Extremely efficient online query encoding for dense retrieval

Nachshon Cohen
Amazon
nachshon@amazon.com

Yaron Fairstein
Amazon
yaronf@amazon.com

Guy Kushilevitz
Amazon
guyk@amazon.com

Abstract

Existing dense retrieval systems utilize the same model architecture for encoding both the passages and the queries, even though queries are much shorter and simpler than passages. This leads to high latency of the query encoding, which is performed online and therefore might impact user experience. We show that combining a standard large passage encoder with a small efficient query encoder can provide significant latency drops with only a small decrease in quality. We offer a pretraining and training solution for multiple small query encoder architectures. Using a small transformer architecture we are able to decrease latency by up to $\sim 12\times$, while $MRR@10$ on the MS MARCO dev set only decreases from 38.2 to 36.2. If this solution does not reach the desired latency requirements, we propose an efficient RNN as the query encoder, which processes the query prefix incrementally and only infers the last word after the query is issued. This shortens latency by $\sim 38\times$ with only a minor drop in quality, reaching 35.5 $MRR@10$ score.[1]

1 Introduction

Information retrieval was revolutionized by semantic matching models (Karpukhin et al., 2020; Xiong et al., 2021; Gao and Callan, 2021, 2022). Such models encode the corpus of passages[2] and the query in a shared embedding space, where retrieval is performed using an (approximated) nearest neighbors search (Johnson et al., 2021). These models increase the quality of search results dramatically (Zhao et al., 2022), but suffer from a large computational overhead (Chen et al., 2021). While training a large model and encoding the corpus is costly, this can usually be done offline once (or every couple of days/weeks) and cost is bounded by the size of the corpus. On the other hand, encoding queries is a major part of the retrieval system that is performed frequently and online, making latency an important consideration.[3] Hence, cutting the latency of this component directly leads to a cut in the online-latency of the whole system.[4]

Today, practically all semantic retrieval models use the same architecture to embed both the corpus (passages) and the queries. Knowledge distillation (Hinton et al., 2015) has been used to improve efficiency by creating smaller models. However, mainly transformer-based architectures (Vaswani et al., 2017) of medium size were considered (Gao et al., 2020; Chen et al., 2021), putting a bound on the achievable latency of the query encoding.

Balancing between the latency and cost requirements is challenging; while sophisticated GPU implementations can run BERT inference in just a few milliseconds, this hardware is very costly. This is especially problematic in an over-provisioning setting, where utilization is kept low to handle burst of traffic. Further, as query encoding is run online, it is often necessary to use a batch size of 1, which also limits the GPU utilization. Therefore, it is often necessary to use a CPU for query encoding, which in-turn increases the latency overhead. This challenge calls for a query-embedding solution that can balance cost and latency, while still providing quality embeddings for retrieval.

The simple, yet crucial observation we make in this paper is that queries are usually very short; often just 3-5 words, and rarely exceeding 15 words. This is in contrast to passages (or documents),

[1] Code can be found at https://github.com/amzn/extremely-efficient-query-encoder

[2] In this paper we consider the passage retrieval task. Retrieving documents or other textual units is similar in concept.

[3] The other significant part typically run online is an approximate KNN-search. We experiment with ScaNN (Guo et al., 2020), a popular KNN solution. We use it with standard parameters to retrieve from MS-MARCO's corpus and find that query embedding takes $\sim \times 4$ more time than the KNN search. Hence, we can determine that the significant portion of online latency is spent on embedding the query.

[4] Additional avenues for reducing latency are presented in (Seo et al., 2019; Fang et al., 2020; Lewis et al., 2021; Formal et al., 2021).

Findings of the Association for Computational Linguistics: NAACL 2024, pages 43–50
June 16-21, 2024 ©2024 Association for Computational Linguistics

which consist of dozens of words or more in some settings. Therefore, we argue that while large, complex models and a vast amount of training data are crucial for quality passage embedding[5], for query embedding it is sufficient to use smaller, simpler models. With this observation, we propose a method to trade-off latency and quality of online query encoding for dense retrieval, reaching low latency while preserving reasonable quality.

