Moving Stuff Around

A study on the efficiency of moving documents into memory for Neural IR models

Arthur Câmara
A.BarbosaCamara@tudelft.nl
Delft University of Technology
Delft, Netherlands

Claudia Hauff
c.hauff@tudelft.nl
Delft University of Technology
Delft, Netherlands

ABSTRACT

When training neural rankers with Large Language models, a practitioner generally leverages multiple GPUs, which increases the amount of available VRAM, allowing larger batches, and decreasing training time. However, managing how data moves between disk (HDD), memory (RAM) and video memory (VRAM) is generally overlooked by most research implementations when training such data-hungry models. Instead, most implementations use a naive approach of loading all documents from disk into memory, which delegates to the framework (e.g., PyTorch) the task of moving data into VRAM. With the increasing sizes of datasets available, a natural question arises: *Is this the optimal solution for optimizing training time?* Therefore, we study the impact on performance (samples per second) and memory footprint (MBs allocated) of three approaches to moving documents in this hierarchy and how they scale with multiple GPUs. Specifically, we look into: (i) Pre-loading into memory, (ii) Reading from disk files with a lookup table and (iii) Reading from compressed files on disk using a memory map. We show that when using some of the most common libraries for IR research, loading all documents into memory is not always the fastest option and also not feasible for more than a couple of GPUs, given its large memory footprint. Meanwhile, a good implementation of data handling is considerably faster and more scalable. Additionally, we demonstrate that using multi-processing instead of multi-threading can prevent problems caused by Python's Global Interpreter Lock (GIL). Finally, we also discuss how popular techniques for improving loading times, like memory pining, multiple workers, and RAMDISK usage, can reduce the training time further with minor memory overhead.

ACM Reference Format:

Arthur Câmara and Claudia Hauff. 2022. Moving Stuff Around: A study on the efficiency of moving documents into memory for Neural IR models. In *Proceedings of Workshop on Reaching Efficiency in Neural Information Retrieval @ SIGIR 2022 (RENEUIR@SIGIR'22).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3406522.3446012

This research has been supported by *NWO* projects *SearchX* (639.022.722) and *Aspasia* (015.013.027).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

RENEUIR@SIGIR'22, July 15, 2022, Madrid, Spain

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-8055-3/21/03…$15.00
https://doi.org/10.1145/3406522.3446012

1 INTRODUCTION

Current advances in Neural IR, mainly fueled by pre-trained, Large Language Models like BERT, have finally made deep learning a practical, and even expected, part of most current Information Retrieval research. However, even with pre-trained models for the most common corpora, like MSMarco, training (or rather, fine-tuning) a neural ranker is still commonplace, given a new dataset, new architecture, or even new training scheme.

The problem of efficiently moving data from disk into memory is not exclusive to Information Retrieval. For example, a recent work by [13] shows that some deep learning models for vision and speech processing can waste between 10 and 70% of their training time on I/O blocking operations.

Much attention is devoted to creating the most parameter-efficient or faster training methods for neural models [19]. On the other hand, not much work has been dedicated to one of the most time-consuming steps when training such rankers: the time spent *moving* data from disk into main memory. This step can take upwards of 15% of the training time, as seen in Figure 1, and is mainly neglected, with most open-source implementations of neural rankers resorting to loading all the corpus documents into the main memory when starting the training process.

As corpora grow more extensive, it quickly overcomes the available RAM, making it infeasible to load all documents into memory. Take, for instance, TREC's Disks 4 and 5 [25], used by Robust04 [24], a commonly used ranking dataset. In its uncompressed form, its size accounts for about 2GB. Meanwhile, MsMarco [14], the dataset used for TREC's Deep Learning Track in the last few years [2–4], takes over ten times that, at over 22GB, with its newer version, V2, surpassing the 100GB mark. Finally, other text corpora, like the Clean Common Crawl Corpus, used by the T5 model [20], can go even further, with more than 800GB needed to store the corpus.

The method of pre-loading all of the data from a corpus into memory, here called `in-memory`, allows for simple document loading, with usually a single line of code being necessary for accessing a document in constant time, using a lookup table (`documents[doc_id]`). However, as discussed here, this approach is not only impossible for large datasets but also *slower* than reading from disk in most scenarios and does not scale with the number of GPUs. This is further exacerbated when following best programming practices (i.e. using one *process* per GPU instead of one *thread* per GPU).

