Tevatron: An Efficient and Flexible Toolkit for Neural Retrieval

Luyu Gao∗ Carnegie Mellon University Pittsburgh, United States luyug@cs.cmu.edu
Xueguang Ma University of Waterloo Waterloo, Canada x93ma@uwaterloo.ca
Jimmy Lin University of Waterloo Waterloo, Canada jimmylin@uwaterloo.ca
Jamie Callan Carnegie Mellon University Pittsburgh, United States callan@cs.cmu.edu

ABSTRACT

Recent rapid advances in deep pre-trained language models and the introduction of large datasets have powered research in embedding-based neural retrieval. While many excellent research papers have emerged, most of them come with their own implementations, which are typically optimized for some particular research goals instead of efficiency or code organization. In this paper, we introduce Tevatron, a neural retrieval toolkit that is optimized for efficiency, flexibility, and code simplicity. Tevatron enables model training and evaluation for a variety of ranking components such as dense retrievers, sparse retrievers, and rerankers. It also provides a standardized pipeline that includes text processing, model training, corpus/query encoding, and search. In addition, Tevatron incorporates well-studied methods for improving retriever effectiveness such as hard negative mining and knowledge distillation. We provide an overview of Tevatron in this paper, demonstrating its effectiveness and efficiency on multiple IR and QA datasets. We highlight Tevatron’s flexible design, which enables easy generalization across datasets, model architectures, and accelerator platforms (GPUs and TPUs). Overall, we believe that Tevatron can serve as a solid software foundation for research on neural retrieval systems, including their design, modeling, and optimization.

CCS CONCEPTS

• Information systems → Retrieval models and ranking.

KEYWORDS

Neural IR, Dense Retrieval, Sparse Retrieval, Toolkit

ACM Reference Format:

1 INTRODUCTION

The popularity of neural retrieval in the research community has greatly grown in the past few years, from neural rerankers based on the cross-encoder architecture to the most recent neural retrievers based on the bi-encoder architecture [6, 13, 16, 22, 25]. However, many recent research papers [13, 25] have focused on developing their own software stacks with specialized support only for specific datasets and models. This approach can be limiting as it does not allow for flexible generalization across different models and datasets. To address these issues, Tevatron provides researchers with access to state-of-the-art models and makes it easy for them to start a new research project on a new dataset.

Taking dense retrieval as an example: in our past papers [6, 7, 18], we have run into several engineering challenges. For example, in terms of resources, large corpora and training sets require a large amount of CPU memory; accelerator (GPU/TPU) memory usage also grows with model size. While orthogonal to actual research, these engineering hurdles slow down and constrain researchers, especially those with limited hardware resources. With Tevatron, we aim at providing a unified solution to common engineering problems. To address these challenges, Tevatron provides a unified framework that incorporates several popular and widely used open-source packages. These packages include datasets, transformers, FAISS, and Pyserini, which respectively serve as the backbone for data management, neural network modeling, embedding-based retrieval engines, and evaluation metrics.

In addition, to accommodate different research needs, we select two deep learning frameworks for Tevatron, PyTorch [21] and JAX [3]. PyTorch’s eager execution patterns and intuitive object-oriented design have gained it a massive user base in the research community. On the other hand, JAX, backed by just-in-time (JIT) XLA compilation, offers smooth transitions across hardware stacks with optimized performance.

Overall, Tevatron aims to streamline and simplify the research process, enabling researchers to focus on their actual research problems rather than engineering challenges. Additionally, Tevatron enables collaboration and ensures reproducibility, as others can easily reproduce the same experiments using the same packages.

The rest of the paper is organized as follows: Section 2 gives an overview of Tevatron. Section 3 demonstrates usage of Tevatron’s command-line interface. Section 4 shares experimental results of running Tevatron with various models and datasets.

2 TOOLKIT OVERVIEW

Tevatron1 is packaged as a Python module available on the Python Package Index. The toolkit can be installed via pip, as follows:

$ pip install tevatron==0.1.0

In this section, we provide an overview of the core components of Tevatron. We demonstrate how these components support the full pipeline of data preparation, training, encoding, and search, respectively. Code and documentation are available at tevatron.ai.

