

Augmenting Fast-Forward Indexes: Dynamic Hybrid Re-Ranking with Parameter-Efficient Adaptors

Master Thesis
Adrian Segura Lorente

Delft University of Technology



Augmenting Fast-Forward Indexes: Dynamic Hybrid Re-Ranking with Parameter-Efficient Adaptors

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Adrian Segura Lorente

born in Celaya, Guanajuato, Mexico

Thesis Committee:

Chair, Supervisor:

Supervisor:

Committee Member:

Project duration: December 2024 - September 2025

Faculty: Faculty of EEMCS, TU Delft

Prof. Dr. Avishek Anand, TU Delft

Dr. Jurek Leonhardt, TU Delft

Prof. Dr. Julián Urbano, TU Delft



Web Information Systems Research Group
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Augmenting Fast-Forward Indexes: Dynamic Hybrid Re-Ranking with Parameter-Efficient Adaptors

Author: Adrian Segura Lorente
Student id: 5011094

Abstract

Transformer-based architectures have significantly advanced the field of Information Retrieval (IR) by enabling semantic understanding that surpasses traditional term-frequency models. Hybrid approaches, which combine efficient sparse retrievers like BM25 with effective dense retrieval methods are becoming more popular for balancing performance and efficiency. However, the application of advanced hybrid systems often requires dedicated infrastructure and technical expertise, as existing toolkits are primarily research-oriented and not designed for simple integration.

To bridge this gap, this work introduces an open-source Elasticsearch plugin that implements the interpolation-based re-ranking framework from Leonhardt et al. [23] This plugin simplifies the adoption of hybrid re-ranking by making the technique easily accessible within existing pipelines.

Furthermore, this work explores learning-based optimization methods to optimize the interpolation constant α . The research investigates two main strategies: (1) direct, gradient-based optimization to determine an optimized global value for α , and (2) the development of lightweight adaptor models that dynamically predict context-aware α values for each query or query-document pair. These adaptors are based on Feedforward Neural Networks and Neural Tensor Network architectures.

Empirical results suggest that these learning-based methods, particularly the dynamic adaptors, can outperform traditional grid search methods for tuning α , due to their dynamic adjustment in a query/document basis. The benefits were especially strong in out-of-domain scenarios, where the adaptors showed improved performance for encoders that were not pre-trained on the target domain, without requiring fine-tuning of the full backbone architecture.

This work delivers a practical, easy-to-use tool for hybrid re-ranking in Elasticsearch, a novel methodology for optimizing its core re-ranking parameter, and proposes adaptor models that can deliver better performance than a fixed interpolation value in hybrid re-ranking.

Preface

This thesis marks the culmination of my Master's degree in Computer Science at TU Delft. My time at this institution, and all of its exceptional professors and members, have given me a life-changing view of science, and about its essential importance in our society. Furthermore, I am immensely grateful for the rigorous, high-quality, and inspiring academic environment fostered by the dedicated faculty at TU Delft, which has been essential to my professional and personal growth during this period.

My specific interest in this topic started with the Natural Language Processing and Information Retrieval courses, taught by Prof. Dr. Avishek Anand. The passion and clarity with which he delivered the material motivated me to pursue this area for my thesis research. I am grateful that he suggested the foundational topic for this project and agreed to supervise it.

I would like to express my most sincere gratitude to both of my supervisors: Dr. Jurek Leonhardt, and Prof. Dr. Avishek Anand, for their kind and thorough guidance, patience, and understanding. Their consistent support and exceptional mentoring have been invaluable throughout this entire process. I have truly learned a great deal from both of you.

I extend my heartfelt thanks to my family and my girlfriend for their steadfast support and encouragement. This has been one of the most challenging projects that I have undertaken, but also one of the most fulfilling. This journey would not have been possible without you.

I finalize this stage filled with gratitude and motivation for the challenges to come.

Adrian Segura Lorente
Delft, the Netherlands
September 1, 2025

Contents

Preface	iii
Contents	v
List of Figures	vii
List of Tables	ix
1 Introduction	1
2 Background	3
2.1 Information Retrieval (IR) Fundamentals	3
2.2 Traditional IR Models	5
2.3 Neural Information Retrieval (Neural IR)	6
2.4 Re-ranking in IR	10
3 Related Work	13
3.1 Efficient Neural Ranking using Fast-Forward Indexes	13
3.2 Parameter-Efficient Fine-Tuning (PEFT)	14
3.3 Neural Tensor Networks (NTNs)	15
3.4 Relaxed Sorting Operator	15
3.5 NeuralNDCG	16
3.6 Margin Ranking Loss	16
4 Methodology	17
4.1 Elasticsearch Re-ranking Plugin	17
4.2 Auto-Alpha Tool Set	22
4.3 Overview: Elasticsearch + Auto-Alpha (RQ 3)	25
4.4 Plugin Testing	26
4.5 Direct Optimization of Interpolation Constant α	27
4.6 Adaptor Methods	30
5 Experimental Setup	37
5.1 Datasets and Baselines	37
5.2 Evaluation Details	38
5.3 Implementation Details	38
5.4 Model Training Details	39

5.5	Elasticsearch Plugin Testing Setup	39
5.6	Code Implementation	41
6	Results and Discussion	43
6.1	Direct α optimization & Adaptor Models	43
6.2	Plugin Performance Testing	52
7	Conclusions and Future Work	57
7.1	Conclusion	57
7.2	Future Work	58
7.3	Disclosure	59
	Bibliography	61
A	Supporting Contents and Analysis	67
A.1	Full Relative Scores	67
A.2	Plugin Experimentation	68

List of Figures

2.1	Graphical representation of an encoder block, adapted from Vaswani et al. [43].	7
2.2	Graphical representation of a cross-encoder. The query and document are fed together. Depending on the architecture they can be combined using different formats.	8
2.3	Dual encoder: document and query encoder are separate models.	9
3.1	High-level diagram of the fast-forward indexes technique. Blue block highlights document embedding and collection indexing.	13
3.2	Reparameterization process in LoRA, only matrices A and B are trained, reducing learnable parameter landscape.	15
4.1	High-level system diagram of elasticsearch and the plugin.	18
4.2	Sequence diagram showcasing the execution of a query using elasticsearch and the FFWI-Rescorer plugin.	19
4.3	Class diagram of the FFWI-Rescorer plugin.	20
4.4	High-level architecture of the alpha-optimization procedure. The steps are the following: 1. The user provides a yaml configuration file. 2. The prepared dataset created using the data preparation tools is fetched from the specified paths, plus the pre-computed memory mapped embeddings are loaded. 3. The training procedure optimizes α . 4. After optimization, the final value of alpha is registered as an elasticsearch's index setting for the specified index, thereby making it readily available for use by the ES reranking plugin.	23
4.5	Class diagram of how the text encoder could be extended, and the expected types for each method/input/output.	24
4.6	High-level view of the data generation pipeline. The process is designed to run sequentially.	25
4.7	Setup for testing procedure: three dedicated instances are spin up for this process. One runs the rally testing service, one hosts the elasticsearch service as a dedicated single ES node, and a dedicated instance runs the query-encoding service, making use of GPU acceleration.	27
4.8	High-level architecture of model to directly optimize α (logit of sigmoid). The backbone encoder(s) remain non-trainable, Alpha (the logit which follows as input of a sigmoid function) remains trainable. The flame symbol represents trainable parameters.	28
4.9	Triplet assembly: k random documents sampled from original retrieved documents and used to assemble triplets together with relevant documents.	29
4.10	Adaptor models in this context have the aim of adapting pre-computed query/document embeddings and BM25 scores, agnostic to the method used to compute the inputs, and finally predict an output dense score or α value.	30
4.11	FFNN adaptor, embeddings' features are transformed by using an element-wise product and also their absolute difference. The concatenation of the transformations are used as input for FFNN 1 which outputs a dense score which is in turn combined with a BM25 score by FFNN 2.	32

4.12	NTN-based adaptor: the NTN components extracts relevant features for each bilinear interaction (B.I.) block. Then the Hadamard product and absolute differences of the query and document embeddings are combined with the features as inputs to FFNN_ϕ , and finally combined by a second FFNN.	33
4.13	NTN Dynamic Alpha adaptor. *The Hadamard and absolute difference vectors between q and d are also used as inputs for the FFNN, however this is not shown here for simplicity. . .	34
4.14	NTN Dynamic Alpha adaptor, Per-Query.*The Hadamard and absolute difference vectors between q and d are also used as inputs for the FFNN, however this is not shown here for simplicity.	35
4.15	FFNN Dynamic Alpha adaptor, Per-Query.	36
6.1	Relative gain (%) of each metric, for the best-performing methods over the Hyperparameter Search baseline for each encoder and dataset. Gains are computed as $\frac{\text{Best-Hyperparam}}{\text{Hyperparam}} \times 100$. .	46
6.2	Training time in seconds (x-axis) and the resulting nDCG@10 for each encoder-model combination A) TREC-DL-2019 PSG'), B) ANTIQUE. Colors represent a family of encoder models used for the query and document dense embedding representations. Each marker represents a specific model architecture / technique. For TREC, training times for adaptor models were at least approximately $19\times$ lower than the alpha grid-search process. Training times for ANTIQUE models remained lower than hyperparameter search baseline, but were considerably higher on average, due to the larger training set size.	48
6.3	Total number of trainable parameters (x-axis) and the resulting nDCG@10 for each encoder-model combination. Colors represent a family of encoder models used for the query and document dense embedding representations. Each marker represents a specific adaptor model architecture.	49
6.4	Correlation for features and predicted α values across the test split during testing, the figures on the top row correspond to correlation maps between the high-level data features and the predictions for the Document-level Dynamic Alpha models (NTN-DA), and the bottom row corresponds to correlation maps between the high-level data features and the predictions for the Query-level NTN Dynamic Alpha models.	50
6.5	Window Size (x-axis) vs. Service Time (y-axis) at the 50th, 90th, and 99th percentiles, measured across window sizes: 100, 250, 500, and 1000 top-k documents to re-rank.	53
6.6	Window size (x-axis) vs. 50th percentile service time (y-axis) for first-stage retrieval, and for re-scoring phase respectively. Measurements are shown across window sizes for 100, 250, 500, and 1000 top-k documents to re-rank.	54
6.7	Window size (x-axis) vs. mean throughput in queries per second (y-axis), for the first-stage retrieval (baseline), and for re-ranking stage. Measurements are shown across window sizes for 100, 250, 500, and 1000 top-k documents to re-rank.	54
6.8	50th, 90th and 99th percentiles for the query-encoding processing time per request.	55
A.1	Relative gain/decrease (%) of each metric, for all methods over the Hyperparameter Search baseline for each encoder and dataset. Gains are computed as $\frac{\text{Best-Hyperparam}}{\text{Hyperparam}} \times 100$	68

List of Tables

5.1	Hyperparameter Configurations for best performing models for each dataset and encoder family.	39
5.2	Hardware and Software Specifications of the Experimental Setup	40
6.1	Re-ranking performance on TREC-DL-Psg’20 and ANTIQUE (BM25 first-stage retrieval). Best results for each metric in bold denote the top performance within that backbone family. The <u>underlined</u> value indicates the single best overall result for that metric across all models. Hyperparameter search and direct alpha optimization models use a fixed global value for alpha. Such values are reported in the first column for each dataset group: α_{TREC} and α_{ANTIQUE} , respectively. Models using dynamic alpha values have alpha reported with the mean, plus-minus one standard deviation.*MRL stands for Margin Ranking Loss (direct α optimization with that objective). Baseline methods are indicated with the superscript ^B . Dagger superscripts [†] indicate statistically significant improvements over the strongest baseline for each subgroup (Hyperparameter grid search), using a paired t-test with a significance level of 95%, and adjusting for Family-Wise Error Rates using Holm-Bonferroni [42]. . . .	44

Chapter 1

Introduction

Advancements in the field of Machine Learning, particularly the transformer architecture [43], have brought important progress to the field of Information Retrieval (IR) by enabling the consideration of latent semantic features in information resources, rather than purely relying on frequency-based approaches. While simplistic, sparse retrieval techniques such as BM25 remain widely used due to their efficiency and robustness. Moreover, hybrid approaches can combine benefits from both paradigms, therefore their use has been on the rise. For example, Leonhardt et al. [23], proposed an interpolation-based re-ranking method which combines sparse and dense retrieval signals for re-ranking, achieving a performance improvement while maintaining efficiency. Despite its promise, the practical application of hybrid systems of this nature commonly require technical expertise and infrastructure that is not readily available to all users. Existing toolkits are usually research-oriented and are not designed for simple integration into existing retrieval frameworks or pipelines. To address this gap, this thesis presents an open-source Elasticsearch plugin that implements the interpolation-based re-ranking method proposed by Leonhardt et al [23]. The plugin supports efficient re-ranking by fusing BM25 and dense similarity scores through an interpolation parameter α , which controls their relative contributions.

This work also experiments with methods to automatically optimize the interpolation constant that defines the final weighted query-passage score, by using gradient-based optimization methods. Experimentation is carried out with methods that directly optimize the interpolation constant, attempting to find an optimal global value. Furthermore, it is hypothesized that a fixed global value of α may limit re-ranking effectiveness. This work explores whether this is the case, and to address this limitation, four adaptor model architectures that dynamically predict α interpolation values are proposed. Such models are based on Feedforward Neural Networks, and Neural Tensor Network architectures [41]. Notably, the obtained results suggest that the proposed dynamic, context-aware adaptors are specially beneficial for out-of-domain scenarios.

The main research questions addressed by this study are the following:

- **RQ 1** How can learning-based methods optimize the interpolation weight α in hybrid re-ranking?
- **RQ 2** Can lightweight adaptors improve performance when using pre-computed embeddings from non-domain-specific encoders?
- **RQ 3** Can a native Elasticsearch (or equivalent framework) plugin/tool be developed to implement the Fast-Forward Indexes approach while making its setup simple? Moreover, how can insights from RQ1 be applied to this implementation?

Chapter 2

Background

2.1 Information Retrieval (IR) Fundamentals

The field of Information Retrieval deals with the task of identifying and retrieving relevant information resources based on a given information requirement. The most common tasks involve querying a large collection of items to retrieve relevant resources in an efficient and effective manner.

Ad-hoc retrieval

Ad-hoc retrieval is the task of retrieving relevant information resources based on a user's query given in a natural language format.

Relevance

Given a set of retrieved information resources, relevance defines how well do those resources meet the user's information requirement. Relevance labels can vary in granularity, depending on the task at hand. For instance, relevance labels can be binary, or graded in a given non-binary numerical scale.

2.1.1 Evaluation Metrics

In order to evaluate the performance of information retrieval systems, evaluation metrics are necessary. A variety of metrics can be used to evaluate different aspects of an information retrieval system. Relevant metrics to this work are described in the subsections below.

Normalized Discounted Cumulative Gain (nDCG)

The discounted cumulative gain (DCG) can be used to determine the quality of the ranking of a set of resources retrieved by an IR system up until a given cutoff of retrieved resources $k \in \mathbb{Z}^+$. The DCG@k attains a higher value whenever resources appear in the right order, according to their relevance (rel). Furthermore, this metric penalizes documents that appear out of order later in the ranking, attaining a lower value the more out-of-order the results are positioned. This occurs since the denominator grows with each ranking consecutive index $j \in \{1, \dots, k\}$. The numerator remains constant given the resource's relevance. The DCG@k function, defined in equation (2.1).

$$\text{DCG}@k = \sum_{j=1}^k \frac{2^{\text{rel}_j} - 1}{\log_2(j + 1)} \quad (2.1)$$

nDCG is a normalized version of DCG, defined in equation (2.2), where maxDCG is the optimal

(maximum) attainable DCG@k if documents were sorted by their relevance scores. This normalized version is useful to define a common scale across comparisons.

$$\text{nDCG@k} = \frac{\text{DCG@k}}{\text{maxDCG@k}} \quad (2.2)$$

Precision

Given a query, and a set of retrieved documents, the precision is defined as the proportion of retrieved documents that are relevant. More formally:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (2.3)$$

That is, the ratio of true positives (TP), out of those true positives and the false positives (FP), which are retrieved documents that are not relevant.

Recall

The recall or true positive rate is a metric that denotes the proportion of all relevant documents that were successfully retrieved out of the whole pool of relevant documents. More formally:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (2.4)$$

That is, the ratio of true positives (TP), out of the sum of true positives and false negatives (FN). False negatives are relevant documents that were not retrieved by the system.

Mean Average Precision (MAP)

MAP evaluates the quality of ranked retrieved elements when there may be multiple relevant documents for a given query. It is the mean of the average precision (AP) across multiple queries, where AP is defined as follows:

$$\text{AP} = \frac{1}{R} \sum_{k=1}^n P(k) \cdot \text{rel}(k) \quad (2.5)$$

Where R is the total number of relevant documents, n is the total number of retrieved results for a given query, $P(k)$ is the precision up until rank k , and $\text{rel}(k)$ is a function evaluating to 1 if the retrieved document at that rank k is relevant, 0 otherwise.

Finally, MAP is defined as the mean of AP across a set of queries and their retrieved documents. More formally, given Q queries, and their respective AP scores:

$$\text{MAP} = \frac{1}{Q} \sum_{q=1}^Q \text{AP}_q \quad (2.6)$$

Mean Reciprocal Rank (MRR)

Reciprocal rank measures how early the first relevant document appears within a set of retrieved documents. The earlier in the ranking a relevant document is positioned, the higher the metric. The reciprocal rank for a given query q is defined as follows:

$$RR_q = \frac{1}{rank_q} \quad (2.7)$$

Where $rank_q$ is the position in the ranking where the first relevant document occurs for query $q \in Q$. Finally, MRR is defined as the mean of the reciprocal rank of a set of queries:

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank_q} \quad (2.8)$$

2.2 Traditional IR Models

IR models are used to retrieve information resources based on their features, both inherent and derived. Traditional IR models make use of statistical features such as term frequency, inverse document frequencies, probabilistic features, etc. The subsection below introduces two traditional and common approaches to IR models: TF-IDF and BM25.

2.2.1 Term Weighting Schemes

TF-IDF

Term Frequency-Inverse Document Frequency, as its name implies makes use of the term frequency and the inverse document frequency statistics. The term frequency denotes the proportion of the frequency of a term for a given document with respect to its total terms. That is:

$$tf(d,t) = \frac{f_{t,d}}{f_{t',d} \in d} \quad (2.9)$$

Where d is a document, t the term, and $t' \in d$ denotes any term within d .

Moreover, inverse document frequency considers how rare terms are. It is the ratio between the total number of documents in a given set of documents D , and the amount of those documents $d \in D$ containing the given term. This ratio incorporates a logarithm to dampen its scale which can grow large with large sets:

$$idf(t,D) = \log \frac{|D|}{|\{d \in D : t \in d\}|} \quad (2.10)$$

Finally, the scoring of a term within a given document $d \in D$ can be obtained by combining tf and idf into the TF-IDF definition:

$$tf-idf(t,d,D) = tf(t,d) \cdot idf(t,D) \quad (2.11)$$

2.2.2 Probabilistic Models

BM25

Best Matching 25 (BM25) [39] is a bag-of-words retrieval model that makes use of term-frequency and inverse document frequency to rank a set of documents with respect to a query. Given a query-document pair (Q, d) , where the query contains terms $q \in Q$, the BM25 score is defined by the equation below:

$$\text{BM25}(d, Q) = \sum_{q \in Q} \text{idf}(q, D) \cdot \frac{\text{tf}(q, d) \cdot (k_1 + 1)}{\text{tf}(q, d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{\mu_{dl}})} \quad (2.12)$$

Where $|d|$ represents the document's length, and μ_{dl} is the average document length across the collection. Moreover, k_1 and b are free scalar parameters.

2.3 Neural Information Retrieval (Neural IR)

Neural information retrieval concerns the application of neural models and architectures to information retrieval tasks [31]. In this paradigm, techniques are commonly used to represent documents and queries with learned continuous vector representations (embeddings), rather than sparse representations like in traditional IR approaches. These representations can be learned automatically, mapping documents and queries to low-dimensional vectors in latent space [48]. Embedding representations learned with neural models allow to model semantic features of documents and queries [31], which makes them powerful for information retrieval tasks and sometimes more robust than traditional IR techniques.

There exist a variety of techniques to produce dense embeddings, however we focus on transformer-based encoder models.

2.3.1 Dense Retrieval

In the dense retrieval paradigm, queries and documents' dense vector representations are used to compute a relevance score through a similarity function [48]:

$$\phi_D(q, d) = f_{\text{sim}}(\zeta(q), \eta(d)) \quad (2.13)$$

In the context of dense retrieval, $\zeta \in \mathbb{R}^l$ and $\eta \in \mathbb{R}^l$ are commonly neural network-based models that encode queries q , and documents d , respectively.

Encoder Models

Encoder models are commonly deep neural networks based on the transformer architecture. They are used to encode a given input sequence, such as a text from a query or paragraph, and transform it into a contextualized vector representation that captures the sequence's semantic information. These representations can be used for more accurate contextualized matching, without the need of explicit lexical overlap.

Transformer Models & Attention

Vaswani et al. [43] introduced encoder and decoder components for their transformer architecture, however for brevity this section is kept to the encoder portion only.

The encoder portion of a transformer model is composed of multiple encoder blocks, where each block contains: a Multi-head self-attention layer, a Residual and Layer Normalization component, and a Per-Item Feedforward Neural Network. Figure 2.1 shows this sequence.

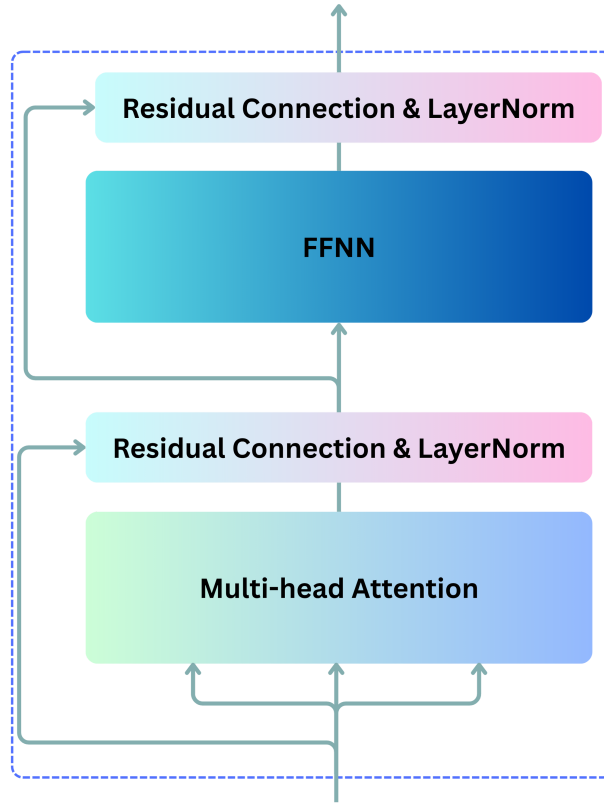


Figure 2.1: Graphical representation of an encoder block, adapted from Vaswani et al. [43].

The role of the multi-head self-attention is to compute the attention interactions between the input tokens to the encoder block. The attention operation is defined in equation 2.14 [43], where $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$, $\mathbf{K} \in \mathbb{R}^{n \times d_k}$, and $\mathbf{V} \in \mathbb{R}^{n \times d_v}$ are matrices obtained from mapping all n input tokens having a corresponding word embedding of dimension d by using projection matrices $W^Q \in \mathbb{R}^{d \times d_k}$, $W^K \in \mathbb{R}^{d \times d_k}$, and $W^V \in \mathbb{R}^{d \times d_v}$, which are learnable matrices. The idea is that each of these projections abstracts different useful features from the input token embeddings, which are then interacted by multiplying the query and key matrices $\mathbf{Q}$ and $\mathbf{K}$, and further normalizing the outputs by $\sqrt{d_k}$ and using softmax per row to attain a set of normalized weights that add up to 1.0, thereby weighting the contribution of each value projection from $\mathbf{V}$ in a final aggregation. This produces the final contextualized representation for the self-attention mechanism per token. Such attention process occurs across multiple parallel attention blocks, denoted heads, where each head has its own set of learnable key, query, and value matrices. Hence the name "Multi-Head Attention".

In order to compute the final combined contextualized representations for each input token, the embedding outputs of each head are concatenated. That is, given the input sequence of token embeddings $s \in \mathbb{R}^{n \times d}$, and the contextualized outputs from h heads $H \in \mathbb{R}^{h \times n \times d_v}$, the final aggregation is done through concatenation of each token embedding's representation for each head. The concatenation yields a tensor $T \in \mathbb{R}^{n \times h \cdot d_v} = \text{Concat}(\text{head}_1, \dots, \text{head}_h)$, where $\text{head}_i \in H$. The concatenation is finally projected by a learnable matrix $W^O \in \mathbb{R}^{h \cdot d_v \times d}$ which maps each vector to its final representation back to its original dimension d , this process is formally described in equation 2.15 [43].