Most available PyTorch implementations of ranker models that use this method of pre-fetching documents also resort to using the `DataParallel` (`DP`) module from PyTorch when training with multiple GPUs. While convenient and easy to implement, this approach is not only *not* recommended by PyTorch's official documentation[1],

[1] https://pytorch.org/docs/1.11/notes/cuda.html#cuda-nn-ddp-instead

it also spawns one thread for each GPU, which, given Python's nature and Global Interpreter Lock (*GIL*) [2], can cause racing conditions, which leads to considerate overhead when more than one thread tries to read from the same memory pool.

Instead, the preferred method to use when training with multiple GPUs is to use the built-in PyTorch library for multi-processing (i.e. `DistributedDataParallel` (`DDP`) [9]). Spawning one process for each GPU makes it possible to sidestep the GIL completely. Additionally, `DistributedDataParallel` also allows a practitioner to divide the training between multiple nodes (i.e. multiple computers or hosts). It also allows the model parameters to be divided for larger models, making larger models that would not be trainable in a single host or GPU usable.

In this work, we argue that these two practices (using `in-memory` for loading documents and `DP` for dividing the task among multiple GPUs), while easy to implement, *should not be used* when training neural rankers with large amounts of data. Moreover, as we discuss further, these approaches lead to (i) code that is of lower quality (i.e. not attaining the best PyTorch practices) and (ii) waste of resources, as methods that take less memory and have higher performance (i.e. can process more samples per second) are available.

We show that, with minor code changes and by being smarter when moving documents from disk, we can increase the performance of our training loops, greatly decrease memory usage, and increase the scalability of our models, making better use of the limited computing resources usually made available to academic researchers.

Therefore, we show how three different approaches to loading data (here referenced as `Loaders`) differ in performance (i.e. samples processed per second) and memory usage (i.e. GB of main memory used) when working on 1, 2, 4 or 8 consumer-grade GPUs. More specifically, we compare (i) `in-memory` loading (i.e. pre-fetching all documents in memory), (ii) `indexed`, reading all documents directly from disk, while using a simple lookup table to locate document positions in files, and (iii) `ir-datasets` [11], a toolkit for loading datasets specifically for Information Retrieval research, that uses highly efficiently ordered `numpy` [6] indexes in memory and compressed files in disk, that is shown to have a good performance when reading documents from disk. We also examine if common tricks for improving data loading latency, specifically `pin_memory`, RAMDISK and increasing the number of threads dedicated to data loading speed up training in these scenarios.

The code used in our experiments is available at https://github.com/ArthurCamara/ir_efficiency. Additionally, a WandB project dashboard with results is also publicly available at http://wandb.ai/acamara/ir_efficiency.

2 METHODOLOGY

This section outlines how we set up our experiments and the testing environment. We start with a brief description of the main libraries and classes used, followed by our machine configuration and how the example neural ranker was defined. We close this section by describing our implementations for `Loaders` and how we organized our experiments regarding hyperparameters and different configurations.

[2] https://docs.python.org/3/glossary.html#term-global-interpreter-lock

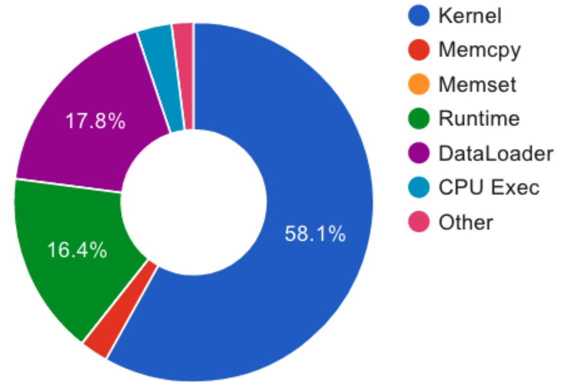

Figure 1: Tensorboard profiler for a CrossEncoder training run (500 steps) using the `ir-datasets` package for reading documents. 17.8% **of the runtime was spent on `DataLoader`, that is, reading and processing raw data from disk and sending to RAM**

2.1 Libraries

While other deep learning frameworks exist, most open-source research implementations of neural rankers rely on the same handful of libraries. Therefore, we also use these popular Python libraries in our experiments, with their latest stable version. Namely, we use Python, PyTorch [17], and Hugging Face Transformers [26], all installed using `pip` (cf. `Pipfile` in Listing 1 for the specific version numbers) sourcing from the default Python Package Index `PyPI`.