1http://tevatron.ai
within the Tevatron toolkit. For instance, with a single command, Tevatron will first download the raw data set we hosted through HuggingFace.

Tevatron's model class EncoderModel supports models in the HuggingFace transformers library that perform fast parallel data formatting and tokenization. Processed data is internally represented as a datasets.Dataset object and is stored in the Apache Arrow format, which can be memory-mapped directly affecting training/encoding performance. In Tevatron, we adopt the following core design: (1) text data are pre-tokenized before training or encoding occurs, (2) tokenized data are memory-mapped instead of lazy-loaded or residing in-memory. The former avoids overhead when running sub-word/piece level tokenizers and also reduces data traffic compared to raw text. The latter allows random data access in the training/encoding loop without consuming a large amount of physical memory.

Tevatron defines raw input format templates, shown in Fig. 1. We organize a training instance into an anchor query, a list of positive target texts, and a list of negative target texts. The positive targets are usually human-labeled and the negative targets are usually non-relevant texts from the top results of a baseline retrieval system such as BM25 [6, 13].

Users can pass raw data file pointers and processing specifications to Tevatron’s dataset classes (HFTrainDataset for training, HFQueryDataset and HFCorpusDataset for encoding), which will perform fast parallel data formatting and tokenization. Processed data is internally represented as a datasets.Dataset object and is stored in the Apache Arrow format, which can be memory-mapped and randomly accessed by offset.

For researchers who are focusing on building new models, we make a collection of popular open-access datasets self-contained within the Tevatron toolkit. For instance, with a single command, one can load the training set of MS MARCO. Under the hood, Tevatron will first download the raw data set we hosted through HuggingFace.

```python
{
    "query_id": "<query id>",
    "query": "<query text>",
    "positive_passages": [
        {
            "docid": "<passage id>",
            "title": "<passage title>",
            "text": "<passage body>", ...
        }, ...
    ],
    "negative_passages": [...]       
}
```

Figure 1: Tevatron raw data template for IR datasets.

2.1 Data Management

Having data ready to use is a critical preliminary step before training or encoding starts. Data access overhead and constraints can directly affect training/encoding performance. In Tevatron, we adopt the following core design: (1) text data are pre-tokenized before training or encoding occurs, (2) tokenized data are memory-mapped instead of lazy-loaded or residing in-memory. The former avoids overhead when running sub-word/piece level tokenizers and also reduces data traffic compared to raw text. The latter allows random data access in the training/encoding loop without consuming a large amount of physical memory.

At a high level, the bi-encoder model produces textual representations to Tevatron’s dataset classes (HFTrainDataset for training, HFQueryDataset, and HFCorpusDataset for encoding), which will perform fast parallel data formatting and tokenization. Processed data is internally represented as a datasets.Dataset object and is stored in the Apache Arrow format, which can be memory-mapped and randomly accessed by offset.

For researchers who are focusing on building new models, we make a collection of popular open-access datasets self-contained within the Tevatron toolkit. For instance, with a single command, one can load the training set of MS MARCO. Under the hood, Tevatron will first download the raw data set we hosted through HuggingFace. Then it will run the corresponding pre-defined pre-processing script to format and tokenize the downloaded data.

2.2 Neural Encoder Models

Tevatron’s model class EncoderModel is a PyTorch nn.Module subclass that defines the encoder in a standard bi-encoder architecture, which converts queries and passages individually into vector representations. Based on the type of the retrieval model, the representation can be a low-dimensional dense vector or a high-dimensional sparse vector. Functionally, the class wraps an underlying Transformer model and provides methods for text encoding and loss computation. Thanks to duck typing in Python, the EncoderModel class supports models in the HuggingFace transformers library that extend the standard base model. This means new Transformer models can be loaded into Tevatron as soon as they are available in the transformers model hub. This design helps our toolkit reduce code duplication. The EncoderModel class also handles loss computation during training. It implements a contrastive loss with in-batch negatives and can perform negative sharing across devices using the collective communication technique introduced in the NVIDIA NCCL library.³