The resulting embeddings undergo addition with a residual from before the attention mechanism, followed by layer normalization. As a final step in the encoder block, the resulting vectors go through a feedforward neural network, and each output is again combined with a residual (from before the FFNN) and layer

normalized.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right) \mathbf{V} \quad (2.14)$$

$$\text{Multi-Head}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O \quad (2.15)$$

2.3.2 Cross-Encoders

Cross-encoders are neural network architectures that have two simultaneous inputs, such as a document and a query. The given inputs are processed together to model richer interactions. An example is shown in figure 2.2, where the query and the document are given together as an input. For instance, in BERT models, two inputs can be given, separated by a special token [SEP]. Moreover, at the embedding level, "segment embeddings" are used to encode whether tokens come from sentence A, or B [9]. It is common for the outputs of a cross-encoder to be pooled in some way in order to be subsequently used in tasks such as classification, regression, etc. For instance, BERT models have a special token [CLS] prepended to the given input, whose output aggregates global level information and is therefore used for such tasks.

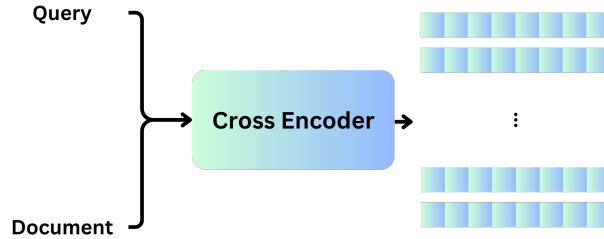


Figure 2.2: Graphical representation of a cross-encoder. The query and document are fed together. Depending on the architecture they can be combined using different formats.

2.3.3 Dual-Encoders

Dual-Encoders are relevant architectures for the IR domain, specifically for Dense Passage Retrieval (DPR). Unlike cross-encoders, dual-encoders de-couple the query and document encoding processes by having a separate encoder model for each purpose. That is, there is a query encoder model $\zeta(\cdot)$ and a separate document encoder $\eta(\cdot)$. The encoders could be a siamese neural network, but they don't need be, they could also be different. This can be appreciated graphically from figure 2.3.

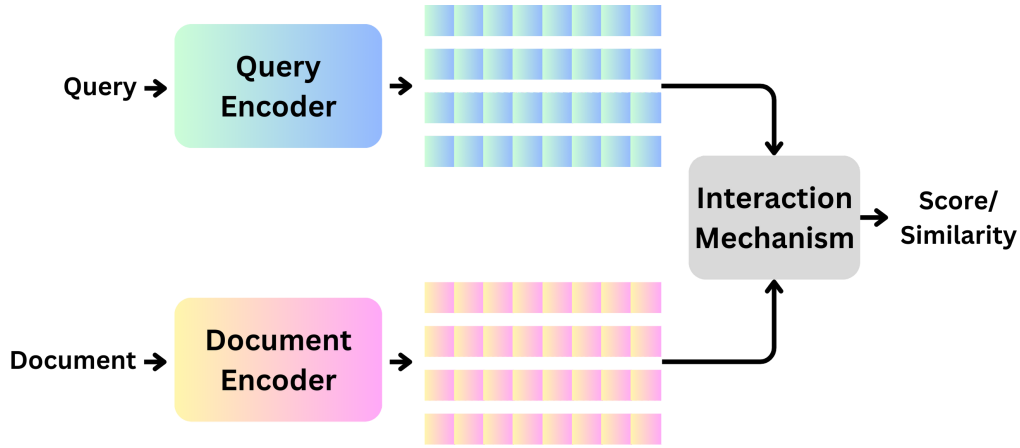


Figure 2.3: Dual encoder: document and query encoder are separate models.

Similarity between query and document embeddings is computed through an external operator such as dot-product.

The output embedding representations of the query and document encoders can be pooled, but not necessarily.

The main advantage of separating the document and the query encoders is that documents can be pre-computed and stored for instance for dense retrieval tasks. Moreover, the query does have to be computed every time, since queries can change during real retrieval tasks such as during web-search, etc.

2.3.4 Relevant Model Architectures

BERT

Bidirectional Encoder Representations from Transformers (BERT), was introduced by Devlin et al. [9], where they proposed a way to train deeply bidirectional representations of words, improving contextual understanding. Models previous to BERT did not commonly encode text representations from both directions: left and right. While previous models like ELMO [34] did use bi-directional LSTM architectures to process text, the integration of the encoded context was done until the end. BERT on the other hand is a transformer-based architecture, therefore it allows for tokens to attend to each other at every encoder block, making it bi-directional naturally. BERT is an encoder model based on the architecture proposed by Vaswani et al. [43], however the decoder portion of the transformer is not used.

BERT is pre-trained on two tasks [9]:

1. Masked Language Modeling (MLM), or Cloze: Randomly masks approximately 15% of tokens in the input and trains the model to predict them based on their context. This task allows for the model to learn bidirectional context.
2. Next Sentence Prediction (NSP): Trains the model to understand sentence relationships by predicting whether a given sentence follows from another. During pre-training two sentences are used, A and B. They are separated by the special [SEP] token, and also have this detail specified with the segment embeddings. In NSP, 50% of the time sentence B follows after A, in the rest of the occasions it is a random sentence from the corpus. This pre-training method was shown to not be useful, and even harmful in some other ablations, such as in the work by [28].

BERT initially came in two main variants:

1. BERT base: featuring $L = 12$ encoder blocks, $A = 12$ attention heads per block, and hidden dimension of vector embeddings of $H = 768$.
2. BERT large: featuring $L = 24$ encoder blocks, $A = 16$ attention heads per block, and hidden dimension of vector embeddings of $H = 1024$.

ColBERT

In traditional dense IR techniques, query/document representations are aggregated into a single vector, for instance in BERT, the [CLS] token’s output embedding. However, this paradigm can lead to loss of semantic details. Khattab et al. [21] introduced ColBERT to alleviate this issue by using a late interaction mechanism they denominated ”MaxSim”. MaxSim is defined by equation (2.16)[21], where E_q is the set of output token embeddings from the query encoder and E_d is the set of output token embeddings from the document encoder.

$$S_{q,d} = \sum_{i \in [|E_q|]} \max_{j \in [|E_d|]} E_{q_i} \cdot E_{d_j} \quad (2.16)$$

Contriever

Contriever is a dense retrieval model introduced by Izacard et al. [17]. This model is trained in an unsupervised manner, which makes it flexible and robust to few-shot and zero-shot domains. This model is trained by using the contrastive learning objective InfoNCE, defined in equation (2.17) [17], where q is a given query representation, k_+ is a positive document representation, and each k_i denotes a negative document. Moreover, $s(\cdot)$ is a scoring function that assigns a score to the query-document pair (q, k) . This loss objective is minimized, thus optimization pushes for negative documents to be assigned a lower value, and for positive documents to attain a higher score. Furthermore, they generate positive examples by employing a technique denoted inverse cloze. This technique samples random a span of tokens from a given text/passage. This span is used as a query, while the complementary span, that is, everything surrounding than span of text, is taken as the positive pair [17]. As for negative documents, they use in-batch negatives and also across batch negatives [17].

$$\mathcal{L}(q, k_+) = -\log \left(\frac{\exp(s(q, k_+)/\tau)}{\exp(s(q, k_+)/\tau) + \sum_i^K \exp(s(q, k_i)/\tau)} \right) \quad (2.17)$$

TAS-B

Hofstätter et al. [13] introduce the Topic Aware Sampling (TAS) technique whereby they cluster queries from an information retrieval training dataset. They do so by using k-means [18], resulting into topical clusters. They train a dense retrieval model, for which during its training process, instead of sampling randomly from the whole dataset for each training batch, queries are sampled from a single cluster, preserving topic-specific information for the batch. This step creates batch elements which are related. Moreover, in-batch negatives are used, combined with an extra step where selected passage pairs are balanced for difficulty. This procedure makes challenging enough and useful for training, resulting in a more efficient training procedure [13]. Additionally, the authors develop a dual teacher supervision method where they train a lightweight BERT_{DOT} model by following the supervision signals of a ColBERT [21] and a CAT-BERT model [6].

2.4 Re-ranking in IR

Re-ranking is the process of using an initial list of retrieved documents, ranked by some retrieval model, and reordering them using a more accurate, typically more expensive model in a more optimized manner. The first stage retrieval in retrieve and re-rank settings is commonly efficiently ranked by sparse models, such as BM25. Re-ranking can be approached with a variety of techniques, varying from traditional LTR

methods, to neural re-ranking techniques that make use of cross-encoders, or dual-encoders for accurate ordering and semantic signal consideration.

2.4.1 LTR Models

LTR is formulated as a supervised learning task where given a set of queries $Q = \{q_1, \dots, q_n\}$, there is a set of associated documents/passages $X = x_1, \dots, x_n$ for which a ground truth relevance label $y_1, \dots, y_n$ is given. The goal of LTR is to learn a function f that can output scores for each element such that the resulting ordered ranking is optimal. LTR methods can be divided into the following types [22]:

- **Point-wise LTR:** Commonly aims to learn a function that predicts a relevance score for a given query-document item. Is treated as a regression task often.
- **Pair-wise LTR:** Aims to learn a function that can score pairs of documents with respect to a query. That is, given a training example with documents A and B , the task can be treated as a binary classification task, whereby if $s(A) > s(B)$ the label is 1 or 0 otherwise.
- **List-wise LTR:** Directly optimizes on a set of documents with respect to a query q . That is, given q and its set of related documents $X = \{x_1, \dots, x_n\}$, the model learns a function that predicts a score s_i , $1 \leq i \leq n$ for each document in the set X . The resulting score list is used to rank the documents in descending order and used as part of a list-wise loss objective, such as Neural nDCG [35].

2.4.2 Neural Re-Ranking

Neural re-ranking methods make use of neural network architectures to score and re-rank a set of given documents. Two paradigms of neural re-ranking are introduced below.

Cross-encoders

Cross-encoders provide high quality neural re-ranking performance at the cost of computational resources and latency, since cross-encoding requires computing the encoded representations for both the documents and queries to be re-ranked, on-the-fly, as described in section 2.3.2. An example of a popular architecture used as a cross-encoder is BERT, which can be used to encode the query and its prospective documents, only to be re-ranked afterwards based on cosine similarity by using the [CLS] token [9].

Dual-encoders

Dual encoders have been used as a more efficient alternative to cross-encoders, while retaining a good re-ranking performance. A clear example is the work by Leonhardt et al. [23], where they employ dual-encoders to pre-compute document embeddings, only to efficiently retrieve the stored representations at retrieval time to perform scoring through a convex combination of the dot product between query and pre-computed document embeddings, and the BM25 score for each query-document pair. Section 3.1 provides more details about this technique.

Chapter 3

Related Work

3.1 Efficient Neural Ranking using Fast-Forward Indexes

Leonhardt et al. [23] explored the usage of dual-encoders to develop an efficient and high-quality re-ranking method. They achieve efficiency by allowing for the pre-computation of document embedding representations, which are indexed directly and can be retrieved on the fly in the same way as their respective documents. In this technique, a first stage retrieval is carried out using a sparse retrieval method, followed by a second-stage re-ranking step, by using dense scores obtained through the dot product between a query embedding and a document embedding, where the document embeddings are pre-computed for efficiency purposes and the query embeddings are computed on-the-fly. Figure 3.1 shows this process graphically.

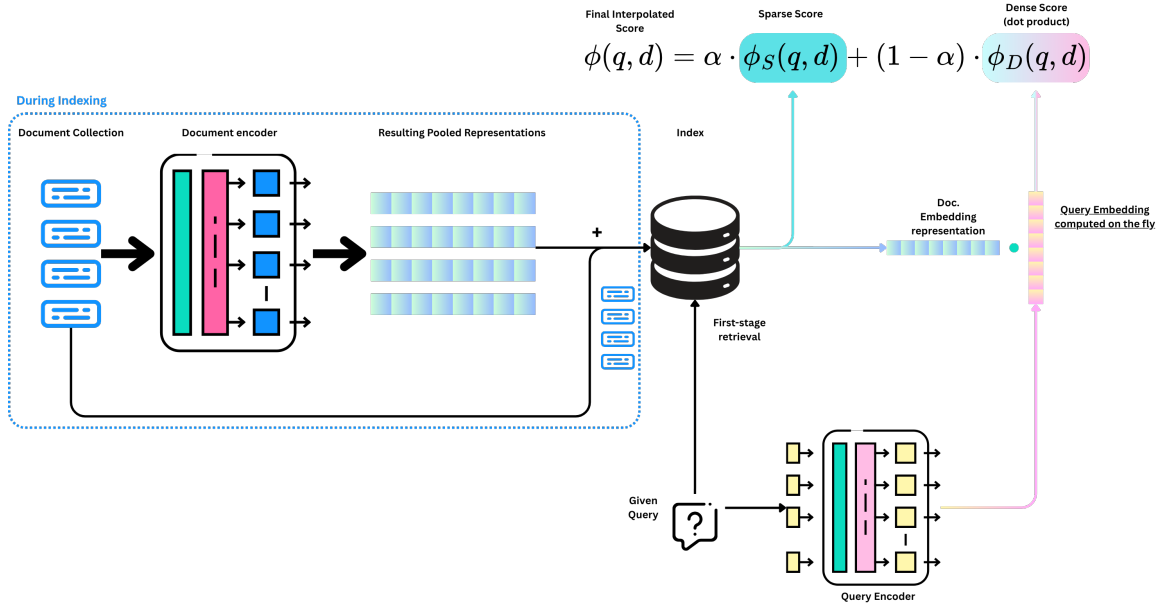


Figure 3.1: High-level diagram of the fast-forward indexes technique. Blue block highlights document embedding and collection indexing.

The computation of dense scores is described in equation (3.1), whereby function $\zeta : \mathbb{R}^{N \times k} \mapsto \mathbb{R}^a$ represents a query encoder, and function $\eta : \mathbb{R}^{M \times k} \mapsto \mathbb{R}^a$ represents a document encoder [23].

$$\phi_D(q, d) = \zeta(q) \cdot \eta(d) \quad (3.1)$$

Their formulation of the interpolated score is defined below, in equation (3.2), where ϕ_S represents the

function that computes the sparse score component of the final score [23]. This research is extended further, therefore it is used as a foundation in this work, together with their introduced notation.

$$\phi(q, d) = \alpha\phi_S(q, d) + (1 - \alpha)\phi_D(q, d) \quad (3.2)$$

3.2 Parameter-Efficient Fine-Tuning (PEFT)

Parameter Efficient Fine-Tuning (PEFT) methods have revolutionized the way pre-trained language models (PLMs) are fine-tuned to specialized tasks. PEFT methods make fine-tuning of PLMs more parameter efficient by reducing the number of model parameters to adjust during fine-tuning, thereby reducing the training memory footprint and its hardware requirements, while attaining similar performance to full-fine-tuning methods [46]. There has been research on a variety of PEFT mechanisms, such as additive fine-tuning, where additional parameters are introduced for fine-tuning. An example of this paradigm is the work by Houlsby et al. [14] where they introduce adaptor blocks consisting of a bottleneck architecture that performs down-projections, followed by projections back to the original hidden input size. The adaptors are introduced after each multi-head attention module and each FFNN module within the transformer encoder block respectively. Re-parameterized fine-tuning is also a popular paradigm within PEFT techniques, where low rank transformations are used to lower the number of trainable parameters while preserving high dimensional matrix operations [46]. A prominent technique developed based on this principle is Low-Rank Adaptation (LoRA), developed by Hu et al. [15] In this technique, the model's parameters remain frozen, while a re-parameterized representation of the model's parameters is introduced, by using rank decomposition matrices. Given an arbitrary matrix of learnable parameters $W \in \mathbb{R}^{d \times d}$ from the model to be fine-tuned, its parameters remain frozen, and during the training process two matrices $A \in \mathbb{R}^{r \times k}$ and a matrix $B \in \mathbb{R}^{d \times k}$ are trained instead, such that $r \ll d$. During training given an input vector $x \in \mathbb{R}^d$, the output of the projections from A , followed by B , result in the same output dimension d as the original matrix of weights W , thereby reducing the space of parameters to be optimized since matrices A , and B ideally have much less parameters than the full rank matrix W . Furthermore, during training, the outputs are computed for both: the introduced reparameterization, and for the original model. The outputs are then combined into a single hidden output $h_o \in \mathbb{R}^d$. Only the parameters from matrices A and B are optimized. After training, the resulting matrices A, B can be combined into a single matrix $\Delta W = AB$, which can be added to the original set of weights W [15]. Figure 3.2 This process can be performed across arbitrary portions of the matrices from the model at hand. A result of their work was the reduction of the trainable parameters of GPT-3[5] by $10,000\times$ [15].

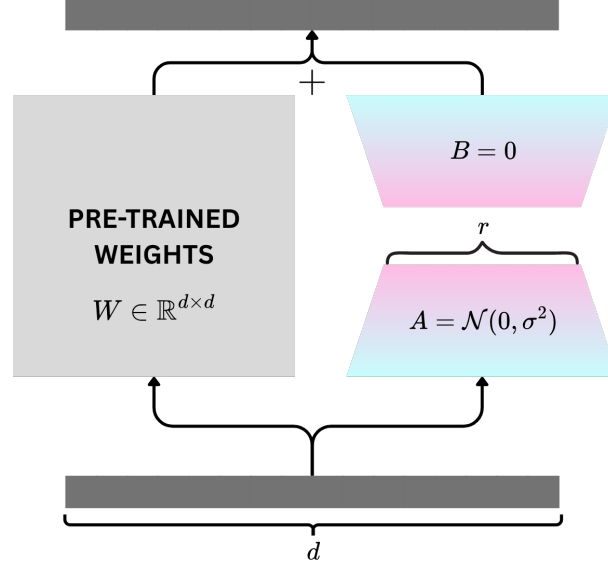


Figure 3.2: Reparameterization process in LoRA, only matrices A and B are trained, reducing learnable parameter landscape.

3.3 Neural Tensor Networks (NTNs)

Neural Tensor Networks are a type of neural network architecture introduced to model relations between pairs of feature embeddings in a way that captures interactions across multiple dimensions. They were first introduced by Socher et al. [41] in the context of knowledge base completion and relational reasoning, where they used them to compute a score to determine whether two WordNet entities were related. The formulation for the NTN is originally described by Socher et al. [41] as:

$$g(e_1, R, e_2) = u_R^T f(e_1^T W_R^{[1:m]} e_2 + V_R \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} + b_R) \quad (3.3)$$

Where u_R^T is a learnable parameter vector specific for each relationship, f is a nonlinear function, $e_i \in \mathbb{R}^d$ are feature vectors related to a given entity, $W_R^{[1:m]} \in \mathbb{R}^{m \times d \times d}$ denotes a learnable tensor, and $V_R \in \mathbb{R}^{m \times 2d}$, $b_R \in \mathbb{R}^m$ are standard affine parameters. This formulation allows for the operation $e_1^T W_R^{[1:m]} e_2$ to model bilinear interactions.

3.4 Relaxed Sorting Operator

Sorting functions are not commonly differentiable, making gradient-based optimization based on them impossible without approximations. Sorting a set of elements can be captured as an operation through a permutation matrix $P \in \{0, 1\}^{n \times n}$, defined as a square matrix where every row has a total sum of 1, and also every column. In other words, a double stochastic matrix. Moreover, each element in the matrix is either 0, or 1: $P_{i,j} \in \{0, 1\}$ [11]. A permutation matrix can be multiplied with a vector $v \in \mathbb{R}^n$, resulting in a permutation of the vector's components, dictated by the position of the value 1 components of the matrix.

Grover et al. [11] developed a technique that relaxes the formulation of permutation matrices into a version that permits differentiable operations and that can be mapped to its discrete formulation. This allows to incorporate sorting operations into models without compromising the possibility of gradient-based optimization. That is, given the operation $\arg \text{sort}(s)$ of a continuous vector $s \in \mathbb{R}^n$, there is a

permutation matrix $P_{\text{sort}(\mathbf{s})}$ defining this order. Such matrix is formulated in a relaxed fashion. The definition of the i 'th row for such relaxed matrix $\hat{P} \in \mathbb{R}_{0 \leq}^{n \times n}$ is given in equation (3.4), where $\tau > 0$ is a control parameter that defines the fidelity of the approximation to the deterministic permutation matrix P_{sort} , and A_s is a matrix of absolute pairwise differences $A_s[i, j] = |s_i - s_j|$ from $\mathbf{s}$. As τ approaches 0 from the positive side, the approximation becomes closer to the real matrix [11].

$$\hat{P}_{\text{sort}(\mathbf{s})}[i : s](\tau) = \text{softmax}[(n + 1 - 2i)\mathbf{s} - A_s \mathbf{1}]/\tau \quad (3.4)$$

3.5 NeuralNDCG

nDCG is not a differentiable function by itself due to the sorting operation within the maxDCG component, therefore optimizing on it directly is not possible. Pobrotyn et al. [35] formulated a relaxed version of nDCG called Neural nDCG which is differentiable end-to-end. The implementation makes use of the sorting operator relaxation introduced earlier [11]. Their formulation of neural nDCG is defined in equation (3.5) [35]. Where $\mathbf{s}$ is a vector of score outputs for each candidate document $\mathbf{x}_i$ to rank from a LTR model, $\mathbf{y}$ is the vector of ground truth relevance labels for each of the candidate documents to rank. In this formulation, $\hat{P}$ is the approximated permutation matrix given the ground truth $\mathbf{y}$. Function g represents the gain of the ground truth relevances. The gain is the numerator of the *DCG* function. That is, it is simply defined as $g(y_i) = 2^{\text{rel}(y_i)} - 1$. Similarly, the discount function d is the denominator of *DCG*, defined as $d(y_i) = \frac{1}{\log_2(i+1)}$. The approximate permutation matrix is multiplied with $g(\mathbf{y})$, yielding an approximately sorted version of the gains. The function "scale" is the application of Sinkhorn scaling [40], in order to transform $\hat{P}$ into a row and column-stochastic matrix [40, 35]. Finally, the term N_k represents the maxDCG at rank k . This formulation is usable as a loss objective, by multiplying it by -1 and optimizing it using any gradient-based optimization method.

$$\text{NeuralNDCG}_k(\tau)(\mathbf{s}, \mathbf{y}) = N_k^{-1} \sum_{j=1}^k (\text{scale}(\hat{P}) \cdot g(\mathbf{y}))_j \cdot d(j) \quad (3.5)$$

3.6 Margin Ranking Loss

Margin ranking loss is a triplet-based contrastive loss objective function. This objective is especially useful to train retrieval models.

Let $x_i = (q, d^+, d^-)$ be a triplet where the anchor q represents a query, d^+ a positive (relevant) document, and d^- a negative (irrelevant) document. The notation of the formulation is adapted from the research of Wang et al. [44] to fit this work's notions (3.6). $\phi(q, d^\pm)$ denotes a score between a query-document pair, and $\gamma \in \mathbb{R}_{0 \geq}$ is a margin used to promote a stronger distinction between negative and positive document scores. The objective enables the optimization of the model's parameters, such that scores assigned to positive query-document pairs are in general at least as large as negative query-document pairs, plus the given margin γ . This means that the optimized model must learn more explicit and clear separation boundaries between negative and positive documents, as the margin increases. This function is used as part of the experimentation for one of the proposed direct alpha optimization model variants.

$$l(q, d^+, d^-) = \max\{0, \gamma + \phi(q, d^-) - \phi(q, d^+)\} \quad (3.6)$$

Chapter 4

Methodology

This section provides a theoretical overview of the methods employed within this work. Starting with details on the architecture and design choices of the developed Elasticsearch re-ranking plug-in, and also of the Auto-Alpha toolset. Furthermore, this section introduces the theoretical framework necessary for the gradient-based optimization methods, in order to optimize the interpolation constant α in hybrid re-ranking, and also for the proposed adaptor methods.

4.1 Elasticsearch Re-ranking Plugin

This work provides an Elasticsearch (ES) implementation of the Fast-Forward Indexes (FFWI) technique from Leonhardt et al. [23], while making its integration simple, as it is provided as a native ES software plugin. This allows for the re-ranking plugin to be seamlessly installed in an existing ES setup, and to use it immediately, provided that the user has the necessary requirements, outlined in the upcoming sections. Moreover, this tool further validates the practical applicability of the Fast-Forward Indexes approach and provides a platform for the application of the findings on α -optimization (RQ1). Furthermore, this section details the design and implementation choices of this plugin and tool, and details how it achieves the goal of a simplified, yet powerful hybrid re-ranking integration.

Finally, this work also provides a python-based tool set that facilitates the application of the techniques outlined in section 4.5, which allow to automatically find an appropriate interpolation constant α for the hybrid re-ranking ES plugin.

4.1.1 Motivation

The implementation and application of research in IR techniques is often limited to experts and researchers. That is, until open-source or commercial projects enable their adoption through tools and software applications. This work aims to help achieve this process for the work of Leonhardt et al. [23], together with other contributions from this thesis.

Choice of Elasticsearch

A perfect environment for the practical applicability of a re-ranking technique such as FFWI is Elasticsearch, due to its massive adoption as a search backbone both in open source, and in commercial applications.