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
torch = "1.11.0"
transformers = "4.18"
accelerate = "0.6.2"
wandb = "0.12.15"
tqdm = "4.64.0"
psutil = "5.9.0"
ir-datasets = "0.5.1"

[requires]
python_version = "3.9"
```

Listing 1: Pipfile used for the experiments

When handling data, we kept as much as possible to standard PyTorch classes. For example, our implementations for all dataset classes inherit from `Dataset`; we also use the standard `DataLoader` class for batching and splitting the data between GPUs [3].

For our experiments with multiple GPUs, we use both the standard `DataParallel` (`DP`) for *multi-threaded* experiments and Hugging

[3] Throughout this paper, we suppress the parents of the standard PyTorch classes for simplicity

Face Accelerate[4] for *multi-processed* experiments. It is generally preferred to run parallel programs in Python using multiple independent processes rather than one process with multiple threads. This is because Python, in its current state, is limited in its multi-threaded capabilities by its Global Interpreter Locker (GIL).

The biggest drawback, and probably the main reason most research implementations don't adopt `DDP`, is that doing so is not as simple as `DP`. It requires, in some cases, explicit synchronization and communication between different processes and, therefore, more changes to the training code when compared to `DP`. However, many libraries provide abstractions over `DDP`, making it easier to use multi-processing without the added implementation complexities. Some of these libraries include the previously mentioned Accelerate, PyTorch Lightning [5] and Ray [6]. It is worth mentioning that, according to the PyTorch documentation[7], using `DDP` is the preferred implementation when using multiple GPUs.

For tracking our experiments, we rely on Wandb [1]. It can track multiple hyperparameters and metrics in real-time while making it easy to evaluate multiple runs at the same time[8].

We ran all of our experiments in the same machine, equipped with a 56-core Intel Xeon E5-2690 v4 running at 2.60GHz with 128GB of DDR4 main memory running at 2400 MT/s. The datasets are stored on a 2TB SSD Drive, connected via a SATA 3.0 connector. We also make use of 8 Nvidia GTX 1080 Ti. While slightly older and without Tensor Cores for accelerating Tensor-based operations, it is a still commonplace GPU for smaller laboratories and research groups. It also potentially makes the latency of loading data into memory less noticeable since the tensor operations may take longer. Therefore, the impacts reported here could be potentially more significant on more modern systems, with faster kernel operations but similar memory transfer speeds between disk and memory.

2.2 `Loaders` implementations

We aim to cover a comprehensive set of possibilities when training a neural ranker model. However, it is not practical to cover every possible setup or scenario. Additionally, we are not interested in finding the best model effectiveness-wise but rather studying how the different options discussed here behave. Therefore, we restrict our tests to one common ranker model and reuse it in all our experiments. We use a commonly used Cross-encoder [10, 15, 28] with a DistilBERT-Base model [22] that concatenates query and documents with a `[SEP]` token. The output `[CLS]` token embedding is fed to a 1-layer feed-forward network that estimates the relevance of the document w.r.t. the query. This model is also called BERT_{CAT} [7].

We also train with a fixed batch size of 16 per GPU, the largest size we could fit given our hardware. Each batch consists of triples of queries, positive and negative samples from the document ranking set from the TREC Deep Learning (DL) 2019 shared task [3], where the documents come from the MSMarco document dataset [14]. The negative samples are randomly selected from all documents in the collection at runtime[9].

We implement three different approaches for loading data into memory (`Loaders`). Other possibilities—not discussed here—are using an in-memory database for storing documents, like Redis, or an inverse-index solution, like Anserini [27] or Terrier [12]. While the former could be helpful in a situation of high throughput or a production-ready scenario, we did not find a significant difference in keeping documents in memory in past experiments and therefore do not pursue it further. As for other IR toolkits, while highly optimized for high throughput and fast lookup, they can add significant overhead when looking for individual documents, as shown by MacAvaney et al. [11].

We now discuss our three `Loaders` in turn.

