A Pruning-Based Deep Learning Approach for Information Retrieval

Luca Zanchetta
Sapienza University of Rome
zanchetta.1848878@studenti.uniroma1.it

Pasquale Mocerino
Sapienza University of Rome
mocerino.1919964@studenti.uniroma1.it

Simone Scaccia
Sapienza University of Rome
scaccia.2045976@studenti.uniroma1.it

Abstract

This paper introduces a refined approach to information retrieval by enhancing an existing inverted index framework through pruning and fine-tuning techniques within the realm of deep learning. Our work focuses on optimizing an established model, denoted as f, which takes a query q as input and produces a ranked list of document IDs. By leveraging deep learning methodologies, we have meticulously pruned and fine-tuned the existing framework to better capture semantic relationships between queries and documents, thereby improving retrieval accuracy and efficiency. Our approach builds upon the foundation of prior research, refining it to address the contemporary challenges posed by massive datasets and expensiveness of reaching state-of-the-art performance with limited hardware resources. Extensive experimentation on the proposed MS Marco dataset [1] validates the efficacy of our refined approach, showcasing notable enhancements in retrieval performance compared to the unmodified baseline. We have analyzed the effects of pruning and fine-tuning methods in order to understand how they improve our model. This helps us see exactly how we can make the model work better within the inverted index system. In summary, our work contributes to the ongoing evolution of information retrieval methodologies by refining an established inverted index framework through principled pruning and fine-tuning techniques. We believe that our optimized approach holds promise for enhancing the effectiveness and scalability of information retrieval systems across diverse domains.

***K*eywords** Deep Learning · Neural Inverted Index · DSI · Lightweight · Pruning

1 Introduction

The realm of Information Retrieval (IR) has always been focused on striving for speed and efficiency, while retaining accuracy. Historically, traditional approaches performed first indexing of documents and then querying of relevant information. This paradigm is called index-then-retrieve. However, the Neural Inverted Index paradigm was proposed as an innovative alternative to seamlessly integrate the distinct two phases of precedent solutions within a single framework. The core of this groundbreaking approach lies in the idea of Differentiable Search Index (DSI), a solution encapsulating the entire corpus of information within a Transformer language model. The main goal of the Neural Inverted Index is to develop a unified model replicating the behavior of a conventional index and performing enhanced retrieval by leveraging the power of neural networks. In practice, given a query, the model is trained to return a ranked list of document IDs. Starting from the innovative design and implementation of DSI, our approach targets to explore the pruning technique to improve the performance of Neural Inverted Index in information retrieval tasks.

2 Methodology

In this section we present the methodology of our work. We first present our adaptation of MS MARCO dataset, then we describe our baseline model idea and finally we propose the relative innovation technique. The complete Python code (compliant with PEP8 guidelines) has been embedded in a Google Colab [2] notebook.

2.1 Dataset

The MS MARCO (Microsoft MAchine Reading Comprehension) [1] is a collection of datasets focused on deep learning in search. The first dataset was a question answering dataset featuring 100,000 real Bing questions and a human generated answer. Since then, different datasets were released. We accessed the MS MARCO dataset through the Pyserini toolkit, the Python interface to Anserini, designed to support reproducible IR research. Specifically, we chose the 0.12.0 release [3], adapting the BM25 baseline for the MS MARCO passage ranking task to our needs. The queries are already distributed in Pyserini. We initialized a searcher with a pre-built index, which Pyserini will automatically download, in order to search for a specific query with associated ranked document IDs. So, we selected the most relevant K document IDs for each query of the dataset, as well as the first 1000 document IDs by relevance for the Recall@1000 metric computation, used only for this purpose and not for training. Since we used the Hugging Face T5 tokenizer [4], we tokenized our dataset accordingly by first extracting the maximum sequence lengths for encoder and decoder and then by applying the T5 tokenizer with the proper padding to maximum length. Finally, the dataset was splitted into 70% training set, 10 % validation set and 20% test set.