The integration of the FFWI and direct alpha optimization techniques make together a useful contribution for the general public and for the research community, enabling simple access and configuration through ES. Moreover, the choice of a native ES plugin provides seamless integration into existing ES clusters,

making use of the ES aggregation and distribution logic, and providing a familiar search and re-scoring API for developers.

4.1.2 Architecture Overview

At a high level, the chain of operations in a search request that is coupled with the re-ranking plugin is the following:

1. Client elicits search with query and its corresponding parameters: Interpolation constant to weight BM25 and dot product contributions, number of results k to re-rank, etc.
2. Elasticsearch propagates the given request within its internal nodes, and shards retrieve an initial set of top- N results, scored using BM25.
3. After the initial retrieval step, shards trigger the Re-scoring plugin.
4. Within the re-ranking plugin, a dense embedding representation for the query is obtained by sending a request to an external query-encoding service, and the results are re-ordered based on the interpolation of the BM25, and computed dense scores, only for a window of k documents, such that $k \leq N$.
5. After re-scoring, results are fetched and aggregated by the coordinator node, and the results are then re-ordered according to their scores.
6. Results are returned to the client

The high-level steps can also be appreciated below in Figure 4.1. Additionally, to capture the full flow of the system in a more precise manner, a sequence diagram is also presented in figure 4.2.

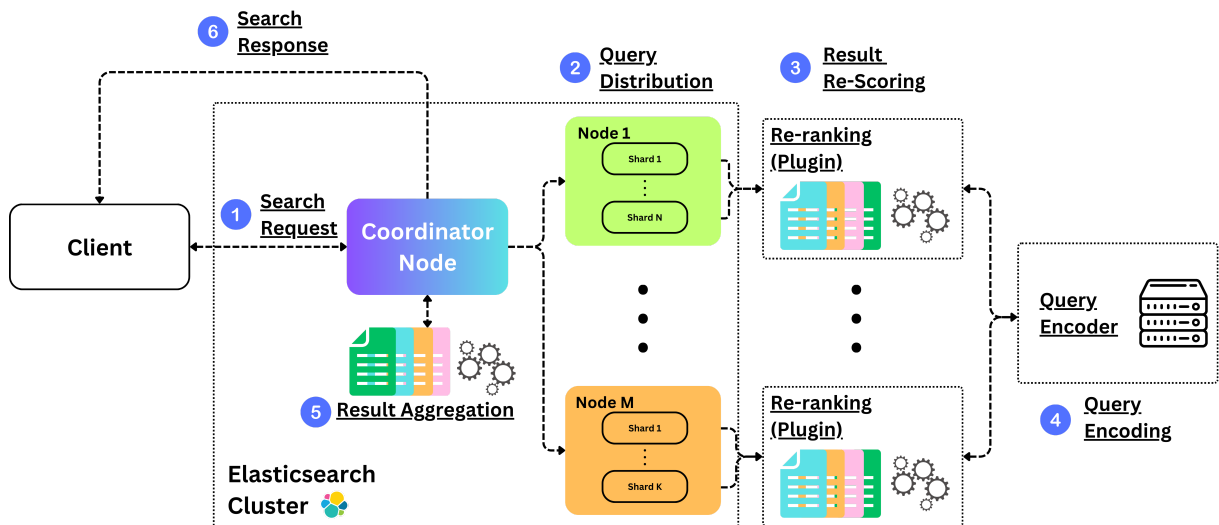


Figure 4.1: High-level system diagram of Elasticsearch and the plugin.

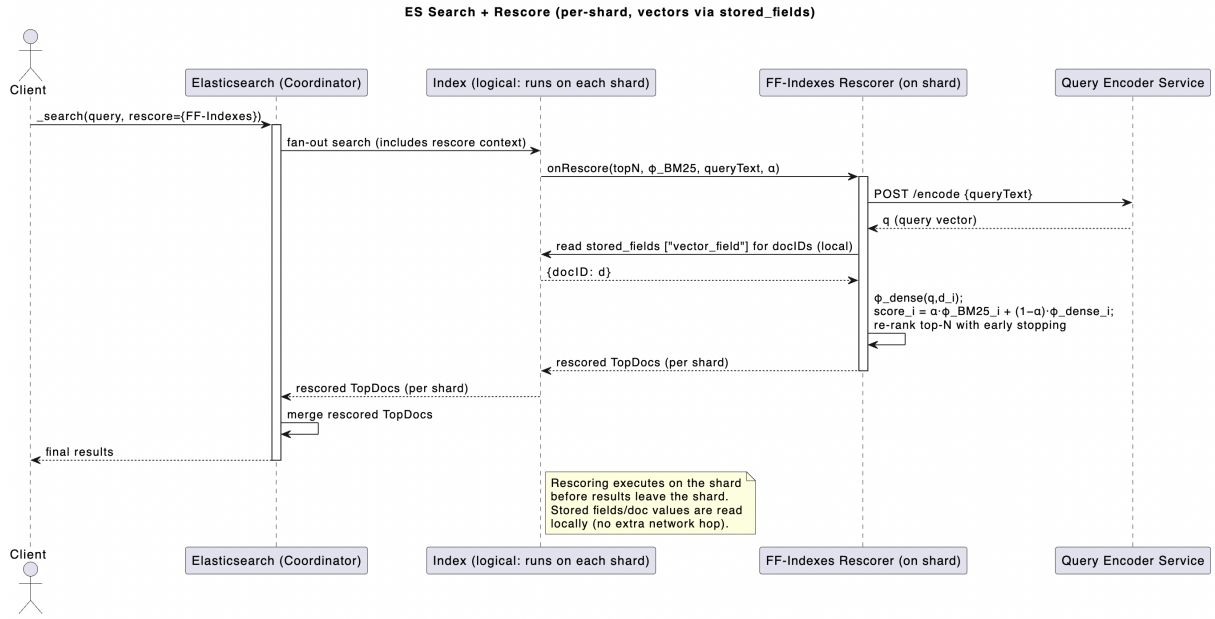


Figure 4.2: Sequence diagram showcasing the execution of a query using elasticsearch and the FFWI-Rescorer plugin.

Since the system uses a combination of the dot product between query-document embeddings and their corresponding BM25 scores, the interpolation α constant is required to weight the contributions in a linear combination, as denoted before in equation (3.2). A tool to automatically tune α is therefore provided, which is detailed later in section 4.2.

4.1.3 Plugin Component-Level Design

Class Components

The plugin can be broken into 5 main components with the following responsibilities:

- **FFWIRescoreBuilder**: Parsing of plugin request fields, such as `alpha`, `rescoreWindow`, `dense_vector_field_name`, and fetching of index-level settings for the plugin.
- **FFWIRescorePlugin**: Orchestrates instantiation of the `RescoreBuilder` class and extends ES' `Plugin` class, which is the template superclass used as a starting point for native ES plugins. Moreover, it allows to define and fetch Index/Node level settings. Three index settings were introduced. The first setting is for the value of α to use for the given index. This allows for the Auto-Alpha tool to set this value without the user having to do so manually. The second and third index settings are used to setup the query-encoder API host address, and the limit for a connection timeout in milliseconds per query-encoding request.
- **ServerCaller**: This class implements the infrastructure necessary to call the query-encoder service, and also the methods to unpack the payload of the response.
- **Reranker**: This class implements the re-scoring logic, and makes use of efficient matrix multiplications to batch compute the dense scores.
- **QueryVector**: This is a wrapper class to store the query embedding representations, together with their query text.
- **ScoredDocument**: Used as a wrapper to keep track of each of the top documents' attributes, such as: `id`, `BM25 score`, `assigned rank in first-stage retrieval`, `dense score`, `embedding representation`, and its `hybrid score after re-scoring`.

Figure 4.3 presents the class diagram for this plugin, showcasing in detail how the classes are related to each other. The architecture cleanly separates concerns into distinct components: plugin integration, request handling, core logic, and data modeling.

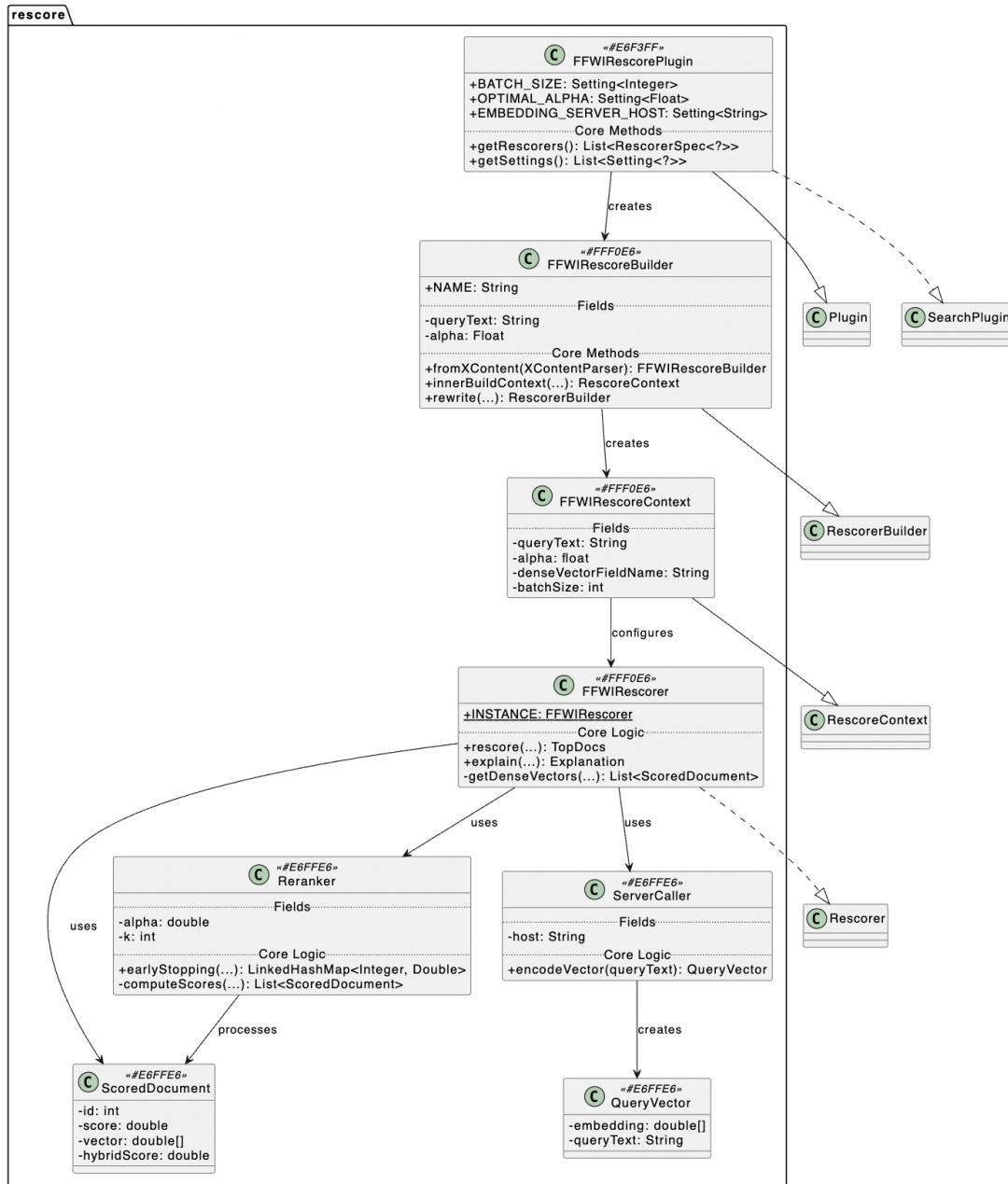


Figure 4.3: Class diagram of the FFWI-Rescorer plugin.

Early Stopping

In order to efficiently re-rank the retrieved top- N documents, the Reranker class implements a variant of the early stopping algorithm proposed by Leonhardt et al. [23], where a re-ranking window of size k is given, such that $k < k_s$, where k_s is the retrieval depth. The algorithm aims to only re-rank the window of k documents, without compromising re-ranking quality. This is achieved by making a reasonable estimation of the maximum achievable hybrid score of document d_{k+j} following the lowest ranked document at position d_k . This approximation is done without directly computing the dense score

for document d_{k+j} , but by considering what its hybrid score $\phi(\cdot)$ would be if using the previously highest observed dense score $s_D = \max \{\phi_D(d_i, q)\}$ for documents $d_i \in D_k$. Where D_k is the set of the top- k documents. This process continues until no improvement is observed. Leonhardt et al. showed that this is a reasonable approximation when s_D is the true maximum of the dense scores [23]. This implementation of the early stopping algorithm makes use of batches to efficiently compute the query-document dense scores in groups. The pseudocode of the modified early stopping algorithm is given below in algorithm 1.

Algorithm 1 Batched Early-Stopping Hybrid Re-ranking (adapted from [23])

Require: Documents D , query vector q , batch size B , target k , interpolation $\alpha \in [0, 1]$
Ensure: Top- k by $\phi(q, d) = \alpha \phi_S(q, d) + (1 - \alpha) \phi_D(q, d)$

```

0:  $Q \leftarrow$  empty min-heap keyed by  $\phi(\cdot)$ 
0:  $s_D \leftarrow -\infty$ ;  $stoppedEarly \leftarrow \text{false}$ 
0: for  $i \leftarrow 1$  to  $|D|$  step  $B$  do
0:    $C \leftarrow D[i : \min(i + B - 1, |D|)]$ 
0:   compute dense scores  $\phi_D(q, d)$  for all  $d \in C$  given  $q$  ▷ Efficient matrix multiplication
0:   for each  $d \in C$  do
0:      $removed \leftarrow \text{null}$ 
0:     if  $|Q| = k$  then
0:        $removed \leftarrow \text{PopMin}(Q)$ 
0:        $s_{\min} \leftarrow \phi(q, removed)$ 
0:        $s_{\text{best}} \leftarrow \alpha \phi_S(q, d) + (1 - \alpha) s_D$ 
0:       if  $s_{\text{best}} \leq s_{\min}$  then
0:          $\text{Push}(Q, removed)$ 
0:          $stoppedEarly \leftarrow \text{true}$ 
0:       break ▷ exit inner loop
0:    $d_\phi \leftarrow \alpha \phi_S(q, d) + (1 - \alpha) \phi_D(q, d)$ 
0:   if  $removed \neq \text{null}$  then
0:     if  $d_\phi > \phi(q, removed)$  then
0:        $\text{Push}(Q, d)$ 
0:     else
0:        $\text{Push}(Q, removed)$ 
0:   else
0:      $\text{Push}(Q, d)$ 
0:   if  $stoppedEarly$  then
0:     break ▷ exit outer loop
0: return top- $k$  items in  $Q$ 

```

Design choice: External Query-Encoder

Since only the document embeddings are pre-computed in the fast-forward indexes paradigm, query embeddings must be computed at runtime during re-ranking. The query-encoder model used must match the type of encoder used for document embeddings' pre-computation. There were two main options to meet this requirement:

1. Implement native Java-based infrastructure that supports a fixed set of models, with the advantage of local computations without the latency of an external dependency.
2. Outsource the computation of the query embeddings, thereby adding flexibility, scalability, and maintainability, with the drawback of added latency.

In this implementation, the latter option was chosen, since limiting the selection of encoder model variants would limit the subset of users to those employing those specific types of model architectures, and would also make maintainability of this tool more involved, since the state-of-the-art models change constantly, potentially drawing the pre-selected models obsolete soon. Furthermore, burdening the Elasticsearch nodes directly with heavy computations such as query encoding could potentially result in more overhead and instability, rather than outsourcing this process, depending on the specific setup. Finally, scalability can be an important detail to consider, and having an external query-encoder means that its infrastructure can be managed and scaled separately from Elasticsearch nodes. More details about the benefits, and how the external query-encoding service works are given under section 4.2, since this is part of the Auto-Alpha tool set.

4.2 Auto-Alpha Tool Set

A set of minimal tools set was developed with the ultimate objective of applying the direct alpha optimization technique described in section 4.5. Moreover, it also includes a simple, yet extensible query-encoding service that can be spin-up and used at the same time as the ES plugin.

The tools offered are the following:

- **Alpha Optimization:** Train the α parameter using the assembled training data, using the neural nDCG-based loss objective [35]. Moreover, if an Elasticsearch cluster details are provided, and the FFWI-Rescore plugin is installed, then the alpha settings are automatically setup for the desired index.
- **Query-Encoding Service:** A minimal server that works out of the box with Huggingface’s transformers library [45], provided that the model is part of Huggingface’s model hub and supports AutoModel and AutoTokenizer. Moreover, the components of this service are easy to extend/adapt, as a register pattern was used to provide ample customization in a straightforward manner.
- **Training Data Preparation:** Create training examples (lists for Neural nDCG) using the retrieved documents, their scores, and the provided queries with their respective qrels.
- **Embedding Computation Module:** Given that the user does not have a pre-computed set of embeddings, they can be generated.

The set of tools taking part of Auto-Alpha are described in more detail in the upcoming subsections.

4.2.1 Alpha-Optimization

An efficient, yet effective tool was developed in order to find a suitable α value. The ideal scenario is when the user has a subset of labeled data from its real collection. That is, having a set of queries and labeled qrels with some relevant documents, preferably having graded relevance, since the loss objective, although binary relevance can also be employed. This set likely does not necessarily need to be large. For instance, a set of the experiments carried out in this work partially made use of only 38 queries from TREC-DL-Psg’ 2019, for which α optimization was possible⁵. Nevertheless, a larger subset is preferable if available.

The training procedure is identical to the one presented in section 4.5. Since neural nDCG [35] is used as loss objective. Moreover, the data sampling procedure described in section 4.5.2 is also used.

The high-level training procedure is described in figure 4.4.

Moreover, the optimization procedure can be configured through yaml files. The configuration parameters and expected inputs are can be found in detail within the github repository of the tool ¹.

¹https://github.com/TUDThesis/alpha_auto_tune

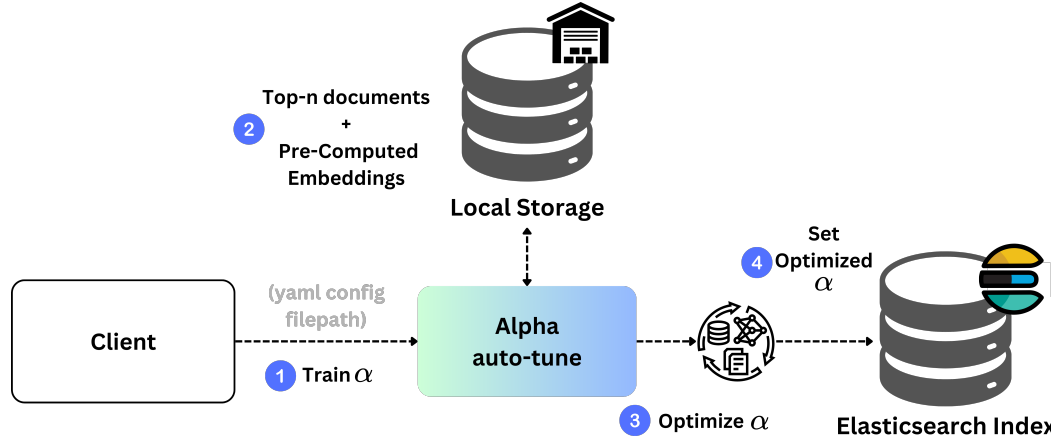


Figure 4.4: High-level architecture of the alpha-optimization procedure. The steps are the following: 1. The user provides a yaml configuration file. 2. The prepared dataset created using the data preparation tools is fetched from the specified paths, plus the pre-computed memory mapped embeddings are loaded. 3. The training procedure optimizes α . 4. After optimization, the final value of alpha is registered as an elasticsearch’s index setting for the specified index, thereby making it readily available for use by the ES reranking plugin.

4.2.2 Query-Encoding Service

A query-encoding service is also included in the set of utilities. The query-encoding service is made using the FastAPI framework[2], coupled with a Uvicorn [1] server. This service was designed to work alongside the ES re-ranking plugin, however it is not limited to only query-encoding. It is generic and can be used for either documents or queries.

Multiple Encoder/Pooler Support

A variety of models can be used for query encoding and document encoding out-of-the-box. This is possible through the use of the `transformers` [45] library. This is achieved by using their `AutoModel`/`AutoTokenizer` classes, which automatically download and setup a model and its respective tokenizer by using a provided model name or identifier. The only requirement for this is that the model is registered within Huggingface’s [45] model hub. Moreover, the user can make use of arguments to adjust the float precision level employed for output embeddings. Additionally, an arbitrary number of tokenizer execution-time keyword arguments can be passed, in order to adjust the behaviour of the tokenizer at hand during query/document tokenization. Finally, configuration for the encoder models and the underlying pooling strategies can be given through `.yaml` files.

Since pooling is an essential part of query/document encoding, two pre-defined common ways of pooling are provided:

- **Mean pooling:** Taking the mean value of all embeddings within the last hidden state.
- **CLS pooling:** Makes use of the CLS special token hidden output representation.

Cutomization of the Encoder/Pooler

The strategies mentioned in the previous subsection already provide flexibility for a range of needs, however, they might not be enough for some specific use-cases. For this reason, a base encoder class, and a base pooler class are given. These classes are structured by using structural typing through Python’s `Protocol` class. This means that they are extensible by simply adhering to the structure of the base classes provided, and by following the expected input/output signatures defined for the methods of the base classes. A minimal example class diagram of how this would look at a class level is given in figure

4.5. This means that the users are not constrained to only use models available through Huggingface, but can define their own custom encoder logic.

The same principle applies for custom pooler implementations. More detailed information about how to do this, together with code examples can be found in the Auto-Alpha repository ². Finally, in order to provide a simple way of interfacing the existing infrastructure with any given existing encoder/pooler implementation, the path to a Python module that implements the custom classes can be provided, as part of the `.yaml` configuration. The service will then automatically import them and initialize them with the rest of the provided encoder/pooler arguments from the provided configuration.

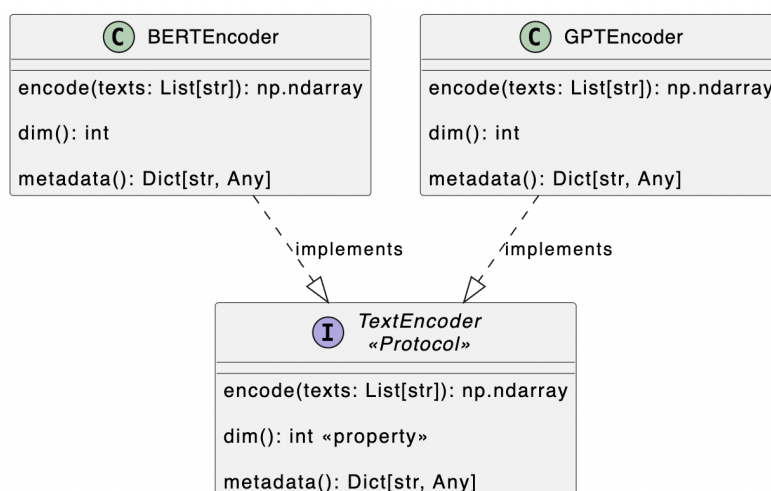


Figure 4.5: Class diagram of how the text encoder could be extended, and the expected types for each method/input/output.

Query-Encoder Profiler

The query encoding service provides the possibility to profile the processing time of all requests performed across its lifespan. At the end of the process' execution, the profiler saves a file containing the 50th, 90th and 99.9th, and 100th percentiles for processing times.

4.2.3 Data Preparation

Even when there is labeled data for training available, it has to be massaged into a specific format. For this reason, the Auto-Alpha toolkit provides utilities to process it into the appropriate files and format. The optimization procedure for alpha requires the following input components:

- **Top-N Results (for training split only):** A dataframe containing the top-N scoring documents based on a standard Lucene index search, for each query in the queries provided. Each row should contain a query id, document id, ranking, BM25 score, and relevance label.
- **Lucene Index Path:** The path to a Lucene index, containing the indexed collection to validate and test against during alpha optimization.
- **Memory Mapped Embedding Paths:** Paths to the `.mmap` embedding files for query and document embeddings. Memory maps are used to keep embeddings in disk while being able to access them during training as if they were in main memory. Moreover, the paths to their id maps are also required. That is, a `.json` file containing the mappings from query/document ids to the index position in the `.mmap` array.
- **Queries/Qrels file paths (for validation and testing)**

²https://github.com/TUDThesis/alpha_auto_tune

Data Preparation Pipeline

The data preparation pipeline to ultimately produce the aforementioned files is described in figure 4.6. The process runs sequentially, and requires an input `.yaml` file with configuration. The details for the configuration files can also be found in the Auto-Alpha repository ³. The output files from this procedure are stored under a common application folder for Auto-Alpha. A hash is computed based on the input configuration, generating a unique folder for the combination of datasets and parameters. This pipeline also supports `ir-datasets` [30] paths, automatically downloading the collection/queries/qrels if used, for each part of the process.

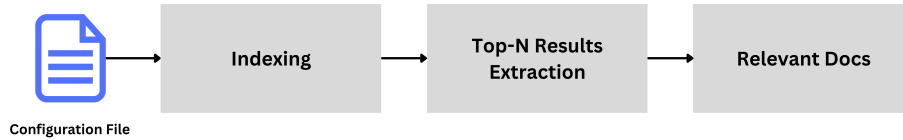


Figure 4.6: High-level view of the data generation pipeline. The process is designed to run sequentially.