To complete the neural retriever training setup, we introduce the TevatronTrainer class that implements miscellaneous training utilities. It controls basic setup such as the batch size and the number of training epochs. When running on multiple GPUs, the trainer will properly set up distributed training and wrap models for gradient reduction. In Tevatron, we also implement the GCTrainer class, which is a subclass of TevatronTrainer that uses gradient caching to support large batch training on memory-limited devices [9].

Taking dense retrieval as an example, DenseModel is a subclass of EncoderModel, which encodes text into dense vector representations. Representative dense retrieval models such as DPR [13] and ANCE [25] can be implemented using this class. By combining the data processor, dense retrieval model, and trainer, Tevatron abstracts the training loop of a dense retrieval model into the code block shown in Fig. 2. The training process is defined similarly for neural sparse retrieval models (e.g., SPLADE [5], uniCOIL [17]).

2.3 Vector Search

At a high level, the bi-encoder model produces textual representations for both documents and queries. In order to perform end-to-end retrieval, Tevatron implements indexing and search modules for vector search (i.e., top-k retrieval) that rely on either dense or sparse vector representations. These modules are facilitated by the use of existing libraries.

For dense retrieval, we use the FAISS library [11] as our backbone for top-k retrieval. It implements several efficient indexes in C++ and exposes them through Python interfaces. For users who want the best effectiveness, Tevatron provides a simple class called BaseFaissIPRetriever that wraps the faiss.IndexFlatIP flat

---

³https://developer.nvidia.com/nccl

---
We use dense retrieval as an example. With a flexible design in data management and support for neural architecture, custom indexes can be built for retrieval. Users can take advantage of this interface to build approximate search indexes like HNSW [19] or PQ [10].

For sparse retrieval, Tevatron directly utilizes Pyserini [15] to build inverted indexes for conducting efficient lexical search.

2.4 Additional Modules

JAX. Tevatron has a sub-package tevax that implements core functionality for JAX. Following JAX’s functional nature [3], tevax is designed with a different philosophy. We define loss functions that can be composed with other JAX transformations. In practice, they can be combined with Flax models in the HuggingFace transformer library for dense retriever training. With JAX as the backend, tevax makes it possible for a single piece of code to run on a single GPU, multiple GPUs, or TPUs.

Rerankers. Neural rerankers based on a cross-encoder architecture are commonly used in a rerank stage to further improve retrieval effectiveness. In Tevatron, we implement reranker training and inference based on the same data processing pipeline as retrievers. The reranker training implements the Localized Contrastive Estimation technique proposed by Gao et al. [8].

Distillation. Distilling knowledge from rerankers to retrievers is a commonly used technique to improve the effectiveness of retrieval models. We implement knowledge distillation by using a reranker to assign soft labels to the positive and hard negative passages for each query. The retriever learns from the reranker teacher by optimizing KL divergence.

3 TOOLKIT USAGE

On top of the various core components, Tevatron provides a set of command-line interfaces (CLIs) to drive the retrieval pipeline. With a flexible design in data management and support for neural retrievers, one can conduct many types of research without writing code. In this section, we provide an example using the Tevatron CLI to run the previously discussed components to learn a model and perform open-domain retrieval on Natural Questions (NQ) [14]. We use dense retrieval as an example.

