86.

[45] Aleksei Ustimenko and Liudmila Prokhorenkova. 2020. Stochasticrank: Global optimization of scale-free discrete functions. In *International Conference on Machine Learning*. PMLR, 9669–9679.

[46] Mingrui Wu, Yi Chang, Zhaohui Zheng, and Hongyuan Zha. 2009. Smoothing DCG for learning to rank: A novel approach using smoothed hinge functions. In *Proceedings of the 18th ACM conference on Information and knowledge management*. 1923–1926.

[47] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. 2008. Listwise approach to learning to rank: theory and algorithm. In *Proceedings of the 25th international conference on Machine learning*. 1192–1199.

[48] Yisong Yue and C Burges. 2007. On using simultaneous perturbation stochastic approximation for learning to rank, and the empirical optimality of LambdaRank. *Microsoft Res., Redmond, WA, USA, Tech. Rep. MST-TR-2007-115* (2007).

A EXAMPLE WORKFLOW

This appendix provides a reproducible example of using a Rax ranking loss to optimize a linear ranking model on a synthetic LTR dataset [14]. The code in Listing 8 uses a linear model as indicated by the `score` function definition in line 22. The model is essentially defined as a dot-product between the `weights` (representing the model we wish to optimize) and `x` (representing the features of lists of items). The training step is implemented as a single PSC function as defined on line 27. This training step does a few things: First, it defines a loss function on line 28, which is expressed as a mapping of weights to a loss value. Second, it uses this loss function to compute gradients with respect to the weights on line 29. Third, a single gradient descent step is performed on line 30. The training step is repeatedly called in a training loop on line 37. Prior to calling the training step we first evaluate the model on NDCG on lines 39 and 40. Given that this toy dataset is a synthetic example, it is trivial to reach the optimal NDCG quickly. From the output of this code we can see that it reaches the optimal NDCG in just three iterations. This code demonstrates how easy it is to use a Rax ranking loss and metric to train and evaluate an LTR model. Furthermore, although this toy problem is very small and has no performance bottlenecks, the training step is compiled using `jax.jit` which means it can run on an accelerated device such as a TPU without requiring any extra implementation effort.

[14]This dataset is the example train dataset from SVM-rank [25] obtained from https://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html

Listing 8: Workflow of training a linear ranking model using Rax and JAX.

```
1  import jax
2  import rax
3
4  # Synthetic LTR data.
5  x = jax.numpy.array([[[1., 1., 0., 0.2, 0.],
6                        [0., 0., 1., 0.1, 1.],
7                        [0., 1., 0., 0.4, 0.],
8                        [0., 0., 1., 0.3, 0.]],
9                       [[0., 0., 1., 0.2, 0.],
10                       [1., 0., 1., 0.4, 0.],
11                       [0., 0., 1., 0.1, 0.],
12                       [0., 0., 1., 0.2, 0.]],
13                      [[0., 0., 1., 0.1, 0.],
14                       [1., 1., 0., 0.3, 0.],
15                       [1., 0., 0., 0.4, 1.],
16                       [0., 1., 1., 0.5, 0.]]])
17 y = jax.numpy.array([[2., 1., 0., 0.],
18                      [0., 1., 0., 0.],
19                      [1., 2., 3., 0.],])
20
21 # Linear model scoring function
22 def score(weights, x):
23   return jax.numpy.dot(x, weights)
24
25 # Define training step as a single PSC function.
26 @jax.jit
27 def train_step(weights, x, y):
28   loss_fn = lambda w: rax.softmax_loss(score(w, x), y)
29   grads = jax.grad(loss_fn)(weights)
30   weights -= 0.1 * grads
31   return weights
32
33 # Initialize model weights to 0.
34 weights = jax.numpy.zeros((5,))
35
36 # Perform three gradient descent iterations.
37 for i in range(3):
38   # Print NDCG for current model weights.
39   ndcg = rax.ndcg_metric(score(weights, x), y)
40   print(f"NDCG: {ndcg:.4f}")
41
42   # Perform gradient descent step.
43   weights = train_step(weights, x, y)
44
45 # Prints:
46 # NDCG: 0.7705
47 # NDCG: 0.9880
48 # NDCG: 1.0000
```