Embedding Generation (`.mmap` files)

Generating embeddings for a given collection and for a set of queries can be time-consuming, therefore a utility is also provided for this. This utility makes use of the same backbone tools created for the query-encoding service, therefore it is as customizable. The configuration is given through a `.yaml` file too, providing the configuration for the collection path / queries path, encoder model type, embedding pooler type, float precision to use, `ir-datasets`' path (potentially), etc.

The output of the embedding generation procedure consists of two files: a `.mmap` file, and a document/query id $\rightarrow$ index mapping `.json` file, which is used to find the index containing the corresponding embedding for a given document/query within the memory mapped array.

4.3 Overview: Elasticsearch + Auto-Alpha (RQ 3)

The provided native Elasticsearch plugin, combined with the Auto-Alpha tool set described in the previous section provide a comprehensive package that allows for a faster and simpler integration of the fast-forward indexes re-ranking paradigm [23], while adding the extra feature of an automatic α optimization that uses gradient-based optimization. This bundle of contributions also appeal to a variety of potential users due to Elasticsearch's popularity.

Considering the general setup, research question 3 can be answered (RQ 3: "Can a native Elasticsearch (or equivalent framework) plugin/tool be developed to implement the Fast-Forward Indexes approach while making its setup simple? Moreover, how can insights from RQs 1 and 2 be applied to this implementation?"):

The answer is yes, the Elasticsearch-based plugin implements the Fast-Forward Indexes hybrid re-ranking paradigm, while making its setup simple, since installation and configuration of the plugin is fast and straightforward. Moreover, insights of the alpha optimization research were applied by implementing a tool for the automatic optimization of the parameter through a gradient-based approach, as described in sections 4.5 and 4.2.1. Insights from adaptor models are however not applied, and remain as potential future work.

³https://github.com/TUDThesis/alpha_auto_tune

4.4 Plugin Testing

Testing is performed in order to understand the expected performance of the plugin in its current state. To this end, Elasticsearch Rally⁴ [10] was used to design and run a benchmark task to capture the following metrics:

- **Min/Mean/Max Throughput:** Measures the number of queries per second achievable by the system, for the type of search at hand (BM25, or BM25 + Re-ranking).
- **Latency Percentiles:** Measuring the 50th/90th/99.9th/100th latency percentiles during the benchmark. That is, the total time it takes for the request to be served and returned to the client, including queue waiting times if any.
- **Service Time Percentiles:** Measuring the 50th/90th/99.9th/100th service time percentiles during the benchmark. That is, the total amount of time taken from the moment the request is sent to Elasticsearch, up until a response is received, without considering queue times, but considering the time spent sending the request and receiving the response.
- **Error Rate:** Rate of failed queries during run.

The aforementioned metrics can be insightful to know what is the expected performance of the service in reality. The following two subsections provide more details about the testing structure and methodology. The specific technical details of the testing setup are also provided within the experimental section 5.

4.4.1 Benchmark Run Structure

Rally allows to define custom "races", that is, custom benchmarks that can be run and have parameters that can be overridden at run-time. In this case, a benchmark was defined to obtain insights on both the performance of a traditional BM25 search, and a search that includes the re-ranking process using the plugin. The run is structured in such a way so that 5,000 queries are performed for pure BM25, followed by 5,000 queries with BM25 + Re-ranking with the plugin. This process is repeated 4 times per run. This interleaved procedure was chosen to mitigate the effect of caching in the final statistics. Moreover, 3,000 request are issued by both BM25 and BM25+Re-ranking prior to the start of the benchmark for "warmup" purposes which helps attain a more consistent set of results for the main tasks. The benchmark was setup for a range of re-scoring window sizes, namely for the top: 100, 250, 500, and 1000 documents from the first-stage retrieval. Finally, the query-encoder service was equipped with a profiling mechanism to keep track of the processing-times for query embedding computations, along a full run, measuring the 50th/90th/99.9th percentiles for it.

The end-to-end benchmarking process is depicted in figure 4.7.

⁴<https://github.com/elastic/rally>

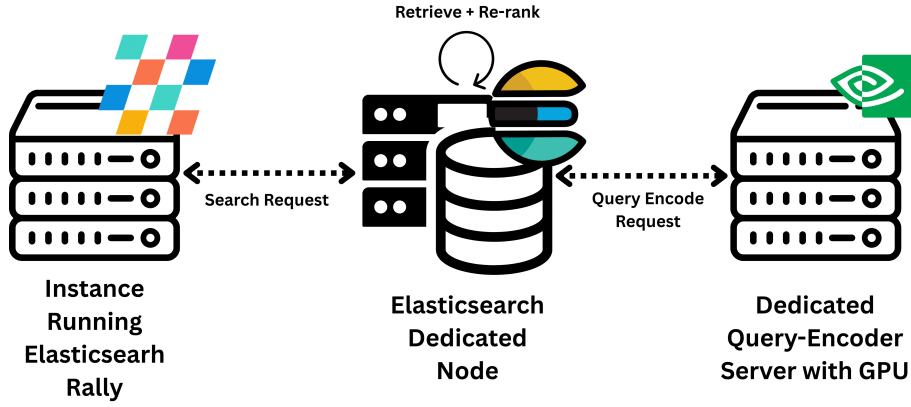


Figure 4.7: Setup for testing procedure: three dedicated instances are spin up for this process. One runs the rally testing service, one hosts the elasticsearch service as a dedicated single ES node, and a dedicated instance runs the query-encoding service, making use of GPU acceleration.

4.4.2 Deriving Final Results

In order to gather insights from the test runs, the service time percentiles for pure BM25 search results were subtracted from the BM25 + Re-rank search results. This difference helps estimate what is the approximate service time incurred by the re-ranking procedure itself, without considering the first-stage BM25-based query. Tests were carried out for each re-ranking window size, therefore service-time and throughput percentiles were also captured for each of those window sizes.

All of the computed results are presented within the results section 6. Additionally, more technical details about the setup of the tests are also provided within the experimental setup section 5.

4.5 Direct Optimization of Interpolation Constant α

In this strategy, the parameter α is optimized directly using a gradient-based optimization method. A high-level overview of the method is depicted in figure 4.8.

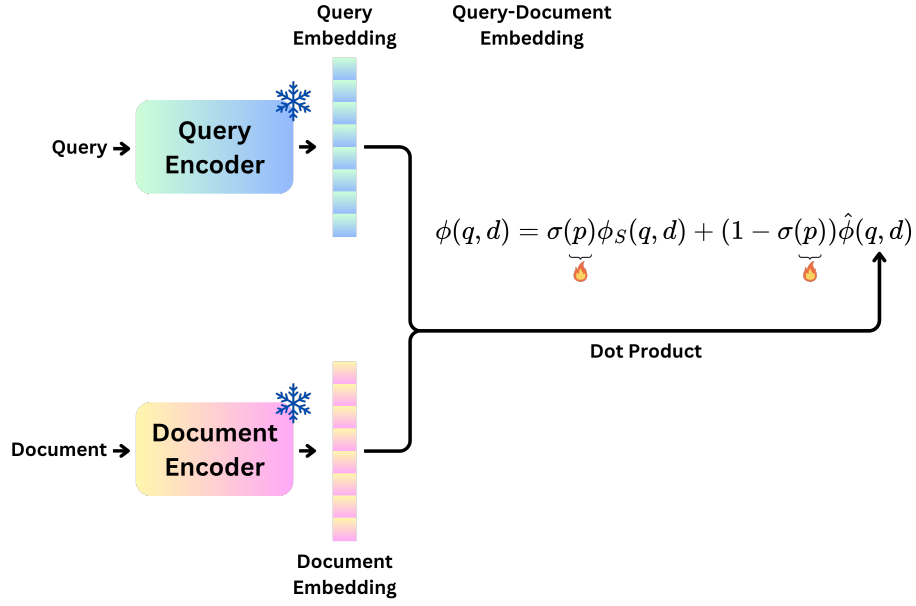


Figure 4.8: High-level architecture of model to directly optimize α (logit of sigmoid). The backbone encoder(s) remain non-trainable, Alpha (the logit which follows as input of a sigmoid function) remains trainable. The flame symbol represents trainable parameters.

Optimization is performed on a logit parameter $p \in \mathbb{R}$, which is eventually the input of a sigmoid function, ensuring that alpha is between 0 and 1, thus, the final interpolation constant is defined in equation (4.1).

$$\alpha = \sigma(p) \in [0, 1] \quad (4.1)$$

Experimentation is carried out on two different loss objectives: Margin Ranking Loss [44], and Neural nDCG [36], their setup is described further below.

4.5.1 Optimization through Margin Ranking Loss

This loss formulation implicitly steers α towards a value that promotes higher scores for positive documents than those of negative ones. Moreover, the margin $\gamma > 0$ pushes for the score of the positive document $\phi(q, d^+)$ to be at least as large as the score assigned to the negative example, plus the defined margin: $\gamma + \phi(q, d^-)$. The logit p is optimized across the set of training triplets T , as described in equation 4.2.

$$p^* = \arg \min_p \sum_{(q_i, d_i^+, d_i^-) \in T} \max\{0, \gamma + \phi(q, d_i^-) - \phi(q, d_i^+)\} \quad (4.2)$$

Triplet Assembly for Margin Ranking Loss

Margin Ranking Loss requires triplets, therefore their assembly is necessary if only queries and query relevance labels are provided for a given dataset. Triplets are built by making use of the pre-computed top- n BM25 scoring documents for a given query, in this case using a pre-built Lucene index. From those top-scoring documents, a pool P_k of only k documents is randomly sampled without replacement, by using weighted sampling, assigning higher selection probability to higher scoring documents. The sampling distribution is defined by using a softmax function based on the BM25 scores of the documents as described in equation 4.3:

$$f(d_i) = \frac{e^{(d_i \text{BM25})}}{\sum_{j=0}^{n-1} e^{(d_j \text{BM25})}} \quad (4.3)$$

Once a sample of k documents is obtained, one of the known relevant documents d^+ for the given query is sampled at random (uniformly) from the pool of relevant documents. Finally, triplets are generated by choosing documents $d^- \in P_k$, such that

$rel(q, d^+) > rel(q, d^-)$, where d^+ is the randomly selected relevant document, and d^- is the chosen document from the pool with lower relevance than d^+ . Figure 4.9 shows the procedure graphically. This approach is inspired on the research by Karpukhin et al. [20], where they sample negative documents by selecting high BM25 scoring non-relevant documents.

This method is useful for datasets that have either binary, or graded relevance label annotations. In the case of the latter, triplets can be built by simply ensuring that the positive example is of higher relevance than the negative example.

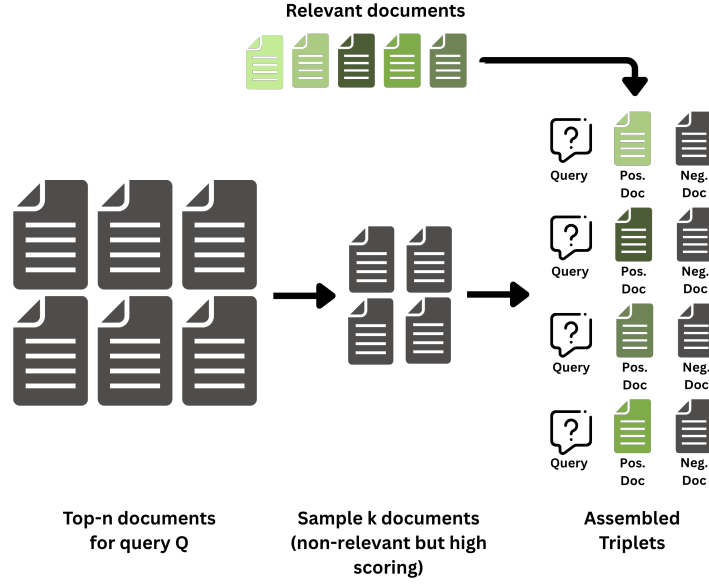


Figure 4.9: Triplet assembly: k random documents sampled from original retrieved documents and used to assemble triplets together with relevant documents.

4.5.2 List-wise Optimization through Neural nDCG

Experimentation is also carried out by optimizing the interpolation constant by using a list-wise learning-to-rank objective. Neural nDCG [35], a surrogate objective for nDCG, is used to optimize on the correct order of documents based on their explicit relevance labels. This surrogate allows to approximate the nDCG metric at an arbitrary cutoff rank k . Then the loss objective is shown in equation 4.4, where N is the total number of queries in the training set, s_i the list of computed scores for each of the documents associated with query q_i , and y_i their corresponding ground truth relevance labels. The next subsection explains how documents are sampled for each training query.

$$p^* = \arg \min_p - \sum_{i=0}^{N-1} \text{NeuralNDCG}(s_i, y_i) \quad (4.4)$$

Sampling procedure for Neural nDCG

During training, examples are selected in a similar fashion to those of the procedure described in section 4.5.1. That is, from the pre-computed top- n documents for each query within the training set, a pool of k documents P_k is randomly selected using the same weighted sampling distribution described in equation

4.3. The sampled pool P_k is then directly used as the list of documents to be scored using the current logit p , for all $d_j \in P_k$, as described in equation 4.5.

$$\phi(q_i, d_j) = \sigma(p) \cdot \phi_S(q_i, d_j) + (1 - \sigma(p)) \cdot \phi_D(q_i, d_j) \quad (4.5)$$

4.6 Adaptor Methods

It is theorized that adaptor models could be used to learn functions that are able to combine the query/document embedding features, and their corresponding BM25 scores in a meaningful way to gain a performance edge over out-of-the-box dot product similarity-based re-ranking, and standard hybrid re-ranking, providing adaptation to data collections for which the pre-trained encoders might not have been pre-trained on. Moreover, it is also conjectured that this can be done while being agnostic to the specific encoder model/architecture used to compute query/document embeddings and first-stage scoring mechanism. Additionally, it is desirable for this adaptor learning process to be runnable with consumer systems without the need of advanced hardware.

The proposed adaptors live on top of the pre-computed query/document embeddings, and are applied after the first stage retrieval during an ad-hoc retrieval task. The generic architecture is depicted in figure 4.10.

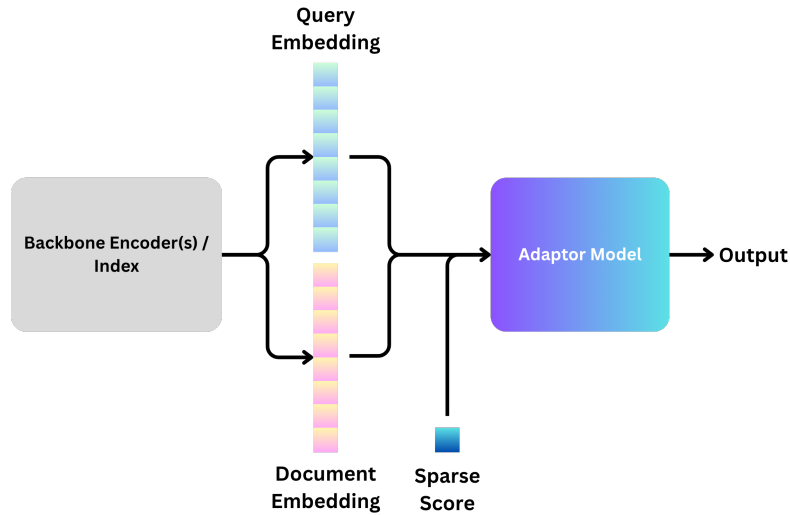


Figure 4.10: Adaptor models in this context have the aim of adapting pre-computed query/document embeddings and BM25 scores, agnostic to the method used to compute the inputs, and finally predict an output dense score or α value.

The adaptors experimented with are of two types:

1. Interpolation variable prediction: Model with the aim of predicting a dynamic interpolation value α per query-document example.
2. Learned dense score prediction: Directly outputs a dense score combining query/document embeddings and their corresponding BM25 scores.

Leonhardt et al. [23] incorporate interpolation of sparse scores and dense scores by weighting them with a fixed α constant as described under section 3.1. Having a fixed value for this weighting constant is effective and simple, however, it is theorized that predicting dynamic values of α on-the-fly could improve convex combination-based re-ranking performance in some cases. Finally, we explore whether a second type of model can learn to combine the features in a relevant way to directly output a final score without direct interpolation scoring.

4.6.1 Loss Objective: Neural NDCG

Methods described under this section 4.6 make use of the list-wise Neural nDCG loss objective described before in section 3.5. As formulated in equation (3.5), the surrogate function for nDCG is optimized by making use of the predicted/assigned relevance scores to a set of n documents, together with the ground truth relevance labels. In this case, since this work deals purely with re-ranking tasks, top- n represents the highest first-stage retrieval scoring documents. During the re-ranking step, the re-ranking adaptors output a relevance score $s_i \in \mathbb{R}$ for every query-document pair, which is used together with its ground truth relevance label $y_i \in \mathbb{Z}$.

4.6.2 Learned Dense Scoring

This section outlines two learned approaches that aim to predict dense relevance scores for document ranking. Given a query embedding $q \in \mathbb{R}^l$, the top- k retrieved documents along with their corresponding embeddings $D \in \mathbb{R}^{k \times d}$, and their associated BM25 scores, these methods seek to compute a final relevance score for each document. The goal is to explore whether learning to combine dense representations with lexical signals can yield more effective ranking strategies than standard dot-product similarity and potentially dense and lexical interpolation. The approaches described here use neural adaptors to combine the input features: query and document embeddings, along with traditional retrieval scores into a single, dense scoring function. The first model variant uses direct concatenation of the query and document embeddings, whereas the second one makes use of a variant of a Neural Tensor Network (NTN) [41] layer, which relies on bilinear interaction operations through learned weight matrices, with the aim of extracting useful non-linear feature relationships from the input query and document embeddings.

Feedforward Neural Network Adaptor

The premise of this model is to be simple and lightweight, therefore experimentation is carried out with a shallow feedforward neural network.

Given our feedforward model $f(\mathbf{c}) : \mathbb{R}^{2l} \mapsto \mathbb{R}$, let $\mathbf{q} \in \mathbb{R}^l$ be the query dense embedding representation, and $\mathbf{d} \in \mathbb{R}^l$ be the dense document embedding representation. Moreover, let ϕ_S represent their BM25 score $\text{BM25}(d, q)$. The input $\mathbf{c} \in \mathbb{R}^{2l}$ is then defined as the concatenation of the Hadamard product between the query and document embeddings, together with the absolute difference between them, as depicted in equation 4.6.

$$\mathbf{c} = \text{Concat}(|\mathbf{q} - \mathbf{d}|, \mathbf{q} \odot \mathbf{d}) \quad (4.6)$$

The use of the absolute difference has been used in other studies successfully to capture semantic differences between dense embeddings [37, 32], and Hadamard products have also been used as interaction features before [37].

Once $\mathbf{c}$ is computed, it is the input of the FFNN, which in turn outputs a dense score $\phi_D = f(\mathbf{c})$. This dense score is finally concatenated together with the BM25 score and fed as input to a very shallow FFNN which uses two-dimensional inputs and outputs a final combination. This process is formally described by equation 4.7. The final FFNN f_2 learns a linear combination between the output dense score ϕ_D and the BM25 score. The non-linear operations occur throughout the first FFNN $f(\cdot)$. The architecture of this model is shown in figure 4.11.

$$\phi = f_2(\text{Concat}(f(\mathbf{c}), \text{BM25}(q, d))) \quad (4.7)$$

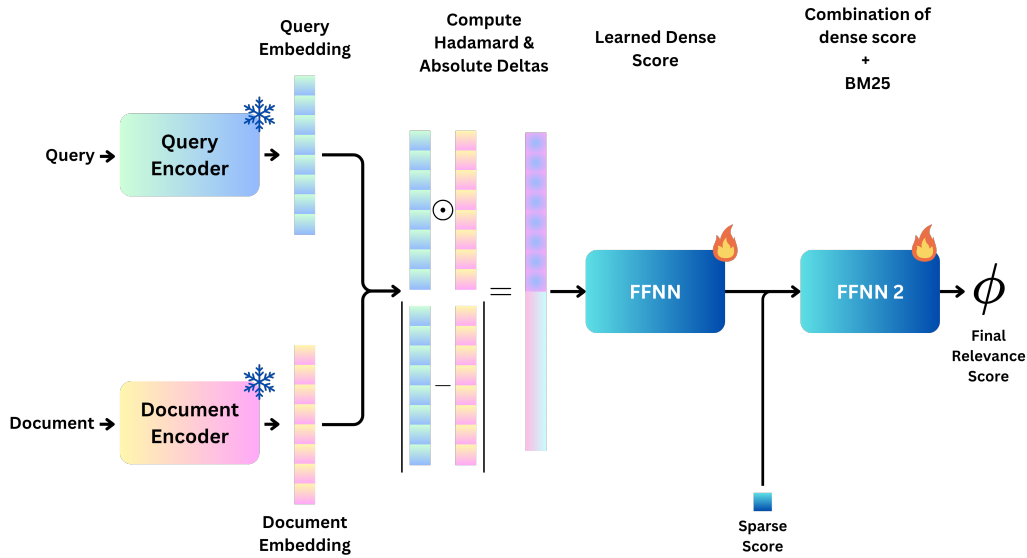


Figure 4.11: FFNN adaptor, embeddings' features are transformed by using an element-wise product and also their absolute difference. The concatenation of the transformations are used as input for FFNN 1 which outputs a dense score which is in turn combined with a BM25 score by FFNN 2.

Neural Tensor Network (NTN) Adaptor

This method consists of a more complex architecture, based on an NTN backbone. The idea is that the bilinear interactions introduced by the NTN blocks could uncover richer feature interactions than other simpler model architecture, e.g., with linear layers. The inputs to this model are: the document and query embeddings, and their corresponding BM25 signal. The adaptor has three main components: An NTN-based feature extractor, and two feedforward neural networks. The first FFNN is meant to combine the features into a dense score, and the second FFNN is meant to weight them into a final score. The FFNN uses the extracted features, together with the Hadamard and absolute difference vectors from the input query and document embeddings. Figure 4.12 shows the architecture and the order of the two components.

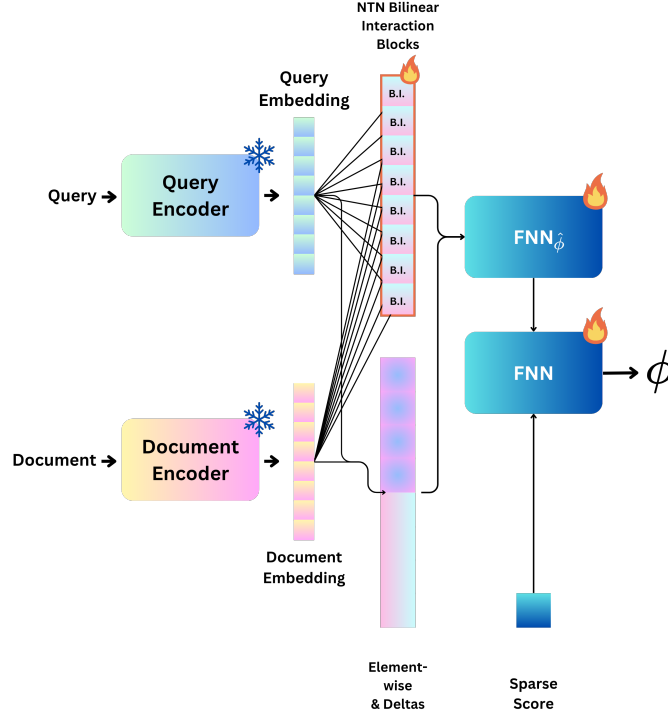


Figure 4.12: NTN-based adaptor: the NTN components extracts relevant features for each bilinear interaction (B.I.) block. Then the Hadamard product and absolute differences of the query and document embeddings are combined with the features as inputs to FFNN_ϕ , and finally combined by a second FFNN.

Let the query embedding $\mathbf{q} \in \mathbb{R}^l$ and the document embedding $\mathbf{d} \in \mathbb{R}^l$. The query and document embeddings are inputs to the NTN model, which in turn outputs a feature vector $\mathbf{o} \in \mathbb{R}^m$, where m denotes the number of learned matrices used to compute the interactions within the NTN, as outlined in section 3.3. This work uses a modified formulation from the one proposed by Socher et al. [41]. In this case, the NTN model does not make use of a projection vector $u^T \in \mathbb{R}^m$, nor its parameters are indexed like defined in (3.3), instead, it is reformulated to output a feature vector $\mathbf{o} \in \mathbb{R}^m$. The reformulation is defined in equation (4.8).

$$\text{NTN}(q, d) = f(q^\top W^{[1:m]}d + V_R \begin{bmatrix} q \\ d \end{bmatrix} + b) \quad (4.8)$$

Let $\mathbf{c} \in \mathbb{R}^{2l}$ be computed as described in equation 4.6 by using $\mathbf{q}$ and $\mathbf{d}$. Additionally let $f_{\hat{\phi}}(\cdot)$ be the FFNN that computes the dense score $\hat{\phi}$, and $f(\cdot)$ the FFNN that computes a final linear combination between $\hat{\phi}$ and the sparse score. The final score ϕ is thus computed as described in equation 4.10.

$$\hat{\phi} = f_{\hat{\phi}}(\text{Concat}(\mathbf{o}, \mathbf{c})) \quad (4.9)$$

$$\phi = f(\text{Concat}(\hat{\phi}, \text{BM25}(q, d))) \quad (4.10)$$

4.6.3 Adaptors: Document-Level Dynamic α Prediction

The methods outlined in this section aim to learn a function that can predict the interpolation value α in a dynamic manner, based on the given input query and document embeddings, and also based on their BM25 signal. The intuition is that query and document embedding features and BM25 scores could provide cues about their contribution to the final relevance score, thus having a dynamically changing value of α could be beneficial.