`in-memory`. The most common approach in research code for neural rankers is to load all of the corpus into memory with a standard Python dictionary. This approach is easy to implement and use, generally requiring a single `for` loop over the documents for loading into memory and a single lookup in a dictionary that takes constant time. However, this approach leads to high memory usage, especially when using multiple processes, or a considerable overhead when using multiple threads. When using `DistributedDataParallel` (i.e., multiple processes), each process loads a copy of the whole corpus on its own memory space during startup. When using multiple threads (i.e. `DataParallel`), the data remains in a shared memory space, which can lead to overhead and race conditions due to Python's GIL. In these cases, an alternative is to increase the number of workers in the `DataLoader` object, but that can trigger a copy of the whole dataset to each loader, which can increase memory usage linearly with the number of workers used. [10]

`indexed`. As an alternative to `in-memory`, `indexed` loads only a pointer to each document byte-level address in the original file. Therefore, when requesting a document, this `Loader` looks into a small in-memory dictionary with the document's address in a file stored on disk. If there already exists an opened `FileIO` object (i.e., an open file object), Python's `.seek()` function is $O(1)$ in run time, since it calls Linux's `lseek` function, that executes in constant time. Therefore, reading a document into memory is only dependent on the size of the document, not the size of the corpus, as a naive lookup would be, with minimal memory overhead. A simplified version of this `Loader` is shown in Listing 2.

`ir_datasets`. `ir_datasets` [11] is a recent Python library that deals with a similar problem as we discuss here. The authors proposed a toolkit for abstracting common operations performed on datasets used for Information Retrieval research by simplifying downloading, storage and iteration over the documents, queries, and relevance labels for over 40 datasets. The dataset implements several optimizations for speed and quick experimentation, like lazily downloading datasets and a `docs_store` API with a similar implementation as the above `indexed`, but using `lz4` for compression of documents in disk and a sorted index offset, built with memory

[4] https://github.com/huggingface/accelerate
[5] https://www.pytorchlightning.ai
[6] https://www.ray.io
[7] https://pytorch.org/docs/1.11/generated/torch.nn.DataParallel.html
[8] The workspace with all runs and logging data can be accessed at https://wandb.ai/acamara/ir_efficiency

[9] Since we pre-load all documents identifiers in memory, the selection runs in constant time, and the document fetching stage performs according to each `Loader`.
[10] SBERT [21], SPLADE [5] and MatchMaker [8] are examples of open-source implementations that use `DP`.

maps from Numpy (`numpy.memap`) for quickly finding documents in disk. This is also similar to the implementation of Transformers-Rank[11], used in Penha and Hauff [18]. We expect this recently open-sourced library to be used more often in the future in combination with other toolkits. As a case in point, PyTerrier [12] already includes `ir_datasets` as part of the toolkit.

2.3 Other optimizations

Other possible optimizations that we also experiment with are mainly related to how PyTorch deals with moving data from disk and main memory into the GPU's memory and are commonly accepted as good optimizations for PyTorch[12].

First, we experiment with setting pinned memory (i.e., setting `pin_memory=True` when defining a `DataLoader`). Pinning memory causes PyTorch to pre-allocate a region of non-paged main memory for it, avoiding one implicit data copy inside the machine's main memory. Pinning memory also frees the CPU to fetch more data when done instead of waiting for the GPU operations to finish.

Another commonly used option is for the `DataLoader` to use in multiple threads by setting the `num_workers` parameter from the `DataLoader` object to a value greater than 0 (its default value). Essentially, it allows the CPU to fetch data in multiple threads simultaneously without waiting for the GPU to finish its computations.

Finally, we explore if we can mimic the fast access provided by the `in-memory` approach by caching the files into RAMDISK. Using RAMDISK is a way to store data in the main memory while accessing it as a regular file. In practical terms, it means moving the corpus files from disk into a region in memory, usually on `/dev/shm` in most Linux distributions. In theory, moving the files into memory can mimic the speed benefits of `in-memory` without its higher memory usage, since each thread or process would *not* need to copy or re-read, respectively, the whole corpus into memory, as the OS sees the data as a regular file in disk.