Specifically, we propose training two different variations of small models for query encoding. The first straightforward option is a small, efficient transformer. This leads to impressive results, barely hurting the retrieval quality, but the decrease in online-latency is limited to $12\times$, reaching 2.1 milliseconds. To extend our solution and also deal with cases where a more significant decrease in online latency is needed, we propose using an RNN-based model. As mentioned above, queries tend to be short, making RNNs a viable option.

Apart from being efficient, the RNN architecture offers another benefit for online latency. Since an RNN processes tokens sequentially, the system can feed the model with the prefix of the query as it is typed by the user. When the user issues the query, the model only needs to process its last word. This method is denoted as *incremental inference* in this paper, and is able to further reduce online-latency. Our smallest proposed model reaches a $38\times$ drop in latency compared to the baseline, with an online latency of only 0.7 milliseconds running on a CPU, while also achieving competitive quality results. Finally, for cases where online-latency is of utmost importance, we suggest a method to practically nullify the contribution of query-encoding to the online-latency at the cost of $\sim 4\times$ rise in compute.

2 Design

We want to train a dual encoder system composed of a small and efficient query encoder and a standard larger transformer passage encoder. We are not interested in the training procedure of the large encoder, which was already studied thoroughly. Therefore, we assume one is available.

We denote by T_{enc} the large Transformer encoder, and S_q as the small query encoder (either a smaller transformer, or an RNN). Even though the passage and query encoders cannot share all their weights due to their different sizes and architectures, we opt to keep the token embeddings of both models tied. This ensures that a token has the same “meaning” in both models (Dong et al., 2022). This decision is further discussed in Appendix A. To train the efficient query encoder we operate in stages, as detailed in this section.

2.1 Pretraining via Distillation

A large encoder T_{enc}, trained for passage and query encoding, is available. Thus, we use it as a teacher to the smaller S_q. We train S_q to imitate the embeddings T_{enc} generates for all queries in the train set. We use a standard cosine similarity loss, pushing the embeddings generated by S_q towards the embeddings generated by T_{enc}. We pretrain for 10 epochs, as discussed in Section 4.4.1.

2.2 Training on Labeled Data

The large passage encoder T_{enc} and the small query encoder, starting from the pretrained S_q, are trained for dense retrieval. We use the standard training procedure of (Gao and Callan, 2022), including the selection of negative samples and other hyperparameters. Further details appear in Appendix B.

2.3 Small Model Architectures

RNN. In this work we use a GRU (Cho et al., 2014) as the architecture of S_q. In order to increase the capability of the network, we consider models with different capacities by stacking multiple recurrent networks together and adding a feed-forward (FF) layer on top of the embedding generated for the last token of the query. This, of course, comes with a latency cost. The FF network is defined as:

$$FF(x) = LayerNorm(x + W_2(Gelu(W_1 \cdot x + b_1) + b_2)$$

Small Transformer (ST). When using a transformer based model to implement S_q, we use a BERT-like architecture with different number of layers. We initialize the model from the first layers of the pretrained encoder T_{enc}.

3 Incremental Inference with RNNs

When using an RNN to encode a token-sequence, the encoding of the prefix of tokens is independent of the rest of the tokens:

$$\mathrm{RNN}(\mathrm{pref} + \mathrm{suff}) = \mathrm{RNN}(\mathrm{RNN}(\mathrm{pref}), \mathrm{suff})$$

This property enables incremental encoding of user queries before they are fully composed. Upon

[5] Gao et al. (2020) show that in order to properly distil an encoder for retrieval a vast amount of data is needed.

query completion, encoding only the remaining part accelerates encoding and minimizes latency.

However, while the model can encode tokens incrementally, the tokenization process is not independent of the prefix. For example, while `hell` is a prefix of `hello`, their token representations are not. Luckily, word boundaries (e.g., a space) are not crossed by the tokenizer, so prefix encoding can immediately be applied when coming across such a boundary. When the user issues the query, only the last word has to be encoded.

A single word can span multiple tokens (e.g., 'cephalosporin' consists of 5 tokens), necessitating multiple inference steps. Still, in the MS MARCO dev set queries, the last word's token count percentiles (p50, p90, p95, and p99) are 1, 2, 3, and 4 (respectively). In complete queries the same percentiles correspond to 9, 12, 14, and 18 tokens. This suggests that by only processing the last word of a query on the critical path of inference, we can significantly reduce the latency. Note, the current Guinness record for fast typing is 212 words per minute[6], or 283ms per average word. As our computation speed per word is significantly smaller, computations of the query prefix are done before the last word is issued. While this approach could increase overall computation time[7], in most cases the latency of the critical path is more important than overall latency. Section 4.1 shows that incremental encoding can vastly reduce this measure.