Neural Tensor Network - Document-Level NTN-DA

This adaptor variant utilizes an NTN to extract features from the document and query embeddings, in a similar fashion to the method described in section 4.6.2. Furthermore, using an NTN allows for the model to abstract interactions between the features of the query and document vectors.

Let the output of the NTN be defined below (4.11).

$$\mathbf{o} = \text{NTN}(\mathbf{q}, \mathbf{d}) \in \mathbb{R}^m \quad (4.11)$$

Moreover, let a FFNN $f : \mathbb{R}^{m+2l+1} \mapsto [0, 1]$ be the learned function that outputs α for the given inputs, which in turn are a concatenation of $\mathbf{o}$, the BM25 score between q and d , and the concatenated Hadamard and absolute difference vectors between $\mathbf{q}$ and $\mathbf{d}$; that is (4.12):

$$\mathbf{c} = \text{Concat}(|\mathbf{q} - \mathbf{d}|, \mathbf{q} \odot \mathbf{d}) \in \mathbb{R}^{2l} \quad (4.12)$$

Alpha is then computed as shown in equation (4.13), where at the last layer of f a sigmoid function is applied to constrain the output to the range $[0, 1]$. The output is an α value for each document in the set of top candidate documents per query.

$$\alpha_{q,d} = \sigma(f(\text{Concat}(\mathbf{o}, \mathbf{c}, \text{BM25}(q, d)))) \quad (4.13)$$

The final score is obtained by interpolation [23], weighting the dot product $\phi_D(\mathbf{q}, \mathbf{d}) = \mathbf{q} \cdot \mathbf{d}$ and the BM25 score $\phi_S(q, d) = \text{BM25}(q, d)$:

$$\phi_{q,d} = \alpha_{q,d} \cdot \phi_S(q, d) + (1 - \alpha_{q,d}) \cdot \phi_D(\mathbf{q}, \mathbf{d}) \quad (4.14)$$

The process can be appreciated graphically in figure 4.13 below:

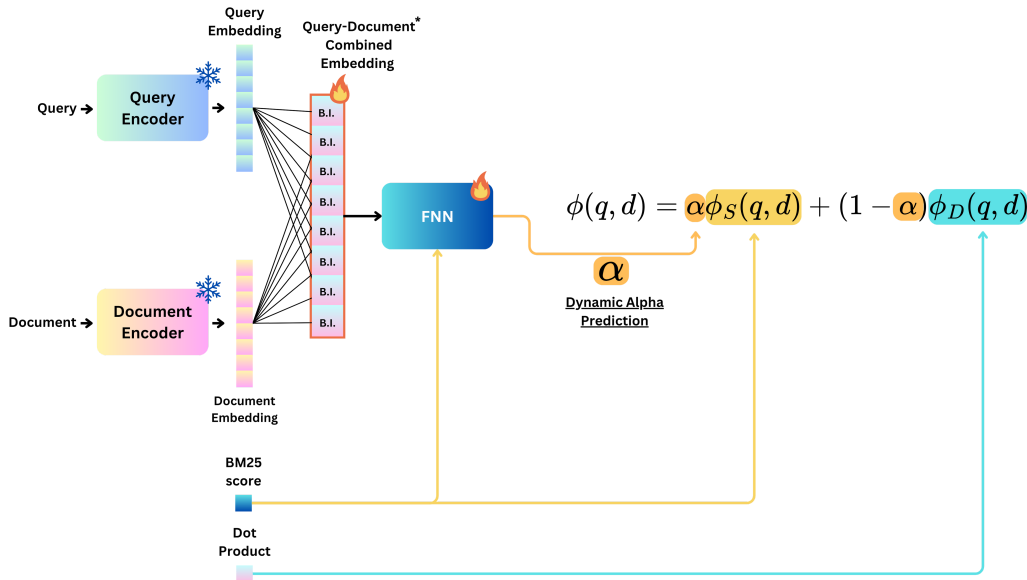


Figure 4.13: NTN Dynamic Alpha adaptor. *The Hadamard and absolute difference vectors between q and d are also used as inputs for the FFNN, however this is not shown here for simplicity.

4.6.4 Adaptors: Query-Level Dynamic α Prediction

Query-Level NTN-DA

This method is similar to the previously described one (NTN-DA, Document-level), however, the main difference is that instead of predicting an alpha value for each document in the given set of top- k candidate

documents, it produces a single value per query, applying the same interpolation criterion for all document results for that query.

Formally, let the output $\mathbf{o}_i$ of the NTN feature extraction of i 'th document for query q be defined as in equation (4.11). Moreover, let each query and the i 'th document's embedding be combined into feature vector $\mathbf{c}_i$, obtained by using a Hadamard product, and their absolute difference, just as described in the previous section under equation (4.12). Additionally, let the difference between the normalized BM25 scores and the dot-products for each query-document be used as a feature, instead of purely the BM25 score as:

$$\delta_i = \text{BM25}(\mathbf{q}, \mathbf{d}_i) - \phi_D(\mathbf{q}, \mathbf{d}_i) \quad (4.15)$$

This is done in this method since z-score normalization is applied to both dot-products and BM25 scores, bringing them to a similar scale. Ultimately, each query-document feature vector is assembled by concatenating the aforementioned elements, followed by a pooling operation to summarize all the query-document vectors into a single pooled representation $\mathbf{v}_q$. The mean is used for this purpose, as described under equation (4.16).

$$\mathbf{v}_q = \frac{1}{k} \sum_{i=0}^{k-1} \text{Concat}(\mathbf{o}_i, \mathbf{c}_i, \delta_i) \in \mathbb{R}^{m+2l+1} \quad (4.16)$$

The final value for α is computed for the given query through a FFNN $f : \mathbb{R}^{m+2l+1} \mapsto [0, 1]$:

$$\alpha_q = \sigma(f(\mathbf{v}_q)) \quad (4.17)$$

The process can be appreciated below 4.14.

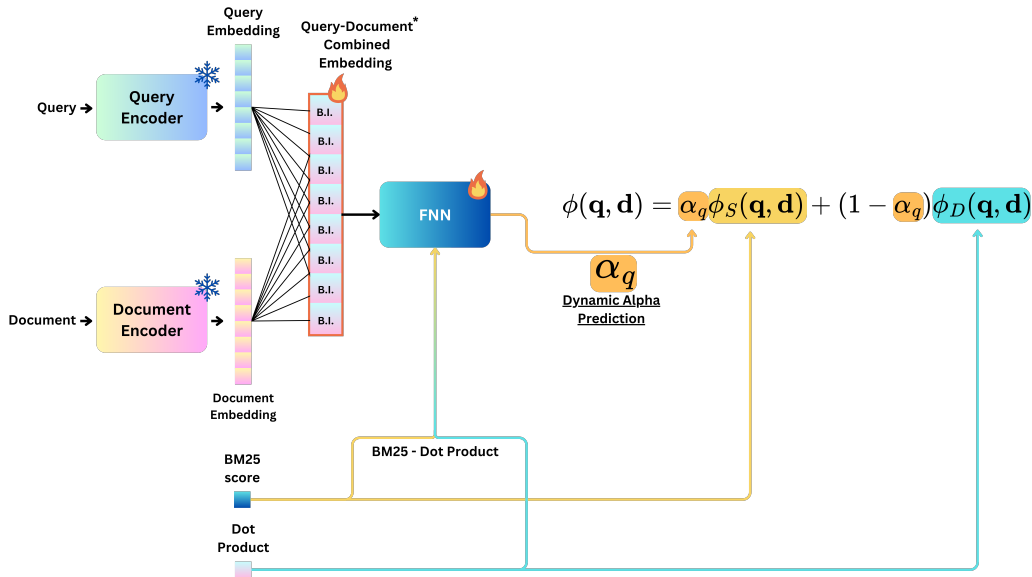


Figure 4.14: NTN Dynamic Alpha adaptor, Per-Query.*The Hadamard and absolute difference vectors between $\mathbf{q}$ and $\mathbf{d}$ are also used as inputs for the FFNN, however this is not shown here for simplicity.

Query-Level FFNN-DA

This model variant also predicts a single α value per query, however, it makes use of a Feedforward Neural Network exclusively, and completely omits the use of the NTN previous to its use. More precisely, this model's architecture is identical to the Query-Level NTN-DA but without the NTN component.

As described in the previous section, let each query embedding and the i 'th document's embedding be combined into feature vector $\mathbf{c}_i$, obtained by using a Hadamard product, and by computing their absolute

difference, as described in equation (4.12). Moreover, let δ_i be the difference between the BM25 score and the dot product for a given query, and the i 'th document in the set of top- k documents coming from the first stage retrieval, as described in equation 4.15. The process can be graphically appreciated below.

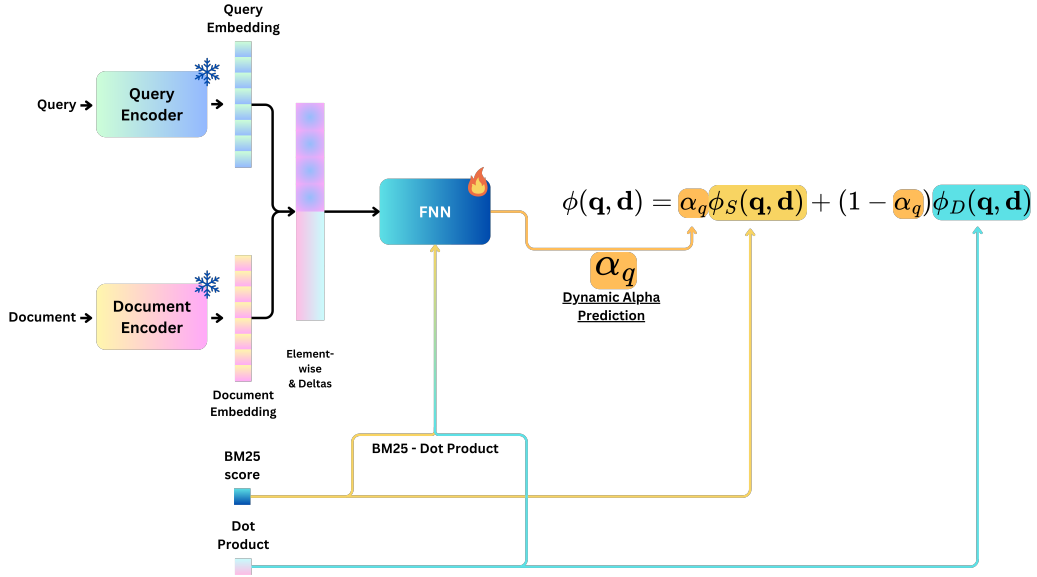


Figure 4.15: FFNN Dynamic Alpha adaptor, Per-Query.

Chapter 5

Experimental Setup

This section outlines relevant experimental details, such as the datasets and baselines employed, training, testing details, and evaluation details.

5.1 Datasets and Baselines

This work makes use of two main datasets:

1. TREC Deep-Learning Track (2019, 2020) [7, 8]: A dataset designed with the goal of studying Ad-hoc ranking for large datasets. It is based on the MSMARCO [4] corpus, and it has two main subsets: Passage ranking and document ranking. Relevance-grading is important for the experiments in this work, therefore the relevance graded test set of TREC-DL-Psg’ 2019 is used for training and validation in an 85%/15% split, and the full test set of TREC-DL-Psg’ 2020 is used for testing purposes.
2. ANTIQUE [12]: ANTIQUE is a dataset consisting of 2626 non-factoid, open-domain questions, spanning a wide range of topics. Of those total queries, 200 are part of a test set. This dataset is particularly useful for this work’s setup as it contains 34,011 relevance-graded annotations. In contrast to TREC-DL-2019-Psg’ 19/20 which has a much smaller number of queries and graded relevance labels. In the case of this work, a split of 2226 queries is used for training, and a split of 200 queries is used for validation.

5.1.1 Baselines

The following baselines are used in this work:

- **BM25 Ranked Baseline:** We use the sparse retrieval BM25 score as a baseline, following the convention set by Leonhardt et al. [23]. A re-ranker in this work is expected to surpass this baseline, as it determines the initial order of retrieved items.
- **Dense Re-Ranking Baselines:** We also include re-ranking scores from pure dot-product re-ranking as a baseline, a practice also adopted by Leonhardt et al. [23]. For these experiments, we use three encoder architectures: TCTColBERT [27], Contriever [16], and TAS-B [13].
- **Interpolation Re-ranking Baselines:** We also include interpolation re-ranking as baseline, where the value for the interpolation constant is searched by employing a grid search. This grid search is defined to be in the range $[0, 1]$, and is done for two decimal digits. That is, every value such that $\{i \cdot 0.01 \mid i \in \mathbb{Z}, 0 \leq i \leq 100\}$. This gives a total of 101 steps. This hyperparameter search is run on the full test set of TREC-DL-PSG’ 2019, which is repurposed as a development/validation set due to its limited size. The final evaluation is conducted on TREC-DL-2020 Passage, which is held out and not consulted during tuning. The hyperparameter search for ANTIQUE is performed on the validation set, and testing on the ANTIQUE test set. The final interpolation constant is chosen by

selecting the value that scored the highest nDCG@10 performance on the development/validation set.

5.2 Evaluation Details

5.2.1 Evaluation Metrics

Since this work deals with re-ranking performance, the reported metrics align with commonly used metrics for re-ranking. The metrics are detailed within section 2.1.1. The employed metrics for this task are specifically:

1. MAP@1k: Provides insight on re-ranking quality since a higher MAP indicates that on average, relevant documents are retrieved within the top-k documents, and also tend to appear earlier in the ranking.
2. nDCG@10: This metric is useful for re-ranking evaluation as it gives insight on how good are the models at ordering documents according to their relevance labels, while also accounting for their correct position in the ranking, penalizing more relevant documents appearing later in the ranking out of order.
3. MRR@10: Mean Reciprocal Rank can be useful to know on average how early the first relevant document appears in the ranking, accounting for the first 10 elements of the re-ranked results.

5.3 Implementation Details

5.3.1 Data Indexing and Preparation

Indexing for first-stage retrieval processes is performed by using Pyserini [25], which in turn creates a Lucene index. The similarity parameters for the BM25 function are defined as: $k_1 = 1.2$, $b = 0.75$, following Elasticsearch’s convention [3]. Moreover, for consistent dataset access, the python library `ir-datasets` [30] is also used, since it provides consistent access to a variety of datasets by using a similar interface for every dataset.

5.3.2 Metric Computation

Since it is desirable to have consistent and reproducible metric computation, the python library `ir-measures` [29] is employed.

5.3.3 Experiment Tracking and Model Storage

Mlflow [47] is used in this work in order to keep track of performed experiments, its results, models and artifacts. This is paired with the use of Amazon Web Services (AWS) to automatically store the resulting trained models and their artifacts in S3, together with any metadata emitted by Mlflow, all stored in an RDS database.

5.3.4 Model Design and Pre-Trained Models

TCT-ColBERT, TAS-B, and Contriever are used as the backbone encoders for the query and document embeddings used in the experiments.

The encoder models are downloaded and used from Huggingface’s [45] model hub. Moreover, the base implementation for TAS-B and Contriever for this setting was adapted from the work of Leonhardt et al.

[23]¹. The implementation the direct alpha optimization technique, and also of all adaptor models, their training and validation was performed by using the Pytorch [33] machine learning framework.

5.4 Model Training Details

5.4.1 Training Hardware

All model training/optimization runs were conducted on a 2021 MacBook Pro with an Apple M1 Pro processor and 16GB of RAM. The CPU was exclusively used for all training and optimization tasks, with no specialized hardware acceleration. This choice was made deliberately to validate the feasibility and efficiency of the proposed techniques on modern, available consumer hardware, demonstrating that they do not require dedicated GPU servers for effective implementation.

5.4.2 Hyperparameter Settings

The hyperparameters for all models were determined empirically. The final configurations for the best-performing models for both datasets on the validation splits, for each of the encoder families are detailed in table 5.1. Moreover, the models are chosen based on their validation nDCG@10 performance.

AdamW was used as the optimizer for all adaptor models, The full set of configurations can be found in the public experimentation repository of the project ².

Table 5.1: Hyperparameter Configurations for best performing models for each dataset and encoder family.

Model	Dataset	Encoder	Learning Rate	Optimizer	Epochs	Batch Size	Hidden Dims	Bilinear Blocks	NTN Dropout
Direct α Opt.	TREC-DL	TCT-ColBERT	1×10^{-1}	SGD	300	2	-	-	-
Direct α Opt.	TREC-DL	Contriever	1×10^{-3}	SGD	200	2	-	-	-
Direct α Opt.	TREC-DL	TAS-B	1×10^{-1}	SGD	300	2	-	-	-
Query-level NTN-DA	TREC-DL	TCT-ColBERT	5×10^{-5}	AdamW	5	4	[1024]	100	0.0
Query-level FFNN-DA	TREC-DL	Contriever	1×10^{-4}	AdamW	29	4	[2048]	-	-
Query-level NTN-DA	TREC-DL	TAS-B	5×10^{-5}	AdamW	29	4	[1048]	100	0.0
Direct α Opt.	ANTIQUE	TCT-ColBERT	1×10^{-3}	SGD	35	4	-	-	-
Direct α Opt.	ANTIQUE	Contriever	1×10^{-1}	SGD	20	4	-	-	-
Direct α Opt.	ANTIQUE	TAS-B	1×10^{-1}	SGD	9	2	-	-	-
Query-level NTN-DA	ANTIQUE	TCT-ColBERT	1×10^{-5}	AdamW	9	4	[1048]	50	0.0
Document-level NTN-DA	ANTIQUE	Contriever	1×10^{-4}	AdamW	5	4	[256, 128, 64, 32, 128]	50	0.4
Query-level NTN-DA	ANTIQUE	TAS-B	1×10^{-4}	AdamW	9	4	[256, 128, 64, 32, 128]	50	0.4

5.5 Elasticsearch Plugin Testing Setup

As described in section 4.1, the plugin was tested by using Rally [10], which allows to simulate a set of clients eliciting search requests to an ES cluster. In order to do this, test runs have to be defined through run JSON configuration files called "tracks". A track is a blueprint of different benchmarking scenarios with a specific corpus. It is also used to specify the involved indices, data files and the operations that are employed. To this end, a custom JSON track run was set up, together with a python script used to assemble the body of each test request. The run file defines two main operations:

- **BM25 Search:** A standard BM25 Elasticsearch keyword search.
- **Rescore Search:** A two-step process beginning with a BM25 search, and followed by re-ranking of the top results using the rescoring methodology.

¹<https://github.com/mrjleo/fast-forward-indexes/tree/main>

²<https://github.com/TUDThesis/experimentation>

5.5.1 Infrastructure Setup

As mentioned, dedicated compute instances were used for each of the testing components. In this case, the compute components are defined as follows:

- **Elasticsearch Node:** The instance running the Elasticsearch server, together with the installation of the plugin. The Elasticsearch service was defined as a single node.
- **Load Generator Node (Rally):** The instance running the Rally testing tool. This instance executes the defined test "race" and sends the requests to the ES node, simulating real usage.
- **Query-Encoder Node:** Runs the query-encoder service and listens for requests coming from the ES node.

Hardware and Software Specifications

The hardware and software specifications for the dedicated server instances used for testing are provided in table 5.2. All instances were provisioned within the same geographic region (Amazon Web Services servers in Stockholm).

Table 5.2: Hardware and Software Specifications of the Experimental Setup

Component	Instance Type	vCPU	Memory (GiB)	GPU	VRAM (GiB)	Storage	OS / Main Software Components
Elasticsearch Node	m6i.xlarge	4	16	-	-	512 GiB gp3 (3000 IOPS)	Amazon Linux 2023 Elasticsearch 9.1.2 OpenJDK 17 (8 GiB Heap) FFWI-Rescore Plugin
Load Generator Node (Rally)	c6i.large	2	8	-	-	64 GiB gp3	Amazon Linux 2023 Rally 2.12.0
Query Encoder Node	g4dn.xlarge	4	16	NVIDIA T4	16	125 GB gp3	Ubuntu 22.04 LTS PyTorch transformers Auto-Alpha FastAPI Uvicorn

5.5.2 Dataset and Index Configuration

The experiment was conducted using the MS MARCO passage collection, which is a standard benchmark for information retrieval tasks, containing 8.8 million passages. Each passage was indexed along with its corresponding pre-computed dense vector representation, which was generated using the TCT-ColBERT encoder model. The performance evaluation was focused on a single-node environment, thus the Elasticsearch index was configured with one primary shard and zero replicas. The index mapping defined two core fields: `passage`, mapped as a standard text field for full-text search, and `vector_field`, which contained the dense embeddings. The `vector_field` was mapped as a float array with indexing disabled, as the rescoring logic is managed by the plugin, making native vector indexing unnecessary and saving storage space. The vector data was explicitly stored with the document, enabling a custom plugin to access the raw embedding for rescoring during the search phase. The complete specification for the used index mapping is available in the Appendix A.2.1.

5.5.3 ES Rally Test Setup

The benchmarking workload was structured into two primary test procedures: a baseline `bm25-search` and an experimental `rescore-search`. The baseline test simulated a standard search scenario. It was

configured to run with 4 client threads for a 240-second period, four times, aiming to maintain a constant throughput of 18 queries per second (qps), divided among the clients.

The main experiment focused on evaluating the performance of the custom dense vector rescoring logic. This series of tests systematically varied the rescore window size, which determined the number of top documents from the initial BM25 retrieval phase to be re-ranked, to measure its impact on service time and throughput. For each of the window sizes, a corresponding target throughput was set based on preliminary tests to establish a sustainable load for the system. The four configurations were:

- Window size of 100 documents: 18 qps
- Window size of 250 documents: 8 qps
- Window size of 500 documents: 3.7 qps
- Window size of 1000 documents: 1.8 qps

The complete Rally track definition, including all operation parameters, is available in the appendix, under section A.2.2.

The first 20,000 queries from the training query subset of MSMARCO-Passage were used during the testing procedure with rally.

5.6 Code Implementation

All the code used for this work is openly available within the following GitHub repositories:

- Elasticsearch Plugin: [TUDThesis/FFWI_es_plugin](https://github.com/TUDThesis/FFWI_es_plugin)³
- Overall Experimentation: [TUDThesis/experimentation](https://github.com/TUDThesis/experimentation)⁴
- Auto-Alpha Toolset: [TUDThesis/alpha_auto_tune](https://github.com/TUDThesis/alpha_auto_tune)⁵

³https://github.com/TUDThesis/FFWI_es_plugin

⁴<https://github.com/TUDThesis/experimentation>

⁵https://github.com/TUDThesis/alpha_auto_tune

Chapter 6

Results and Discussion

This section presents the results from the alpha optimization methods and adaptors (section 6.1.1), and also provides results for testing performed on the Elasticsearch re-ranking plugin, in order to determine its expected service time and throughput capabilities at the current state, under the setup described in the experimental setup section 5.

6.1 Direct α optimization & Adaptor Models

6.1.1 Aggregated Results

This subsection presents the obtained results after training and testing the methods outlined in sections 4.5, and 4.6. The results are aggregated in table 6.1. The best performing outcomes are emboldened for each dataset/encoder group. Moreover, a dagger “†” superscript is used to mark the results for which there was an observed improvement over the grid search alpha baseline, and that were also statistically significant. A paired t-test was performed, using a significance level of 95% and correction for Family-Wise Error Rates using the Holm-Bonferroni method[42].

This section is used to outline the results, and to provide a discussion that helps to answer RQs 1 and 2 of this work:

- **RQ 1:** How can learning-based methods optimize the interpolation weight α in hybrid re-ranking?
- **RQ 2:** Can lightweight adaptors improve performance when using pre-computed embeddings from non-domain-specific encoders?

Results Structure

The performance of the proposed adaptor models and direct α optimization techniques are evaluated on the test splits for datasets TREC-DL-Psg’20 and ANTIQUE. AP@1k, nDCG@10, and MRR@10 are used as the primary evaluation metrics, and the results are given for the encoders: TCT-ColBERT [27], Contriever [16], and TAS-B[13].

The baselines used include the first-stage retrieval BM25 performance, the re-ranking dot product performance (dense re-ranking), and the performance of interpolated re-ranking using a grid search to find a global value for α . The grid search for alpha made use of two decimal places, that is, 101 search steps of size 0.01 in the range $\alpha \in [0, 1]$. More details are provided under section 5.1.1.