2.2 Baseline

As our starting point, we partially re-implemented the solution proposed in [5]. We performed the fine-tuning of a pre-trained T5 model, in order to adapt it to our task: given a query *q*, the model should output a ranked list of document IDs. T5, originally presented in [6], is an encoder-decoder model pre-trained on a multi-task mixture of unsupervised and supervised tasks and for which each task is converted into a text-to-text format. The pretraining includes both supervised (converted tasks from GLUE and SuperGLUE benchmarks) and self-supervised training (using corrupted tokens). Relative scalar embeddings are used. We used the Hugging Face implementation of T5 [4], with the provided initial weights.

The T5 model was included in a PyTorch Lightning module, useful to define our specific training, validation and testing logic. We handled the aggregation logic of losses and metrics. T5 uses the regular cross-entropy loss, and we used Adam optimizer for training our model (with betas fixed to 0.9 and 0.98). The training hyperparameters we have considered are mainly the number *K* of retrieved document IDs for each query, the *batch size*, the number of *epochs*, the *learning rate*, the *epsilon* associated to the optimizer and the chosen T5 model.

Our objective is to obtain a lightweight DSI model, so we decided to progressively switch towards lighter versions of T5 provided by Hugging Face. Furthermore, since we used the free version of Google Colab, our runtime environment was constrained to fixed available resources; this fact justifies our approach. Indeed, we experienced crashes due to resource saturation by conducting some tests on the *t5-large* version of T5 model. Hence, we switched to *t5-base* model and we first implemented an early stopping mechanism monitoring the validation loss. However, we noticed that the metrics performance on validation set was increasing without early stopping, i.e. by letting the model overfit. The continuous improvement of metrics performance, and specifically the decrease of the validation loss at some point, as showed in section 3, suggested the possibility of an epoch-wise double descent effect [7]: we expect this to happen while training in a much less constrained environment. In section 3, we show the best tested configuration for *t5-base* baseline, from which we started our innovation experiments. However, as we will see in section 2.3, in which we are going to explain the issues occurred in the innovation of *t5-base*, we finally switched to *t5-small*, without any early stopping mechanism, and we performed *hyperparameter tuning* in order to obtain the best metrics performance on the baseline. The experiments we have conducted and the results we have obtained are available in section 3.

2.3 Innovation

Our approach employs transfer learning on a pre-trained T5 transformer model. Our baseline is the result of a first fine-tuning of the whole model. Our proposed innovation is to provide a model compression technique on this baseline in order to use this model in a resource constrained environment, showing promising future improvements.

Since the *t5-large* baseline was not supported by the Colab runtime, we first implemented unstructured pruning on *t5-base* model, but we experienced RAM saturation and the consequently runtime crash. Indeed, removing individual weights from the network can pose challenges for PyTorch Lightning in terms of performance. The impact on performance can vary depending on several factors, such as the magnitude of pruning, model architecture and hardware resources. In our context, we experienced runtime resource saturation even for a pruning rate of 0.2, so it suggested a greater influence of model complexity and less powerful hardware. In fact, complex models with many layers and parameters may experience more significant performance degradation compared to simpler models; similarly, more powerful hardware may mitigate some of the performance issues associated with unstructured pruning. These considerations are the reasons behind the baseline final switch to *t5-small* model without early stopping. So, the best values for baseline hyperparameters constitute the starting point for our model compression procedure, from which we have obtained a lighter model and a better performance. The experiments we have conducted and the results we have obtained are all available in section 3.

2.4 Metrics

We assessed our model using the following metrics:

- **MAP (Mean Average Precision)**, used in IR to evaluate the effectiveness of search systems. To implement it, we started from *Precision@K*, a measure of relevancy calculated as the number of relevant retrieved document IDs divided by the total number of retrieved document IDs, so K. Hence, we defined the *Average Precision (AP)* as the average of precision values calculated at the points in the ranking where each relevant document is retrieved. Finally, the MAP is consequently defined as the mean of the average precision scores from a set of queries.
- **Recall@1000**, indicating the proportion of relevant document IDs found in the top-1000 results.