3.1 Extreme Incremental Encoding

There are cases where reducing latency is drastically more important than computation cost. Assuming that the user has to hit the Enter key to initiate the search, we show that by encoding each intermediate string, each requiring a single RNN computation step, the query encoding can be computed before the user hits Enter, practically translating to an online latency of 0.

We start by stating a property of tokenizers:

Property 1 *For every string S and non-space character c, the tokenization of $S + c$ consists of a sequence of tokens T such that $T[: -1]$ corresponds to the tokenization of a prefix of S.*

Thus, adding a single character to a string corresponds to adding just a single token to a prefix of the computed tokens. Assuming we store the embeddings of all token prefixes, the embeddings of the tokens of the new string can be computed with a single RNN step. We note that the encodings generated by this process are equivalent to those generated by the vanilla RNN approach. Therefore, result presented in Section 4.1 for the RNN models are reached by this method as well, with an online latency of 0. However, this incurs a computational cost, as we encode every possible prefix of the string. The number of steps grows by a factor equal to the number of non-space characters divided by the number of tokens in the query. In MS MARCO, this is equal to $\sim 4\times$ compared to the vanilla RNN approach.

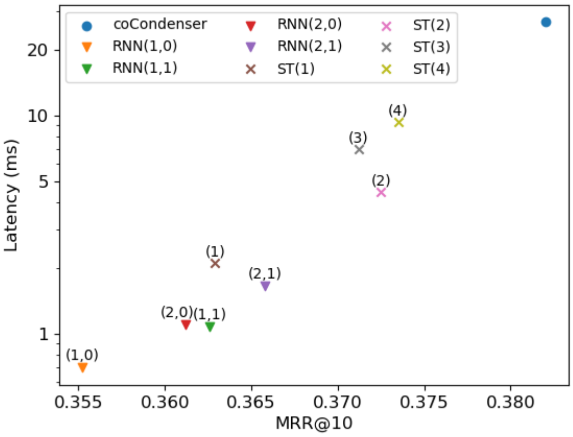

Figure 1: Illustrating the quality (MRR@10) - latency (in milli-seconds) tradeoff. **x** represents the small transformers architecture, $\triangledown$ represents the RNN architecture.

4 Experiments

Our models are based on the Tevatron framework (Gao et al., 2022) and therefore coCondenser is the main baseline we compare to. For completeness we also include the results of BM25, DPR (Karpukhin et al., 2020) and ANCE (Xiong et al., 2021). We follow many previous works and train and test our methods on the MS MARCO dataset (Nguyen et al., 2016) using $MRR@10$ as the main metric and $R@50/1000$ as complementary metrics, and on the NQ dataset (Kwiatkowski et al., 2019) using $R@5/20/100$ as metrics. For T_{enc} we use the pretrained version of coCondenser, trained on the MLM task in a retrieval-friendly way. S_q is implemented both using an RNN model and a ST model as described in Section 2. We denote by RNN(ℓ, f) an RNN model with ℓ layers and f feed-forward layers. ST(ℓ) is an ST model with ℓ layers.

[6] https://www.academyoflearning.com/blog/the-fastest-typists-in-the-world-past-and-present

[7] The overhead of invoking PyTorch is non-negligible.

Query encoder	p95	Params	MS-MARCO			Natural Question		
			MRR@10	R@50	R@1k	R@5	R@20	R@100
BM25	-	-	18.7	-	85.7	-	59.1	73.7
DPR	26.81	110	-	-	-	-	74.4	85.3
ANCE	26.81	110	33.0	-	95.9	-	81.9	87.5
coCondenser	26.81	110	38.2	86.5	98.4	75.8	84.3	89
RNN(1,0)	0.70	27.4	35.5	82.6	97.0	67.45	80.36	87.45
RNN(1,1)	1.07	32.1	36.2	83.8	97.8	67.64	81.13	88.11
RNN(2,0)	1.10	30.9	36.1	84.1	97.8	68.39	80.41	87.72
RNN(2,1)	1.65	35.6	36.5	84.6	97.9	68.61	81.24	88.25
ST(1)	2.1	31.5	36.2	83.7	97.7	68.5	81.19	88.03
ST(2)	4.44	38.6	37.2	85.6	98.3	69.88	82.13	88.69
ST(3)	6.99	45.7	37.1	86.2	98.3	70.94	82.32	88.64
ST(4)	9.31	52.8	37.3	86.5	98.4	71.82	83.15	88.86

Table 1: Online latency vs quality of different query encoder models. Number of parameters is reported in millions. Online latency is measured in milliseconds and the p95 percentile is reported.