```
def load_index(corpus_path:str):
    """Read all docs positions and store in-memory"""
    file_handler = open(corpus_path, "b")
    line = file_handler.readline()
    index = {}
    current_position = file_handler.tell()
    while line!=b"":
        line = line.decode("utf-8")
        doc_id, doc_text = line.split("\t")
        index[doc_id] = position
        current_position = file_handler.tell()
        line = file_handler.readline()
    return file_handler, index

def get_doc(index:dict, file_handler:FileIO, doc_id:str):
    """Return a doc given its id, a file and index"""
    position = index[doc_id]
    file_handler.seek(position) #O(1), calls lseek()
    return file_handler.readline().decode("utf-8")
```

Listing 2: Simplified implementation for `indexed`

[11] https://github.com/Guzpenha/transformer_rankers

[12] c.f. PyTorch's documentation on performance tuning and this NVidia's ECCV 2020 [23] tutorial on Accelerating Computer Vision with Mixed Precision

3 RESULTS

In this section, we describe the results of the experiments described in Section 2. Recall that we are not interested in which method would yield better overall retrieval effectiveness (as they all share the same model and hyperparameters, it is unlikely that any significant difference would arise). Hence, we run each experiment for 1K training batches (between 16K and 128K query-document pairs in total, depending on the number of GPUs used) and report the average number of samples processed per second, as well as the total amount of memory used by the process. Note that, when running in multiple processes (i.e. using `DDP` instead of `DataParallel`), the reported memory usage from Python's `psutil` library is only for the main process, rather than for the whole program. Therefore, for a fair comparison, we multiply the reported memory usage by the number of GPUs used in that experiment. Also, note that we are *not* considering the time each model takes to load the dataset. This step is only performed once and, given the relatively small number of steps performed, would bias the results against `in-memory`, as it naturally takes considerably longer to load the corpus into memory.

3.1 Samples per second

We begin by looking into different `Loaders` described in Section 2, and how they perform when used with either `DP` or `DDP`, with 1, 2, 4 or 8 GPUs. The results for how many samples per second each `Loader` can process is presented in Table 1.

Perhaps surprisingly, we observe that the performance advantage of running with `in-memory` is minimal to negative, especially as we increase the number of GPUs when using `DDP`. And, if using `ir_datasets` on two or more GPUs, its performance is better than `in-memory` in every scenario. We believe this drop in performance of `in-memory`, especially when using `DP`, has two leading causes. First, modern operating systems (OS) are highly optimized for reading data from disk, especially if using fast SSDs. Therefore, OS-level features, like pre-fetching, can play an important role here. But, even more interesting, the way that `DP` is implemented and how it splits batches between the GPUs is probably even more critical. Given Python's GIL and the multi-threaded nature of `DP`, the `in-memory` object resides in a shared memory pool between each thread (i.e. each GPU has one thread by default). Therefore, when two or more threads request data at the same time, the GIL is used to control access to that pool with a single semaphore lock, only allowing one thread at a time to access this shared pool, which delays the access of multiple threads to the dataset. The same issue appears with `indexed`, since Using optimizations like increasing the number of workers on the `DataLoader` initialization can help, as we show in Section 3.3.

Another interesting finding is that, in most cases, using `DDP` is slightly *faster* than using `DP`, despite the communication overhead between processes. Again, we can blame the GIL. The more threads try to access the same memory pool, the more significant the bottleneck. The only noteworthy exception is `ir_datasets` on 8GPUs. The reason is hard to pin down, but, looking at the other differences between `DDP` and `DP`, it seems like the communication overhead between 8 processes actually made a significant impact.

The drop in performance of `indexed` on `DP` is also worth noticing. As more GPUs are added, the GIL impact is essentially doubled.

Loader	1 GPU		2 GPUs		4 GPUs		8 GPUs	
	DP	DDP	DP	DDP	DP	DDP	DP	DDP
in-memory	39.56	39.65	72.33	73.73	140.47	OOM	263.54	OOM
indexed	**39.88 (0.80%)**	**39.71 (0.15%)**	72.09 (-0.34%)	74.98 (1.69%)	139.92 (-0.39%)	140.04 (-)	262.56 (-0.37%)	262.62 (-)
ir_datasets	**39.88 (0.80%)**	39.16 (-1.25%)	**72.78 (0.61%)**	**75.45 (2.33%)**	**140.66 (0.13%)**	**141.31 (-)**	**268.29 (1.80%)**	**264.08 (-)**

Table 1: Number of samples/s processed by each `Loader` on 1, 2, 4 and 8 GPUs, using either `DataParallel`(`DP`) (i.e. Multiple Threads) or `DistributedDataParallel` (`DDP`) (i.e. Multiple Processes). Values in (parenthesis) indicate the difference, in percentage, between that value and `in-memory`. Results in blue indicates a better performance when compared to `in-memory`. Results in pink indicates a worst performance. Results in bold indicates the better performance on that column. Results underlined indicates the best result for that number of GPUs. Results in red indicates that that experiment failed to run due to a Out Of Memory Error