3 Results

In this section, we will briefly go over the workflow explained in section 2 once again, in order to present the main results we have obtained, by showing some plots and tables summarizing our contribution. We omit discussing the test we have conducted by means of the *t5-large* model: indeed, as explained before, the Google Colab runtime limitations were too strict for this model to work properly.

The first test we have conducted was the hyperparameter tuning on the *t5-base* model; our aim was to understand the hyperparameters leading to the best performance in terms of the Mean Average Precision (MAP) and Recall@1000 metrics. The most notable results we have obtained are summarized in table 1.

Table 1: Baseline *t5-base*

K	Batch size	Epochs	Learning rate	Patience	Test loss	MAP	Recall@1000
25	16	30	0.001	5	3.514	0.00100	0.0
20	16	30	0.001	5	4.628	0.00135	4.00E-05
5	8	20	0.001	∞	8.323	0.00352	5.00E-05
5	**8**	**23**	**0.001**	∞	**8.604**	**0.00563**	**0.00011**

As we can see from table 1, increasing the value of K makes the model learn slowly; therefore we have decided to reduce it. We can also see that, despite the higher test loss, both the metrics have reached a higher value compared to the cases when the training stopped in advance due to the early stopping mechanism; just have a look at the last two rows of table 1. This fact leads to the hypothesis by which as the model keeps overfitting, despite the increasing test loss, the metrics keep increasing too; therefore, this could result in an epoch-wise double descent effect [7]. See also figure 1. Obviously, in order for us to verify this hypothesis, we should work in a less constrained environment, where we could conduct more accurate experiments.

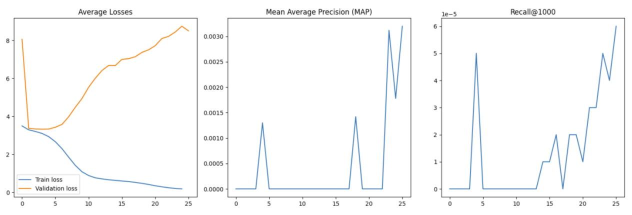

Figure 1: Plots of the best baseline *t5-base* result.

Given the promising results we have obtained, we tried to apply the methodology we have proposed in this paper. Specifically, we have used the same hyperparameters that are highlighted in table 1, and we have applied a **train-prune-recovery** approach: we have trained the baseline for 10 epochs, then we have pruned the model with a pruning rate of 0.2, and finally we have fine-tuned the model on the same dataset for other 15 epochs. While conducting this experiment, we faced some memory limitations imposed by the Google Colab runtime; therefore, we couldn't complete the execution. As we have mentioned in section 2, this out-of-memory behavior is due to the fact that pruning makes the model sparser by removing parameters. However, most deep learning frameworks and hardware architectures are optimized for dense computations. So, even though the model itself is smaller, the sparse representation might not lead to proportional savings in memory during fine-tuning because the computations often require the dense representation.

The above considerations lead us to perform hyperparameter tuning on the *t5-small* model; once again, our aim was to understand the hyperparameters leading to the best performance in terms of the Mean Average Precision (MAP) and Recall@1000 metrics. The most notable results we have obtained are summarized in table 2.

Table 2: Baseline *t5-small*

K	Batch size	Epochs	Learning rate	Patience	Test loss	MAP	Recall@1000
25	8	20	0.001	∞	8.558	0.0	0.0
10	8	50	0.001	∞	11.197	0.00100	0.0
5	**8**	**25**	**0.001**	∞	**9.339**	**0.00119**	**2.00E-05**

Table 2 shows similar results in terms of performance as table 1, indicating the correctness of our approach in performing hyperparameter tuning. Obviously, since the size of the model is smaller, we expect a decreasing performance given the same hyperparameters. See also figure 2: we can see a much noisier improving of the metrics.

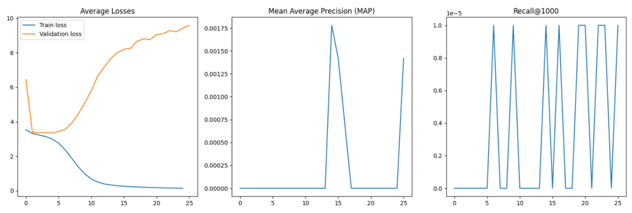

Figure 2: Plots of the best baseline *t5-small* result.