4.1 Main Results

Main results are provided in Table 1.[8] Using a small query encoder can indeed be very rewarding. For example, on the MS-MARCO dataset reducing the query-encoder from the standard 12-layer transformer to a 2-layer transformer drops latency by $\sim 6\times$ for only a modest drop in the $MRR@10$ score (from 38.2 to 37.2) and barely any change in the $Recall@1000$ measure. On a different note, in Figure 1 it can be seen that the RNN methods are highly effective in extending the latency/quality trade-off curve. While the smallest transformer can reduce $\sim 12\times$ in latency compared to the baseline with a drop from 38.2 to 36.2 in $MRR@10$ score, the smallest RNN model extends the drop in latency to $\sim 38\times$ reaching 35.5 $MRR@10$ score. A similar trend can be seen in the results for NQ, with a slight difference in behavior at the top and bottom of the lists metrics. We further elaborate on this topic in Section 4.2.

4.2 Fine-grained Topical Understanding

Table 2 compares the fine-grained topical understanding of our smallest architecture, RNN(1,0), with that of coCondenser. As expected, the smaller models are less capable in capturing more complex nuances, affecting its R@k scores for small k-s. Yet, it is interesting to note that its performance is almost on-par with that of coCondenser for large k-s, showing impressive coarse-grained understanding. Another observation is that the performance of the small models follow a similar trend on both datasets, with some advantage in MS-MARCO at small k-s and a slight advantage in NQ at large k-s.

R@k	MS-MARCO	NQ
1	92%	88%
5	93%	89%
10	94%	93%
20	95%	95%
50	96%	96%
100	96%	98%

Table 2: Performance of RNN(1,0) measured in percentage w.r.t. the performance of coCondenser.

4.3 RNNs Dependence on Query Length

A concern one might have regarding using RNN models as query encoders, due to the recursive inference process of RNNs, is that the encoding quality will drop significantly for longer queries.

To measure whether quality drops (more than the baseline) when the query becomes longer, we computed the quality drop for each query by subtracting the $MRR@10$ score of an RNN model from the score of the coCondenser baseline, computed on the MS-MARCO dataset. We then compute Pearson correlation between the score drop and the query length. We found that the correlation is only 0.018 and 0.045 for the RNN(1,0) and RNN(2,0) models respectively. These results suggest that the RNN architecture is capable of computing quality embeddings even for the longer queries in the dataset.

[8] For brevity, we report only the 95^{th} percentile as the latency measure in this table. Extended latency results and measurements can be found in Appendix C, where it can be seen that trends are kept across all percentiles.

Epochs	MRR@10	
	ST(2)	RNN(2,1)
0	34.1	31.5
5	37.2	36.5
10	37.3	36.6
15	37.3	36.7

Table 3: Pretraining effect.

model type	passage embeder	MRR@10
RNN	pretrained_coco	0.362
RNN	fine-tuned_coco	0.353
transformer	pretrained_coco	0.372
transformer	fine-tuned_coco	0.362

Table 4: Starting the retrieval training from a trained/pretrained passage encoder model.

4.4 Ablation Study

We study some of the design decisions made when training the models. Specifically, we discuss the pretraining procedure and the teacher model used.

4.4.1 Pretraining

Table 3 shows that the pretraining procedure described in Section 2.1 improves the MRR@10 scores. For the RNN-based models, pretraining is especially important. This makes sense as pretrained RNN weights are not available for initialization, as opposed to the transformer which is initialized from the first layers of a pretrained model.