The final interpolation constant values found through the optimization methods are shown in table 6.1 under columns α_{TREC} and α_{ANTIQUE} , corresponding to TREC-DL-PSG’20 and ANTIQUE datasets respectively. For the models that do not make use of an interpolation constant, this does not apply, thus “-” is used as placeholder. Models predicting dynamic alpha values have alpha reported with the mean, plus-minus one standard deviation.

Table 6.1: Re-ranking performance on TREC-DL-Psg’20 and ANTIQUE (BM25 first-stage retrieval). Best results for each metric **in bold** denote the top performance within that backbone family. The underlined value indicates the single best overall result for that metric across all models. Hyperparameter search and direct alpha optimization models use a fixed global value for alpha. Such values are reported in the first column for each dataset group: α_{TREC} and $\alpha_{\text{ANTIQU}}E$, respectively. Models using dynamic alpha values have alpha reported with the mean, plus-minus one standard deviation.*MRL stands for Margin Ranking Loss (direct α optimization with that objective). Baseline methods are indicated with the superscript ^B. Dagger superscripts [†] indicate statistically significant improvements over the strongest baseline for each subgroup (Hyperparameter grid search), using a paired t-test with a significance level of 95%, and adjusting for Family-Wise Error Rates using Holm-Bonferroni [42].

Model & Method	TREC-DL-Psg’20				ANTIQU			
	α_{TREC}	AP _{1k}	nDCG ₁₀	MRR ₁₀	$\alpha_{\text{ANTIQU}}E$	AP _{1k}	nDCG ₁₀	MRR ₁₀
<i>Global Baseline</i>								
BM25 ^B	-	0.2827	0.4770	0.6653	-	0.2118	0.4541	0.5917
<i>TCT-ColBERT Encoder</i>								
↳ Dot-product ^B	-	0.4403	0.6789	0.8092	-	0.2430	0.4287	0.6493
↳ Grid search ^B	0.14	0.4502	0.6892	0.8161	0.1	0.2738	0.4914	0.6862
↳ Direct α opt.	0.1368	0.4500	0.6899	0.8153	0.4669	0.2723	0.5273 [†]	0.6787
↳ Direct α opt. (MRL)*	0.0181	0.4449	0.6842	0.8184	0.0634	0.2668	0.4726	0.6845
↳ NTN	-	0.4136	0.6389	0.7888	-	0.2614	0.4890	0.6888
↳ FFNN	-	0.3907	0.6049	0.7483	-	0.2377	0.4326	0.6528
↳ Document-level NTN-DA	0.4804 ± 0.0023	0.4130	0.6467	0.7701	0.5048 ± 0.0001	0.2799	0.5351[†]	0.6967
↳ Query-level FFNN-DA	0.1252 ± 0.0048	0.4473	0.6901	0.8097	0.3080 ± 0.0034	0.2824	0.5347 [†]	0.6960
↳ Query-level NTN-DA	0.1268 ± 0.0091	0.4466	0.6939	0.8212	0.2518 ± 0.0022	0.2861[†]	0.5325 [†]	0.7063
<i>Contriever Encoder</i>								
↳ Dot-product ^B	-	0.2724	0.4668	0.5626	-	0.2372	0.4425	0.6644
↳ Grid search ^B	0.09	0.3107	0.5224	0.6829	0.03	0.2705	0.5141	0.6796
↳ Direct α opt	0.5002	0.2872	0.4910	0.6696	0.0312	0.2717	0.5157	0.6829
↳ Direct α opt. (MRL)	0.2114	0.2976	0.5160	0.6700	0.0	0.2377	0.4433	0.6696
↳ NTN	-	0.3032	0.5079	0.6533	-	0.2266	0.4716	0.6048
↳ FFNN	-	0.2823	0.4779	0.6653	-	0.2113	0.4526	0.5914
↳ Document-level NTN-DA	0.4091 ± 0.0108	0.2976	0.5166	0.6709	0.1349 ± 0.0215	0.2771	0.5358[†]	0.6924
↳ Query-level FFNN-DA	0.3308 ± 0.0129	0.3116	0.5222	0.6698	0.2752 ± 0.0096	0.2739	0.5309 [†]	0.6801
↳ Query-level NTN-DA	0.4650 ± 0.0032	0.3098	0.5180	0.6736	0.3477 ± 0.0622	0.2729	0.5337 [†]	0.6815
<i>TAS-B Encoder</i>								
↳ Dot-product ^B	-	0.4610	0.6915	0.8458	-	0.2143	0.4055	0.5052
↳ Grid search ^B	0.38	0.4635	0.6997	0.8256	0.26	0.2386	0.4550	0.5527
↳ Direct α opt	0.1846	0.4699	0.7025	0.8503	0.5270	0.2608 [†]	0.5044 [†]	0.6054 [†]
↳ Direct α opt. (MRL)	0.0316	0.4639	0.6969	0.8550	0.3950	0.2497	0.4796	0.5735
↳ NTN	-	0.4048	0.6268	0.8139	-	0.2346	0.4748	0.6178
↳ FFNN	-	0.4097	0.6410	0.7509	-	0.2487	0.4777	0.6545
↳ Document-level NTN-DA	0.4785 ± 0.0004	0.4701	0.7043	0.8591	0.5251 ± 0.0003	0.2526 [†]	0.4854 [†]	0.5792 [†]
↳ Query-level FFNN-DA	0.0726 ± 0.0048	0.4735	0.7037	0.8576	0.3201 ± 0.0075	0.2694 [†]	0.5256 [†]	0.6469 [†]
↳ Query-level NTN-DA	0.0053 ± 0.0019	0.4642	0.7048	0.8519	0.3866 ± 0.0119	0.2702[†]	0.5274[†]	0.6524[†]

Discussion of Main Findings

Dynamic α optimization methods making use of an NTN/FFNN, and the Direct α optimization (Neural nDCG) methods are the best performing ones, however, their effectiveness varies based on the specific encoder and dataset. Notably, direct alpha optimization and all dynamic alpha model variants proved

specifically stronger for the ANTIQUE dataset, across all encoders, but showed less improvements for the case of TREC. For instance, using the TCT-ColBERT encoder, the Document-level NTN-DA model achieves a 8.89% relative improvement in nDCG@10 over the grid search baseline on ANTIQUE, whereas the Query-Level NTN-DA model yields a mere 0.68% relative gain on TREC-DL. This trend, visualized in Figure 6.1, is also observed across the other dataset/encoder cases. This behavior can possibly be explained by factors such as:

- **Domain Specificity:** The proposed methods showed promise in out-of-domain scenarios. ANTIQUE is a dataset that none of the backbone encoders were pre-trained on. This is reflected in their low baseline dense re-ranking performance, which often failed to surpass pure BM25 retrieval performance. In this more complex setting, the dynamic α models provided adaptation, resulting in improvements, which were often statistically significant.
- **Performance Saturation and Data Scarcity:** The TREC-DL-2019 Psg’ dataset is built from MSMARCO, a corpus that both TCT-ColBERT and TAS-B were partially trained on [26, 13]. This in-domain advantage resulted in an already robust grid search baseline, leaving less room for improvements. This effect was likely reinforced by the small training set used for TREC-DL (38 queries) compared to ANTIQUE (2,226 queries), making it difficult for the models to learn significant improvements.

While dynamic and direct optimization strategies were mostly successful, not all adaptation approaches worked. The attempt to replace dense scores with standalone FFNN or pure NTN models often resulted in lower performance than the baseline. A possible explanation is that the dot-product signal from modern PLMs like TCT-ColBERT is already highly optimized, and learning a more complex scoring function from scratch with limited data proved to be an ineffective approach, compared to the optimized interpolation of the already existing strong signals.

The full relative performance matrix is also provided under the Appendix, in section A.1.

Effectiveness of Dynamic Alpha Prediction Methods Per Encoder Family

Dynamic alpha prediction models had a strong performance, especially the query-level variants. These models commonly outperformed the strongest baseline, often with statistical significance. With the TCT-ColBERT backbone on ANTIQUE, all three dynamic alpha models (Document-level NTN-DA, Query-level FFNN-DA, and Query-level NTN-DA) achieved statistically significant improvements in nDCG@10 over the grid search baseline, with Document-level NTN-DA achieving the highest score at 0.5351. Moreover, the Query-level NTN-DA model achieved the best overall performance on this dataset, attaining the highest AP@1k (0.2861), and MRR@10 (0.7063).

For the Contriever backbone, which shows the weakest performance of the three encoders, the dynamic alpha models provide some improvements. The Document-level NTN-DA model achieves the best results for this family on ANTIQUE, across all metrics: AP@1k (0.2771), nDCG@10 (0.5358), and MRR@10 (0.6924), demonstrating its ability to enhance a less effective base encoder.

For the TREC-DL dataset, the dynamic models attained the best overall performance: Query-level FFNN-DA achieved the highest AP@1k (0.4735), while Query-level NTN-DA achieved the top nDCG@10 (0.7048), and the Document-level NTN-DA model achieved the top MRR@10.

For the TAS-B based models in ANTIQUE, all dynamic prediction models obtained a higher performance than the baseline, while also being statistically significant.

Overall, results suggest that learning to predict the interpolation parameter on a per-query, or per-document fashion is a more effective strategy than relying on a single, globally obtained value.

Comparison of Direct Alpha Optimization and Adaptor Models

The Direct α optimization technique, which uses a list-wise objective (Neural nDCG) to find an optimized global value for α , showed that it can be used as a competitive and efficient alternative to a grid search. It



Figure 6.1: Relative gain (%) of each metric, for the best-performing methods over the Hyperparameter Search baseline for each encoder and dataset. Gains are computed as $\frac{\text{Best-Hyperparam}}{\text{Hyperparam}} \times 100$.

achieved equal or better results in most dataset/encoder cases. On the ANTIQUE dataset with the TAS-B encoder, it attained statistically significant improvements across all metrics. However, the performance of optimizing α with Margin Ranking Loss (MRL) was generally less consistent and often underperformed compared to the NeuralNDCG-based approach and the dynamic models. This phenomenon can be potentially explained by the richer training signals obtained through a list-wise objective, compared to the pair-wise counterpart.

6.1.2 Tradeoffs: Model Complexity and Training Time

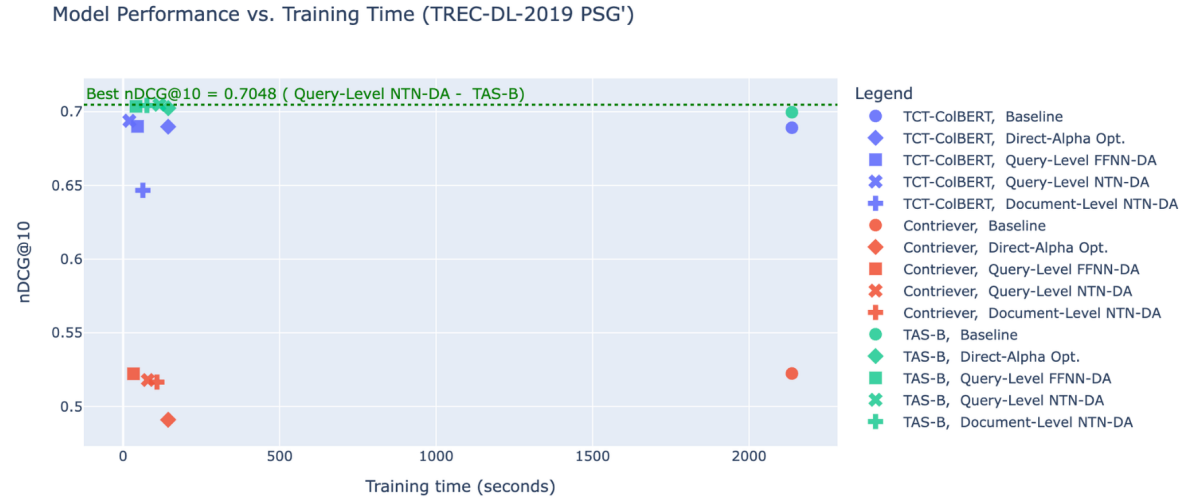
This section compares some of the tradeoffs between the adaptor models, the direct alpha optimization technique, and the baseline hyperparameter search. Two main factors are analyzed: total number of trainable parameters, and training time. These factors are plotted against the attained nDCG@10 score for each encoder and model. The reason nDCG@10 is used as the comparison metric is its usefulness for this re-ranking setting, where nDCG can express how good is the learned model for predicting scores that accurately reflect the relevance of each of the re-ranked documents in their appropriate order. Moreover, the training time and total number of model parameters are two practical and interesting features which play an important role to analyze training and inference efficiency.

Training Time vs. nDCG@10

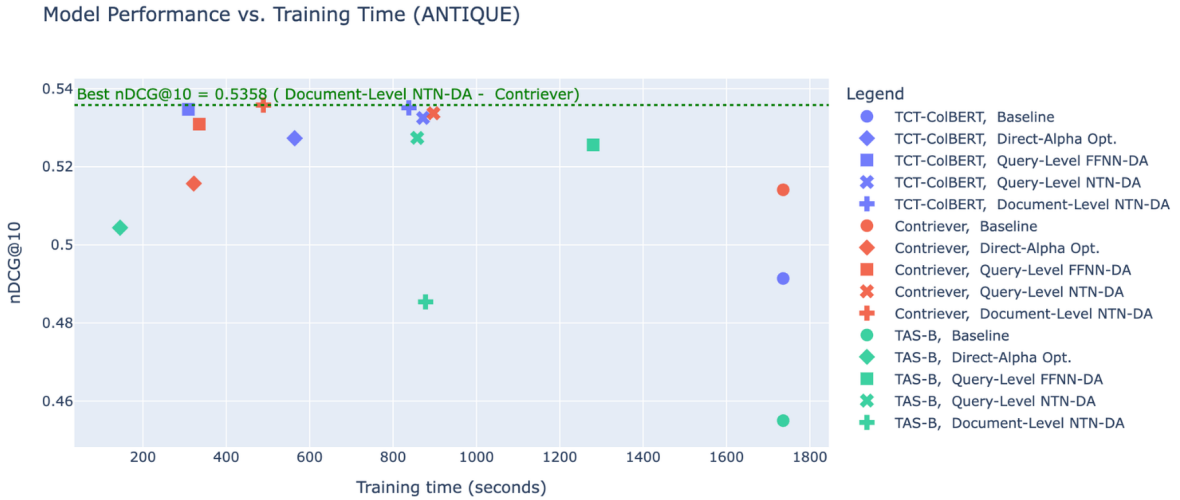
The tradeoff between training times and attained nDCG@10, for each of the adaptor models, can be appreciated from the figures below 6.2. Colours correspond to the encoder family used, and shapes correspond to the type of model/technique.

Sub-figure (A) depicts the case for which training was performed on the TREC-DL-2019 PSG' test set, and also displays the time that it took for the grid search of the baseline fixed value of alpha to run. Adaptors took much less time to train overall, compared to the grid-search baselines, which took more than 2,100 seconds on average. In contrast, all adaptor models were successfully trained within 110 seconds for this dataset, where Contriever-based Document-level NTN-DA model took the longest training time, at 108 seconds, and the TCT-ColBERT-based, Query-Level NTN-DA model took the least, with 20 seconds. Additionally, direct alpha optimization ran within 145 seconds. The main factor for the varying training times was the number of training epochs, since batch sizes used during training were identical for all models.

In a similar structure, sub-figure (B) figure shows the same tradeoffs of training time and nDCG@10, but for ANTIQUE. Note that compared to TREC, training of the adaptor models took longer on average due to the larger set of examples, however, the training times still remained lower than the time it took for the grid-search of alpha to run. Adaptor models outperformed every baseline in terms of nDCG@10 performance for this dataset, for each encoder family respectively.



A) TREC-DL-PSG' 2019



B) ANTIQUE

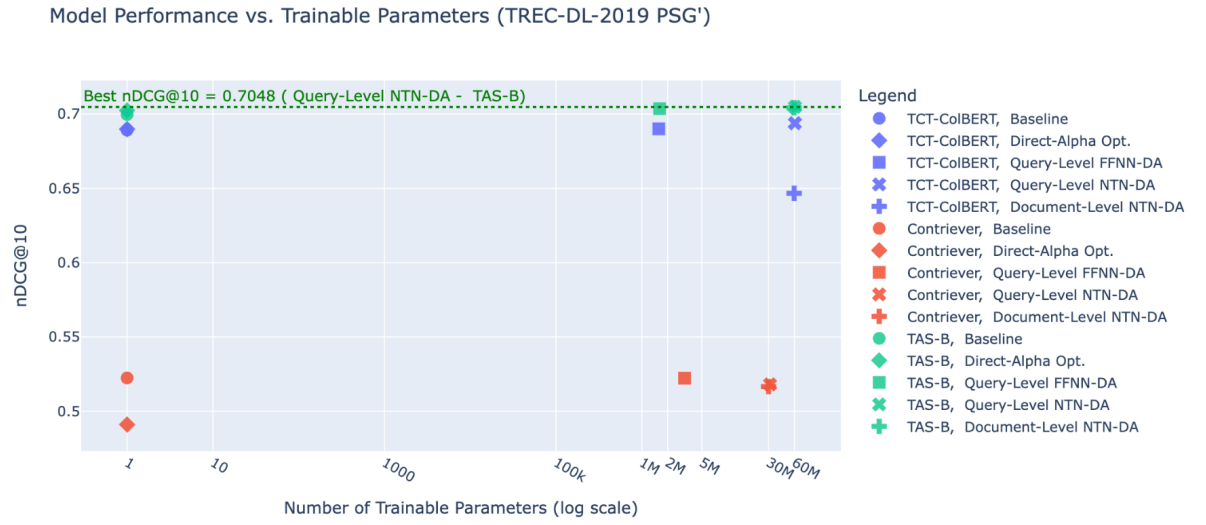
Figure 6.2: Training time in seconds (x-axis) and the resulting nDCG@10 for each encoder-model combination A) TREC-DL-2019 PSG'), B) ANTIQUE. Colors represent a family of encoder models used for the query and document dense embedding representations. Each marker represents a specific model architecture / technique. For TREC, training times for adaptor models were at least approximately $19\times$ lower than the alpha grid-search process. Training times for ANTIQUE models remained lower than hyperparameter search baseline, but were considerably higher on average, due to the larger training set size.

Trainable Parameters

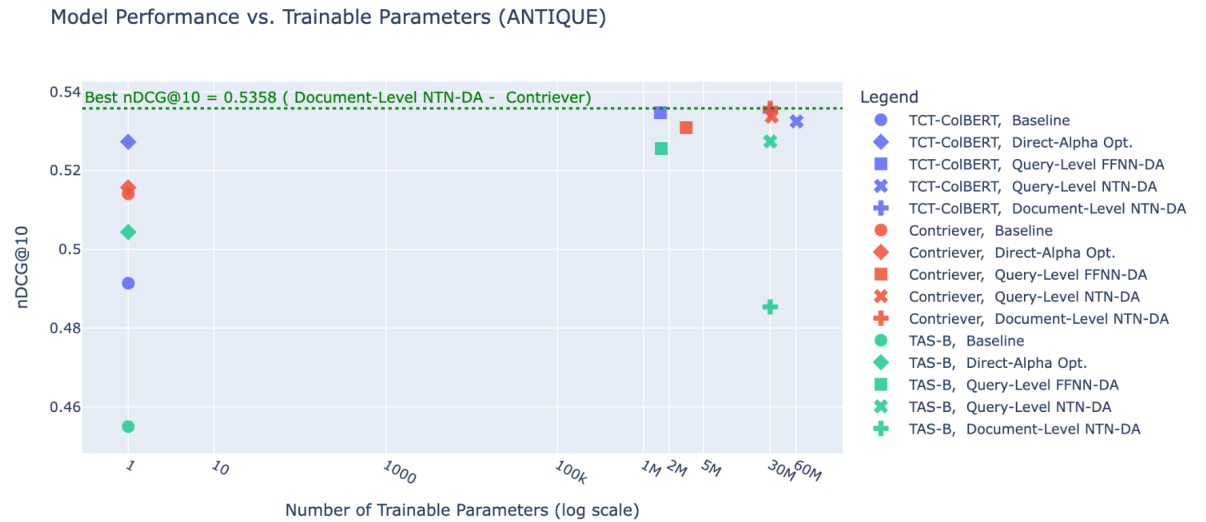
Figure 6.3 in this subsection compares the number of trainable parameters for each model/technique, with respect to their nDCG@10 performance. Sub-figure (A) shows the tradeoffs for models trained on TREC-DL-2019 Psg', whereas sub-figure (B) does it for models/techniques trained on ANTIQUE. Overall, best performance in terms of nDCG@10 is attained by NTN-based models (Query-Level NTN-DA, and Document-Level NTN-DA respectively), at the cost of a larger parameter footprint. In contrast,

FFNN-based models have a much lower parameter count. Regardless of the smaller parameter space, Query-level FFNN-DA models closely followed, or sometimes outperformed Query/Document-level NTN-based models, in terms of nDCG@10 performance, making them efficient and effective alternatives to the NTN-based dynamic alpha models.

Interestingly, direct alpha optimization outperformed the Document-level NTN-DA model for TREC/ColBERT and also for ANTIQUE/TAS-B. A possible explanation for this phenomenon is the fact that NTN-DA models have more hyperparameters to be considered, and are more sensitive to their selection. Moreover, there is a tradeoff with respect to the performance, and complexity of the models for training purposes: the optimization landscape of the single-parameter, direct alpha optimization technique is much simpler than the optimization landscapes of the more parameter-dense dynamic models.



A) TREC-DL-PSG' 2019



B) ANTIQUÉ

Figure 6.3: Total number of trainable parameters (x-axis) and the resulting nDCG@10 for each encoder-model combination. Colors represent a family of encoder models used for the query and document dense embedding representations. Each marker represents a specific adaptor model architecture.

6.1.3 Feature Correlations: Underlying Query/Document-Level NTN-DA model mechanisms

To potentially interpret some of the learned mechanisms that drive changes in alpha within the dynamic alpha models, Spearman’s rank correlation coefficients were computed by considering features extracted from results of runs on the test splits for both TREC-DL-2020 PSG’, and ANTIQUE. Since low-level embedding features are difficult to interpret, more high-level data features were calculated, such as: query lengths, query Shannon entropies, document lengths, etc.

Figure 6.4 shows four spearman correlation heatmaps, each showing correlation coefficients between features from TREC-DL-2020 PSG’/ANTIQUE test splits. Crucially, this analysis provides the correlations between high-level features and the dynamic α values predicted by the document and query-level NTN-DA Contriever-based models. Contriever-based models were selected since they displayed the largest variance for the dynamic predictions of α , as shown in table 6.1.

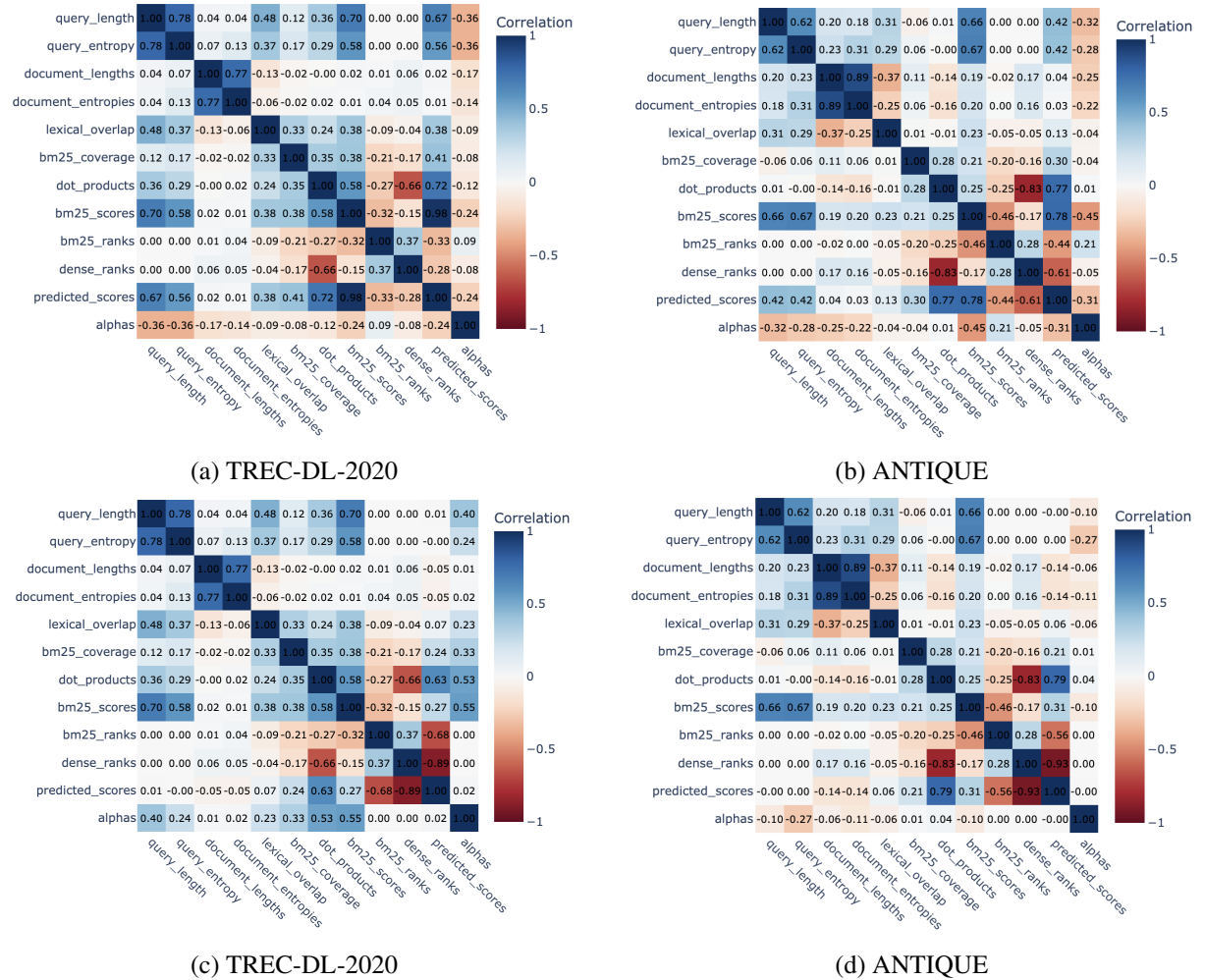


Figure 6.4: Correlation for features and predicted α values across the test split during testing, the figures on the top row correspond to correlation maps between the high-level data features and the predictions for the Document-level Dynamic Alpha models (NTN-DA), and the bottom row corresponds to correlation maps between the high-level data features and the predictions for the Query-level NTN Dynamic Alpha models.