First, there is a need to read from the shared lookup dictionary, which is not thread-safe. The GIL is, however, invoked a *second* time, as reading from a file is also **NOT** thread-safe in Python. There are some possibilities to improve performance in this scenario, like using a `Queue` [13], but that would require considerable re-working of the code.

Finally, we note that, given its multi-processing nature, `DDP` combined with `in-memory` is not a feasible option for more than a couple of GPUs (at least for this dataset). With each process replicating the complete dataset in its memory space, we quickly run into Out-Of-Memory errors, with 4 and 8 GPUs crashing the machine before even starting the training process.

3.2 Memory footprint

A large dataset kept in memory would, naturally, quickly exhaust a machine's memory if replicated multiple times. However, if the performance can be increased with a suitable memory usage increase, the trade-off can be worth it. Therefore, in Table 2 we report the memory footprint for each `Loader`, parallelism method and number of GPUs. From that, it is clear that, overall, `ir_datasets` has a significantly smaller memory footprint than either `in-memory` or `indexed`, regardless of parallelism method or number of GPUs.

Unsurprising, `in-memory` has, by far, the largest memory footprint, with about 40GB allocated for each running process. Given the collection size on disk (22GB uncompressed), this is a significant increase over the original size. As expected, when using `DDP`, memory usage grows linearly with each new instance, regardless of `Loader`. This, combined with the large footprint of `in-memory` makes it impractical to be used in scenarios with more than a couple of GPUs, especially on a single-node training scheme, where each process would use memory from the same machine instead of sharing resources among multiple nodes.

However, when dealing with `DP`, `in-memory`'s footprint does not seem to grow as fast as `indexed`, or `ir_datasets`. This lower-than-expected footprint can be explained by the simplicity of the `in-memory` implementation. By using a single, simple data structure (a `dict()`, that is essentially a hash map), it can easily share this same structure with all threads (albeit not in a thread-safe manner, given the GIL).

Meanwhile, the growth shown by `indexed`'s implementation appears to be triggering some caching by the OS, growing the footprint significantly faster than other methods (however, it still stays almost 70% smaller than the footprint from `in-memory`, even with 8 GPUs). As for `ir_datasets`, its efficient implementation of Numpy's memory mapping greatly diminishes how fast its memory footprint grows for each extra thread.

Both `indexed` and `ir_datasets`, however, still increases its memory usage significantly faster than `in-memory` when using `DP`. Due to their complexity, we believe this happens conversely to the simplicity of `in-memory`'s data structures. Given that extra layer of indexes and optimized data formats, it is possible that either the OS or PyTorch's internals are replicating part of the data structure or pre-fetching documents from disk[14].

3.3 Common optimization strategies

Given the popularity of PyTorch, several optimization strategies are often suggested to increase its performance when loading data from disk. Therefore, we also explore how these may or may not impact performance for training neural rankers. Table 3 shows the impact of some of these techniques. For the sake of comparison, for this analysis, we fix the loader and parallel approach to the ones with best performance in Tables 1 and 2 (i.e., `ir_datasets` on `DP`) and 4 GPUs.

From the results, it is clear that most of them seem to have impacts ranging from negative, with deteriorated final results, to significant improvements, increasing performance even more than changing `Loader` or parallelism mechanism. For instance, using `pin_memory` alone shows an decrease of 1.05 samples/s. However, increasing the number of processes dedicated to data loading from 0 (no parallelism) to 1 adds 5.3 samples/s, while, paradoxically, changing to 8 workers decreases the performance by 9.26 samples/s compared with no worker parallelism. Again, the GIL is to blame. While having one thread dedicated to loading data is useful, as the main thread can be dedicated to other tasks, adding more workers increases the probability that more than multiple threads try to access the same memory space simultaneously, running into race conditions.

Combining optimization, however, does seem to give a slight boost to performance. Using both `pin_memory` and one worker increases the performance by 5.5 samples/s, slightly more than using

[13] https://docs.python.org/3.9/library/queue.html#Queue.Queue

[14] We cannot find evidence, in PyTorch's documentation or code, that it does so explicitly. Therefore, we are inclined to believe this is due to some preemptive optimization by the OS.

Loader	1 GPU		2 GPUs		4 GPUs		8 GPUs	
	DP	DDP	DP	DDP	DP	DDP	DP	DDP
in-memory	40.23	40.28	40.84	80.51	42.14	OOM	44.89	OOM