With Tevatron, we can replicate the training of a DPR model for the NQ dataset (details in Section 4) with a single command:

```bash
python -m tevatron.driver.train \
  --dataset_name Tevatron/wikipedia-nq \
  --model_name_or_path bert-base-uncased \
  --train_n_passages 2 \ 
  --num_train_epochs 40 \ 
  --learning_rate 1e-5 \ 
  --fp16 \ 
  --grad_cache \ 
  --output_dir model-nq
```

As introduced in Section 2.1, Tevatron will automatically handle the downloading and pre-processing of our self-contained training data Tevatron/wikipedia-nq. Then, the pre-processed dataset and initialized DenseModel will be fed into GCTrainer class, as shown in

<table>
<thead>
<tr>
<th>NQ</th>
<th>TriviaQA</th>
<th>SQuAD</th>
<th>CuratedTrec</th>
<th>WebQuestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPR [13]</td>
<td>78.4 / 85.4</td>
<td>79.4 / 85.0</td>
<td>63.2 / 77.2</td>
<td>79.8 / 89.1</td>
</tr>
<tr>
<td>Tevatron</td>
<td>79.8 / 86.9</td>
<td>80.2 / 85.5</td>
<td>62.3 / 77.0</td>
<td>84.0 / 90.7</td>
</tr>
</tbody>
</table>

Table 1: Top-20/top-100 retrieval accuracy of DPR model replication on five open-domain QA datasets.

In this section, we demonstrate the effectiveness and efficiency of Tevatron by replicating retrieval results on QA tasks [2, 12, 14, 23, 24] reported in the original DPR paper [13]. We compare the models trained under the “Single” setting defined in the original paper, where each model is trained using the corresponding individual dataset. Following similar hyperparameters settings, we train the models with a learning rate of 1e-5 for 40 epochs with batch size 128. In Table 1, we see that Tevatron produces slightly higher top-k accuracy on four datasets compared to the original DPR paper, although the values are slightly lower on SQuAD. Overall, we conclude that this is a successful replication and that the Tevatron pipeline is effective.
We demonstrate the efficiency of Tevatron by comparing it with code from the original DPR repo\textsuperscript{4} on three dimensions: RAM usage, GPU memory usage, and training time. The experiments are conducted on a machine with NVIDIA A100 GPUs. In both DPR-repo and Tevatron-default settings, we train the dense retrieval model on 4 GPUs in the distributed data-parallel mode of PyTorch. By comparing the first two rows in Table 2, we see that Tevatron is more efficient along all three dimensions than the original codebase: Tevatron consumes 3/4 less RAM, 12G less GPU memory, and is 1/4 faster. This means that given the same resources, Tevatron has the potential to support more training data, larger batch sizes, and faster training.

The gradient caching feature of Tevatron can further improve GPU memory efficiency [9]. DPR training requires batch size 128 to obtain the level of retrieval accuracy reported above. With the original DPR repo, users cannot train a model with a sufficiently large batch size if GPU resources are limited, which will result in a drop in retrieval accuracy. Tevatron provides users the option to train dense retrievers using limited GPU resources, retaining the same batch size for each optimization step. To illustrate this, we conduct experiments with Tevatron-GradCache on a single GPU. Tevatron-GradCache trains dense retrievers by splitting each batch of size 128 into sub-batches of size 32. By conducting two rounds of forward steps [9], the model update step of Tevatron-GradCache is mathematically equivalent to Tevatron-default. This experiment only consumes 4G RAM and 15G GPU memory, to train a DPR model on the NQ dataset with the desired equivalent batch size (but at the cost of a longer training time). By reducing the sub-batch size, Tevatron-GradCache can save even more GPU memory.

We also evaluate the performance of training a dense retrieval model using the JAX backend of Tevatron on a V3-8 TPU VM. When DPR [13] was first published, it costs around a day to train the model on the NQ dataset using the original DPR repo with 8× A100 GPU or V3-8 TPU with a learning rate of 5e-6 and batch size 64 for 3 epochs. HN indicates hard negative mining.

Table 3 shows the results of training dense retrievers using Tevatron with different Transformer backbones on the MS MARCO passage ranking task. The models are trained on 4× A100 GPU or V3-8 TPU with a learning rate of 5e-6 and batch size 64 for 3 epochs. HN indicates hard negative mining.

---

\textsuperscript{4}https://github.com/facebookresearch/DPR; to be clear, the efficiency results are based on the master branch on 2022-02-12.
REFERENCES