Document-Level NTN-DA Models

The first two correlation matrices (sub-figures A, and B) correspond to the document-level dynamic alpha NTN-based models. There are multiple observable non-zero correlations occurring between alpha, and the rest of the features. Focusing on the query length and query entropy features, their observed negative correlations suggests that whenever a query is more semantically challenging or "rare", the dense model contributes more to the final score. No explicit entropy or length features are used as part of model predictors, signaling that the models likely make use of implicit frequency and term co-occurrence statistics, which could translate into entropy statistics present in the query embeddings. These feature can also possibly be derived from the learned interactions between document and query embeddings within the models' components. This view aligns with previous research that has shown that dense embeddings features from models such as BERT, correlate with term frequency and co-occurrence statistics [24, 19]. Furthermore, note that there are correlations between document length and entropy features, in contrast to the query-level models, which show very weak correlations between those features and alpha values. This suggests that document complexity could also be a decisive factor for alpha's value. These negative document correlations, much like their query feature counterparts, may signal that as a document's rarity or complexity increases, its alpha value decreases. As a consequence, the dense score gains more importance relative to the BM25 score.

Negative correlations are also observed between alpha predictions and BM25 scores. This could be explained by revisiting the previous analysis of term frequency. Less frequent terms in a query tend to produce higher BM25 scores, particularly for documents of average or shorter length. As a result, documents matching these rarer query terms will, on average, receive a higher BM25 score.

Dot product correlation with alpha is weaker on both datasets, possibly indicating that the learned mechanism relies mostly on BM25 signals, entropy signals, query entropy, and length cues to predict alpha values. Interestingly, negative correlation with BM25 scores is stronger on ANTIQUE. A possible explanation to this phenomenon is that ANTIQUE is a non-factoid Q&A dataset based on Yahoo! Answers [12], where questions and answers can be more informal, and overall include less common terms and more slang terms.

Query-Level NTN-DA Models

The second row of subfigures (C) and (D) show the correlation matrices for the Query-Level NTN-DA models, having Contriever as their query/document encoder backbone. Focusing on the correlations with predicted alpha values, it is interesting that the correlations with respect to document features like length and entropy are lower than the ones observed for the correlation maps corresponding to the Document-level NTN-DA models. An intuitive explanation would be that in query-level models, a single alpha is predicted for the entire set of top-N documents. For this weight to work across all documents, the model must aggregate a single effective score. Therefore, the model likely relies more on query-related features, which likely contain information about the expected retrieved results as a whole.

A second interesting detail is that correlations with respect to alpha values, for the TREC dataset are positive. This could indicate that the model learned to "trust" the BM25 scores. This is not entirely surprising due to the composition of the TREC-DL-2019 PSG' dataset, whose relevance labels were assigned by expert annotators, on top pooled MSMARCO documents which were most likely to be relevant [7], in contrast to ANTIQUE, which was annotated using crowd-sourcing, and is based on Yahoo! Answers users' responses, which are commonly highly noisy, ambiguous, and often contain informal language and grammatical mistakes [12].

Finally, sub-figure (D) signals more reliance on the dense score as the complexity of queries and documents increases, but mostly of queries. This is also not surprising due to ANTIQUE's previously described nature.

6.1.4 Discussion: Answering RQ1, and RQ2

Given the previous results and discussion, an answer can be provided for research questions RQ1, and RQ2:

RQ1: How can learning-based methods optimize the interpolation weight α in hybrid re-ranking?

This work showed that learning-based methods can effectively optimize the interpolation weight α , outperforming traditional grid search in most cases when training data is available. Two main approaches were proposed:

- **Direct Optimization:** Optimizing α directly using a Neural nDCG objective function proved to be a highly effective and efficient alternative to grid search, achieving on-par or superior results in most cases.
- **Dynamic Prediction:** The most effective strategy was to predict α dynamically for all results to re-rank for the given query, or for each query-document pair. This family of models mostly outperformed both the grid search baseline and the direct optimization method, showing the benefit of a query/document-specific interpolation weight over a fixed global one.

RQ2: Can lightweight adaptors improve performance when using pre-computed embeddings from non-domain-specific encoders?

Obtained results suggest that lightweight adaptors can effectively improve performance, though their impact is influenced by the initial effectiveness of the pre-trained encoder on the target dataset.

The adaptors demonstrated their highest value on the ANTIQUE dataset, yielding statistically significant performance gains over the baseline. It is hypothesized that this occurs because of ANTIQUE’s more extensive training data, and the base models performed poorly out-of-the-box. In this scenario, the adaptors provided an important and quantifiable benefit.

In contrast, on the TREC-DL-2020 dataset, the performance improvements were not statistically significant. The base encoders, TCT-ColBERT and TAS-B, had already been trained on a subset of MSMARCO, making them less ”non-domain-specific” for this task. It is theorized that their already high baseline performance make further significant improvements with the adaptors more difficult to achieve, coupled with the lower number of training examples.

In general, the evidence suggests that dynamic α adaptors are a valuable tool for enhancing performance, especially when a pre-trained encoder is applied to a new domain where it initially struggles.

Conversely, not all adaptation strategies were successful. Attempts to use FFNN and pure NTN models as standalone adaptors, and attempting to learn a new dense scoring function resulted in a general decline in performance. These models were outperformed by all methods that simply optimized the interpolation of existing scores. The most likely explanation is that the base dot-product signal is already exceptionally strong. The backbone PLMs are exceptional at producing embeddings where the dot-product is an efficient and informative measure of relevance. For instance, TCT-ColBERT distills ColBERT’s MaxSim operator, which aggregates dot-product similarities across all token representations [26]. This means its final dot-product is already the result of a very optimized procedure. Attempting to learn a more complex scoring function from scratch, therefore, proved to be an ineffective strategy.

6.2 Plugin Performance Testing

The Elasticsearch re-ranking plugin was tested according to the methods and setup described in sections 4.4 and 5 respectively. Service time and throughput metrics are used below for the analysis.

6.2.1 Service Time Measurements

Figure 6.5 shows each of the used window sizes for experimentation, and their respective service time percentiles (50th, 90th, and 99th) in milliseconds, for the BM25 searches with the Re-ranking plugin enabled. There is an almost linear trend between an increase in the window size compared to the service time. Naturally, the best performing window size was for the top-100 documents, at a median service time of 214.23 milliseconds.

It is insightful to know what is the proportion of the total service time that is used on the first-stage retrieval phase, and on the re-ranking phase. To this end, the stacked 50th percentile service times for pure BM25 retrieval, and for the Re-ranking step are presented in figure 6.6. The average observed 50th percentile for the first stage retrieval was of approximately 25 milliseconds, depicted as the blue slices from the figure. In contrast, the overhead of the rescore plugin is much larger, beginning with 188.42 ms for the rescore window size of 100 documents. This larger overhead can possibly be explained by the fact that accessing the pre-computed dense embedding representation for each document requires additional I/O operations after the first stage retrieval phase, in order to fetch the embeddings from disk, or cache if available. Moreover, dot product operations are executed as a single-threaded, CPU-bound procedure on the ES node. The lack of parallelization for these computations creates a potential bottleneck. This is left as an area for future exploration and improvement.

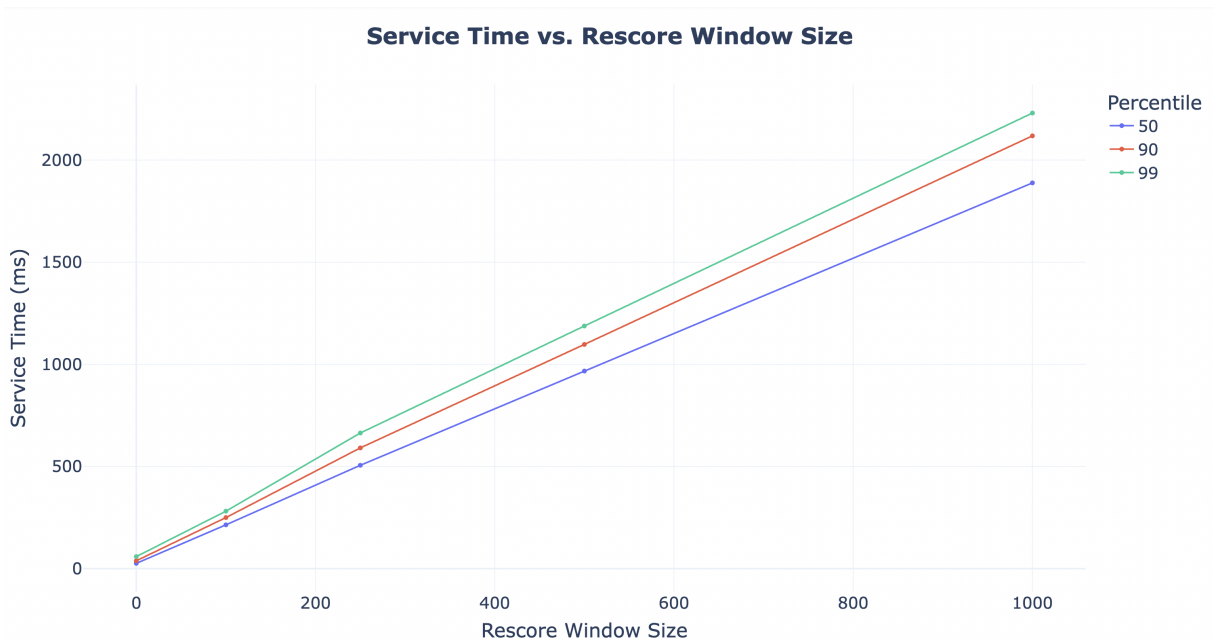


Figure 6.5: Window Size (x-axis) vs. Service Time (y-axis) at the 50th, 90th, and 99th percentiles, measured across window sizes: 100, 250, 500, and 1000 top-k documents to re-rank.

Throughput Performance

The throughput performance can be a useful metric to decide the extent to which an ES cluster can process requests without getting overwhelmed. Thus, the throughput is calculated as part of the testing procedure. The mean throughput, for each run corresponding to each window size test is shown in figure 6.7. Throughput exhibits an inverse-like decay with window size, and not an almost linear relationship like it was observed previously for the service time measurements. This is an expected tradeoff, since a larger service time means that the total number of processable operations per second lowers when

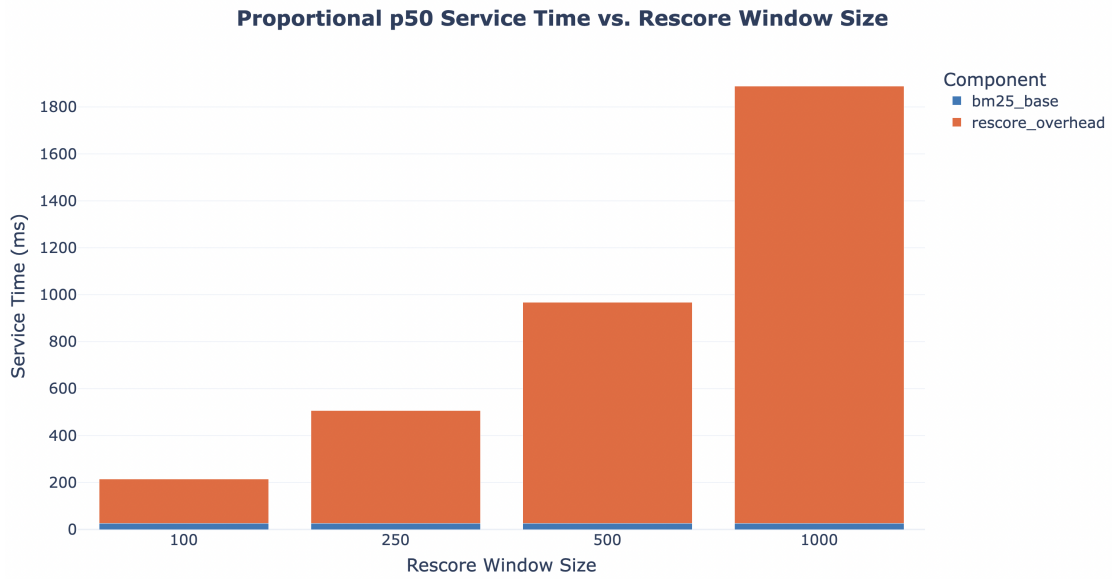


Figure 6.6: Window size (x-axis) vs. 50th percentile service time (y-axis) for first-stage retrieval, and for re-scoring phase respectively. Measurements are shown across window sizes for 100, 250, 500, and 1000 top-k documents to re-rank.

using the same amount of resources. That is, the more documents that are considered for the re-ranking procedure, the slower queries become, making queries per second decrease inversely.

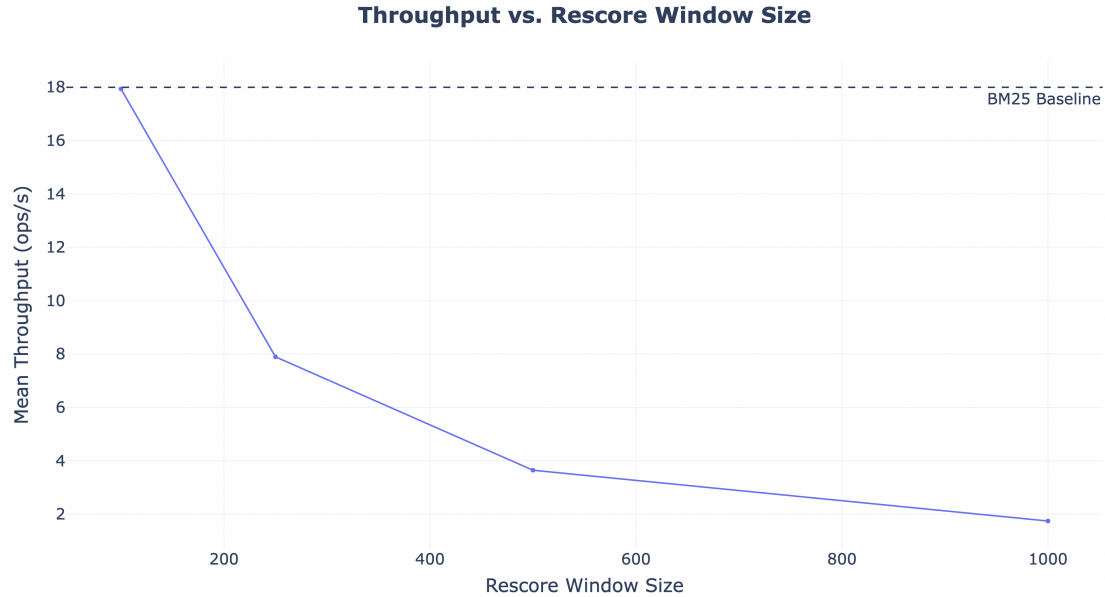


Figure 6.7: Window size (x-axis) vs. mean throughput in queries per second (y-axis), for the first-stage retrieval (baseline), and for re-ranking stage. Measurements are shown across window sizes for 100, 250, 500, and 1000 top-k documents to re-rank.

6.2.2 Query Encoder Processing Times

As described in Section 4.1, the query encoder is implemented as an external service to guarantee flexibility and scalability. For this reason, its performance is measured separately. To do this, all the

requests from a full rally run are profiled by using the built-in profiler from the Auto-Alpha query encoding service (see Section 4.2.2). As a result, the computation time for each query embedding, and the 50th, 90th, and 99.9th percentiles of these times are shown in Figure 6.8. Notably, the median processing time is very low (12.57 ms), compared to the plugin’s total re-ranking service time (Figure 6.6), indicating that the query encoding service does not constitute a significant portion of the total service time.

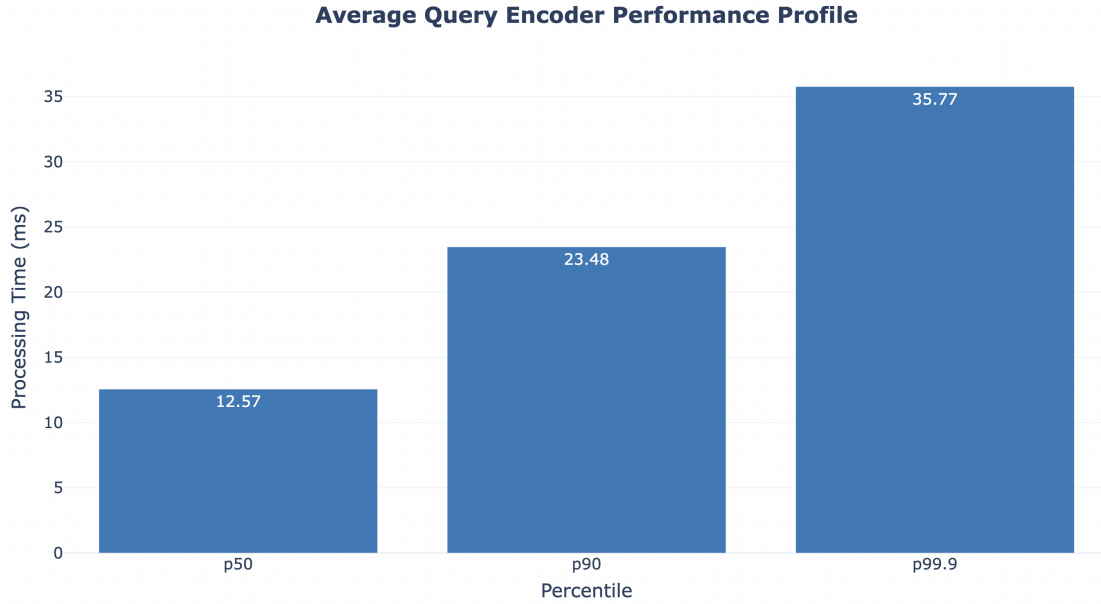


Figure 6.8: 50th, 90th and 99th percentiles for the query-encoding processing time per request.

6.2.3 Discussion: Answering RQ3

In hindsight of the previous results, observations, and overall design of the developed reranking plugin, the research question RQ3: *“Can a native Elasticsearch (or equivalent framework) plugin/tool be developed to implement the Fast-Forward Indexes approach while making its setup simple? Moreover, how can insights from RQs 1 and 2 be applied to this implementation?”*, can be answered.

Yes, this project showed that the Fast-Forward Indexes approach [23] can be implemented as a native Elasticsearch plugin designed for simplicity and ease of use. By building the tool as a native plugin, it seamlessly integrates into an existing Elasticsearch cluster, making the installation and configuration process fast and straightforward. Its architecture validates the practical applicability of the research by making advanced hybrid re-ranking more accessible.

The implementation also addresses how insights from the preceding research questions were applied:

- **Application of RQ1:** The findings on α -optimization are directly implemented with the python-based toolset “Auto-Alpha”. This toolset provides users with data preparation utilities, a flexible query encoding service, and a gradient-based method to automatically find an effective interpolation weight (α) for their specific dataset, directly translating the research on direct alpha optimization into a practical, value-added feature.
- **Application of RQ2:** It is important to clarify that the insights from the adaptor models in RQ2 have not yet been implemented in the current version of the plugin. While the research demonstrated the potential of dynamic, query-aware adaptors, integrating these models into the plugin remains an opportunity for future work.

The plugin is a successful proof-of-concept for a native ES-based Fast-Forward implementation and a direct application of RQ1’s findings. It also clearly defines a path for further enhancements based on the promising results from RQ2.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

This thesis presents two primary contributions to the field of information retrieval. Methodologically, it enhances the Fast-Forward Indexes framework [23] by developing techniques for optimizing the interpolation weight α . This includes an efficient gradient-based optimization method that surpasses grid search and, more significantly, the introduction of lightweight adaptor models. These adaptors dynamically adjust the hybrid weight at query or document granularity, demonstrating improved performance over a static weight without requiring modification to the backbone encoder.

Practically, this work delivered an open-source software package to make the techniques accessible, comprising of a user-friendly Elasticsearch plugin for simple integration of the Fast-Forward Indexes hybrid re-ranking paradigm, and a Python toolset that provides the following main utilities:

- **Automatic α -Optimization:** The toolset includes the proposed gradient-based method to automatically find an optimal interpolation weight, removing the need for grid search when training data is available.
- **Seamless Query Encoding:** A highly customizable and flexible service that provides real-time query embeddings and integrates effortlessly with the re-ranking plugin.
- **Simplified Data Preparation:** Utilities that make the process of preparing training data for optimization fast and straightforward.
- **Flexible Embedding Generation:** For collections without existing embeddings, the toolset can compute them on-demand, while remaining as flexible and extensible as the query encoding service.

Moreover, the experimentation that was carried out enabled answering the three main research questions proposed in this work:

- **RQ 1: How can learning-based methods optimize the interpolation weight α in hybrid re-ranking?** A: In response to RQ1, this research demonstrated that learning-based methods can effectively optimize the interpolation weight α , often surpassing traditional grid search. The investigation resulted in two successful strategies: direct optimization of a single interpolation constant, and the dynamic prediction of an interpolation value for each query/document. The dynamic approach proved superior, highlighting that a context-aware interpolation weight is more effective than a fixed, global one.
- **RQ 2: Can lightweight adaptors improve performance when using pre-computed embeddings from non-domain-specific encoders?** A: In answer to RQ2, this thesis concludes that lightweight adaptors are a valuable tool, particularly for domain adaptation where a pre-trained encoder initially underperforms. The benefit was stronger where adaptors compensated for a domain mismatch. Conversely, on datasets where the encoder already had a strong baseline, they offered no significant advantage. This research also discovered an important insight: strategies aimed at replacing the existing dot-product scoring mechanism were ineffective. This suggests that the relevance signals

from modern encoders are very robust, and that in this hybrid re-ranking setting, the most effective adaptation strategies are those that modulate the existing signals rather than attempting to learn a new one from scratch.

- **RQ 3: Can a native Elasticsearch (or equivalent framework) plugin/tool be developed to implement the Fast-Forward Indexes approach while making its setup simple? How can insights from RQ 1 be applied to this implementation?** A: This research demonstrated that the Fast-Forward Indexes approach can be implemented as a user-friendly, native Elasticsearch plugin, providing an affirmative answer to RQ3. The plugin’s design prioritized simple integration and simplicity, making advanced hybrid re-ranking more accessible. Furthermore, the theoretical findings on alpha-optimization (RQ1) were practically applied through the ”Auto-Alpha” toolset, which automates the tuning of the interpolation weight. While the potential of the adaptor models studied in RQ2 was confirmed, their integration was not included in the current implementation and remains a promising direction for future development.

Limitations

The conclusions drawn from this work should be considered in the context of the following limitations:

- **Plugin Implementation Scope:** The current open-source Elasticsearch plugin properly implements the Fast-Forward Indexes re-ranking framework and the direct optimization of a global α value (addressing RQ1 and part of RQ3). However, it does not yet integrate the dynamic adaptor models, whose potential was demonstrated in the response to RQ2. The integration of these dynamic models is identified as a direction for future work.
- **Performance Evaluation Environment:** The performance benchmarks for the Elasticsearch plugin were conducted in a controlled, single-node environment with a single index shard. While this setup is sufficient for a proof-of-concept evaluation, the observed latency and throughput metrics may not be directly generalizable to larger, multi-node production clusters, which would involve additional complexities like network overhead and shard routing.
- **Constrained Training Data:** The limited performance gains observed on the TREC-DL dataset may be partially attributable to the small size of the training set, which consisted of only 38 queries. This data scarcity likely constrained the ability of the learning-based models to significantly outperform the already strong baselines.
- **Domain Specificity of Adaptors:** The adaptors demonstrated their most significant value in the out-of-domain context of the ANTIQUE dataset, where the backbone encoders initially underperformed. Conversely, on the in-domain TREC-DL dataset, their improvements were not statistically significant, suggesting that their benefit is most pronounced when bridging a domain gap.
- **Ineffectiveness of Score Replacement:** The experiments revealed that adaptor models designed to entirely replace the dot-product dense score with a newly learned function consistently underperformed. This indicates a limitation of the ”from-scratch” learning approach in this context and suggests that modulating existing strong signals (BM25 and dot-product) is a more effective strategy than attempting to replace them.