indexed	4.21 (-89.52%)	4.26 (-89.44%)	5.67 (-86.12%)	8.45 (-89.50%)	8.57 (-79.67%)	16.84 (-)	13.96 (-68.91%)	34.14 (-)
ir_datasets	**2.79 (-93.07%)**	**2.83 (-92.97%)**	**3.23 (-92.09%)**	**5.54 (-93.12%)**	**4.34 (-89.69%)**	**11.20 (-)**	**6.72 (-85.03%)**	**22.59 (-)**

Table 2: Memory footprint (in GBs) for each `Loader` on 1, 2, 4 and 8 GPUs, using either `DataParallel` (`DP`) (i.e. Multiple Threads) or `DistributedDataParallel` (`DDP`) (i.e. Multiple Processes). Results in bold indicates the better performance (lower memory usage) on that column. Results underlined indicates the best result for that number of GPUs. Results in red indicates that that experiment failed to run due to a Out Of Memory Error

Optimization Strategy	Value	Samples/s	Memory (GB)
pin_memory	No	**141.62**	10.27
	Yes	140.57	**10.21**
num_workers	1	**146.96**	10.00
	2	145.29	10.00
	4	141.66	10.00
	8	132.37	**9.99**
pin_memory + num_workers	1	147.16	**10.07**
	2	147.09	10.20
	4	147.13	**10.07**
	8	**147.26**	10.15
RAMDISK	No	**141.62**	10.27
	Yes	140.81	**10.23**
pin_memory + 8 workers + RAMDISK		147.23	10.05

Table 3: Impact of some of the most traditional optimization techniques for PyTorch using `ir_datasets` and `DDP` on 4 GPUs. Results in bold indicate best performance for that type of strategy. The underlined result is the best result overall. RAMDISK also adds the size of the corpus files (22GB in this case) to the total memory usage, as the files are copied to main memory

`pin_memory` alone, and the impact of adding extra workers does not seem to be negative, as without the pinned memory.

It is interesting to note that using RAMDISK (i.e., using main memory as disk space) decreases performance. Still, if used with other optimizations, line `pin_memory` and more workers, it doesn't have a meaningful impact.

Memory-wise, the difference is minor, at less than 300MBs, since `ir_datasets` already have a considerably low memory footprint and can probably be ignored in most cases.

4 CONCLUSION AND RECOMMENDATIONS

In this work, we presented how different methods for loading document corpora from disk into main memory behave in multiple scenarios. The main findings from our experiments are: (i) A naive implementation of loading the corpus in memory (i.e. the `in-memory` loader) does not lead to performance gains, especially with more than one GPU (ii) Python's Global Interpreter Lock (GIL) can quickly impact performance when using more than one GPU and (iii) Using `DistributedDataParallel` to sidestep the GIL leads to better scalability, but combining it with `in-memory` is not a viable option for more than a few GPUs.

With these results, we have a few recommendations, presented as **Dos** (✓) and **Don'ts** (✗) for practitioners and researchers wishing to train a neural ranker using PyTorch. As shown, when adopted in conjunction, these should increase training performance and scalability of training procedures:

- ✗ **Don't load all of the documents in-memory**. As shown, `in-memory` does not lead to any performance improvement. On the contrary, it is **slower** than other approaches, greatly increases memory usage, and does not allow the training to scale well to multiple GPUs.
- ✗ **Don't use `DataParallel`**. Not only is `DP` not recommender by PyTorch's documentation, it can quickly run into bottlenecks with Python's GIL and is slower than `DDP`.
- ✓ **Do use more efficient methods for reading documents from disk** and be aware that they may also be impacted by the GIL. In particular, `ir_datasets` [11] seems to strike a good balance between ease of use and performance, while providing support to a large number of datasets in multiple formats.
- ✓ **Do use `DistributedDataParallel`, preferably with a wrapper library**. It is the recommended method by PyTorch, it performs better than `DP` and can handle multiple training nodes if needed. To counter the extra code needed, wrappers, like Accelerate, can be used.
- ✓ **Do use `pin_memory` and increase `num_workers` to at least 1**. While they may not improve much individually, combining common optimizations, specially setting only one extra thread for loading data, therefore avoiding the GIL, and setting `pin_memory` to True can result in a sensible increase in performance.

4.1 Caveats

Naturally, we could not cover every possible combination of parameters or libraries. Rather, we choose to focus on simple approaches that can better extract extra performance when training neural rankers.

Other strategies that can be useful, but were not studied here include: Setting gradients to `None` instead of using PyTorch's `zero_grad` after each backward pass, and using AMP and FP16. The former can bypass some of the overhead of explicitly calling `zero_grad` while having the same effect of zeroing the gradients of the network.The latter are two techniques that decrease the precision of the weights in the model, from 32bit to 16bit, and, in the case of AMP, allow for mixing between both automatically. The GPUs used for this experiment do not have support for this, and therefore, we could not test it. However, it's expected that this should lead to a

considerable performance increase, since the lower precision can drastically decrease computation time and decrease VRAM usage, allowing for larger batches.