The above results lead us to perform pruning on the *t5-small* model, in order to see which kind of performance we could expect on an even smaller model. Here, we have not performed a complete hyperparameter tuning, since we have already found what we think is the best configuration for our task. We only did a partial hyperparameter tuning, in order to find the pruning rate that leads to the best performance. The most notable results we have obtained are summarized in table 3.

Table 3: Pruned *t5-small*

K	Batch size	Epochs (total)	Learning rate	Patience	Pruning rate	Test loss	MAP	Recall@1000
5	8	25	0.001	∞	0.1	9.596	0.00057	2.00E-05
5	8	25	0.001	∞	0.15	9.710	0.00071	1.00E-05
5	**8**	**25**	**0.001**	∞	**0.2**	**9.901**	**0.00214**	**2.00E-05**
5	8	25	0.001	∞	0.25	10.207	0.00062	1.00E-05
5	8	25	0.001	∞	0.3	10.352	0.00211	2.00E-05
5	8	25	0.001	∞	0.4	10.464	0.00071	1.00E-05
5	8	25	0.001	∞	0.5	10.768	0.00167	1.00E-05

Table 3 shows that the pruning rate leading to the best performance is 0.2. Figure 3 shows a comparison between the plots of the best result we have obtained in table 3. As we can see from tables 2 and 3, the *train-prune-recovery* cycle we propose in this paper is actually improving the performance of the model with respect to the baseline, proving that our approach can be valid also in a less constrained environment.

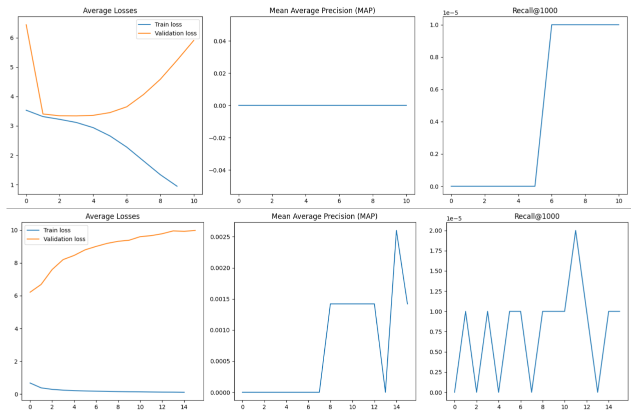

Figure 3: *Top*: fine-tuned *t5-small* baseline without pruning. *Bottom*: fine-tuned *t5-small* pruned model.

4 Conclusion

In this paper, we have presented a methodology for enhancing the efficiency of deep learning models in search tasks over the MS MARCO dataset. Through our experiments, we have demonstrated the effectiveness of our approach in improving model performance in a highly constrained environment. Our experiments have shown promising results, indicating that our approach can significantly improve the efficiency of DSI models in resource-constrained environments. Future work could explore further optimization techniques and evaluate the applicability of our approach in real-world scenarios.

References

[1] Microsoft. MS MARCO. `https://microsoft.github.io/msmarco/`, 2016. Accessed on March 2024.

[2] Google LLC. Google colaboratory. `https://colab.research.google.com`, 2019. Accessed on March 2024.

[3] PyPI. Pyserini. `https://pypi.org/project/pyserini/0.12.0/`, 2021. Accessed on March 2024.

[4] Hugging Face. T5. `https://huggingface.co/docs/transformers/model_doc/t5`. Accessed on March 2024.

[5] Yi Tay, Vinh Q. Tran, Mostafa Dehghani, Jianmo Ni, Dara Bahri, Harsh Mehta, Zhen Qin, Kai Hui, Zhe Zhao, Jai Gupta, Tal Schuster, William W. Cohen, and Donald Metzler. Transformer memory as a differentiable search index, 2022.

[6] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[7] Cory Stephenson and Tyler Lee. When and how epochwise double descent happens. *CoRR*, abs/2108.12006, 2021.