7.2 Future Work

The contributions of this thesis lay the groundwork for diverse promising research and development directions:

Plugin Enhancements and Production-Readiness

While the current plugin validates the core approach, future work should focus on preparing it, and validating it against more demanding scenarios:

- **Performance Optimization:** The efficiency of internal operations, specifically the dense score computations, could be further optimized to improve throughput and reduce latency.
- **Scalability and Robustness Testing:** The plugin's performance should be rigorously evaluated in complex environments, such as a multi-node, multi-shard Elasticsearch cluster under higher query loads, to ensure its stability and scalability.
- **Native adaptor Implementation:** It would be interesting to explore whether natively implementing one of the dynamic α -prediction models within the plugin is possible, without harming the core mechanisms of Elasticsearch. This would be a challenging task, due to factors such as: the overhead of running inference on JVM-based ES nodes, model format incompatibilities, and the need for a robust Java-based ML library.

Advancements in adaptor Models

The adaptor models themselves can be refined and explored further:

- **Low-Rank NTN Evaluation:** The repository for this work includes a preliminary implementation of a low-rank [15] variant for the NTN-based adaptors, however, a dedicated study could be conducted to formally evaluate the implications for the parameter efficiency, and the impact it would have on the adaptor's performance.
- **Exploration of Other Architectures:** Research could also investigate other parameter-efficient model architectures for the dynamic adaptors to further improve the trade-off between performance and computational cost.

Broader Experimental Validation

To further solidify the finding of this work, the proposed techniques should be tested in new and more challenging contexts:

- **Testing on Specialized Domains:** The effectiveness of the α -optimization and adaptor techniques could be validated on datasets with more specialized language. For instance, the TREC-COVID dataset [38], with its focus on scientific and medical text, would be an excellent test case.
- **Evaluating Domain Adaptation:** Future experiments could continue to focus on scenarios where the encoder backbone has not been pre-trained on the target domain, providing a stronger test of the adaptors' ability to bridge the domain gap.

7.3 Disclosure

According to the publishing policies of TU Delft ¹, I hereby acknowledge that some portions of text from this work have been rephrased using AI tools powered by ChatGPT. GitHub Copilot has also been employed for code assistance during the development of the technical code-related aspects of this work. Finally, the cover image of this work was generated using Google Gemini. Most importantly, all AI generated content has been thoroughly verified to ensure its validity.

¹<https://www.tudelft.nl/library/actuele-themas/open-publishing/about/policies>

Bibliography

- [1] encode/uvicorn: An ASGI web server, for Python. . URL <https://github.com/encode/uvicorn>.
- [2] fastapi/fastapi: FastAPI framework, high performance, easy to learn, fast to code, ready for production. URL <https://github.com/fastapi/fastapi>.
- [3] What is Elasticsearch? — Elasticsearch Guide [8.17] — Elastic. URL <https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro-what-is-es.html>.
- [4] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, Mir Rosenberg, Xia Song, Alina Stoica, Saurabh Tiwary, and Tong Wang. MS MARCO: A Human Generated MACHine Reading COMprehension Dataset. *CEUR Workshop Proceedings*, 1773, 11 2016. ISSN 16130073. URL <https://arxiv.org/abs/1611.09268v3>.
- [5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 2020-December, 5 2020. ISSN 10495258. URL <https://arxiv.org/pdf/2005.14165>.
- [6] Cen Chen, Xinjing Huang, Feng Ji, Chengyu Wang, Minghui Qiu, Jun Huang, and Yin Zhang. CAT-BERT: A Context-Aware Transferable BERT Model for Multi-turn Machine Reading Comprehension. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12682 LNCS:152–167, 2021. ISSN 16113349. doi: 10.1007/978-3-030-73197-7{_}10/FIGURES/3. URL https://link.springer.com/chapter/10.1007/978-3-030-73197-7_10.
- [7] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, Daniel Campos, and Ellen M. Voorhees. Overview of the TREC 2019 deep learning track. 3 2020. URL <https://arxiv.org/abs/2003.07820v2>.
- [8] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, and Daniel Campos. Overview of the TREC 2020 deep learning track. *29th Text REtrieval Conference, TREC 2020 - Proceedings*, 2 2021. URL <https://arxiv.org/pdf/2102.07662>.

- [9] Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 1:4171–4186, 10 2018. URL <https://arxiv.org/pdf/1810.04805>.
- [10] Elastic. elastic/rally: Macrobenchmarking framework for Elasticsearch. URL <https://github.com/elastic/rally>.
- [11] Aditya Grover, Eric Wang, Aaron Zweig, and Stefano Ermon. Stochastic Optimization of Sorting Networks via Continuous Relaxations. *7th International Conference on Learning Representations, ICLR 2019*, 3 2019. URL <https://arxiv.org/abs/1903.08850v2>.
- [12] Helia Hashemi, Mohammad Aliannejadi, Hamed Zamani, and W Bruce Croft. ANTIQUE: A Non-Factoid Question Answering Benchmark. URL <https://ciir.cs.umass.edu/downloads/Antique/>.
- [13] Sebastian Hofstätter, Sheng Chieh Lin, Jheng Hong Yang, Jimmy Lin, and Allan Hanbury. Efficiently Teaching an Effective Dense Retriever with Balanced Topic Aware Sampling. *SIGIR 2021 - Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 113–122, 4 2021. doi: 10.1145/3404835.3462891. URL <https://arxiv.org/pdf/2104.06967>.
- [14] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzȳbski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-Efficient Transfer Learning for NLP. *36th International Conference on Machine Learning, ICML 2019*, 2019-June:4944–4953, 2 2019. URL <https://arxiv.org/pdf/1902.00751>.
- [15] Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. *ICLR 2022 - 10th International Conference on Learning Representations*, 6 2021. URL <https://arxiv.org/pdf/2106.09685>.
- [16] Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised Dense Information Retrieval with Contrastive Learning. URL <https://github.com/facebookresearch/contriever>.
- [17] Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised Dense Information Retrieval with Contrastive Learning. *Transactions on Machine Learning Research*, 2022-August, 12 2021. ISSN 28358856. URL <https://arxiv.org/pdf/2112.09118>.
- [18] Xin Jin and Jiawei Han. K-Means Clustering. *Encyclopedia of Machine Learning*, pages 563–564, 2011. doi: 10.1007/978-0-387-30164-8{_}425. URL https://link.springer.com/rwe/10.1007/978-0-387-30164-8_425.
- [19] Cheongwoong Kang and Jaesik Choi. Impact of Co-occurrence on Factual Knowledge of Large Language Models. *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 7721–7735, 10 2023. doi: 10.18653/v1/2023.findings-emnlp.518. URL <https://arxiv.org/pdf/2310.08256>.
- [20] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen Tau Yih. Dense Passage Retrieval for Open-Domain Question Answering. *EMNLP 2020 - 2020 Conference on Empirical Methods in Natural Language Processing, Proceedings*

- of the Conference, pages 6769–6781, 4 2020. doi: 10.18653/v1/2020.emnlp-main.550. URL <https://arxiv.org/pdf/2004.04906>.
- [21] Omar Khattab and Matei Zaharia. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. *SIGIR 2020 - Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 39–48, 4 2020. doi: 10.1145/3397271.3401075. URL <https://arxiv.org/pdf/2004.12832>.
- [22] Justin Lee, Gabriel Bernier-Colborne, Tegan Maharaj, and Sowmya Vajjala. Methods, Applications, and Directions of Learning-to-Rank in NLP Research. *Findings of the Association for Computational Linguistics: NAACL 2024 - Findings*, pages 1900–1917, 2024. doi: 10.18653/V1/2024.FINDINGS-NAACL.123. URL <https://aclanthology.org/2024.findings-naacl.123/>.
- [23] Jurek Leonhardt, Henrik Müller, Koustav Rudra, Megha Khosla, Abhijit Anand, and Avishek Anand. Efficient Neural Ranking using Forward Indexes and Lightweight Encoders. *ACM Transactions on Information Systems*, 42(5), 11 2023. ISSN 15582868. doi: 10.1145/3631939. URL <https://arxiv.org/abs/2311.01263v1>.
- [24] Bohan Li, Hao Zhou, Junxian He, Mingxuan Wang, Yiming Yang, and Lei Li. On the Sentence Embeddings from Pre-trained Language Models. *EMNLP 2020 - 2020 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pages 9119–9130, 11 2020. doi: 10.18653/v1/2020.emnlp-main.733. URL <https://arxiv.org/pdf/2011.05864>.
- [25] Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, Rodrigo Nogueira, and David R Cheriton. Pyserini: An Easy-to-Use Python Toolkit to Support Replicable IR Research with Sparse and Dense Representations. 2 2021. URL <https://arxiv.org/pdf/2102.10073>.
- [26] Sheng-Chieh Lin, Jheng-Hong Yang, Jimmy Lin, and David R Cheriton. Distilling Dense Representations for Ranking using Tightly-Coupled Teachers.
- [27] Sheng-Chieh Lin, Jheng-Hong Yang, Jimmy Lin, and David R Cheriton. Distilling Dense Representations for Ranking using Tightly-Coupled Teachers. 10 2020. URL <https://arxiv.org/pdf/2010.11386>.
- [28] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, Veselin Stoyanov, and Paul G Allen. RoBERTa: A Robustly Optimized BERT Pretraining Approach. 7 2019. URL <https://arxiv.org/pdf/1907.11692>.
- [29] Sean MacAvaney, Craig Macdonald, and Iadh Ounis. Streamlining Evaluation with ir-measures. 11 2021. URL <https://arxiv.org/pdf/2111.13466>.
- [30] Sean Macavaney, Andrew Yates, Sergey Feldman, Doug Downey, Arman Cohan, Nazli Goharian, and Arman Co-Han. Simplified Data Wrangling with ir_datasets; Simplified Data Wrangling with ir_datasets. 2021. doi: 10.1145/3404835.3463254. URL <https://doi.org/10.1145/3404835.3463254>.
- [31] Bhaskar Mitra and Nick Craswell. An introduction to neural information retrieval. *Foundations and Trends in Information Retrieval*, 13(1):1–129, 12 2018. ISSN 15540677. doi: 10.1561/15000000061.
- [32] Jonas Mueller and Aditya Thyagarajan. Siamese Recurrent Architectures for Learning Sentence Similarity. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1):2786–2792, 3 2016. ISSN 2374-3468. doi: 10.1609/AAAI.V30I1.10350. URL <https://ojs.aaai.org/index.php/AAAI/article/view/10350>.

- [33] 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 Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 32, 12 2019. ISSN 10495258. URL <https://arxiv.org/pdf/1912.01703>.
- [34] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *NAACL HLT 2018 - 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 1:2227–2237, 2 2018. doi: 10.18653/v1/n18-1202. URL <https://arxiv.org/pdf/1802.05365>.
- [35] Przemysław Pobrotyn and Radosław Białobrzeski. NeuralNDCG: Direct Optimisation of a Ranking Metric via Differentiable Relaxation of Sorting. 2 2021. URL <https://arxiv.org/abs/2102.07831v2>.
- [36] Przemysław Pobrotyn and Radosław Białobrzeski. NeuralNDCG: Direct Optimisation of a Ranking Metric via Differentiable Relaxation of Sorting. 2 2021. URL <https://arxiv.org/abs/2102.07831v2>.
- [37] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP-IJCNLP 2019 - 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Proceedings of the Conference*, pages 3982–3992, 8 2019. doi: 10.18653/v1/d19-1410. URL <https://arxiv.org/pdf/1908.10084>.
- [38] Kirk Roberts, Tasmeer Alam, Steven Bedrick, Dina Demner-Fushman, Kyle Lo, Ian Soboroff, Ellen Voorhees, Lucy Lu Wang, and William R. Hersh. Searching for scientific evidence in a pandemic: An overview of TREC-COVID. *Journal of Biomedical Informatics*, 121, 9 2021. ISSN 15320464. doi: 10.1016/j.jbi.2021.103865. URL <https://arxiv.org/pdf/2104.09632>.
- [39] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 4 2009. ISSN 15540669. doi: 10.1561/15000000019;WGROU:STRING:ACM. URL <https://dl.acm.org/doi/10.1561/15000000019>.
- [40] Richard Sinkhorn. A Relationship Between Arbitrary Positive Matrices and Doubly Stochastic Matrices. <https://doi.org/10.1214/aoms/1177703591>, 35(2):876–879, 6 1964. ISSN 0003-4851. doi: 10.1214/AOMS/1177703591. URL <https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-35/issue-2/A-Relationship-Between-Arbitrary-Positive-Matrices-and-Doubly-Stochastic-Matrices/10.1214/aoms/1177703591.full><https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-35/issue-2/A-Relationship-Between-Arbitrary-Positive-Matrices-and-Doubly-Stochastic-Matrices/10.1214/aoms/1177703591.short>.
- [41] Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in Neural Information Processing Systems*, 2013.
- [42] Sture Holm. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979. ISSN 03036898.

-
- [43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *Advances in Neural Information Processing Systems*, 2017-December:5999–6009, 6 2017. ISSN 10495258. URL <https://arxiv.org/abs/1706.03762v7>.
- [44] Jiang Wang, Yang Song, Thomas Leung, Chuck Rosenberg, Jingbin Wang, James Philbin, Bo Chen, and Ying Wu. Learning fine-grained image similarity with deep ranking. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2014. doi: 10.1109/CVPR.2014.180.
- [45] 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. Transformers: State-of-the-Art Natural Language Processing. URL <https://github.com/huggingface/>.
- [46] Lingling Xu, Haoran Xie, Si-Zhao Joe Qin, Xiaohui Tao, and Fu Lee Wang. Parameter-Efficient Fine-Tuning Methods for Pretrained Language Models: A Critical Review and Assessment. 12 2023. URL <https://arxiv.org/pdf/2312.12148>.
- [47] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the Machine Learning Lifecycle with MLflow. 2018.
- [48] Wayne Xin Zhao, Jing Liu, Ruiyang Ren, and Ji Rong Wen. Dense Text Retrieval based on Pretrained Language Models: A Survey. *ACM Transactions on Information Systems*, 42(4), 11 2022. ISSN 15582868. doi: 10.1145/3637870. URL <https://arxiv.org/pdf/2211.14876>.

Appendix A

Supporting Contents and Analysis

A.1 Full Relative Scores

This section provides the full set of relative performances of the direct alpha optimization techniques, and the adaptor models. The results are presented in a heatmap in figure A.1.

General Observations

Overall the dynamic alpha adaptor models showed the most relative improvements, followed by the direct alpha optimization technique. Moreover, the least performing models were the standalone NTN and FFNN models, followed by the direct alpha optimization method with the MRL objective.

The fact that ANTIQUE shows more improvements which are also significant, in comparison to TREC-DL-2020 PSG', reinforces the argument that adaptors can be effective to "adapt" the interpolation parameters and improve the performance of interpolation-based re-ranking techniques on out-of-domain datasets.

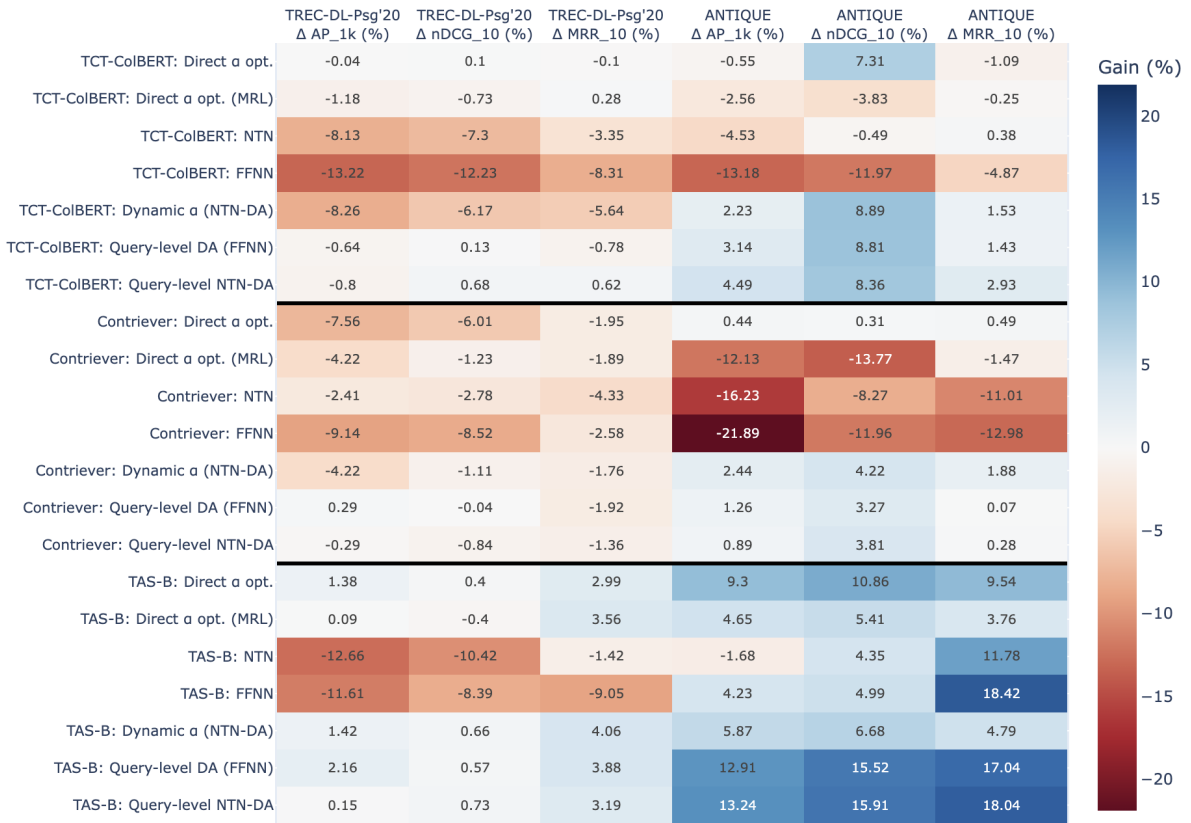


Figure A.1: Relative gain/decrease (%) of each metric, for all methods over the Hyperparameter Search baseline for each encoder and dataset. Gains are computed as $\frac{\text{Best} - \text{Hyperparam}}{\text{Hyperparam}} \times 100$.

A.2 Plugin Experimentation

A.2.1 Indexing Mapping

During indexing of the MSMARCO passage collection, the mapping presented below was used:

```
"mappings": {
  "properties": {
    "passage": {
      "type": "text",
      "analyzer": "standard"
    },
    "vector_field": {
      "type": "float",
      "store": True,
      "index": False,
      "doc_values": False
    }
  }
},
"settings": {
  "number_of_shards": 1,
```

```

    "number_of_replicas": 0,
    "refresh_interval": "30s" # Optimize for bulk indexing
}

```

A.2.2 Rally Full Track Definition

The Rally track definition JSON that was used for the test setup of the ES' plugin performance is provided below:

```

{
  "version": 2,
  "description": "Interleaved BM25 vs rerank_ffwi rescore (latency A/B, cache-fair)",
  "operations": [
    {
      "name": "bm25-search",
      "operation-type": "search",
      "param-source": "ffwi-queries",
      "params": {
        "file": "queries.txt",
        "mode": "bm25",
        "index": "{{ index | default('msmarco_v1') }}",
        "field": "{{ field | default('passage') }}",
        "size": "{{ size | default(10) }}",
        "source_fields": [
          "passage"
        ],
        "profile": false,
        "preference": "{{ preference | default('ab-bench') }}"
      },
      "request-params": {
        "request_cache": "false",
        "preference": "{{ preference | default('ab-bench') }}"
      }
    },
    {
      "name": "rescore-search",
      "operation-type": "search",
      "param-source": "ffwi-queries",
      "params": {
        "file": "queries.txt",
        "mode": "rescore",
        "index": "{{ index | default('msmarco_v1') }}",
        "field": "{{ field | default('passage') }}",
        "size": "{{ size | default(10) }}",
        "window_size": "{{ window_size | default(100) }}",
        "alpha": "{{ alpha | default(0.123) }}",
        "vector_field": "{{ vector_field | default('vector_field') }}",
        "batch_size": "{{ batch_size | default(64) }}",
        "source_fields": [
          "passage"
        ]
      }
    }
  ]
}

```

```
    ],
    "profile": false,
    "preference": "{{ preference | default('ab-bench') }}"
  },
  "request-params": {
    "request_cache": "false",
    "preference": "{{ preference | default('ab-bench') }}"
  }
}
],
"challenges": [
  {
    "default": true,
    "name": "interleaved-a_b_intense",
    "schedule": [
      {
        "name": "bm25-warm",
        "operation": "bm25-search",
        "clients": 4,
        "warmup-time-period": 180,
        "time-period": 1,
        "target-throughput": 18
      },
      {
        "name": "rescore-warm",
        "operation": "rescore-search",
        "clients": 4,
        "warmup-time-period": 180,
        "time-period": 1,
        "target-throughput": 18
      },
      {
        "name": "bm25-1",
        "operation": "bm25-search",
        "clients": 4,
        "time-period": 240,
        "target-throughput": 18
      },
      {
        "name": "rescore-1",
        "operation": "rescore-search",
        "clients": 4,
        "time-period": 240,
        "target-throughput": 18
      },
      {
        "name": "bm25-2",
        "operation": "bm25-search",
        "clients": 4,
        "time-period": 240,
```

```

    "target-throughput": 18
  },
  {
    "name": "rescore-2",
    "operation": "rescore-search",
    "clients": 4,
    "time-period": 240,
    "target-throughput": 18
  },
  {
    "name": "bm25-3",
    "operation": "bm25-search",
    "clients": 4,
    "time-period": 240,
    "target-throughput": 18
  },
  {
    "name": "rescore-3",
    "operation": "rescore-search",
    "clients": 4,
    "time-period": 240,
    "target-throughput": 18
  },
  {
    "name": "bm25-4",
    "operation": "bm25-search",
    "clients": 4,
    "time-period": 240,
    "target-throughput": 18
  },
  {
    "name": "rescore-4",
    "operation": "rescore-search",
    "clients": 4,
    "time-period": 240,
    "target-throughput": 18
  }
]
},
{
  "name": "interleaved-a_b",
  "schedule": [
    {
      "name": "bm25-1",
      "operation": "bm25-search",
      "clients": 4,
      "warmup-iterations": 200,
      "iterations": 20000,
      "target-throughput": 18
    }
  ],

```

```
{
  "name": "rescore-1",
  "operation": "rescore-search",
  "clients": 4,
  "warmup-iterations": 200,
  "iterations": 20000,
  "target-throughput": 18
}
],
{
  "name": "bm25-capacity-find",
  "schedule": [
    {
      "warmup-time-period": 30,
      "name": "bm25-capacity-find-run-1",
      "operation": "bm25-search",
      "time-period": 60,
      "target-throughput": 20
    },
    {
      "warmup-time-period": 30,
      "name": "bm25-capacity-find-run-2",
      "operation": "bm25-search",
      "time-period": 60,
      "target-throughput": 40
    },
    {
      "warmup-time-period": 30,
      "name": "bm25-capacity-find-run-3",
      "operation": "bm25-search",
      "time-period": 60,
      "target-throughput": 80
    },
    {
      "warmup-time-period": 30,
      "name": "bm25-capacity-find-run-4",
      "operation": "bm25-search",
      "time-period": 60,
      "target-throughput": 120
    }
  ]
},
{
  "name": "rescore-capacity-find",
  "schedule": [
    {
      "name": "rescore-search-run-1",
      "operation": "rescore-search",
      "clients": 4,
```

```

        "warmup-time-period": 30,
        "time-period": 60,
        "target-throughput": 20
    }
]
},
{
    "name": "profile-sample",
    "schedule": [
        {
            "name": "bm25-profile",
            "operation": "bm25-search",
            "warmup-iterations": 5,
            "iterations": 50,
            "target-throughput": 5,
            "params": {
                "mode": "bm25",
                "profile": true,
                "file": "queries.txt"
            }
        },
        {
            "name": "rescore-profile",
            "operation": "rescore-search",
            "warmup-iterations": 5,
            "iterations": 50,
            "target-throughput": 5,
            "params": {
                "mode": "rescore",
                "file": "queries.txt",
                "index": "{{ index | default('msmarco_v1') }}",
                "field": "{{ field | default('passage') }}",
                "size": "{{ size | default(100) }}",
                "window_size": "{{ window_size | default(100) }}",
                "alpha": "{{ alpha | default(0.123) }}",
                "vector_field": "{{ vector_field | default('vector_field') }}",
                "batch_size": "{{ batch_size | default(64) }}",
                "profile": false,
                "preference": "{{ preference | default('ab-bench') }}"
            }
        }
    ]
}
]
}

```