4.4.2 Teacher Selection

Training S_q relies on a teacher model. The main results uses a pre-trained version of coCondenser as the teacher T_{enc}, which utilizes a self-supervised MLM training. An alternative option would be to utilize the fine-tuned coCondenser model, trained on ground truth labels. On one hand, starting from a well trained model may result in converging to a better model, but on the other hand, it might result in overfitting the training data. We report results both on the RNN architecture (with 2 RNN layers and without feed-forward layers) and the transformer architecture (2 layers) in Table 4. It can be seen that the encoders benefit from learning the retrieval task simultaneously, as opposed to starting the training from a well-trained passage encoder and an untrained query encoder.

5 Conclusions

In this paper, we point out that queries are significantly shorter and simpler than passages, suggesting that using similar architectures for both passage and query encoders might be wasteful. Indeed, we show that small transformer-based query encoders improve latency with only a minor hurt to quality. We also introduce incremental inference with RNN-based encoders, and show they produce an even lower latency, better suited for cases where latency is highly constrained. Again, we show this improvement in latency comes with only a small drop in the quality of the generated embeddings.

6 Limitations

While incremental inference with RNNs drops latency significantly when running on CPUs this is not the case when using GPUs. The overhead of calling the GPU is high compared to the embedding time; in addition, GPUs are not well optimized for the RNN architecture. This means that the benefit of the proposed method is limited. CPUs are often used for retrieval as discussed in Section 1, but there are cases where GPUs are used in which RNN-based architectures are expected to give a lesser gain.

Another limitation of our method is that it requires running two training procedures. First, training a large encoder, and only after it is trained we can start the pretraining and training procedure of the smaller query encoder. Furthermore, since training the query encoder involves inferencing passages (using a larger passage encoder) the training time of a small model is very similar ($\sim$ 10 hours) to the training time of the large transformer. Nevertheless, since online query encoding can run a vast amount of time and the query encoder is trained once, in most cases we believe this is a price worth paying.

References

Xuanang Chen, Ben He, Kai Hui, Le Sun, and Yingfei Sun. 2021. Simplified tinybert: Knowledge distillation for document retrieval. In *Advances in Information Retrieval - 43rd European Conference on IR Research, ECIR 2021, Virtual Event, March 28 - April 1, 2021, Proceedings, Part II*, volume 12657 of *Lecture Notes in Computer Science*, pages 241–248. Springer.

Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. In *Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014*, pages 103–111. Association for Computational Linguistics.

Zhe Dong, Jianmo Ni, Dan Bikel, Enrique Alfonseca, Yuan Wang, Chen Qu, and Imed Zitouni. 2022. Exploring dual encoder architectures for question answering. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 9414–9419.

Yuwei Fang, Shuohang Wang, Zhe Gan, Siqi Sun, Jingjing Liu, and Chenguang Zhu. 2020. Accelerating real-time question answering via question generation. *arXiv preprint arXiv:2009.05167*.

Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE: sparse lexical and expansion model for first stage ranking. *CoRR*, abs/2107.05720.

Luyu Gao and Jamie Callan. 2021. Condenser: a pre-training architecture for dense retrieval. *arXiv preprint arXiv:2104.08253*.

Luyu Gao and Jamie Callan. 2022. Unsupervised corpus aware language model pre-training for dense passage retrieval. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2843–2853, Dublin, Ireland. Association for Computational Linguistics.

Luyu Gao, Zhuyun Dai, and Jamie Callan. 2020. Understanding BERT rankers under distillation. In *ICTIR '20: The 2020 ACM SIGIR International Conference on the Theory of Information Retrieval, Virtual Event, Norway, September 14-17, 2020*, pages 149–152. ACM.

Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2022. Tevatron: An efficient and flexible toolkit for dense retrieval. *CoRR*, abs/2203.05765.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-scale similarity search with gpus. *IEEE Trans. Big Data*, 7(3):535–547.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick S. H. Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 6769–6781. Association for Computational Linguistics.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur P. Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. 2019. Natural questions: a benchmark for question answering research. *Trans. Assoc. Comput. Linguistics*, 7:452–466.

Patrick Lewis, Yuxiang Wu, Linqing Liu, Pasquale Minervini, Heinrich Küttler, Aleksandra Piktus, Pontus Stenetorp, and Sebastian Riedel. 2021. Paq: 65 million probably-asked questions and what you can do with them. *Transactions of the Association for Computational Linguistics*, 9:1098–1115.

Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. 2016. MS MARCO: A human generated machine reading comprehension dataset. In *Proceedings of the Workshop on Cognitive Computation: Integrating neural and symbolic approaches 2016 co-located with the 30th Annual Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain, December 9, 2016*, volume 1773 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Minjoon Seo, Jinhyuk Lee, Tom Kwiatkowski, Ankur Parikh, Ali Farhadi, and Hannaneh Hajishirzi. 2019. Real-time open-domain question answering with dense-sparse phrase index. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4430–4441, Florence, Italy. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul N. Bennett, Junaid Ahmed, and Arnold Overwijk. 2021. Approximate nearest neighbor negative contrastive learning for dense text retrieval. In *9th International Conference on Learning*

Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net.

Wayne Xin Zhao, Jing Liu, Ruiyang Ren, and Ji-Rong Wen. 2022. Dense text retrieval based on pretrained language models: A survey. *arXiv preprint arXiv:2211.14876*.

A Tying the Passage and Query Encoders Embeddings

In this section we provide justification for our decision to tie the token embeddings of the passage and query encoders. This decision can be split into two; For the transformer based models, that are loaded from the first layers of some pretrained model, it is not very significant. Experiments show that tying the embeddings has very small effect on these models (e.g. for a 2-layer transformer $MRR@10$ results increase from 37.25 to 37.28). On the other hand, we do not have available pretrained models to initialize GRU-based models. Tying the embeddings allows us to transfer some of the knowledge acquired during the pretraining of the transformers to the GRU models. Further, if a dev query contains a token that does not appear at all in the train set, during testing on the dev set the token embedding will be totally random, and the model will not be able to correctly encode the query. Indeed, experiments show that for the GRU-models tying the embeddings is extremely important as without doing so they have a hard time to converge.

B Training Procedure

This work does not focus on the training procedure of the model. Thus, we chose to utilize the popular training procedure of (Gao and Callan, 2022). For completeness we provide the technical details of their procedure in this section.

We assume we have at hand a pre-trained model. The procedure starts by retrieving hard negative examples using a model denote by S_1 (described below). Then, our model is trained for three epochs and a batch size of 64 using a contrastive loss. We used the AdamW optimizer with a $5e-6$ learning rate and a linear learning rate schedule.

The model S_1 is trained using the same training procedure. It only differs in the set of negative samples used. Specifically, when training S_1 the negative samples are retrieved by BM25.

C Complete Latency Report

In Table 5 we give a full latency report. For each model we report latency in milliseconds of 50, 90, 95 and 99 percentiles. We report both online-latency (marked as pX) and full latency (marked as pXf). Online and full latency differ only For RNN-based models where online latency is considered as latency when applying incremental inference as described in Section 3. We measure latency on a c6i.2xlarge EC2 machine featuring Ice Lake processor with 8 hyperthreads. Each evaluation is repeated 1020 times, and we discard the first 20 to allow the model to warm up. We report the average of the remaining runs. We note that utilizing a GPU typically requires provisioning a separate machine with a GPU. Since network latency is above 5ms, this does not decrease the total inference cost, so we avoid measuring it here.

Encoder	Layers	FF Layers	Params	p50	p90	p95	p99	p50f	p90f	p95f	p99f
BERT	12	NA	110	21.05	23.69	26.81	25.98	21.05	23.69	26.81	25.98
GRU	1	0	27.4	0.43	0.58	0.70	0.84	1.2	1.52	1.66	2.04
GRU	1	1	32.1	0.87	0.99	1.07	1.23	1.66	1.99	2.23	2.53
GRU	2	0	30.9	0.66	0.90	1.10	1.43	2.30	2.85	3.24	4.09
GRU	2	1	35.6	1.19	1.49	1.65	1.97	2.84	3.60	3.89	4.67
Transformer	1	NA	31.5	1.6	1.91	2.10	2.29	1.6	1.91	2.10	2.29
Transformer	2	NA	38.6	3.53	4.03	4.44	4.70	3.53	4.03	4.44	4.70
Transformer	3	NA	45.7	5.69	6.23	6.99	7.83	5.69	6.23	6.99	7.83
Transformer	4	NA	52.8	7.48	8.23	9.31	9.33	7.48	8.23	9.31	9.33

Table 5: Full latency report.