Our work is limited to PyTorch and Hugging Face Transformers. While the most popular frameworks, especially for research, we recognize that other options exist, like TensorFlow and TF-Ranking [16], that are widely used, especially in industry.

Finally our conclusions regarding differences between `DP` and `DDP` rely heavily on how Python currently implements multiple threads, specially the GIL. This may, however, change in the future, as exploratory work to remove the GIL is underway [15]. We are excited to see where this could lead, and what impact it may have on training performance. In theory, the removal of the GIL could mean the best of both worlds, with no performance degradation of multiple threads and lower memory usage, since there would be no need of data replication across workers. We note, however that, regardless of that, GIL or no GIL, a strategy like `in-memory` still requires that the corpus fits in main memory.

REFERENCES

[1] Lukas Biewald. 2020. *Experiment Tracking with Weights and Biases.* Retrieved June 23, 2022 from https://www.wandb.com/

[2] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, and Daniel Campos. 2020. Overview of the TREC 2020 Deep Learning Track. In *Proc. 29th TREC (NIST Special Publication)*. National Institute of Standards and Technology (NIST).

[3] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, Daniel Campos, and Ellen M. Voorhees. 2019. Overview of the TREC 2019 deep learning track. In *Proc. 28th TREC (NIST Special Publication)*. National Institute of Standards and Technology (NIST).

[4] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, Daniel Campos, Ellen M. Voorhees, and Ian Soboroff. 2021. TREC Deep Learning Track: Reusable Test Collections in the Large Data Regime. In *Proc. 44th ACM SIGIR*. 2369–2375.

[5] Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE: Sparse Lexical and Expansion Model for First Stage Ranking. In *Proc. 44th ACM SIGIR*. 2288–2292.

[6] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[7] Sebastian Hofstätter, Sheng-Chieh Lin, Jheng-Hong Yang, Jimmy Lin, and Allan Hanbury. 2021. Efficiently Teaching an Effective Dense Retriever with Balanced Topic Aware Sampling. In *Proc. 44th ACM SIGIR*. 113–122.

[8] Sebastian Hofstätter, Hamed Zamani, Bhaskar Mitra, Nick Craswell, and Allan Hanbury. 2020. Local Self-Attention over Long Text for Efficient Document Retrieval. In *Proc. 43th ACM SIGIR*. 2021–2024.

[9] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018.

[10] Sean MacAvaney, Andrew Yates, Arman Cohan, and Nazli Goharian. 2019. CEDR: Contextualized Embeddings for Document Ranking. In *Proc. 42nd ACM SIGIR*. 1101–1104.

[11] Sean MacAvaney, Andrew Yates, Sergey Feldman, Doug Downey, Arman Cohan, and Nazli Goharian. 2021. Simplified Data Wrangling with ir_datasets. In *Proc. 44th ACM SIGIR*. 2429–2436.

[12] Craig Macdonald, Nicola Tonellotto, Sean MacAvaney, and Iadh Ounis. 2021. PyTerrier: Declarative Experimentation in Python from BM25 to Dense Retrieval. In *Proc. 30th ACM CIKM*. 4526–4533.

[13] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. *Proc. VLDB Endow.* 14, 5 (2021), 771–784.

[14] 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@30th NIPS*.

[15] Rodrigo Nogueira and Kyunghyun Cho. 2019. Passage Re-ranking with BERT. *ArXiv* abs/1901.04085 (2019).

[16] Rama Kumar Pasumarthi, Xuanhui Wang, Cheng Li, Sebastian Bruch, Michael Bendersky, Marc Najork, Jan Pfeifer, Nadav Golbandi, Rohan Anil, and Stephan Wolf. 2018. TF-Ranking: Scalable TensorFlow Library for Learning-to-Rank. In *Proc 25th ACM SIGKDD*. 2970–2978.

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proc 32nd NeurIPS*. 8024–8035.

[18] Gustavo Penha and Claudia Hauff. 2021. Weakly Supervised Label Smoothing. In *Proc. 43rd ECIR*. 334–341.

[19] Ronak Pradeep, Yuqi Liu, Xinyu Zhang, Yilin Li, Andrew Yates, and Jimmy Lin. 2022. Squeezing Water from a Stone: A Bag of Tricks for Further Improving Cross-Encoder Effectiveness for Reranking. In *Proc. 44rd ECIR*. 655–670.

[20] 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. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.

[21] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proc. 9th ACL EMNLP*. 3980–3990.

[22] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR* abs/1910.01108 (2019).

[23] Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm. 2020. Computer Vision – ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings. *Computer Vision – ECCV 2020* (2020).

[24] Ellen M. Voorhees. 2004. Overview of the TREC 2004 Robust Track. In *Proc. 13th TREC (NIST Special Publication)*. National Institute of Standards and Technology (NIST).

[25] Ellen M. Voorhees and Donna Harman. 1997. Overview of the Sixth Text REtrieval Conference (TREC-6). In *Proc. 6th TREC (NIST Special Publication)*. National Institute of Standards and Technology (NIST), 1–24.

[26] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proc. 10th ACL EMNLP*. 38–45.

[27] Peilin Yang, Hui Fang, and Jimmy Lin. 2017. Anserini: Enabling the Use of Lucene for Information Retrieval Research. In *Proc. 40th ACM SIGIR*. 1253–1256.

[28] Zeynep Akkalyoncu Yilmaz, Wei Yang, Haotian Zhang, and Jimmy Lin. 2019. Cross-Domain Modeling of Sentence-Level Evidence for Document Retrieval. In *Proc. 9th ACL EMNLP*. 3488–3494.

[15] https://pyfound.blogspot.com/2022/05/the-2022-python-language-summit-python_11.html